DRAFT

D3.1 Programmability framework

Dissemination level	PU
Version	1.0
Due date	31.10.2014
Version date	14.11.2014

This project is co-funded
by the European Union

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Document information

Authors

iMinds – Wouter Tavernier, Sachin Sharma, Sahel Sahhaf

ETH – Róbert Szabó, Dávid Jocha

ACREO – Pontus Sköldström

EHU – Jon Matias, Jokin Garay

OTE – George Agapiou

BME – Balazs Sonkoly

TUB – Matthias Rost

BISDN – Tobias Jungel

EAB – Ahmad Rostami, Xuejun Cai

Coordinator

Dr. András Császár

Ericsson Magyarország Kommunikációs Rendszerek Kft. (ETH) AB

KONYVES KALMAN KORUT 11 B EP

1097 BUDAPEST

HUNGARY

Fax: +36 (1) 437-7467

Email: andras.csaszar@ericsson.com

Project funding

7th Framework Programme

FP7-ICT-2013-11

Collaborative project

Grant Agreement No. 619609

Legal Disclaimer

The information in this document is provided ‘as is’, and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law.

© 2013 - 2014 by UNIFY Consortium

DRAFT

Revision and history chart

Version	Date	Comment
0.1	28-05-2014	M31-M32 document finalized and serving as starting point for this deliverable.
0.2	16-09-2014	Update outline of the document including related work/gap analysis, decomposition and orchestration sections.
0.3	01-10-2014	Version incl. draft for NF-FG model formalization, decomposition and orchestration processes sections.
0.4	15-10-2014	Update of abstracted interfaces, revised sections in programmability framework and updated gap analysis.
0.5	20-10-2014	Document released for internal review.
0.6	27-10-2014	Re-alignment with WP2 architecture terminology
0.7	03-11-2014	Integration and addressing of review comments
1.0	14-11-2014	Final release of deliverable

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Table of contents

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

DRAFT

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

DRAFT

List of figures

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

List of tables

DRAFT

Summary

This deliverable documents the service programmability framework for the UNIFY architecture. This framework will detail relevant process flows, interfaces, information models and orchestration functionality in support of service programming in UNIFY. In order to make sure that the characterization of the programmability framework does not occur in isolation of existing work in the research, open source initiatives or standardization, an extensive related work overview and corresponding gap analysis has been made (Annex 2 and Section 5). In addition, this work has been performed in continuous (re-)alignment with the UNIFY architecture defined by WP2, the monitoring processes defined by WP4, and the universal node design defined by WP5. Annex 1 of this deliverable lists the WP3 objectives. These are referred to as OBJ-x in this summary in order clarify the relationship of the performed work to the goals of work package 3.

Network service programmability is related to many aspects problem spaces. A first dimension of programmability relates to the definition and decomposition of different components and traffic flows in order to compose a network service (referring to the concept of a Service Graph), and the mapping of these components to physical resources (Orchestration challenges). Another dimension is concerned about the programming and configuration of these components itself. More complexity is involved when services need to be programmed for tackling dynamic events, involving monitoring metrics and appropriate reactions such as scaling in or out. At last, all of these dimensions need to be aligned such that they can be triggered in an automated way initiated by clients (referring to SDN-control). In the proposed framework we progressively tackle these challenges in the following parts.

The first part of the programmability framework is about the *characterization of the interfaces, their requirements and the identification of re-usable technologies* corresponding to the defined reference points between different layers: 1) User and Service Layer, 2) Service-Resource Orchestration, 3) Resource-Orchestration-Controller Adaptation, 4) Control Function-Resource Orchestration and 5) Controller Adaptation-Infrastructure. The most significant gaps with respect to the requirements for these interfaces (Section 4) and existing work (Annex 2) are identified on the interfaces 2, 3 and 4. For this reason, programmability in UNIFY focuses on these interfaces. Two information models are crucial in this context: the Service Graph (Section 6.3) and the Network-Function Forwarding Graph (NF-FG, Section 6.4). The Service Graph refers to the service request made by the user to the Service Layer, while the role of the NF-FG is two-fold: i) it acts as the main information model to describe the service request in sufficient detail to enable resource orchestration, and ii) it enables resource Orchestrators to interact with each other in a recursive manner by delegating NF-FG requests (top-down) to the responsibility of other resource Orchestrators (e.g., to the local Orchestrator of a UN, but

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

also to the Orchestrator of other domains). The interface to the Universal Node has been investigated in more detail in WP5 and summarized in this document (Section 7).

A second component of the programmability framework is about the high-level *provisioning process flow* and the involved information models which are exchanged across different reference points (OBJ-3). The provisioning process is characterized around two flows (see Section 6.1): i) a top-down Service Invocation Flow, and ii) a bottom-up Service Confirmation Flow. The first is initiated by a service request by the user which initiates a cascade of interactions between components at different layers down to the physical infrastructure. The second flow reflects the way in which infrastructure resource information as well as instantiated Service- and Network Function information is propagated from the Infrastructure Layer via the Orchestration Layer towards the Service Layer.

Because of the important and particular role of the NF-FG in the UNIFY architecture, an initial formal information model is defined for the *NF-FG* within WP3 (OBJ-6). The core primitives of this model (Section 6.4) are endpoints, Network Functions, network elements and monitoring parameters. While the first two primitives are rather self-explanatory in this context, the introduction of network elements enable a range of abstraction possibilities enabling network abstraction with different degrees of transparency (e.g. Big Switch abstraction). We build further on the ETSI MANO VNF-FG characterization for the *Service Graph model*. The characterization of monitoring functionality as well as required reaction in response to events might be programmed within the NF-FG itself using constructs from the MEASURE language documented in Section 6.7.4.5.

The role of *service decomposition* is important to enable multi-stage service programming (Section 6.6). In many cases, a user is less concerned about particular implementations of desired service functionality. For example an Intrusion Detection Service (IDS) can be implemented in different ways using more or fewer Network Functions of different kinds. A service decomposition framework enables decomposition at the appropriate stages of the orchestration process. We consider white-box decompositions guided by exposed rules (e.g., an IDS might be decomposed using a Firewall and a Deep Packet Inspection component, a Firewall might be implemented by an Open vSwitch FW, etc.). These rules might be given by the Service Layer and stored in a Network Function Information Base (NF-IB). A second type of decomposition might be steered by particular control Network Functions. The latter enable dynamic decomposition according to application-specific logic (e.g., dynamic decomposition into multiple NFs based on an internal learning algorithm).

A crucial part in service programming is centred on the role of *orchestration functionality* (Section 6.7). The main goal of resource orchestration is to map the components of NF-FGs on infrastructure resources. This process is referred as (virtual network) embedding. Several existing approaches for optimizing this process and remaining challenges have identified and documented in this document (OBJ-1, OBJ-2 and OBJ-4). When combining

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

the infrastructure resources of both cloud and network providers, orchestration processes must scale to support tens of thousands of elements, with dynamic changes that in some cases put strict timing requirements on the embedding, scaling, and failure handling systems. In complement to the possibility of recursively stacking Orchestration Layers as enabled by the defined architecture, abstraction and decomposition mechanisms, in combination to the multi-domain considerations are described in the context of this document to reach this goal (OBJ-5). In order to support scalability at the service or Network Function level, an initial set of scale-in and –out mechanisms are documented. The latter is closely related to interaction with monitoring functionality at different layers in the architecture, as for example, detected performance degradation might trigger these scaling processes. A range of required functionalities in the context of these dynamic processes have been identified and listed.

Several concepts and processes of the proposed programmability framework can be brought together in the application of *concrete use cases* (Section 8). For this purpose, scaling in and out of an elastic router has been taken as example. In addition, a more advanced use case focusing on video content services has been investigated. These act as a starting point for initial integrated prototyping work and components based on the already available prototyping efforts.

Future work in WP3 service programming will focus on further formalizing developed information models and corresponding interface protocols, as well as fine-tuning the required components for dynamic orchestration.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

1 Introduction

The UNIFY project targets flexible service creation, provisioning, and programmability in heterogeneous network environments from home to enterprise networks, through aggregation and core networks to data centres. One of the crucial enablers to support this process is the definition of open interfaces (application programming interfaces - API's) between all possible layers of the control and data plane architecture and their interacting users. Open API's enable programmatic control of available functionality in a range of components.

Flexible service definition and creation start by formalizing the definition a service into the concept of a Service Graph (SG) and subsequently a Network-Function Forwarding Graph (NF-FG) as described in D2.1. These graphs represent the way in which customer end points are interconnected to desired Network Functionalities such as firewalling, load balancing, and other functionalities represented in the use cases documented in the above mentioned document. Service Graph representations form the input for the UNIFY control and orchestration framework which is responsible for mapping these service requirements to specific physical resources in the network. Open data plane interfaces enable the effective provisioning of these mappings in the physical devices.

Figure 1.1: Core parts of the programmability framework

The goal of this document is to design a coherent set of processes, mechanisms, interfaces and information models serving as a programmability framework for network services. The architectural basis for this framework is the result from WP2 which consists of a Service Layer, an Orchestration Layer and an Infrastructure Layer. Rather than explicitly including

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

a terminology section in this document, we refer to the Annex A of D2.2 which includes a full overview of terminology used in the UNIFY project.

Section 2 introduces abbreviations and definitions which are not yet introduced in D2.1 or D2.2. Next, in Section 3, a brief architecture view is given, recapitulating the reference points resulting from this layered architecture which are important with respect to service programming, as well as the relevant actors for service programming.

In Section 4, the programmability requirements are fine-tuned in relation to the reference points in order have a clear understanding on what is required from the framework. Section 5 identifies the gaps in the fulfilment of these requirements with respect to applicable existing technologies and protocols documented in Annex 2.

The core of the proposed framework is documented in Section 6. The latter contains subsections on the core programmability aspects:

- Programmability process flows with a focus on provisioning
- Overview of programmability Information Models (and flows)
- Specification of the Service Graph model
- Specification of the Network Function-Forwarding Graph
- Structure of the Network Function-Information Base
- Characterization of the service decomposition framework
- Detailing orchestration processes related to programmability
- Refinement of abstract interface definitions
- Overview of multi-domain considerations.

Section 7 zooms in on the interface with the Universal Node in relation to the work of WP5. Two use cases are selected: an Elastic Network Function and a Video Content Service in order to apply the proposed models and mechanisms. Finally, Section 8 will conclude the document with lessons learned and directions for future work.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

2 Abbreviations and definitions

2.1 Abbreviations

Abbreviation	Meaning
API	Application Programming Interface
BGP	Border Gateway Protocol
BFD	Bidirectional Forwarding Detection
BSS/OSS	Business Support System/Operations Support System
CLI	Command Line Interface
CNF	Compound Network Function
DPI	Deep Packet Inspection
ENF	Elemental Network Function
FIB	Forwarding Information Base
KPI	Key Performance Indicator
KQI	Key Quality Indicator
MPLS	Multiprotocol Label Switching
MIB	Management Information Base
NBI	NorthBound Interface
NF	Network Function
NF-FG	Network Function Forwarding Graph
NSC	Network Service Chaining
ODL	OpenDayLight
OP	Observation Point
OTT	Over The Top
OVS-DB	Open vSwitch Database Management Protocol
PBB	Provider Backbone Bridge
PBB-UCA	Provider Backbone Bridge - Use Customer Address
QoS	Quality of Service
RIB	Routing Information Base
SA	Service Availability
SG	Service Graph
SAP	Service Access Point
SBI	Southbound Interface
SG	Service Graph
SLA	Service Level Agreement
SLS	Service Level Specification
SW	SoftWare
TCAM	Ternary Content-Addressable Memory
UN	Universal Node
VM	Virtual Machine
VNF	Virtual Network Function

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

2.2 Definitions

For definitions not included in this document, we refer to Section 2.2 of D2.2. Here we focus on the additional concepts specific to the programmability framework.

Control Application or CtrlApp (also VNF CP or Control NF) is a Network Function which has the ability to interact directly with the resource orchestration (through the Cf-Or interface), enabling instantiated services to dynamically change the NF-FG request with respect to NFs, their interconnection or required resources, through a programmatic interface.

Service decomposition is the process of transforming a NF-FG containing abstract NF(s) to NF-FG(s) containing less abstract, more implementation-close NF(s). This can also include dividing the functionality of a complex NF to more, less complex NFs. In UNIFY, we have a generic concept of UNIFY(ed) service decomposition, and two realization options, the NF-IB-based (aka white-box) and the CtrlApp based (aka black-box) decomposition as described in Section 6.6.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

3 Architecture overview

The design of the UNIFY architecture is described in three incremental steps in deliverables of WP2. The first (i.e., overarching architecture) and the second (i.e., functional architecture) design steps are documented in Deliverable 2.1 (D2.1) and the third design step (i.e., system architecture) is currently in progress and will be documented in Deliverable 2.2 (D2.2).

The overarching architecture defines the high level design principles such as layers (i.e., Service Layer, Orchestration Layer, and Infrastructure Layer) and main interfaces between the layers (i.e., U-Sl, Sl-Or, Or-Ca, Ca-Co, Co-Rm, and Cf-Or) as described in deliverable D2.1. The functional architecture illustrates subspecialized elements of the layers and specifies the interaction between the elements within one layer or across different layers.

With regard to designing the architecture, WP4 contributes to the UNIFY framework by identifying the narrow-waist that meets following principles:

- The narrow-waist harmonizes and unifies all the operations performed below it
- The narrow-waist offers a generic resource provisioning service
- The narrow-waist component must work on abstract resources and capabilities types, virtual resources corresponding to network, compute and storage virtualization
- The narrow-waist component must not understand any higher layer logic, function, configuration, etc.

Figure 3.1 depicts a three layered model that the UNIFY framework follows. The narrow-waist is shown at the resource orchestration point in the figure. Note that the architecture represents a user plane that is shown separately from the service provider in the figure, thus it is not considered part of the three layered model.

The Service Layer is connected to the application layer through its northbound interface and communicating with users, e.g., end user, retail provider, OTT provider, content provider, and a service provider. The service request from the user turns into consumable services on this layer by defining and managing service logics and by establishing programmability interface to users. The service is described by a chain of high-level Network Functions and pre-defined parameters which is generally referred to as a Service Graph (a.k.a., Network Service Chaining) all through the UNIFY framework. The Service Layer also interacts with the Orchestration Layer via its southbound interface and provides further detailed description of the service chain as a form of Network Function Forwarding Graph (NF-FG).

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

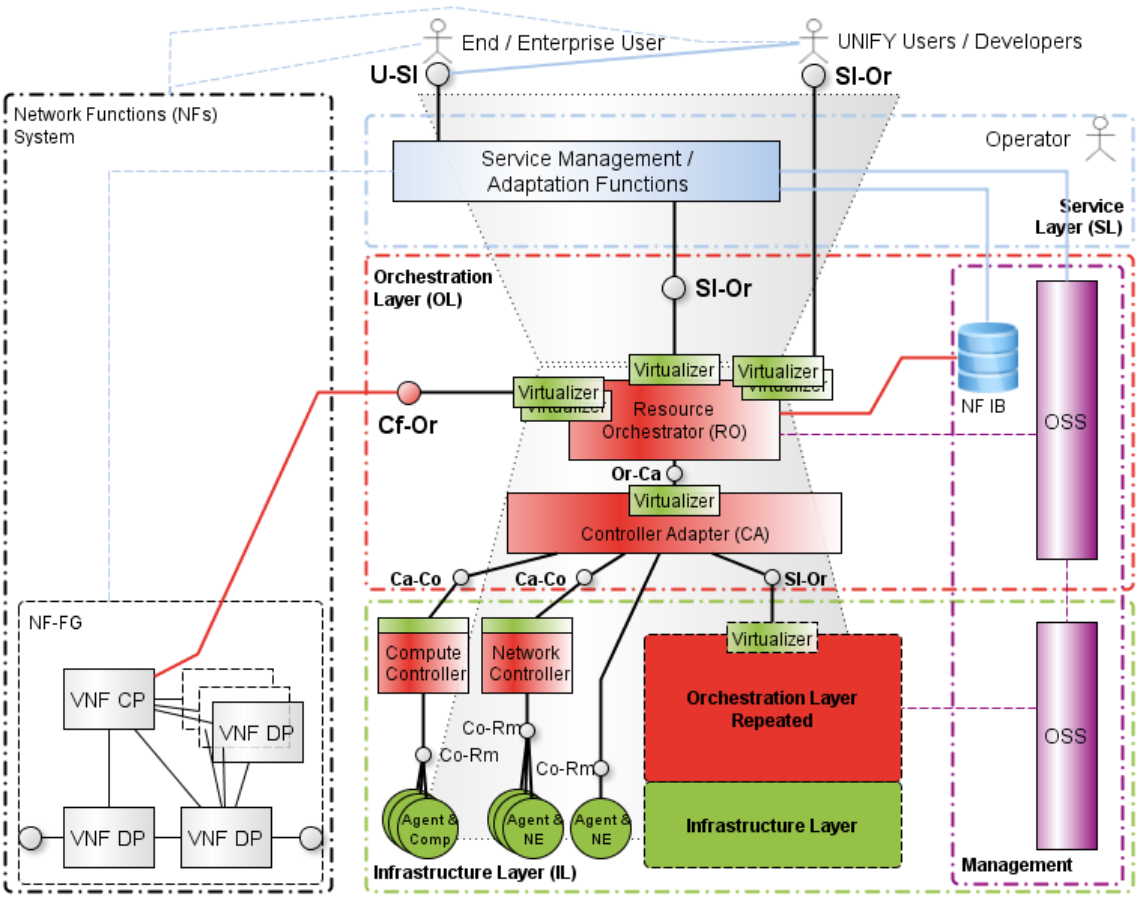

Figure 3.1: The three layered UNIFY architecture

The Orchestration Layer maintains a global view of the network and bridges between the Service Layer and the Infrastructure Layer, thus it is the core of the UNIFY system. The Orchestration Layer is designed to provide a unified representation of all underlying resources and capabilities. For this, the Orchestration Layer receives a logical chain of the service (in an NF-FG form) from the Service Layer via its northbound interface and maps physical/virtual resources into the logical service chain. The architecture also considers an eastbound interface (Cf-Or depicted in Figure 3.1) for receiving updates from the deployed Service itself interfacing with the Resource Orchestrator through a CtrlApp or Virtual Network Function Control Plane component. Based on this mapping, the Orchestration Layer reserves and configures resources and management functions (e.g., monitoring and troubleshooting) through its southbound interface towards the Infrastructure Layer. Moreover, the Orchestration Layer receives and analyses the status information of resources that is notified by the Infrastructure Layer and forwards it to the Service Layer.

Finally, the Infrastructure Layer encompasses all networking, compute and storage resources. By exploiting suitable virtualization technologies this layer supports the creation of virtual instances (networking, compute and storage) out of the physical resources. To put it concretely, Universal Nodes (see D5.2 for detail), Data Centres, SDN nodes (e.g., OpenFlow switches), and legacy appliances are primarily considered as physical resources.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Each of these physical resources has a different northbound interface (NBI) and capabilities (e.g., level of programmability). Therefore, the Orchestration Layer must be able to interact with each of this NBI exposed by these resources. For this reason, the Orchestration Layer is further divided into three sub-layers and the Control Adaptation (CA) and multiple controllers are mainly responsible for communicating with various types of physical resources.

3.1 Actors relevant to programmability

Considering the recent development hypes around Software Defined Networking [Chua2013], it is inevitable to consider and possibly build on already existing software components. Such component based design would also allow modular and independent development of functionalities if interfaces are cleverly defined.

If we take a look at the development landscape, we can identify different actors who contribute with different components to create a virtualization and orchestration framework up to the users. Below, we identify a few key actors and describe their relations to creating a value chain.

In the simplest case for any business relationships we have to identify users and service providers. Users consume communication and cloud services. Users can be residential or enterprise end users, other service providers (multi domain setup), over the top (OTT) providers, content providers, etc. Users sign a contract with the service provider for specific services with service level agreements (SLA). Service providers provision, operates and finally bill services to their users [TMF,ETOM]. In the SDN and Cloud era service providers would like to reduce both their operational and capacity expenses through virtualization.

Softwarization of the infrastructure involves creating global resource views and orchestrating those resources. Infrastructure vendors (e.g., of universal node, data centres, etc.) will continue to create the hardware elements providing optimized execution environment for virtualized Network Functions. Controller software managing both the data centre and the physical networking resources are developed mostly in open sources communities (e.g., OpenDaylight[1], ONOS[2]). Orchestration functionality, on the other hand, is an added value on the top of the generic controller functionality, hence will become the differential platform services offered to the service providers to run their networks.

[1] http://www.opendaylight.org/
[2] http://onlab.us/tools.html

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

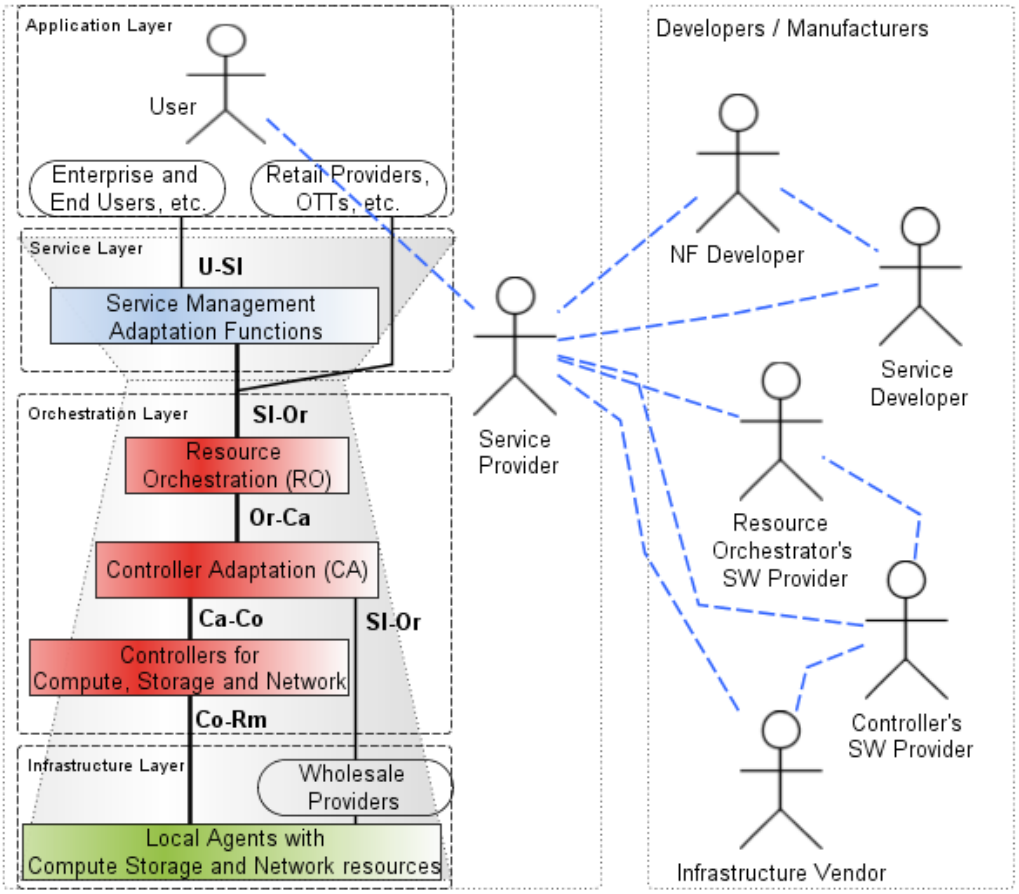

Figure 3.2: Business actors in Service Programming

- User: Users consume communication and cloud services. Users can be residential or enterprise end users, other service providers (multi domain setup), over the top (OTT) providers, content providers, etc. End/Enterprise Users (also referred as client) interface with the U-Sl interface, while retail/OTT providers directly consume the UNIFY Resource Service (see D2.2). Users sign contracts with the service provider for specific services with service level agreements (SLA).

- Service Provider: Service providers offer services to users subject to specific SLAs. Service providers make direct use of logical resource management (from Orchestration SW Providers) and DP & Virtualization Management (from Controller SW Providers). Service providers access the resources via a resource manager functionality of an Infrastructure provider.

- Orchestration SW Provider: Software developers (e.g., vendors, 3rd party) who create software functions (services, libraries and apps) to manage the global view of abstract resources.

- Controller SW provider: Software developers (e.g., open source communities, vendors or 3rd parties) developing data plane managers (e.g., OpenFlow) and cloud

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

managers (e.g., OpenStack) to present abstraction of the underlying resources (networking and cloud).

- Infrastructure Vendor: Providers of physical resources including both networking and virtualization environments.

- NF Developers: internal to the service provider or third party developers who designs, develops and/or maintains Network Functions. The orchestration framework shall support the development cycle through service provider DevOps (see WP4).

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

4 Programmability requirements

The programmability requirements are largely driven by the interfaces corresponding to the defined reference points (see previous section). An initial definition of these abstract interfaces has been defined in D2.1 Section 7.2. The resulting functionality is depicted in Figure 4.1. Section 4 of the same document reports a full list of requirements relative to UNIFY. Subsection 4.2 focuses on the programmability and orchestration aspects and corresponding requirements in general (in direct relationship to the ETSI NFV requirements). Meanwhile the functional architecture has reached a mature state (documented in D2.2) which requires minor reconsideration from programmability aspects as well. A refined and more detailed version of the top-level functional model supporting recursive orchestration is shown in Figure 4.2.

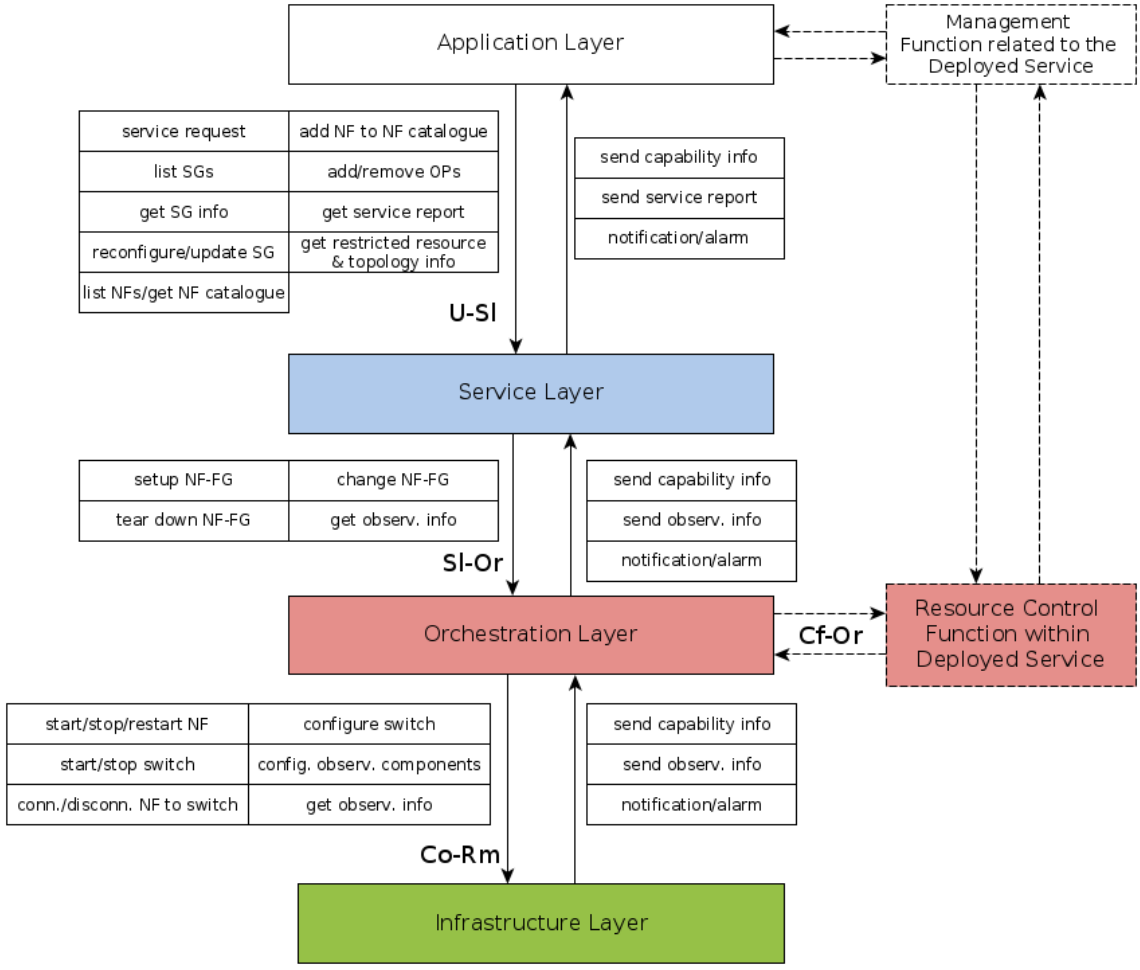

Figure 4.1: Initial interface description driving programmability requirements

A more detailed version of the top-level functional model supporting recursive orchestration is shown in Figure 4.2.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

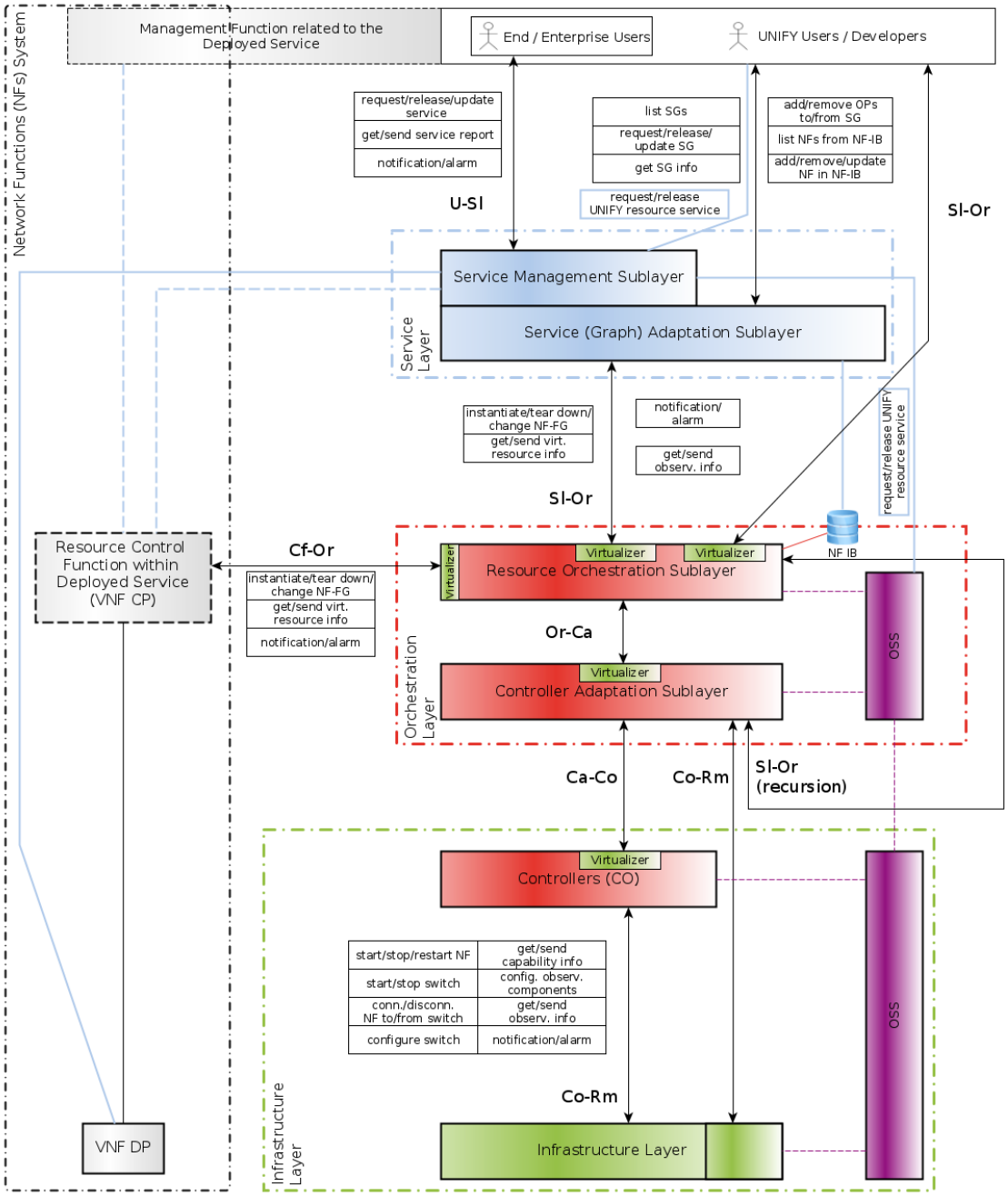

Figure 4.2: A more detailed view on the top level functional blocks and interfaces

4.1 U-Sl/Sl-Or interface

A service request (involving a Service Graph) from the Application Layer towards the Service Layer, has the following programmability requirements:

1. MUST include which SAPs are involved, and which NFs (both virtual and physical NFs MUST be supported) are required in the service (given that these NFs are listed in the NF catalogue)

2. MUST include a specification of connectivity types and connectivity levels in between NFs and/or SAPs. This SHOULD support flow space definitions.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

3. SHOULD be able to provide SLA parameters on traffic requirements and its scope

4. SHOULD support the attachment of performance indicators or Key Quality Indicators[3] to NFs, the connectivity between NFs or on combinations of both

5. SHOULD support constraining the mapping of service components to the physical infrastructure (including pinning down NFs to particular resources)

6. SHOULD be able to specify resiliency required of NFs, connectivity between NFs or combinations of both

7. MAY support the characterization of optimization triggers related to the mapping of service components to the physical infrastructure (e.g., related to traffic characteristics)

8. SHOULD be able to specify scaling requirements of service components

9. SHOULD specify restrictions on what traffic is allowed in the Service Graph

10. SHOULD be able to specify service-specific policies defined by users

11. MAY specify how billing should be performed

The reconfiguration of a service MUST support the addition or removal of NFs, links or SAPs, and the modification of any of the characteristics mentioned in the above requirements.

4.2 Sl-Or interface

The Sl-Or interface can be considered as a an enriched U-Sl interface, where the SG is enriched towards a Network Function-Forwarding Graph. The requirements listed for the SG, also apply on the NF-FG description. For NFs part of the NF-FG, the following requirements apply:

1. The NF description MUST include resource requirements in terms of computation, storage and memory requirements in order to enable mapping to infrastructure

2. Key Performance Indicators (KPI) related to ENFs or interconnected groups of NFs MUST be measurable

4.3 Cf-Or interface

The following functionality is required from a Resource Control Function within a Deployed Service and the resource Orchestration Layer. These are similar to the ones on the Sl-Or interface:

1. When programming the VNF as a component of the Service Graph its description MUST be able to contain compute and store resource demands.

2. SHOULD be able to create and upgrade or remove NF images in an operational environment

[3] This may involve requirements related to resiliency, QoS, etc.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

3. SHOULD be able to modify (add/remove) links between NFs

4. SHOULD be able to change the NF description-related requirements:

5. SHOULD be able to modify the link requirements

- SHOULD enable scaling of NFs (e.g. resize NF resources)

4.4 Or-Ca interface

Requirements on the Resource Orchestration - Controller Adaptation (Or-Co) interface:

1. MUST support the Sl-Or interface requirements, as described there.

2. MUST support the resource mapped NF-FG description

3. MUST not be specific to any controller

4. SHOULD support the merged NF-FG view (because it will be scoped to domains/controllers by the CA)

5. In case of the Orchestrator and the Controller Adaptation are not separated, this interface MAY not exist or MAY be internal/proprietary in the given implementation

4.5 Ca-Co interface

Requirements on the Controller Adaptation – Controller (Ca-Co) interface:

1. MUST support a subset of the north bound interface (NBI) of the controller

2. SHOULD support at least the minimal subset needed to initiate a NF and interconnect the initiated NF with the domain boundary (if applicable)

3. MUST NOT contain information which is not related to the domain/controller scope (except reference to domain edges to other domains)

4. SHOULD be specific to the given Controller, i.e.

5. In case of networking, it MUST be able to describe the connectivity between the NFs

6. In case of computations, it MUST be able to manage NFs (including initiating, configuring, …)

7. MAY be skipped, in case of a domain which is able to directly receive NF-FGs.

4.6 Co-Rm interface

In addition, the following base functionality is expected to be initiated by Controller(s) in the Orchestration Layer towards the Infrastructure Layer:

1. MUST support at least one north-bound interface of network switching equipment in order to start/stop, configure, model and discover switching functionality

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

2. MUST support at least one north-bound interface of a server platform in order to start/stop, configure, model and discover NF and server functionality

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

5 Programmability gap analysis

Legend: X = intended applicability [x] = potential applicability, although not intentional	U-Sl/Sl-Or	Cf-Or	Or-Ca	Ca-Co	Co-Rm
Multi-scope configuration and modelling frameworks					
SNMP					x
NETCONF/YANG	[x]	[x]			x
(Web) Interface Description Languages	[x]	[x]			
Semantic (Web) Modelling frameworks	[x]	[x]			
Infrastructure modelling frameworks					
Common Information Model	[x]				x
Directory-Enabled Networking(-NGRG)					x
RSpec					x
Network Description Language					x
Network Markup Language					x
Infrastructure and Networking Description Language					x
Network Programming and Control					
Node-level programming and Control					
OpenFlow					x
OVSDB					x
OF-Config					x
Click Modular Router	x				
HILTI	x				
ForCES					x
Network-level Programming and Control					
SDN Controller (incl. ODL)			[x]	X	
Network Programming Language Overview	[x]	[x]	[x]	x	
Frenetic/Pyretic	[x]			x	
Akamai Query System	[x]				
Simple Management API	x		x		
NM-WG schema	x		x	x	
I2RS			x		
ABNO	x		x		
Cloud Programming and Control					
Cloud-level Programming and Control					
OpenStack				x	
Cloud Controller Overview			x	x	
Service-level Programming and Control					
ScaleDL	x				
ETSI MANO VNF Graph model	x				

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

In this section, we analyze the technologies reported in Annex 2 and investigate how they can be applied in order to meet UNIFY requirements.

Orchestration and control functionality in the UNIFY architecture might be accessed as a (northbound) web interface. As **REST** provides good performance and scalability, it is the RPC paradigm to be used in most of the cases (Web services). Its simplicity and capability of using different data formats compared to SOAP protocol (using only XML) make it a potential interface paradigm to any orchestration and/or control software layer in the UNIFY architecture, thus of potential value for Sl-Or, Cf-Or and Or-Ca (the latter in both directions).

5.1 U-Sl interface

The closest match of the Service Graph information model and corresponding interface as defined in the UNIFY architecture in D2.1 is the **Network Service (NS) model defined by ETSI MANO**. The latter already defines the concepts of NFs, their links and the resulting graph. While this is ongoing work, UNIFY might base the SG model on ETSI, and extend it in order to support the notion of NF and service scalability.

For service scalability, UNIFY might rely on the work performed by the CloudScale project. **ScaleDL** is a language defined by the project particularly expressing scaling properties of NFs and the service.

Wherease the above proposals mainly focus on the syntactic/interface properties of services, it might be useful to consider the additional value of **adding semantics** to the description of NFs and services. The latter might inherit from the work done in research on ontologies and the semantic web (services). While traditional web services have a different goal compared to the services UNIFY intends to deliver[4], there might be several characteristics which might be re-used. Frameworks such as **BPEL** (its extensions) and **OWL-S** enable the definition of composite web services. These syntactic and semantic frameworks have interesting properties in order to characterize composite UNIFY service in the form of Service Graphs. Capability of QoS parameters specification and fault handling are other features of BPEL which are useful for service description in UNIFY.

In order to consolidate information of lower layers towards the user, UNIFY might rely on SMI. The **Simple Management Interface** (SMI) provides a simple and common management interface for multiple services deployed in cloud or other platforms. SMI can be used both with SOAP and REST interface. An operation “Get ManagementReport” is defined to return information about service instance health, failure and metrics. It could be used to query monitoring metrics or subscribe the metrics report and alarm. However, it doesn’t provide capabilities to describe monitoring functions or metrics to be associated with the Network Functions in service/network graph. It may be applied into U-Sl and SI-Or interfaces but must be extended and adapted in order to be used in UNIFY.

[4] UNIFY intends to offer services which provide functionality at lower layers than at the http-layer, involving for example raw packet processing elements.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The specification of a Service Graph not only describes the interconnection of NFs, but also assumes that the NFs themselves can be characterized in an accurate manner. NFs might be programmed using barebone system calls on top of, for example a *NIX-based OS, but ideally higher level libraries or frameworks are available. In addition to the frameworks which are investigated in WP5 (e.g., DPDK framework[5]), we identified Click Modular Router and HILTI as potential frameworks of value in this space. **Click modular router** is a potential candidate for implementing Network Functions (NF) indicated in the NF-FG. The modular structure in Click enables implementation of atomic Network Functions and more advanced Network Functions can be defined as Click scripts (combination of Click elements/atomic Network Functions). The **HILTI toolkit** might be used to compose NFs which focus on traffic analysis and inspection.

5.2 Sl-Or interface

Several of the technologies discussed in the previous section (a.o., **ETSI MANO VNNF** model), might be re-used and extended for the Sl-Or interface. In this context, the NF-FG model should be able to characterize the interconnection of NFs in a closer relation to the available infrastructure and to the end points via fixed and logical links respectively, supporting recursively splitting the graph into multiple domains (see D2.1 Section 6.3).

The **NM_WG XML schemas** introduced by OGF (Open Grid Forum) define a neutral representation for network measurements and can be extended to support new types of data. It could be a candidate format used to describe the monitoring functions and the measurement metrics. However, it must be extended to support the concept of NF-FG defined in UNIFY and provide more generic abstract for various monitoring functions. In addition, as no all interfaces will use XML based format, the conversion with other format is to be considered.

5.3 Cf-Or interface

The Cf-Or interface has many similarities to the Sl-Or interface, but has a more restricted scope. The technologies discussed in the above section(s) might therefore be re-considered and potentially constrained.

5.4 Or-Ca interface

Network Programming languages are not directly considered in UNIFY. However, we can benefit from them in defining service programming approaches. That is, some of languages can be extended and applied in specification of the UNIFY architecture interfaces. The advantage of many of these languages is that they offer high-abstraction level primitives for controlling networks. These concepts might be re-used for the abstract interface between the resource orchestration component and the controller adaptation component. Specifically, the following four languages are relevant for the Or-Ca interface.

[5] Intel Data Plane Development Kit: http://dpdk.org/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Frenetic/Pyretic provides high level abstractions to query and perform other network management tasks. It also lacks the capabilities to describe monitoring functions or metrics to be monitored.

NetCore operates with network policies described at a high abstraction level. This approach could be useful for describing traffic steering at a higher abstraction level. However, the major drawback of NetCore is that it is not network-wide language and the user must specify which network element implements given policies.

NetKAT uses regular expressions on network policies to describe the network behaviour. These regular expressions could be extended to involve NFs as well. Regular expressions could be a natural way to describe service chains or Service Graphs, therefore NetKAT or some components of that might be useful.

Merlin is able to automatically partitioning network policies expressed by a declarative language, and allocating resources. It could be applied for service chain/Service Graph description. Furthermore, we can borrow ideas for decomposition and resource mapping tasks as well. However, here is that the source code of Merlin is not yet available yet.

5.5 Ca-Co interface

Although the UNIFY framework intends to be compatible be with (potential extension to) any controller framework, the following two frameworks are of particular interest because of their very active development community and wide support of the industry.

OpenDaylight : The supported northbound interfaces to OpenDaylight include OSGi framework and bidirectional REST. In particular, the REST interface enables remote applications or higher layer controllers (e.g., Orchestrator) to describe the required transport between the NFs. Accordingly, the REST interface of the OpenDaylight can be the basis for designing a UNIFY-specific interface between the controller adaptation layer and the controllers (i.e., Ca-Co interface).

OpenStack : OpenStack's NBI is the management and control interface for OpenStack based cloud infrastructure. It is RESTful and based on JSON/HTTP. Each core project in OpenStack will expose one or more HTTP/RESTful interface for interacting with higher layer. OpenStack NBI claims to have good extensibility and discovery mechanisms. Therefore the interface may be used to manage NF VMs in the datacentre domain, and may be applied to other domain with extensions.

Due to the recursive nature of the UNIFY architecture, the **NF-FG model** might also be used on this interface to interact with lower layer Orchestrators. Because of the bi-directional nature of this interface, infrastructure resource might also be exposed from lower layers to higher layers using this model.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

5.6 Co-Rm interface

WP3 in UNIFY does not focus on the Co-Rm interface, as many protocols already exist in this space. Nevertheless, below a short overview is given on technologies which might be re-used.

SNMP may be used for specifying the Co-Rm interface, however in this case all the relevant MIB for the managed entities should be defined. This seems to be inflexible compared to e.g. NETCONF.

DEN may be used to model the knowledge about the network users, applications, network elements and their interactions. Using the information model in **DEN-ng**, policies can also be handled. This model is mainly used for management of devices and can be used to send capabilities of the devices over Co-Rm interface to the Orchestration Layer. However, it should be extended to support network virtualization technologies to be considered in this interface.

OpenFlow is obviously a crucial protocol (cfr. programmability requirements) that can be used in the communication between SDN Controllers and network resources (i.e., forwarding element). Specifically, the SDN controllers can utilize OpenFlow to program the forwarding elements in a per flow basis. The OpenFlow protocol will play a central role in realizing the Co-Rm interface of the UNIFY architecture, as it will enable dynamic traffic steering between (virtual) Network Functions, and therefore allows the complete realization of NF-FG.

NETCONF/Yang can be potentially used in Co-Rm interface to define NF-related operations and abstract data structures viewed by the Controller layer (or higher sub-layers of Orchestrator?). Procedures, such as starting/stopping NFs, requesting parameters of running NFs, notifications in case of failures or any other events can be defined by Yang language and implemented via NETCONF transport. Additionally, abstract data structures exposed toward upper layers can be given by Yang data models.

The general models such as **NDL** and **NML** focus mainly at generic network descriptions which can be extended or incorporated in other models. The later models such as **NDL-OWL** and **INDL** rely on these general models and also request-like models (e.g. VxDL) to enable i) users to define their requests easily and ii) management software to match the requests to available infrastructure. The semantic web nature of the general models enables them to be easily embedded in other models. Using OWL a graph structure can be generated which matches the infrastructures (a graph of connected resources). The other advantage is that OWL provides a clear split between semantic and syntax and this enables mixing/stacking several ontologies. Therefore, NDL-OWL and INDL may be of interest for the Co-Rm interface because unlike NDL and NML which are network-centric, they can model all network, compute and storage infrastructures and users requests can be modelled as well.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

ForCES provides an extendible framework and protocol for (dynamic) composition of various processing pipelines in the data plane. Specifically, ForCES provide interfaces and methods for control and management of logical functional blocks (LFB) in the forwarding plane, where the concept of LFB can be extended to Network Functions as considered in the project. One of the main advantages of the FOrCES is that it is oblivious to the type of processing (LFBs), i.e., not caring if the data plane processing is virtual or physical. Accordingly, ForCES can potentially be used for instantiation, configuration and life-cycle management of various (virtual) Network Functions, as well as dynamically interconnecting them to provide complex Network Functions within the Infrastructure Layer.

The most relevant **OVSDB** functionalities for UNIFY could be:

- The Network Configuration Service: The current default OVSDB Schema's support the Layer2 Bridge Domain services as defined in the Networkconfig.bridgedomain component.

- Overlay Tunnel Management: Network Virtualization using OVS is achieved through Overlay Tunnels. The actual Type of the Tunnel (GRE, VXLAN, STT) is of a different topic. The differences between these Tunnel Types are mostly on the Encapsulation and differences in the configuration. But can be treated uniformly for the sake of this document. While Establishing a Tunnel using configuration service is a simple task of sending OVSDB messages towards the ovsdb-server, the scaling issues that would arise on the state management at the data-plane (using OpenFlow) can get challenging. Also, this module can assist in various optimizations in the presence of Gateways, and also helps in providing Service guarantees for the VMs using these Overlays with the help of underlay orchestration.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

6 Programmability framework

Rigid network control limits the flexibility of service creation. Network and service virtualization aims to enable rich and flexible services and operational efficiency. Virtualization is controlled through Orchestrators (data centre and network), which offer northbound interfaces (NBI) to various users. The possibility for innovation highly depends on the capabilities and openness of these northbound interfaces. We believe that these interfaces should introduce high level programmability besides policy and service descriptions.

It is the vision of UNIFY that service function chaining will be used by the network operators to offer services to their customers (residential, enterprise, content providers, other operators, etc.). Both, operators and customers will like increased flexibility and dynamism in their control. This may be achieved through allowing them to program (directly or indirectly) the service chains.

ETSI in [ET2013a] – among other things - defined their Network Orchestrator as interfaces to the outside world to allow interaction with the orchestration software. Even though there may not be consensus in the splitting of functionality between orchestration and controllers, we re-define these terms as we use them throughout this document.

Our goal with the introduction of UNIFY's programmability framework is to enable on-demand processing anywhere in the physically distributed network and clouds. Our objective is to create a programmability framework for dynamic and fine granular service (re-)provisioning, which can hide significant part of the resource management complexity from service providers and users, hence allowing them to focus on service and application innovation similarly to other successful models like the IP narrow waist, Android or Apple IOS. A programmability framework consists of the definition of processes, mechanisms, interfaces and information models in order to support highly dynamic and flexible service provisioning.

Before delving into the details of the framework, a short overview of the global mapping process is given below. While most important concepts will be described in this context, a more complete overview of recurring terminology in UNIFY can be found in Section 2 of D2.2.

DRAFT

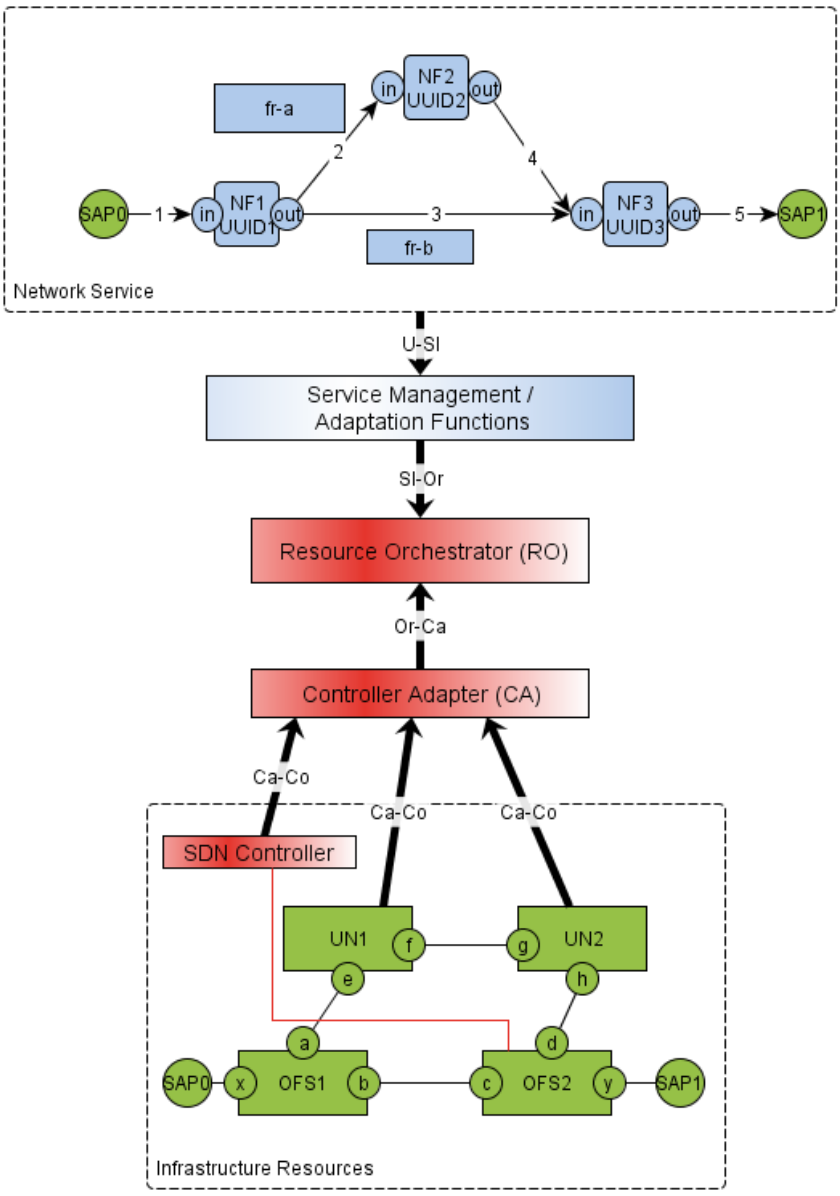

Figure 6.1: Orchestration as mediator between Service Graph requests and Infrastructure Resource availability

Flexible service provisioning needs to reconcile two sides of a spectrum: on one side there is the service definition, on the other side there is a heterogeneous landscape of infrastructure on which services need to be deployed. The first reflects the Service Layer, the latter is part of the Infrastructure Layer. In between, it is the goal of the Orchestration Layer to bring both together (see Figure 6.1). The Orchestration Layer receives the service information on its north-bound information from the Service Layer, and receives infrastructure resource models from network and cloud controllers on its southbound interface.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Service provisioning starts with the user[6] defining a service request in the form of a Service Graph (SG). An SG describes a service requested by a user and defines how (which Network Functions) and where the service is provided (which Service Access Points); and how successful delivery of the service is measured. Figure 6.1 depicts a simple SG consisting of three NFs.

In order to enable mapping of the individual components of the SG to the infrastructure, the Service Layer performs translation of NF descriptions into palatable resource requirements, as well as translating NF interconnections into concrete forwarding abstractions which can be mapped to network abstractions such as Big Switch with Big Software (BiS-BiS) connectivity between NFs (see Figure 6.2). The BiS-BiS abstraction is defined in D2.2, and refers to the virtualization of a Forwarding Element with a Compute Node, enabling to instantiate and interconnect NFs.

The result of this adaptation is the Network Function-Forwarding Graph (NF-FG), and is forwarded to the Orchestration Layer. Based on the resource model obtained via controllers interfacing with infrastructure, the resource orchestrator decomposes and maps NFs to server infrastructure, and network forwarding abstractions to infrastructure switching functionality. The mapping is the UNIFY Resource service provided by the Orchestration Layer. In the particular example of Figure 6.2, the VNFs of the NF-FG on the left upper side are deployed on two separate Universal Nodes (UNs), and the Big Switch abstraction interconnecting them is decomposed into the combined switching functionality of two OpenFlow switches and the virtual switching capabilities of UN1 and UN2. The output of the orchestration is the mapping/embedding of instantiable Network Functions to physical or virtual resources defined as a Network Function-Forwarding Graph.

