D4.1: Provisional API and Information Model Specification

Deliverable

Document ID	NECOS-D4.1
Status	Final
Version	1.0
Editor	Panagiotis Papadimitriou (UOM)
Due	31/08/2018
Submitted	02/10/2018

Abstract

This deliverable presents an initial version of the NECOS information model through presenting its overall, the infrastructure description and the slice specification viewpoints, the supported client-to-cloud and cloud-to-cloud APIs, as well as the workflow for resource discovery during slice provisioning. The deliverable further provides a state-of-the-art analysis of relevant information models and cloud APIs.

TABLE OF CONTENTS

LIST OF FIGURES

CONTRIBUTORS

Contributor	Institution
Panagiotis Papadimitriou	University of Macedonia (UOM)
Lefteris Mamatas	University of acedonia (UOM)
Ilias Sakellariou	University of Macedonia (UOM)
Sofia Petridou	University of Macedonia (UOM)
Chronis Valsamas	University of Macedonia (UOM)
Sotiris Skaperas	University of Macedonia (UOM)
Antonis Tsioukas	University of Macedonia (UOM)
Rafael Pasquini	Federal University of Uberlândia (UFU)
Raquel Fialho Lafetá	Federal University of Uberlândia (UFU)
Fábio Luciano Verdi	Federal University of São Carlos (UFSCAR)
Paulo Ditarso	Federal University of São Carlos (UFSCAR)
André Beltrami	Federal University of São Carlos (UFSCAR)
Javier Baliosian	Universitat Politècnica de Catalunya (UPC)
Fernando Farias	Federal University of Pará (UFPA)
Billy Pinheiro	Federal University of Pará (UFPA)
Antonio Abelem	Federal University of Pará (UFPA)
Christian Rothenberg	University of Campinas (UNICAMP)
David Moura	University of Campinas (UNICAMP)
Marcos Salvador	University of Campinas (UNICAMP)
Fransesco Tusa	University College London (UCL)
Stuart Clayman	University College London (UCL)
Sand Correa	Federal University of Goias (UFG)
Leandro Freitas	Federal University of Goias (UFG)
Luis M. Contreras	Telefónica Investigación y Desarrollo (TID)

Reviewers

Reviewer	Institution
Fábio Luciano Verdi	Federal University of São Carlos (UFSCAR)
Luis M. Contreras	Telefónica Investigación y Desarrollo (TID)
Rafael Pasquini	Federal University of Uberlândia (UFU)

Acronyms

AP	Access Point	PNF	Physical Network Function
API	Application Programming Interface	PNM	Physical Network Manager
B2B	Business-to-Business	RAN	Radio Access Network
C2B	Customer-to-Business	RC	Resource Control
COMS	Common Operation and Management of network Slicing	RDF	Resource Defined Template
ECA	Event-Condition-Action	REST	Representational State Transfer
EPA	Extended Platform Awareness	RT	Resource Topology
ETSI	European Telecommunications Standards Institute	SCM	Slice Charging Management
FIB	Forwarding Information Base	SCPO	Slice Capacity Planning and Optimization
GPS	Global Positioning System	SDO	Service Data Object
IaaS	Infrastructure as a Service	SFM	Slice Fault Management
IM	Information Model	SLA	Service Level Agreement
IMT	International Mobile Telecommunications	SLMCCS	Slice Lifecycle Customer Care Support
ISG	Industry Specification Group	SLO	Service Level Objective
ITU	International Telecommunications Union	SP	Slice Provisioning
KPIs	Key Performance Indicators	SRA	Slice Resource Alternatives
LINP	Logical Isolated Network Partitions	SRMA	Slice Resource Monitoring and Analytics
LSCD	Lightweight Slice Defined Cloud	SSH	Secure Shell
MANO	Management and Orchestration	SSM	Slice Security Management
MdO	Multi-domain Orchestrator	SRR	Slice Resource Repository
ML	Machine Learning	UML	Unified Modeling Language
MPLS	Multiprotocol Label Switching	VDU	Virtual Deployment Unit
NFVO	Network Function Virtualization Orchestrator	VIM	Virtualized Infrastructure Manager

NFV-O	NFV Orchestrator	VL	Virtual Link
NML	Normalized Maximum Likelihood	VNC	Virtual Network Computing
NOVI	Networking Innovations Over Virtualized Infrastructures	VNF	Virtual Network Function
NS	Network Service	VNFCs	VNF Components
NSD	Network Service Descriptor	VNFFG	VNF Forwarding Graph
OAM	Operations and Management	VNP	Virtual Network Provider
OCCI	Open Cloud Computing Interface	VRM	Virtual Resources Manager
OGF	Open Grid Forum	WIM	WAN Infrastructure Manager
OSM	Open Source MANO	XML	Extensible Markup Language
OWL	Web Ontology Language	YAML	YALM Ain’t Markup Language
PDT	Partially Defined Template	YANG	Yet Another Next Generation

Executive Summary

The NECOS project aims at the design and implementation of a system architecture for cloud slicing across multiple administrative domains. One of the novel aspects of the project is the on-demand instantiation of a Virtual Infrastructure Manager (VIM) per slice, which effectively enables the tenant to exercise fine-grained control and management of his slice.