DRAFT

[6] End-user, business user, retail provider, OTT Service Provider

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

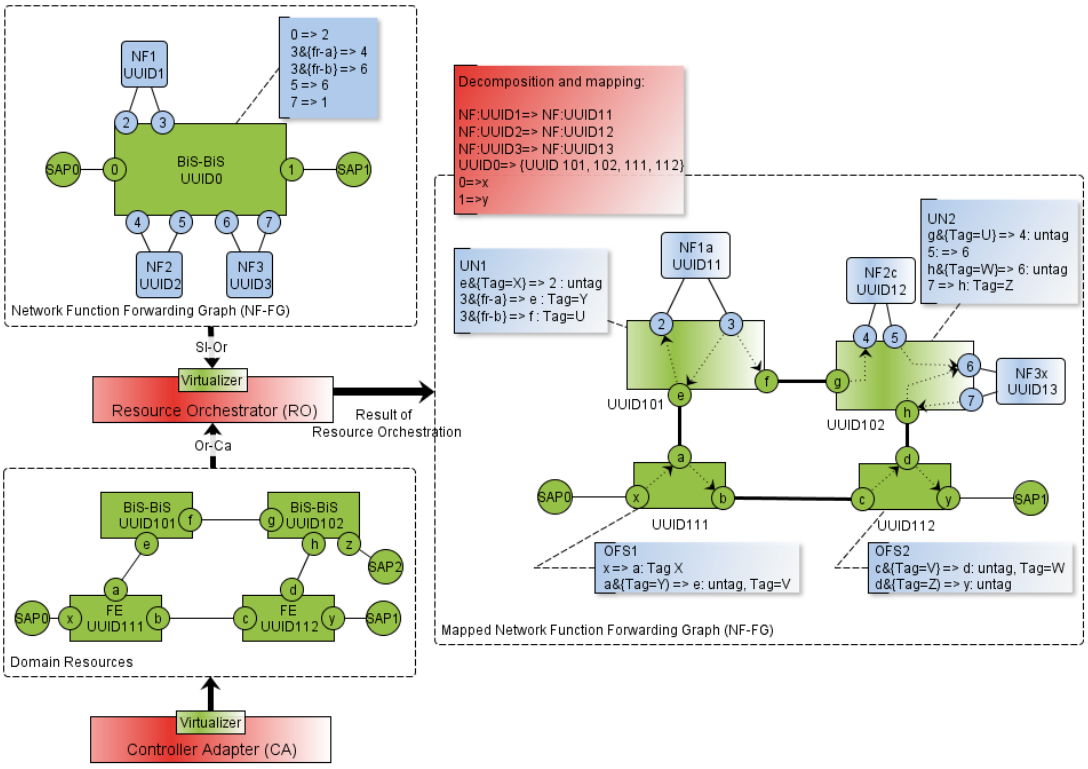

Figure 6.2: Mapping the Network Function-Forwarding Graph to infrastructure

6.1 Programmability process flows

Note, that only the programmability process flow is described, other functionalities like authentication, authorization, access control, charging, etc. will be defined later on. Additionally, monitoring and management aspects are defined in the DevOps framework and will be integrated into the overarching architecture. The programmability framework is used to (re-)provision services. The (re-)provision triggers may come from the user, the service management system, the resource management system or control plane. Handlings of these triggers are considered for further studies.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

6.1.1 Service Invocation: top-down

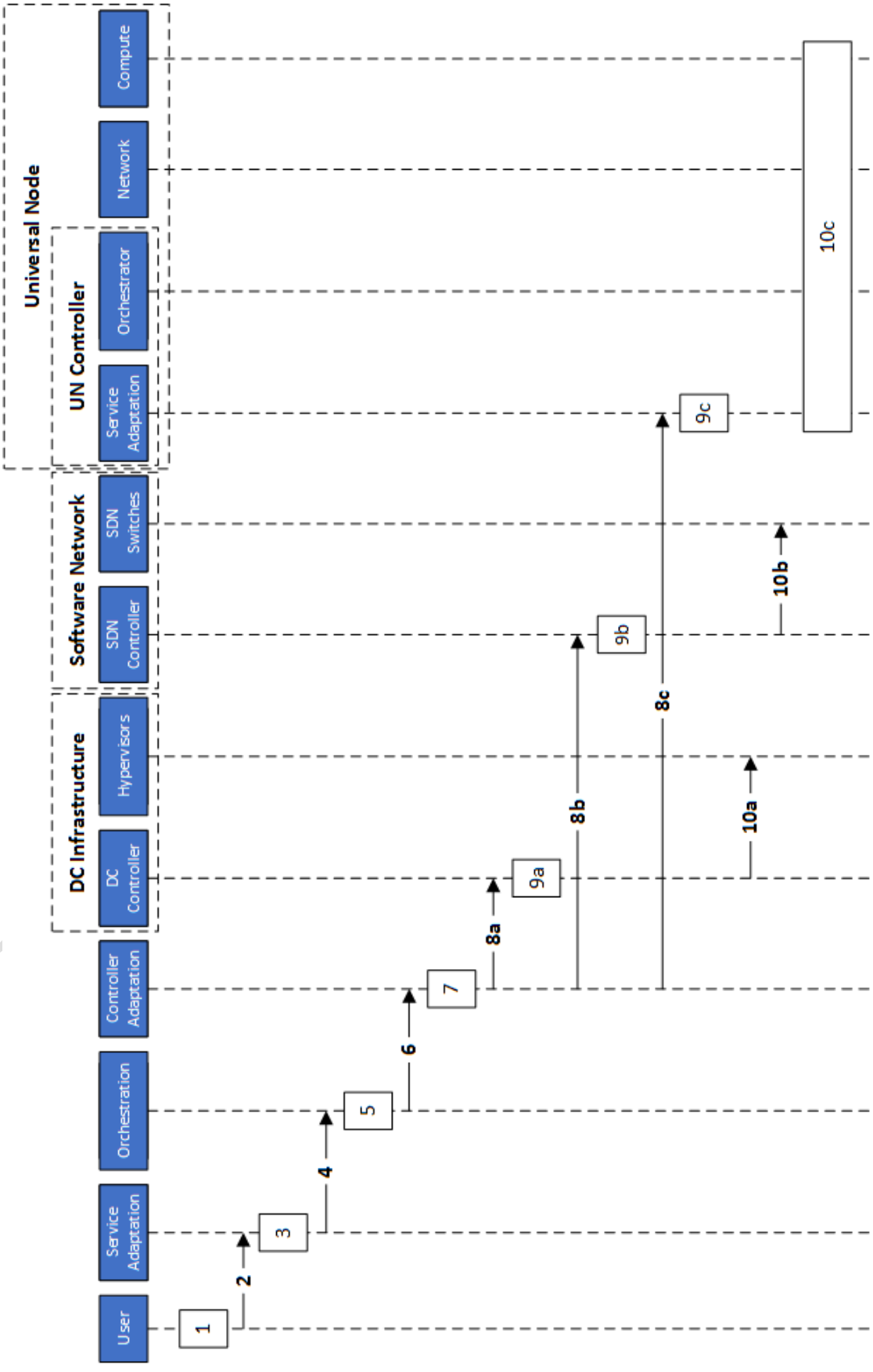

Figure 6.3: Sequence diagram: Service Graph resolution

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The service initiation process flow consists of the following steps, see Figure 6.3:

1. The User creates its Service Graph based on the available service components (or catalogue) or service templates or simply picks one of the service (graph) offered by the Service Provider. The Service Graph includes service functions (atomic or compound) as components, their logical connectivity and corresponding service level specifications (SLS) as part of the service level agreement (SLA).

Although Section 6.3 will describe in more detail the particular characteristics of a Service Graph, we can consider the following example of a parental control service as working assumption. It consists of 2 Service Access Points and three Network Functions: a firewall, a web-cache and a NAT function. The NFs are interconnected via three links, splitting web-traffic from other traffic between the firewall and the other NFs. As indicated on the figure, every NF in a SG has a unique identifier (UUID), enabling to refer to a NF instance. The latter can be shared between different SGs.

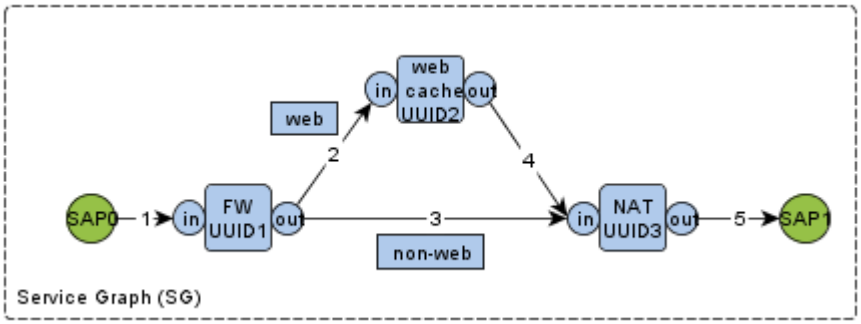

Figure 6.4: Service Graph example of a parental control service

2. The service request is sent to the service adaptation as a Service Graph according to the U-Sl reference point.

3. Upon receiving the Service Graph the service adaptation logic – besides traditional management functions like AAA, charging, etc. – may expand the details of the Service Graph definition using decomposition rules (see Section 6.5 and Section 6.6) and may translate any service level specifications requirements (e.g., by defining key quality indicators (KQI)) to compute, storage and networking requirements and measurable indicators (e.g., key performance indicators – KPIs). The relation of KQI's and required monitoring and observation points is described into more detail in 6.7.4. In addition, service adaption functionality might involve mapping (Service Layer-orchestration) to virtualized resources as exposed by the underlying Orchestration Layer (i.e., by the virtualizer component of the underlying layer). The mapping can be as simple as mapping the links of the SG to the ports of a virtualized Big Switch infrastructure component, but can also become more complex in case exposed virtual infrastructure consists of multiple components (see Section 6.2.1).

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

4. The NF-FG is sent to the Orchestrator according to the Sl-Or reference point (see Section 2). All components of the NF-FG are known in the SP's NF-IB.

5. The orchestration component bears with the global compute, storage and networking resource view at the corresponding abstraction level (see also Section 6.3 of D2.1). As detailed in Section 6.6, the Orchestrator can further decompose NFs according to available rules and resources in the catalogue, and/or delegate orchestration to lower-level domains/Orchestrators. The (lowest-level) orchestration function breaks down the Network Functions defined in the NF-FG until they are instantiable according to the given service constraints (e.g., proximity, delay, bandwidth, etc.), available resources and capabilities and operational policies (e.g., target utilization). The output of the orchestration is the mapping/embedding of instantiable[7] Network Functions to physical or virtual[8] resources in the form of a resource-mapped Network Function-Forwarding Graph.

6. The mapped Network Function Forwarding Graph (with outstanding compute, storage and networking requirements) is sent to the Controller Adaptation according to Or-Ca reference point.

7. Upon receiving a NF-FG, the Controller Adaptation: i) can split the NF-FG into sub NF forwarding graphs according to the capabilities of the different underlying controllers and ii) translates the information according to the Controllers' northbound interfaces. The information format below the Controller Adaptation depends on the type of resource.

8. Controller Adaptation sends scoped requests to the underlying controllers according to their resource types:

a) For compute/storage instantiation in data centres some compute Orchestrator must be invoked, e.g., OpenStack to instantiate VMs at a data centre or compute node (see 8a in Figure 6.3).

b) For the forwarding overlay allocation in the network an SDN controller must be contacted (e.g., OpenDaylight).

c) For compute, storage and networking resources in the Universal Node, the UN's Controller must be contacted. Within the UN, we foresee a similar stack of orchestration functions as in the overarching UNIFY domain, i.e., adaptation functions, orchestration and compute and networking resource managers. Therefore we foresee that the UN can receive definitions and requirements according to a NF-FG, which is a sub-graph of the output of the upper level orchestration.

[7] Note: Instantiable has scoped meaning, i.e., one Orchestration Layer may believe that a NF is directly instantiable at some of its resources; however, there may be additional abstraction/virtualization layer(s) involved underneath.
[8] Provided by the resource service provided by the underlying layer.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

9. Different Controllers act as virtualization managers according to the underlying technologies/virtualizations:

a) The Compute Infrastructure Manager receives the requested Network Function Virtual Instances and CPU, storage constraints per node. Depending on the type of the resource where the function is to be launched, it will bootstrap an appropriate virtual machine or reserve resources on an appliance.

b) The Network Controller will receive the desired network connectivity between the Network Function instances. Based on the type of requested connectivity, the capabilities of the network equipment and the actual network state, it will decide on the realization.

c) The Universal Node will receive a NF-FG, and will do the internal resource orchestration and allocation similar to point 4-9.

10. The Infrastructure components will receive the requests from their associated Controllers/Managers via the applicable protocols (e.g., OpenFlow, libvirt) and will start providing the requested functionality.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

6.1.2 Service Confirmation: bottom-up

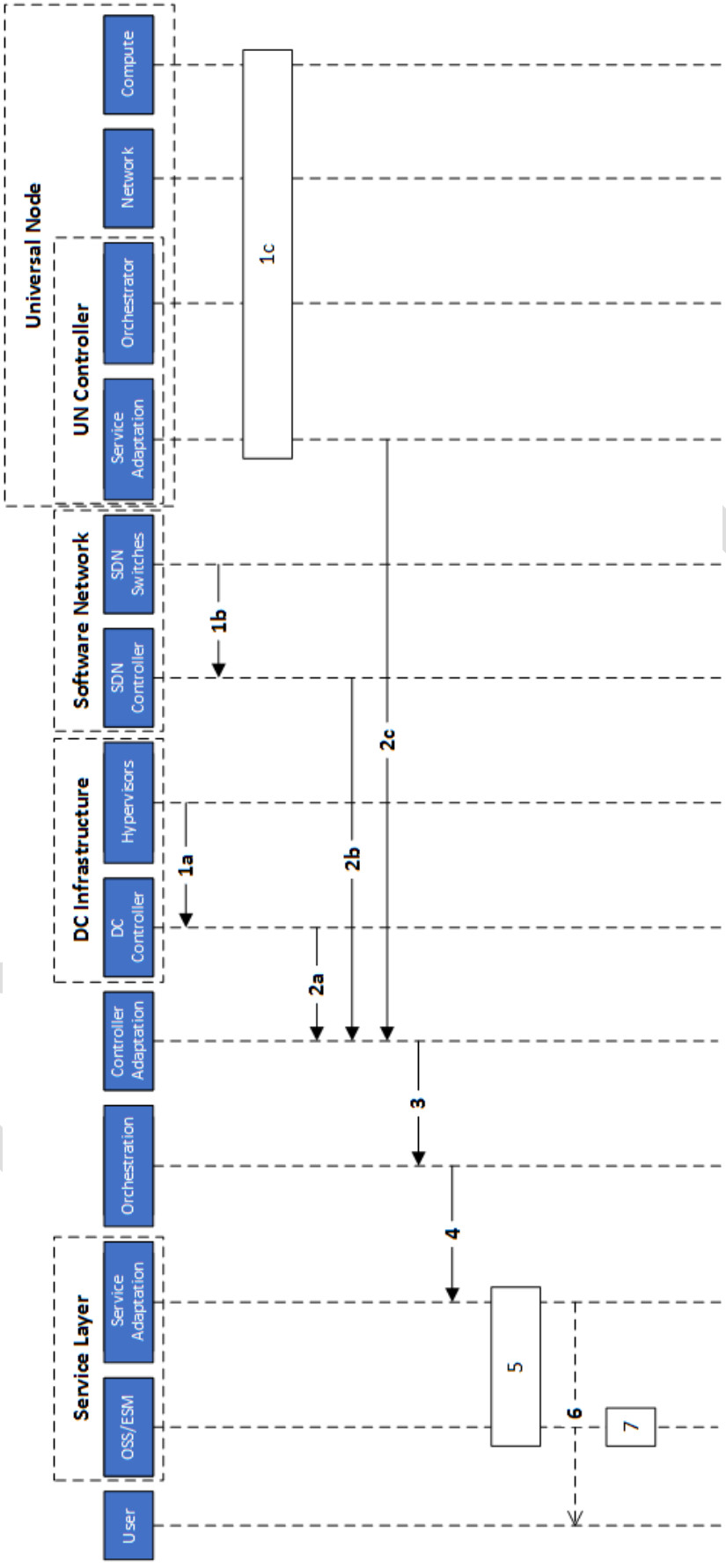

Figure 6.5: Sequence diagram: service confirmation

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

References to particular instances in the Infrastructure Layer can be assigned in a top-down manner. Once the Service Graph is instantiated at the Infrastructure Layer the individual instantiation of components can be acknowledged and propagated back to the Service Layer in order to allow operation and management tasks to be performed. This bottom up notification process is shown in Figure 6.5.

1. The Controllers / virtualized infrastructure managers collect the resource identifiers corresponding to the instantiated resources.

2. Controller Adaptation collects status and identifications to the allocated resources.

3. The Orchestration function collects resource allocation status of network configuration and VMs.

4. The Orchestration logic notifies Service Adaptation about the resource allocation regarding the NF-FG.

5. The Operation Support System (OSS) and / or Element Management System in the Service Layer configure the service logic in the NFs. (Some of the configuration might be done by the User)

6. The User is notified about the available services and service access points.

7. The Operation Support System (OSS) and / or Element Management System in the Service Layer operate and manage the instantiated services according to the SLA. (Note: management might be partially or fully done by the User.)

Note: While the requests to create a NF-FG and the associated status reports go through all the layers (programmability flow) the actual configuration of the NF logic (e.g., filling in the rules of a firewall) will go directly from the OSS/EMS to the various Network Functions.

6.2 Information models according to the reference points

The information models form the essential information units transferred between different reference points in the programmability process. As indicated in the introduction, the role of the Orchestration Layer is to reconcile the bottom-up resource information flow driven by the infrastructure with the top-down service information requests. Because of this dependency, we start with the description of the bottom-up information flow before going into the top-down information flow. The report corresponding to Milestone M4.1 as well as Section 6.7.4 provide further refine this process with respect to the monitoring process.

6.2.1 Bottom-up information flow

Information concerning networking, compute, storage resources or particular capabilities flows from the Infrastructure Layer up to the Service Layer on various timescales and different level of detail. Networking resources refer to available interfaces, bandwidth, delay characteristics, compute resources are for example CPU characteristics, RAM memory, and storage refers to available disk space. The possibility for infrastructure

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

elements to expose particular capabilities enables to expose specific execution environments (e.g., hardware-optimized implementations), particular Network Functions (e.g., firewall of type x).

Basic resource information, e.g., the existence of a switch or link, is seldom updated unless equipment is added, removed, or upon failure. This is a multi-level process: individual infrastructure resources announce themselves to their immediate controllers, and controllers consolidate information towards the Resource Orchestrator. In addition, resource virtualization might be applied in order to shield lower layer details to higher layers. Resource virtualization might occur at the level of compute and/or network controllers, the Controller Adapter or the Resource Orchestrator. More volatile information such as monitoring results on network links or Network Function utilization may be updated several times per second. High-volume data might be aggregated and modelled statistically to reduce the rate of updates.

6.2.1.1 **Co-Rm reference point**

Much, if not most, of the resource data such as available CPUs, RAM memory, link bandwidth originates from the Infrastructure Layer, where each node has to discover its own resources and capabilities. The Infrastructure Layer encompasses all networking, compute and storage resources. By exploiting suitable virtualization technologies this layer supports the creation of virtual instances (networking, compute and storage) out of the physical resources. Primarily, three domains of physical resources are considered:

- Universal Node (see D5.2 for details)
- SDN enabled network nodes (like OpenFlow switches)
- Data Centres (like controlled by OpenStack)

Exactly which resources these are depend on the type of infrastructure node but some examples may be network interfaces, CPUs, RAM and persistent memory, and other hardware resources such as acceleration cards for offloading packet processing or TCAMs for storing forwarding entries.

Detailed information about these resources might not be needed or allowed by the higher layer. Instead the virtualization functionality is responsible for providing a customized resource view for particular higher layer consumers and for required policy enforcements. In the case of an OpenFlow-enabled switch it is the OpenFlow agent software running on the device that provides this functionality. It hides the low-level resource details concerning RAM & TCAM memories and physical ports and maps them to parts of the conceptual OpenFlow switch. So, for example, instead of providing detailed information about such memories they are shown as FlowTables with a maximum number of entries (depending on the size of the memory). Similarly, not all physical ports may be shown to the higher layer, but only those enabled as part of the OpenFlow switch.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

How the virtualization is represented and transferred to the higher layers not only depend on the type of device but also on the protocol and protocol version used (see A.2.3.1.1). The nature of the information is also different among different technologies. For example, an OpenFlow-enabled switch can report which network ports it has but it doesn't include any link information. Such information has to be discovered by higher layers using for example link discovery protocols such as LLDP [LLDP].

6.2.1.2 Ca-Co reference point

The role of network and compute controllers is to consolidate and expose the collection of individual infrastructure resources of their corresponding domain towards the Controller Adapter. This not only involves resource fully contained within their domain, but also the exposure of interfaces towards other domains. As controllers might interface with multiple parties, they might virtualize the consolidated resources as part of this process. This enables hiding of lower layer details, as well as resource slicing setups.

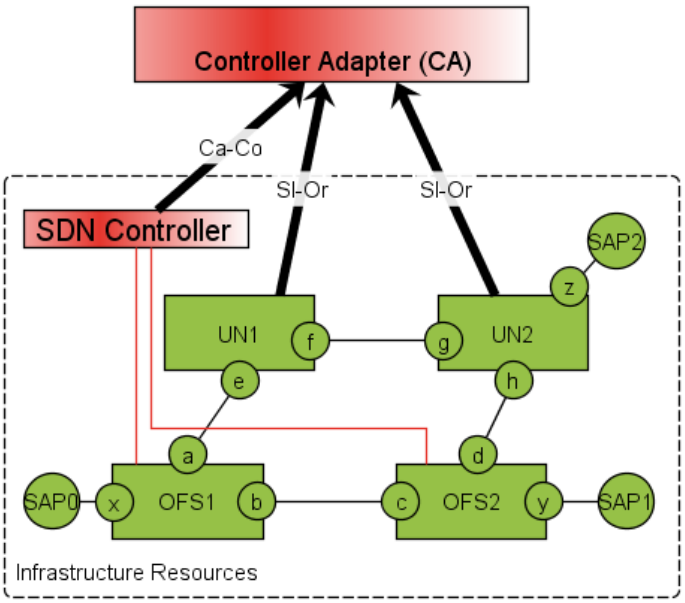

Figure 6.6: Bottom-up information flow at Ca-Co reference point

Figure 6.6 depicts the bottom-up information flow where two compute controllers (corresponding to UNs) and one SDN (network) controller expose information towards the Controller Adapter. This enables the Controller Adapter to consolidate the information towards the Resource Orchestrator (next section).

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

6.2.1.3 **Or-Ca reference point**

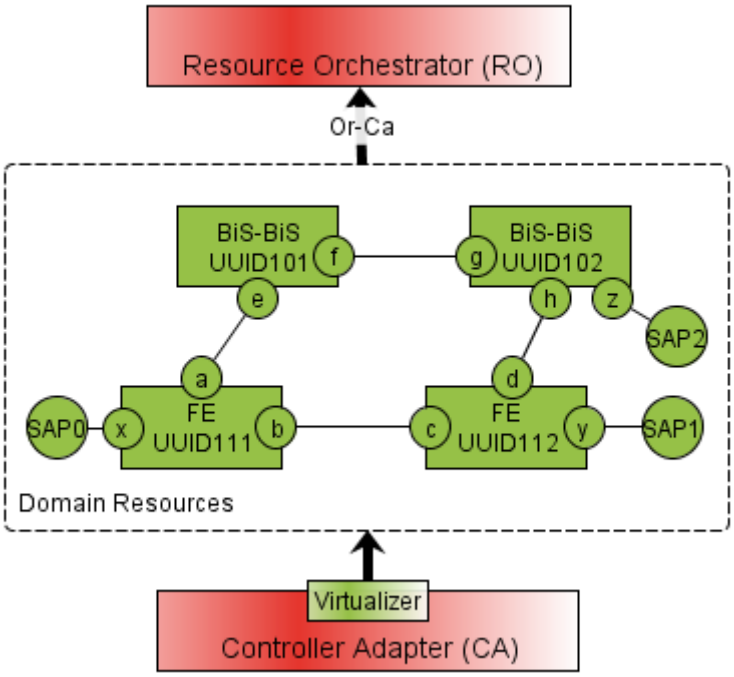

Figure 6.7: Bottom-up information flow at Ca-Ro reference point

The Orchestration Layer is split into two sub-components: resource orchestration and controller adaptation. The resource orchestration is a logically centralized function. Below, there could be many underlying controllers corresponding to different domains or technologies in practice. The controller adaptation is responsible to bridge between the controllers and resource Orchestrators. It offers technology independent, virtualized resources and resource information. Hence, the resource orchestration collects and harmonizes virtualized resources and resource information into a global virtualized resource view at its compute, storage and networking abstraction. It is important to note here, that the aim of the resource orchestration is to collect global resource view.

The global resource view in the Orchestrator consists of four main components; forwarding elements, compute host capabilities, hardware based or accelerated Network Function capabilities, and the data plane links that connect them. All of the resources must have associated abstract attributes (capabilities) for the resource provisioning to work.

In order to obtain this global view, consolidation might happen at different layers. While individual controllers might expose a virtualized view of the underlying resources and topologies, the consolidated view might rely on discovery mechanisms to detect further details, e.g., links (cfr. LLDP in previous paragraph). For example, Figure 6.7 illustrates the consolidated topology integrating the received views from the individual controllers (cfr. Figure 6.6) into one global topology.

Another type of resource that has to be discovered is Service Access Points (SAPs) representing devices connected to providing interconnection to customer networks.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The information sent from Controllers to the Controller Adaptation and Resource Orchestration may be adapted through virtualization functionality in order, for example, to hide lower layer details, slice network, compute or storage resources towards higher layers. The Controller Adaptation integrates virtualized topologies and resources coming from multiple controller entities, each responsible for different segments of network or compute domains. As the virtualized resource topology and capabilities is the main resource used to perform the orchestration of NF-FGs. To this end, a domain global view of all virtualized resources and capabilities are offered to the Resource Orchestrator:

- Compute and storage resources

  - Identifiers, location, capabilities (e.g., KVM, Unified Node type A, etc.), capacity, resource use

- Networking resources

  - Identifiers, links, forwarding elements, capabilities, capacity, resource use

- Service Access Points

  - Identifiers, location in topology, and, if possible, information on what they represent (such as an OSPF neighbour, etc.)

- Virtual/Physical Network Function instances/slices

  - Identifiers, Network Function types (note: this includes deployed VNFs and appliance based NFs), attachment points, capacity, current usage

6.2.1.4 **Sl-Or reference point**

The information flow to the Service Layer can be reduced based on what information is needed by different users. For example, certain users may have a very restricted network view that only presents them with the network endpoints that they are allowed to create Service Graphs in-between. These restricted views could also include logical/abstract links and nodes connecting the endpoints that the user can control in order to represent the limitations in, e.g., network bandwidth and latency. Included virtualized nodes may be used to restrict which Network Functions are available to the user, and the minimum/maximum capacity of those. Based on user policies defining which SAPs, which Network Functions and the level of network abstraction the total information transferred between the Resource Orchestrator and the Service Layer is virtualized.

In addition to the resource information coming from the Orchestration Layer, the Service Layer also needs additional information from other sources, for example information needed to translate the actual topological network endpoints (“IP address 1.2.3.4 on port 1 on switch 1”) to a physical Service Access Point (“Plug 1 in building 1 on street 1”) and finally to a user or site (“Jane Doe” or “Office 1”).

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

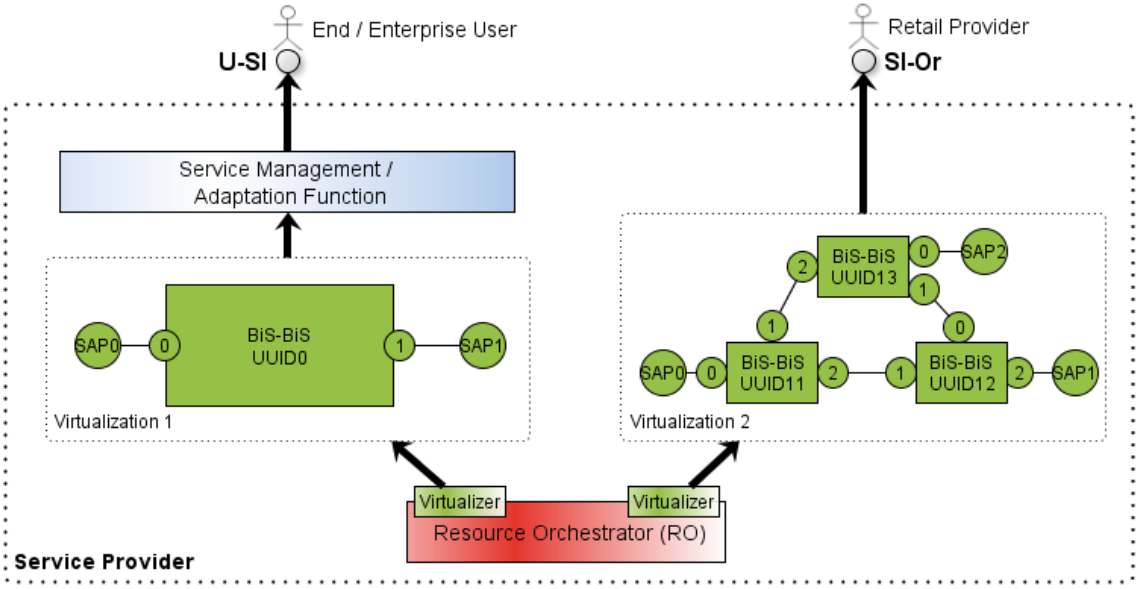

Figure 6.8: Bottom-up information flow at Sl-Ro reference point

The virtualization of resources from the Resource Orchestrator towards particular users is depicted in Figure 6.8. Depending on the user, a different resource virtualization is exposed. On the left side the network between the SAPs is exposed as a single Big Switch with Big Software towards the Service Management and Adaptation Function to a End/Enterprise User, whereas on the right side, more information is exposed towards a Retail Provider. Towards the latter a virtualized network of three BiS-BiSes each interconnecting with a particular SAP is exposed. This enables the Retail Provider to exploit this knowledge in order to formulate NF-FGs which use disjoint paths between SAPs or use the path which has lowest delay.

To summarize what information is transferred from the Orchestration Layer to the Service Layer:

- Service Access Points (SAP) that the user is allowed to connect in his Service Graphs, may be, e.g., its own offices and abstract endpoints such as “internet”

- Abstract/logical links and/or nodes summarizing the network topology (BIS-BIS) and its limitations, for example artificial limits such as a restricted amount of bandwidth, and physical limitations such as the latency between two network endpoints

- Abstract/logical nodes summarizing compute and/or storage resources, taking into account the user’s contract limitations (e.g., number of compute nodes with particular CPU and memory), but might also include particular Network Function instantiations such as a physical firewall appliance

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

6.2.2 Top-down information flow

6.2.2.1 U-Sl reference point

A Service Graph (exemplified in Figure 6.9) describes the service requested by a User and defines *what* service is provided, between which Service Access Points (SAP) is provided *where* the service is provided, and what are the associated service requirements, i.e., requirements on key quality indicators (KQIs). The provided service is described using Network Functions and their directed logical connectivity, and the Service Access Points. Finally, requirements on Key Quality Indicators (KQIs) attached to both Network Functions and the logical connectivity describes the service level agreements. These requirements and KQIs could be used to derive service level monitoring.

The Network Functions offered by the service provider may be either Elemental Network Functions (ENF), which perform a specific function (such as a NAT, Traffic classifier, transparent HTTP Proxy, Firewall function, etc.) or Compound Network Functions (CNF) that internally consists of multiple ENFs. A CNF performing, for example, a Parental Control function could internally consist of a sub-graph starting with a Traffic classifier followed by a HTTP Proxy and a Firewall. When traffic passes through the Classifier it could inform the following functions and steer traffic to either of them for either re-writing parts of the HTTP requests/replies (e.g.,., certain image URLs) in the HTTP Proxy or fully block the flow in the Firewall. *In the Service Graph we make no distinction if the requested function is a CNF or ENF, they are both represented simply as Network Functions.*

Connectivity is described as directed logical links connecting the Network Functions to each other and to Service Access Points (SAP). At this level a SAP is not necessarily tied to the network (e.g., as a specific IP address or switch port), instead it represents attachment points at the service level and may be, for example, a particular branch office identifier, a user name, a group of users, or a connection to another network such as the Internet. SAPs are depicted as part of the (virtualized) Infrastructure Layer.

The key quality indicator (KQI) requirements attached to Network Functions and to the logical links interconnecting them, represent quality goals matching the level (layer) of the service request, such as the number of users handled by a Network Function, the number of request per second handled, or the total service availability (SA) percentage. These KQIs are either calculated from related resource facing Key Performance Indicators (KPIs) – compute, storage and networking level performance indicators such as bandwidth, delay, etc.,– or they are KPIs themselves.

From information model point of view, the U-Sl reference point must pass a Service Graph with vertices and edges plus associated KQIs according to arbitrary grouping of connected NFs and logical links.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

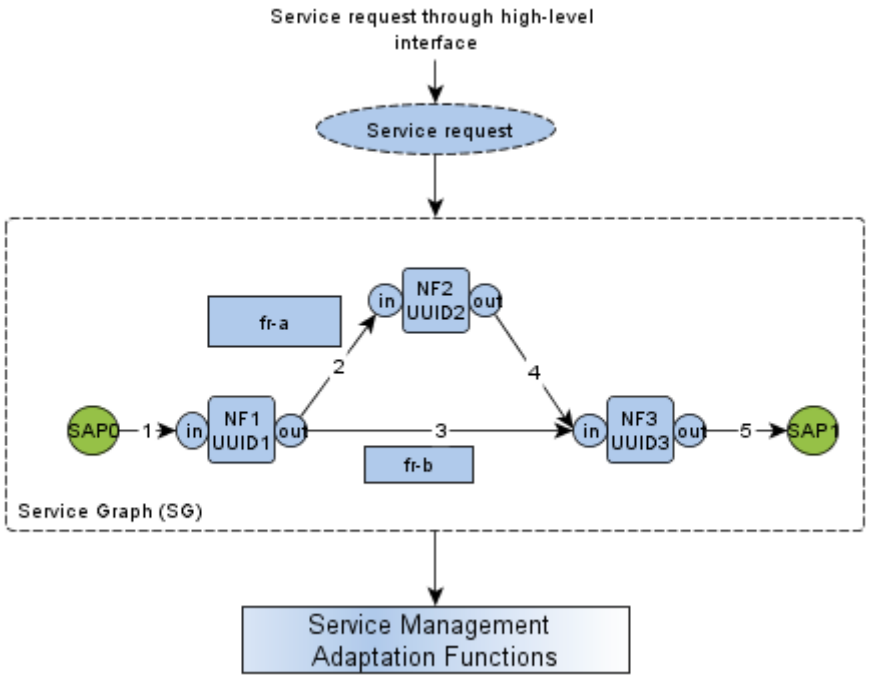

Figure 6.9: Top-down information flow at U-Sl reference point

6.2.2.2 Sl-Or reference point

The *Network Function Forwarding Graph (NF-FG)* passed through the Sl-Or reference point is a translation of the Service Graph to match the Orchestration Layer, at a level of detail suitable for orchestration (shown in Figure 6.10). This includes all the components of the Service Graph; Network Functions are translated/expanded into Elementary Network Functions (ENF) to which known decomposition to corresponding instantiable Network Function types exist in the NF Catalogue Manager of which the most important component is the Network Function Information Base (NF-IB). The latter maintains representation (images, code, etc.) of NFs together with deployment-related information (how to install, what are dependencies, etc.). A more detailed characterization of the NF-IB is given in Section 6.5. The latter is for example used for decomposing the previously mentioned Parental Control function into three NFs with internal connectivity); Service Management Adaptation Functions make sure that SAPs are translated/expanded into endpoints, identifiers meaningful at the network level such as a certain port on a switch or a collection of IP addresses; KQIs are mapped to resource facing and measurable KPIs and requirements on the ENFs. The KQI mapping may result in insertion of additional NFs into the NF-FG for measuring certain KPIs that cannot be provided in other ways. These KPIs and other configuration parameters are used by the Monitoring Functions (MF) which are also part of the NF-FG. The MFs provide different levels of information depending on the role (operator or user).

The differences in the information passed in the Sl-Or compared to the U-Sl reference point are that

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- (compound or abstract) Network Functions may be translated and decomposed into Network Function types, which are known to the Orchestrator for instantiation (Note: known Network Functions are defined in a Network Function information base, see Section 6.5); Note that this process is guided by the decomposition principles documented in Section 6.6

- All constraints and requirements must be formulated against compute, storage and networking resources. Note that KQIs as indicated in the SG are also decomposed into measurable KPI requirements which can be expressed in terms of compute, storage and networking requirements. As further detailed in Section 6.7.4, this might result into the interaction with monitoring components which themselves have compute, storage and networking requirements.

- NFs are mapped to exposed resources by the virtualizer of the Resource Orchestrator. The resource model exposed by the RO might be a single Big Switch with Big Software (as indicated in the previous section), but also might be a more complex resource model consisting of multiple BiS-BiSes.

Figure 6.10 depicts a NF-FG describing a service consisting of three NFs mapped to a virtualized Big Switch resource model. On the left side of the figure the mapping towards a single BiS-BiS resource model is depicted (for example for an End/Enterprise User), while the right side illustrates the mapping of NFs of the SG to the resource model consisting of multiple BiS-BiSes. This illustrates that a first level of orchestration might happen at the Service Layer, for example for OTT providers.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

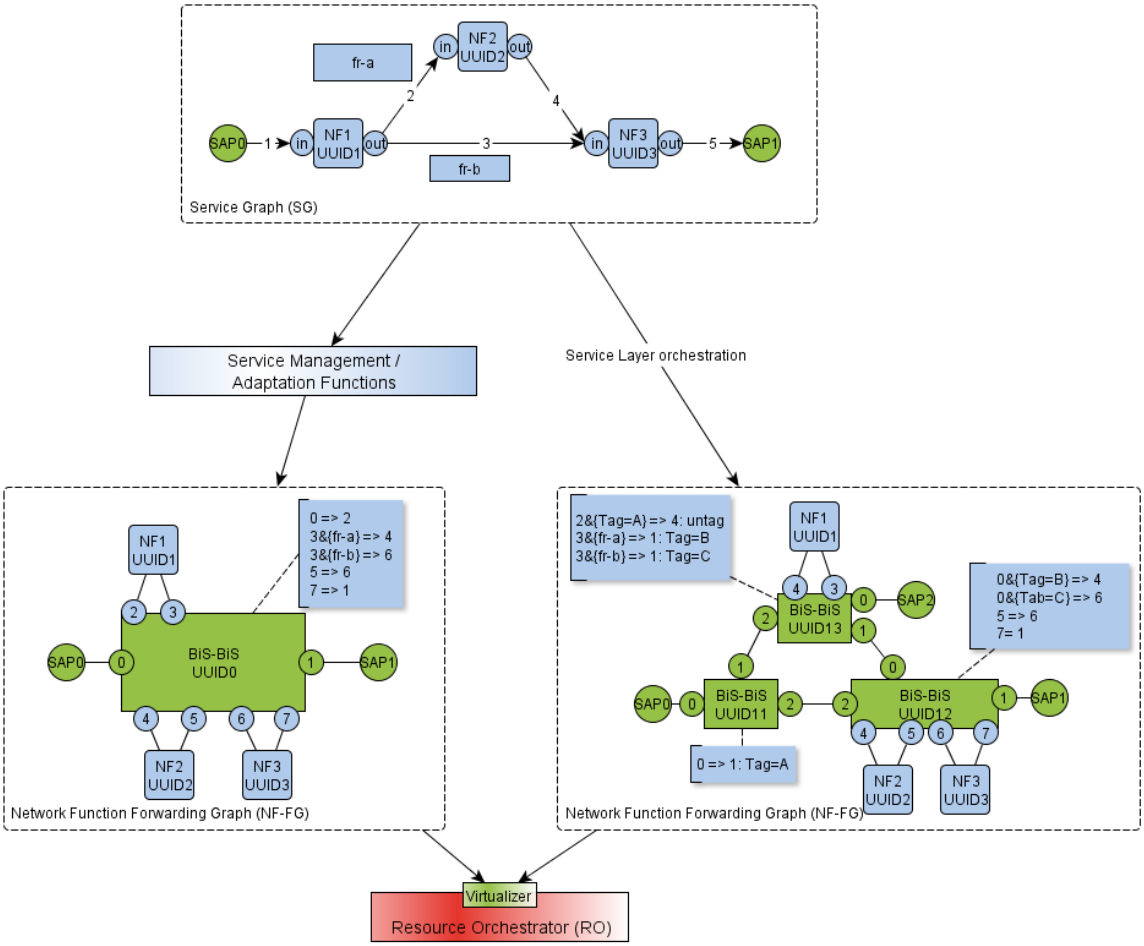

Figure 6.10: Top-down information flow at Sl-Or reference point

6.2.2.3 **Or-Ca reference point**

The role of the Orchestration Layer is to decompose and map NFs of the NF-FG received by the Service Layer to resources exposed by lower layers. The output of the Orchestrator is an instantiable Network Function-Forwarding Graph, which assigns infrastructure resources to each NF-NG component and adds the necessary overlay, producing a mapped NF-FG. This implies that individual deployable VNFs will be given UUIDs (Universally Unique IDentifiers), and will be mapped to infrastructure able to instantiate such VNFs. In addition, necessary network forwarding rules will be mapped to (virtual) switching functionality either in the form of OpenFlow switches or, for example, as virtualized switching rules in software switches of UNs.

The resulting information model represents the mapping, without actually deploying it. The actual instantiation will be triggered when the Control Adaptation component splits the required control actions and translates them towards the responsible network and or cloud controllers (via their northbound interface).

The resulting process is illustrated in Figure 6.11 with respect to the NF-FG received at the Sl-Or reference point. The green components depict the infrastructure as exposed by lower layers. The NF-FG of Figure 6.10 involving a virtualized Big Switch infrastructure is mapped

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

to a (potentially virtualized) infrastructure involving two Universal Nodes (UN) and two OpenFlow Switches (OFS). This implies the translation of switching rules of Big Switch with UUID0 to switching rules on the ports of the resulting UNs and OFSes.

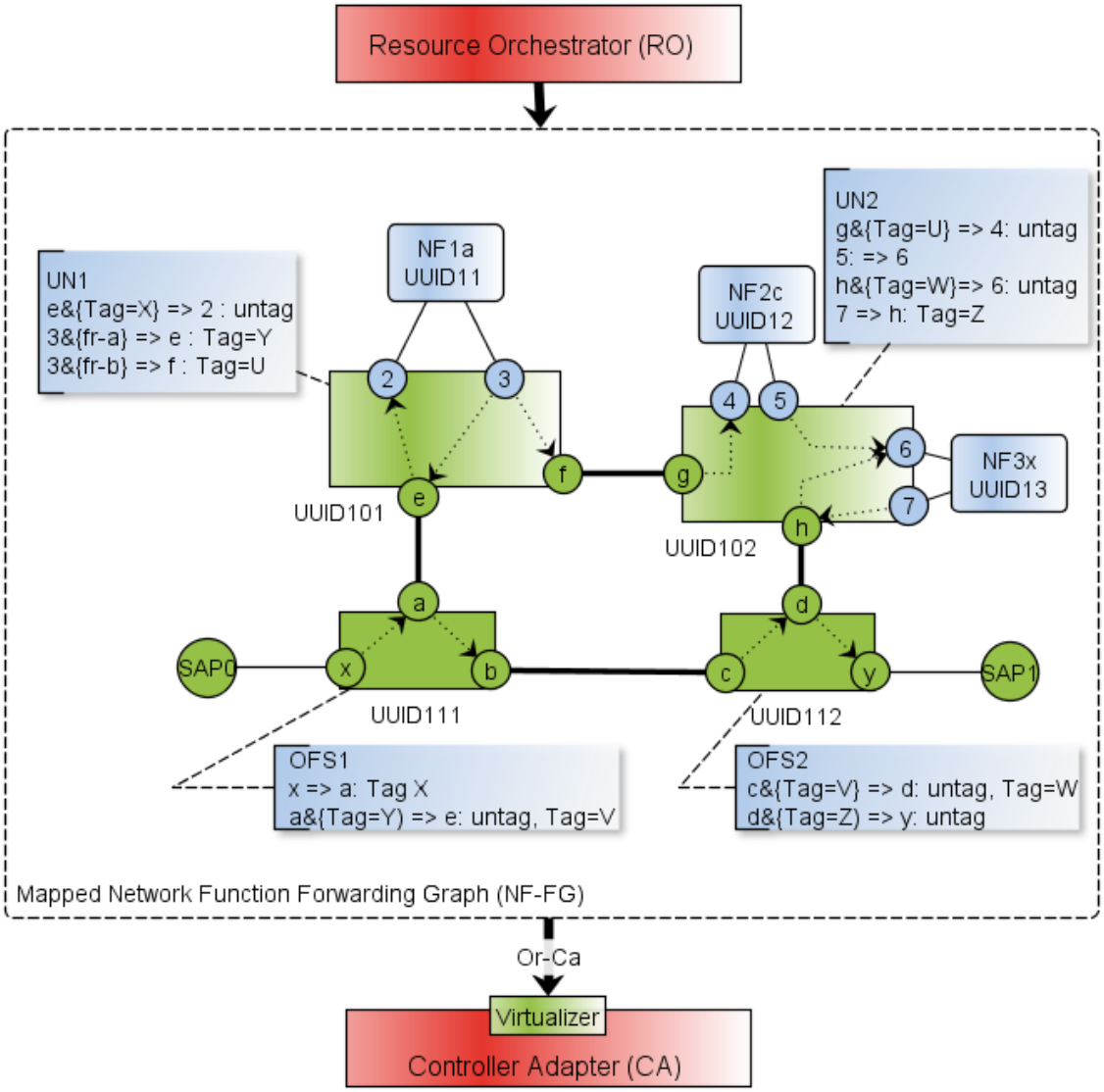

Figure 6.11: Top-down information flow at Or-Ca reference point

6.2.2.4 **Ca-Co reference point**

The Ca-Co reference point adapts various Northbound Interfaces related to different virtualization environments. In UNIFY we pursue the reuse and integration of some well accepted virtualization infrastructure managers like OpenStack for data centres and OpenDaylight for software-defined programmable networks.

On the other hand, when a Universal Node is connected to the Controller Adaptation, we foresee that the same NF-FG representation can be used to define the local scoped service request (NFs and forwarding overlay) and constraints as the output of the orchestration logic. Hence, the Controller Adaptation should only extract and pass the corresponding sub-graph to the UN.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Figure 6.12 depicts this process for the previously mapped NF-FG. Recursive orchestration occurs between the CA and the Orchestrators of UN1 and UN2 (following the same interface as for Sl-Or) for two partial NF-FGs, while the interconnection of these NF-FGs is delegated and translated to the northbound interface of particular SDN controllers. Existing technologies for these interfaces are discussed in A.2.3

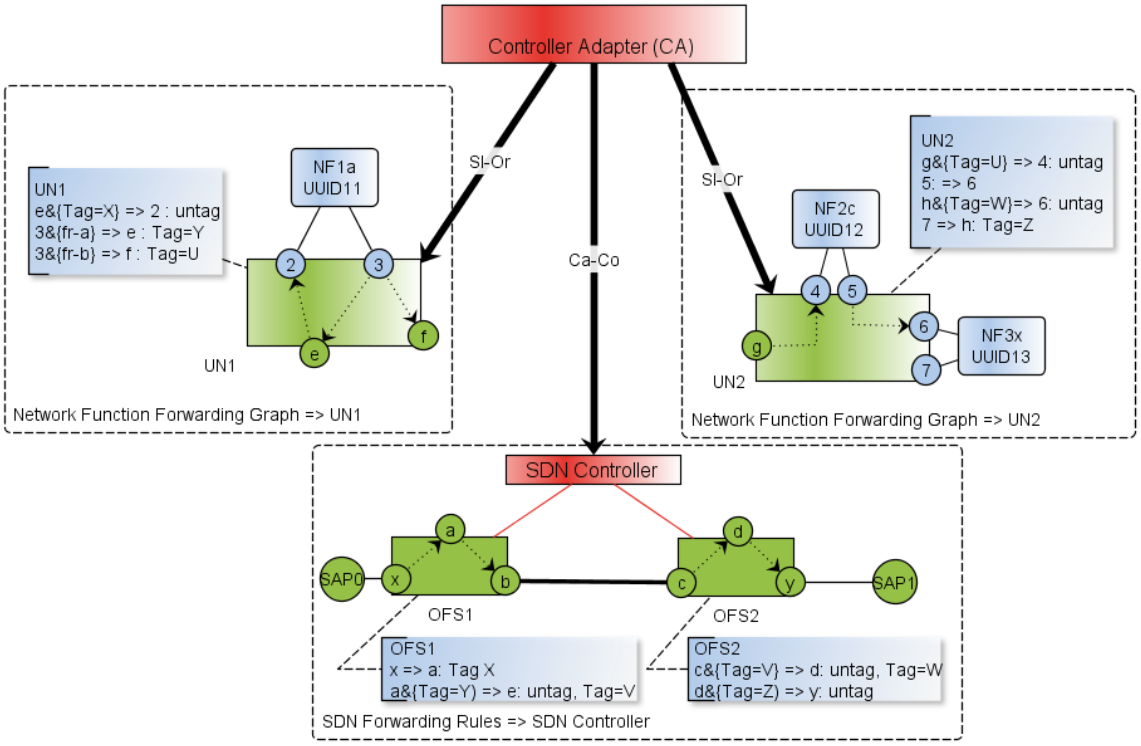

Figure 6.12: Top-down information flow at Ca-Co reference point

6.2.2.5 **Co-Rm reference point**

The Co-Rm reference point captures various southbound protocols of the different controllers. Such potential protocols and frameworks are described in Section A.2.3.1. In UNIFY we aim at exploiting and relying on existing controllers (e.g., OpenStack and OpenDaylight) as much as possible. However, for the Universal Node, due to the recursiveness of the orchestration architecture, partial NF-FGs can be delegated to the Universal Node, using the same interface as the Sl-Or interface. More details about this interface can be found in Section 7 and referred WP5 deliverables.

6.3 Specification of Service Graph

The Service Graph (SG) describes the actual service that the user is requesting to the UNIFY control and orchestration framework. The SG is a representation of the requested service that defines the service functions and its logical connectivity (i.e. how the service is provided), the Service Access Points to the service (i.e. where the service is provided) and the Service Level Specification to meet the Service Level Agreement (i.e. how to measure the successful delivery of the service).

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

In relation to the UNIFY architecture, the SG is created by the Application Layer to describe the service request to the Service Layer by using the U-Sl reference point. The Service Layer then translates the SG (service related definition) into the corresponding NF-FG (resource related definition based on compute, storage and networking).

UNIFY will base the SG model on the ongoing work done by the ETSI MANO. Concretely, the ETSI defines the Network Service (NS) element, which is the closest element related to the SG defined at UNIFY. This means that the SG definition will be based on the ETSI's NS.

The main components of the NS, and also in the SG, are described below. The SG describes the relationship between the Network Functions (NFs) and the links used to interconnect them. The links are also used to interconnect the NFs to the endpoints, which provides the interface to the existing network (they define the boundaries of the SG). The SG includes the following elements:

- Endpoints (EP): the EPs define the external interface of the SG (inbound and outbound traffic). They are related to the Service Access Points of the service.

- Virtual Network Function (VNF) and Physical Network Function (PNF): both the VNF and the PNF describe the Network Functionality to be performed, as well as its deployment and operational behaviour requirements.

- Virtual Link (VL): the VL describes the resource requirements that are needed for a link between VNFs, PNFs and endpoints of a SG.

- VNF Forwarding Graph (VNFFG): the VNFFG describes the topology of the SG by referencing the VNFs/PNFs and the VLs used to interconnect them. The VNFFG is defined as a set of Network Forwarding Paths (NFP), which describes individually each of these interconnections.

- Monitoring parameters: they represent the parameters that can be tracked for this NS to assure the proper level of service (also referred as service flavours) requested by the user.

Apart from the aforementioned elements, the NS defined by the ETSI MANO also include additional elements such as the service deployment flavour (which represents the service KPI parameters and its requirements for each deployment flavour of NS) or the auto scale policy (which represents the policy metadata and criteria for triggering the scaling of the NS).

6.4 Specification of Network Function-Forwarding Graph

The Network Function Forwarding Graph (NF-FG) is one of the key enablers of the programmability framework. The NF-FG information model is described in this section, being the basic element that supports the interactions between the Service Layer and the Orchestration Layer. The Service Layer translates the Service Graph provided by the user into the NF-FG, which contains enough details to perform the service orchestration. The

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

NF-FG is also used internally inside the Orchestration Layer for supporting different functions such as decomposition, embedding, scaling and optimization. The NF-FG evolves from its original definition in the Sl-Or interface (i.e. the initial NF-FG requested by the Service Layer) based on two orthogonal dimensions: layering and time. On the one hand, while the NF-FG progresses down in the architecture, more details and elements will be specified or added. Moreover, the functionalities performed by the Orchestration Layer will also decompose the components (e.g. Network Functions) and/or split the NF-FG into smaller sub-graphs. On the other hand, during the service lifecycle the NF-FG will also evolve from the initial service request to its final deployment in the physical infrastructure. Furthermore, internal re-optimization processes, any modification to the original service definition (at the Sl-Or interface) or external changes (e.g. infrastructure update) will cause modifications in the NF-FG.

According to the overall architecture, the NF-FG can be also used in the Ca-Co reference point, which is the NBI exposed by the Universal Node. The Orchestration Layer will split the service NF-FG into smaller sub-graphs and send the corresponding portion (sub-graph) to the target UN as a result of the embedding process. As shown in Section 7, this means that the Local Orchestrator at the UN consumes the NF-FG sub-graph and internally orchestrates the local resources (i.e. compute, storage and networking) to implement appropriately (based on KPIs) the requested functionality. This process will also further detail the NF-FG adding new elements related to the actual deployment. For instance, the Network Functions will be detailed with deployment-related parameters, such as the number and type of CPUs, memory size, storage size or network interfaces (e.g. virtual NICs). As a consequence, the NF-FG will become the central element of service transformation from the initial service request to the actual deployment at the Infrastructure Layer.

Based on preliminary work done by ETSI[9] the NF-FG model described in this section and shown in is an ongoing work at UNIFY project. The NF-FG is an abstract representation used to describe the service and the resources (i.e. compute, storage and network) involved to provide that service. Basically, there are four main top-level elements defined to describe the service: endpoints (EP), Network Functions (NF), network elements (NE) and monitoring parameters:

- The EP represents a reference point that defines the attachment of the NF-FG to the other elements outside.

- The NF represents the compute element that performs the Network Functionality demanded by the Service Layer or further decomposed by the Orchestrator.

[9] http://docbox.etsi.org/ISG/NFV/Open/Latest_Drafts/NFV-MAN001v061-%20management%20and%20orchestration.pdf

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- The NE represents the networking element that determines the interconnection between the NFs (including the EPs).
- The monitoring parameters that must be assured by the NF-FG to guarantee that the KPI requirements imposed by the Service Layer are met.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

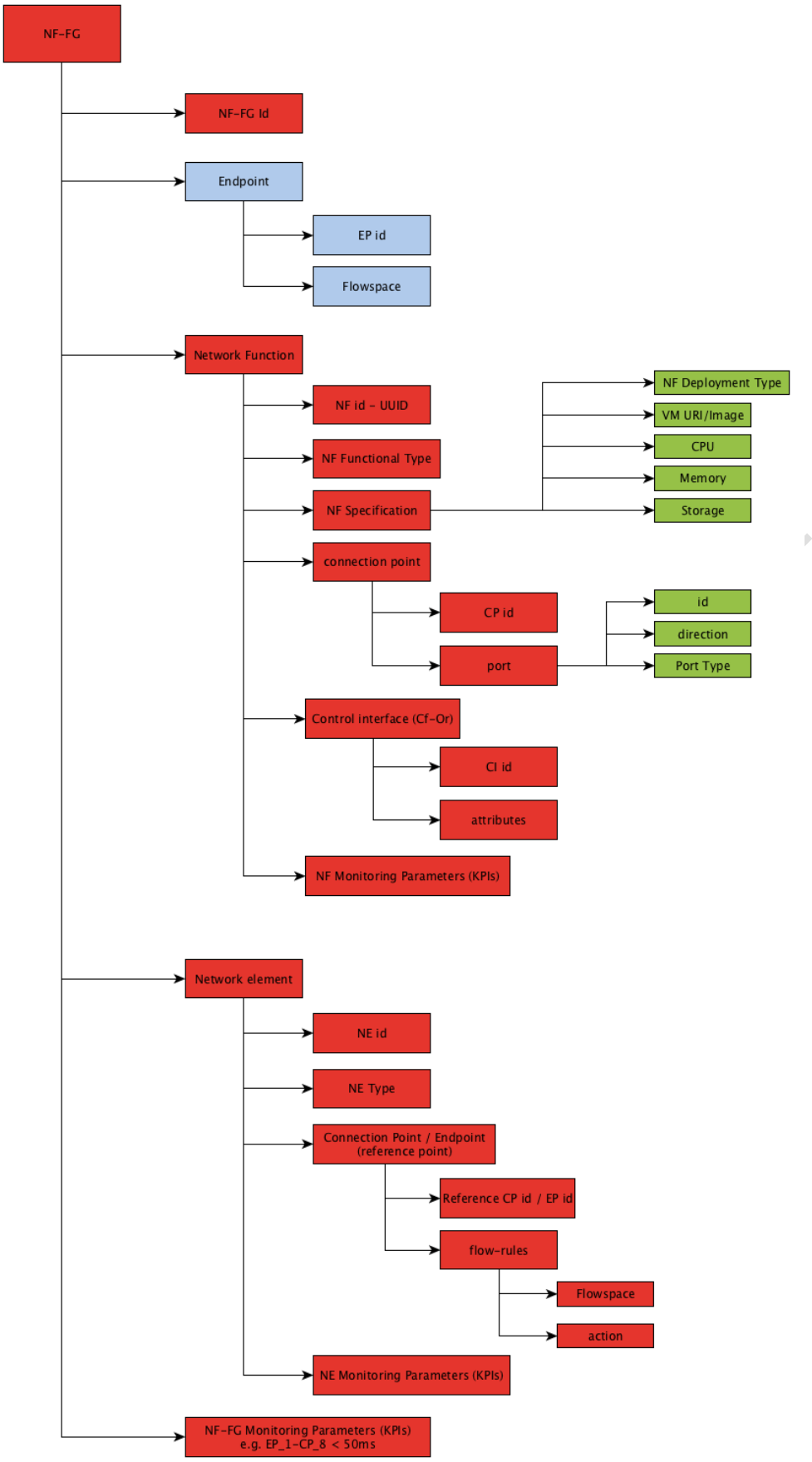

Figure 6.13: Network Function - Forwarding Graph (NF-FG) model

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The contents of the NF-FG must cover a wide range of processes and support several different views. On the one hand, it must support a dual view of service and deployment information. The former would remain the same as long as the service does not change, whereas the later would be dynamically constructed during the deployment process. On the other hand, the NF-FG must support a means for specifying both the resources requested by the Orchestration Layer and the final resources assigned after the deployment process is completed.

The complete set of top-level elements that comprises the NF-FG is detailed in the following Table 6.1. A first simplified version of the NF-FG has already been implemented and used in two separate demonstrations [Csoma2014a], [Csoma2014b], however, the terminology was slightly different.

Table 6.1: Top-level elements of NF-FG model

Element	Card	Description
NF-FG Id	1	Unique identifier of the NF-FG for a given domain (scope: domain).
Endpoints	0-N	Set of external reference points/interfaces of the service/NF-FG (Detailed in Section 6.4.1).
Network Functions	0-N	Set of (virtualised) Network Functions defined in the NF-FG (Detailed in Section 6.4.2).
Network Elements	0-N	Set of network elements defined in the NF-FG (Detailed in Section 6.4.3).
Monitoring Parameters	0-N	Set of monitoring parameters (KPIs) related to the NF-FG (Detailed in Section 6.4.4).

6.4.1 Endpoints

The endpoint is an external reference point (or interface) of the service (or the NF-FG) to the existing network. The endpoints are also used to define the boundaries of a network administrative domain. They must be uniquely defined at the Service Layer by means of a flowspace.

Table 6.2: Elements of endpoints

Element	Card	Description
Endpoint Id	1	Unique identifier of the endpoint for this NF-FG (scope: NF-FG). The EP id can be referenced at any place in the NF-FG when needed.
Flowspace	1	The flowspace that uniquely describes the endpoint. It could be generically defined as a header space extended with a collection of any network parameter

DRAFT

		(L1-L7) that univocally describes the reference point. A set of possible values for defining this element are the following: • Node identifier. • Ingress physical port. • MAC header parameters (source/destination MAC address, Ethertype, VLAN id, VLAN pcp). • IP header parameters (source/destination IP address, IP protocol, IP ToS). • TCP/UDP header parameters (source/destination port).

6.4.2 Network Functions

As defined in D2.2, the Network Function (NF) is a basic component of the UNIFY's architecture that performs a specific Network Functionality, being the core building block for data processing.

Based on the result of the decomposition process described in Section 6.6, the NF can be decomposed into a graph (i.e. described by another NF-FG) with additional NFs and NEs. The mapping between the reference points defined in the NF (i.e. connection points) and in the NF-FG (i.e. endpoints) is crucial for supporting the decomposition and the traceability of the overall process.

Table 6.3: Elements of NFs

Element	Card	Description
NF Id	1	Unique identifier of the NF for a given domain NF-FG - UUID (scope: domain). The same NF can be shared by several NF-FG at the same time.
NF Functional Type	1	Define the functional type of NF requested to perform a given Network Functionality. It represents an abstract functionality that can be implemented in different manners and the particular NF deployed in the end depends on different processes performed by the Orchestration Layer, such as the decomposition. The NF Functional Types are related to the information stored in the NF-IB, which also defines the possible alternatives to implement and decompose a given functionality.
NF Specification	1	Detailed description of the NF instance (described below). The elements included in the NF Specification totally depend on the NF Deployment Type and the actual implementation.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

		When dealing with a Universal Node, the NF Specification element is fully qualified internally by the Local Orchestrator based on the NF description (including the NF Functional Type) provided by the Orchestration Layer and the available resources in the physical node.
Connection points	1-N	Define the reference points of connection between the NF and the network elements (detailed below). The level of definition of these points can depend on the abstraction level at which it is defined.
Control Interface	0-N	Define the control interface exposed by the NF towards the Orchestrator. This interface is related to the Cf-Or interface defined by the UNIFY architecture, that allows a CtrlApp, which resides in a NF, to interact with the Orchestration Layer.
NF Monitoring Parameters	0-N	Set of monitoring parameters related to the compute resources to be performed at the NF.

The NF Specification depends on the NF Functional Type and gives additional details of the actual instance that implements the NF. Moreover, the specific set of components detailed by the NF Specification also depends on the NF Development Type. This later type defines all the possible alternatives described so far to implement a given functionality based on the architecture of the Universal Node defined in D5.2.

The NF Specification defines the resource requirements specified by the NF-FG, whilst the final resource assignment is determined once the NF is deployed on the infrastructure.

For instance, a NF Deployment Type 1, i.e. when the functionality is implemented by a full virtual machine, is described by the following elements:

Table 6.4: Elements of deployed NFS of the NF-FG

Element	**Card**	**Description**
NF Deployment Type	1	Define the type of the NF to be deployed. It can be also set of possible alternatives for implementing the needed functionality (*e.g. {Type1, Type2, Type3}*). The five possible deployment types of NFs considered at the Universal Node are: • NF Deployment Type 1: full virtual machine. • NF Deployment Type 2: isolated container running on the host. • NF Deployment Type 3: process running on the host. • NF Deployment Type 4: plugin to the Virtual Switch Engine.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

		• NF Deployment Type 5: a switch. An additional NF Deployment Type 0 is defined to represent an abstract type of NF not yet specified. In this case, the rest of elements of this NF Specification assume a NF Deployment Type 1, i.e. a full virtual machine.
VM URI/Image	1	Reference to the image used by the VM to implement the NF.
CPU	1	Detailed description of the CPU: model, architecture, number of cores, clock speed.
Memory	1	Detailed description of the memory: type, size.
Storage	1	Detailed description of the storage: type, (root and ephemeral) filesystem size.

The "connection point" (CP) is the external reference point of each NF element, which allows describing how the NF element is connected to other elements in the NF-FG (i.e. other NFs or Endpoints). The CP is shared between the NF, where the CP is properly detailed, and the NE, where the CP is referenced and its interconnection with other CPs and/or endpoints is defined. The CP is described by the following elements:

Element	Card	Description
CP Id	1	Unique identifier of the connection point for this NF-FG (scope: NF-FG).
CP Port	1	Detailed description of the port of the NF associated with the connection point.

Depending on the specific implementation and nature of the CP, the CP Port element could be detailed by a different set of elements. One possible alternative contains the following elements:

Element	Card	Description
CP Port Id	1	Unique identifier of the CP port for this NF (scope: NF).
Direction	1	Define the direction of the port. Possible values are: In, Out, or both.
CP Port Type	1	Define how the NF exchanges the packets with the underlying components. This element is platform specific. When using Intel DPDK possible values could be KNI or IVSHMEM.

DRAFT

The “Control Interface” (CI) element represents the interface exposed by the NF towards the Orchestrator (related to the Cf-Or interface). It is described by the following elements:

Element	Card	Description
CI Id	1	Unique identifier of the control interface for this NF-FG (scope: NF-FG).
Attributes	0-N	Detailed description of the attributes associated with the control interface.

6.4.3 Network Elements

The Network Element (NE) is an abstract representation used to describe the interconnection between the different elements in the NF-FG (e.g. endpoints and/or connection points) as a virtualized SDN Forwarding Element. Therefore, it describes the interconnection between the NFs and represents the networking resources of the NF-FG. The endpoints are also connected to them, being the incoming and outgoing reference points of the NF-FG. On the one hand, the NE can be used to abstract the whole interconnection as a Big Switch and uses the flow-rules to describe the connections between the NFs. On the other hand, the NE can be used to describe a networking resource from the underlying infrastructure, such as an OpenFlow switch or a Universal Node (the networking part).

The network element is described as follows:

Table 6.5: Elements of Network Elements

Element	Card	Description
NE Id	1	Unique identifier of the network element for this NF-FG (scope: NF-FG).
NE Type	1	Define the type of the network element. Currently different types are considered, such as “Big Switch” (BS), “OpenFlow Switch” (OFS) and “Universal Node” (UN).
Connection point / Endpoint (reference point)	0-N	Define the set of CP and/or EP that comprises this NE. Both the CP and the EP are detailed by the proper NF or the endpoint, respectively. The NE only refers to the appropriate reference point by its identifier. Because of this, the CP id and the EP id must be unique at the NF-FG scope.
NE Monitoring Parameters	0-N	Set of monitoring parameters related to the networking resources to be performed at the network element.

The connection point / endpoint element is described as follows:

Element	Card	Description

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Reference CP Id / EP Id	1	Reference to a connection point or an endpoint. To assure the uniqueness for this reference, the CP id and the EP id must be unique at the NF-FG scope.
Flow-rules	0-N	Describe each connection between the NFs (i.e. endpoints and connection points) as a flow-rule (detailed below).

The Flow-rules element contains the following elements:

Element	Card	Description
Flowspace	1	Detailed description of the "flowspace" to be performed at the Big Switch. The priority of the rule and a set of matches are detailed. Possible values for defining the match element are the following: • Ingress port (e.g. connection point or endpoint). • MAC header parameters (source/destination MAC address, Ethertype, VLAN id, VLAN pcp). • IP header parameters (source/destination IP address, IP protocol, IP ToS). • TCP/UDP header parameters (source/destination port).
Actions	1-N	Detailed description of the "action" to be performed by the Big Switch when the parameters detailed in the previous element are matched. The most basic action is defined by a "type" element (e.g. output) and the egress port (e.g. connection point or endpoint).

6.4.4 Monitoring parameters

The Service Graph processed by the Service Layer describes how the service delivery must be measured by adding Key Quality Indicators (KQIs) to the Network Functions and their connectivity. These KQIs represent the quality goals to achieve the expected service level and must be transformed by the Service Layer into the proper Key Performance Indicators (KPIs) associated to the elements described by the NF-FG.

The KPIs can be attached to the overall NF-FG as a "monitoring parameters" element, which defines the goals for the whole NF-FG. The bandwidth and delay are some of the possible KPIs already considered.

There are other more specific KPIs that can be attached directly to some NF and/or NE. A "monitoring" element has been added to the NF and NE elements to address this possibility.

DRAFT

6.5 Network Function Information Base

The Service Graph initially requested by a user is described by Network Functions (NF)/apps and their logical connectivity. At this level the Network Functions/apps might be either Elemental Network Functions (ENF) with specific function (e.g. NAT) or Compound Network Functions (CNF) which means that they are composed of several ENFs. Both ENF and CNF are represented as Network Functions in a Service Graph. As explained in Section 6.2 the Network Functions in such a graph (CNFs) are further decomposed into ENFs in the following layer and the NF-FG might be expanded with additional Network Functions required for measuring KPIs. Therefore, Network Functions appear at multiple layers and the same NF will have different views (Compound Network Function, Elementary Network Function or Application) at the different layers. However, for the Orchestrator to understand the NF abstractions at networking, compute and storage resource level, there must be a NF database or catalogue containing these models. This database is referred as Network Function Information Base (NF-IB). To be more precise, this catalogue includes the following information for each NF: i) NF interface descriptions ii) NF implementation and iii) NF resource requirements. Figure 6.14 illustrates the model used for NF description in NF-IB. Also for each NF, an id, type and a list of dependency on other NFs should be stored in the database. The definition of id and type used in this model are similar to the definition in the NF-FG model explained in Section 6.4.

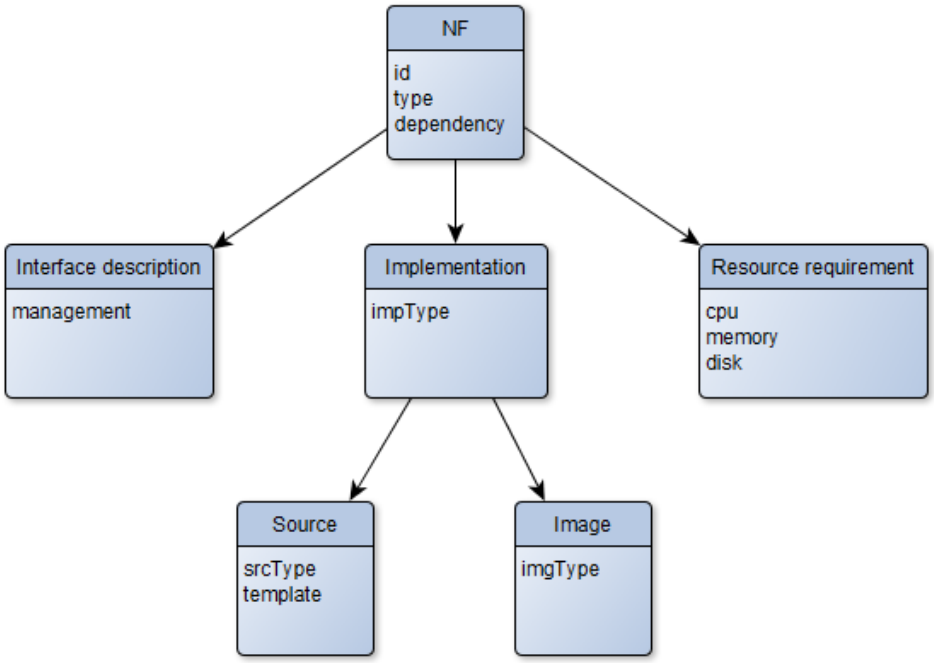

Figure 6.14: NF description model in NF-IB

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Rather than providing a complete formal model, below we give an example on how these attributes might be filled for a NAT (V)NF implemented in Click modular router the following information is stored in the NF-IB:

- Id: NAT
- Type: type2/type3 (it can be run as a Click process in the host or it can be run in an isolated container in a host)
- Dependency: IPAddrRewriter, Classifier (these Click elements/(V)NFs are required to be able to implement a NAT in Click)
- Interface description:
  - Management: Click element read/write handlers such as mappings(read-only), nmappings(read-only)
- Implementation:
  - ImpType: Click modular router
  - SrcType: C++
  - Template: Click script template
- Resource requirement:
  - CPU: 1 CPU core x86 with at least 600 Mhz clock frequency
  - Memory: 10 MB
  - Disk: 10Mb

In this example, we considered that the NF is implemented as an element in Click modular router. Therefore, its source code (C++) and its corresponding Click script template are required to be stored in the NF-IB to be able to run this (V)NF as a Click process in a host/container in case it is requested in a service chain. However there are other possibilities to implement a (V)NF. An example is to have x86, amd64 VM images with firewall functionality, for AWS, QEMU. Similar to Click implementation these images should be stored in NF-IB to be used once they are requested in a service chain.

In Table 6.6, we report some of the possible Network Functions to be stored in NF-IB.

Table 6.6: Network Functions

Category	**Network Functions**
Switching elements	BNG, CG-NAT, router
Tunneling gateway elements	IPSec/SSL VPN gateways
Traffic analysis	DPI, QoE measurement
Security functions	Firewalls, virus scanners, intrusion detection systems, spam protection, parental control
Mobile network nodes	HLR/HSS, MME, SGSN, GGSN/PDN-GW
Converged and network-wide functions	AAA servers, policy control, charging platforms
Application-level optimization	CDNs, Cache Servers, Load Balancers, Application

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

	Accelerators
NGN signalling	SBCs, IMS
Functions for home environment virtualization - RGW	DHCP server, PPPoE client, Port mapping
Functions for home environment virtualization - STB	Media streaming (VOD, NPVR, TSTV, OTT clients), Media cache

As explained in Deliverable 2.2 Section 4.2, in the Service (Graph) Adaptation sublayer of Service Layer there is a module responsible for managing the NFIB..Tasks such as database updates (add/remove/update NFs) and NF request are addressed by this module. Figure 6.15 illustrates the functional architecture of the Service Layer and NF-IB manager in this figure is the sub-module performing all the NF-catalogue related tasks.

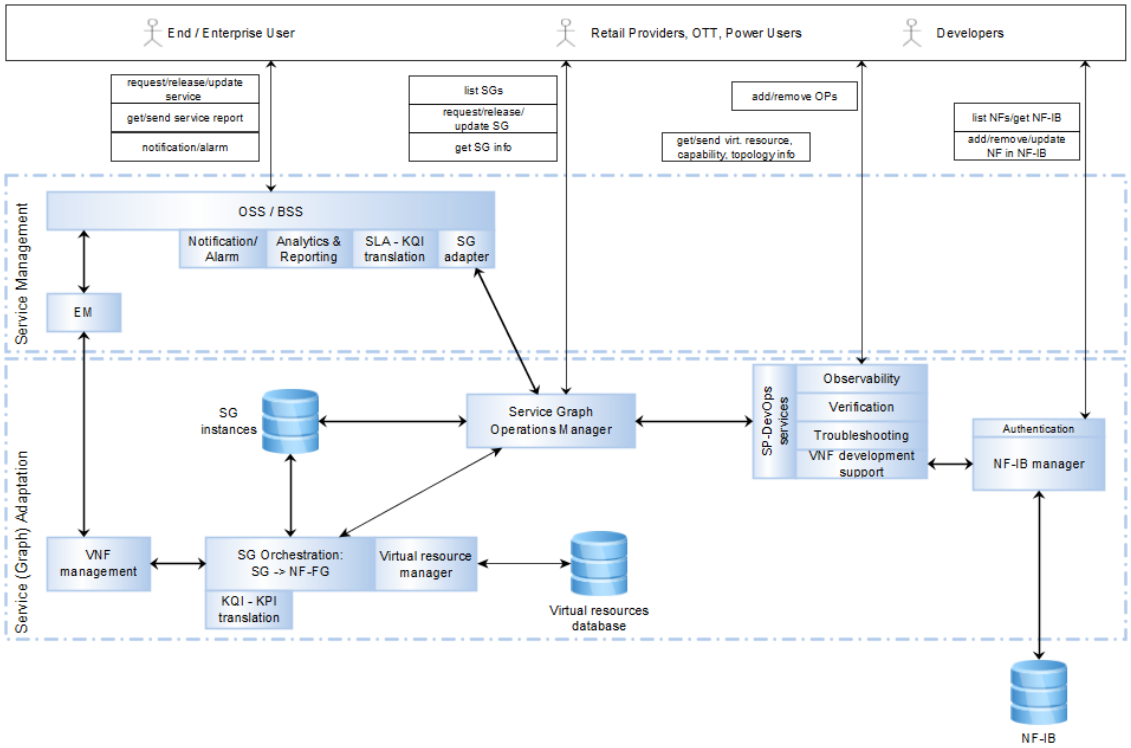

Figure 6.15: Service Graph Abstraction module

This catalogue is a cross-layer (Service/Orchestration) entity in the sense that it is understood and populated at the NF level by the Service Layer, but may be used by the Orchestration Layer to translate and optimize NF placement according to the given constraints, resources and operation policies. For the NF placement optimization, it is possible that the Orchestrator further decomposes the NFs based on existing decomposition rules. Service decomposition is detailed in Section 6.6. As will be explained, the decomposition rules should be stored in the NFIB or a mapping database which NFIB interacts with. Each of the decomposition rules can be represented by the NF-FG model explained in Section 6.4. Therefore, the database can include a list of dictionaries, each of which defining a NF but the values in the dictionaries are lists of decomposition rules represented as different NF-FGs.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

To be more precise in describing the required interactions between different modules and databases, we briefly report the used information in different stages (layers) stored in multiple databases.

- Once the Service Layer receives the high-level service requests from the customers, the SG adapter module of the Service Management sublayer (see Figure 6.15) translates the service request to a Service Graph.

- The Service Layer is also responsible to map the Service Graph to NF-FG. For this translation, the Service Layer uses a database referred to as virtual resources database which includes information such as restricted topology view. The mapping (SG->NF-FG) is stored in a special database called SG instances and the NF-FG is sent to the Orchestration Layer.

- In the lowest Orchestration Layer the NF-FGs coming from the Service Layer should be mapped to the resource topology. To this end information in NFIB and possible decomposition rules for NFs are required to have an optimal mapping. The mapping can be optimized considering different objectives such as minimizing the resource consumption and cost, maximizing the requests acceptance ratio, minimizing network load, etc. If the decomposition rules are stored in a separate mapping database then, there should be an interaction between this database (Mapping DataBase) and NFIB to provide the NFs decomposition rules.

Figure 6.16 illustrates the interaction between different layers and the two databases, NFIB and Mapping DataBase.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

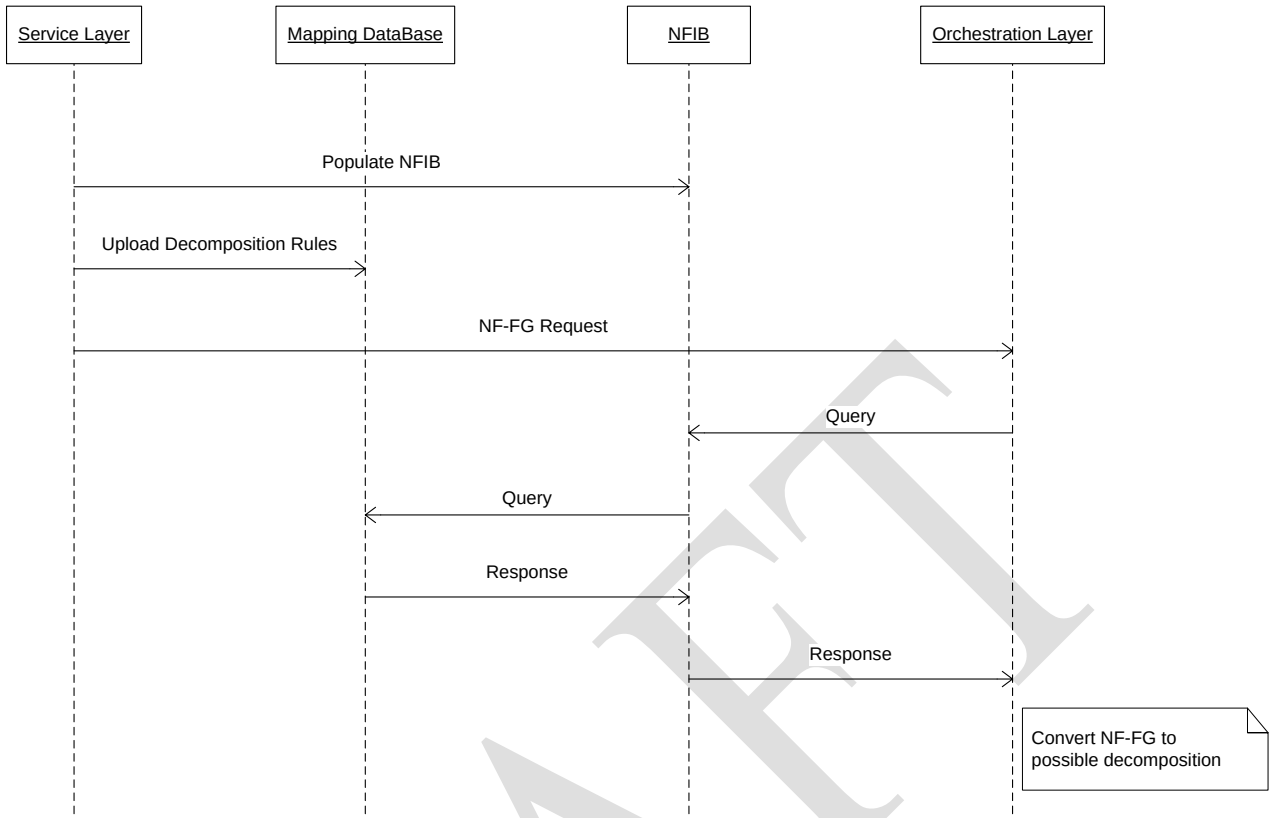

Figure 6.16: Interactions between different databases in different layers

Note that both Mapping DataBase and NFIB are populated by the Service Layer but used for the Orchestrator Layer for optimization (based on resource requirements/models), automated NF decomposition (see Figure 6.19), and instance deployment through the controllers.

Now, we explain the steps required to add an NF to the NF-IB (NF-IB population in Service Layer).

Figure 6.17 illustrates the corresponding sequence diagram. Once the NF developer uploads the source code/template/image of a Network Function (e.g. through a Web GUI) to the NF catalogue manager in the Service Layer, several checks should be performed before adding the new NF to the database: i) In case of new source code, it should be checked that the code is compiled ii) in case of dependency on other NFs e.g. IPAddrRewriter and Classifier in case of NAT example, their existence in the database should be checked iii) the NF (mgm) interfaces should be extracted. Then the NF catalogue manager could, for example interact with an entity referred as resource estimator to get the information about the required resources of the NF. Later on, we explain a possible approach for implementing this resource estimator. Such an entity is required to quantify the resource requirements of NFs to enable/simplify the mapping of (V)NFs to the infrastructure. In case of a success in all the above steps, the new NF is added to the

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

database and the NF developer is informed of the successful addition. It might be the case that NF developers are not willing to provide the source code of their apps. Therefore, some of the mentioned steps such as compiling the source code and (mgm) interface extraction might be skipped.

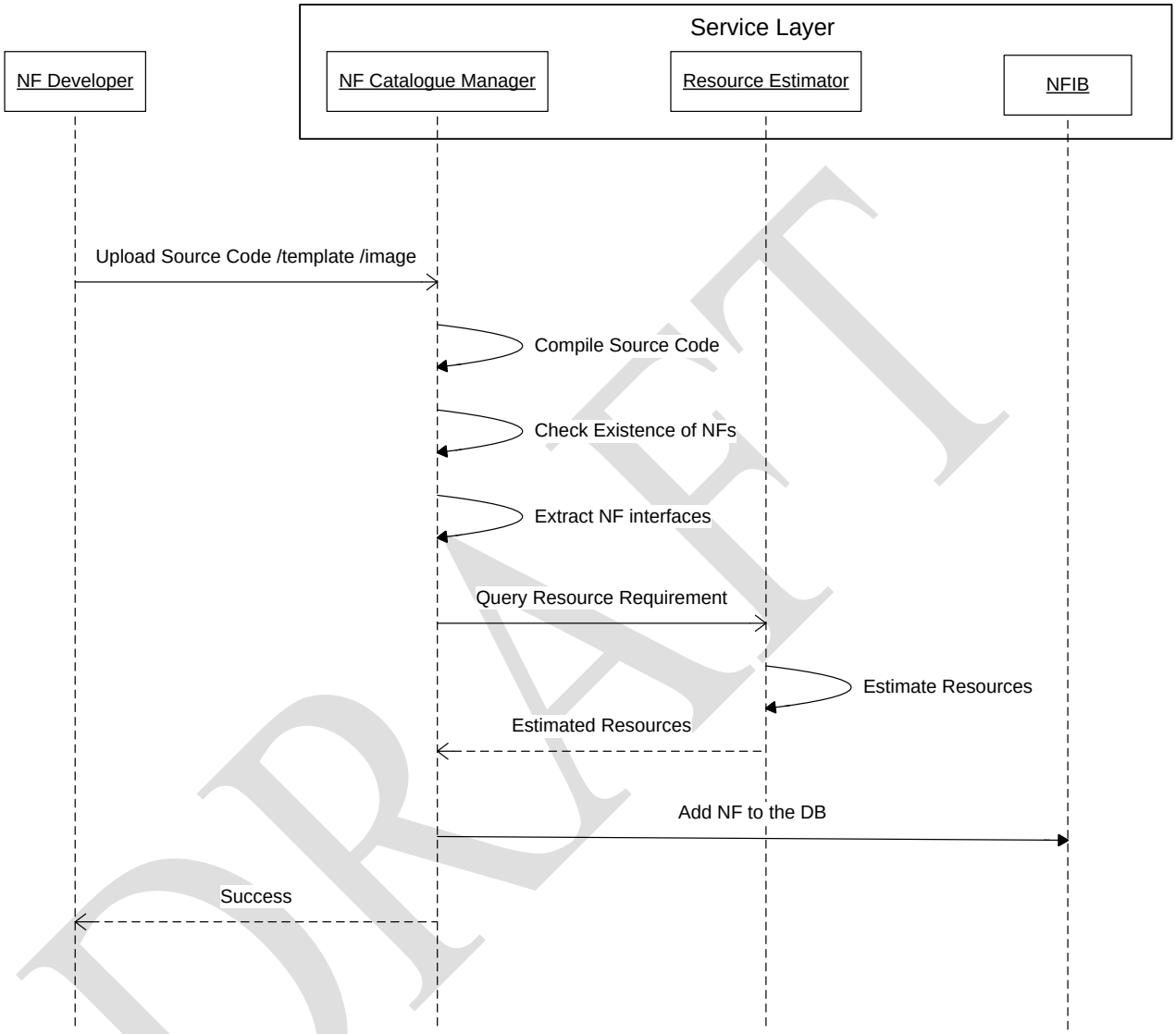

Figure 6.17: Processes to add new NF to the NF-IB

It is worth mentioning that in case of removing a NF from the database, the NF catalogue manager should first check the dependency of other NFs. In case of no dependency, it can be removed from the database.

We referred to a resource estimation entity for obtaining resource requirements. In order to estimate the required resources for NFs to be included in the NF catalogue, the framework depicted in Figure 6.18 can be considered. In this framework, NF resource model is conditioned on (discrete) parameter values (e.g. number of flows, inputs, etc.) and is characterized by CPU, MEMORY and DISK USAGE requirements.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The derivation of NF resource model is characterized by Test Environment which is performed through the following tasks:

- The required resources (CPU, MEM, Disc usage) for a NF are measured.
- The measurement is performed for discrete values of parameters.
- The estimation/evaluation is conditioned by the test machine properties (CPU, OS, etc).

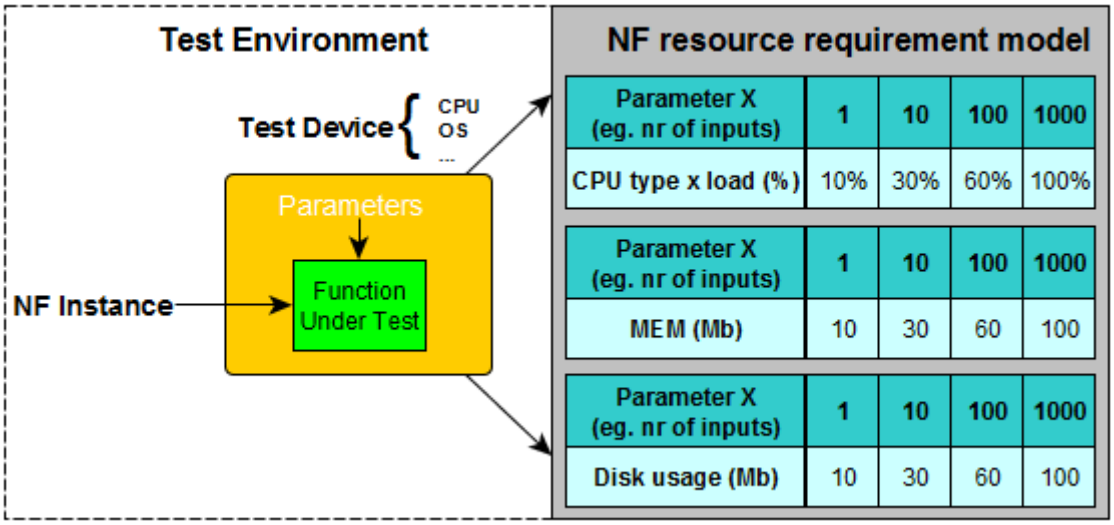

Figure 6.18: Resource estimation framework

Alternatively, resource estimation methods can be used as proposed in [Wang2013]. Their estimation is based on analysing the interaction between user behaviour and network performance. Therefore, the method can dynamically adjust the resource estimation in case there is a change in the QoS requirements.

A Network Function will have an *abstract type* view in the interface between the Service Provider and the Orchestrator. This view will include the type and the identifier (id) of the NF, as well as the resource requirements of the NF as explained in the NF-FG model in Section 6.4. Requirements can include, compute (CPU), memory and storage parameters. An example for such a NF abstraction is: “Firewall Type 1”. The Orchestrator will map the Network Function types to virtual NF instances. The virtual NF instance is a given type of realization of the abstract Network Function type at a given resource and location (as seen by the Orchestrator). The realization type depends on the type of the infrastructure at the given location. For example, a firewall Network Function type can be realized on a dedicated firewall hardware appliance, or it can be realized in a virtual machine which may run on a generic x86 environment.

At the Infrastructure Layer that implementation of the virtual NF instance will be used which is on first hand compatible with the infrastructure environment and on second hand, which is the best optimized for the given environment. Deciding which implementation to

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

use depends on the Service Controller or the Universal Node (see Section 7 for details). Examples of the possible implementations of a firewall are x86, amd64 VM images with firewall functionality, tailored, e.g., for AWS, Qemu or other virtualization environments. A scalable/elastic route example is also described in details in Section 8.1.

A basic version of the NF-IB containing only the most relevant, essential components have been implemented and demonstrated in [Csoma2014a]. Other parts will be added to our prototypes during the next phases of the project.

6.6 Service decomposition

Service decomposition is the process of transforming a NF-FG containing abstract NF(s) to NF-FG(s) containing less abstract, more implementation-close NF(s). This can also include dividing the functionality of a complex NF to more, less complex NFs. In UNIFY, we have a generic concept of UNIFY(ed) service decomposition as described in this section, and two realization options, the NF-IB-based (aka white-box) and the CtrlApp based (aka black-box) decomposition as described later in this section.

UNIFY(ed) service decomposition is an important concept in UNIFY, sometimes referred to as Model-based decomposition. This implies on one hand a *time aspect* of the decomposition: there is a decomposition Model, made (decided) in design time, while in execution time the Model is static, the instantiation is dynamic taking e.g. actual resources into account. The model is basically the decomposition model set. A decomposition rule in generic form is a NF-FG → {NF-FG} mapping (which means that a NF-FG will be transformed to another NF-FG or a set of NF-FGs).

On the other hand, the model based decomposition implies *abstraction*. The model describes abstract type to type mappings, not instances: the model stored in the NF-IB can live without ever instantiating any NF. The model is also a dynamic entity, it can change over time: e.g. new possible realization options can be developed for an abstract NF.

The model based service decomposition also refers to the *way the Orchestrator decomposes* an NF-FG's NFs. It will decompose them as long as there are no possible further decompositions. In this sense the Orchestrator thinks that it has received a high level NF-FG with abstract NFs; while it decomposes to low-level, "atomic" NFs, a.k.a., instances, according to his view. Taking into account the recursion possibility in the UNIFY architecture, an NF "instance" of a higher layer can be an abstract NF to be further decomposed by a lower layer.

The model-based service decomposition concept includes additional aspects, like:

- The approach that there can be multiple decomposition options for an NF
- The approach that the decomposition is automatic
- The workflow (Service Layer, Orchestrator, CtrlApp)

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- The involved entities (OR, NF-IB, Ca-Or interface)

Most of the concepts and processes introduced in this section are initial or ongoing working assumptions to give initial directions to the decomposition functionality to be further developed in later stages of the UNIFY project.

6.6.1 NF-IB based decomposition

The model-based service decomposition allows for the step-wise translation of high-level (compound) Network Functions into more refined Network Functions, which can eventually be mapped onto the infrastructure. The decomposition model can be stored in the NF-IB as a set of decomposition rules. The service decomposition offers the following:

- *Adaptation logic for ensuring SLAs*: high-level Network Functions are decomposed according to the required SLAs and parameters as e.g. the number of users. The abstract firewall functionality can e.g. be decomposed into a single firewall for one user, or into several load balanced and redundant firewalls, for 10,000 users.

- *Support for DevOps concept*: the initially limited NF-IB can be extended over time, allowing "to subclass" Network Functions targeted at specific hardware environments. While initially only a purely software-based firewall might be offered, the firewall functionality can be optimized for the execution on Universal Nodes.

The Service Layer's task is to decompose the abstract / compound Network Functions, until an instantiable NF-FG is obtained, which can be passed to the Orchestrator. However, as there might exist a multitude of possible NF-FG realizations for a single Service-Graph, the following questions arise:

1. To which extent does the Service Layer already take resource availability into account?

2. To which extent should non-fully decomposed Network Functions be passed to the Orchestrator?

While failing to take resource availability into consideration might prolong the provisioning process of a service-graph as the Service Layer "blindly" proposes decompositions, the same holds true once the Service Layer takes all resource information into account, effectively superseding the orchestration process.

Regarding the second question, we generally observe the *necessity* to allow for passing non-fully decomposed Network Functions in the light of layered or recursive orchestration stacks. Consider e.g. a multi-provider with multiple sub Orchestrators for different domains. In this case, different sub Orchestrators may allow for different competing implementations and the inner workings of the sub Orchestrators' should not be disclosed. This will also generally hold true for cloud providers, as the *efficient* resource orchestration is a company secret. Note that the support for passing non-fully decomposed Network Functions is not only beneficial in multi-provider scenarios, but can be of use for

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

appropriately dimensioning NF-FGs. Given e.g. a software-based firewall, the throughput will heavily depend on the node's hardware configuration, on which it is instantiated. While the choice of the type of firewall solution should be made by the Service Layer, also generating appropriate SLAs, KPI requirements and observability points, the Orchestration Layer may determine the best choice of hardware configurations in its own right.

In short, we note the following *requirements for the decomposition process* in UNIFY:

- The Service Layer SHOULD take information pertaining to the type of hardware into account by excluding any decomposition options, which cannot possibly be supported.
- Depending on the granularity of available resource information and the type of function, the Service Layer MAY restrict the number of potential implementation templates, to allow for an efficient orchestration process.
- Any decomposition decisions that introduce additional functionality on the Service Layer, MUST be made by the Service Layer before passing the *abstract* NF-FG to the Orchestrator.

To summarize, in the UNIFY model based service decomposition instead of making decomposition decisions upon requesting a service instance at Service Layer and allocating resource at the Orchestration level, we choose a more flexible way. The decomposition rules are given from the Service Layer, where multiple options may be present to realize a Service Graph. No decision is made at Service Layer, as it will give "abstract" NFs in the NF-FG to the Orchestrator. The Orchestrator knows the theoretically possible decompositions of a Service Graph from the Service Layer via the NF-IB. The Orchestrator will decide how to actually decompose and where to run the components, based on available resources.

An example decomposition rule-set for an imaginary "forest service" can be seen in Figure 6.19. The dashed arrows represent the possible decomposition steps. Such a decomposition database could be stored in the NF-IB.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

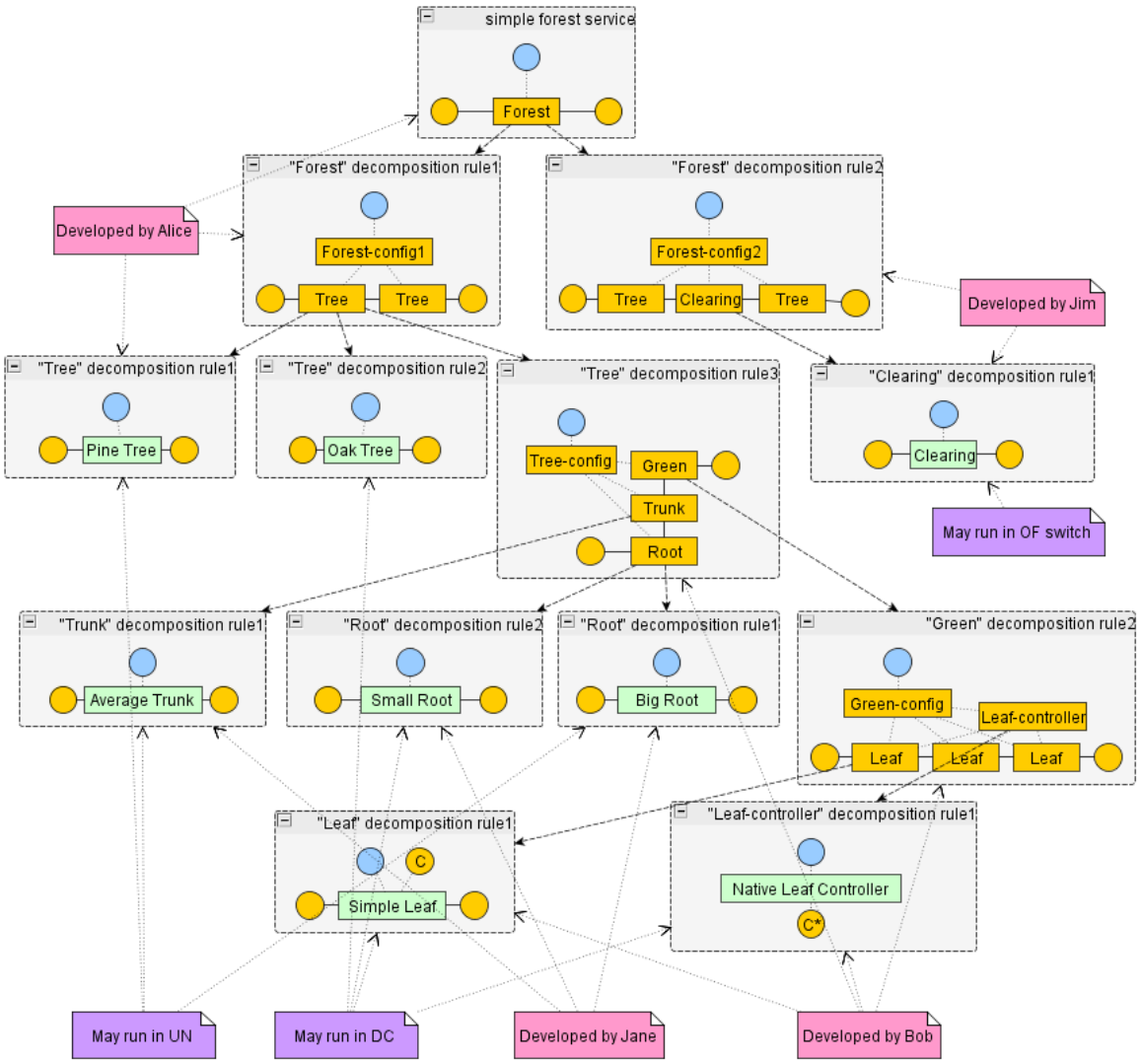

Figure 6.19: Model based service decomposition example

6.6.2 ControlApp-driven decomposition vs. VNF scaling

As introduced in Section 2, a deployed service might involve a resource control function (or ControlApp or CtrlApp) which is able to dynamically change requested services. While decomposition can occur as a static process part of the resource orchestration, it can also be a result of the interaction with the ControlApp. So, ControlApp based service decomposition is a second option to implement UNIFY(ed) service decomposition, which can be seen as a black-box service decomposition, since the logic performing the decomposition is internal to the CtrlApp and is not visible from outside. From that perspective, the task of decomposing a VNF requires similar functionality as the task of scaling a VNF once it is running (see Section 6.7.2 for details). This section will further detail similarities and give a first indication on how this decomposition process might be further formalized.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

During UNIFY(ed) decomposition a VNF *type* is given together with requirements *R* and the expected output is a partial NF-FG that can take the place of the VNF *type* in the original NF-FG. The decomposition function could be written as:

Partial NF-FG = *decompose*(VNF *type*, Requirements *R*)

The same output is expected in the case of scaling a VNF, a partial NF-FG representing a VNF implementation, capable of dealing with the current or expected future. However, the input to a scaling function is a bit different. Scaling applies to a particular VNF *implementation* with the functionality of a VNF type, of which there may be many per VNF type. The requirements *R* are still involved, and a change of these requirements might be the reason the scaling function is invoked. For scaling there is also the addition parameter of internal VNF state *S* representing VNF state specific to an implementation, such as number of rules inserted in a firewall for example. Changes in *S* may also be a reason to call the scaling function, e.g. in order to automatically scale the VNF once there is no more space to insert additional firewall rules. The scaling function also has a third parameter, measurement triggers *M* that could for example indicate that the VNF processing latency has exceeded some threshold and the VNF needs to scale to reduce latency. With these parameters in mind the scaling function could be described as:

Partial NF-FG = *scale*(VNF *implementation*, Requirements ***R***, State ***S***, Measurements ***M***)

Ideally, the output of *decompose*(VNF *type*, ***R***) and *scale*(VNF *implementation*, ***R***, ***Ø***, ***Ø***) should be identical for any ***R***, otherwise a newly decomposed partial NF-FG may be instantiated and then immediately need to be modified since the *decompose()* and *scale()* functions disagree. One way of solving this is to merge the *decompose*() and *scale*() functions to a single one, with ***S*** and ***M*** set to ***Ø*** in the initial decomposition step.

A VNF type might be represented by multiple VNF implementations, i.e. the VNF type *Firewall* could have several implementations, $Firewall_A$, $Firewall_B$, etc, therefore one call to *decompose*(Firewall, ***R***) translates into multiple *scale*($Firewall_A$, ***R***, ***Ø***, ***Ø***), one for each of the different implementations. This results in multiple partial NF-FGs, which may need additional decomposition if the partial NF-FGs also contain VNF *type*s. Based on some criteria one of the resulting partial NF-FGs has to be selected for instantiation, e.g. depending on the amount of resources required.

Calling VNF CtrlApps

As the scaling function requires knowledge of the internal state ***S*** a good place to implement the *scale*() function is in a VNF CtrlApp for a specific VNF implementation, since that is where internal state of the VNF data plane components are known. This means that if a *decompose*() call translates to *scale*() calls, the scale() function in VNF CtrlApps must be callable somehow.

One way to solve this would be to have instances of all the VNF CtrlApp implementations always running, isolated without any connections to corresponding VNF data plane

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

components, and expose an RPC interface allowing the Orchestrator to call their *scale*() functions during the decomposition process. Another solution could be to instantiate relevant VNF CtrlApps during each decomposition call, trading higher latency for less resource usage.

A more lightweight approach could be to force the VNF developers to export their *scale*() functions in some way that could be integrated into the Orchestrator, e.g. as Java classes, Python scripts, or simple compiled executables. Those could then be placed in the NF-IB, and obtained by the Orchestrator during the decomposition process, the Orchestrator then executes them to obtain the partial NF-FGs.

Control-App-driven decomposition example

Rather than fully formalizing the decomposition process already, in this subsection we describe the dynamic decomposition as can be induced by a ControlApp as illustrated in Figure 6.21. Figure 6.20 gives a detailed overview on the sequence of required interactions between different entities, and Figure 6.22 depicts the finally resulting decomposed NF-FG.

It is important to note here, that the NF-FG shown in Figure 6.14 is purely a theoretical NF-FG, as it won't be available in this final decomposed form neither in any of the UNIFY architecture components, nor at any reference points or interfaces. This figure helps the reader to see the whole final picture of the decomposed service and understand the process, however only NF-FGs A1-3, B1-3 and C1-2 of Figure 6.22 will appear really. Figure 6.14 is the superposition of the previous NF-FGs.

In the example we assume a hierarchy of 2 domains (conform the principle of recursiveness documented in D2.1 Section 6.3), where the Orchestrator of domain of level 1 (ORCH1) can delegate sub-graphs of the NF-FG towards the Orchestrator of the sub-domain at level 2 (ORCH2). In the next overview, we will shortly discuss the most essential steps of the process for an imaginary Gold Forest service request (top of Figure 6.21.). A detailed sequence diagram is depicted in Figure 6.20, we will walk through the main steps according to the sequence diagram. (Please look at in Figure 6.21 and Figure 6.22. parallel while following the description below.)

1. The User requests for a Gold Forest service between two SAPs to the Service Layer.

2. The Service Layer uses a set of rules/templates in order to convert the service request to a corresponding NF-FG and passes the request to the resource orchestration of the top-level domain (ORCH1).

3. ORCH1 decides to decompose the Forest NF to an NF-FG with the same scope (i.e., connection points) involving a (non-atomic) Forest ControlApp which interacts with ORCH1 using the Or-Cf interface (purple). In addition, it delegates the further resource Orchestrator of this Forest ControlApp to ORCH2.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

4. ORCH2 instantiates the Forest ControlApp as David's Forest CtrlApp on the infrastructure.

a. The corresponding Forest-CtrlApp instance is started and returns its interfaces to ORCH1. This results into a cascade of Interface announcements (see middle of Figure 6.22) until it reaches the user.

5. As the instantiated CtrlApp is now in charge of further decomposition actions, it can further decompose the NF-FG within its scope (between the SAPX and SAPY and interface to CtrlApp SAP2). This results into NF-FG B1 which decomposes to a Tree Pair which is directed towards ORCH1 over its Cf-Or interface.

6. Based on the received NF-FG B1, ORCH1 can decompose the resulting Tree Pair into a Pine Tree and Oak tree which can be further decomposed/instantiated by ORCH2 (NF-FG B2).

7. Based on the received NF-FG B2, ORCH2 instantiates these into David's PineTree and David's OakTree CtrlApp (NF-FG B3), where the last one has a Or-Cf interface with ORCH2.

8. The created instances return the resulting interfaces to ORCH2.

9. David's OakTree CtrlApp generates a new decomposition, resulting into Root, Trunk and Tree NFs communicated to ORCH2 (NF-FG C1).