This deliverable is focused on slice specification and provisioning. As such, the deliverable initially presents two information models specified by the project for the infrastructure description and the slice specification, respectively. Furthermore, D4.1 provides a detailed description of a wide range of API methods for (i) slice request, management, and configuration by the client, and (ii) slice request, instantiation, and run-time management by the NECOS orchestrator. In addition, the deliverable describes the methods supported by NECOS for the discovery of resources during slice provisioning.

The information models, cloud APIs and resource discovery methods will be further developed and refined, as the project progresses and the NECOS system is built. Their final version will appear in the next version of this deliverable, *i.e.*, D4.2.

1 Introduction

The NECOS project addresses the challenging problem of network slicing across multiple cloud environments, such as cloud datacenters and edge clouds. In this respect, NECOS aims at building a platform for the provisioning, management, and resource orchestration of network slices, enabling a new cloud computing model, namely ***Slice as a Service***. One of the novel aspects of the project is the on-demand instantiation of a Virtual Infrastructure Manager (VIM) per slice, which effectively enables the tenant [1]to exercise fine-grained control on his slice, eliminating unnecessary provider interventions during the slice lifetime. NECOS supports various slicing operational model, with the current project focus being on VIM-independent slicing (termed as *Mode 0*) and VIM-dependent slicing (termed as *Mode 1*). In particular, *Mode 0* grants the tenant with direct access to a dedicated VIM, whereas *Mode 1* provides a shared VIM among multiple tenants.

This deliverable is focused on information models and Application Programming Interface (API) specifications for slice request, provisioning, and run-time management. Slice creation raises the need for means to describe slice requests as well as physical resources. This inherent need is satisfied through an information model that provides resource descriptions at different levels of abstraction, meeting the requirements of slice specifications and infrastructure description. This provisional information model has been developed after a careful inspection of related information models, such as COMS (Common Operations and Management on network Slices) and ETSI NFV MANO. The deliverable provides an initial description of the NECOS information model, which will be further refined, based on inputs from the NECOS system implementation.

The deliverable further reports on a set of cloud APIs to enable slice request, creation, configuration, and run-time management. The respective cloud APIs have been subdivided into two classes: (i) client-to-cloud APIs, which include API methods invoked by the tenant for slice request as well as slice management and control upon the slice creation, and (ii) cloud-to-cloud APIs, which are associated with interactions between NECOS system components residing in different domains (*e.g.*, in the case of cloud federation), such as the *Slice Resource Orchestrator*, the *Slice Builder*, *Slice Broker*, the *Slice Agents*, and the *Slice Controllers*. More specifically, the set of *Cloud-to-Cloud APIs* comprises the following APIs: (i) *Slice Request Interface*, (ii) *Slice Instantiation Interface*, (iii) *Slice Marketplace Interface*, and (ii) *Slice Runtime Interface*. Similar to the information model, all cloud API specifications are preliminary and are expected to undergo potential modifications and/or extensions, as the NECOS slice orchestration platform is being built and results are collected from the feasibility and performance tests.

In addition, the deliverable presents the main workflow for resource discovery towards slice creation. The resource discovery methods rely on the information model and the cloud APIs specified by NECOS and essentially constitute workflows for information exchange, including slice requests, resource requests, and resource offerings. The discovery methods involve various NECOS components, such as the *Slice Builder*, the *Slice Broker*, and the *Slice Agents*.

1.1 Deliverable Structure

The deliverable structure provides a clear separation between the three main outputs of NECOS WP4 (i.e., information model, cloud APIs, resource discovery methods) and the state-of-the-art (SOTA), helping the reader to grasp the project contributions and further understand how the project goes beyond the SOTA in the area of cloud slicing.

In further detail, the deliverable is structured as follows. Section 2 provides a SOTA analysis of relevant cloud APIs and information models. After the analysis, the deliverable identifies the gaps and extracts the useful features of these APIs and information models for network slicing. Section 3 provides a detailed documentation of the information model for the slice specification and the infrastructure

[1] The terms *tenant* and *client* are used interchangeably, representing the cloud service consumer (*i.e.*, Slice-as-a-Service consumer, in the context of NECOS).

description. Initially, there is a high-level representation of the whole model, followed by more detailed descriptions of (i) the slice, as specified, requested and viewed by the *Tenant*, and (ii) the physical infrastructure, which includes very detailed specifications of the main infrastructure components for resource availability, allocation and monitoring by the infrastructure provider. Section 4 elaborates on the resource discovery framework and the supported methods for resource discovery during slice creation. Section 5 documents the two classes of cloud APIs (namely *client-to-cloud* and *cloud-to-cloud*), providing a description of all supported API methods. Each API class is presented in a separate subsection. Finally, Section 6 provides a summary of the project contributions with respect to slice specification and provisioning, as well as an outline of the next steps for the information model and the cloud APIs, whose final version will be documented in D4.2.

1.2 Contribution of this Deliverable to the project and relation with other Deliverables

This deliverable documents the outputs of WP4 and, more specifically, an information model for slice specification and infrastructure description, and a set of API methods for slice request, creation, configuration and run-time management. These contributions complement the NECOS slicing architecture, which is presented in D3.1. In particular, D4.1 provides the necessary means for slice specification and provisioning, based on the architecture that appears in D3.1. This deliverable will further provide inputs to D5.1 and D6.1, and more specifically, to the feasibility and performance tests that will be conducted based on the NECOS proof-of-concept system implementation. The inputs to D5.1 and D6.1 comprise the information model, the cloud APIs and the resource discovery framework, which will be developed and integrated into the NECOS system for slice provisioning. Finally, D4.2 will be based on this version, documenting the final version of the information model and the cloud APIs, as these will evolve during the course of the project.