10. ORCH2 instantiates VNFs for the received decomposition.

11. Interfaces and statuses of the newly instantiated components are propagated and consolidated such that finally the user is informed that the Forest service is up and running.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

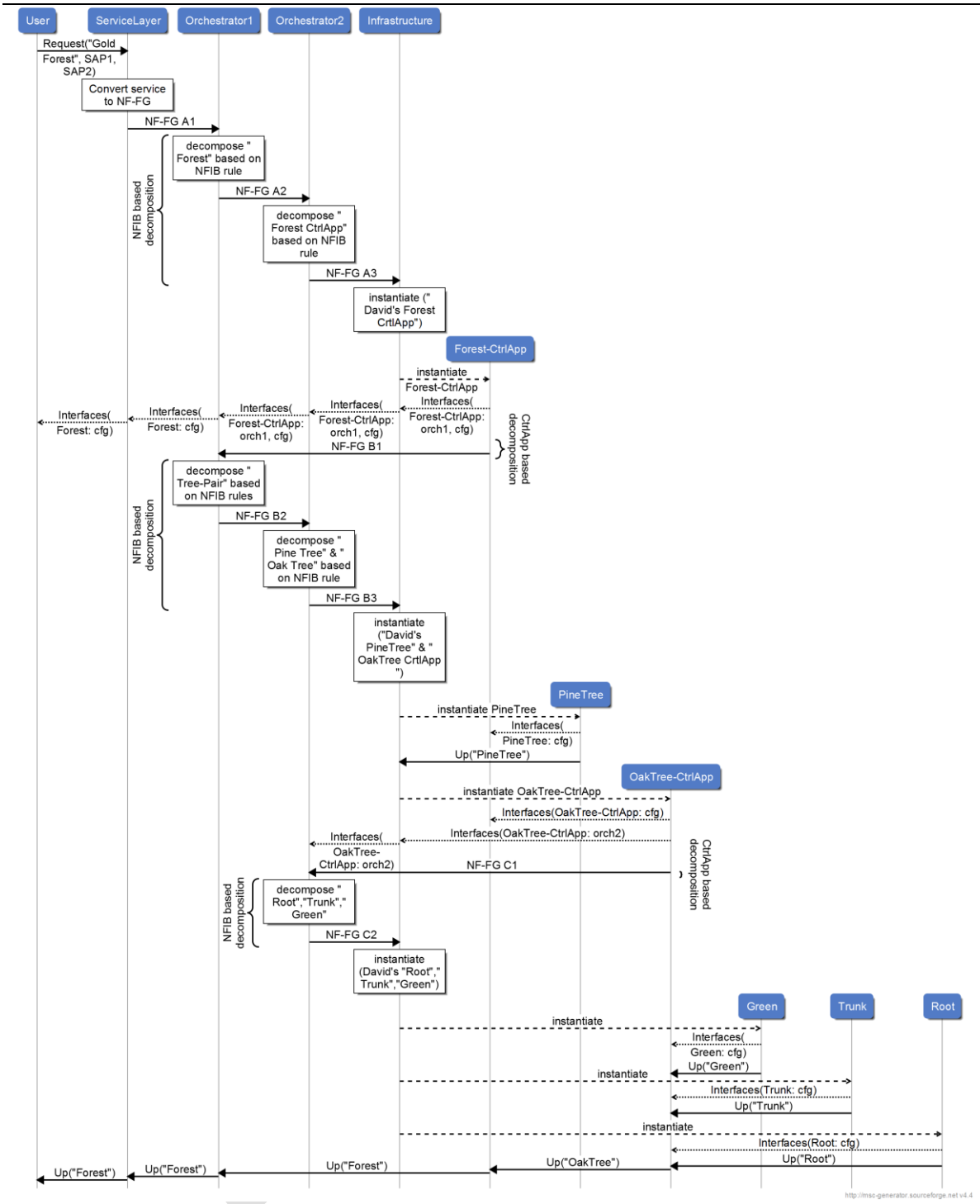

Figure 6.20: CtrlApp based decomposition example, sequence and messages

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

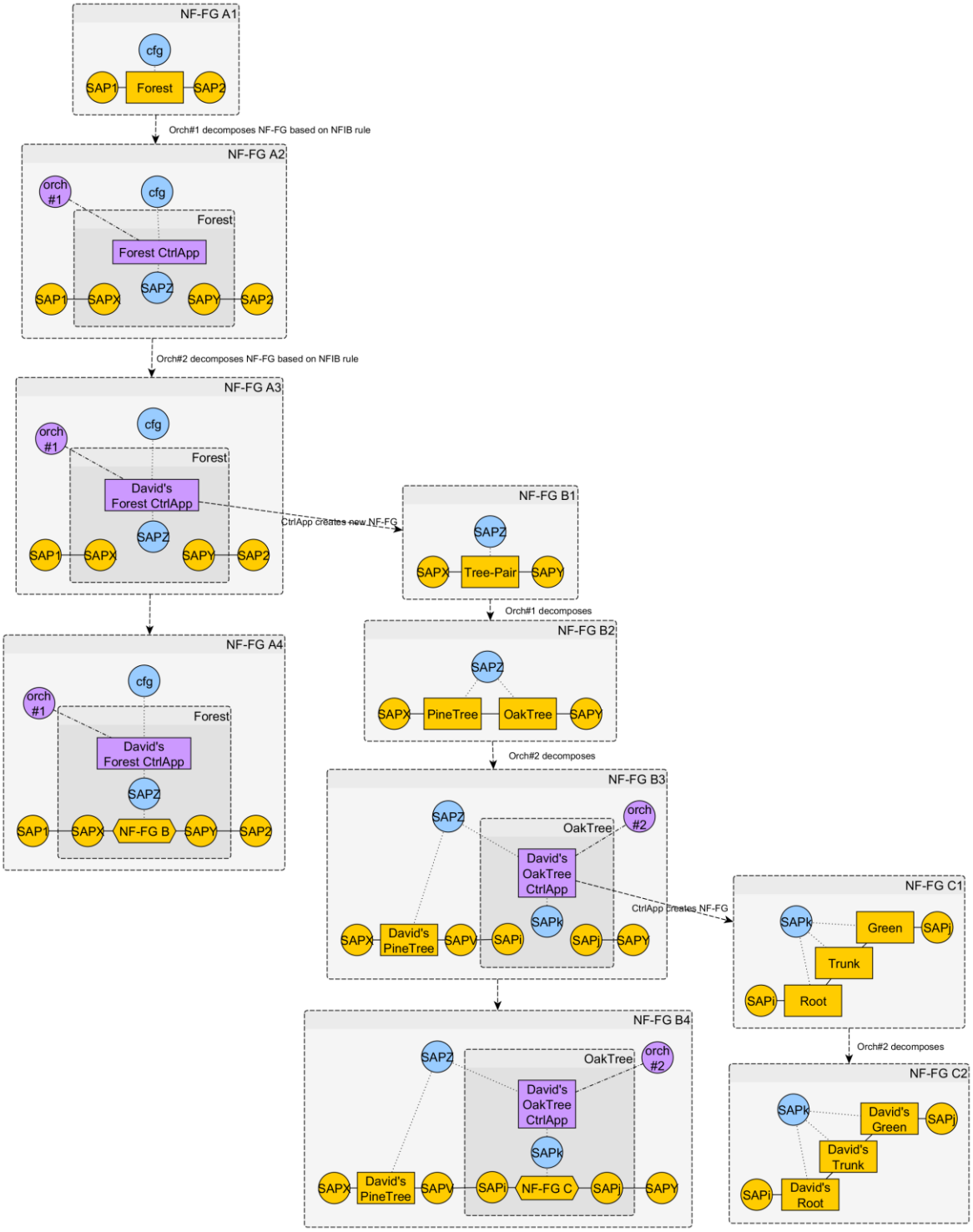

Figure 6.21: CtrlApp based decomposition example, NF-FGs

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

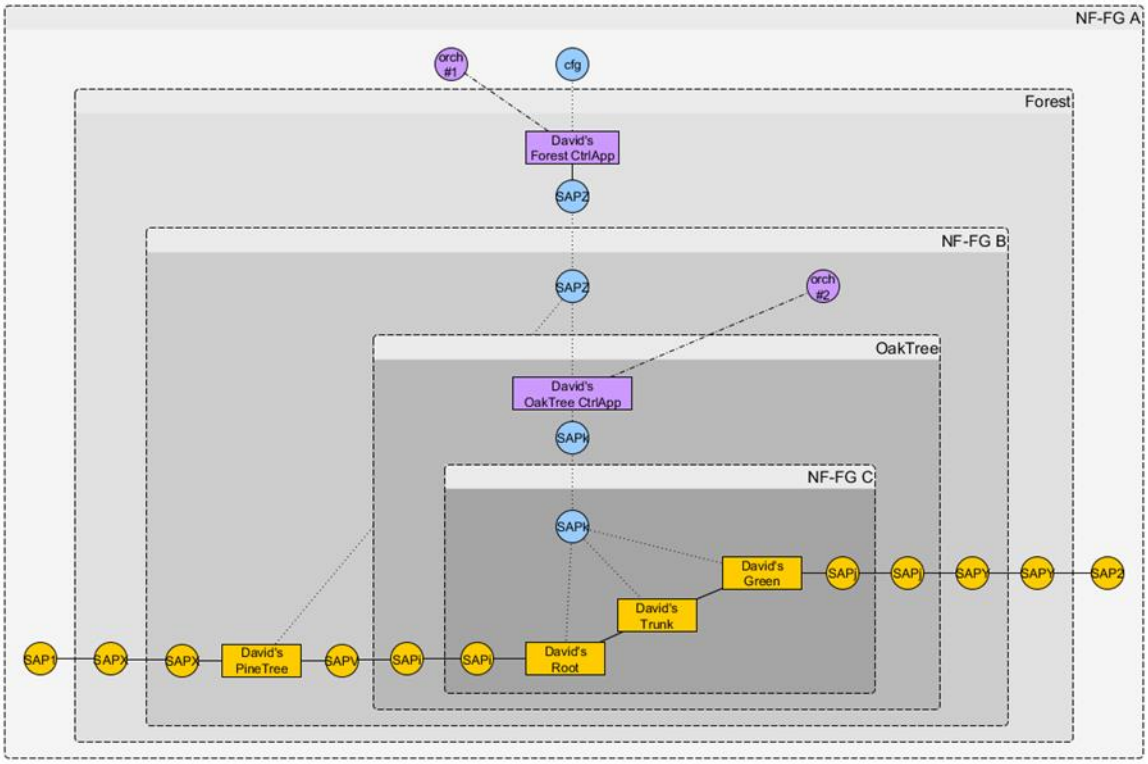

Figure 6.22: CtrlApp based decomposition example, final theoretical decomposed NF-FG

6.6.3 Decomposition of KQI, KPI and resource parameters and decomposition types

In addition to the decomposition of pure NFs or parts of the NF-FG, decomposition might also be needed in order to accommodate quality/performance or resource parameters. Again, the purpose of this section is mainly to introduce related concepts, issues and potential further working directions.

Service related parameters have to be decomposed as well. For example in a high level form of a NF-FG there is an abstract "Firewall NF" component to serve 2 users, while the VM implementation to which it can be decomposed is able to serve 10 MB/s. In this example there is a need to map from "users" to "MB/s". Such parameter decomposition rules can be very simple or complex ones, as well as they may be generic or implementation specific. We envision that such parameter decomposition rules may be given in the NF-IB. In this section we elaborate on these possible rules.

Resource/performance parameters will be given for each NF in the NF-IB. Decomposition and/or conversion rules of parameters will be given in the NF-IB.

As a first approach, we introduce the following set of rule types:

- Type A rule: function decomposition without parameter conversion
  - Example: Forest(X tourists) → PineTree(X tourists) + OakTree(X tourists)
- Type B rule: parameter conversion without function decomposition

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- Example: PineTree(X tourists) → PineTree(1+2X walk/hour)

- Type AB rule: function decomposition with parameter conversion

- PineTree(X tourists) → David'sPineTree VM (1+2X walk/hour)

- Type C rule: instantiation rule

- Example: David'sPineTree VM Needs "small" (1 VCPU, 128 MB RAM, 0 GB HDD) execution environment, Capacity: 50 walks/hour

The decomposition/orchestration can be complex even by using rules only from the types above. Therefore we differentiate the scenarios in the following section and discuss them one by one.

The scenarios can be classified according to three main dimensions: levels of decomposition, dynamicity and number of choices available. First we present the simplest scenario and then the variations on it along each of the three dimensions.

6.6.4 Decomposition example scenarios

6.6.4.1 **Single choice scenario**

Assumptions: single level, static, single choice

In this scenario there is one possible matching for each service parameter decomposition, i.e. there must not exist multiple matching rules. For example, in case of "C" type rules, the capacity of realizations must be non-overlapping, like:

- David'sPineTree VM Needs "small" (1 VCPU, 128 MB RAM, 0 GB HDD) execution environment, Capacity: <=50 walks/hour

- David'sPineTree VM Needs "big" (2 VCPU, 215 MB RAM, 0 GB HDD) execution environment, Capacity: >50, <=100 walks/hour

This scenario makes service parameter decomposition easy, however requires strict coordination of decomposition rules.

6.6.4.2 **Multi-choice scenario**

Assumptions: single level, static, multi choice

Compared to the first scenario, here there are multiple matching rules for a decomposition step, these multiple rules may differ only in the NF, in the parameters, or both.

Example1:

- PineTree(X tourists) → David'sPineTree VM (X tourists)

- PineTree(X tourists) → Joe'sPineTree VM (X tourists)

Example2:

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- David'sPineTree VM Needs "small" (1 VCPU, 128 MB RAM, 0 GB HDD) execution environment, Capacity: <=50 walks/hour
- David'sPineTree VM Needs "big" (2 VCPU, 215 MB RAM, 0 GB HDD) execution environment, Capacity: <=100 walks/hour

This scenario makes service decomposition and service parameter conversion part of the optimization process. The optimization process is to be discussed later in the project.

6.6.4.3 **Multi-level decomposition**

Assumptions: multi-level, static, single choice

Compared to previous scenarios, in this case an "atomic" NF or a NF "instance" of a higher Orchestration Layer will be seen as an abstract NF by a lower layer Orchestrator and will be further decomposed. This will raise the questions what NF parameters are allowed at various interfaces of the UNIFY architecture. Without this multi-level behavior the NF parameters below the Orchestrator could be limited to the narrow-waist CPU, memory and storage parameters. However, with multi-level orchestration more abstract parameters and requirements (e.g. number of users served) are allowed to be passed to lower layer orchestration.

6.6.4.4 **Dynamic decomposition**

Assumptions: single level, dynamic, single choice

In this scenario the decomposition is made by a ControlApp (described in more detail in Section 6.6.2), extended with the various parameters described above.

A real life scenario can contain any combination of the scenarios listed, which implies complexity of the implementation. These aspects will be further worked on by the project.

6.7 Orchestration process

The Orchestration Layer has as its input an NF-FG and is responsible for mapping the resource requirements described in the NF-FG to the available resources in the Infrastructure Layer, giving as its output a decision where to instantiate certain Network Functions, how to connect them to each other and the Service Access Points included in the NF-FG. The decision should not only fulfil the requirements posed by the input NF-FG but also be as close to an optimal placement as possible, as defined by certain goals. The Orchestration Layer also takes part in the life-cycle of a deployed NF-FG, dynamically reacting to changes in both the infrastructure and in the requirements of the NF-FG to either fulfil requirements updated by the client or updated by the system itself in order to automatically adapt to the demand required by a particular deployed NF-FG. In this section we discuss some of the design options we have when designing this process, how it can be built to scale to a large scale network of resources, how individual NFs and NF-FGs can scale, and what interactions with other components can be expected.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

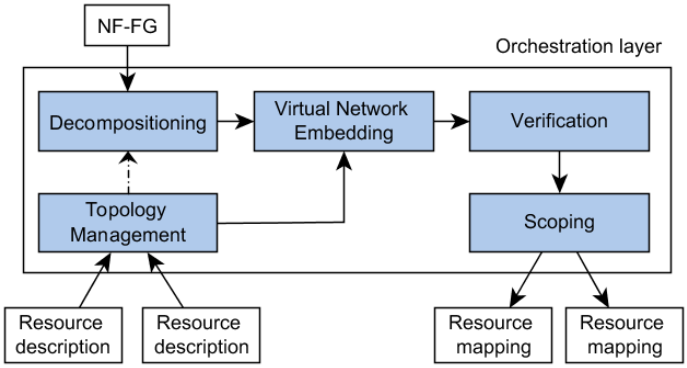

Figure 6.23: High level view of the orchestration process.

A simplified and high level view of the functions in the Orchestration Layer can be seen in Figure 6.23. The basic flow is that an incoming NF-FG is decomposed into appropriate VNFs, potentially with input from the topology manager in order to take into account resource restrictions, as described in Section 6.1, 6.2 and 6.5. The decomposed NF-FG is then handed over to the Virtual Network Embedding module responsible for find a near-optimal way to map the resource requirements to the topology. If a mapping is found the verification process verifies that the mapping is correct before handing it over to the scoping process. The scoping process partitions the mapping into appropriate pieces and sends them to the lower layers to be instantiated.

The verification processes is mainly developed as part of work package 4, and may verify several different aspects of the NF-FG as it passes through the overall orchestration process:

- The complexity of solving the Virtual Network Embedding Problem increases with the level of information provided (see Section 4.6). To simplify the algorithms employed for orchestration and reduce their runtime they may utilize a limited set of core information. The verification process could as a second step verify that the found embedding actually is a valid one. If the verification fails, the VNE process should be restarted with additional input to produce a new embedding.

- As the orchestration process naturally happens in a distributed setting, resource information may change while the orchestration algorithm is computing a solution. The verification step may be necessary to check that the resources used by the embedding are still available afterwards.

- The verification functionality should check the correctness of the outcome of the decomposition process to ensure that the decomposed NF-FG still fulfil all the requirements (as SLAs, KPIs etc.) that were defined for the initial NF-FG. This step may occur before the VNE process.

- Before passing the obtained NF-FG to the scoping process, certain verification routines should be executed to check for the topological correctness of the defined NF-FG.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

These checks might verify reachability constraints e.g. that end hosts can actually communicate or that no forwarding loops exist in the NF-FG.

The scoping processes has two main responsibilities, partitioning the embedded NF-FG into appropriate pieces for the lower layer orchestrators or controllers, and allocate necessary cross-domain handles to allow the partitions to be stitched together in the Infrastructure Layer. To partition the embedded NF-FG requires an understanding of where in the NF-FG the domain boundaries are, how the NF-FG can be split and what information is necessary to bridge them, in a lower layer. This could for example be to split cross-domain links into two intra-domain links and translate KQIs/KPIs from the single link to be applicable to two links. Part of this split includes either directly allocating necessary matching traffic tags (e.g. VLAN-, VXLAN-, or MPLS tags or wider flow-space definitions) so that traffic from one domain can be identified in the other domain, or allocating temporary handles that lower layer orchestrators or controllers can use to negotiate traffic tags between themselves.

The basic flow depicted in Figure 6.23 hides all the complexity that will be introduced in order to provide scalability, e.g. it is likely that several of these processes are implemented in a hierarchical fashion, with topology management, decomposition, and virtual network embedding taking place multiple times. Additional complexity is added by the need to handle dynamic processes described in Section 6.7.3, to handle e.g. automated scaling, arriving at more detailed processes is not done in this section, we aim at describing the options we have and what the detailed processes should handle.

In addition, a preliminary prototyping framework has been established to support the development of all highlighted modules. The benefits of this framework and proof-of-concept implementations were demonstrated in [Csoma2014a], [Csoma2014b].

6.7.1 Orchestration scalability

To be able to orchestrate NF-FGs over large scale networks, scalability features in the Orchestration Layer needs to be supported. We have three major mechanisms for improve the scalability of the orchestration system. These mechanisms each individually improve the scalability of the system but they can also be combined.

The first attempts to reduce the amount of details and options for placements that the Orchestration Layer needs to manage through network abstractions, by hiding parts of the topology and details about its components in lower layers in order to reduce the size of the topology graph that has to be taken into account by the virtual network embedding algorithms responsible for finding a placement within the required parameters. Reducing the amount of packet forwarding nodes, compute/storage nodes and links connecting them can have a big impact on the performance of VNE algorithms since the time and memory requirements typically scale in proportion to the complexity of the graphs they are operating on.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The second mechanism is parallelized orchestration, in which we attempt to split the problem of virtual network embedding into smaller problems *within one logical Orchestrator* and distribute the virtual network embedding calculations among a number of peers, either running as multiple threads or processes in a single machine or among multiple machines that form an orchestration cluster.

The third mechanism is hierarchical orchestration. Here, like in parallelized orchestration, the problem is divided into smaller pieces and delegated to multiple Orchestrators on a lower level, letting each of them handle a subset of the problem on a smaller topology, the network domain that they are responsible for.

A combination of these could be to have a hierarchical orchestration setup, where each Orchestrator in the hierarchy is implemented as a parallelized/distributed Orchestrator. Each level in the hierarchy could in addition calculate an appropriate network abstraction which it communicates to higher layers.

6.7.1.1 Network abstractions

When discussing *network abstractions* here what we refer to can also be called *virtual topologies, abstract topologies, network maps,* etc. These represent different ways of hiding information about the actual topology and present a simplified or reduced view of the actual topology to whoever is the receiver. We do not refer to other concepts sharing the same name, such as network abstraction layers as used in e.g. the OSI 7-layer model. Figure 6.24 shows a scenario where two physical networks controlled by two controllers which export their topology to a single logical Orchestrator. Since a controller is only aware of the topology under their own control, the Orchestrator has to construct the full topology (seen in the rectangle in the right side of the figure) in some way. When communicating this topology to a higher layer the control- and/or Orchestration Layer can hide parts of the topology in order to reduce the complexity of the graph, hide irrelevant or confidential details, and simplify calculations at a higher layer. In terms of performance, this simplification in most cases is a trade-off between optimality of the traffic engineering / virtual network embedding and the computational cost of the associated algorithms. Here we will go through some network abstraction techniques and discuss their pros and cons.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

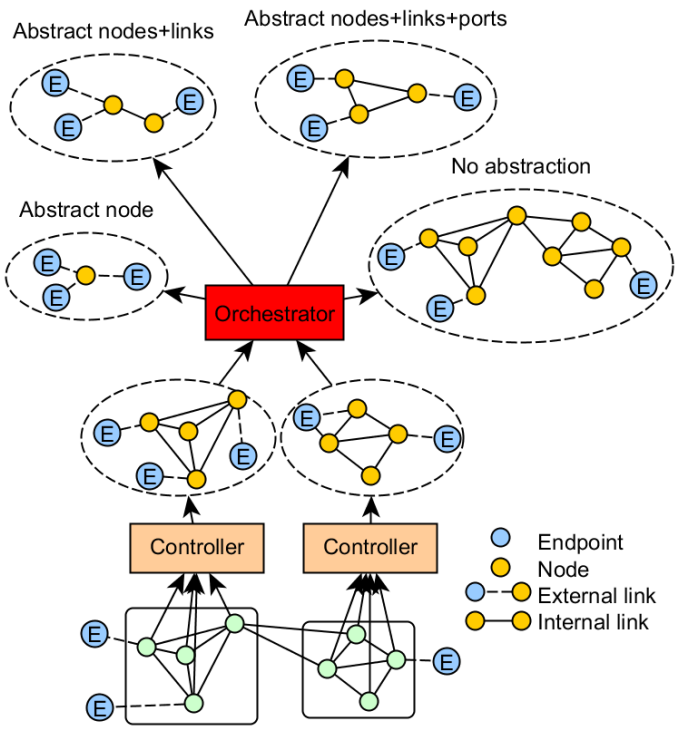

Figure 6.24: Various modes of abstracting the topology to higher layers. Blue 'E's represent external nodes whereas yellow circles are representations of nodes in the Orchestrator topology.

Abstract node ("Big Switch")

One of the extreme options the Orchestrator has is to reduce the entire topology to a single abstract node (seen at the leftmost in Figure 6.24). In this case all internal nodes and internal links are hidden within a single node, external links and end-points which represent the connections to other networks or nodes remain the same.

The abstract node view efficiently reduces the number of components in the topology, in the example in Figure 6.24 the number of components is reduced from a total of 26 objects (links, nodes, etc.) to 7 objects in the abstract topology. The "efficiency" of the reduction is proportional to the number of internal nodes and links vs external links and endpoints, the larger the internal network the more is gained.

However, with the reduced view we lose all information about the internal network, for example we lose any geographical information about nodes, and all information about internal connectivity and performance. For links these include bandwidth/latency/jitter parameters and for nodes this may be for example current load. We may have link parameters on the external links, but it is difficult to say what they actually represent. For example if two endpoints, connected with two external links to the abstract node, each external link with a bandwidth of 1 Gb/s, this does not necessarily mean we can connect the two endpoints with 1 Gb/s, since internal links may not have sufficient bandwidth. This is an issue unless the internal network is constructed in such a way that it is non-blocking even when all external links are saturated. Another issue with this extreme abstraction is

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

that network partitions cannot be represented; it looks like all endpoints can connect to each other even if the internal network is partitioned.

Some of these issues can be alleviated by re-introducing some of the information we removed in the abstraction and adding it as node related parameters on the abstract node itself, this could for example include a "Connectivity matrix" showing e.g. maximum bandwidth between all endpoints. An alternative to explicitly carrying information in nodes could be an interface for asking certain questions about specific endpoints, e.g. "C*an endpoint A and B be connected, and with what bandwidth?*". Such an interface allows the management of the topology to remain simple at the cost of additional latency in the processes using the information.

Abstract nodes and links

A less extreme option is to allow the abstract topology to contain multiple abstract nodes, connected with abstract links to represent e.g. bottlenecks in the topology. The abstract nodes can be used to group e.g. endpoints with fairly uniform traffic parameters between each other (i.e. similar bandwidth, latency, etc.) by creating an abstract node and attaching a group to it.

This approach reduces the size of the network topology while preserving some information that is useful for traffic engineering, and can also represent partitions in the network. As nodes that are geographically close are more likely to get grouped together it can keep at least some of the geographical information in the topology. This approach also allows a trade-off between traffic engineering quality and topology size, depending on how the abstract topology is constructed and with which primary goal the end result may be range from the single abstract node case to a full topology. Automation of the abstract topology construction may be possible to formulate as a graph clustering problem [Schaeffer2007]. If this is possible existing algorithms may be applied automatically and periodically at the Orchestrator(s) in order to generate a suitable abstract topology. If the "aggressiveness" of clustering can be tuned, this could perhaps be used as a parameter controlling the trade-off between traffic engineering and topology complexity. Examples of clustering approaches can be found in e.g. [Beck2013] and [Fuerst2013].

Abstract nodes, links, and ports

This approach could be seen as a different way of implementing the "single abstract node with connectivity matrices", without the matrices. Recall that the matrices represent connectivity, bandwidth, latency, etc, between endpoints in the graph which are stored as node parameters on the single abstract node. These matrices may also be expressed as links in a graph, generating something that could range from a full mesh between endpoints to a reduced mesh in case of network partitioning, nodes which cannot reach each other, etc. However, as the number of links would tend to scale as N^2 with N either being the endpoints (or nodes connecting multiple endpoints) this approach does not scale well. As an alternative to matrices it may simplify the implementation of algorithms

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

operating on the graph, by reducing the amount of information stored in nodes which may be an issue depending on how the topology is stored, updated etc.

No abstraction - full topology

Depending on how the algorithms that utilize the topology operate, it may not be worthwhile to provide an abstract view at all. Some algorithms need to iterate through the whole graph and thus scale badly as the size of the topology grows. Other algorithms may depend more on the type of topology (random, tree, circular, etc) and e.g. the number of geodesics (number of shortest paths in the network). For example if the underlying topology is a tree topology there is only one shortest path from a leaf to the root, if the typical use of the graph is to calculate the bandwidth from a leaf to the root it may not be worth to try to simplify the graph.

6.7.1.2 Parallelized/distributed orchestration

With parallelized orchestration a single logical Orchestrator may be implemented over multiple CPUs and the whole or parts of the VNE algorithms calculations distributed among these CPUs either in a SMP/multi-core system or over multiple servers. The VNE problem could potentially be parallelized/distributed using two general approaches, illustrated in Figure 6.25:

1. Dividing the incoming NF-FG into partitions and calculating them in parallel, or calculate the original NF-FG in parallel in isolated or restricted parts of the shared topology (top of the figure).

2. Parallelization of parts of the VNE algorithm itself (bottom of the figure). Some operations within the VNE algorithms can potentially be executed in parallel e.g. calculating a number of constrained paths between certain points or evaluating different embeddings.

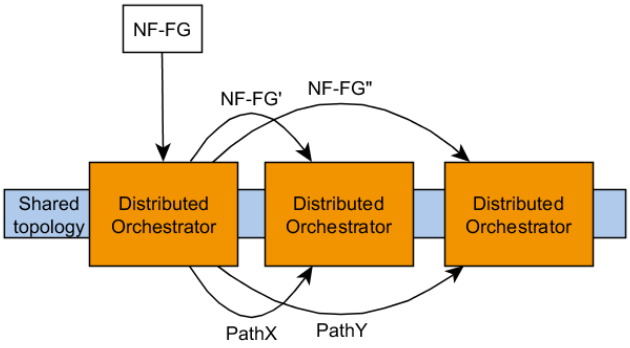

Figure 6.25: Distributed Orchestrators with a shared topology.

How much the second parallelization approach , distribution of internal calculations, can aid scalability depends heavily on the VNE algorithm itself, if none of the steps in the algorithm can be carried out in parallel there is nothing to gain in terms of lowering the time it takes to perform a single embedding. This is illustrated by “Amdahl’s Law” which estimates how much speedup gain *S* one can expect depending on the proportion *P* of the

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

program that can benefit from parallelization, for *N* processors the gain can be estimated as:

$$S(N) = \frac{1}{(1-P) + \frac{N}{P}}$$

That is, if P=0.5, i.e. 50% of the execution of the program can be parallelized, we can expect at most a factor two of speedup gain, regardless of how many CPUs we dedicate to the 50% of the program that can be parallelized.

However, dedicating multiple CPUs to an Orchestrator even if the VNE algorithm itself is not very parallelizable can still be useful in order to allow multiple NF-FG embeddings to be carried out in concurrently without having to wait for other embeddings to finish before starting with a second. How useful this is in practice depends on how likely different embeddings are to affect each other. If for example two embeddings are calculated concurrently without synchronizing used resources they may end up assigned to the same resource in the topology, even though that resource can only accommodate one of the embeddings. This requires one of the embeddings to be calculated again once it has been rejected since the other one used the critical resource first. That issue can be managed by for example adding locks to the topology data structure used to calculate the embedding, in order to communicate between concurrent processes that a particular resource may already be occupied. This in turn adds latency to the calculations as locks has to be synchronized between concurrent processes.

6.7.1.3 Hierarchical orchestration

Hierarchical orchestration differs from the distributed/parallelized approach by not having a shared topology view between the Orchestrators; instead the topology at each level is limited to a subset of the total topology (depicted in Figure 6.26). The limited topology may come from a practical placement of the Orchestrators based on e.g. geography or by placing Orchestrators based on knowledge of the total topology, e.g. by setting a maximum number of nodes in an Orchestrator's topology and allocating Orchestrators based on this limit.

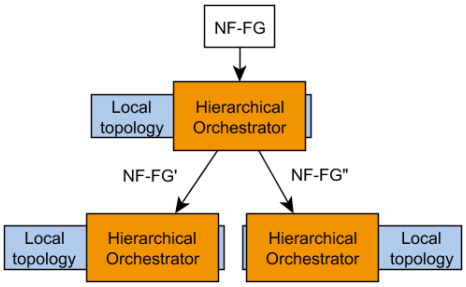

Figure 6.26: Hierarchical Orchestrators.

The hierarchical orchestration process is of specific interest within the UNIFY project, as ISP networks can easily be considered to be hierarchical (see Annex 2): Assuming that NFV

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

functionality is placed at POPs, each of these POPs can be considered an independent orchestration domain such that the global orchestrator of the ISP just ensures appropriate connectivity and passes the orchestration of Network Functions towards one of the POP orchestrators.

This approach is taken in [Beck2013] which proposes a hierarchical placement of Orchestrators based on clustering of the total network topology to create a tree topology of Orchestrators. A short summary of how this proposal operates:

1. When an Orchestrator receives a NF-FG it calculates a heuristic based on the number of nodes, required bandwidth and other parameters of the NF-FG, and compares it to the same heuristic calculated on the topology in its child Orchestrators in the hierarchy.

2. It then forwards the incoming NF-FG to a child whose heuristic is larger to the one of the NF-FG itself, starting with the child closest to the NF-FG requirements.

3. When the recursion stops, i.e. there are no more potential children, the Orchestrator tries to perform a VNE embedding, and if that fails the parent one step above in the hierarchy continues to the next potential child.

4. If none of the children that could potentially embed the NF-FG are able to do so, the Orchestrator tries to do it using the topology on its own level, if that also fails it again backs up until all an embedding is found or the request fails.

In this proposal the NF-FG itself is not partitioned, the original request is used throughout the hierarchy, in terms of Figure 6.26, NF-FG' and NF-FG'' are identical to the original NF-FG. The authors additionally propose a locking scheme which in conjunction with a coordination protocol allows the Orchestrators to process multiple NF-FG requests in parallel.

One potential extension of this proposal could be to allow Orchestrators to partition an NF-FG into e.g. two new requests, NF-FG' and NF-FG'', before forwarding the request to two of its children, given that the connectivity requirements between the two partitions are fulfilled by the connectivity between the topologies managed by the child Orchestrators.

6.7.2 NF and NF-FG Scaling

The possibility to dynamically scale Network Functions at run-time in an automated fashion is one of the main advantages offered by the VNF approach, providing both better resource utilization and better service at a lower cost. There are multiple reasons for initiating a scaling procedure; a user could request higher capacity ahead of time in order to deal with a known increase of demand in the future, the procedure could be triggered by increased demand on the fly, or the procedure could be triggered by changes in the network / compute substrate itself, for example when additional hardware is added or is taken off-line for administrative purposes.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Scaling a VNF can be done in many ways, which one is appropriate in a particular case depends heavily on the precise function provided by the NF, for example; the type of traffic it operates on and at which layer in the networking stack, requirements on synchronization, requirements on traffic during the scaling event, dependencies on other NFs, ability to run multi-threaded, as well as the operators ability to modify the NF to better support scaling. Here we first try to break down the types of VNFs and the main options we have when increasing/decreasing the performance of a VNF.

6.7.2.1 VNF taxonomy

Table 6.7: VNF taxonomy, properties of a VNF implementation

Sensitivity	State/Flow	Support	Flow type	Consistency	Modification
Packet loss	None	None	Non-divisible	None	None
Packet reordering	Individual flow	Resource aware	Layer 1-4	Low rate	Low rate
Service interruption	Multiple flow	Scaling protocol	Layer 4-7	High rate	High rate
Progressive move	All flows	--	--	--	--

Various scaling properties for a VNF implementation can be seen in Table XX, where six different properties/attributes have been defined, some of them are mutually exclusive whereas for others multiple options may apply to a single implementation:

Sensitivity represents flags that characterize the VNF's tolerance of effects caused by the scaling process itself; a sensitive VNF requires a more complicated control plane to ensure that these guarantees are met.

- *Packet loss*, the VNF cannot accept any packet losses during scaling
- *Packet reordering*, the VNF cannot accept any re-ordering of packets during scaling
- *Service interruption*, the VNF cannot accept any interruption during scaling
- *Progressive move*, whether or not state has to be immediately transferred to new instances

State/Flow represents the type of the state stored in the VNF; here multiple types of state may be present in a single VNF, this affects the granularity and speed at which the control plane may move flows and their associated state from one VNF instance to another.

- *None*, no state associated to individual flows is stored; however there may still be VNF configuration parameters.
- *Individual flow*, there is a one-to-one association between a flow and a state block
  - *E.g. a packet counter in a firewall for a single specific flow, 10.0.0.0/24*
- *Multiple flow*, there is a many-to-one association between flows and a state block
  - *E.g. a packet counter in a firewall for multiple flows, 10.0.0.0/8*
- *All flows*, there is a all-to-one association between all flows and a single state block
  - *E.g. a packet counter in a firewall counting all packets*

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Support indicates what type of scaling support a VNF has, a VNF may support multiple of these. This affects what scaling options we have and what can be guaranteed during flow/state transfer.

- *None*, the VNF has no internal support for scaling
- *Resource aware*, the VNF is able to take advantage of increased local resources
- *Scaling protocol*, the VNF has a protocol to coordinate state transfer and traffic redirection

Flow type indicates the type of traffic flow a VNF is processing, this affects the complexity of distributing traffic among multiple VNF instances.

- *Non-divisible*, the traffic flow cannot be sub-divided into smaller flows.
- *Layer 1-4*, the traffic flow can be sub-divided based on layer 1-4 information (e.g. IP addresses, TCP ports)
- *Layer 4-7*, the traffic flow can be sub-divided based on layer 4-7 information (e.g. HTTP session identifier, SIP session,

Consistency refers to state synchronization requirements between VNF when multiple VNF instances are running simultaneously to distribution load. This affects the requirements on the intra-VNF control network needed to maintain consistency.

- *None*, the VNFs can run independently without sharing/synchronizing state.
- *Low rate*, the VNFs need to share/synchronize state but at a low traffic rate, e.g. no strict requirements on high bandwidth/low latency connections between the VNFs.
- *High rate*, the VNFs need to share/synchronize state at a high traffic rate, e.g. with strict requirements on high bandwidth/low latency connections between the VNFs.

Modification refers to the rate of internal state updates (in respect to the traffic rate) during normal operation of a VNF, this affects the migration of running functions and the requirements on both a intra-VNF control network as well as on the control plane / scaling protocol.

- *None*, the VNF has no internal state affected by traffic.
- *Low rate*, the VNF seldom updates its internal state, e.g. a NAT may update the state only when new sessions are established or when they time out.
- *High rate*, the VNF often updates its internal state, e.g. an NF counting packets per flow is updating its state for each packet.

These parameters greatly affect the options available when scaling a VNF and the complexity of the required scaling solution. For example a VNF with a non-divisible *flow type* cannot be scaled by starting multiple VNF instances since we cannot spread the traffic to more than one instance. A VNF with no requirements on packet loss, re-ordering, and service interruption requires no sophisticated coordination of flow and state transfer.

6.7.2.2 Scaling approaches

We have identifier three major approaches to scaling a VNF; individual scale-up/down by adding/removing resources, scale-out/in by adding/removing instances and redirecting traffic, and dependency scaling where a VNF's performance is depending on another VNF/service which in turn has to be scaled using previous methods or perhaps through

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

improving the link to that VNF/service. These approaches may also be combined, for example by first trying to assign more resources to a VNF and when no more local resources are available, adding additional instances and spreading the load. In Figure 6.27 the initial setup is depicted. A VNF is deployed connected to Service Access Points (SAPs), transferring all data bi-directionally on the link (indicated by the <*,*> notation). A VNF CtrlApp is connected to the VNF in order to control and manage its internal state. The CtrlApp is also connected to the Resource Orchestrator in order to update the initial NF-FG deployment by e.g. adding additional NFs or modifying the traffic forwarding rules on the links.

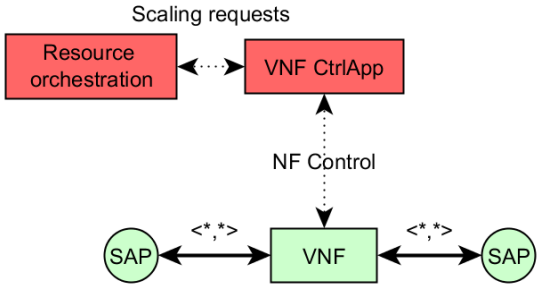

Figure 6.27: Initial setup before any scaling events.

Scale-up/down of individual NFs

The simplest approach may be to scale up rather than scale out, i.e. increasing the performance of a single NF instance rather than adding more instances. One way of doing this would be to increase the allocation of existing host node resources to the VNF running on that host, e.g. allocating more CPUs, memory, or I/O resources to a VM, or in the case of the VNF running as a container/process, relaxing the resources constraints placed on them. This process is shown in a simplified way in the left part of Figure 6.28. Here a request for increased/decreased resources is sent to the Orchestrator which adds/removes resources to VNF_1, which may need a notification to realize that the situation has changed and adapt to the situation.

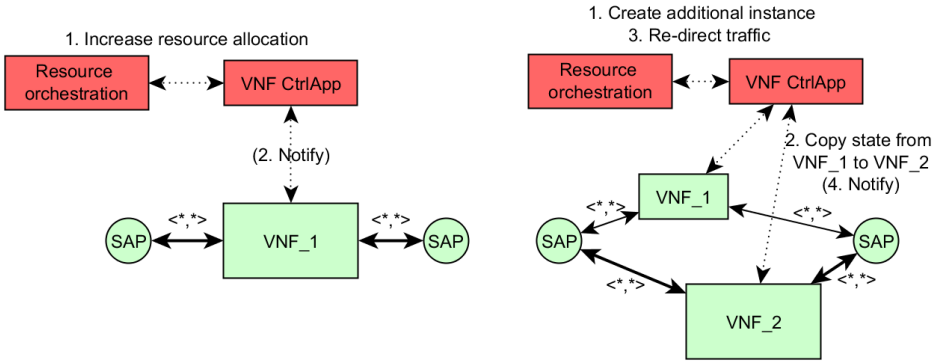

Figure 6.28: Scaling by resizing existing resources (left). Scaling by migrating to a new VM/container (right).

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

This approach requires that 1) host resources are available, and 2) that the NF is able to take advantage of the additional allocated resources. That a NF is able to use additional resources is not always the case, the implementation may for example be single-threaded and therefore unable to take advantage of additional CPUs (see Support in the VNF taxonomy). The performance bottleneck additionally may be in a resource that cannot be easily controlled, for example the main memory bus might be saturated rather than the CPU itself. This approach also requires that the virtualization or container system is capable of assigning additional resources to a running instance, and that the instance is able to notice that additional resources has been added.

If additional host resources are not available an extension of this approach is to migrate the instance to a different host machine where more resources are available, using e.g. traditional live-migration techniques such as KVM[10] live-migration or process/container migration in CRIU[11] depending on the virtualization system used (depicted to the right of Figure 6.28). Here a request for an additional instance is sent to the Orchestrator which allocates a VM/container (VNF_2) with more resources that the original (VNF_1). After allocation the existing state is transferred to the new instance, traffic redirected via the Orchestrator and a notification sent to VNF_2 if necessary. This still requires that the VNF is able to take advantage of the new resources, and typically would cause an interruption of the service during the final phase of the migration process. Live migration usually involves transferring all VM/process memory to the new host, marking memory that has been modified during the transfer process, and finally stopping the original VM/process to transfer the modified memory before starting the new VM/process. How long the interruption is varies on the virtualization system used, the NF itself, the network used to transfer the memory, etc. Additionally, during this process traffic must be redirected from the original host to the new, during the redirection packets may be lost or re-ordered and established connections broken (e.g. TCP connections terminating on the NF). Depending on the service provided by the VNF this may or may not be acceptable behaviour. There are more sophisticated ways of transferring state and flows, e.g. OpenNF [Gember-Jacobson2014], these are discussed below in Section **Scaling protocols**.

Scale-out by adding instances

Scaling NFs may also be done by creating additional VNF instances and spreading the load among them. The complexity of the process for this ranges from fairly simple in the case of a stateless VNF that accepts packet loss and reordering to quite complex for stateful VNFs with rapid state updates, high synchronization requirements, and a transition where both packet loss and reordering is unacceptable.

[10] http://www.linux-kvm.org/page/Migration

[11] http://criu.org/Main_Page

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

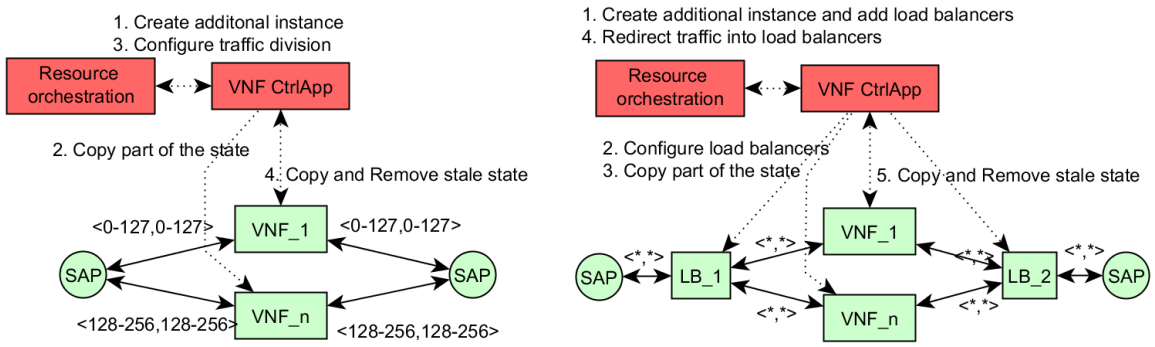

Figure 6.29: Scale-out example for Layer 1-4 traffic (left). Scale-out example for Layer 4-7 traffic (right).

The Scale-out approach can only be used if the traffic into the VNF is divisible, either at layer 1-4 or layer 4-7. Splitting the traffic in case of layer 1-4 could be performed by for example an OpenFlow-enabled switch or other programmable switches/routers. If higher layer headers are needed to split the traffic a particular Load balancer for those protocols is required, one capable of dividing traffic based on those higher layer protocol(s). These two options are shown in a highly simplified manner in Figure 6.29, an actual implementation may require many more steps to perform the function in a controlled fashion. In the left side of the figure a new VNF instance is created and the relevant existing state is copied to the new instance. Once the copying process is complete associated traffic flows are redirected by the Orchestration Layer and modified state once again copied and removed from the original VNF. In the right side of the figure the process is similar in the case of Layer 4-7 traffic, however the load balancers require configuration by the CtrlApp and traffic is completely redirected from the original VNF into the load balancers by the Orchestration Layer, rather than divided among the VNFs. One can imagine cases where the Layer 1-4 case is handled the same way using e.g. an OpenFlow switch as Load balancer in order not to burden the Orchestration Layer with many traffic redirection updates.

Depending on the particular VNF implementation there may be internal state associated with one or more traffic flows, this state also has to be divided and transferred to the new instance as well. The transfer of state(s) and the associated flow(s) needs to be synchronized to avoid race conditions, e.g. if one transfers the state associated with a flow first and then instructs a switch or load balancer to redirect traffic flow(s) to the new VNF instance, packets that were already in transit to the original instance may arrive and affect the state that was just copied to the new instance. One way to avoid this problem is to redirect only new flows (called *Progressive move* in the VNF taxonomy), which has no associated state in the original VNF; to the new VNF until the load of the VNF instances are balanced. However, this approach does not work in cases where there is state shared among all flows (or, sometimes between multiple flows), and may take a long time if flows

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

are long lived, new flows are rare, etc. It also requires that one is able to distinguish between new and existing flows (in the entity redirecting traffic), not a trivial problem.

In the cases when state is shared between multiple flows one can move groups of flow in unison i.e. move all the multiple flows sharing state atomically, however when this is not possible the groups might need to be subdivided and the shared state kept synchronized between multiple VNF instances during run-time (this is also true when state is shared between all flows). Depending on how often the state is updated and the required consistency model (e.g. eventual consistency, strong consistency, etc.), this problem could be handled by the same protocol responsible for the scale-out in the first place or it might require a dedicated separate system built-in to the VNFs with high bandwidth / low-latency data plane connectivity between the VNF instances. These issues (called *Modification* and *Consistency* in the VNF taxonomy, Table 6.7) may strongly affect how performance of the distributed VNF system scales with regards to the number of VNF instances, as in any distributed or parallelized system.

Finally, the scale-out event itself additionally burdens the original VNF during the process of scaling out, generating traffic for transmitting state, perhaps requiring locking of data structures within the VNF, etc. This has to be taken into account when the decision to scale-out is taken, perhaps one should anticipate increased load already at 80% load and start the process already at that level rather than wait until the load hits 100% for example. It is worth mentioning that observability points can provide observed performance metrics and monitoring information to be used for the scale-out decision.

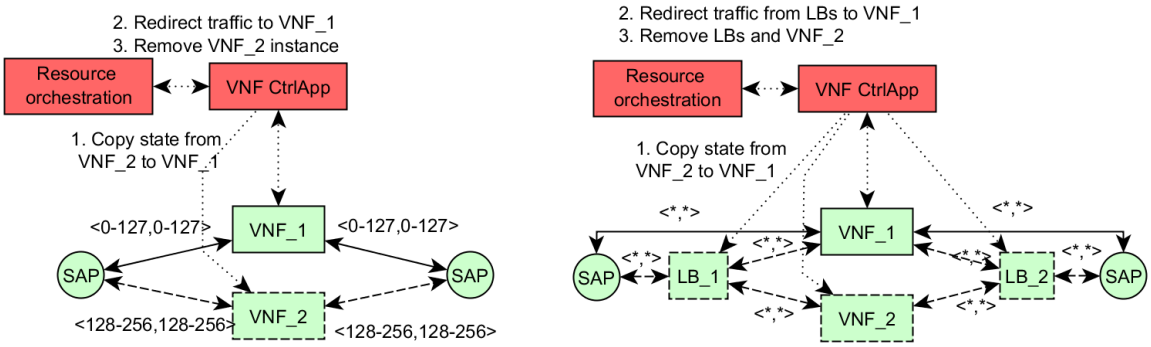

Figure 6.30: Scale-in of a Layer 1-4 VNF *(left)*. Scale-in of a Layer 4-7 VNF *(right)*.

Similarly to scaling out by adding instances, it is possible to scale-in by removing instances. This process is very similar to the opposite and requires state to be moved from one instance and the traffic redirected, likely with the same requirements (depending on the VNF) as for the scale-out event. This has to be overseen by the CtrlApp which may move individual, multiple, or all flows stepwise into a VNF instance, and finally signal the Orchestration Layer to remove the now “empty” VNF instance. Simplified examples of this process is shown in Figure 6.30, to the left a Layer 1-4 VNF is scaled-in by first copying

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

state from VNF_2 to VNF_1, then the traffic redirected, and finally the VNF_2 instance removed. In the right hand side a similar procedure is performed for a Layer 4-7 VNF.

In both the case of scale-out and scale-in a process of optimizing flow rules in both Layer 1-4 and 4-7 devices once the move has been completed may be useful, in order to reduce the number of entries used to move fine-grained flows into a smaller number of more generic flow entries.

Scaling dependencies

The performance of a VNF may also depend on other VNFs or functions that are not contained within the VNF instance itself and therefore not controlled by the VNF CtrlApp. This could be e.g. a block device attached over the network such as an iSCSI drive, a database used for authenticating flows, etc.. In these cases the performance bottleneck may appear to be the VNF when the real reason could be the bandwidth to the iSCSI device or an overloaded database server. Dealing with issues such as this requires wider understanding of the system than what a VNF CtrlApp alone can be expected to have and may be a topic for troubleshooting studies in WP4, which may apply techniques for root cause analysis which could potentially detect real issue.

Scale-out/in protocols

One system for dealing with the transfer of state and re-direction of flows with optional guarantees on packet loss and re-ordering is OpenNF [Gember-Jacobson2014]. The OpenNF architecture, seen in Figure 6.31, consists of three components, a CtrlApp responsible for directing scaling events by transferring flows using its knowledge of NF internal state and external input (e.g. measurements), a NF State Manager responsible for state transfer and event buffering, and a Flow Manager responsible for redirecting the traffic flows to and between NFs.

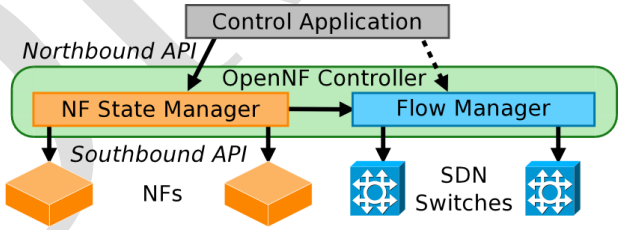

Figure 6.31: OpenNF architecture, taken from [Gember-Jacobson2014]

To perform loss-free and ordered state transfers two APIs are used, one southbound API that is implemented in the VNF itself used for retrieving and inserting state as well as keeping track of state updates, and a northbound API utilized by the CtrlApp to direct which flows (and state) to move, copy, and share between NFs. The southbound API consists of commands to retrieve, insert and delete state for the three different types of state to flow mappings, individual flows, multiple flows and all flows:

multimap<**flowid**,**chunk**> getPerflow(**filter**)
void putPerflow(multimap<**flowid**,**chunk**>)

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

void delPerflow(list<**flowid**>)
multimap<**flowid**,**chunk**> getMultiflow(**filter**)
void putMultiflow(multimap<**flowid**,**chunk**>)
void delMultiflow(list<**flowid**>)
list<**chunk**> getAllflows()
void putAllflows(list<**chunk**>)

Here *filter* represents a rule matching a particular or set of flows (much like a *Match* in OpenFlow), *flowid* identifies individual flows, and *chunk* is a "raw" block of state data (since different NFs may use various types of internal structures to store their state). In addition to these eight functions for transferring state two more commands are needed to observe and prevent state updates during a scaling event, in order to guarantee consistency:

void enableEvents(**filter**,**action{process | buffer | drop}**)
void disableEvents(**filter**)

These two commands are used to observe and prevent state updates caused by packets matching *filter*, the action *drop* causes packets to be redirected and buffered at the CtrlApp, the action *buffer causes* to be buffered locally, and finally the action *process* is used to notify the controller that a certain packet is being processed (and thus might modify the local state).

On the northbound API side, three commands are available:

move(**srcInst**,**dstInst**,**filter**,**scope**,**properties**)
copy(**srcInst**,**dstInst**,**filter**,**scope**)
share(list<**inst**>,**filter**,**scope**,**consistency{strong | strict}**)

The **move()** command transfers the state and traffic corresponding to *filter* from VNF *srcInst* to *dstInst*, with a *scope* (per-flow, multi-flow, all-flows), and certain *properties* i.e. loss-free and/or ordered. Similarly, the **copy()** command is used to clone state between two VNF instances, in order to provide eventual consistency for shared state but does not redirect traffic. Finally the **share()** command ensures *strong* or *strict* consistency for certain state for VNFs in the *inst* list.

Without going into details on how these commands are implemented and how the signalling works it seems that in principle the required framework for handling scaling of VNFs is there. There are however some issues that could be improved with regards to carrier environments, for example the current OpenNF system is designed to do significant amount of packet buffering and processing in the CtrlApp, something probably not feasible for VNFs which has to deal with large amounts of traffic. Tweaking the OpenNF design to move this functionality either into the VNF host system (e.g. a Universal Node) or into the VNFs themselves could be one approach to deal with these issues.

6.7.3 Dynamic processes

The Orchestration Layer is not only responsible for the initial deployment of a Service Graph but also has the responsibility of maintaining it during its lifecycle from start to

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

stop. During this lifecycle dynamic processes may require that the Orchestration Layer adds or removes components, change the amount of resources allocated for components, or even initiate migration of logical links or VNFs. In this section we detail what these dynamic processes are and outline some strategies on how the Orchestration Layer can deal with them. In Figure 6.32 a simplified view of the UNIFY architecture is depicted, showing with red circles the main sources of dynamic changes.

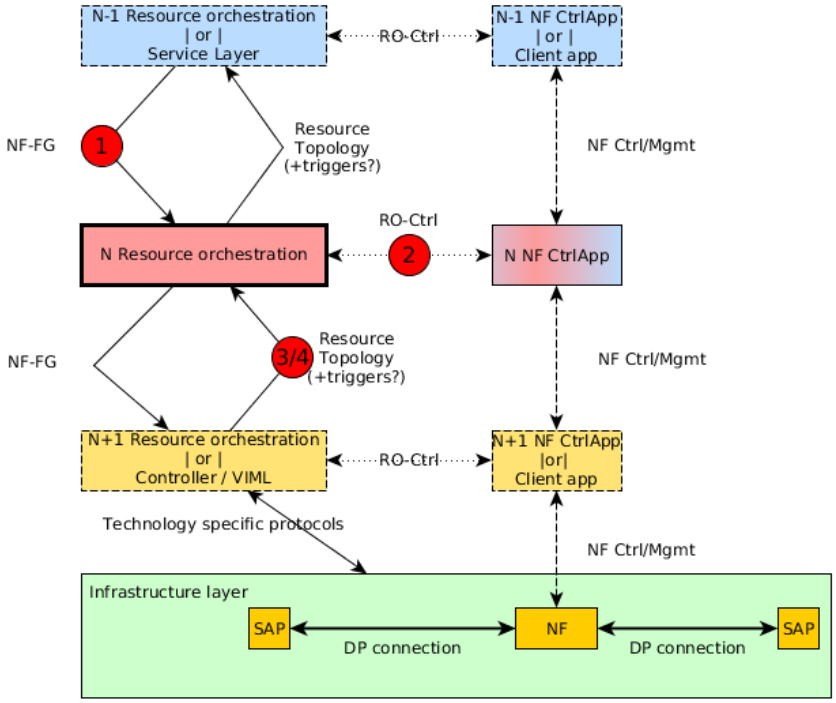

Figure 6.32: Simplified view of the UNIFY architecture with focus on dynamic processes affecting the Resource orchestration. The red circles highlight the interfaces which interact with the Resource Orchestrator.

1) Changes to Service Graph/NF-FG on the Northbound Interface from higher layers, e.g. modifications to an existing Service Graph directly from a client, or from a higher level Orchestrator in the case of hierarchical orchestration. The dynamic changes that may occur here are:

a) Addition/removal of SAPs, VNFs, and links between VNFs/SAPs

b) Modification of requirements on VNFs, Links, and end-to-end requirements

2) Changes to *part* of the NF-FG on the CtrlApp interface triggered either by a client through the NF control interface, or automatically based on internal VNF state e.g. if a CtrlApp determines that a function needs to scale out due to resource constraints in the NF (see Section 6.7.2). Possible dynamic changes here are:

a) Addition / removal of VNFs, and links between VNFs/end-points

b) Modification of requirements on VNFs, Links, and end-to-end requirements

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

3) Resource topology changes on the Southbound Interface; these may be caused by e.g. new infrastructure hardware or SAPs changing location in the network topology. Additionally, the intent to perform administrative work on some component, such as upgrading the firmware on a switch, may trigger the need to migrate traffic to other paths. The changes coming from the lower/Infrastructure Layer are:

a) Addition/removal/"administrative down" of SAPs, VNFs, compute/storage resources, links

b) Migration of SAPs

4) Triggers and reports from monitoring components, e.g. link failures or node failures, and performance degradations can affect a link that is explicitly part of an NF-FG from the start, connecting for example VNF A with VNF B, or it could be internal to a scaled-out VNF.

a) Failure or performance degradation triggers for physical or logical infrastructure (links, logical links, nodes, VNFs, etc.)