2 State-of-the-Art Analysis

This section provides an analysis of SOTA in terms of cloud APIs and information models for network resource descriptions. Our main goal is to present a comprehensive description of the most relevant efforts in research projects and other initiatives, as well as identify gaps in terms of network slicing and extract useful features from existing APIs and models, which can be fed into the respective specifications of NECOS. In this respect, Section 2.1 discusses relevant cloud APIs, whereas Section 2.2 describes relevant information models. Section 2.3 summarizes the SOTA analysis and identifies the suitability of existing APIs and information models.

2.1 Cloud APIs

2.1.1 5G-PPP

The 5G-PPP white paper, entitled "*View on 5G Architecture*", proposes the 5G architecture illustrated in Figure 1, as a result of the composition of various individual 5G-PPP initiatives and research projects. On this basis, the following functionalities of the architecture are analyzed: network slicing, programmability and softwarization, management and orchestration, 5G security, and RAN architecture.

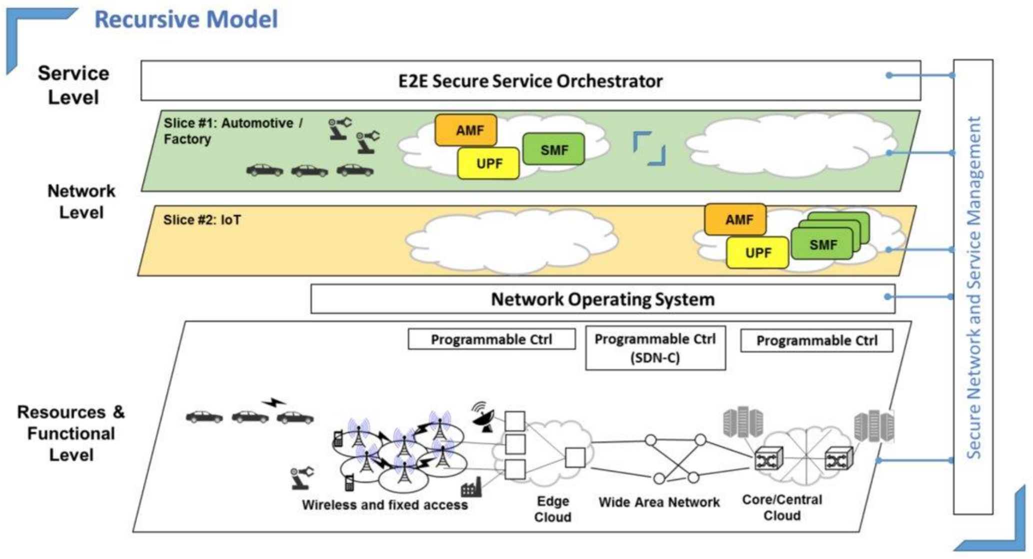
Recursive Model
Service Level
E2E Secure Service Orchestrator
Slice #1: Automotive / Factory
AMF
UPF
SMF
Network Level
Slice #2: IoT
AMF
SMF
UPF
Network Operating System
Programmable Ctrl
Programmable Ctrl (SDN-C)
Programmable Ctrl
Resources & Functional Level
Wireless and fixed access
Edge Cloud
Wide Area Network
Core/Central Cloud
Secure Network and Service Management

Figure 1. 5G overall architecture
(Source: 5G-PPP, View on 5G Architecture v2.0).

For the overall architecture, a recursive structure is proposed, defined as "the ability to build a service out of existing services". In a network slicing point of view, this capability allows a slice instance operating on top of the infrastructure resources provided by the slice instance below. The tenant can operate its virtual infrastructure as it operates the physical one, allocating and reselling part of the resources to other tenants in a recursive manner. As such, each tenant can own and deploy its own MANO system. To provide support for this key functionality, a set of homogeneous APIs are needed to provide a layer of abstraction for the management of each slice and controlling the underlying virtual resources.

The structure of APIs are not on the main objectives of the white paper. However, in a conceptual framework, APIs functionalities are considered in architectures related to the NECOS project and more specifically, network slicing, and management and orchestration architectures.

In terms of network slicing, 5G-PPP proposes a set of APIs for the interaction between Network Services (NS) and the corresponding VNFs that encompass the following attributes: network-slice ID, nodes, links, connections points, storage resources, compute resources, topologies, network services, service specific managers, network functions, virtual network functions, network function specific managers and predefined function blocks. Moreover, these information elements are currently under standardization in the ETSI NFV ISG, in OASIS TOSCA standards and in IETF. As network slicing services can be grouped to two different levels, *i.e.*, (i) the provisioning of Virtual Infrastructures (VI) and (ii) the provisioning of tenants owned NS, 5G-PPP also provides a general categorization of APIs needed to enable both services providing to different degree of control of network slices, defined as follows:

- Network Service Allocation / Modification / De-allocation API,
- Virtual Infrastructure Allocation /Modification / De-allocation API,
- Virtual infrastructure control API with limited control, and,
- Virtual infrastructure control API with full control.

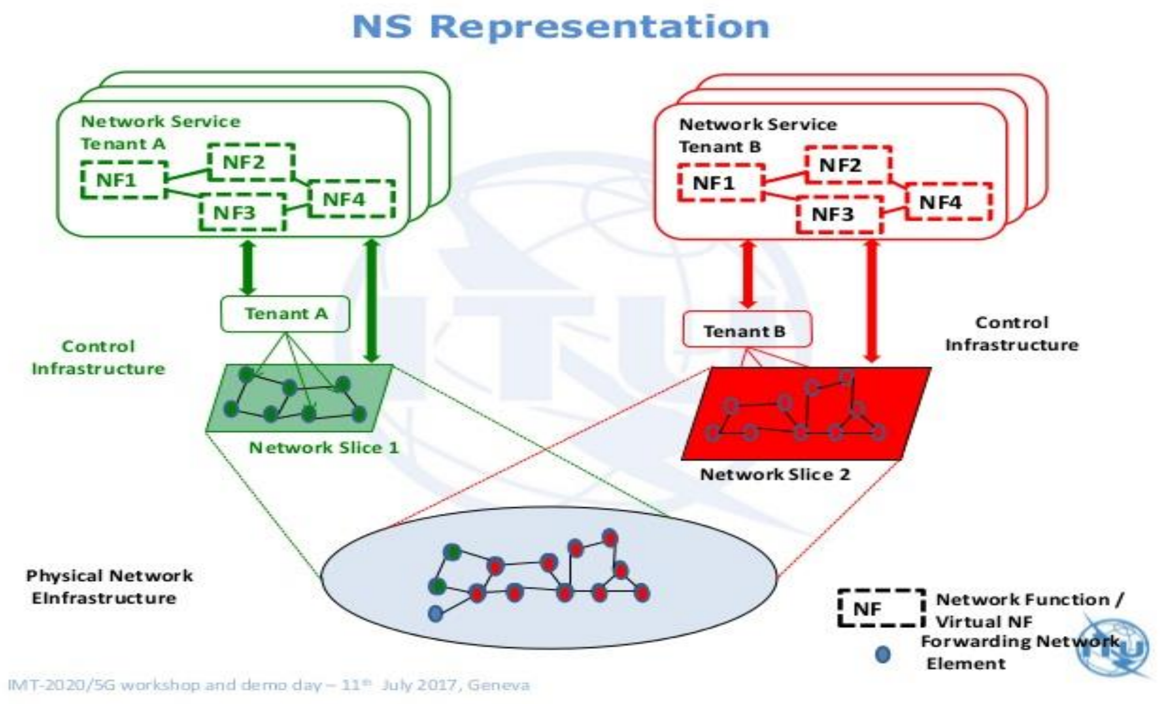

Figure 2. Network service representation
(Source: 5G-PPP, View on 5G Architecture v2.0).

A set of APIs is also required for the multi-domain orchestration, which includes the automated management of services and resources in multi-technology environments (multiple domains involving different cloud and networking technologies) and multi-operator environments (multiple administrative domains). This challenging plane, consisting of various concepts as depicted in Figure 2, has to be supported by several APIs.

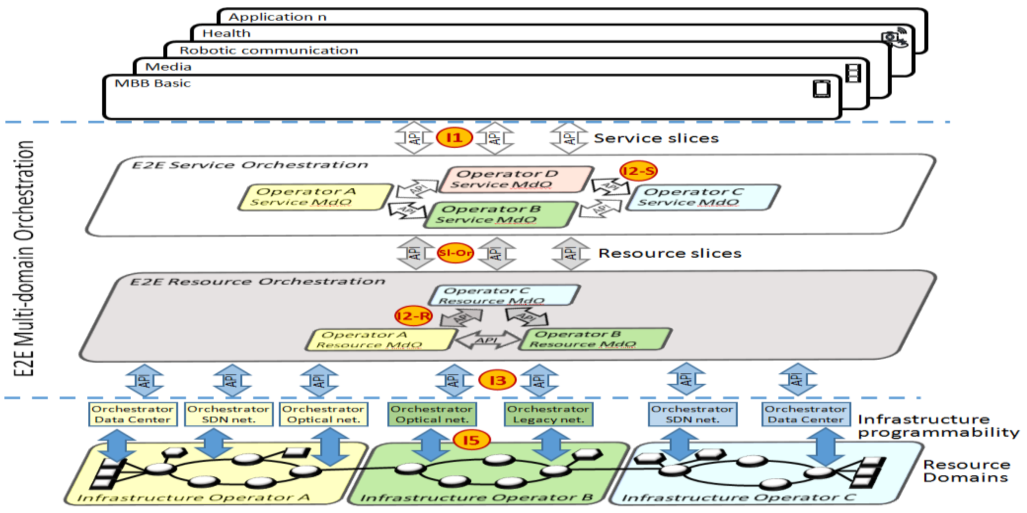

Figure 3. 5G-PPP APIs.

At the lower plane of Figure 3, there are resource domains, exposing resource abstraction on interface I5. Domain orchestrators perform resource orchestration and/or service orchestration exploiting the abstractions exposed on I5 by resource domains.