Modifications to a running NF-FG, or a subset of the NF-FG, coming from higher layers or a CtrlApp, (1) and (2) respectively, typically comes as a an modified version of the existing NF-FG description.

These dynamic changes coming from different layers and for different reasons are very diverse in their characteristics in terms of how often we expect an event to happen, and the amount of time the system has available to react to the event. In Table 6.8 we try to categorize the dynamic event types into six groups and give a ballpark figure on how often we expect the event to occur and how much time the system has to handle an event.

Table 6.8: Types of dynamic events, expected frequency and reaction times.

Dynamic event cause	Event frequency	Expected reaction time
Seasonal demand cycles (e.g. New Year's eve, Christmas, vacation period)	Monthly/weekly. Increased/decreased can likely be predicted in advance.	Hours to days.
Daily demand cycles (business hours, movement between industrial/residential areas)	Daily. Could likely often be predicted in advance, although unpredictable events may occur (e.g. public events, exceptional news stories)	Minutes to hours.
Updated service definitions, e.g. adding new SAPs such as additional offices, modifying	Daily/weekly. A customer updating their services is not expected to be very frequent, though it depends on the type of customer and type of	Minutes to hours.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

service requirements	change. E.g. connecting new offices happen less frequently than changing from silver to gold home internet subscription.	
SAP mobility, e.g. if a SAP represents a cell phone or outside broadcasting truck	Seconds to yearly, depending on what the SAP represents	Seconds to hours, can be service and customer dependent
Infrastructure changes, e.g. addition of new links and nodes or administrative stops for upgrades etc.	Daily to monthly for new infrastructure, daily for administrative changes	Minutes to days, depending on action (administrative stop vs new resources)
Physical/logical infrastructure failures	Mean time between failures unknown	Milliseconds to seconds, can be service and customer dependent

Our expectation is that infrastructure failures and SAP mobility puts the highest requirements on the Orchestration Layer, in terms of updating an existing NF-FG deployment in a timely fashion. This is followed by other unpredictable events, such as administrative stops, unexpected demand changes either coming from customers as updated requirements or from automatic scaling due to unexpected surges in traffic. Finally we have changes that can be somewhat predicted such as daily, weekly, and seasonal changes where there is time to pre-calculate reactions.

Strategies for dealing with these events are limited by how time consuming various operations in the UNIFY architecture are, across the various layers in the architecture. This includes e.g. time to start a new VNF or migrate an existing, time to create or move a network connection, time to decompose a NF, time to calculate placement for a whole or partial NF-FG, etc. For example, if the whole chain from initial NF-FG to deployment could be handled within the expected reaction time, re-deployment from scratch could be a viable strategy to deal with most of these events, however it seems unlikely that the whole process could be carried out within the milliseconds required to deal with infrastructure failures.

Detailing strategies for dealing with these events without timing information is premature; however we can outline some possibilities:

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

1. Recalculate the deployment and transform the existing deployment to the new configuration

2. Recalculate affected parts of the deployment and update certain aspects of the existing deployment. This approach could reduce time spent in the Orchestration Layer as well as time to update the infrastructure.

3. Pre-calculate failure scenarios and update the deployment when a particular event occurs. This approach can effectively remove any time spent in the Orchestration Layer, but pre-calculated scenarios may be outdated when a failure occurs.

4. Quickly handle certain events in lower layers (e.g. failures), and later re-optimize the deployment. For example for a link failure, immediately calculate and deploy new network paths and inform the Orchestrator what has happened, which then can take action to initiate re-optimization.

While there are many different events that can occur and each of them should be considered, many of them can likely be dealt with in a similar fashion. For example, in Figure 6.32, the events originating from higher layers (1) and events at the same level due to CtrlApp decisions (2) are very similar and can likely be treated identically.

6.7.4 Monitoring component interaction

The monitoring functionality developed in work package 4 needs to be integrated with the programmability framework described in this deliverable. The integration of WP3 and WP4 components involves several components and processes from both work packages and requires close integration in some processes. The main questions here are:

1. How to model monitoring functionality and resource requirements in Service Graphs and NF-FGs

2. How to orchestrate and operate monitoring functionality in the infrastructure

3. How to utilize monitoring results in WP3 processes, e.g. orchestration and VNF control

In the following sections we will describe some potential answers to these questions and describe some implications for the programmability framework.

6.7.4.1 Modeling monitoring functions – VNFs or node capabilities

How monitoring functions are modeled and what their roles consist of is described in Milestone M4.1 from WP4 but we include a brief summary here. The various monitoring functions under development in WP4 generally fit the model described in Figure 6.33, which shows the relationship of the three different monitoring components within a simple view of a Universal Node.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Monitoring of a NF-FG (partially or fully) as well as monitoring of the (virtualized) infrastructure[12] itself is performed by the means of one or several Monitoring Functions (MFs) under the control of a control application; this control application is not necessarily the same as a VNF control application. The functional scope of an MF typically covers one or several Observability Points (OPs) deployed in the infrastructure. Depending on the type of MF, the OP implementation capabilities includes measurement mechanisms, aggregation and analytics, as well as communication between OPs.

The three monitoring components are:

- A **Local Data Plane (LDP) component** is part of an OP and performs low-level monitoring functions such as accessing packet/byte counters in a OpenFlow switch, generating probe packets and injecting them into user traffic flows, piggy-backing monitoring data on user traffic passing through a switch, or monitoring some low-level aspect of the VNF Execution Environment such as CPU or memory usage.

- A **Local Control Plane (LCP) component** is also part of an OP and performs configuration and/or triggering of one or more LDP components in the same OP. In addition to this functionality it may perform some node-local analytics such as aggregating and analyzing monitoring data from one or more LDP components.

- A **Control Application / Control Plane component** performs similar functionality as the LCP component with a broader scope, controlling one or more MFs (in turn consisting of one or more OPs) stretching over multiple nodes in the network. It can configure and/or trigger Monitoring functions, aggregate measurement data from them, and perform analytics.

DRAFT

[12] Note that exposed infrastructure can always be the product of a virtualization function at lower layers.

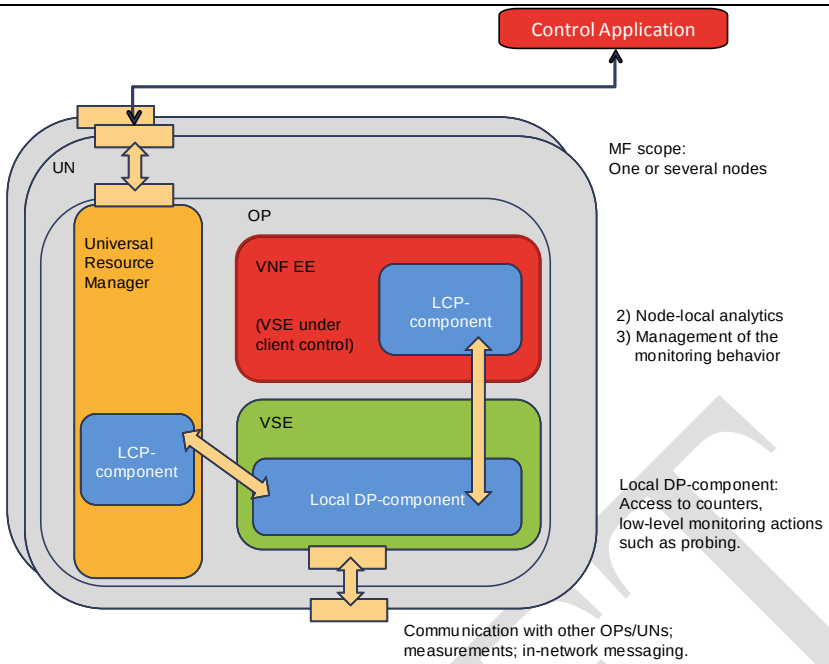

Figure 6.33: An overview of the mapping of MF and OPs on UNs.

When the purpose is to monitor the (virtual) infrastructure itself these components, the instantiation and management of these components are separated from the NF-FG orchestration process. This concept is referred as 'shared monitoring' and is explained in Section 3.2.9 of D2.2. However when monitoring a service defined as a Service Graph / NF-FG, the placement and instantiation should be part of the same process that is allocating the NF-FG resources, in order to fulfill both the service and monitoring requirements. Depending on whether the monitoring is done on (virtual) infrastructure or service level, and who the consumer of the monitoring results are, we define four different roles for monitoring.

Monitoring for the operator may be on both (virtual) infrastructure and service level, the *shared monitoring* is done in order to e.g. detect failures in the infrastructure, assist the orchestration process with a clear view of the resource status in the network, and trigger physical layer troubleshooting and failovers. *Service level monitori*ng can be useful for the operator in order to e.g. monitor SLAs of established services, perform root-cause analysis upon failures, and trigger logical layer failovers.

Monitoring for a client is only on the *service level* with purposes such as SLA monitoring, monitoring resource usage in order to trigger requests for additional resources, and troubleshooting service behavior. A client may also wish to insert monitoring functions that it has full access to, like any of the VNFs it is controlling, in order to perform e.g. fine grained monitoring of the certain parts in its service. For example if the client is a VNF developer it may wish to perform detailed DPI on packets going in and out of the developed VNF function.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

These four monitoring roles limits the options on how monitoring can be integrated in the programmability framework. For example when monitoring is performed for clients we cannot provide direct access to infrastructure components as this may allow a client to obtain information about services belonging to other clients as well as disrupt its own and others services.

6.7.4.2 Example monitoring function

One monitoring function under development is a function for monitoring link delay and loss monitoring utilizing a time stamping mechanism on packets traversing the network. As depicted in Figure 6.34, there is an LDP function on each network hop that adds a timestamp to a particular packet before forwarding it. The actual LDP function is realized as an OpenFlow action inside an OpenFlow switch. Local analysis and triggering of the LDP function is performed by a LCP component running on the SDN controller managing the various OpenFlow switches, the LCP component then reports aggregated results to a control application. The control application both manages the LCP components as well as reports results to higher layers.

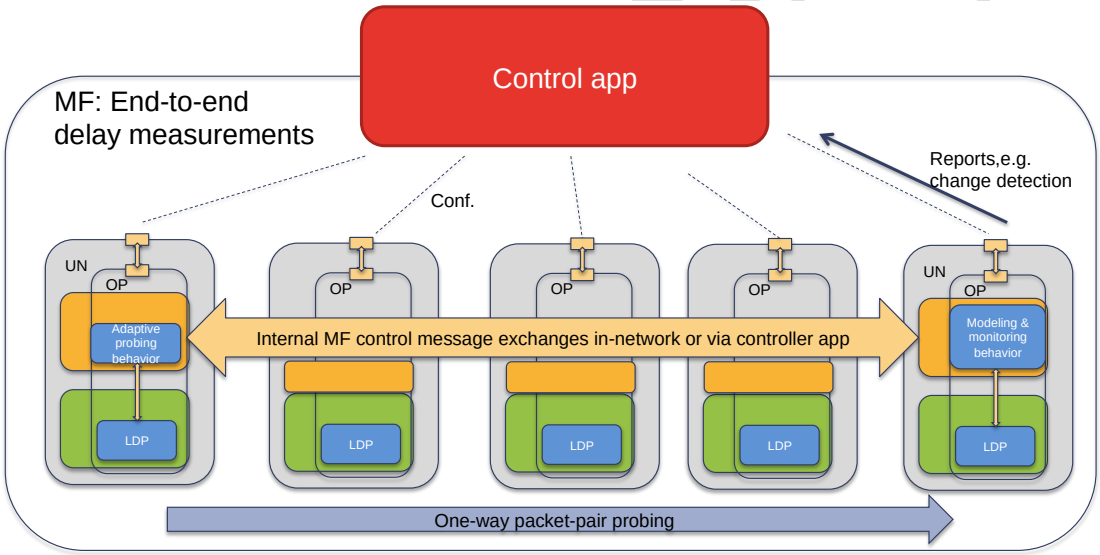

Figure 6.34: Example mapping of the link monitoring MF in the Infrastructure Layer.

6.7.4.3 Shared monitoring

When monitoring the virtual infrastructure there is no explicit connection to the NF-FG orchestration process, the monitoring functions have to be activated by a separate process. This process could be part of the bootstrapping phase of the whole system and could utilize the same functionality as the orchestration process, such as the topology database, underlying controller layers, etc. Once the system is operational, other processes have to ensure that new hardware added to the system has monitoring functions started if necessary. As the infrastructure controller layers already has bootstrapping phases for handling e.g. topology discovery, infrastructure monitoring instantiation and configuration could potentially be added to those processes. A way of describing and instantiating infrastructure monitoring is introduced below in Section 6.7.4.5.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

6.7.4.4 Service monitoring as monitoring VNFs

Since monitoring functions will be instantiated on the physical infrastructure connected to the forwarding graph and also require certain resources in order to function (e.g. CPU) it could be possible to include monitoring functions as additional VNF(s) in either the original incoming Service Graph in case the client wants to have control over the monitoring functionality, or inserted into the NF-FG as part of the decomposition process. This model works well for certain monitoring functions when the monitoring role is that where the client has full control over monitoring.

Assuming that the initial Service Graph connects SAP1 and SAP2 with a Firewall VNFs as depicted in Figure 6.35. Once the initial Service Graph has gone through the decomposition process it may look as in the bottom part of the figure. At this stage all links are still logical links only, without any mapping to physical forwarding nodes or link, similarly the placement of VNF nodes and their associated control applications have not yet been performed.

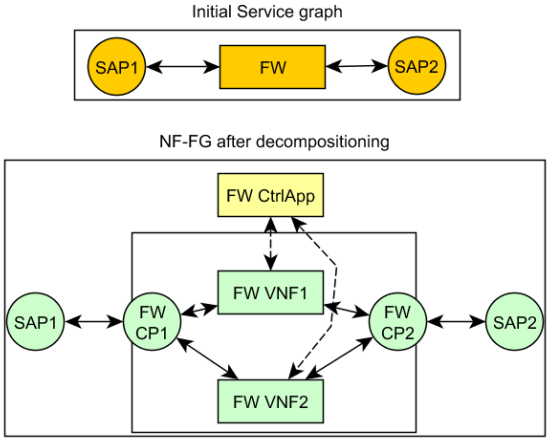

Figure 6.35: Initial service without monitoring

Assuming we would like to monitor the delay between FW-SAP1 and FW-SAP2 connected to the Firewall VNF. Doing this with monitoring functions as VNFs would look like Figure 6.36 where two delay monitoring functions have been included. As the decomposition is done the end result has included two functions into the graph that can now be embedded by the VNE algorithm. As the links are still logical at this point the only thing we can do to affect the placement of the monitoring VNF is to set the link latency requirements very low between e.g. Delay SAP2 and FW-SAP1 and hope that the leftmost Delay MF is placed very close to FW-SAP1. However, we still have no guarantees that no additional links are included in the placement so we cannot be sure we are not actually monitoring the latency between FW-SAP1 and FW-SAP2 plus a number of links. Additionally, modelling this type of monitoring functions as VNFs and inserting them between existing SAPs in the Service Graph require that the all the user traffic is passed through the monitoring VNFs. Passing client traffic through the monitoring VNF both affects the traffic we try to measure (e.g. adding latency) and adds instability to the system as a failure in the monitoring VNFs may

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

cause client traffic to be dropped in case the monitoring VNFs get overloaded. One could insert the Delay MFs with connections to the FW VNF but without connections to the external SAP1 and SAP2 to avoid passing user traffic through them, but this is often not advised since one typically wants monitoring probe packets to be treated as close to normal traffic as possible, and follow the same path through the network.

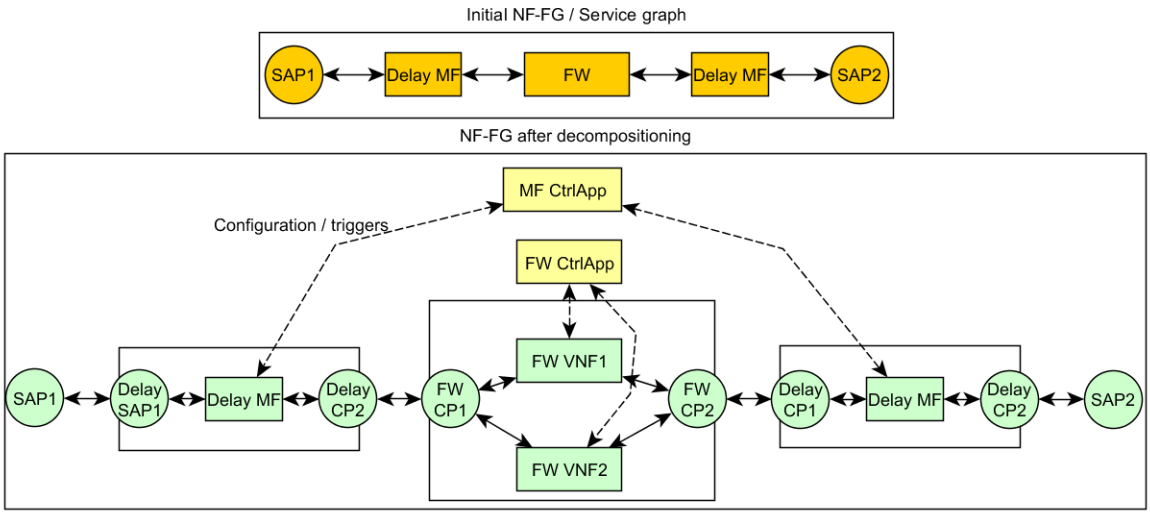

Figure 6.36: SG / NF-FG Extended with Delay Monitoring functions inserted as VNFs

However for certain monitoring functions this may be the intended behaviour, for example if a VNF developer wishes to perform DPI on traffic entering and leaving a VNF to find bugs in the way it processes packets.

Looking back to the monitoring example shown in Figure 6.34, where components have to be inserted into OpenFlow switches in order to perform the measurements, we can see that the VNF model is inadequate to express this requirement, as all links in the final decomposed NF-FG are still logical, without any mapping to the physical nodes and links that will realize the logical connectivity, and it is in those physical nodes where the functionality has to be installed or activated. Like the in the case of a VNF developer wanting DPI functionality, this functionality may be modelled as VNFs if the client requests OpenFlow switches and controllers as VNFs in its Service Graph, perhaps with the functionality preinstalled. However this would be inefficient as packets would have to pass through not only the physical switches part of the infrastructure but also through switches running in VNF execution environments, and would again not guarantee that we are measuring exactly what we intend to measure.

6.7.4.5 **Service monitoring as annotations**

As modeling monitoring functions as VNFs seems adequate only for fulfilling one of the three service monitoring roles we need some other method for including monitoring in the Service Graph and NF-FG. One option could be to include them in separated from the service definition itself and instead as accompanying annotations or metadata, and have a separate process to allocate and configure the monitoring functions instead of using the

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

VNF orchestration process. To describe the monitoring requirements one could reference the existing components of the Service Graph / NF-FG, e.g. the delay monitoring function from Figure 6.36 could be expressed as "Delay between SAP1 and SAP2" for an end-to-end measurement, or "Delay over VNF FW" for a measurement only over the firewall function. The metadata would have to be understood and modified appropriately through the layers, for example during decomposition it would have to be updated to appropriate fit the decomposed NF-FG. Once a NF-FG is decomposed and ready to be placed into the infrastructure we also need to provide a different way to express the resource requirements needed to instantiate not only the VNFs themselves but also the monitoring functions. While VNFs requirements are on compute and storage resources monitoring function requirements are difficult to describe in terms of e.g. CPU resources, in the example in Figure 6.34 the requirements are that the OpenFlow switches realizing the logical links are capable of performing the time stamping action and that their controller is able to run the Local Control Plane component for controlling those actions. These requirements could be described as node capabilities in the resource topology and the VNE algorithm restricted to place VNFs and their logical links only across compute and network resources that have the capability to fulfill the necessary monitoring capabilities. While adding complexity to the resource topology description as well as the VNE algorithm, similar placement restrictions are caused by SLA requirements such as latency, bandwidth, and CPU requirements.

The MEASURE Monitoring language

To specify *which* measurement functions should be activated, *what* and *where* they should measure, how they should be *configured*, how the measurement results should be *aggregated*, and what the *reactions* to the measurements should be, WP3 and WP4 is developing a language called "MEASURE" (for "**Mea**surements, **S**tates, and **Re**actions"). The MEASURE language definition is divided into in three main components; measurement-, state-, and action definitions.

Measurement definitions describe which measurement function should be activated, where the particular measurement should be taken, and how the measurement should be configured. This is described like a typical function call in a C-like language, i.e. "variable = function(placement, parameters)".

State definitions specify how measurement results should be aggregated and define thresholds for a combination of aggregated results, state definitions results in one or more finite state machines (FSM). The states and are described by a arithmetic expressions, e.g. "state = variable < value && function(variable)", where functions could be used to calculate e.g. averages.

Finally, **Action definitions** specify actions that are taken both when moving between states and while within a particular state. Actions may typically be to send a notification to another component in the UNIFY architecture, e.g. "S1: Notify(component, message)",

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

but additional actions may be useful to for example trigger actions in the data plane such as failovers.

A simple example for measuring the delay between two SAPs using MEASURE could be “M1 = Delay(SAP1, SAP2, interval=100ms)”, defining the measurement M1 as delay between SAP1 and SAP2, performed every 100 milliseconds. State definitions for this example could be “S1 = (M1 < 5 ms), S2 = (M1 > 5 ms)”, creating two states, one below 5 ms delay and one above. Actions in this case could be “S1: Notify(CtrlApp1,5min,M1), S1→S2: Notify(CtrlApp1,"ERROR",M1)”, that while in state S1 informs a control application with the value of M1 every 5 minutes. If M1 goes above 5ms and we enter state S2, an ERROR message is sent to the control application with the current measurement value attached.

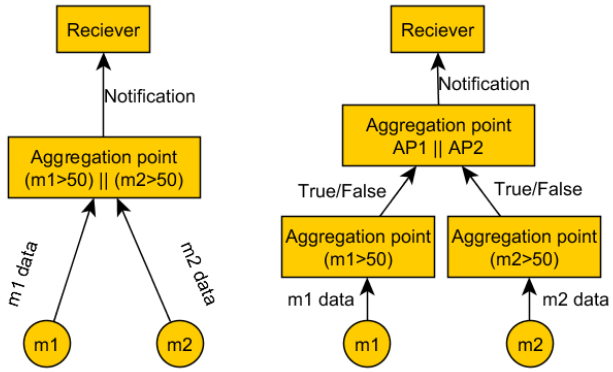

Figure 6.37: Two implementations of a MEASURE description

In Figure 6.37 an example of potential implementations of a slightly more complicated MEASURE definition is shown, based on input from two measurements, m1 and m2. In the left part of the figure the aggregation and state logic for both states is placed in a single aggregation point, which receives the raw measurement data from the two measurement functions, this aggregation point could be e.g. the monitoring control application component in the Figure 6.34 example. The aggregation logic is applied and a notification is sent to a relevant receiver if the evaluation is true.

Sending raw measurement data to the aggregation point in the MF control application may generate too much data on the network, in that case we can further distribute the aggregation logic by moving the aggregation logic closer to the measurement sources, e.g. into the Local Control Plane components, and let them perform an initial aggregation, for example “(m1>50)” and “(m2>50)”. They in turn could be configured to send a notification to the higher layer aggregation point in the MF control application when the value changes from true to false and vice versa. While this aggressive aggregation may not be necessary most of the time, we have the potential to automatically divide the aggregation logic into independent pieces and distribute it into different places in the network depending on traffic and processing load. However, doing so increases the latency from when a measurement is performed until a reaction can be taken, which may not be appropriate in all cases.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

While the description of the MEASURE language here is shown in relation to service monitoring the description language itself is not tied to services other than by referencing components in a Service Graph or NF-FG. The language could equally contain references to physical components in the infrastructure and be used to describe and the same process used to instantiate monitoring for infrastructure monitoring as well, using a similar description as a NF-FG but without NFs.

6.7.5 Resources related optimization

Dynamic decomposition can lead to multiple potential instantiations for one NF-FG. For example, in the case where a VNF type maps directly to three kinds of VNF instances; large (200 users), medium (100 users), or small (50 users) and we want to support 200 users, the exact solutions are given by the Diophantine equation 200L+100M+50S = 200, for which there are 4 solutions. On the other hand if the VNF should support 400 users the number of potential implementations grows to 9, and for 4000 users there are 441 solutions.

If the NF-FG contains three different VNFs each with 441 potential implementations, that is $441^3 \approx 8.6 \times 10^6$ potential solutions for the entire NF-FG. This is a slight overestimation however since it assumes that there is no overlap between the 86 million implementations in terms of actual resource requirements, a more accurate estimation may be around the number of solutions to 200L+100M+50S=4000x3, which is roughly 4000.

This is in the case where a VNF type maps directly to VNF instances which have clear resource requirements. In the case where there are intermediate VNF types, for example VNF type *Firewall* maps to *Firewall Linux* and *Firewall FreeBSD* which in turn map to three instance types, the potential combinations are much higher since *Firewall* for 200 users could map to *Firewall Linux* for X users and *Firewall FreeBSD* for 200-X users for all integers X between 0 and 200. This gives us 201 intermediate mappings, which in turn each may be mapped to a number of actual instances, as described by the Diophantine equation above.

This problem has also been considered to some extent in the related work. In [Meraghdam2014] the authors consider commutative Network Functions, such that an exponential number of Network Function orderings are needed to be considered. Based on this fast growing number of possibilities, the authors propose a heuristic, to find “good” orderings.

However, the problem of finding a good mixture of implementations and/or deciding which types of Network Function to choose can be considered a multi-dimensional knapsack problem (see e.g. [Freville2005]). These types of problems can be solved quite quickly by employing e.g. mixed-integer programming. A naïve solution would be to iterate through all potential solutions until one that can be orchestrated is found, however with such a solution is that we may have to iterate through all potential solutions until we find one that can be placed. In the previous case where there were 441 different solutions, if no large or medium instances are available within e.g. the bandwidth or latency requirements

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

on the NF-FG we may have to try 440 times before we find the only one that can be implemented, i.e. the one constructed only from small instances.

This argues for a tight integration between orchestration and decomposition, e.g. by informing the decomposition process how many of the large, medium, and small implementation could be instantiated, perhaps with some estimations of the available bandwidth, latency, etc., between them. This could assist the decomposition process to rule out combinations that are unlikely to be possible to orchestrate. In addition to this, potential solutions could be prioritized by applying a cost function which could prioritize e.g. a smaller amount of large instances over many small, instances that can be shared by multiple NF-FGs over those that can't, etc.

6.8 Abstract interfaces

Based on the architecture, the reference points and the information models described in previous sections, the next step is the definition of interfaces between the separate layers. We follow a three-step approach in the definition of interfaces. First, we have identified the reference points between the relevant functional components in order to be able to refer them during the architectural design (see Section 2 and D2.2). Second, we define abstract, high-level functions which should be supported by the interacting components. By this means, we define abstract interfaces. This section is devoted to this task. Third, we will implement the abstract functions realizing the interfaces in different ways (during the next phases of the project).

The interfaces between the main layers, i.e., U-Sl, Sl-Or, and Co-Rm interfaces, will be characterized (and slightly revised starting from the initial definition documented in MS3.1-3.2) in this section. Besides these inter-layer interfaces, we give the first definition of internal interfaces of the Orchestration Layer, namely, Or-Ca and Ca-Co interfaces and we give an additional interface regarding the new reference point called Cf-Or. The functions which MUST/SHOULD/MAY be supported via given interfaces are derived from the requirements (referred as req. x) declared in Section 3.

6.8.1 Application-Service (U-Sl) interface

The interface between the application (End / Enterprise Users, UNIFY Users, Developers) and the service provider is at the highest abstraction level in the UNIFY architecture. Via U-Sl interface, a normal user is able to request a given service or different reports on the service from the service provider interacting with an OSS. UNIFY Users (e.g., retail provider, OTT provider, content provider) or Developers can have lower-level access to the system with advanced functionalities. They can operate directly with Service Graphs, manage NF-IB and request UNIFY resource service using this interface.

U-Sl interface / API has to support the following operations:

Function	**request/release/update service**

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Description	A service is set up / released / updated by Service Layer (with the help of lower layers) and the status of the operation is sent back to the user as an answer.
Input	The user submits a request (start/stop/update) referring to a service which can be initiated e.g., in the GUI of a management system
Output	Service Layer sends back the status of the operation

Function	**get/send service report**
Description	Service Layer provides high-level measurement reports related to the SLA
Input	The user requests/polls KQIs (or reports are provided automatically according to initial parameters given in the service request)
Output	Service Layer provides measurement reports on KQIs regarding the SLA or only on requested parameters (see req. 3 of Section 4.1.)

Function	**notification/alarm**
Description	Service Layer is able to send notification to the user in case of failure or violation of the SLA or KQI requirements
Input	-
Output	notification/alarm is sent to the user identifying the event

Function	**list Service Graphs**
Description	Service Layer lists the running Service Graphs (SG) of a given UNIFY User or Developer
Input	UNIFY User or Developer queries his/her running SGs
Output	Service Layer lists running SGs belonging to the customer

Function	**request/release/update Service Graph**
Description	A service described by a ServiceGraph is started / released / updated by Service Layer (with the help of lower layers) and the status of the operation is sent back to a UNIFY User or Developer.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

	SG is a data structure describing various types of Network Functions, SAPs, the connections between them, restrictions on allowed traffic, and other service-level requirements (KQIs). For details on carried information, see Section A.2.5.2 and Section 6.2. These parameters are determined by req. 1, 2, 4, 6 and 9 given in Section 4.1. Pinning or restricting graph nodes/edges to a set of potential nodes (e.g., BiS-BiSes) is also supported (see req. 5 of Section 4.1) Both SG structure and service-level requirements can be modified dynamically.
Input	UNIFY User or Developer submits Service Graph to Service Layer.
Output	Service Layer sends back the status of the operation

Function	**get Service Graph info**
Description	Different types of information on a queried SG is provided by Service Layer
Input	UNIFY User or Developer queries information on a given SGs
Output	Service Layer provides different kinds of information (e.g., SLA, KQI reports, current status) on given SG

Function	**add/remove Observability Point to/from Service Graph**
Description	Service Layer adds / removes Observability Point (OP) to/from a given SG according to a Developer's request on-demand. OP can be treated as a special purpose NF, e.g., selected from a catalogue. OP can also become inactive when certain specified conditions have been fulfilled.
Input	Developer initiates the modification of a given SG (add/remove special purpose NFs)
Output	Service Layer sends back the status of the operation

Function	**list NFs from NF-IB**

DRAFT

Description	Service Layer provides the list of available NFs from NF-IB.
Input	UNIFY User or Developer queries available NFs, i.e., the current content of NF-IB
Output	Service Layer provides the list of (abstract) NFs which are currently available in the catalogue and can be used for composing SGs

Function	**add/remove/update NF in NF-IB**
Description	NF can be added to / removed from / updated in the NF-IB catalog by Service Layer according to the request coming from a UNIFY User or Developer. (Typically the service provider or a 3rd party NF developer can add new/modified/updated NF version to the catalog.)
Input	The following parameters and information have to be given as input: an abstract description (e.g., Yang data model); expected interfaces to other NFs/SAPs/architectural components; control channel connectivity; NF type (physical/logical, abstract/concrete); decomposition rules; an optional resource model (resource model can be blank and computed/estimated/measured automatically later); topological and temporal dependencies on other NFs; resource scaling requirements; optimization goals; monitoring parameters; and the NF implementation, as well.
Output	Service Layer updates NF-IB and gives back information on the corresponding NF

Function	**request/release UNIFY resource service**
Description	This management function of the BSS of Service Layer can be used by UNIFY Users or Developers in order to request / release UNIFY resource service. As a result, a new virtualization context (Virtualizer object) will be initiated at Resource Orchestration Sublayer.
Input	UNIFY User or Developer requests / releases a virtual context
Output	Service Layer provides information on the initiated / released virtualization context (Virtualizer object)

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

6.8.2 Service-Resource Orchestration (Sl-Or) interface

Service Layer gives a transformed/enriched Service Graph, namely Network Function Forwarding Graph (NF-FG), to the Orchestration Layer via Sl – Or interface. This data structure contains information needed by resource optimization tasks performed in Orchestrator modules. For details on the data structure and carried information, see Section 6.4. Sl-Or interface also appears between orchestrators realizing multi-level, recursive orchestration (see Figure 4.2).

Sl-Or interface / API has to support the following operations:

Function	**initiate/tear down/change NF-FG**
Description	Orchestration Layer takes the NF-FG request coming from Service Layer, tries to execute it according to its global resource view and sends back the result. The NF-FG request includes resource requirements of NFs and KPIs as well (see req. 1,2,4,6,9 of Section 4.1 and req. 1 of Section 4.2). Preferences on placement should be able to be defined per sub NF-FG and/or per NF (see req. 5 of Section 4.1). Change request includes the following operations: modify NF demands, insert/remove NFs in a SG, sharing NFs between SGs (sharing of NFs can be handled internally, however, Service Layer can also be involved)
Input	Service Layer submits an NF-FG
Output	Orchestration Layer sends back the status of the operation

Function	**get/send virtual resource info**
Description	Orchestration Layer provides resources, capabilities and topology information (e.g., BiS-BiS resource view)
Input	Service Layer queries virtual resource information
Output	Orchestration Layer provides a virtual resource view

Function	**notification/alarm**
Description	Orchestration Layer is able to send notification to Service Layer in case of failure or any violation of KPI thresholds

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Input	-
Output	notification/alarm is sent to Service Layer identifying the event

Function	**get/send observability info**
Description	Orchestration Layer provides observability info to Service Layer
Input	Service Layer requests/polls KQIs corresponding to NF-FGs or virtual resources
Output	Orchestration Layer provides measurement reports on KQIs corresponding to NFs, sub-graphs of NF-FGs or virtual resources (see req. 2 of Section 4.1.)

6.8.3 Resource Orchestration-Controller Adaptation (Or-Ca) interface

The Or-Ca interface is an internal interface of the Orchestration Layer between the Resource Orchestrator and the Controller Adaptation components. The main information exchanged between these components is stored in NF-FG data structures. Therefore, the same functions have to be supported here than we have seen in case of Sl-Or interface.

6.8.4 Controller Adaptation-Controllers (Ca-Co) interface

Controller Adaptation makes it possible to use different, technology dependent controller solutions on top of the infrastructure. This requires adaptation functions which translates information carried by NF-FG into messages which can be sent to the northbound interface of a given controller. Hence, this interface significantly depends on the controller itself as it is the northbound interface of that. In case of controllers not implementing this interface, the controller has to be extended by this functionality.

6.8.5 Controllers-Infrastructure (Co-Rm) interface

The interface at the Co-Rm reference point is determined by the protocols used at the southbound interface of the controllers. Here, several available protocols can be invoked and adapted to our special purpose architecture. For example, available protocols, such as OpenFlow, NETCONF, OFconfig, OVSDB, and libraries such as libvirt, can be used in Co-Rm interface. The definition of this interface is out of the scope of the UNIFY project, however, the required primitives can also be defined in an abstract way.

Co-Rm interface / API has to support the following operations:

Function	**start/stop/restart NF**
Description	Infrastructure Layer starts / stops / restarts an NF
Input	Controller requests to start/stop/restart given NF (see req. 2 of

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

	Section 4.6)
Output	Infrastructure Layer sends back the status of the operation and gives back access information to the NF

Function	**start/stop switch (forwarding element)**
Description	Infrastructure Layer starts / stops a (logical) switch
Input	Controller requests to start/stop forwarding element, e.g., logical OpenFlow switch (see req. 1 of Section 4.6)
Output	Infrastructure Layer sends back the status of the operation and the control interface of the switch

Function	**connect/disconnect NF to/from switch**
Description	Infrastructure Layer connects/disconnects an NF to/from a (logical) switch
Input	Controller requests to connect/disconnect an NF to/from a specified logical switch, e.g., via virtual Ethernet interface
Output	Infrastructure Layer sends back the status of the operation

Function	**configure switch**
Description	Infrastructure Layer configures flow entries into a switch
Input	Proactive: Controller sends flow entries to given switches Reactive: Infrastructure Layer sends request to Controller in case of new flows
Output	Status of the operation

Function	**get/send capability info**
Description	Infrastructure Layer provides info on its capabilities
Input	Controller requests capability information
Output	Infrastructure Layer sends information on different types of capabilities

DRAFT

Function	**notification/alarm**
Description	Infrastructure Layer is able to send notification to upper layers in case of failures or any unexpected events
Input	-
Output	notification/alarm is sent to Controller (or to OSS) identifying the event

Function	**configure observability components**
Description	Infrastructure Layer configures special components, e.g., individual observation points of a Monitoring Function are configured. The exact operation depends on the observability component.

Function	**get/send observability info**
Description	Infrastructure Layer provides observability info towards dedicated components.

6.8.6 Resource Control Function-Resource Orchestration (Cf-Or) interface

The Cf-Or interface supports the same functions as Sl-Or. Additionally, UNIFY architecture supports the delegation of NF-FG (or VNF) decomposition and VNF scaling tasks to a special controller entity running in the Network Functions System which is called Resource Control Function within Deployed Service (CtrlApp).This feature requires further interaction between the control function and the Resource Orchestrator and additional functions at the Cf-Or interface.

Cf-Or interface / API has to support the following additional operations (besides the functions provided by Sl-Or):

Function	**decompose VNF**
Description	Resource Orchestrator delegates VNF decomposition to CtrlApp
Input	Resource Orchestrator sends an abstract VNF (simple NF-FG composed of a single element) to the CtrlApp and delegates the decomposition task. VNF type and requirements have to be added as input parameters.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Output	NF-FG

Function	**scale VNF**
Description	Resource Orchestrator delegates VNF scaling to CtrlApp
Input	Resource Orchestrator sends an abstract VNF (simple NF-FG composed of a single element) to the CtrlApp and delegates the scaling task. VNF type, requirements, optional state and measurement parameters have to be added as input parameters.
Output	NF-FG

6.9 Multi-domain aspects

In Section 6.3 of deliverable D2.1 a first overview of recursion in the UNIFY architecture is provided in order to cover multiple domains. A domain refers either to a set of resources under the control of the same administrative entity. The set of resources could either be a set of network resources (switches, routers, etc.), a.k.a. a network domain, or a set of cloud resources (compute, storage and network resources), a.k.a. cloud domain (e.g., a DC domain).

In such a recursive/hierarchic UNIFY architecture a Global Orchestration Layer interacts via its Controller Adaptation component with the Orchestrator Layers of individual domains. The nature of this interface may support different levels of exposure. Let's consider the following scenario where a Global Orchestrator interacts with the Orchestrator of a commodity (OpenStack-controlled) datacentre (DC) and the controller of a (OpenDaylight-controlled) SDN network domain.

Depending on the level to which the DC wants to expose its resources, we might distinguish between the following scenarios:

a. The SP wants to use third party's (off-the-shelf) DC to deploy VNFs even if it means that the VNFs must be tailor made to handle network tunnel end points (Black Box approach).

b. The SP who owns the DC infrastructure hence is willing to expose control interfaces beyond off-the-shelf APIs (White Box approach).

c. The SP wants to utilize state-of-the-art SDN control and data plane split design to virtualize and hide DC internal networking (Big Switch virtualization).

d. The SP pursues joint compute and network programmability interface for full domain virtualization (NF-FG abstraction).

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

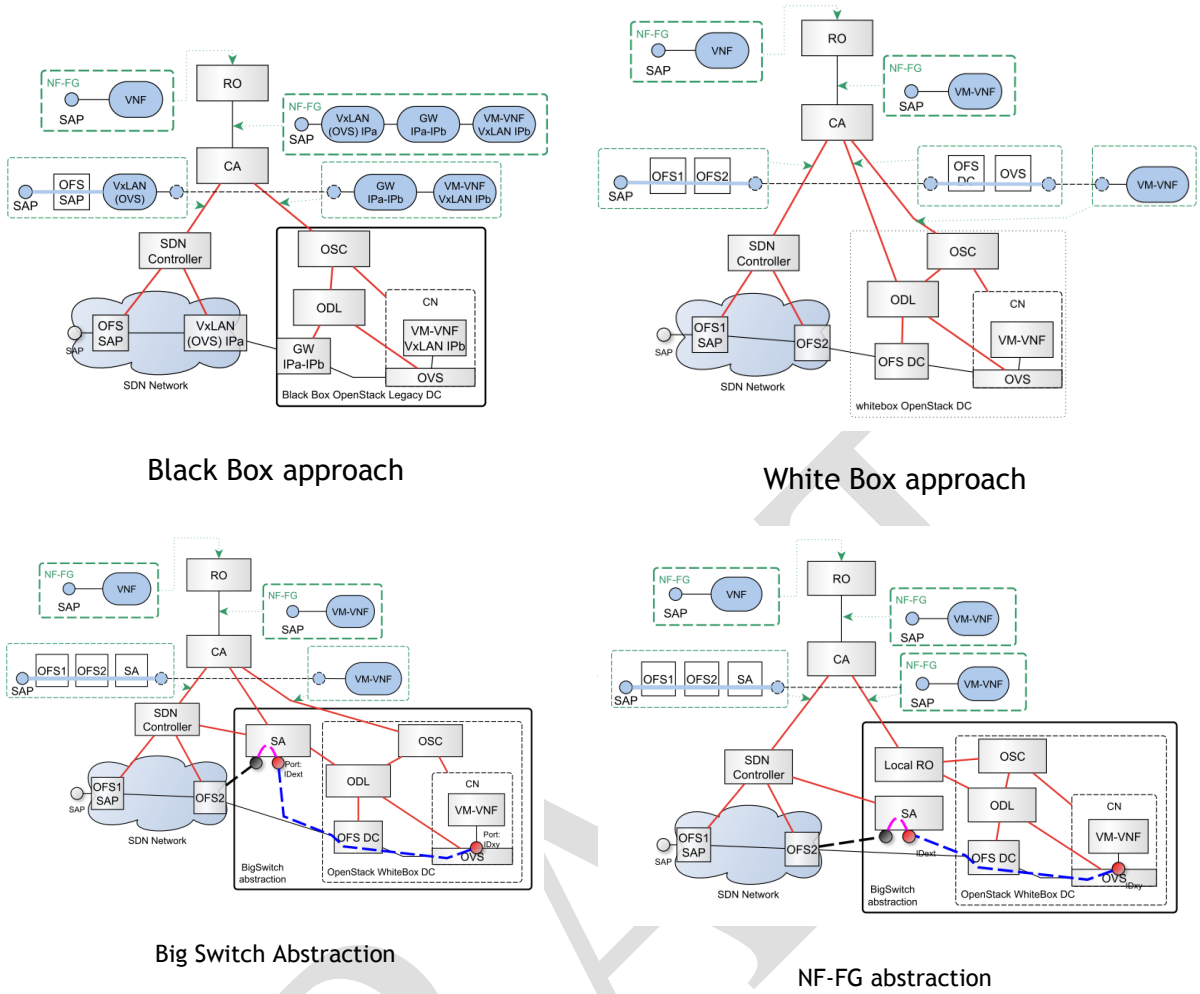

Figure 6.38: Multi-domain abstraction variations

Common to all scenarios are a genuine NF-FG request containing in our simple example one VNF connected to an End Point (or Service Access Point, SAP); an overarching RO, which maps the VNF to the DC; a CA splitting and translating the RO's mapped NF-FG to infrastructure controllers' APIs: an SDN Controller with an SDN domain including the SAP, and various set-ups of DC components' APIs (see Figure 6.38). The SDN controller simply gives global resource view to the North from the South, and does not perform information hiding or abstraction. Providing a simplified, abstract view of an internal topology can be a task of a dedicated entity, like a Switch Agent (SA) in some of our examples below. The OpenStack-internal entities in our figures are the OpenStack Controller (OSC) which can be considered as the Dashboard of OS, the OS-internal OpenDaylight (ODL) controller responsible for realizing the Neutron network of OS, the DC's OpenFlow gateway switch (OFS DC), compute nodes of the DC (CN), and compute node internal Open vSwitches (OVS).

The dashed green boxes in the figures represent the request that is made from the upper component to the lower one through the corresponding red coloured control channel. In

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

our example for simplicity reasons the VNF has only a single network interface, realizing both the incoming and the outgoing interfaces. A real VNF can have multiple interfaces.

Black Box DC (a): In this set-up, due to the lack of forwarding control within the DC no native L2 forwarding can be used to insert the VNF running in the DCinto the service chain. Instead, explicit tunnels (e.g., VxLAN over IP) must be used, which needs termination support within the deployed VNF. Therefore, the genuine VNF of the NF-FG must be decomposed into a VxLAN termination point in the SDN network domain, a routed IP gateway and a LAN network configuration in the DC, and a VxLAN capable Network Function image (VM-VNF-VxLAN).

White Box DC (b): If the internal network of the DC can be exposed in full details through an SDN Controller, e.g., OpenDaylight (ODL), to the overarching RO (see Fig.) then native L2 forwarding may be applied all through from the SAP to the VNF's port in the DC. The implications are that all resource dynamisms of the DC are exposed to the RO.

Big Switch virtualization (c): SDN allows split control and data plane design. If the SP or a third party DC provider wishes to hide the internal network details and dynamism of the DC, then she can do so by adding a software Switch Agent (SA) component corresponding to the control plane of an abstract Big Switch (see Fig). The role of the SA will be to map DC internal VNF ports to service access ports appearing in the transport SDN network domain through the external SDN controller. Note that the SA is only a control plane abstraction and the data plane execution can be mapped to the internal switching resources of the DC.

NF-FG abstraction (d): All the above method requires sequential orchestration of compute and networking resources, i.e., once the VNF is instantiated to a compute node the forwarding overlay is created to attach to it. However, NFV expects that both networking and compute constraints could be considered equally. For example, a VNF in the DC may be instantiated closest to the gateway involved in the service chain or VNFs of the same service chain should be orchestrated with their compute node proximity in mind. This can only be considered if compute and networking requests and requirements are matched and merged together for local orchestration. However, this yields to the idea of genuinely combining and transmitting compute and networking requests. According to the UNIFY programmatic framework, the NF-FG at the Sl-Or is one such combination of resources. Since the Or-Ca contains a mapped NF-FG, the CA may simply send a sub NF-FG graph to the Local RO.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

7 Universal Node interfaces

7.1 Universal Node Architecture

The Universal Node (UN) architecture detailed in [D5.2] describes the latest vision in UNIFY regarding the UN. Figure 7.1 shows the main functional blocks and external interfaces from this architecture. Although this section focuses on the UN interfaces, the components of the architecture are introduced previously for clarity.

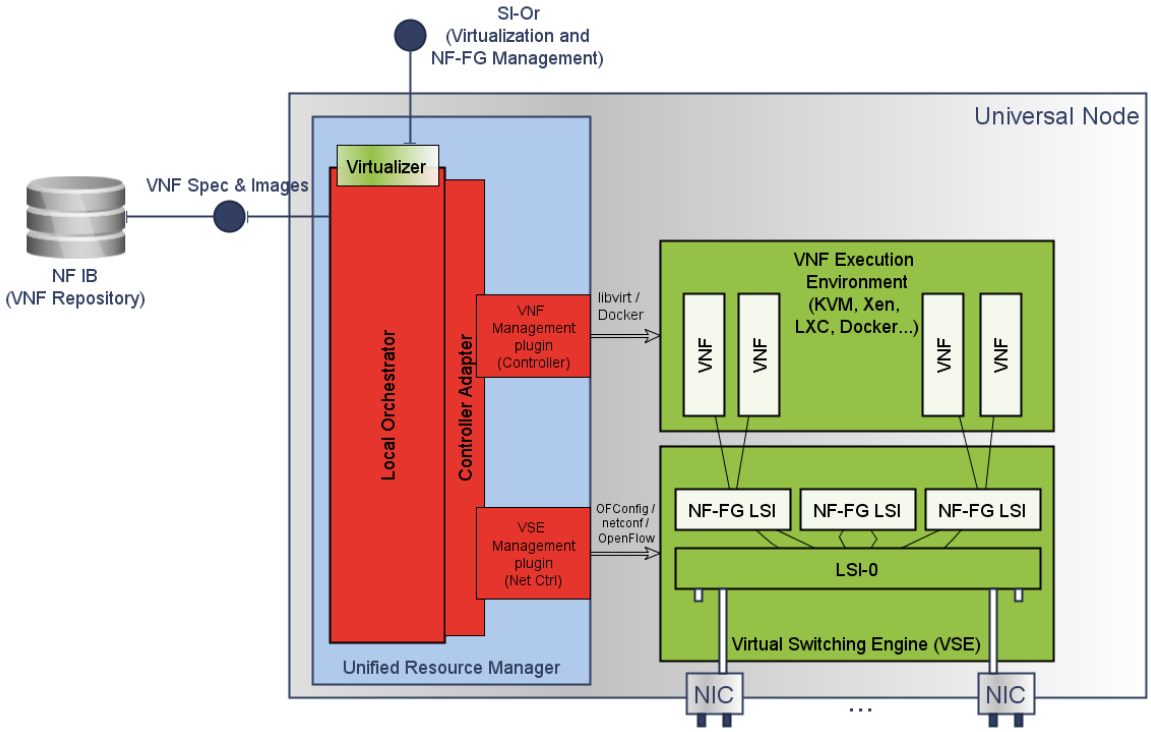

Figure 7.1: Current working UN architecture

As depicted in the figure the three main functional blocks of the UN are the VNF Execution Environment (VNF EE), the Virtual Switching Engine (VSE) and the Unified Resource Manager (URM).

- The VNF Execution Environment (VNF EE) represents the computing resources and several different compute platforms are considered as virtualization solutions for implementing it, from hypervisors to simpler container based approaches like Docker or Linux Containers (LXC).

- The Virtual Switching Engine (VSE) represents the networking resources and focuses on implementing packet switching functionality. It is responsible for managing the physical network interfaces and steer the traffic according to the part of the NF-FG deployed on the UN.

- The Unified Resource Manager (URM) acts as the local orchestrator and has a complete view of the node regarding the available resources, their topology and internal

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

constrains. The UN interfaces are provided by the URM, which then controls the VSE and VNF EE to meet the NF-FG requests and subsequent management actions.

The UN interfaces identified so far are also reflected in the previous figure:

- Resource management interface: this interface is responsible for discovering the resources exposed by the node, updating the list of resources (as a result of actions requested through other interfaces or internal reconfiguration at the node) and reporting the current availability of resources to the upper layers based on the already deployed NF-FGs. The primitives provided by the UN for resource management are summarized in Table 7.1.

Table 7.1: UN Resource Management primitives.

Primitive	Request	Response
Get Node Info and Capabilities	(empty)	• Total processing capacity • Total memory • Local disk capacity • CPU Info • Platform Tag • Ports List • Flow space specification capabilities • Supported VNF Types
Get Available Resources	(empty)	• Available Processing Capacity • Available Memory • Available Local disk capacity • Available capacity on ports

- NF-FG management interface: this interface covers the deployment and management of the NF-FGs at the UN. This interface is the most relevant from the programmability point of view, since it focuses on managing the NF-FG lifecycle. The primitives provided by the UN to deploy and manage NF- FGs are summarized in Table 7.2.

Table 7.2: UN NF-FG Management primitives.

Primitive	Request	Response
Deploy NF-FG	Graph Id / Graph Data	Graph Id / Result Code
Modify NF-FG	Same as Deploy NF-FG, Graph Id must correspond to an already deployed NF-FG.	
Delete NF-FG	Graph Id / Graph Data	Graph Id / Result Code
Get NF-FG List	(empty)	Graph Ids List
Get NF-FG Data	Graph Id	Graph Data

- VNF Template and Images repository interface: this interface is responsible of fetching and recovering the appropriate VNF images from the external/central VNF

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

repository. When a new NF-FG is requested to the UN, it needs to fetch the detailed specification and recover the related binaries to implement the requested VNFs. The list of operations supported by this interface is summarized in Table 7.3.

Table 7.3: UN VNF Template and Images primitives.

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

7.2 UN relation to the UNIFY architecture

In the layered model defined in UNIFY, the Universal Node implements functionalities of both the Infrastructure Layer (completely, by means of the VNF EE and VSE) and the Orchestration Layer (partially, by means of the URM). Regarding the programmability framework, the northbound interfaces exposed by the UN will match the Ca-Co reference point with the following considerations:

- The input format will be a NF-FG as handled in the upper layers (Ca-Co).
- The scope of the input from the Controller Adaptation layer will be a sub-graph containing all the elements to be deployed in the UN.

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

The current working approach in WP5 is that the NF-FG management interface is used to manage all the resources provided by the UN (i.e. VNF and VSE) so the Unified Resource Manager can optimize the placement of the requested NF-FG in its internal resources. The relation of the UN architecture to the reference points defined by the UNIFY architecture is shown in Figure 7.2.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

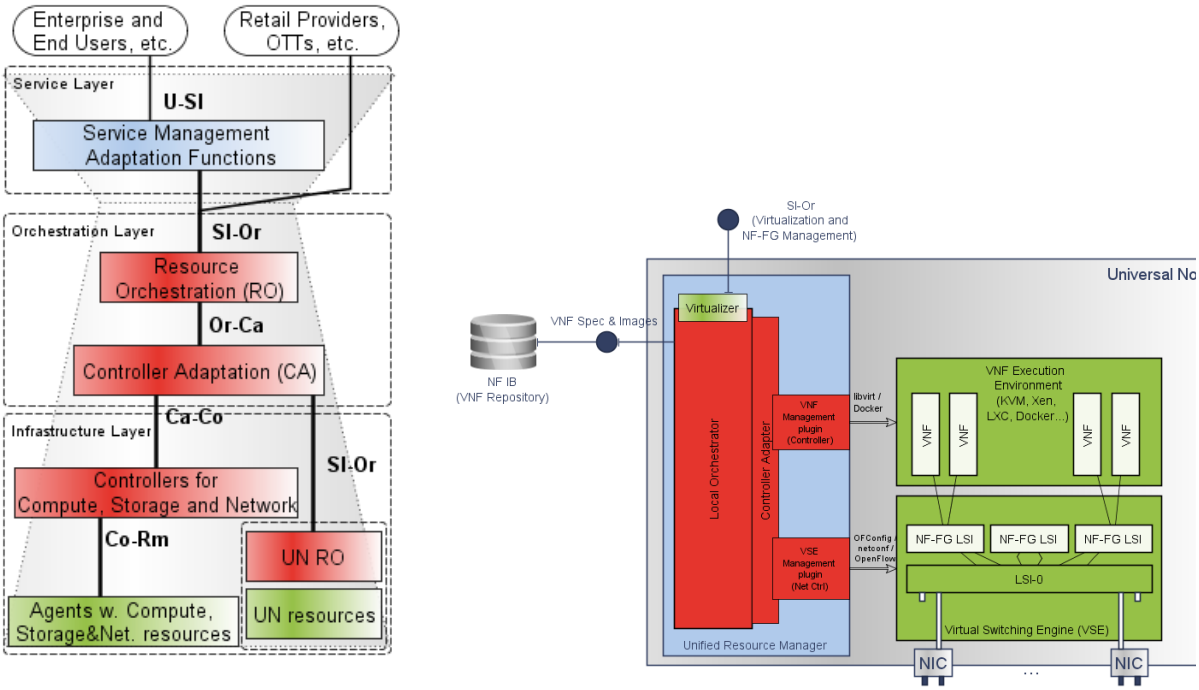

Figure 7.2: UN architecture in relation to reference points

In order to allow the UN to perform internal optimizations of the deployed NF-FG, the scope of the sub-graph must include both the elements related to the NFs to be deployed and the elements related to the traffic steering mechanism (to be defined later on in the project). That is, the scoping performed on the Controller Adaptation layer for the sub-graphs to be deployed on a UN must be done according to a domain criteria and not a functional criteria, as exemplified in Figure 7.3.