At Multi-domain orchestrator (MdO) plane, the resource MdO belonging to an infrastructure operator, for instance operator A, interacts with domain orchestrators, via interface I3 APIs, to orchestrate resources within the same administrative domains. The MdO interacts with other MdOs via interface I2-R APIs (business-to-business or “B2B”) to request and orchestrate resources across administrative domains. Resources are exposed at the service orchestration level on interface Sl-Or to Service MdOs. Interface I2-S (B2B) is used by Service MdOs to orchestrate services across administrative domains.

Finally, the Service MdOs expose, on interface I1, service specification APIs (Customer-to-Business or “C2B”) that allow business customers to specify their requirements for a service. The framework also considers MdO service providers, such as Operator D in

Figure 3, which do not own resource domains but operate a multi-domain orchestrator to trade resources and services.

To sum up, 5G architecture enables new business opportunities meeting the requirements of a wide range of use cases, as well as enables 5G to be future proof by means of: (i) implementing network slicing in a cost-effective way, (ii) addressing both end-user and operational services, (iii) supporting softwarization natively, (iv) integrating communication and computation, and (v) integrating heterogeneous technologies (including fixed and wireless technologies).

2.1.2 5GEx

The 5GEx architecture framework, shown in Figure 4, identifies the main functional components and the interworking interfaces involved in multi-domain orchestration.

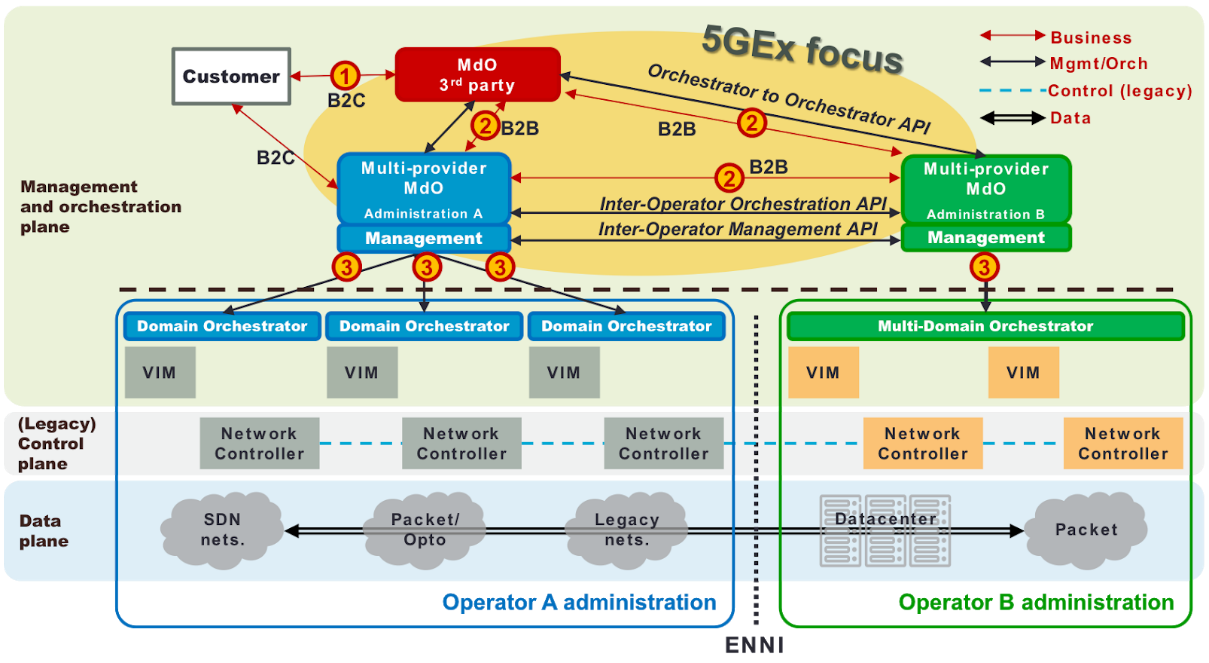

Figure 4. 5GEx architecture reference framework
(Source: 5GEx Deliverable D2.2).

The bottom part of Figure 4 shows different Resource Domains, hosting the actual resources. The middle part shows the Domain Orchestrators that are responsible of performing Virtualization Service Orchestration and/or Resource Orchestration exploiting the abstractions exposed by the lower Resource Domains. The key 5GEx component – the Multi-provider Multi-domain Orchestrator (MdO) – is shown at the top of Figure 4. The MdO handles the orchestration of resources and services from different providers, coordinating resource and/or service orchestration at multi-domain level, where multi-domain may refer to multi-technology (orchestrating resources and/or services using multiple Domain Orchestrators) or multi-provider (orchestrating resources and/or services using Domain Orchestrators belonging to multiple administrative domains).

There are three main interworking interfaces identified in the 5GEx architecture framework, briefly described next. The MdO exposes service specification APIs (Business-to-Customer, B2C) that allow business customers to specify their requirements for a service on Interface 1. The MdO interacts with other MdOs via Interface 2 APIs (Business-to-Business, B2B) to request and orchestrate resources and services across administrative domains. Finally, the MdO interacts with Domain Orchestrators via Interface 3 APIs to orchestrate resources and services within the same administrative domains.