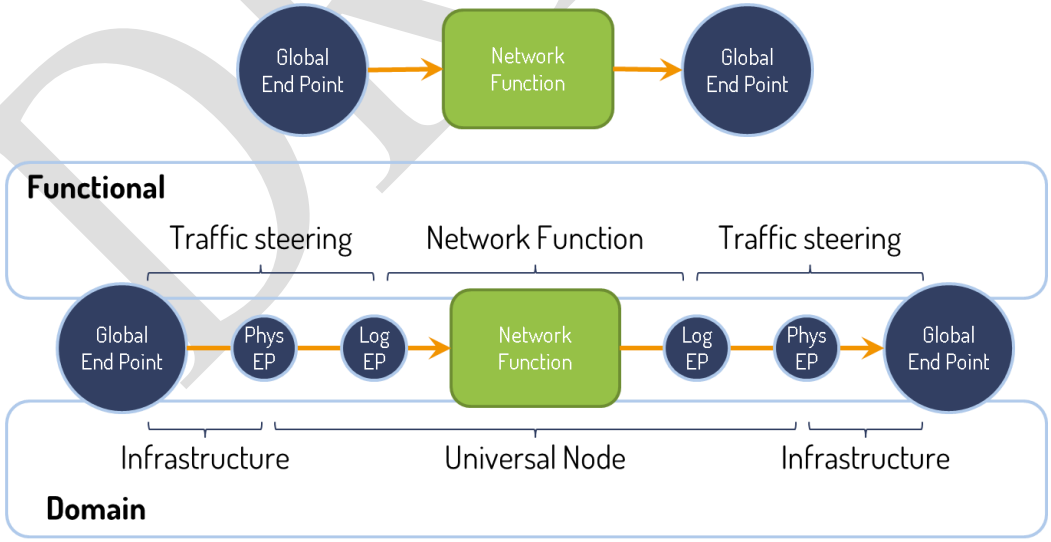

Figure 7.3: Service Graph, NF-FG graph and traffic steering

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

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

4. The scope of WP5 is a single UN, any process involving more than one UN will be handled by WP3.

5. The scope of the NF-FG handled to the UN will be everything related to the global NF-FG to be deployed in that UN:

a. If several NFs of the same NF-FG are deployed in the same UN, the input will be a single NF-FG sub-graph with the information related to all of them (including internal connectivity).

b. If several NFs of different NF-FGs are deployed in the same, the input will be separate NF-FG sub-graphs with the information related to each of them. The isolation between the NF-FGs must be assured.

6. Regarding the fulfilment of KPIs, two scenarios are possible:

a. During deployment, the UN can detect that it is not able to fulfil the requirements and would then reject the deployment request from the Orchestrator (WP5 to WP3 signalling).

b. During runtime, the UN can detect that it is not fulfilling the requirements and would then notify the monitoring process (WP5 to WP4 signalling, and then to WP3 if needed).

7. When the scaling of the NF-FG is NF dependant:

a. If the NF can handle resource changes at runtime, it will be managed by the UN (internally at WP5).

If the application can NOT handle resource changes at runtime, it will be managed by the upper layers

DRAFT

8 Programmability aspects of use cases

8.1 Elastic Network Function use case

The use case covers the deployment and operation of an Elastic Network Function in the UNIFY framework to demonstrate the different methods for scalability supported, as detailed in Section 6.7.2. The approach followed allows for an incremental verification of the capabilities of the framework, with emphasis in those methods more reliant on functionalities of the UNIFY framework, as opposed to methods more dependent on functionalities of the VNF itself.

Following this approach, the use case shall demonstrate:

- Scale-up/down of individual NFs and scale-out by adding instances (Section 6.7.2.2).
- Scalability triggered by changes in the NF-FG from upper layers and triggers from monitoring components (Section 6.7.3).
- Scalability managed by the UNIFY framework (Section 6.6).

8.1.1 Initial assumptions

In order to define the scope of the use case, the following initial assumptions are made:

- The use case considers a single deployment option in the Infrastructure Layer so the Orchestrator placement logic is not involved.
- The Infrastructure Layer bootstrapping process has been completed successfully beforehand so the infrastructure is available for the Orchestrator to deploy a Network Function Forwarding Graph (NF-FG).
- The NF-FG handled to the Infrastructure Layer for deployment is fully characterised and the VNF images to be executed are available at the Infrastructure Layer (the process for retrieving the VNF image is not included in the use case).
- Scalability of the NF is considered in the NF definition, including the elements to scale and the criteria for splitting the job among the different elements.

8.1.2 High level use case process

The use case can be divided in the following process blocks, each of them aiming to demonstrate a different aspect of the UNIFY framework in an incremental manner. The steps pertaining each of these blocks are further detailed in the next subsections:

1. Initial deployment of the NF-FG.
2. Change of the NF-FG requested by the user triggering a scale-up.
3. Monitoring event detection triggering a scale-up.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

4. Monitoring event detection triggering a scale-out.

5. Monitoring event detection triggering a scale-in.

6. Change of the NF-FG requested by the user triggering a scale-down.

8.1.3 Service Graph and Network Function Forwarding graph decomposition

The Service Graph initially requested by the user is decomposed by the different layers as described in Section 6.6. In the scope of the Elastic Network Function use case and based on the assumptions previously stated, the following operations take place (described in Figure 8.1):

- NF-FG initial decomposition, performed in Service Layer by Adaptation Functions: based on the user Service Graph requirements the Elastic Network Function NF-FG is selected.

- NF-FG placement and steering, performed in the Orchestration Layer by Resource Orchestration: the infrastructure node is selected to deploy the NF-FG and the appropriate inbound and outbound traffic steering is determined.

- NF-FG scoping, performed in the Orchestration Layer by Controller Adaptation: the NF-FG is split according to the domain criteria producing five sub-graphs, two for the traffic steering from the Service Graph endpoints up to the infrastructure node physical endpoint, one for everything to be deployed in the infrastructure node (containing both the NF and the traffic steering from the infrastructure node physical endpoint to the NF logical endpoint and two for the traffic steering from the infrastructure node physical endpoint to the Service Graph global endpoints.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

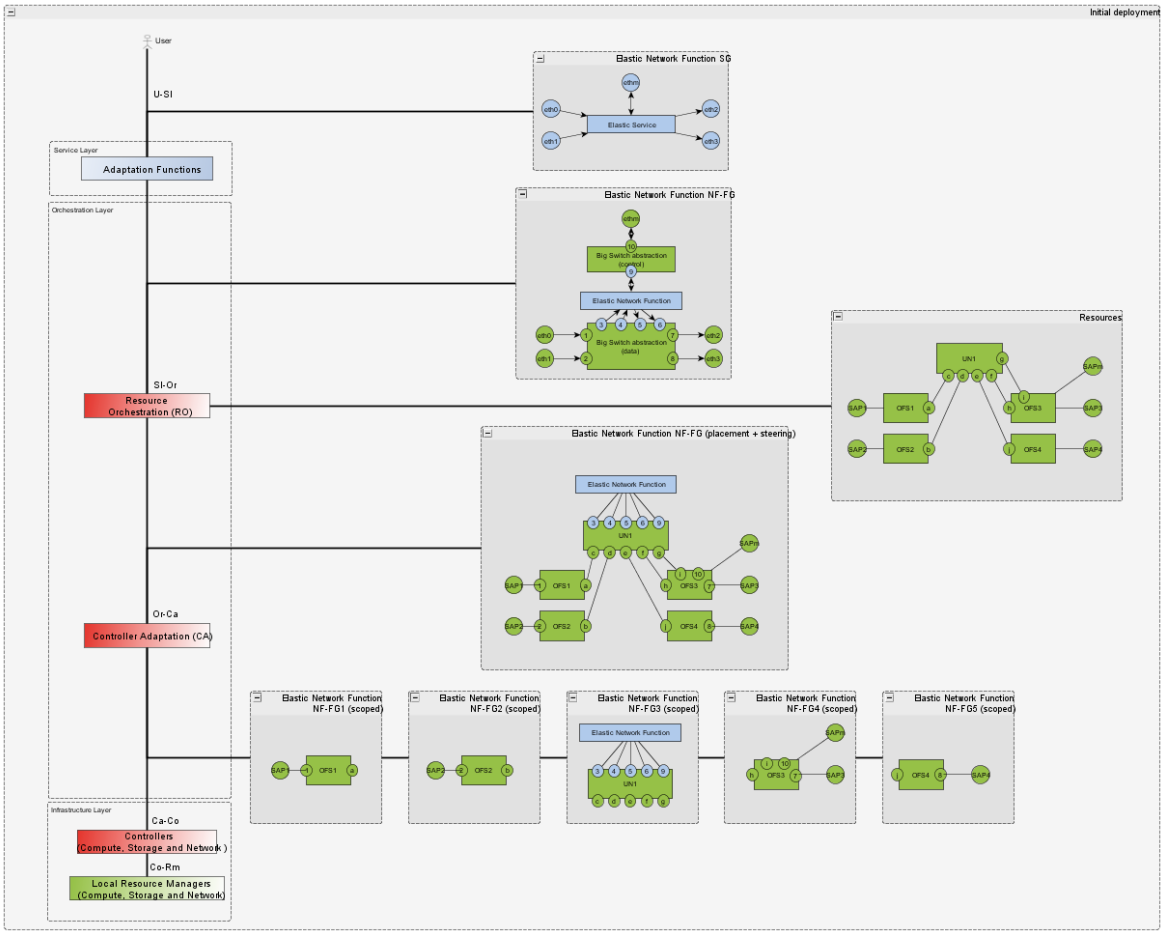

Figure 8.1: Service Graph and Network Function Forwarding Graph decomposition

8.1.4 Detailed Use case process and information flow

8.1.4.1 Initial deployment of the NF-FG

Step	Description	Input	Output	Actor
1.1	Service Graph to deploy handled to Service Layer	N/A	Service Graph	Service User
1.2	Service graph is mapped to a specific NF type supporting scalability	Service Graph	NF-FG with abstract NFs	Service Layer
1.3	Fully characterized NF-FG to deploy is handled to the Orchestration Layer	NF-FG with abstract NFs	Fully Characterized NF-FG	Service Layer
1.4	Based on the NF types in the NF-FG types, requirements and the available resources a	Fully Characterized NF-FG	Place for deployment	Resource Orchestrator

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

	placement is selected			
1.5	NF-FG to deploy is handled to Controller Adaptation	Fully Characterized NF-FG Place for deployment	Fully Characterized NF-FG Place for deployment	Resource Orchestrator
1.6	NF-FG is split based on the placement	Fully Characterized NF-FG	Fully Characterized NF-FG sub-graphs	Controller Adaptation
1.7	The NF-FG subgraphs are handled to the corresponding Controllers	Fully Characterized NF-FG sub-graphs	Fully Characterized NF-FG sub-graphs	Controller Adaptation
1.8	The required resources are instantiated	Fully Characterized NF-FG sub-graphs	Resources for NF-FG subgraphs	Controller(s)
1.9	NF-FG management information is handled up to Service Layer	Resources for NF-FG subgraphs	NF-FG id, NF-FG management interface	Controller(s)

8.1.4.2 **Change of the NF-FG requested by the user triggering a scale-up**

Step	**Description**	**Input**	**Output**	**Actor**
2.1	Service Graph modification requested to Service Layer	N/A	NF-FG id Modified Service Graph	Service User
2.2	Based on NF definition the scaled up NF-FG containing the required changes is obtained	NF-FG id Modified Service Graph	Scaled up NF-FG	Service Layer

DRAFT

2.3	Scaled up NF-FG is handled to the Orchestration Layer	Scaled up NF-FG	Scaled up NF-FG	Service Layer
2.4	Based on the scaled up NF-FG modifications and the available resources a placement is selected	Scaled up NF-FG	Place for deployment of modifications	Resource Orchestrator
2.5	Scaled up NF-FG to deploy is handled to the Controller Adaptation	Scaled up NF-FG Place for deployment	Scaled up NF-FG	Resource Orchestrator
2.6	Scaled up NF-FG is split based on the resource modifications required	Scaled up NF-FG	Scaled up NF-FG sub-graphs	Controller Adaptation
2.7	The modified NF-FG subgraphs are handled to the corresponding Controllers	Scaled up NF-FG subgraphs	Modified NF-FG subgraphs	Controller Adaptation
2.8	The required resources are instantiated and/or modified	Scaled up NF-FG sub-graphs	Resources for scaled up NF-FG subgraphs	Controller(s)
2.9	Scaled up NF-FG management information is handled up to Service Layer	Scaled up Resources for NF-FG	Scaled up NF-FG id (constant) NF-FG management interface	Controller(s)

8.1.4.3 **Monitoring event detection triggering a scale-up**

Step	Description	Input	Output	Actor
3.1	Infrastructure notification is received and handled up to the Controller Adaptation	Infrastructure notification	NF-FG id Monitoring Notification	Controller

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

3.2	Based on NF definition necessity for a scale up is determined and requested to the Resource Orchestrator	NF-FG id Monitoring Notification NF-FG scaling information	NF-FG id Request for scale up	Controller Adaptation
3.3	Based on NF definition the scaled up NF-FG containing the required changes is obtained	NF-FG id Request for scale up	Scaled up NF-FG	Resource Orchestrator
3.4	Based on the scaled up NF-FG modifications and the available resources a placement is selected	Scaled up NF-FG	Place for deployment of modifications	Resource Orchestrator
3.5	Scaled up NF-FG to deploy is handled to the Controller Adaptation	Scaled up NF-FG Place for deployment	Scaled up NF-FG	Resource Orchestrator
3.6	Scaled up NF-FG is split based on the resource modifications required	Scaled up NF-FG	Scaled up NF-FG sub-graphs	Controller Adaptation
3.7	The modified NF-FG subgraphs are handled to the corresponding Controllers	Scaled up NF-FG subgraphs	Modified NF-FG subgraphs	Controller Adaptation
3.8	The required resources are instantiated and/or modified	Scaled up NF-FG sub-graphs	Resources for scaled up NF-FG subgraphs	Controller(s)
3.9	Scaled up NF-FG management information is handled up to Service Layer	Scaled up Resources for NF-FG	Scaled up NF-FG id (constant) NF-FG management interface	Controller(s)

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

8.1.4.4 **Monitoring event detection triggering a scale-out**

Step	Description	Input	Output	Actor
4.1	Infrastructure notification is received and handled up to the Controller Adaptation	Infrastructure notification	NF-FG id Monitoring Notification	Controller
4.2	Based on NF definition necessity for a scale out is determined and requested to the Resource Orchestrator	NF-FG id Monitoring Notification NF-FG scaling information	NF-FG id Request for scale out	Controller Adaptation
4.3	Based on NF definition the scaled out NF-FG containing the required changes is obtained	NF-FG id Request for scale out	Scaled out NF-FG	Resource Orchestrator
4.4	Based on the scaled out NF-FG modifications and the available resources a placement is selected	Scaled out NF-FG	Place for deployment of modifications	Resource Orchestrator
4.5	Scaled out NF-FG to deploy is handled to the Controller Adaptation	Scaled out NF-FG Place for deployment	Scaled out NF-FG	Resource Orchestrator
4.6	Scaled out NF-FG is split based on the resource modifications required	Scaled out NF-FG	Scaled out NF-FG sub-graphs	Controller Adaptation
4.7	The modified NF-FG subgraphs are handled to the corresponding Controllers	Scaled out NF-FG subgraphs	Modified NF-FG subgraphs	Controller Adaptation
4.8	The required resources are instantiated and/or modified	Scaled up NF-FG sub-graphs	Resources for scaled up NF-FG subgraphs	Controller(s)
4.9	Scaled out NF-FG management	Scaled out	Scaled out	Controller(s)

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

	information is handled up to Service Layer	Resources for NF-FG	NF-FG id (constant) NF-FG management interface	

8.1.4.5 **Monitoring event detection triggering a scale-in**

Step	Description	Input	Output	Actor
5.1	Infrastructure notification is received and handled up to the Controller Adaptation	Infrastructure notification	NF-FG id Monitoring Notification	Controller
5.2	Based on NF definition necessity for a scale in is determined and requested to the Resource Orchestrator	NF-FG id Monitoring Notification NF-FG scaling information	NF-FG id Request for scale in	Controller Adaptation
5.3	Based on NF definition the scaled in NF-FG containing the required changes is obtained	NF-FG id Request for scale in	Scaled in NF-FG	Resource Orchestrator
5.4	Based on the scaled in NF-FG modifications and the released resources a placement for modifications is selected	Scaled in NF-FG	Place for deployment of modifications	Resource Orchestrator
5.5	Scaled in NF-FG is handled to the Controller Adaptation	Scaled in NF-FG Place for deployment of modifications	Scaled in NF-FG	Resource Orchestrator
5.6	Scaled in NF-FG is split based on the resource modifications required	Scaled in NF-FG	Scaled in NF-FG sub-graphs	Controller Adaptation

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

5.7	The modified NF-FG subgraphs are handled to the corresponding Controllers	Scaled in NF-FG subgraphs	Modified NF-FG subgraphs	Controller Adaptation
5.8	The required resources are released and/or modified	Scaled in NF-FG sub-graphs	Resources released and/or modified	Controller(s)
5.9	Scaled in NF-FG management information is handled up to Service Layer	Scaled in Resources for NF-FG	Scaled in NF-FG id (constant) NF-FG management interface	Controller(s)

8.1.4.6 **Change of the NF-FG requested by the user triggering a scale-down**

Step	Description	Input	Output	Actor
6.1	Service Graph modification requested to Service Layer	N/A	NF-FG id Modified Service Graph	Service User
6.2	Based on NF definition the scaled down NF-FG containing the required changes is obtained	NF-FG id Modified Service Graph	Scaled down NF-FG	Service Layer
6.3	Scaled down NF-FG is handled to the Orchestration Layer	Scaled down NF-FG	Scaled down NF-FG	Service Layer
6.4	Based on the scaled down NF-FG modifications and the released resources a placement for modifications is selected	Scaled down NF-FG	Place for deployment of modifications	Resource Orchestrator
6.5	Scaled down NF-FG is handled to the Controller Adaptation	Scaled down NF-FG Place for	Scaled down NF-FG	Resource Orchestrator

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

		deployment of modifications		
6.6	Scaled down NF-FG is split based on the resource modifications required	Scaled down NF-FG	Scaled down NF-FG sub-graphs	Controller Adaptation
6.7	The modified NF-FG subgraphs are handled to the corresponding Controllers	Scaled down NF-FG subgraphs	Modified NF-FG subgraphs	Controller Adaptation
6.8	The required resources are released and/or modified	Scaled down NF-FG sub-graphs	Resources released and/or modified	Controller(s)
6.9	Scaled down NF-FG management information is handled up to Service Layer	Scaled down Resources for NF-FG	Scaled down NF-FG id (constant) NF-FG management interface	Controller(s)

8.2 Video Content Service

This use case covers a video content service. At the application layer the user is requesting the Service Graph of, e.g., a video content service, which is constructed at the Service Layer and includes the SG that is composed of a Traffic Optimizer NF and a Video Content Cache NF between two SAPs (for example access two different telecom provider networks). Depending on the specific characteristics of this service, two decompositions might be applied at the Service Layer: one for a SD video content service, and one for a HD video content service. While an SD service decomposition just uses a TOS marker NF for traffic optimization, and a simple video cache NF, the HD decomposition involves more advanced elements:

- BW accelerator transforming HD video streams into compressed streams optimally using bandwidth
- Duplication of caches close to the SAPs
- Two monitoring components close to the SAPs
- A control NF interconnected with the monitoring components which can trigger scale-in or scale-out events according to monitored metrics

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

A level diagram of this use case is shown in Figure 8.2, and will be further discussed in the next sections.

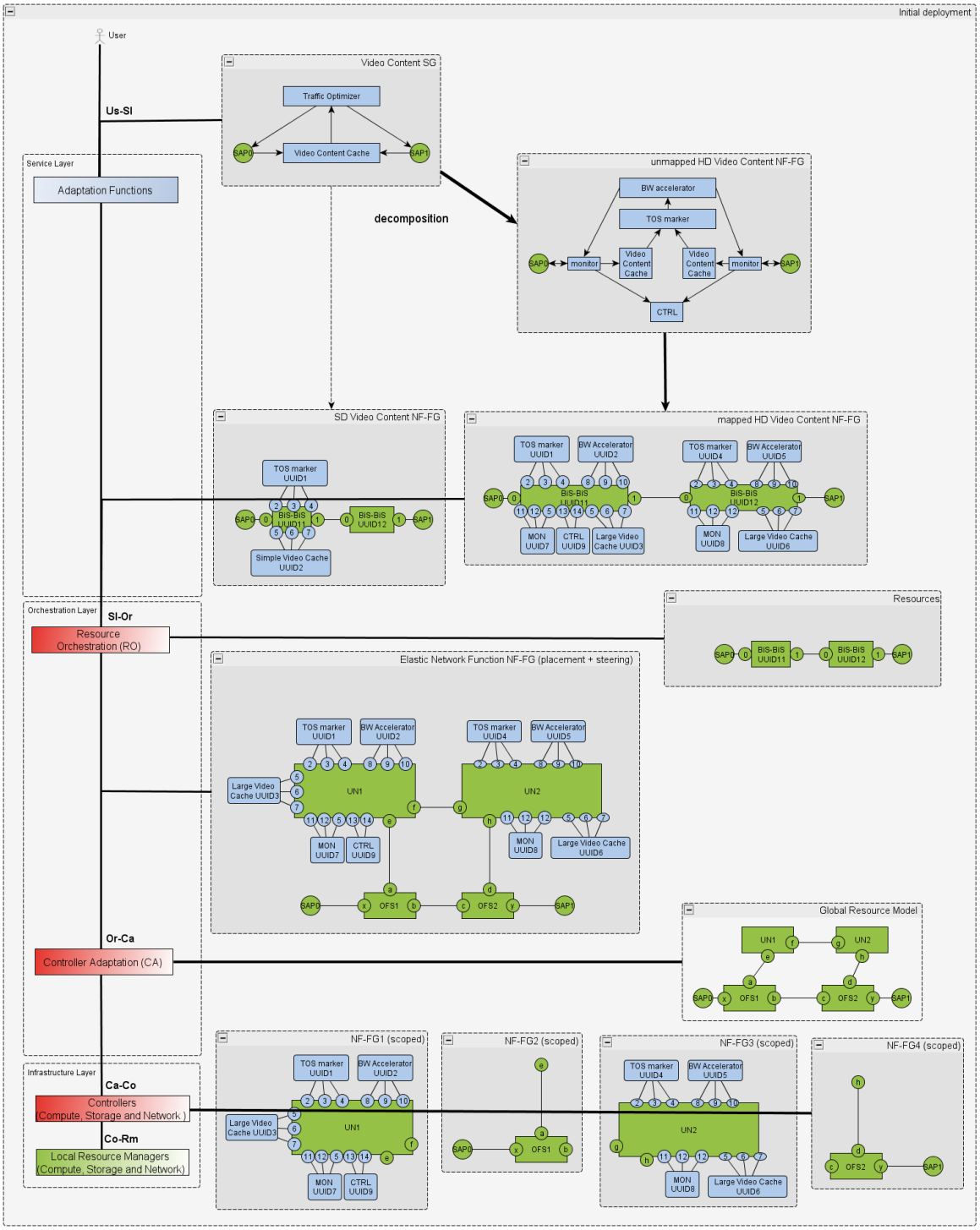

Figure 8.2: Information models and process for Video Content Service

8.2.1 Initial assumptions

In order to define the scope of the use case, the following initial assumptions are made:

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- The Infrastructure Layer bootstrapping process has been completed successfully beforehand so the infrastructure is available for the Orchestrator to deploy a Network Function Forwarding Graph (NF-FG).

- The NF-FG handled to the Infrastructure Layer for deployment is fully characterised and the VNF images to be executed are available at the Infrastructure Layer (the process for retrieving the VNF image is not included in the use case).

8.2.2 Service Graph and Network Function Forwarding graph decomposition

The high-level use case process is very similar to the Elastic Network Function use case, and therefore will not be repeated. The Service Graph initially requested by the user is decomposed by the different layers as described in Section 6.6. In the scope of this use case and based on the assumptions previously stated, the following operations take place (described in Figure 8.2):

- At the Service Layer a SG is selected representing a Video Content SG with associated KQI (e.g., HD or SD video).

- NF-FG initial decomposition performed in Service Layer by Adaptation Functions for either SD or HD video content service. Figure 8.2 focuses on the decomposition for the HD case (indicated by the thicker arrows). This decomposition introduces additional NFs: BW accelerator, TOS marker, content caches, monitoring and a CtrlApp. . In addition, it maps (Service Layer orchestration) the resulting NF-FG to the received resource model from the (virtualizer component of the) Orchestration Layer consisting of two BiS-BiS components interconnected with the SAPs. Individual NFs have multiple ports in order to enable correct interconnection with other NFs/SAPs according to the unmapped NF-FG shown at the right upper corner of the figure. The forwarding rules in the individual BiS-BiS are not depicted in the figure in order to avoid unnecessary clutter. Their configuration is very similar to the ones explained in Section 6.2.

- NF-FG placement and steering, performed in the Orchestration Layer by Resource Orchestration: the mapped NF-FG w.r.t. virtualized BiS-BiS is translated and placement to infrastructure in the the global resource model is determined. This implies that NFs of the received NF-FG connected to the first BiS-BiS (UUID11) are mapped to UN1, while the NFs connected to the second BiS-BiS (UUID12) are mapped to UN2.

- NF-FG scoping, performed in the Orchestration Layer by Controller Adaptation: the NF-FG is split according to the domain criteria producing four sub-graphs, two for the individual UNs, and two for traffic steering between the SAPs and the corresponding UNs.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

9 Conclusion

This deliverable provides a programmability framework for controlling carrier and cloud networks. As a starting point for the design of this framework, requirements were formulated per reference point in the UNIFY architecture. These were fed back into a gap analysis with respect to existing protocols, models and software in order to maximally focus programmability contributions on novelty.

Two core programmability process flows were identified and detailed with respect to their interaction between different architectural components and required data to be exchanged. The main information models crucial in this process are the Service Graph and the Network Function-Forwarding Graph. The first mainly inherits from the ETSI MANO VNFFG model, while the more novel model of the NF-FG was formalized in order to maximally benefit from the scalable, recursively layered UNIFY architecture. These models will be potentially be adapted at later stages of the project in order to support newly identified use cases, re-use the NF-FG as a model for exposing the resources between orchestration components of different domains.

The foundations of a decomposition model were described in Section 6.6. Two ways of decomposition are foreseen: static NF-IB rule-based decomposition and dynamic CtrlApp-based decomposition. In the first case decomposition rules stemming from the Service Layer are exposed to Orchestration Layers via the NF-IB component. Here, resource orchestration logic can decide to decompose NFs according to available rules in the NF-IB. In the second case the decomposition process is steered by a deployed control NF which has a direct interface to a resource orchestration component (Cf-Or). The way in which NFs are decomposed is entirely determined by the control application. Future work will consist of formalizing the rules guiding static decomposition, the way in which rules are exposed by the Service Layer, how they are stored within the NF-IB.

Abstract interface descriptions for any of the architectural reference points were identified and checked against existing work. These interfaces identify most important functionality required by different components within the architecture. Several aspects of these interfaces are experimentally supported by a set of prototypes, however future experimentation will further refine required functionality, as well as the identification of more technology-oriented characterizations of these interfaces.

Several crucial elements of orchestration functionality were identified and described in Section 6.7. The latter involved ways to address scalability of both the orchestration framework itself, as well as scaling approaches of NFs and services. In addition, challenges and frameworks for supporting dynamic processes were characterized. This paves the way for future work focusing on tight integration of monitoring points and their impact on dynamic (re-)orchestration within the developed service programmability framework. This might involve the characterization of monitoring functionality within the service definition

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

itself, for example using constructs such as the MEASUREMENT language of Section 6.7.4.5. Existing work on the virtual network embedding problem and remaining challenges were identified. Later stages of the project will focus on the scalability of these techniques and apply them in the context of already developed prototypes.

In a first stage, these prototypes will focus on simple scenario's which nevertheless encompass a wide range of programmability facets described in this framework. For this purpose, the elastic router use case was detailed and mapped to the different models and processes. Future work in the project will consist of: i) prototyping core functionality represented by this use case, and ii) extending the use case, as well as the prototype functionality towards more the more complex use cases as characterized in D2.1.

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Annex 1 Work package objectives

The table below describes the WP3 objectives as indicated in the Description of Work (DoW). The objectives are referred by number in the rest of the document as follows: OBJ-nr.

1. In this work package we will derive a generic optimization framework which supports a variety of services and service chains, infrastructures, and objective functions.

2. Our solution will jointly optimize the network and node resources, and across the network from the data centre over the access network to the network core. There are five main novelties: (1) joint optimization of network and nodes; (2) opportunity for in-network processing in "middle-boxes" and "nano data centres"; (3) time aspects and flexibility; (4) support for multi-stage mapping and chaining; (5) clustering of services and functions.

3. A major objective of the optimization is the reduction of the complexity (e.g., need for configurations), rendering the management of the deployment less labour intensive. This reduces the OPEX/CAPEX costs.

4. We will implement the service Orchestrator role of the overarching management and control: the Orchestrator implements the optimization, based on the Network Information Base (NIB).

5. The Orchestrator is redundant and resilient, and also the location of the Orchestrator itself: it may be realized close to the elements under its control to improve latency.

6. This WP will define the necessary functionality of this programmability framework and determine the primitives required. The focus is not on the language itself but on the necessary functionality and semantics of the primitives.

7. This WP will also propose a subset of the framework and functionality that will be part of the prototype. The selected elements and building blocks will be implemented and available as Service Programming, Orchestration and Optimization Prototype (SPOOPro).

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Annex 2 Related work

A.2.1 Multi-scope configuration and modelling frameworks

A.2.1.1 Remote Procedure Call frameworks

Programmability between distributed software components relies on remote procedure calls (RPC). A RPC is initiated by the client, which sends a request message to a known remote server to execute a specified procedure with supplied parameters. The remote server sends a response to the client, and the application continues its process. While the server is processing the call, the client is blocked (it waits until the server has finished processing before resuming execution), unless the client sends an asynchronous request to the server, such as an XHTTP[13] call.

There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols. Most RPC protocols are currently focusing on web services (according to W3C: a software system designed to support interoperable machine-to-machine interaction over a network.).

The two most important interaction paradigms/protocols for web services are: REST-ful service interaction, and Simple Object Access Protocol (SOAP) based service interaction. Although the first denotes rather a paradigm, while the second refers to a detailed protocol, these categories are generally agreed on[14].

SOAP is an access protocol for Web services which is based on XML and relies on other application layer protocols such as HTTP or SMTP for message transmission. Representational State Transfer (REST) [Fieldings2000] is a newer paradigm which provides a simpler method than SOAP to access Web services and tries to fix SOAP's problems. Both techniques have advantages and disadvantages which should be considered upon selection. The simplicity of REST makes it an interesting option in most of the cases. It allows different data formats while in SOAP only XML can be used. It also provides better performance and scalability. On the other hand, SOAP provides more security features than REST and it supports ACID transactions. Also a reliable messaging is provided in SOAP which is not the case is REST. In REST, clients are expected to deal with communication failures by retrying.

A.2.1.2 (Web) Interface Description Languages

An interface description language (or alternatively, interface definition language - IDL), is a specification language used to describe a software component's interface. IDLs describe an interface in a language-independent way, enabling communication between software components that do not share a language – for example, between components written in C++ and components written in Java. IDLs are commonly used in remote procedure call software.

[13] http://xhttp.org/
[14] http://kswenson.workcast.org/2005/RestVsSoap.pdf

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

In these cases the machines at either end of the "link" may be using different operating systems and computer languages. IDLs offer a bridge between the two different systems.

The Web Services Description Language[15] (WSDL pronounced "wiz'-dul") is an XML-based interface definition language that is used for describing the functionality offered by a web service [Chinnici2007]. The acronym is also used for any specific WSDL description of a web service (also referred to as a WSDL file), which provides a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns. It thus serves a purpose that corresponds roughly to that of a method signature in a programming language.

The WSDL describes services as collections of network endpoints, or ports. The abstract definitions of ports and messages are separated from their concrete use or instance, allowing the reuse of these definitions. A port is defined by associating a network address with a reusable binding, and a collection of ports defines a service. Messages are abstract descriptions of the data being exchanged, and port types are abstract collections of supported operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding, where the operations and messages are then bound to a concrete network protocol and message format. In this way, WSDL describes the public interface to the Web service.

WSDL is often used in combination with SOAP and an XML Schema to provide Web services over the Internet. A client program connecting to a Web service can read the WSDL file to determine what operations are available on the server. By accepting binding to all the HTTP request methods (not only GET and POST as in version 1.1), the WSDL 2.0 specification offers better support for RESTful web services, and is much simpler to implement.

Q-WSDL is an extension to WSDL to describe non-functional aspects or quality of service (QoS) characteristics of a web service [D'Ambrogio2006]. These characteristics include performance, reliability, availability, security, etc.

IBM has proposed a standard for SLA documents referred as WSLA framework [Keller2003]. It is based on XML and provides machine-readable SLAs for Web services using the WSDL service descriptions. However, it is not limited to only WSDL and can be extended to deal with other service-based technologies. WSLA accommodate SLA structure in 3 sections:

- Parties: This section determines all the contractual parties.
- Service Description: The characteristics of the service and its parameters are described in this section.

[15] Most of the paragraphs on WSDL are taken from the Wikipedia webpage (http://en.wikipedia.org/wiki/Web_Services_Description_Language)

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- Obligations: All the guarantees and the restrictions imposed on SLA parameters should be identified in this section.

BPEL is a XML-based language to be used for specifying business process behavior based on Web services [Jordan2007]. It is mainly influenced by WSDL and its process model is a layer on top of service model defined in WSDL. This language is used to define an assembly of a set of services (Web services) for composite service description. The BPEL structure consists of two main sections: i) Partner Link and ii) BPEL Process. The former (together with the corresponding WSDL interface) is used to interact with the BPEL core process and the outside world. In other words, Partner Link is a logical link from BPEL process to another Web service or the client who invoked the service.

Note that the composition of QoS requirements, capabilities, measurements and SLAs is potentially very complex and although BPEL is based on WSDL, the extensions such as Q-WSDL might not be useful in description of non-functional properties in composite services. These non-functional properties of a composite service should be estimated from the information of the partner services. Works such as [Christos2009] and [Mukherjee2008] provides extensions to BPEL to specify QoS parameters as well. Another interesting feature in BPEL is the exception/fault handling. BPEL introduces systematic mechanisms for dealing with exceptions and processing faults. This capability allows for switching to the next best solution when the originally selected candidate is unavailable.

In order to deal with faults, we need to determine possible faults that might occur in a service component. Then we need to set up a fault handler for each of them in the BPEL process.

Unified Service Description language (USDL) has been introduced to capture the business and operational aspects of services and align them with the technical perspective [Cardoso2010]. Some of the services that are addressed by USDL are human services (e.g., consultancy), business services (e.g. purchase order requisition), software services (e.g., WSDL and RESTful services), infrastructure services (e.g., CPU and storage services), etc. In USDL, the business description of the services is derived from the E3Service ontology [Baida2005], the PAS 1018 [Mörschel2001] and the taxonomy introduced by O'Sullivan [O'Sullivan2006] (it can represent the non-functional properties of services such as availability, payment, price, discounts, obligations, rights, penalties, trust, security and quality.). The technical description is influenced by WSDL, WSMO[16] and OWL-S[17].

Using USDL, the service description includes information such as: i) pricing ii) legal iii) service provider iv) interaction methods and v) service level agreements. In USDL, the services

[16] http://www.wsmo.org/
[17] http://www.ai.sri.com/~daml/services/owl-s/1.2/overview/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

described as a black box with no information about internal interactions and components connections which is similar to WSDL.

Linked USDL[18] is an effort to promote the use of USDL on Web. Linked USDL remodels USDL based on Linked Data[19] principles. The existing USDL specifications are remodeled as RDF vocabulary.

Semantic (Web) Modelling frameworks

In computer science and information science, ontologies are used to formally represent knowledge within a domain. An ontology is defined as a formal, explicit specification of a shared conceptualization. It provides a common vocabulary to denote the types, properties and interrelationships of concepts in a domain. Ontology languages are formal languages used to construct ontologies. They allow the encoding of knowledge about specific domains and often include reasoning rules that support the processing of that knowledge. Ontology languages are usually declarative languages, are almost always generalizations of frame languages, and are commonly based on either first-order logic or on description logic.

In order to increase the quality of the interaction between distributed software components, not only the syntax and format of the interface might be formalized through IDLs, but also the semantics of different elements of these interfaces might be documented through ontology frameworks. Adding semantics to interface description of web services is referred as the semantic web.

Resource Description Framework (RDF)[20] is a model used for conceptual description of information in Web. It is machine-readable and is written in XML (RDF/XML). Therefore, different types of computers based on different operating systems can easily exchange RDF information. This method decomposes any type of information into small pieces with some rules about the meaning of those pieces. It is similar to other conceptual modelling approaches such as entity-relationship in the sense that it expresses a fact about (Web) resources using a triple in the form of (Subject, Predicate, Object).

Web Ontology Language (OWL) is a Semantic Web language based on RDF/XML [McGuinness2004]. It is used by applications to process the content of information. It provides more facilities for expressing meaning and semantics than RDF.

There are several other ontology languages used to formally encode the ontology. Common Algebraic Specification Language is a de facto standard in the area of software specifications developed within IFIP working group 1.3 “Foundations of System Specifications” [Astesiano2002]. MOF[21] and UML[22] are two standards of the Object Management Group (OMG).

[18] http://www.linked-usdl.org/
[19] http://linkeddata.org/
[20] http://www.w3.org/RDF/
[21] http://www.omg.org/mof/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

OntoUML is a UML profile which is used for conceptual modelling of ontologies. DOGMA [Jarrar2002], SBVR[23] and IDEF5[24] are other examples of such languages.

OWL-S is an ontology based on Web Ontology Language (OWL) used for description of Semantic Web Services. It provides the automatic discovery, composition, invoking and monitoring of Web services for users and software agents.

Unlike languages such as WSDL and USDL, OWL-S enables definition of composite services. The notion of 'process' is the building block of process model in this language. It consists of both atomic and composite processes. Each process in OWL-S has inputs, outputs, pre-conditions and conditional effects.

- Preconditions: a set of conditions which should hold before invoking the service
- Inputs: a set of inputs required for service invocation
- Outputs: Results of service request
- Effects: a set of statement which should be true if the service is successful. E.g. package being delivered

WSDL-S is an attempt to add semantic capabilities into WSDL [Akkiraju2005]. Semantics are used to improve discovery and compositions of Web services.

A.2.1.3 SNMP

Simple Network Management Protocol (SNMP) is one of the most widely used network management protocols. SNMPv1 [Case1990] is already declared historical and is non-recommended for use due to its lack of security. Later SNMPv2 [Case1996] added functional enhancements while SNMPv3 [Harrington2002] added security mechanisms and also revised the architecture according to fully modular design.

The basic framework of SNMP, however, did not change over the versions and comprises of managed nodes (agents), management applications and the management protocol. Additionally, SNMP uses a protocol-independent data definition language (ANS.1) for the information repository containing management information definitions (Management Information Base or MIB).

SNMP follows a manager – agent paradigm, in which the agent is responsible for providing access to its local managed objects (MIB) that reflects the resources and activity of the node. The communication pattern between the manager-agent is either transaction oriented (request-response pairs) or the agent can send unsolicited notifications to the manager.

[22] http://www.uml.org/
[23] http://www.omg.org/spec/SBVR/
[24] http://www.idef.com/idef5.htm

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

SNMP is widely used for monitoring purposes for both fault and performance management. There are many counters and state variables defined in MIBs which allow device independent statistics collection or state monitoring all over the network. In our UNIFY environment, SNMP could provide basic monitoring services.

For configuration management and programmability, there are more advanced protocols available, for example NETCONF.

A.2.1.4 NETCONF/YANG

The widely used IETF standard for network management, namely SNMP, dates back to the 1980s. By the beginning of 2000s, it had clearly turned out that SNMP was mainly used only for monitoring network elements and network operators preferred CLI-based configuration methods. In order to define more effective methods for network configuration which can be widely accepted by network operators and device vendors as well, a new working group (NETCONF) had been established in IETF. The work resulted in the definition of a novel protocol. NETCONF (Network Configuration Protocol) is a network management protocol defined by IETF in RFC 4741 [Enns2006] and later revised in RFC 6241 [Enns2011]. In contrast to SNMP, from TMN FCAPS (Telecommunications Management Network – Fault Configuration Accounting Performance Security) model, NETCONF focuses on the configuration part. By now the major equipment vendors support NETCONF in their devices (e.g., in switches, routers, etc.).

NETCONF provides simple mechanisms to manage network devices, retrieve/upload/manipulate configuration data of network elements. The protocol allows the devices to expose formal application programming interfaces (APIs) and applications can use these to send and retrieve full or partial sets of configuration data. NETCONF is based on a remote procedure call (RPC) paradigm where a client encodes an RPC in XML (eXtensible Markup Language) and sends it to a server via a secure transport channel, while the server's response is also encoded into XML. Besides the protocol messages, the encoding of configuration data is based on XML as well[25].

According to NETCONF terminology, the server or agent is running on the network device while the client is an application or a script which is typically part of a network management system. The communication of the peers is based on a simple RPC-based mechanism. NETCONF can be partitioned into four distinct layers as it is shown in Figure 9.1.

[25] There is ongoing work to have a JSON data model: http://datatracker.ietf.org/doc/draft-ietf-netmod-yang-json/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

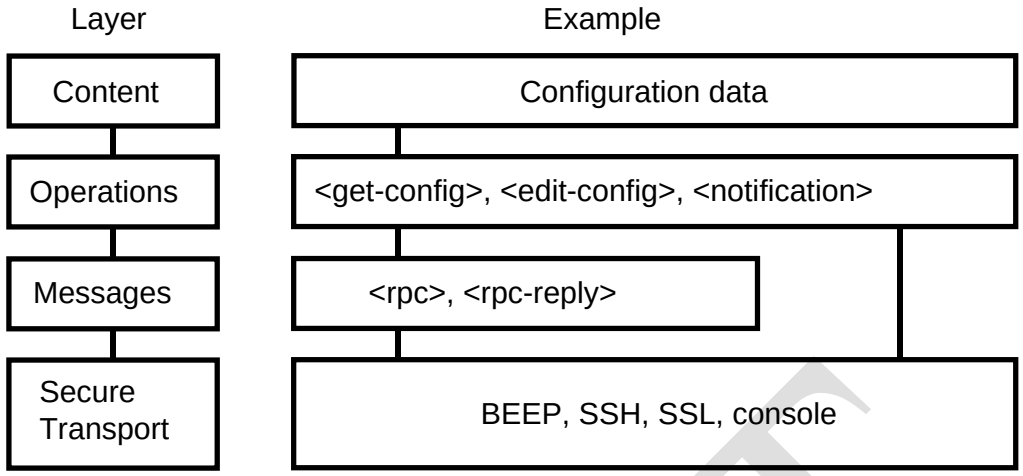

Figure 9.1: NETCONF protocol layers

The roles of the layers from the bottom to the top are the following:

- **Secure Transport** layer provides communication path between the client and the server. The connection-oriented transport mechanism must provide persistent connection between the peers, and reliable, sequenced data delivery. Authentication, data integrity, confidentiality, and replay protection are also required. SSH is mandatory to implement.

- **Messages** layer provides a transport-independent framing mechanism for encoding RPCs (requests and responses) and notifications.

- **Operations** layer defines a set of base protocol operations invoked as RPC methods to retrieve and edit the configuration data. The following operations are included: <get>, <get-config>, <edit-config>, <copy-config>, <delete-config>, <lock>, <unlock>, <close-session>, <kill-session>.

- **Content** layer consists of configuration and notification data. This layer is out of scope of NETCONF standard, data models are defined in separate documents.

The basic functionality of NETCONF can be extended by the definition of NETCONF capabilities. These additional features are communicated between the client and server during the session setup phase. Some capabilities have been defined in RFCs such as, subscribing and receiving asynchronous event notifications [Chisholm2008], partial locking of running configuration [Lengyel2009], monitoring the NETCONF protocol and discovering/retrieving data models supported by a NETCONF server [Scott2010].

In order to develop a human-friendly modelling language for defining the semantics of operational data, configuration data, operations and notifications, a dedicated working group (NETMOD) proposed **YANG** in RFC 6020 [Bjorklund2010]. YANG is a data modelling language for NETCONF which can be used to model both configuration and state data of network elements. It also supports the description of event notifications which can be generated by devices, and makes it possible to define the signature of RPCs that can be invoked on devices

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

via NETCONF. YANG covers two layers of NETCONF protocol, namely Operations and Content layers (see Figure 9.1).

YANG is a modular language with a number of built-in data types. Common YANG data types are defined in RFC 6991 [Schoenwaelder2013] (it obsoletes RFC 6021 [Schoenwaelder2010]) while additional types can be derived from built-in ones and more complex reusable data structures can be created by groupings. Data structures are represented in XML tree format and XPATH expressions can be used to define constraints on the elements.

There are available implementations of NETCONF as well as YANG compilers/builders/validators. From our point of view, the following open source tools could be relevant:

- **OpenYuma**[26] is an open source NETCONF implementation and YANG compiler which is a fork of the de facto standard Yuma project since it went proprietary (YumaPro).
- **libnetconf** [27] is an open source C implementation of NETCONF for GNU/Linux.
- **pyang**[28] is an extensible YANG validator and converter written in Python.

A.2.2 Infrastructure modelling frameworks

A.2.2.1 Common Information Model

The Common Information Model (CIM)[29] is an open standard defined by the Distributed Management Task Force (DMTF). CIM provides a common definition of management information for systems, networks, applications and services, and allows for vendor extensions. CIM's common definitions enable vendors to exchange semantically rich management information between systems throughout the network.

CIM is composed of a Specification and a Schema. The Schema provides the actual model descriptions, while the Specification defines the details for integration with other management models. The CIM Schemas are based on the Managed Object Format (MOF), which is based on the CIM Metamodel. Since the CIM Metamodel is based on a subset of UML, all CIM Schemas can be modelled using UML.

The CIM Specifications and generic Schemas serve as a basis for more specific implementations of CIM like The Cloud Management Initiative (CLOUD)[30], that addresses the management of cloud systems or DMTF's Virtualization Management Standard (VMAN).

Recently DMTF started a new initiative called Network Management Initiative (NETMAN)[31], that is focused on the definition of an integrated set of standards for management of

[26] https://github.com/OpenClovis/OpenYuma
[27] http://libnetconf.googlecode.com/git/doc/doxygen/html/index.html
[28] https://code.google.com/p/pyang/
[29] http://dmtf.org/standards/cim
[30] http://dmtf.org/standards/cloud
[31] http://www.dmtf.org/standards/netman

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

physical, virtual, application-centric and software defined networks. The initiative is supposed to work close with NFV in ETSI[32].

A.2.2.2 Directory-Enabled Networking(-NG)

Directory refers to a special purpose database which stores the information about the nodes (devices) in a network. In directory-enabled networks, network users and applications interact with the network devices and services in a controlled manner to provide repeatable and predictable services. Directory-Enabled Networking (DEN) defines an object-oriented information model based on the Common Information Model. The objective is to provide consistent modelling of network elements and services across heterogeneous directories.

This information model which abstracts the knowledge about the network users, applications, devices and their interactions consists of three parts:

- 6 base class hierarchies to represent network elements and services
- An extensible schema based on inheritance and aggregation used for application-specific properties
- Mechanisms for establishing relationship among object instances

Using DEN, network applications can be designed which provide automatically the proper level of resource access to the users in case of change in the user's status (position, location, etc).

The information model in DEN-ng (Directory Enabled Network next generation) originates from the network management areas [Strassner2002]. This model is closely related with OSS/BSS environments in telecommunications. It enables a common way to represent the management information which is used for consideration of policies. The information model consists of a single root class with three subclasses: i) Entity ii) Value and iii) MetaData. There are special classes (Context, PolicyConcept) which are used for handling policies. An interesting feature of this model is the capability of reusing the created components. Also its extendibility enables coverage of all aspects of a network in particular, network virtualization technologies.

A.2.2.3 Network Description Language

NDL was developed at University of Amsterdam to model network infrastructure in a technology independent manner [VanderHam2006]. To this end, it adopts semantic web for its schemas using RDF in particular. Using NDL, basic network elements such as Devices, Interfaces and Links and also communication flows between different network layers can be defined. It also enables definition of Network domains with different administrators and policies. An interesting feature of NDL is the support for distribution of information which

[32] http://www.etsi.org/technologies-clusters/technologies/nfv

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

means that different network operators can define their networks using NDL and publish them (on the Web). These can be gathered to generate a global description of the network.

In this model, network information is categorized in i) network topologies ii) layers iii) device configurations iv) capabilities and v) network domains. It is worth mentioning that NDL is a network-centric model and does not provide models to describe computing infrastructure.

Other models such as Network Markup Language, NOVI and GEYSERS information model are influenced by NDL.

A.2.2.4 RSpec

RSpec[33] is the ProtoGENI mechanism for advertising, requesting, and describing the resources used by experimenters. ProtoGENI derived many of the basic principles from their previous format used in assign, the network mapper used in the Emulab testbed control framework[34] . The format has the advantage of having a fairly simple structure and allows us to draw upon 8 years of experience in dealing with resources inside of Emulab.

The RSpec has three distinct purposes and therefore we have divided RSpecs up into three different closely-related languages to address each of these purposes in particular.

- Advertisements are used to describe the resources available on a Component Manager. They contain information used by clients to choose resources (components). Other kinds of information (MAC addresses, hostnames, etc.) which are not used to select resources should not be in the Advertisement.

- Requests specify which resources a client is selecting from Component Managers. They contain a (perhaps incomplete) mapping between physical components and abstract nodes and links.

- Manifests provide useful information about the slivers actually allocated by a Component Manager to a client. This involves information that may not be known until the sliver is actually created (i.e. dynamically assigned IP addresses, hostnames), or additional configuration options provided to a client.

The following components are key aspects of RSpec:

- Identifying resources: All nodes and links are identified by URN
- Nodes
- Node have types
- Geographic information can be attached

[33] Most of the description this section is taken from the homepage of the project http://www.protogeni.net/ProtoGeni/wiki/RSpec
[34] http://www.emulab.net/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- Virtualization technology is included as a field
- Descriptions of additional optional facilities provided by the component manager (e.g. login protocols, installation of software, commands to run) can be supplied
- Links
- Links are point-to-point: LANs and other "full connectivity" environments (such at the Internet), a "LAN node" is created, and all members are linked to it.
- Links have bandwidth, a type, etc.
- Links endpoints reference Interfaces on Nodes
- Links have LinkTypes which describe how experimental traffic is to be encapsulated and what layer(s) are supported for that traffic.
- Interfaces
- Endpoint of a link
- Named by node, plus an opaque interface name
- *In progress*: Interfaces will be first-class entities, declared as part of the component they belong to
- Metadata, incl. a "valid until" field and a "generated" time
- ExternalReferences: A mechanism by which CMs can describe how their components connect with the components in other CMs.

A.2.2.5 NDL-OWL

NDL-OWL extends NDL by using Web Ontology Language (OWL) [Baldine2010]. Networks topology, layers, utilities and technologies (PC, Ethernet, fiber switch), cloud computing, virtual machines and service procedures and protocols can be modelled using their ontology.

They extend NDL with more virtualization and service description features to describe their infrastructure. Such description can be used by both client and management software. Clients use them to describe requests and management software uses them to map the requests to the infrastructure.

NDL-OWL provides a flexible semantic query-based programming approach which enables implementation of resource allocation, path computation and topology embedding. There are 5 models considered in their ontology:

- Substrate models: Used to describe resource and topology.
- Delegation models: Used to advertise an aggregate’s resources and services externally.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- Request models: User resource requests are described by this model.
- Reservation models: Used by ORCA (a control framework to provision virtual network systems) brokers to return resource tickets to SM (Slice Manager) controller.
- Manifest models: Access method, state and post-configuration information of the reserved slivers (virtual resources) are described by this model.

A.2.2.6 Network Markup Language

The Network Markup Language (NML) defines a standard schema to exchange network topology information [VanderHam2013]. NML is a standard at the Open Grid Forum (OGF) and tries to combine several initiatives (cNIS, NDL, UNIS, VDXL, OGF) to a single single standard describing its network topology, its capabilities and its configuration.

The following is the set of requirements set for the NML specification:

- Network Infrastructure Agnostic – the NML schema must be not depend on specific network infrastructure.
- Encoding Agnostic – the NML schema must be able to be easily transformed into different wire encodings (e.g. XML, OWL[OWL], etc.).
- Extensible – the NML schema must be extensible to support new network infrastructure types as well as the needs of different applications.
- Concise – the NML abstraction should represent core network definitions (e.g. node, port, link) enough to model the basic network infrastructure. Complex structures should be provided as extensions.
- Scalable – NML must be able to deal with heterogeneous, dynamically growing networks.
- Multi-layer and multi-domain – NML must fit into the applications running in the multi-domain networks, aware of multi-layer structure

A.2.2.7 Infrastructure and Networking Description Language

The Infrastructure and Networking Description Language (INDL) has the aim to provide a technology independent model of computing infrastructure [Ghijsen2012]. INDL can also be utilized to describe virtualized resources and the services offered by the infrastructure. Despite using INDL only it can be well connected with NML to describe compute and network resources altogether.

INDL can be compared to NDL-OWL, since both are built on top of Semantic-Web technologies. While INDL leverages the latest conclusions of OGF's NML-WG to describe networking resources, NDL-OWL uses the slightly older NDL. NDL-OWL is also more detailed on

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

the resource description capabilities, while INDL keeps the resource information rather simple.

A.2.3 Network Programming and Control

The lower layer interface toward the networking elements is generally referred as southbound interface (SBI). During the past decades, several interfaces and protocols have been standardized to control, manage, or configure the network nodes. These SBI protocols address different aspects of the operation. For example, OpenFlow separates the data and control plane of the network and defines a protocol to control the forwarding elements, i.e. the OpenFlow capable switches. This section briefly summarizes the most relevant SBI protocols and tools.

A.2.3.1 Node-level programming and Control

A.2.3.1.1 OpenFlow

The OpenFlow protocol [McKeown2008] specifies an API in which network controllers are able to program how packets must be handled by an OpenFlow-capable switch. It is based on the fact that most modern routers/switches contain a proprietary FIB (Forwarding Information Base) which is implemented in the forwarding hardware using TCAMs (Ternary Content Addressable Memory). OpenFlow provides the concept of a FlowTable that is an abstraction of the FIB. In addition to this, it provides a protocol to program the FIB via "adding/deleting/modifying" entries in the FlowTable.

An entry in the FlowTable consists of: (1) a set of packet fields to match with incoming packets (grouped as flows), (2) "statistics" which keep track of matching packets per flow, and (3) "actions" which define how packets should be processed. When a packet arrives at an OpenFlow switch, the header of the packet is compared with the flow of the Flow Entries in the FlowTable. If a match is found, the actions specified in the matching entry are performed. If no match is found, the packet (a part thereof) is forwarded to the controller. Thereafter, the controller makes a decision on how to handle the packet. It may return the packet to the switch indicating the forwarding port, or it may add a Flow Entry directing the switch on how to forward packets with the same flow.

Stanford University released the first versions of OpenFlow known as version 1.0 [OF1.0] and 1.1 [OF1.1] in year 2009 and 2011 respectively. Industrial players such as Deutsche Telekom, Google, Microsoft, Verizon, and Yahoo! have then formed ONF (Open Networking Foundation) to standardize and release the next versions of OpenFlow versions according to their needs and demands. Since then, many versions (1.2, 1.3.0, 1.3.1, 1.3.2, 1.3.3 and 1.4.0) have been released publicly[35].

[35] https://www.opennetworking.org/sdn-resources/onf-specifications

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

For enabling widespread deployment in a production and carrier environments, new OpenFlow versions provide additional functionalities. Table 9.1 depicts functionalities available in different OpenFlow versions.

Multiple table support, group tables, multiprotocol label switching (MPLS), and controller-connection failure support are available from OpenFlow 1.1. The OpenFlow spec 1.2 and higher also include support for IPv6, and controller redundancy (role-change). Furthermore, the per-flow meter, PBB (Provider Backbone Bridge), and expendedIPv6 support are available from OpenFlow version 1.3, and optical port and PBB-UCA (Use Customer Address) support are available in 1.3.1.

Table 9.1: Functionalities available in different OpenFlow versions

	Main functionalities	version 1.0	version 1.1	version 1.2	version 1.3	version 1.3.1, 1.3.2, 1.3.3, 1.4
1	Single FlowTable support	X	X	X	X	X
2	Slicing	X	X	X	X	X
3	Normal stack (Ethernet switching mode)	X	X	X	X	X
4	Matching support	X	X (more fields are added see Table 2)	X (more fields are added see Table 2)	X (more fields are added see Table 2)	X (more fields are added see Table 2)
5	Queue support	X	X	X	X	X
6	Statistics	X	X (more fields are added see Table 3)	X (more fields are added see Table 3)	X (more fields are added see Table 3)	X
7	Multiple FlowTable support		X	X	X	X
8	GroupTable support		X	X	X	X
9	Tags: MPLS		X	X	X	X
10	Controller connection Failure		X	X	X	X
11	Basic IPv6 support			X	X	X
12	Controller role-change mechanism			X	X	X
13	Per-flow meters				X	X
14	PBB (Provider Backbone Bridge) tagging				X	X
15	Expanded IPv6 support				X	X
16	Optical Port support					X
17	PBB-UCA support					X

Besides the growing number of functionalities in newer OpenFlow versions, matching functionality in the Flow-Entries has also been extended (Table 9.2). Starting from Ethernet, IP and transport layer packets matching in OpenFlow 1.0, MPLS headers and metadata can be matched from 1.1, as well as IPV6 headers (source, destination addresses) matching is

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

supported since 1.2. Furthermore, in OpenFlow version 1.3, the IPV6-extensible and PBB headers can be matched against the flow and in the latest versions, PBB-UCA headers can be matched with the flow of Flow-Entries.

Table 9.2: Flow Matching Fields in different OpenFlow versions

	Matching Fields	version 1.0	version 1.1	version 1.2	version 1.3.0	version 1.3.1, 1.3.2, 1.3.3, 1.4
1	Ingress port	X	X	X	X	X
2	Ethernet fields (src,dst, type, vlan)	X	X	X	X	X
3	Internet protocol fields (IP src, dst, protocol, TOS)	X	X	X	X	X
4	Transport layer fields (TCP src port, TCP dst port)	X	X	X	X	X
5	Metadata		X	X	X	X
6	MPLS field (Label, Traffic class)		X	X	X	X
7	IPV6 (src,dst, label)			X	X	X
8	IPV6 extensible header				X	X
9	PBB headers				X	X
10	PBB-UCA headers					X

Several statistics can be gathered in the controller via the OpenFlow interface towards switches. Table 9.3 shows the statistics fields in different versions. It shows that the per-table, per-port, per flow-entry, per queue statistics can be gather in OpenFlow version 1.0. In version 1.1, this is extended with statistics about Group-Entry and action-bucket statistics. In the latest OpenFlow versions, flow-meter and Flow meter-band statistics can be gathered.

Table 9.3: Statistics Fields in different versions of OpenFlow

	Statistics Fields	version 1.0	version 1.1	version 1.2	version 1.3.0	version 1.3.1, 1.3.2, 1.3.3, 1.4
1	Table statistics	X	X	X	X	X
2	Port statistics	X	X	X	X	X
3	Flow-Entry statistics	X	X	X	X	X
4	Queue statistics	X	X	X	X	X
5	Group-Entry statistics		X	X	X	X
6	Action-bucket statistics		X	X	X	X
7	Flow-meter statistics				X	X
8	Flow meter band statistics				X	X

DRAFT

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Despite the many extensions to OpenFlow specification standards, a couple of functionalities are currently still missing:

- Connectivity verification mechanisms on a per link or per-flow level, such as BFD (Bidirectional Forwarding Detection)[Katz2010]

- Quality of service (QoS) support is limited, for example creation or configuration of output queues is only possible through implementation/vendor-specific extensions or protocols such as OVS-DB (Open vSwitch Database Management Protocol)

A.2.3.1.2 OF-Config

The OF-Config protocol[36] was defined in order to cover management and configuration of OpenFlow datapath elements which are not covered within OpenFlow (e.g., the setup of control channel towards the OpenFlow controller). Figure 9.2 illustrates how an OpenFlow-capable switch is hosting one or more logical OpenFlow switches. The logical switches are the actual network elements controlled by OpenFlow controllers. An OF Configuration Point is a service that interacts with the OF-Config protocol with the OF Capable switch.

The OF-Config protocol is based on NETCONF and focuses on the definition of schema to ensure consistent representation of configuration elements in the switch. The OF-Config specification was given as a YANG model and includes the following functionality:

- Assignment of OF controllers
- Configuration of queues and ports
- Ability to remotely change aspects of ports (up/down)
- Configuration of certificates for secure communication towards OF Controllers
- Discovery of capabilities of OF logical switch
- Configuration of tunnel types such as IP-in-GRE, NV-GRE and VxLAN

[36] https://www.opennetworking.org/images/stories/downloads/of-config/of-config-1.1.pdf

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

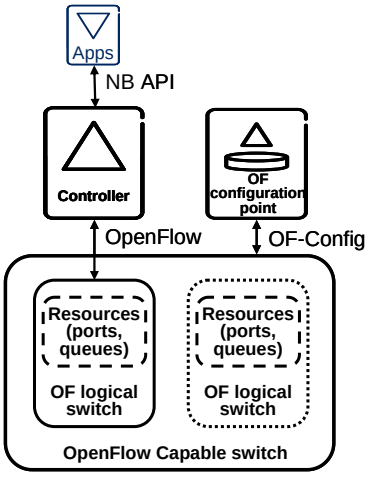

Figure 9.2: ONFs SDN architecture including OpenFlow and OF-Config

A.2.3.1.3 OVSDB

The Open vSwitch Database (OVSDB) management protocol was defined as part of the Open vSwitch (OVS)[37] project. Open vSwitch is an open-source software switch, designed to be used as a virtual switch in virtualized server environments, and it is open to programmatic extension and control using OpenFlow and OVSDB. The architecture of OVS is shown in the Figure 9.3.

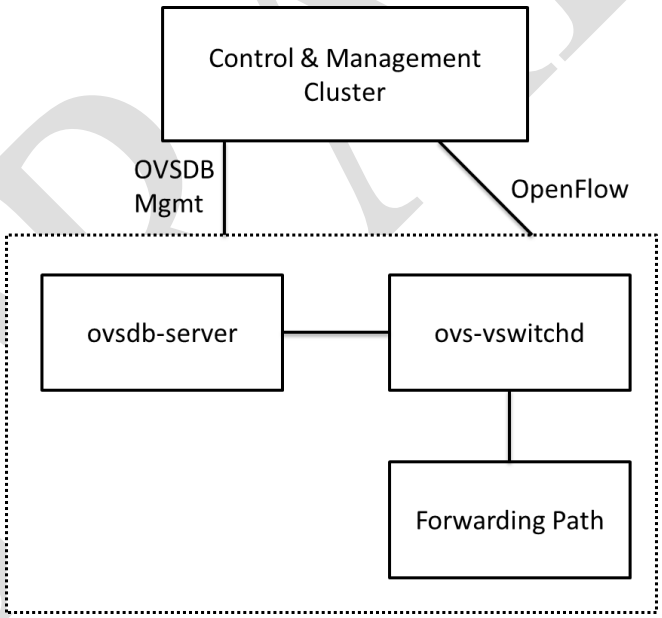

Figure 9.3: OVS architecture

Regarding OVSDB there are two main parts that has to be considered:

- OVSDB protocol, detailed in [Pfaff2013] (The Open vSwitch Database Management Protocol (informational, released December 2013)), is used for interacting with the configuration database for the purposes of managing and configuring Open vSwitch instances,

[37] http://openvswitch.org/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

while it also provides means for discovering the schema in use, but does not define the contents. The OVSDB management protocol uses JSON [Crockford2006] for its wire format and is based on JSON-RPC version 1.038 .

- OVS schema, detailed in Open vSwitch Manual (ovs-vswitchd.conf.db(5)), describes the configuration tables and the relation between them. Figure 9.4 shows the main tables considered and Figure 9.5 shows the complete set of tables included in the schema, representing also the detail of the relationships between them. Edges are labelled with their column names, followed by a constraint on the number of allowed values: ? for zero or one, * for zero or more, + for one or more.

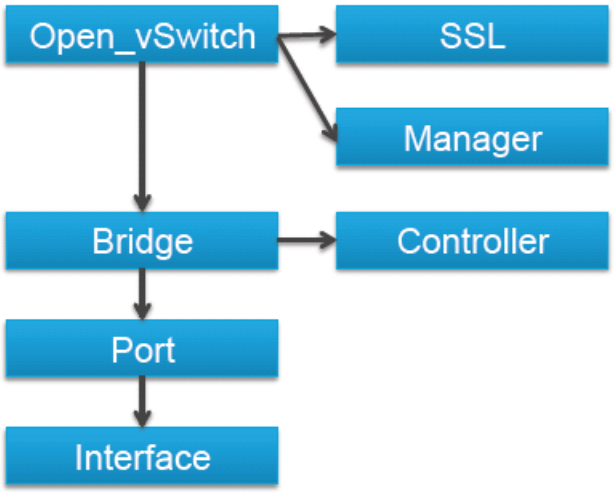

Figure 9.4: OVS main configuration tables

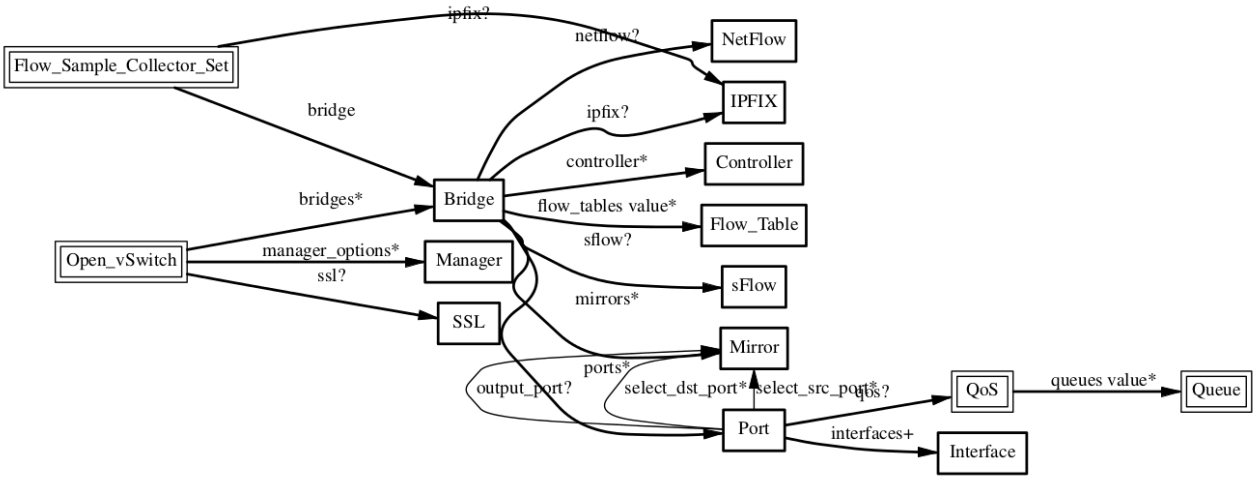

Figure 9.5: Detailed OVS schema with table relations

[38] http://www.jsonrpc.org/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Also to be considered in the scope of UNIFY, OVSDB Integration is a project for OpenDaylight that will implement the Open vSwitch Database management protocol, allowing southbound configuration of vSwitches. The project consists of a library, along with various plugin usages.

The OpenDaylight's OVSDBsouthbound plugin is made up of one or more OSGi bundles addressing different services / functionalities:

- Connection Service - Based on Netty
- Network Configuration Service
- Bidirectional JSON-RPC Library
- OVSDB Schema definitions and Object mappers
- Overlay Tunnel management
- OVSDB to OpenFlow plugin mapping service
- Inventory Service

A.2.3.1.4 Click Modular Router

Click is a software router framework for *nix operating systems focusing on data-plane logic[39]. Click elements are C++ classes implementing a given set of methods defining how packets are received, and what actions should be performed on them. The modularity and extensible nature of the architecture enables building of extensible and complex software routers, forwarders or packet-processing functions with little effort. Click already provides a wide range of packet-processing components such as NAT, Classifier, Encryption/Decryption, but can be easily extended by implementing new C++ classes. Handler interfaces enable external configuration of these components through tcp or unix control sockets. As such, a click packet-processing element is similar to a simple, atomic Network Function.

More advanced/composed Network Functions such as a DHCP or CDN server functionality are typically not implemented as atomic Click elements, but either as Click script (interconnecting multiple simpler components) or as regular *nix OS daemon/process. The latter can possibly interact with regular Click elements through special socket interfaces crossing the regular network stack. Click scripts can thus be a good starting point to describe and implement composed Network Functions.

In ClickOS [Ahmed2012] a Click configuration is deployed in a light-weight virtual, rather than a physical machine. The latter enables to deploy multiple Click-driven Network Functions on the same physical machine. Such a setup is a starting point to enable the implementation of Service Function Graphs involving multiple physical machines based on NFs implemented by Click-scripts.

[39] http://www.read.cs.ucla.edu/click/click

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

A.2.3.1.5 HILTI

HILTI (High-level Intermediary Language for Traffic Inspection) is an environment for traffic analysis which provides high-level data structures, control flow primitives, concurrency support and a secure memory model [Sommer2012]. It can be seen as a middle-layer between the operating system and a host application which tries to close the gap between high-level description of analysis and low-level detailed implementation. It consists of two parts: i) an abstract machine model for networking domain and ii) a compilation strategy which is used to convert the programs for the abstract machine to an optimized code for a given platform.

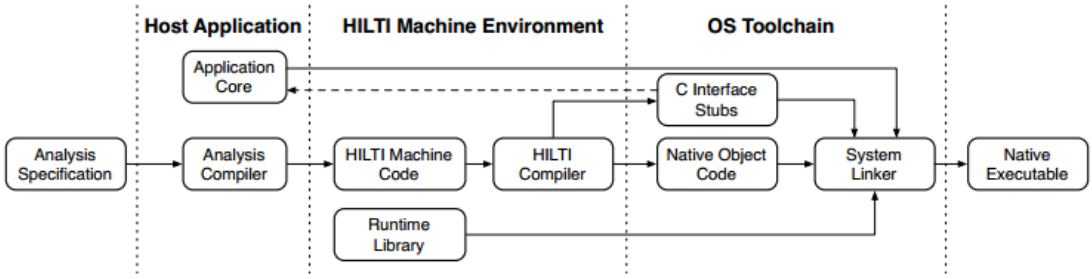

Figure 9.6: Workflow of using HILTI from [Sommer2012]

Figure 9.6 illustrates the workflow for using HILTI. An application usually has an analysis specification which should be deployed (e.g. a set of filtering rules for firewall). This specification should be converted to HILTI code through a custom analysis compiler provided by the application. The compiler generates a set of C stubs for the application to interface with the compiled code. The system linker combines the resulting code, stubs and the application to a single program.

The instruction set of HILTI is built on register-based assembler languages. In addition to standard atomic types such as integers, floating-point, etc, HILTI provides domain-specific types such as IP addresses, transport-layer ports and timestamp types. These types can enable optimization and data flow analyses.

HILTI has an extensive C API which enables access to all of its data types. The exception handling and thread scheduling between applications and HILTI is integrated through the C interface. There exists a Python based AST (Abstract Syntax tree) interface used for building HILTI programs in memory. This API can be used by applications to compile their analysis specifications to HILTI code

A.2.3.1.6 ForCES

Forwarding and Control Element Separation (ForCES) is an approach to network programmability, where the focus is on a clear separation of forwarding functions of a switch/router from its control functions [Doria2010]. The work on ForCES started in early 2000s, when the relatively powerful network processing units (NPU) found widespread use in networking elements like IP routers. The need for having a standard framework for

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

programming such NPUs has contributed to the ForCES initiation. Currently, ForCES is an active working group of IETF.

ForCES specifies a modular architectural framework for structuring separated forwarding elements (FEs) and control elements (CE) in a networking element (NE), as depicted in Figure 9.7. The ForCES architecture defines several reference points between the architectural elements, including between CE and FE. Furthermore, ForCES specifies a protocol for communicating between a CE and a FE.

In the ForCES architecture a FE represent a logical data-path entity performing per-packet processing and forwarding. That is, external input/output data-path ports are connected to FEs. An example of a FE realization is a line card in an IP router. The CE, on the other hand, implements all the control functions of the NE (such as routing and signalling) and control the forwarding behaviour of FEs using the ForCES protocol. FEs and CEs can dynamically join a NE, where a CE controls the operation of several FEs based on the master-slave model. For the sake of redundancy or load balancing, a NE might include several CEs with the same functionality. The management functions of FEs and CEs are performed by the corresponding manager entities, as depicted in Figure 9.7. An example of a management function is the assignment of a FE control to a specific CE.

A forwarding element in a NE is further structured into several logical functional blocks (LFBs) interconnected in a directed graph. A LFB is defined as a logical entity performing a single action on the packets passing through it, and is modelled using the XML. Examples of LFBs include a packet classifier, or a particular packet encapsulation. A CE controlling a FE can instantiate, update or delete LFBs within a FE, based on the library of LFBs supported in the FE. Therefore, The CE can dynamically program the functionality of corresponding FEs.

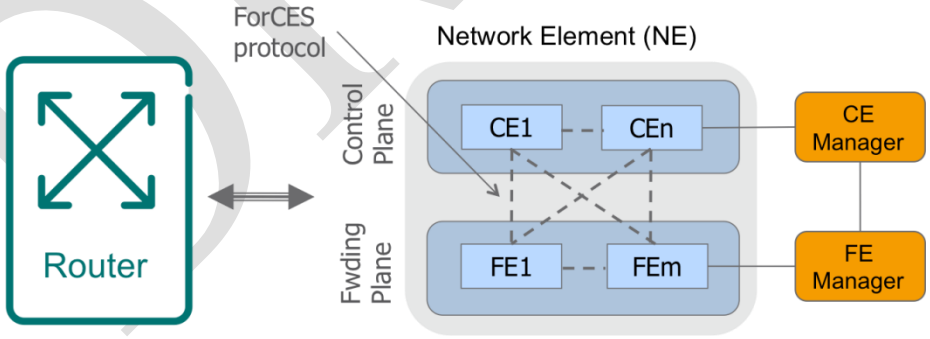

Figure 9.7: ForCES provides a modular framework for structuring a network element (NE) into forwarding elements (FEs) and control elements (CE)

A.2.3.2 Network-level Programming and Control

A.2.3.2.1 SDN control platforms

The role of the controller is crucial in the context of Software-Defined Networks. This section gives a short overview on most common SDN control frameworks. Figure 9.8 gives an overview of a selection of open SDN platforms taken from [Al-Somaidai2014].

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Name	Vendor	Progra-mming language	OpenFlow versions	GUI	Operating system
NOX	Nicira	C ++	1.0, 1.3	NOX GUI	Linux
POX	Nicira	Python	1.0	NOX GUI	Linux, Windows, Mac
Floodlight	Big Switch Networks	Java	1.0	Flood-light web UI, Avior	Linux, Mac
OpenDaylight	Linux Foundation Collaborative Project	Java	1.0, 1.3	Open-Daylight web UI	Linux, Windows
Ryu	Nippon Telegraph and Telephone Corporation	Python	1.0, 1.2, 1.3 and Nicira extension	VND	Linux
Mul	kulcloud	C	1.0, 1.3.1	VND	Linux
Beacon	Stanford University	Java	1.0	VND	Windows, Linux, OSX

Figure 9.8: SDN control platform overview from [Al-Somaidai2014]

The OpenDaylight platform[40] is one of the SDN control platforms which gets more traction and support compared to other frameworks. Deliverable D2.1 Section 2.6 and Annex 2 provides an extensive description of the platform, as it is of substantial interest for the UNIFY project.

The NOX controller[41] was the original (academic) control framework for OpenFlow networks. It is written in C++ and provides a higher-level programmable interface upon forwarding devices and applications. It is designed to support small networks hundreds switches and hosts. NOX's core has features of fast, asynchronous I/O, topology discovery, host tracking possibility, and learning switch feature. The POX platform[42] was derived from NOX controller platform with the main difference is using Python programming language instead of C++ platform. POX uses Python API (version 2.7) to support network virtualization, SDN debugging, and different application such as layer-2 switch, etc. POX and NOX support the same GUI and visualization setup.

Floodlight[43] is another popular SDN control framework developed by Big Switch networks on top of the Java Virtual Machine and is targeting large networks of OpenFlow-capable devices. Floodlight controller realizes a set of common functionalities to control and inquire an OpenFlow network. The controller has features of simple to extend and enhance, easy to setup with minimal dependencies, support for Open Stack Quantum cloud, topology management, and it deals with mixed OpenFlow and non-OpenFlow network. Floodlight supports applications that include a learning switch, firewall, etc. applications.

Ryu[44] is a component-based, open source framework implemented entirely in Python. Ryu targets an operating system for SDN for large networks. Ryu controller includes event

[40] http://www.opendaylight.org/software/
[41] http://www.noxrepo.org/nox/about-nox/
[42] http://www.noxrepo.org/pox/about-pox/
[43] http://www.projectfloodlight.org/floodlight/
[44] http://osrg.github.io/ryu/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

management, in-memory state management, application management, and series of reusable libraries (e.g NetCOONF library, sFlow/NetFlow library and OF-Config library).

The authors of [Kreutz2014] give an extensive overview on the main design characteristics of another set of SDN control platforms. While we won't go into detail of these,

Component	OpenDaylight	OpenContrail	HP VAN SDN	Onix	Beacon
Base network services	Topology/Stats/Switch Manager, Host Tracker, Shortest Path Forwarding	Routing, Tenant Isolation	Audit Log, Alerts, Topology, Discovery	Discovery, Multi-consistency Storage, Read State, Register for updates	Topology, device manager, and routing
East/Westbound APIs	—	Control Node (XMPP-like control channel)	Sync API	Distribution I/O module	*Not present*
Integration Plug-ins	OpenStack Neutron	CloudStack, OpenStack	OpenStack	—	—
Management Interfaces	GUI/CLI, REST API	GUI/CLI	REST API Shell / GUI Shell	—	Web
Northbound APIs	REST, RESTCONF, Java APIs	REST APIs (configuration, operational, and analytic)	REST API, GUI Shell	Onix API (general purpose)	API (based on OpenFlow events)
Service abstraction layers	Service Abstraction Layer (SAL)	—	Device Abstraction API	Network Information Base (NIB) Graph with Import/Export Functions	—
Southbound APIs or connectors	OpenFlow, OVSDB, SNMP, PCEP, BGP, NETCONF	—	OpenFlow, L3 Agent, L2 Agent	OpenFlow, OVSDB	OpenFlow

Figure 9.9: Architecture and design elements of SDN controllers from [Kreutz2014]

A.2.3.2.2 Network Programming Language Overview

Controlling the network behaviour directly via OpenFlow requires low-level programming and managing, considering too much information to achieve the desired operation. As a consequence, several tools, control frameworks and network programming languages have been proposed recently to raise the abstraction level at which network operators can write custom network control software.

Frenetic [Foster2011] proposes a higher-level language design on top of NOX (a development platform for SDN CtrlApps), built around a combination of: i) a declarative query language with an SQL-like syntax, ii) a Functional Reactive Programming (FRP) language, and iii) a specification language for describing packet forwarding.

NetCore [Monsanto2012] as an improvement of Frenetic was proposed by the same authors. It is a declarative programming language describing packet processing in OpenFlow networks at a high abstraction level. It facilitates the compilation of network policies into low-level OpenFlow rules. Network policies are considered as assignments of a set of forwarding target locations (e.g., all ports, controller) to predicates which define a subset of traffic. NetCore provides stateful, dynamic policies which can reactively specialize to traffic. Moreover, different operators can be applied on policies in order to compose more complex ones, and a special mathematical algebra guarantees the correctness of the compilation.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

NetKAT [Anderson2014] is another tool originated from NetCore. It uses regular expressions on network policies to describe the network behaviour. It supports the separation of topology specific and global (topology independent) policies. NetKAT provides a network-wide language and also the concept of network slices to be able to program a dedicated part of a network independently from others. Based on an extended Kleene algebra, the mathematical proof of correctness and soundness can also be provided. High-level questions on the network, such as "Can all hosts talk to each other?", can be answered by algorithmic proof.

Merlin [Soulé2013] is a recently proposed network management framework where network policies can be expressed in a declarative language based on logical predicates (defining a subset of traffic) and regular expressions on Network Functions and bandwidth requirements. Merlin can automatically partition the high-level program into smaller components that can be placed on different types of devices, such as switches, middleboxes and end hosts. Merlin contains a constraint solver and heuristic algorithms to allocate resources according to the demands. Currently, traffic steering is implemented by OpenFlow rules, middlebox functions are realized by generated Click modules, while traffic filtering and rate limiting are implemented by iptables and tc on end hosts.

Alternate designs such as Nettle [Voellmy2011] also enable high-abstraction level network programs through FRP-constructs embedded in the Haskell language. These do not rely on NOX and directly transform network programs into low-level data plane actions. While this approach is more self-contained, it excludes the possibility of having other CtrlApps running alongside them on the same network OS (e.g. NOX). The Lithium architecture of Georgia Tech provides an alternative event-driven network control framework on top of NOX, to enable higher-level languages such as Nettle or network policy languages such as Procera [Voellmy2012] as an additional layer.

FatTire [Reitblatt2013] is a declarative language which enables description of network paths with fault-tolerance requirements. Using this language, each flow can have its own alternative paths in order to deal with failures. Other features such as model checking and dynamic verification are provided by languages such as FlowLog [Nelson2013] and Flog [Katta2012].

The above frameworks provide some support for reasoning about network programs. This enables them, e.g., to assure that the compilation process generates instruction sequences resulting into consistent network states (e.g, avoiding loops in routing). A two-phase commit mechanism to guarantee consistent network updates on top of a runtime system is documented in [Reitblatt2011].

As it is not the purpose of this document to go into detail on all possible network programming language frameworks, we direct the interested reader to [Kreutz2014]. An

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

overview table of the main characteristics of the discussed languages of the referred paper is given in Figure 9.10

Name	Programming paradigm	Short description/purpose
FatTire [189]	declarative (functional)	Uses regular expressions to allow programmers to describe network paths and respective fault-tolerance requirements.
Flog [185]	declarative (logic), event-driven	Combines ideas of FML and Frenetic, providing an event-driven and forward-chaining logic programming language.
FlowLog [184]	declarative (functional)	Provides a finite-state language to allow different analysis, such as model-checking.
FML [137]	declarative (dataflow, reactive)	High level policy description language (e.g., access control).
Frenetic [138]	declarative (functional)	Language designed to avoid race conditions through well defined high level programming abstractions.
HFT [183]	declarative (logic, functional)	Enables hierarchical policies description with conflict-resolution operators, well suited for decentralized decision makers.
Maple [190]	declarative (functional)	Provides a highly-efficient multi-core scheduler that can scale efficiently to controllers with 40+ cores.
Merlin [191]	declarative (logic)	Provides mechanisms for delegating management of sub-policies to tenants without violating global constraints.
nlog [64]	declarative (functional)	Provides mechanisms for data log queries over a number of tables. Produces immutable tuples for reliable detection and propagation of updates.
Nettle [155]	declarative (functional, reactive)	Based on functional reactive programming principles in order to allow programmers to deal with streams instead of events.
NetCore [139]	declarative (functional)	High level programming language that provides means for expressing packet-forwarding policies in a high level.
Procera [156]	declarative (functional, reactive)	Incorporates a set of high level abstractions to make it easier to describe reactive and temporal behaviors.
Pyretic [157]	imperative	Specifies network policies at a high level of abstraction, offering transparent composition and topology mapping.

Figure 9.10: Network Programming language overview from [Kreutz2014]

A.2.3.2.3 Network Monitoring Language Overview

Frenetic/Pyretic

Frenetic [Foster2011] aims to design a simple and intuitive abstractions for programming the three main stages of network management: monitoring network traffic, specifying and composing packet forwarding policies, and updating policies in a consistent way. Frenetic consists of a high-level query language, compiler and run-time system. The query language can subscribe to streams of information about network state, including traffic statistics and topology changes. The runtime system handles the details of polling switch counters, aggregating statistics, and responding to events.

Frenetic's query language allows programmers to express what they want to monitor and control the information they receive using a collection of high-level operators for classifying, filtering, transforming, and aggregating the stream of packets traversing the network., leaving the details of how to actually collect the necessary traffic statistics to the runtime system. Below is an example to query the traffic histogram:

```
Select(bytes) *
Where(inport=2 & srcport=80) *
GroupBy([srcip]) *
Every(60)
```

It uses a syntax that closely resembles SQL, including constructs for selecting, filtering, splitting, and aggregating the streams of packets flowing through the network. The

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

`Select(bytes)` clause states that the program wants to receive the total number of bytes of traffic. The `Where(inport=2 & srcport=80)` clause restricts the query to HTTP traffic arriving on ingress port 2 on the switch. The `GroupBy([srcip])` states to aggregate traffic based on the source IP address. The `Every(60)` says that the traffic counts should be collected every 60 seconds.

The runtime system handles all of the low-level details of supporting queries—installing rules, polling the counters, receiving the responses, combining the results as needed, and composing query implementation with the implementation of other policies. For example, suppose the programmer composes the example monitoring query with a routing policy that forwards packets based on the destination IP address. The runtime system ensures that the first TCP port 80 packet from each source IP address reaches the application's printer routine, while guaranteeing that this packet (and all subsequent packets from this source) is forwarded to the output port indicated by the routing policy. Initial Frenetic run-time system had a reactive, microflow-based strategy for installing rules on switches. At the start of execution, the flow table of each switch is empty, so every packet is sent to the controller and passed to the packet_in handler. Upon receiving a packet, the runtime system iterates through all of the queries, and then traverses all of the registered forwarding policies to collect a list of actions for that switch. The current Frenetic runtime system is proactive (generating rules before packets arrive at the switches) and uses wildcard rules (matching on larger traffic aggregates). It uses an intermediate language, called NetCore, for expressing packet forwarding policies and a compiler that proactively generates as many OpenFlow-level rules for as many switches as possible, but where impossible (or intractable), uses an algorithm called reactive specialization to dynamically unfold switch-level rules on demand. Now the run-time system support OpenFlow 1.0.

Pyretic is a Python implementation of Frenetic and is developed by Princeton University [Reich2013].

The Akamai Query System

The Akamai platform is a network of over ten thousands servers supporting content delivery services including HTTP content, live, on-demand streaming media and etc. The maintenance of such a network requires significant monitoring infrastructure to enable detailed understanding of its state at all times [Cohen2010]. For that purpose, Akamai has developed and uses Query, a distributed monitoring system in which all Akamai machines participate. Query collects data at the edges of the Internet and aggregates it at several hundred places to be used to answer SQL queries about the state of the Akamai network.

Query is partly distributed and partly centralized. The collection of data in thousands of clusters all over the world is fully distributed, but that data need to be aggregated to allow the issuing of SQL queries about the entire Akamai network. A set of a few hundred machines, called Top-Level Aggregators (TLAs) collects data from the cluster proxies and combines data

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

from all the clusters into larger tables. Because it takes all the resources available to most TLAs just to talk to all those clusters and combine their data, TLAs don't have enough processing time left to also answer queries. Therefore they send their aggregated tables to SQL parsers that actually receive queries and compute their answers.

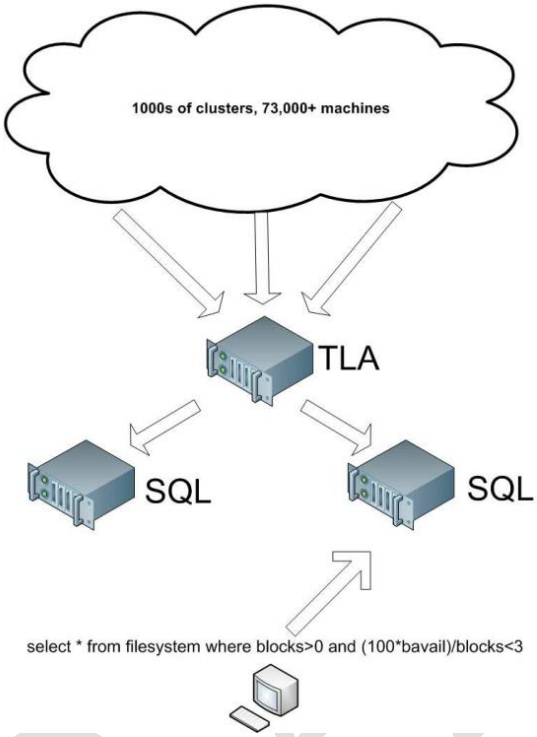

Figure 9.11: The Akamai Query System

Query provides aggregated data in the form of tables that can be accessed using a SQL interface. This interface enables users to easily combine data from multiple data sources, as well as statically generated configuration data. For example, by issuing a query such as the one below, a user can see processes on machines with role "dns" that are using more than 75% of system memory for their RSS:

```
SELECT sys.ip ip, procname, rss, pid
FROM sys, processes
WHERE sys.ip = processes.ip
AND (rss*100)/sys.memtotal > 75
AND sys.ip in
(SELECT ip
FROM machinerole
WHERE role='dns');
```

In Akamai's Query system, alerts can be activated by writing SQL statements which are submitted to the Query system at regular intervals. For example, consider this simplified SQL statement to detect disks with less than 3% of their disk space left free:

```
SELECT
machineip ip key,
mountp mnt key,
```


This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

```
bavail*bsize free space,
(100*bavail)/blocks pct
FROM
filesystem a
WHERE
blocks > 0 and
(100*bavail)/blocks < 3;
ip key mnt key free space pct
------------ ---- ----------- --
10.123.123.1 /var 150,179,840 2
10.123.123.7 /var 72,216,576 1
```

The SQL statement along with other configurable settings defines an alert definition. Each row returned by the SQL statement constitutes a problematic condition, or an alert instance. Each time the alert query is run, the result is compared to the previous result. Any new rows are considered new instances of the alert. As soon as an alert instance is detected, the alert is said to fire. If any rows from the previous iteration are no longer present, the alert is said to clear. Three commonly used alert definition settings deal with these timing parameters: frequency of SQL execution (typically one minute); number of iterations the data are present before an alert fires and amount of time the data must be absent before an alert clears.

Simple Management API

Simple Management API (SMI)[45] is introduced by TM Forum to provide management capabilities for services deployed by service providers.

The following capabilities are available on a SMI interface: Activation/Provisioning of a Service of a Service; State/Usage/Health monitoring of a Service; Update/De-activation of a Service.

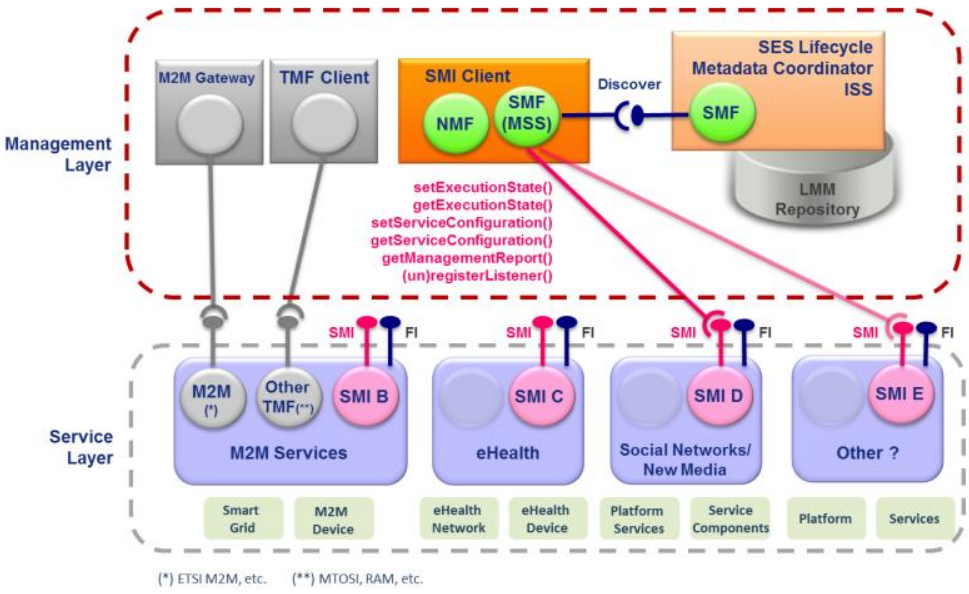

Figure 9.12: Simple Management API architecture

[45] http://www.tmforum.org/TechnicalReports/TR198MultiCloudService/52095/article.html

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The SMI interface can be described by a WSDL description file. The SMI WSDL is provided with SOAP bindings and will support REST as well. A reference REST interface implementation is deployed on Apigee API hosting site.

In addition it pre-provides all the relevant data structure definition (XSD files) which is necessary for using the SID data definitions and therefore enables the use of TM Forum resource modelling.

Several operations are defined in SMI: Get/Set ExecutionState; Get/Set ServiceConfiguration; Get ManagementReport; and Register/UnRegister Listener.

Monitoring related operations are defined in ManagementReport which contains information about the service instance health, execution state, eventual failures and metrics.

In SMI, the *Metric* entity contains the following attributes: *Code* which is used as a code for identifying a particular Metric in a list of metrics; *SourceID* which is used to relate a Metric with a particular service or resource that the reporting service instance depends on; Value which is the value measured; *Reference* which is an optional reference identifier used to correlate the Metric with a particular service consumer or operation context.

OGF NM(C)WG schemas

The Network Measurement Working Group in Open Grid Forum (OGF NM-WG) [Swany2009] has defined a set of extensible schemas for representing network measurement and performance data. These are Extensible Markup Language (XML) schemas, developed using the RELAX NG[46] compact notation. Schemas are defined for information such as the subject of a measurement (e.g. a network path or router interface), the network characteristic measured (e.g. link usage or round-trip time), the time of a measurement and the measurement data themselves.

These schemas are designed to be used together with a base schema for a message type. The message may be one of request, response or store. Only the first two message types described in base schema; a request for particular measurement data, sent by a client, and the corresponding response containing the data, sent by the framework.

The basic schema design is based on the observation that network measurement data can be divided into two major classes: Metadata, which describes the type of measurement data; and Data itself. This structure is present both in the **Messages** sent between various data elements and in data **Stores** – persistent storage of XML documents representing system state. The message structure may contain multiple metadata and data sections. The schema for the top-level message envelope is shown below.

```
namespace nmwg =
"http://ggf.org/ns/nmwg/2.0/"
element nmwg:message {
attribute type { xsd:string } &
```

[46] http://relaxng.org/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

```
( Metadata | Data )+
}
```

In NW-WG, the schema is written in the RELAX NG language. Tools exist to perform translation from RELAX NG to XML Schema when appropriate.

NW-WG only defines base schema and allows independent extensions of the schema to co-exist without central coordination. It has adopted XML namespaces to allow reuse of these same basic element names. The namespace-based approach provides extensibility by defining new basic elements in a tool- or characteristic-specific namespace.

In open source project perfSONAR[47], some of these protocol definitions have been implemented using NM-WG's basic schema and extension.

A.2.3.2.4 I2RS

The Interface to the Routing System Project (I2RS) working group (WG) [Ward2012] of the Internet Engineering Task Force (IETF) was created in 2012 with the goal of creating an architecture revolving around a modern, logically centralized, and programmable interface to a routing system. The I2RS architecture will give a network oriented application the possibility to rapidly influence, and get updated by a routing system.

The routing system is seen as currently implemented control and management plane processes and protocols, as well as the forwarding plane. I2RS is to co-exist with, and complement, the already existing routing system functions, e.g. routing and management protocols, and is to directly interact with relevant parts of this system.

I2RS architecture consists of two main components; the client and the agent. It is through the client that the applications interact with the routing system, and through the agent that the routing system interaction is facilitated (see Figure 9.13).

DRAFT

[47] http://www.perfsonar.net/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

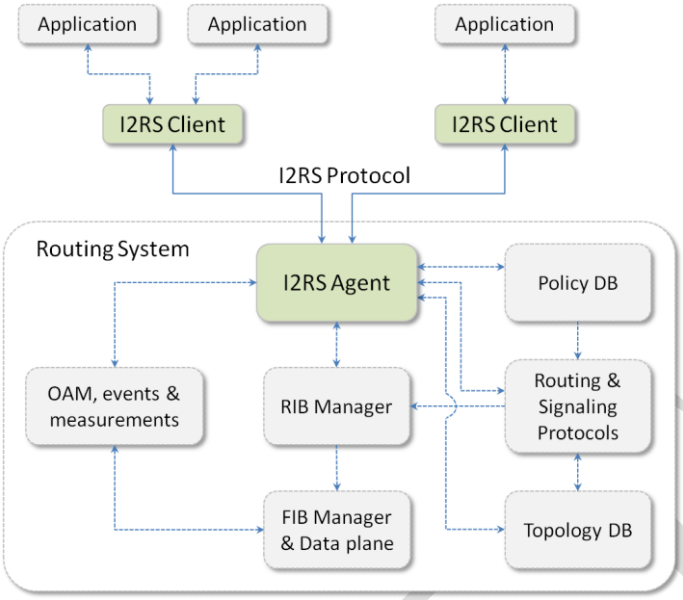

Figure 9.13: I2RS problem space and interaction with relevant routing system functions.

Work within IETF I2RS WG focuses on the I2RS Client and Agent the interface between them. Other functional blocks and interfaces are currently out of scope [Atlas2013, Farrel2013].

- The main objectives of the architecture are to facilitate [Hares2013]:
- an interface that is programmatic, asynchronous, and offers fast interactive access;
- access to structured information and state that is frequently not directly configurable or modelled in existing implementations or configuration protocols;
- ability to subscribe to structured, filterable event notifications from the router;
- operations of I2RS is to be data-model driven to facilitate extensibility and provide standard data-models to be used by network applications.

Currently the WG only have three WG documents; architecture [Hares2013], problem statement [Atlas2013] and information model [Bahadur2013], but quite a few individual drafts. Work is delayed as compared to charter but the work activity is high and progress good.

Besides the I2RS architecture, problem statement and information model, the WG is chartered to, and currently working on:

- The ability to extract information about topology from the network.
- Allowing read/write access to the routing information base (RIB), but no direct access to the Forwarding Information Base (FIB).
- Control and analysis of the operation of the Border Gateway Protocol (BGP) including the setting and activation of policies related to the protocol.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- Control, optimization, and choice of where the traffic exits the network. This can be done based on information external to that provided by the dynamic control plane.

- Distributed reaction to network-based attacks through rapid modification of the control plane behaviour to reroute traffic for one destination while leaving standard mechanisms (filters, metrics, and policy) in place for other routes.

- Service Layer routing to improve on existing hub-and-spoke traffic.

Other things worth mentioning are that injection and creation of topology will not be considered as an initial work item, and that the IETF individual draft on PCE-based Architecture for Application-based Network Operations (ABNO) [King2013] includes I2RS into its architecture.

A.2.3.2.5 ABNO

The goal of the Application-Based Network Operation (ABNO) framework [King2013] is to build on existing functional components and to create a framework that utilizes these components for an application triggered controller of packet and lower layer forwarding technologies, see Figure 9.14. The draft is currently an individual submission, i.e. not a working group (WG), and the intended RFC status is Informational.

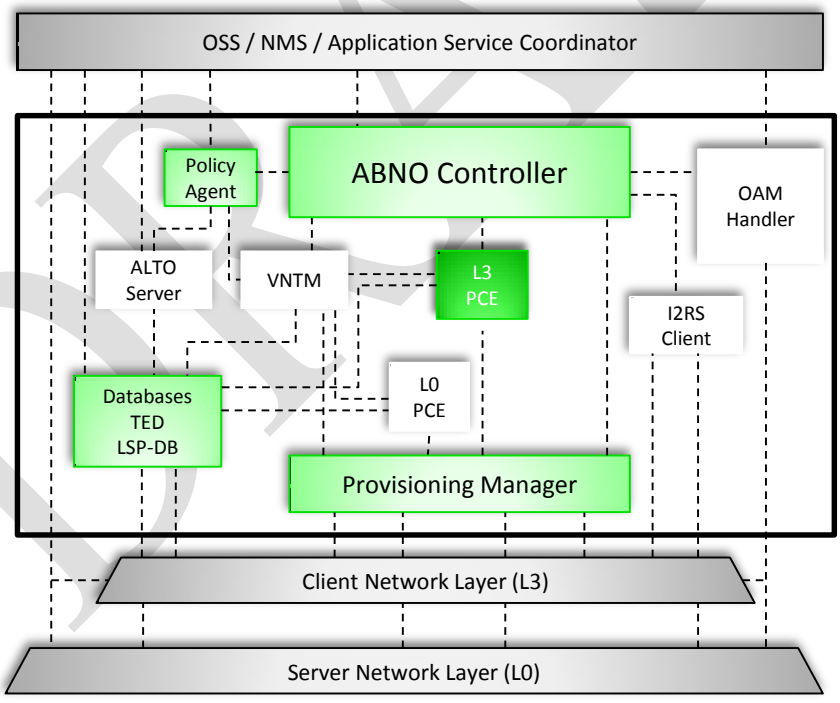

Figure 9.14: Generic functional ABNO architecture

As mentioned above and depicted in Figure 9.14, ABNO includes a number of functional blocks.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The draft indicates the level of extensions the components need in order function as a whole, and it also indicates where interfaces and components are missing.

An OSS or NMS are consumers of resources supplied by the ABNO system. The interaction includes high level service requests, policy specifications, OAM event updates, and direct access to the traffic engineering database (TED DB). All of which will be handled through e.g. programmatic or configuration interface interactions. ABNO also assumes that an application can request services, and it interprets an application in a broad sense and thereby groups all possible implementations of an application into the Application Service Coordinator (ASC) block which has request and status interactions with the ABNO Controller.

The ABNO Controller is the main attachment point to the system and invokes the ABNO components in the right order in response to changing network conditions and application network requirements and policies. In this the Policy Agent plays an important part and is responsible for propagating and coordinating policies into and between the other components of the ABNO system. Another highly important component is the Operations Administrations and Maintenance (OAM) Handler which is responsible for how the network is operating, detecting faults, and taking coordinated actions to possible problems.

Also included are one or more PCEs which can perform and coordinate path computation based on a corresponding set of Traffic Engineering Databases (TED), collecting separated information in e.g. multi-domain or multi-layer use-cases. The Path Computation Element (PCE) can either be stateless or stateful depending on the requirements, where the stateful realization allows for the PCE to take part in the provisioning process.

An ALTO (Application-Layer Traffic Optimization) Server can be used for supplying abstracted views on network information to the Application Service Coordinator, in order to facilitate that relevant information exists for higher layer functions to make decisions. The ALTO Server views are computed based on information in the network databases, taking Policy Agent based policies into account, and through the algorithms used by the PCE.

The Provisioning Manager (PM) is responsible for initiating or channelling requests for establishing LSPs either through control plane triggering or through a programmatic interface. The Virtual Network Topology Manager (VNTM) has a similar function as the PM and in ABNO it can delegate these functions to the PM while focusing on the underlying decisions and policies that are basis for initiating resource allocations, e.g. efficient Server Network Layer allocations in support of Client Network Layer connectivity.

The draft describes a number of different databases that can be used in an ABNO system and points out the two main ones are the TED and LSP Database (LSP-DB), but also databases for topology (ALTO Server), policy (Policy Agent), services (ABNO Controller) etc. The draft identifies contention and sequencing as a possible issue since it is assumed that all functional components can have access to these databases.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Between all these functional blocks there is a need for functional interfaces. The draft, again, tries to build on existing protocols (e.g. OpenFlow, NETCONF, PCEP, IGP-TE, I2RS etc.) while trying to point out areas where more protocol specification is needed.

Finally the draft includes a number of use-cases and exemplifies these through a high level specification on the information flows and decisions that needs be made, between and by the different functional components.

A.2.4 Cloud Programming and Control

A.2.4.1 Cloud-level Programming and Control

A.2.4.1.1 OpenStack

OpenStack[48] is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacentre, all managed through a dashboard that gives administrators control while empowering their users to provision resources through a web interface.

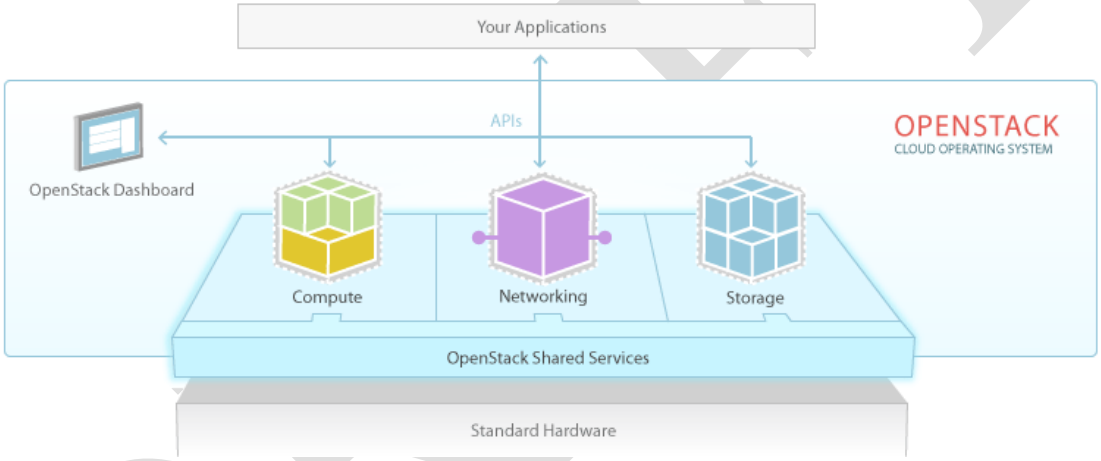

Figure 9.15: OpenStack

OpenStack consists of the following main components:

- Compute (codenamed "Nova") provides virtual servers upon demand. Rackspace and HP provide commercial compute services built on Nova and it is used internally at companies like Mercado Libre and NASA (where it originated).

- Network (codenamed "Neutron") provides "network connectivity as a service" between interface devices managed by other OpenStack services (most likely Nova). The service works by allowing users to create their own networks and then attach interfaces to them. OpenStack Network has a pluggable architecture to support many popular networking vendors and technologies.

[48] https://www.openstack.org/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

- Image (codenamed "Glance") provides a catalogue and repository for virtual disk images. These disk images are most commonly used in OpenStack Compute. While this service is technically optional, any cloud of reasonable size will require it.

- Object Store (codenamed "Swift") provides object storage. It allows you to store or retrieve files (but not mount directories like a fileserver). Several companies provide commercial storage services based on Swift. These include KT, Rackspace (from which Swift originated) and Internap. Swift is also used internally at many large companies to store their data.

- Dashboard (codenamed "Horizon") provides a modular web-based user interface for all the OpenStack services. With this web GUI, you can perform most operations on your cloud like launching an instance, assigning IP addresses and setting access controls.

- Identity (codenamed "Keystone") provides authentication and authorization for all the OpenStack services. It also provides a service catalogue of services within a particular OpenStack cloud.

- Orchestration (codenamed “Heat”) implements an orchestration engine to launch multiple composite cloud applications based on templates.

- Monitoring (codenamed “Celiometer”) can be used for example to collect usage data for billing purposes.

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

DRAFT

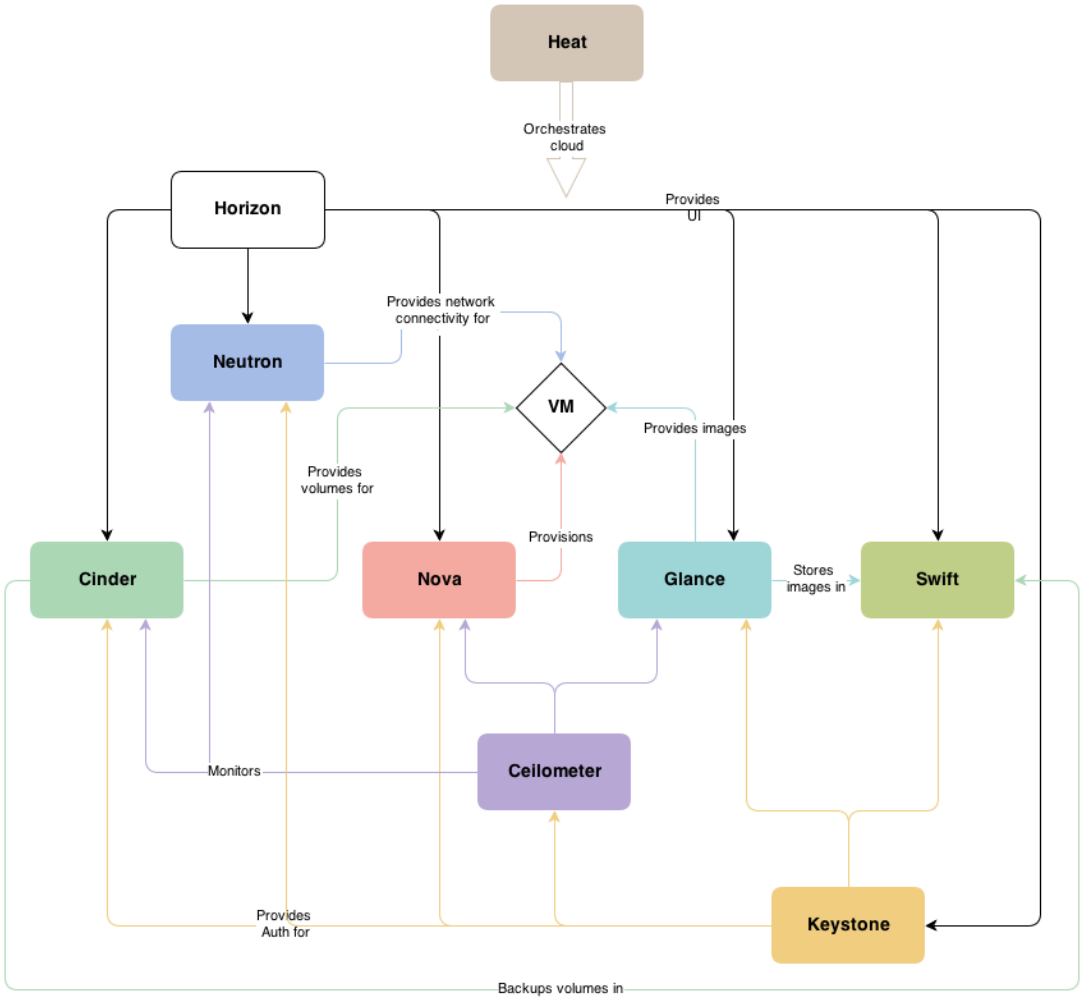

Figure 9.16: OpenStack components

Each of the components provides an API to access it, which can be used as a REST API with the http protocol, or as CLI. The OpenStack APIs are documented in [OS-API].