In 5GEx, the provisioning of multi-domain services involves a series of actions between SPs consisting of 4 steps: (i) the discovery phase, for the distribution and population of the own capabilities, as well as the formation of the entire view of the multi-domain ecosystem by each of the service providers participating on it in the form of service offerings, (ii) the request phase, where the external customers solicit the provision of services, (iii) the fulfilment phase, where the lifecycle management of the required network functions is handled, and the necessary resources are configured and control, and (iv) the assurance phase, where the service environment is monitored and, as consequence of that, more control and management functions for lifecycle of the VNFs and configuration of resources could be performed for ensuring service levels.

The different actions identified above led to the need of a further splitting of the functionalities that the generic interfaces 1, 2 and 3 should support according to the following list:

- Service management (Ix-S)
- Catalogues (Ix-C)
- VNF lifecycle management (Ix-F)
- Resource / Topology (Ix-RT)

- Resource / Control (Ix-RC)
- Monitoring (Ix-Mon)
- SLA (Ix-SLA)

The x-S interface is used in the 5GEx architecture for requesting services. Those services can be requested by external customers, making use of 1-S interface, or can be requested between MdOs of different administrative domains, making then use of 2-S one.

In principle, no differences are foreseen between I1-S and I2-S variants, then the subsequent analysis is generalized and applied to both cases. To some extent, the capabilities of the Ix-S interface are similar to the ones required by the Ix-C, described next. While the Ix-C interface is mainly devoted for the sharing of information, the Ix-S interface is used for invoking the services as described through such information sharing process. From that point of view, it was sensible using the same implementation for both interfaces. That is is based on a YANG information model supported by ETSI.

Catalogues

Ix-C covers: (i) I1-C, which is the interface that allows the interaction with the local catalogue subsystem by the provider of the local domain and by the 5GEx customer, and (ii) I2-C, which is the east/west interface that interconnect catalogue subsystems in two different administrative domains in order to exchange the necessary information to build multi-domain services.

I2-C interface is defined as the interface between two different MdOs through which all the information related to their catalogues (containing Network Services and Network Functions - VNFs) is exchanged, connecting the local Catalogue Management module to its homonym in the neighbour domain.

VNF lifecycle management

The I2-F interface is used to communicate lifecycle management dependencies and workflows of Network Service parts or compound VNFs.

There are several inter provider network scenarios that involve communication between NFVOs that belong to different administrations. The I2-F interface is used to delegate NS and VNF lifecycle management for some selected components of the Network Service to another provider. Delegation of lifecycle management occurs in general during on-boarding of an NSD or a VNFD. This operation, however, may also take place dynamically as part of a specific NS/VNF instantiation.

Resource / Topology

The I2-RT (Resource Topology) interface is used by a MdO to exchange the network topology and resource information with other MdOs. The information collected through I2-RT enables a MdO to: (i) detect the existence of other domains, (ii) learn about the network connectivity between domains, (iii) acquire details about the resources and service capabilities of specific domains, (iv) obtain adequate details about specific domains needed for the placement of VNF in the global infrastructure, (v) set up connectivity between domains, if required, through I2-RC, (vi) orchestrate connectivity between VNFs of different domains through I2-RC.

Resource Control

The Ix-RC interface is used by the MdO to reserve, provision, configure and manage resources through other MdO’s. Two kinds of resources were mainly identified: IT resources and Network resources.

For IT domains, the resources are related to:

- vCPU and memory for the compute node, which are often bundle (e.g. small, medium, big in OpenStack),
- storage space and type of storage,
- IT connectivity between VMs, including remote access outside the IT domain.

For network domains related resource include:

- Bandwidth, loss, jitter, delay to characterize the QoS,
- End-points to determine the tail and head of the connectivity,
- Encapsulation of the packets to describe how the connectivity is rendered.

Monitoring

The realisation of the network service assurance in the context of 5GEx requires the design and implementation of proper mechanisms that allow performing on-demand monitoring of the services instantiated and orchestrated in the considered multi-domain, multi-provider scenario.

A first dimension to be considered for this process referred to the Service Level Agreement (SLA) coming with each submitted service request, which may include different conditions to be verified by checking specific values (*e.g.*, metrics, statistics, etc.) that are relevant for each Service Level Objective (SLO). A second dimension to be considered referred to the particular resources implementing a certain service instance, which are selected at the time of service deployment according to the outcome of the resource orchestration algorithms. The latter may also require the collection of measurements to be used as feedback to the service and resource orchestration processes, thus introducing a third dimension of complexity.

During the analysis it became evident the need for having a separation of concerns on the monitoring functionalities. The Monitoring interfaces were indeed decomposed into two separate sub-interfaces named Ix-Mon control and Ix-Mon data, being applied to both I3 and I2 interfaces.

I3-Mon control is the interface which is expected to provide functionalities for both managing probes lifecycle and requesting the collection/storage of measurements coming from resources related to the local running service instances. The purpose of I3-Mon control would then consist in defining a common way of remotely and dynamically controlling and orchestrating the configuration/activation of monitoring probes to collect and storing measurements from all the different resource domains that are involved in the realization of a given service instance.

The final goal of the probes' activation process consists in enabling the collection of relevant measurements to be used by the MdO management functions for the purpose of service assurance, orchestration of services and resources, etc.

This requirement implies that different measurements, coming from different resource domains but related to the same service instance, will somehow have to logically be linked and then conveyed to a common storage repository in the multi-domain orchestrator.

For the definition of this interface the focus was not on considering the particular mechanisms to be used while interacting with the repository for either writing measurement data (southbound part) or querying them (northbound part) as they may vary with the particular storage technology. In the case of I3-Mon data it was more sensible defining a common, agreed data model to be used when measurements are stored and retrieved.

I3-Mon data in fact required the definition of an abstraction between different resource domains acting as producers of monitoring data, and some MdO management functions taking the role of consumers of monitoring data.

SLA

I3-SLA covers: (i) the flow of monitoring information from the monitoring DB to the SLA manager for its evaluation, and (ii) the events communication between the SLA Manager and the Orchestrator.

I3-SLA interface is an internal domain interface between the local SLA Manager and the local monitoring DB, through which the SLA manager retrieves the monitoring information for the KPIs involved in the SLA contracts for the business transactions.

Every running instance has an associated set of KPIs that needs to be evaluated according to the terms of the SLA. Periodically, the SLA manager will contact the monitoring DB looking for monitoring information for each of the KPIs. This process is split in two steps: the SLA Evaluator will contact the SLA Aggregator which then will process the request. If it is a simple KPI evaluation, the petition will be forwarded to the monitoring DB and the information will be sent back to the SLA Evaluator. If we are dealing with a complex KPI, the SLA Aggregator will create as many requests as needed to the monitoring DB to retrieve all the individual samples for each of the simple KPIs that compose the complex one. After that, the information will be aggregated and sent for evaluation [5GEX].

2.1.3 ITU-T

As well as several SDO, the International Telecommunications Union (ITU) has tried to establish a common ground for all future mobile-broadband communications ecosystem actors, coining the term IMT-2020 (International Mobile Telecommunication system - 2020) to embrace all the efforts to provide an international specification for 5G.

The concept of network slicing stands among the novel concepts therein, as a strategy to build efficient and cost-effective infrastructures that can be shared by several services. According to [GALIS2017], a network slice is "a managed group of subsets of resources, network functions / network virtual functions at the data, control, management/orchestration, and service planes at any given time. The behaviour of the network slice is realized via network slice instances (*i.e.*, activated network slices, dynamically and non-disruptively re-provisioned). A network slice is programmable and has the ability to expose its capabilities".

For ITU-T, network slicing is perceived as Logical Isolated Network Partitions (LINP). According to Recommendation ITU-T Y.3011, a LINP is composed of multiple virtual resources, whose capability may be not bound to the capability of the physical or logical resource, which are isolated and equipped with a programmable control and data plane.

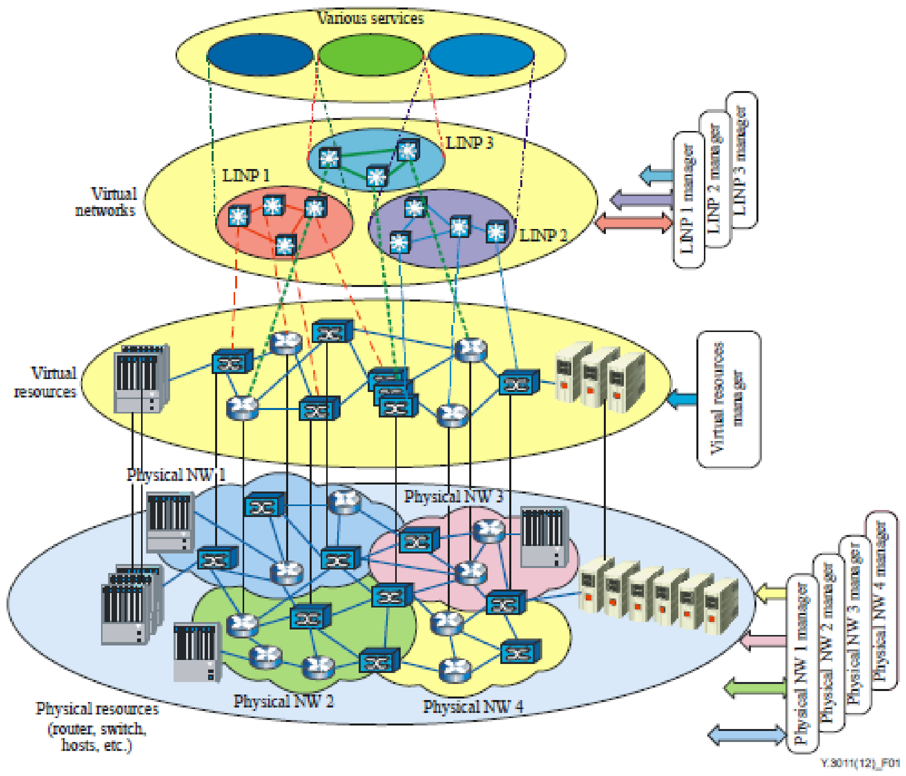

Figure 5. Conceptual architecture of network virtualization (Source: Recommendation ITU-T Y.3011 - 01/2012).