The computation (Nova) API and the networking (Neutron) are the most related to the UNIFY data plane.

The Nova API[49] uses the following concepts:

- Server: A virtual machine (VM) instance in the compute system. Flavour and image are requisite elements when creating a server.
- Flavour: An available hardware configuration for a server. Each flavour has a unique combination of disk space, memory capacity and priority for CPU time.
- Image: A collection of files used to create or rebuild a server. Operators provide a number of pre-built OS images by default. One may also create custom images from cloud

[49] http://docs.openstack.org/api/openstack-compute/2/content

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

servers have launched. These custom images are useful for backup purposes or for producing "gold" server images if someone plans to deploy a particular server configuration frequently.

The possible operations related to servers are:

- List servers: list of servers by image, flavour, name, and status through the respective query parameters
- Create server: This operation asynchronously provisions a new server. The progress of this operation depends on several factors including location of the requested image, network I/O, host load, and the selected flavour.
- Get server details: Gets details for a specified server.
- Update server: Updates the editable attributes of the specified server.
- Delete server: Deletes a specified server.

The possible further actions on given servers are listed below, the possibly significant ones for UNIFY underlined:

- Change password
- Reboot server
- Rebuild server
- Resize server
- Confirm resized server
- Revert resized server
- Create image

There are so called server admin actions, which permit administrators to perform actions on a server are listed below, the possibly significant ones for UNIFY underlined:

- Pause server
- Unpause server
- Suspend server
- Resume server
- Migrate server
- Reset networking on server
- Inject network information

DRAFT

- Lock server
- Unlock server
- Create server backup
- Live-migrate server
- Reset server state
- Evacuate server
- Add security group
- Remove security group
- Add floating IP address

The command line API of Nova[50] can be used to start experimenting with Nova programmability. The first commands to use to start a VM are the following ones:

- Get the list of available compute resources: “nova hypervisor-list”
- Get the list of available VM images to boot: “nova image-list”
- Get the list of available networks: “nova network-list”
- Boot a new VM: “nova boot --image *imagename* --flavor *m1.tiny* --availability-zone nova:*hypervisor_name* --nic net-id=*network_id vm_name*”

Further type of Nova API types are related to accessing server consoles, managing Flavours, administering Projects (containing multiple machines and networks), security, networking, and volumes.

[50] http://docs.openstack.org/cli-reference/content/novaclient_commands.html

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

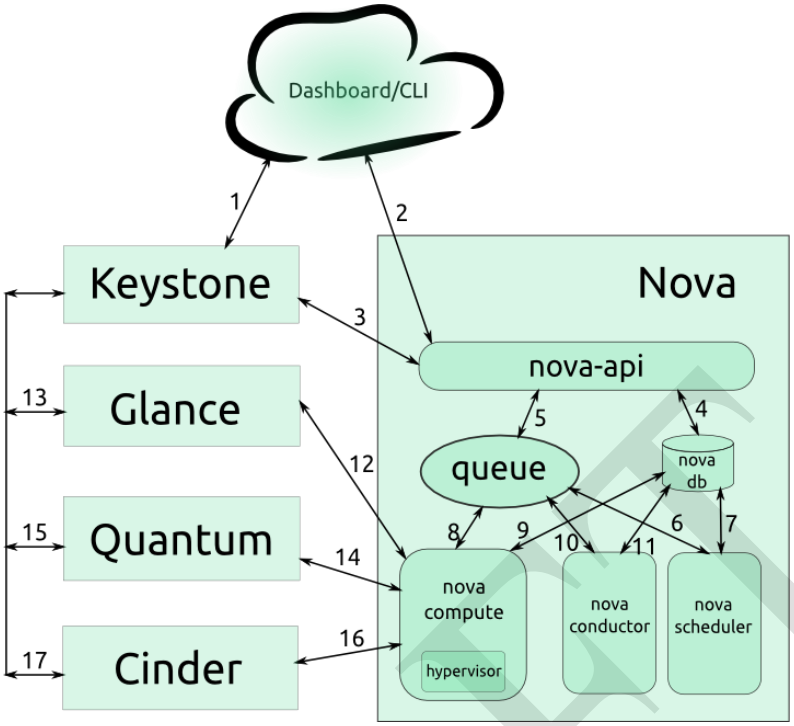

Figure 9.17: Internals of Nova, steps to launch a VM.

OpenStack, as an open-source data centre implementation is a candidate to be managed and orchestrated by UNIFY. The open codebase and interfaces, the widespread use by industry and the wide selection of pluggable hypervisors and networking components supports to use it in UNIFY. Actually this is the legacy datacentre that we consider in UNIFY, because of the large development community and wide industrial support.

Besides using OpenStack as a "legacy" datacentre as a whole to be orchestrated, specific components are also related to UNIFY. The computation (Nova) API can be a candidate to be used for controlling the computing resources in the Unified Node. The networking (Neutron) interface used for intra-datacentre networking provides a subset of the networking functionality needed for the whole UNIFY scope. The orchestration (Heat) interface –which is not an orchestration in the UNIFY terminology, because doesn't consider network and datacentre together and it doesn't make complex optimization on mapping the request to resources/locations – shows a subset of cloud application parameters, to be considered when defining UNIFY Service Graphs.

A.2.4.2 Cloud Controller Overview

There exist several frameworks such as Eucalyptus, Nimbus, OpenNebula, OpenStack and some industry efforts including openQRM[51] and Enomalism[52] which are used to offer virtual machines to users in Cloud computing. OpenStack has been already explained in details and we briefly describe some of the other frameworks in this section.

[51] http://www.openqrm-enterprise.com/
[52] https://www.openhub.net/p/enomalism

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

OpenNebula[53] is an open source IaaS (Infrastructure as a Service) which has a modular design. Such flexible and modular design enables integration with different network infrastructure configurations. Different dynamics such as change in the resource requirements, scaling (resource additions), migration and physical resource failures can be handled in OpenNebula. Another feature of this framework is the capability of cloud federation which offers interface with external clouds to have scalability, multiple-site support and isolation.

Several access interfaces are supported by OpenNebula such as: i) REST-based interfaces ii) OGF OCCI service interfaces and iii) Cloud API standards.

The next framework developed for Cloud computing is EUCALYPTUS54 which stands for Elastic Utility Computing Architecture for Linking Your Program To Useful System. It is compatible with Amazon Web service API (EC2/S3 APIs) used for deploying On-premise private Cloud. It enables collection of heterogeneous virtualization technologies in a single Cloud. It is composed of the following components:

- Cloud Controller: entry point for end user, project managers, developer and administrator
- Cluster Controller: manages the Virtual Machine(VMs) Network.
- Storage Controller: provides block-level network storage
- Node Controller: controls VM activities (installed in each node)

Nimbus55 is also an open-source toolkit for IaaS. The project is focusing on two pieces: i) Nimbus Infrastructure and ii) Nimbus Platform. The former is an EC2/S3 compatible implementation with features including support for proxy credentials, best-effort allocations and batch schedulers. The second piece is providing additional tools which can simplify the management of infrastructure services. This can enable integration with other clouds such as OpenStack and Amazon.

A.2.5 Service-level Programming and Control

A.2.5.1 CLOUDSCALE and ScaleDL

The main goal of this project is to analyse, predict and solve scalability issues in software-based services or in other terms support scalable service engineering. This project provides tools and methods for detecting scalability issues and offers solutions/guidance for the detected issues. ScaleDL is a description language used by service providers as a basis to determine the scalability properties of cloud services.

ScaleDL is a description language for cloud service characterization with the focus on scalability properties. It is composed of 4 sublanguages: i) ScaleDL Usage Evolution ii) ScaleDL

[53] http://opennebula.org/
[54] https://www.eucalyptus.com/
[55] http://www.nimbusproject.org/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Architectural Template iii) ScaleDL Overview and iv) Palladio's PCM extended by SimuLizar's self-adaptation language. A short description for each of these sublanguages is reported below:

- ScaleDL Usage Evolution
- Used by service providers to determine the scalability requirements (e.g. cost metrics of the offered services)
- It specifies scalability requirements by determining the changes of the service workload over time.
- ScaleDL Architectural Template
- Used by architects to model systems according to the best practices and to reuse scalability models provided by architectural template engineers
- ScaleDL Overview
- Used by architects to model the structure of cloud-based architectures at a high level abstractions which is user-friendly as well
- Extended PCM
- Used by architects to model components, their assembly to a system, resources, etc. of the services. PCM is extended to support modelling of self-adaptation (monitoring specifications and adaptation rules)

A.2.5.2 ETSI MANO VNF[56] Graph model

The Network Service describes the relationship between VNFs and possibly PNFs that it contains and the links needed to connect VNFs that are implemented in the NFVI network. Links are also used to interconnect the VNFs to PNFs and endpoints. Endpoints provide an interface to the existing network, including the possibility of incorporating Physical Network Functions to facilitate evolution of the network.

[56] http://docbox.etsi.org/ISG/NFV/Open/Published/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

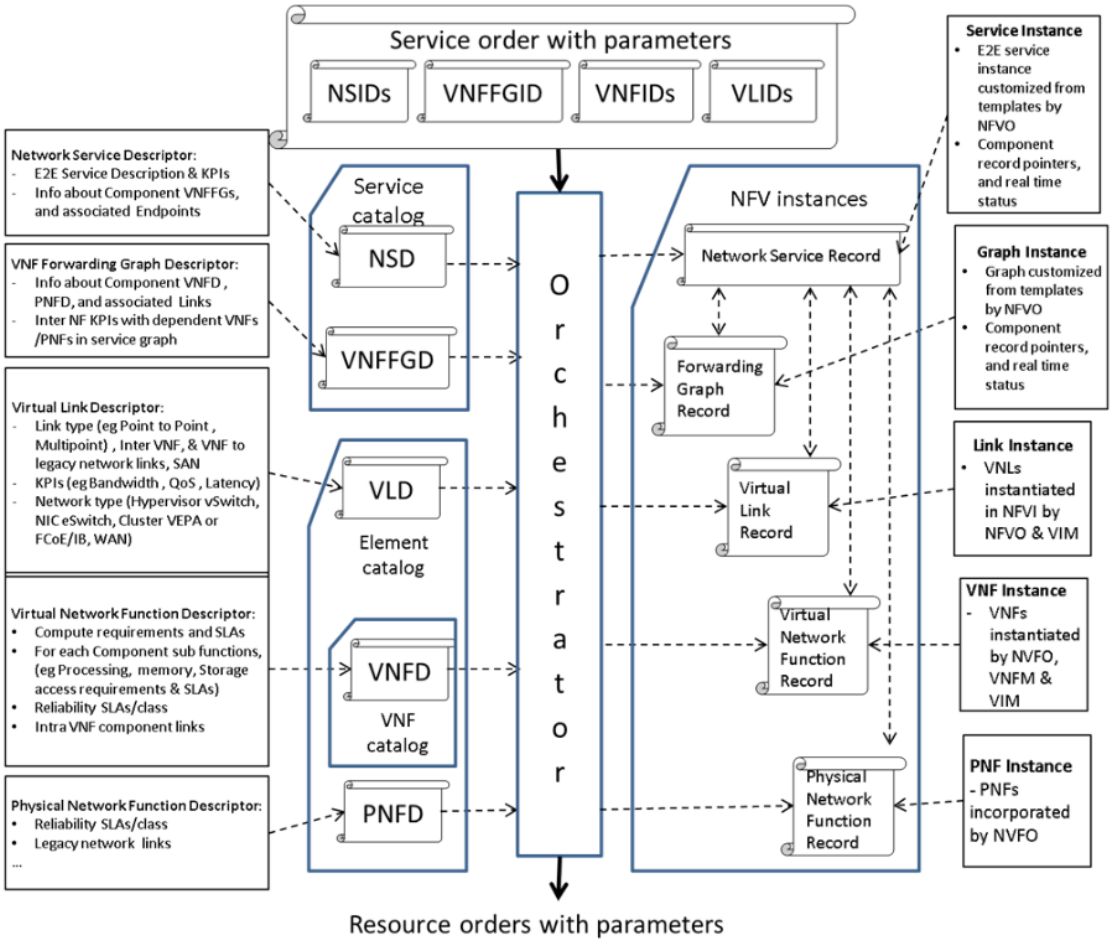

Figure 9.18: ETSI MANO descriptor files

ETSI NFV MANO distinguishes between two categories of information:

- Information that resides in descriptors. These are deployment templates that contain relatively static information used in the process of on-boarding VNFs and NSs.
- Information that resides in records. These contain relatively dynamic run-time data representing e.g. VNF or NS instances; this data is maintained throughout the lifetime of the instance.

To describe a Network Service and the components comprising the Network Service, information elements representing these components are introduced. There are four information elements defined apart from the top level Network Service (NS) information element:

- Virtual Network Function (VNF) information element
- Physical Network Function (PNF) information element
- Virtual Link (VL) information element
- VNF Forwarding Graph (VNFFG) information element

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The information elements can be used in two different contexts: as descriptors in a catalogue or template context or as instance records in a runtime context.

Descriptors:

- A Network Service Descriptor (NSD) is a deployment template for a Network Service referencing all other descriptors which describe components that are part of that Network Service.

- A VNF Forwarding Graph Descriptor (VNFFGD) is a deployment template which describes a topology of the Network Service or a portion of the Network Service, by referencing VNFs and PNFs and Virtual Links that connect them.

- A Virtual Link Descriptor (VLD) is a deployment template which describes the resource requirements that are needed for a link between VNFs, PNFs and endpoints of the Network Service, which could be met by various link options that are available in the NFVI. The NFVO can select an option following consultation of the VNF-FG to determine the appropriate NFVI to be used based on functional (e.g., dual separate paths for resilience) and other needs (e.g., geography and regulatory requirements).

- A VNF Descriptor (VNFD) is a deployment template which describes a VNF in terms of its deployment and operational behaviour requirements. It is primarily used by the VNFM in the process of VNF instantiation and lifecycle management of a VNF instance. The information provided in the VNFD is also used by the NFVO to manage and orchestrate Network Services and virtualised resources on the NFVI. The VNFD also contains connectivity, interface and KPIs requirements that can be used by NFV-MANO functional blocks to establish appropriate Virtual Links within the NFVI between its VNFC instances, or between a VNF instance and the endpoint interface to the other Network Functions.

- A Physical Network Function Descriptor (PNFD) describes the connectivity, Interface and KPIs requirements of virtual Links to an attached Physical Network Function. This is needed if a physical device is incorporated in a Network Service to facilitate network evolution.

A.2.6 Algorithmic Survey: The Virtual Network Embedding Problem

In the early 2000's the Testbed Problem arose when researchers were trying to embed overlay topologies into a given testbed. Back then, the task was to place the overlay nodes in such a fashion that the testbed nodes as well as the testbed links are not over-provisioned [Ricci2003]. In the light of the virtualisation trend, the Virtual Network Embedding Problem (VNEP) arose, to attend the general problem of mapping or embedding a (virtual) graph onto another (substrate) graph.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

The below figure outlines the general idea: Given multiple Virtual Networks (VNets) and a common physical infrastructure, an embedding which maps virtual nodes onto substrate nodes and virtual links onto paths in the substrate is searched for.

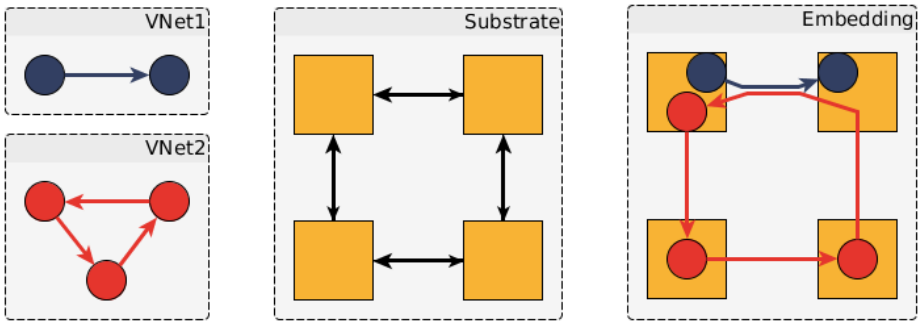

Figure 9.19: Network embedding concept

In the literature many different versions of the VNEP are considered (see for surveys [Belbekkouche2012; Fischer2013]). Based on the very general problem statement and the VNEP's many different applications, we first present a taxonomy of problem types that have been considered, then discuss several algorithmic approaches that were employed to tackle the VNEP and outline how these results can be incorporated into UNIFY.

A.2.6.1 Types of Specification

While we will not detail the mathematic formalisms underlying the general VNEP, we will shortly outline the several different problem types that have been considered in the literature:

- The virtual networks as well as the substrate may either be directed or undirected. While undirected virtual networks may be used to represent bi-directional links, the substrate of wired networks should be represented bi-directed.

- Virtual as well as substrate nodes may specify arbitrary resource demands or capacities. Normally, an abstraction that is based on a combination of single properties as CPU, RAM, etc. is chosen. It is also common, to specify substrate nodes' capacities by the number of virtual machines that may be hosted.

- Additionally to modelling resources, a virtual network may specify restrictions on the mapping of its nodes. This may either be due to incompatible hardware types or the virtual network's spatial specification. In the first cast, a virtual node representing an open flow switch may only be mapped on real open flow switches. In the second case, if the virtual network shall be used to connect multiple locations of a company, then some of the service access points need to be fixed near to these locations.

- Virtual as well as substrate links are generally specified via the necessary or available bandwidth. Additionally latency constraints may be considered.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

A.2.6.2 VNEP Settings

The VNEP has been formulated to attend several different settings, namely:

- There are algorithms attending to the online as well as the offline scenario. In the online scenario (see e.g. [Bienkowski2014]) a single virtual network needs to be embedded instantaneously. In the offline variant, multiple virtual networks are given that are to be embedded in the future ([Rost2014; Schaffrath2012]). Note that the VNEP requires in both cases to perform access control, i.e. to decide whether to embed a virtual network or reject the request. Second level objectives might be to maximize energy savings or to minimize the load of the network by balancing the allocations.

- While centralized approaches that compute embeddings via global knowledge and control over the substrate have initially been the focal point of research, distributed VNEP algorithms have been developed recently (see e.g. [Houidi2008]).

- Due to the generality of the graph-mapping approach, the VNEP has applications both in datacentre and widea-area networks and different algorithms have been developed to obtain good results on either of these topology types (see e.g. [Guo2010] for data centers and [Houidi2011a]).

- While a single virtual node has to be mapped on exactly one node, virtual links may be embedded in a splittable or unsplittable fashion. Splittable here means that to establish a single virtual link multiple paths in the substrate may be used to realize the single link (see e.g. [M. Yu2008]). This may also depend on the specification of the virtual network as some services may deal with packet reorderings while others may not. Especially in the wide-are network further technological routing limitation may need to be considered, e.g. when the standard destination-based routing model is employed.

- The possibility to dynamically alter already existing embeddings, namely the migration of virtual nodes onto other substrate nodes, and the reconfiguration of link realizations, is considered in some of the more recent works. This problem arises e.g. when the infrastructure provider wants to re-balance greedily made embedding decisions or to reduce the virtual networks' footprint by compacting the existing embedding (see e.g. [Cai2010; Fan2006; Houidi2010]).

- For embedding a single virtual network across multiple domains on several different infrastructure providers, the hierarchical or multi-provider VNEP was coined. In this setting, the virtual network must be partitioned a priori to decide which parts of the virtual network shall be embedded on whose infrastructure. The general objective in this case is to minimize the inter-domain traffic, to reduce costs (see e.g. [Choi2013; Hasan2012; Houidi2011b; Xin2011]).

- Lastly, the VNEP may be considered with or without survivability or resiliency constraints, such that some kind of fault tolerance is provided (see e.g. [Rahman2010;

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

Yeow2011; H. Yu2010, 2011]). Especially for large scale service chains as considered by the UNIFY project, resiliency will be of importance.

A.2.6.3 Algorithmic Approaches

As both node and link resources need to be considered at the same time, solving the VNEP is NP-hard in most of the cases [Mcgeer2010]. As detailed in the survey [Fischer2012], several dozen algorithmic approaches have been developed so far. The algorithmic approaches can be subdivided into multiple categories according to the envisioned setting. One of the most prominent categorization is whether the presented algorithm is an heuristic or whether it will yield optimal solutions and is therefore exact. In the realm of exact algorithms, the usage of Mixed-Integer Programming (see [Bertsimas2005] for an introduction) is widely established and also may yield polynomial-time heuristics (see. e.g. [Chowdhury2009; Rost2014]). The literature on heuristic algorithms is much more diverse and ideas from very different fields have been considered. The proposed algorithms range from meta-heuristic approaches like ant-colony optimization [Fajjari2011] over graph isomorphism approaches [Lischka2009] to Markov-Chain random walks [Cheng2011].

Even though the literature is rich on algorithmic proposals to solve the VNEP, many algorithms are specifically designed for a certain setting and are only evaluated on specific scenarios. This severely reduces the comparability of the performance with respect to the quality of found solutions (acceptance ratio, network load, etc.) as well as to their time complexity. Within the UNIFY project, it is therefore advisable to consider both exact and heuristic approaches:

- Heuristic algorithms with polynomial runtime are of high importance for ensuring the elasticity of the service chain deployments. Given e.g. a failure or a scaling request, this issue must be handled within a short frame of time.

- Exact algorithms, especially using Mixed-Integer Programming, allow for obtaining lower and upper bounds for specific scenarios. With respect to the incomparability of solution approaches, employing exact algorithms for obtaining a baseline seems appropriate.

A.2.6.4 Specific Techniques Pertaining to the UNIFY Project

While above a coarse overview on the variety of the VNEP was given, in the following specific problem types and their relation to UNIFY are discussed in-depth.

A.2.6.4.1 Hierarchical VNEP

As introduced above, the hierarchical VNEP considers the scenario where a virtual network must be embedded across multiple different domains or providers. In this scenario the main objective is to minimize the costs arising for inter-provider link usage. Therefore, the approach generally taken is to partition the virtual network into disjoint parts, such that each of these partitions can be fully mapped onto a single provider and the inter-domain bandwidth costs are minimized. In a second optimization step, the sub virtual networks are

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

then mapped by the respective providers [Houidi2011; Wu2012]. The approaches used for partitioning the virtual networks are mainly based on linear or quadratic programming [Bertsimas2005].

The hierarchical VNEP is of special importance when the different providers do not want to share information about how they perform their respective mappings. The reasons brought forward in the literature are the general unwillingness to export topological information as well as the fact that different provider may actually compete [Albarca2013; Dietrich2013].

A.2.6.4.2 Collocation and Clustering

Similarly to the partitioning of virtual networks in the hierarchical VNEP, recent works allow for collocation of virtual nodes, i.e. that a substrate node may host multiple virtual nodes [Fuerst2013; Rost2014]. While again the main idea lies in saving as much bandwidth as possible, the collocation approach easily allows for grouping functionality into a single virtual node by performing a priori graph clusterings. The clustering operation is beneficial in two respects. Firstly, the clustering operation reduces the size of the virtual network, thereby also reducing the runtime of any later on executed embedding algorithm [Fuerst2013]. Secondly, grouping multiple virtual nodes may also be beneficial to estimate the total resource consumption, given all the entailed functionalities. While previous graph algorithms for the VNEP did not allow for such collocations [Lischka2009], the LoCo algorithm presented in [Fuerst2013] already combines the clustering and the embedding steps.

In the light of the recursive orchestration capabilities of the UNIFY architecture, collocation may be used to map service blocks, instead of single VNFs, onto sub-orchestrators. For approximating the capabilities of sub-orchestrators locality-preserving clusterings as presented in [Shen2012] or topology aggregation as presented by [Awerbuch2001] might be used.

A.2.6.4.3 Resiliency

Resiliency or survivability have been considered both in heuristic as well as in exact approaches. While some approaches consider all different types of fault tolerances, namely link failures or node failures, most of the literature considers the problem of ensuring enough (virtual) bandwidth even if a single subtrate link may fail [Fischer2013]. Additionally, some works consider the resiliency under the failure of a whole regional failures [H. Yu2010].

The approaches for achieving resiliency can be subdivided into proactive and reactive ones. While the proactive approaches reserve some fraction of the requested resources along otherwise unused substrate nodes or links, the reactive approaches merely pre-compute how to react under failures, e.g. by initially computing multiple paths, but reserving bandwidth only along a single route. Especially, if the substrate is shared by a multitude of virtual networks and if multiple failures are not very probable, it can be beneficial to allocate failover resources for a set of virtual networks [Rahman2010].

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

A.2.6.4.4 Reconfigurations

Lastly, as the UNIFY architecture will provide scalable Network Functions, we shortly highlight some works in the area of adaptive or reconfigurable virtual network. Reconfigurations, i.e. the ability to change an already existing embedding, were already considered in 2006 in the context of overlay networks in the seminal paper of [Fan2006]: given a virtual network with changing communication requirements, how and when shall routes be computed such that the overall embedding cost over time is minimized?

Within the realm of the VNEP, reconfigurations are considered in multiple scenarios. Firstly, the requirements of a customer may change and trigger a re-embedding [Fan2006]. Secondly, reconfigurations may be used to optimize the overall embedding of multiple virtual networks across the same substrate, e.g. to reduce the bandwidth usage or to save energy [Schaffrath2012]. Thirdly, reconfiguration may become necessary in case of failures in the substrate [Houidi2010]. In all of the above cases, the reconfiguration costs, e.g. bandwidth usage for transferring a virtual machine or management costs, must be traded off with the respective optimization goal.

While both heuristic and exact approaches for reconfigurations are considered in the literature [Fischer2013], Bienkowski et al. were the first to provide a competitive online algorithm for virtual network reconfigurations [Bienkowski2010], such that without knowledge about the future, the found algorithm always achieves a certain approximation guarantee.

DRAFT

Annex 3 Service Provider Scenario for Optimization

Network operators are planning to integrate Cloud/NFV and SDN support into their network infrastructure, in order to simplify the process of provisioning customized service chains, with reduced costs, delivery times and management issues. Enabling Service Chain creation for an ISP means to deploy NFV infrastructure at the edge of his network. Figure 9.20 illustrates a hypothetical architecture for an ISP Point of Presence (POP), which shows a NFV infrastructure next to a edge of devices for Retail and Business accesses. All edge and NFV apparatuses are organized in a hierarchical system, which can easily scale by adding new devices and hierarchical levels if necessary. Note that also the NFV infrastructure and the internal server architecture are usually arranged in a hierarchical setup (e.g. cluster of servers, chassis of servers, servers, CPUs, cores) and all interconnections follow the same pattern accordingly. Such a hierarchical design simplifies also the VNE problem when the task of embedding service chains of virtual/physical appliances is considered.

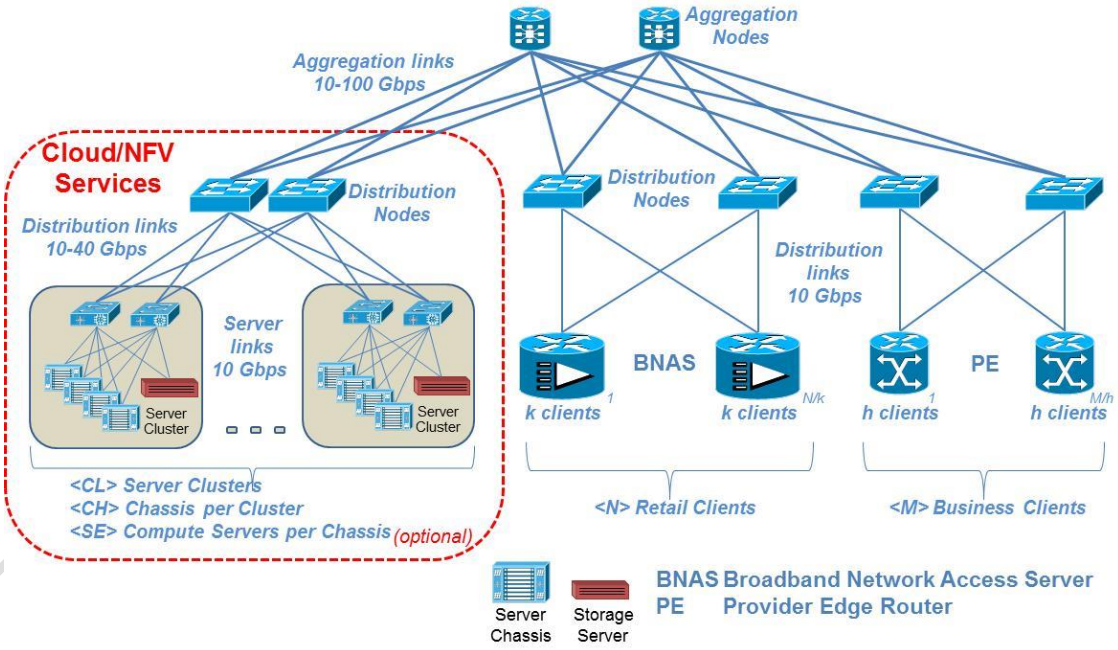

Figure 9.20: ISP Network Point of Presence with integrated NFV infrastructure

NFV subsystems can be deployed gradually in the POPs of the network, or can be deployed only in a small number of central sites. It is possible for the traffic to be steered through service chains implemented in a NFV subsystem also from remote network sites, not only from the co-located site. In this scenario, the VNE problem consists of mapping network resources required by service chains to the set of virtual/physical resources available in the NFV portion of the ISP network. As a matter of fact, it is difficult to conceive an optimization task to be applied at the level of the whole ISP network infrastructure, because most of the network resources (edge access nodes, intra-POP interconnection, backbone infrastructure) are still beyond the control of the optimization process. Initially, only the resources in the NFV

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

infrastructure (e.g. server memory, storage, CPU cores, virtual and physical ports on cloud servers and switches, intra-cloud link bandwidth) are targets of the optimization strategy. In perspective, however, a successful deployment of an orchestration system for an NFV subsystem could be extended to cover all network resources and services of an ISP network, by changing completely the paradigm with which the network is managed and the services are created and provisioned.

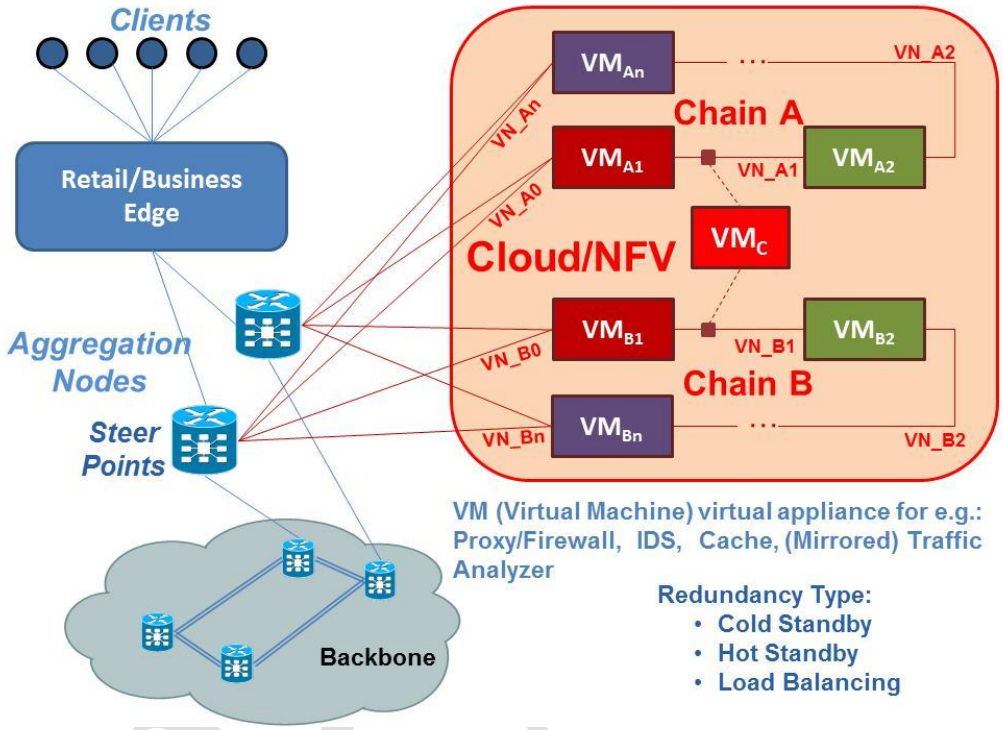

Figure 9.21: Service Chain example with redundant path

A Service Chain is a sequence of virtual/physical appliances through which some traffic is steered (see Figure 9.21). A service chain applies to a «class» of retail/business clients or application flows. Traffic belonging to a «class» is identified at the "steer points" by means of network policies, by interacting with the client authentication system and the routing system. Steer points are ingress/egress nodes where classified traffic is routed through the service chain. Steer points can be also on remote routers which tunnel steered traffic to/from the NFV system.

Service chains may also implement a redundancy scheme (cold/warm/hot standby, load balancing). So a redundant path must be provided, for service reliability under severe conditions (system crashes, power failures, routing faults, overload situations, etc.). Implementing a HA scheme therefore means reserving and allocating additional network, computing and storage resources.

In order to admit service chain requests in an optimized and controlled process, the following information have to be provided for each request:

- The network graph (N, L) of the service chain, describing a set of virtual/physical appliances and their interconnection;
- The bandwidth matrix between nodes in the network graph;
- The specification of service chain virtual/physical nodes, describing main node parameters and constraints;
- The requested instantiation of node parameters (i.e. definition of the expected/required values for each node parameter, coherent with node specification constraints, e.g. number of CPU cores, amount of memory and storage, expected load or session rate, etc.);
- The service chain requirements, which shall be satisfied by the VNE algorithm for the service chain to be deployed; they are, typically, performance requirements (bandwidth, delay, loss) or high availability requirements (e.g. redundancy scheme).

In the above service chain scenario a VNE algorithm/strategy should be used to provide an admission control mechanism in order to accept as many service chain requests as possible, with the following constraints:

- Satisfy the service chain performance requirements for transit delay;
- Minimize resource utilization (in particular the bandwidth utilization);
- Satisfy the service chain requirements for HA (cold/hot standby, Load Balancing);
- Minimize power consumption

Note that, if the network architecture is regular and hierarchical as described at the beginning of this section, it is possible to apply easy criteria for searching optimal virtual network embeddings:

- for a service chain, locate a path of VMs preferably on the same compute server or chassis, alternatively on the same cluster, in order to reduce transit delays and bandwidth consumption;
- locate redundant paths of a service chain on different clusters of the same POP, alternatively on different chassis/server;
- avoid allocation of VMs of a service chain spread over different POPs;
- if a NFV infrastructure is not available in a POP, allocate service chains in the closest NFV site to reduce geographical delays.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

References

[Ahmed2012] Ahmed, M., Huici, F., & Jahanpanah, A. (2012, August). Enabling dynamic network processing with clickos. In Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication (pp. 293-294). ACM.

[Akkiraju2005] Akkiraju, R., Farrell, J., Miller, J. A., Nagarajan, M., Sheth, A., & Verma, K. (2005). Web service semantics-wsdl-s.

[Al-Somaidai2014] Al-Somaidai, M. B., & Yahya, E. B. (2014). Survey of Software Components to Emulate OpenFlow Protocol as an SDN Implementation. American Journal of Software Engineering and Applications, 3(1), 1-9.

[Anderson2014] Anderson, C. J., Foster, N., Guha, A., Jeannin, J. B., Kozen, D., Schlesinger, C., & Walker, D. (2014, January). NetKAT: Semantic foundations for networks. In Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 113-126). ACM.

[Astesiano2002] Astesiano, E., Bidoit, M., Kirchner, H., Krieg-Brückner, B., Mosses, P. D., Sannella, D., & Tarlecki, A. (2002). CASL: the common algebraic specification language. Theoretical Computer Science, 286(2), 153-196.

[Atlas2013] Atlas, A., Nadeau, T. & Ward, D. (2013). Interface to the Routing System Problem Statement. IETF WG draft.

[Bahadur2013] Bahadur, N., Folkes, R., Kini, S. & Medved, J. (2013). Routing Information Base Info Model. IETF WG draft

[Baida2005] Baida, Z., Gordijn, J., Akkermans, H., Saele, H., & Morch, A. Z. (2005). Finding e-service offerings by computer-supported customer need reasoning.International Journal of E-Business Research (IJEBR), 1(3), 91-112.

[Baldine2010] Baldine, I., Xin, Y., Mandal, A., Renci, C. H., Chase, J., Marupadi, V., ... & Irwin, D. (2010, December). Networked cloud orchestration: A geni perspective. InGLOBECOM Workshops (GC Wkshps), 2010 IEEE (pp. 573-578). IEEE.

[Beck, M. T., Fischer, A., de Meer, H., Botero, J. F., & Hesselbach, X. (2013, June). A distributed, parallel, and generic virtual network embedding framework. In Communications (ICC), 2013 IEEE International Conference on (pp. 3471-3475). IEEE.]

[Belbekkouche2012] Belbekkouche, A., Hasan, M. M., & Karmouch, A. (2012). Resource Discovery and Allocation in Network Virtualization. IEEE Communications Surveys & Tutorials, 14(4), 1114–1128. doi:10.1109/SURV.2011.122811.00060

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

[Bertsimas2005] Bertsimas, D., & Weismantel, R. (2005). Optimization over Integers. Dynamic Ideas.

[Bjorklund2010] Bjorklund, M. (2010). YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF.) IETF, RFC6020

[Cai2010] Cai, Z., Liu, F., Xiao, N., Liu, Q., & Wang, Z. (2010). Virtual network embedding for evolving networks. In Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE (pp. 1–5). Ieee.

[Cardoso2010] Cardoso, J., Barros, A., May, N., & Kylau, U. (2010, July). Towards a unified service description language for the internet of services: Requirements and first developments. In Services Computing (SCC), 2010 IEEE International Conference on (pp. 602-609). IEEE.

[Case1990] Case, J., Fedor, M., Schoffstall, M. & Davin, J. (1990). Simple Network Management Protocol (SNMP). IETF, RFC1157

[Case1996] Case, J., McCloghrie, K., Rose, M. & Waldbusser, S. (1996). Introduction to Community-based SNMPv2. IETF, RFC1901

[Cheng2011] Cheng, X., Su, S., Zhang, Z., Wang, H., Yang, F., Luo, Y., & Wang, J. (2011). Virtual network embedding through topology-aware node ranking. ACM SIGCOMM Computer Communication Review, 41(2), 38–47.

[Chinnici2007] Chinnici, R., Moreau, J. J., Ryman, A., & Weerawarana, S. (2007). Web services description language (wsdl) version 2.0 part 1: Core language. W3C recommendation, 26, 19.

[Chisholm2008] Chisholm, S. & Trevino, H. (2008). NETCONF Event Notifications. IETF, RFC5277

[Choi2013] Choi, T., Kim, Y., & Yang, S. (2013). Graph clustering based provisioning algorithm for optimal inter-cloud service brokering. Network Operations and …. Retrieved from http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6665288

[Chowdhury2009] Chowdhury, N. M. M. K. M. M. K., Rahman, M. R. M. R., & Boutaba, R. (2009). Virtual network embedding with coordinated node and link mapping. In INFOCOM 2009, IEEE (pp. 783–791).

[Christos2009] Christos, K., Vassilakis, C., Rouvas, E., & Georgiadis, P. (2009, July). Qos-driven adaptation of bpel scenario execution. In Web Services, 2009. ICWS 2009. IEEE International Conference on (pp. 271-278). IEEE.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

[Chua2013] Chua, Roy, Controller Wars 2.0 – ON.LAB & Juniper Re-Ignite the Open-Source Battleground. http://www.sdncentral.com/market/controller-onlab-juniper-open-source-sdn-battleground-part1/2013/12/

[Cohen2010] Cohen, J., Repantis, T., McDermott, S., Smith, S., & Wein, J. (2010, November). Keeping Track of 70, 000+ Servers: The Akamai Query System. InLISA (Vol. 10, pp. 1-13).

[Crockford2006] Crockford, D. (2006). The application/json Media Type for JavaScript Object Notation (JSON) IETF, RFC4627

[Csoma2014a] Csoma, A., Sonkoly, B., Csikor, L., Németh, F., Gulyás, A., Tavernier, W., & Sahhaf, S. (2014, August). ESCAPE: extensible service chain prototyping environment using mininet, click, NETCONF and POX. In Proceedings of the 2014 ACM conference on SIGCOMM (pp. 125-126). ACM.

[Csoma2014b] Csoma, A., Sonkoly, B., Csikor, L., Németh, F., Gulyás, A., Jocha, D., Elek, J., Tavernier, W., & Sahhaf, S. Multi-layered service orchestration in a multi-domain network environment. In Proceedings of EWSDN European Workshop on Software Defined Networks, September 1-3, 2014 Budapest, Hungary.

[D5.1] Hagen Woesner et al. Deliverable 5.1: Universal Node functional specification and use case requirements on data plane. Tech. rep. UNIFY Project, 2014. url : https://www. fp7-UNIFY.eu/files/fp7- UNIFY- eu- docs/Results/Deliverables/UNIFY- WP5- D5.1-Universalnodefunctionalspecification.pdf.

[D5.2] Hagen Woesner et al. D5.2 Universal Node Interfaces and Software Architecture. Tech. rep.UNIFY Project, Aug. 2014. url : http://fp7-UNIFY.eu/files/fp7-UNIFY-eu-docs/Results/Deliverables/UNIFY-WP5-D5.2-Universal%20node%20interfaces%20and%20software%20architecture.pdf.

[D'Ambrogio2006] D'Ambrogio, A. (2006, September). A model-driven wsdl extension for describing the qos of web services. In Web Services, 2006. ICWS'06. International Conference on (pp. 789-796). IEEE.

[Doria2010] Doria, A., Hadi Salim, J., Hass, R., Khosravi, H., Wang, W. Dong, L., Gopal, R. & Halpern, J. (2010). Forwarding and Control Element Separation (ForCES) Protocol Specification. IETF, RFC5810

[Enns2006] Enns, R. (2006). NETCONF Configuration Protocol. IETF, RFC4741

[Enns2011] Enns, R., Bjorklund, M., Schoenwaelder, J. & Bierman, A. (2011). Network Configuration Protocol (NETCONF). IETF, RFC6241

[ET2013a] ETSI GS NFV 002 v1.1.1 (2013-10), Network Functions Virtualisation (NFV); Architectural Framework, http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.01.01_60/gs_NFV002v010101p.pdf

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

[ETOM] Business Process Framework (eTOM), http://www.tmforum.org/businessprocessframework/1647/home.html

[Fajjari2011] Fajjari, I., Aitsaadi, N., Pujolle, G., Zimmermann, H., & Saadi, N. A. (2011). VNE-AC: Virtual network embedding algorithm based on ant colony metaheuristic. In Communications (ICC), 2011 IEEE International Conference on (pp. 1–6). Ieee.

[Fan2006] Fan, J., & Ammar, M. H. (2006). Dynamic Topology Configuration in Service Overlay Networks: A Study of Reconfiguration Policies. In INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings (pp. 1–12). doi:10.1109/INFOCOM.2006.139

[Farrel2013] Farrel, A. (2013). "Interface to Routing System", presentation, http://www.olddog.co.uk/IIR-SDN-Farrel.ppt

[Fieldings2000] Fielding, R. T. (2000). Architectural styles and the design of network-based software architectures (Doctoral dissertation, University of California, Irvine).

[Fischer2013] Fischer, A., Botero, J. F., Beck, M. T., de Meer, H., & Hesselbach, X. (2013). Virtual Network Embedding: A Survey. IEEE Communications Surveys & Tutorials, 15(4), 1888–1906. doi:10.1109/SURV.2013.013013.00155

[Foster2011] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., & Walker, D. (2011, September). Frenetic: A network programming language. InACM SIGPLAN Notices (Vol. 46, No. 9, pp. 279-291). ACM.

[Fuerst2013] Fuerst, C., Schmid, S., & Feldmann, A. (2013, November). Virtual network embedding with collocation: benefits and limitations of pre-clustering. In Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on (pp. 91-98). IEEE.

[Gember-Jacobson] Gember-Jacobson, Aaron, et al. "OpenNF: enabling innovation in Network Function control." Proceedings of the 2014 ACM conference on SIGCOMM. ACM, 2014.

[Ghijsen2012] Ghijsen, M., Van Der Ham, J., Grosso, P., & De Laat, C. (2012, July). Towards an infrastructure description language for modelling computing infrastructures. In Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on (pp. 207-214). IEEE.

[Hares2013] Hares, S., Nadeau, T., Halpern, J., Ward, D., & Atlas, A. (2013). An Architecture for the Interface to the Routing System. IETF WG draft

[Harrington2002] Harrington, D., Presuhn, R. & Wijnen, B. (2002). An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks. IETF, RFC3411

DRAFT

[Hasan2012] Hasan, M. M., Amarasinghe, H., & Karmouch, A. (2012). Network virtualization: Dealing with multiple infrastructure providers. 2012 IEEE International Conference on Communications (ICC), 5890–5895. doi:10.1109/ICC.2012.6364756

[Houidi2010a] Houidi, I., Louati, W., Ben Ameur, W., & Zeghlache, D. (2011a). Virtual network provisioning across multiple substrate networks. Computer Networks, 55(4), 1011–1023. doi:10.1016/j.comnet.2010.12.011

[Houidi2010b] Houidi, I., Louati, W., Ben Ameur, W., & Zeghlache, D. (2011b). Virtual network provisioning across multiple substrate networks. Computer Networks, 55(4), 1011–1023. doi:10.1016/j.comnet.2010.12.011

[Houidi2008] Houidi, I., Louati, W., & Zeghlache, D. (2008). A distributed virtual network mapping algorithm. In Communications, 2008. ICC'08. IEEE International Conference on (pp. 5634–5640).

[Houidi2010c] Houidi, I., Louati, W., Zeghlache, D., Papadimitriou, P., & Mathy, L. (2010). Adaptive virtual network provisioning. In Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures (pp. 41–48).

[Jarrar2002] Jarrar, M., & Meersman, R. (2002). Formal ontology engineering in the dogma approach. In On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE (pp. 1238-1254). Springer Berlin Heidelberg.

[Jordan2007] Jordan, D., Evdemon, J., Alves, A., Arkin, A., Askary, S., Barreto, C., ... & Yiu, A. (2007). Web services business process execution language version 2.0.OASIS standard, 11, 11.

[Katta2012] Katta, N. P., Rexford, J., & Walker, D. (2012, September). Logic programming for software-defined networks. In Workshop on Cross-Model Design and Validation (XLDI).

[Katz2010] Katz, D., Ward, D. (2010). Bidirectional Forwarding Detection (BFD). IETF, RFC5880

[Keller2003] Keller, A., & Ludwig, H. (2003). The WSLA framework: Specifying and monitoring service level agreements for web services. Journal of Network and Systems Management, 11(1), 57-81.

[King2013] King, D., & Farrel, A. (2014). A PCE-based architecture for application-based network operations. IETF Individual draft

[Kreutz2014] Kreutz, D., Ramos, F., Verissimo, P., Rothenberg, C. E., Azodolmolky, S., & Uhlig, S. (2014). Software-Defined Networking: A Comprehensive Survey. arXiv preprint arXiv:1406.0440.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

[Lengyel2009] Lengyel, B. & Bjorklund, M. (2009). Partial Lock Remote Procedure Call (RPC) for NETCONF. IETF, RFC5717

[Lischka2009] Lischka, J., & Karl, H. (2009). A virtual network mapping algorithm based on subgraph isomorphism detection. In Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures (pp. 81–88).

[LLDP] Link Layer Discovery Protocol (LLDP), http://standards.ieee.org/getieee802/download/802.1AB-2009.pdf

[McGeer2010] Mcgeer, R., Andersen, D. G., & Schwab, S. (2010). The network testbed mapping problem. In 6th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom).

[McGuinness2004] McGuinness, D. L., & Van Harmelen, F. (2004). OWL web ontology language overview. W3C recommendation, 10(10), 2004.

[McKeown2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J. & Turner, J. (2008). OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review, 38(2), 69-74.

[Monsanto2012] Monsanto, C., Foster, N., Harrison, R., & Walker, D. (2012). A compiler and run-time system for network programming languages. ACM SIGPLAN Notices,47(1), 217-230.

[Mörschel2001] Mörschel, I. C., & Höck, H. (2001). Grundstruktur für die Beschreibung von Dienstleistungen in der Ausschreibungsphase. Beuth VerlagGmbH, 2002-12.

[Mukherjee2008] Mukherjee, D., Jalote, P., & Nanda, M. G. (2008). Determining QoS of WS-BPEL compositions. In Service-Oriented Computing–ICSOC 2008 (pp. 378-393). Springer Berlin Heidelberg.

[Nelson2013] Nelson, T., Guha, A., Dougherty, D. J., Fisler, K., & Krishnamurthi, S. (2013, August). A balance of power: Expressive, analyzable controller programming. InProceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (pp. 79-84). ACM.

[O'Sullivan2006] O'Sullivan, J. (2006). Towards a precise understanding of service properties(Doctoral dissertation, Queensland University of Technology).

[OF1.0] OpenFlow Switch Consortium. (2009). OpenFlow Switch Specification Version 1.0. 0.

[OF1.1] OpenFlow Switch Consortium. (2011). OpenFlow Switch Specification Version 1.1. 0

[OS-API] OpenStack API: Resources for application development on private and public OpenStack clouds, http://api.openstack.org/

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

[OS-API] OpenStack API: Resources for application development on private and public OpenStack clouds, http://api.openstack.org/

[Pfaff2013] Pfaff, B. & Davie, B. (2013). The Open vSwitch Database Management Protocol. IETF, RFC7047

[Rahman2010] Rahman, M. R., Aib, I., & Boutaba, R. (2010). Survivable virtual network embedding. NETWORKING 2010, (1), 40–52.

[Reich2013] Reich, J., Monsanto, C., Foster, N., Rexford, J., & Walker, D. (2013). Modular SDN Programming with Pyretic. USENIX; login, 38(5), 128-134.

[Reitblatt2011] Reitblatt, M., Foster, N., Rexford, J., & Walker, D. (2011, November). Consistent updates for software-defined networks: Change you can believe in!. In Proceedings of the 10th ACM Workshop on Hot Topics in Networks (p. 7). ACM.

[Reitblatt2013] Reitblatt, M., Canini, M., Guha, A., & Foster, N. (2013, August). FatTire: declarative fault tolerance for software-defined networks. In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking(pp. 109-114). ACM.

[Ricci2003] Ricci, R., Alfeld, C., & Lepreau, J. (2003). A solver for the network testbed mapping problem. ACM SIGCOMM Computer Communication Review, 33(2), 65–81.

[Rost2014] Rost, M., Schmid, S., & Feldmann, A. (2014). It's About Time: On Optimal Virtual Network Embeddings under Temporal Flexibilities. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium (pp. 17–26). IEEE. doi:10.1109/IPDPS.2014.14

[Schaeffer2007] Schaeffer, Satu Elisa. "Graph clustering." Computer Science Review 1.1 (2007): 27-64.

[Schaffrath2012] Schaffrath, G. (2012). Virtual Network Management. Universit{ä}tsbibliothek.

[Schoenwaelder2010] Schoenwaelder, J. (2010). Common YANG Data Types. IETF, RFC6021

[Schoenwaelder2013] Schoenwaelder, J. (2013). Common YANG Data Types. IETF, RFC6991

[Scott2010] Scott, M. & Bjorklund, M. (2010). YANG Module for NETCONF Monitoring. IETF, RFC6022

[Sommer2012] Sommer, R., CARLI, L., Kothari, N., Vallentin, M., & Paxson, V. (2012). HILTI: An Abstract Execution Environment for Concurrent, Stateful Network Traffic Analysis (p. 20). Tech. Rep. TR-12-003, ICSI.

[Soulé2013] Soulé, R., Basu, S., Kleinberg, R., Sirer, E. G., & Foster, N. (2013, November). Managing the network with Merlin. In Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks (p. 24). ACM.

This is a draft version of Deliverable D3.1. It is subject to pending approval by the European Commission.

[Strassner2002] Strassner, J. (2002). DEN-ng: achieving business-driven network management. In Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP (pp. 753-766). IEEE.

[Swany2009] Swany, M. (2009). An Extensible Schema for Network Measurement and Performance Data. Network Measurements Working Group

[TMF] https://www.tmforum.org/

[VanderHam2006] Van der Ham, J. J., Dijkstra, F., Travostino, F., Andree, H., & de Laat, C. T. (2006). Using RDF to describe networks. Future Generation Computer Systems, 22(8), 862-867.

[VanderHam2013] Van der Ham, J., Dijkstra, F., Apacz, R., & Zurawski, J. (2013). Network Markup Language Base Schema version 1

[Voellmy2011] Voellmy, A., & Hudak, P. (2011). Nettle: Taking the sting out of programming network routers. In Practical Aspects of Declarative Languages (pp. 235-249). Springer Berlin Heidelberg.

[Voellmy2012] Voellmy, A., Kim, H., & Feamster, N. (2012, August). Procera: a language for high-level reactive network control. In Proceedings of the first workshop on Hot topics in software defined networks (pp. 43-48). ACM.

[Wang2013] Wang, B. C., Tay, Y. C., & Golubchik, L. (2013, August). Resource Estimation for Network Virtualization through Users and Network Interaction Analysis. In Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2013 IEEE 21st International Symposium on (pp. 434-443). IEEE.

[Ward2012] Ward, D., & Nadeau, T. (2012). Interface to the Routing System Problem Statement. IETF WG draft.

[Xin2011] Xin, Y., Baldine, I., Mandal, A., Heermann, C., Chase, J., & Yumerefendi, A. (2011). Embedding virtual topologies in networked clouds. In Proceedings of the 6th International Conference on Future Internet Technologies (pp. 26–29).

[Yeow2011] Yeow, W.-L., Westphal, C., & Kozat, U. C. (2011). Designing and embedding reliable virtual infrastructures. ACM SIGCOMM Computer Communication Review, 41(2), 57–64.

[Yu2011] Yu, H., Anand, V., Qiao, C., & Sun, G. (2011). Cost Efficient Design of Survivable Virtual Infrastructure to Recover from Facility Node Failures. In Communications (ICC), 2011 IEEE International Conference on (pp. 1–6). Ieee.

[Yu2010] Yu, H., Qiao, C., Anand, V., Liu, X., Di, H., & Sun, G. (2010). Survivable virtual infrastructure mapping in a federated computing and networking system under single regional

DRAFT

failures. In Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE (pp. 1–6). Ieee.

[Yu2008] Yu, M., Yi, Y., Rexford, J., & Chiang, M. (2008). Rethinking virtual network embedding: substrate support for path splitting and migration. ACM SIGCOMM Computer …, 38(2). doi:10.1145/1355734.1355737

DRAFT