Thus, network virtualization is seen as a method that allows multiple LINPs to coexist in a single physical network. Figure 5 presents the conceptual architecture of network virtualization. It can be seen that a single physical resource can be shared among multiple virtual resources and each LINP consists of multiple virtual resources. Each LINP is managed by an individual LINP manager. Moreover, the physical resources in a physical network(s) are virtualized and may form a virtual-resource pool, which is managed by the virtual resources manager (VRM). The VRM interacts with the physical network manager (PNM) and performs control and management of virtual resources.

Figure 6 depicts the LINP concept and coexistence among a multitude of LINPs, comprising several resources, physical or virtual, to support network virtualization. As presented in Recommendation ITU-T Y.3011 – 01/2012, there shall have a strict relationship between a LINP and user requirements. Such requirements provide a basis for VRMs to coordinate the allocation of appropriate LINPs to a given user/set of users, based on VRM administration policy. Moreover, each LINP is controlled and managed by an LINP manager. The VRM which is controlling all virtual resources creates an LINP manager and allocates appropriate authorities to control each LINP.

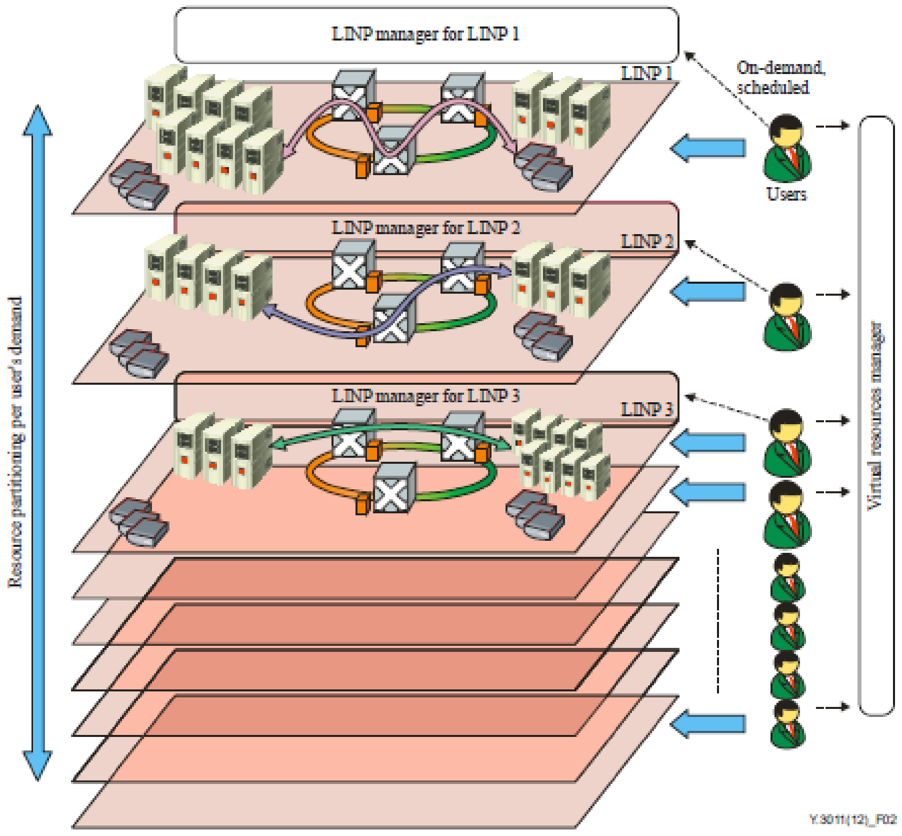

Figure 6. Concept of LINP provided by network virtualization
(Source: Recommendation ITU-T Y.3011 – 01/2012).

A LINP generated by network virtualization has various characteristics, such as partitioning, isolation, abstraction, flexibility or elasticity, programmability, authentication, authorization, and accounting. However, to achieve such a set of goals, ITU has identified several missing points, based on a gap analysis cited in [ITU-T Y.3011]:

- Lack of an unified network management structure;
- Non-standardized Operations and management (OAM) protocols;
- Non-existing strategies to manage and orchestrate the softwarized network components, as well as to softwarize network management and orchestration functionality; and
- Lack of a "network slice-driven" lifecycle management and orchestration.

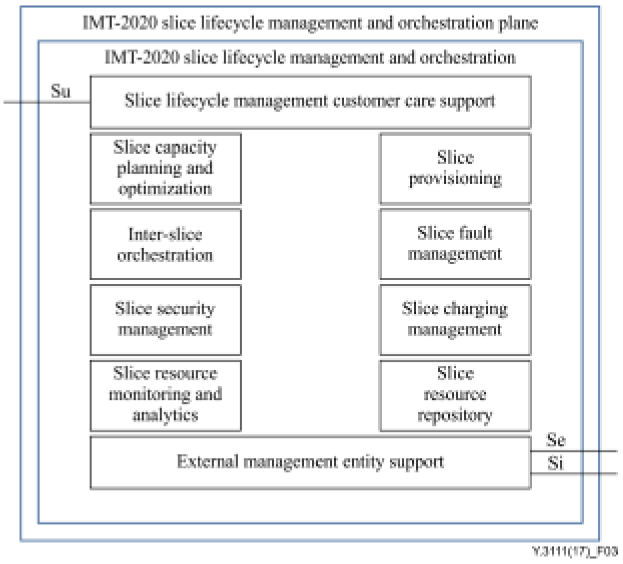
IMT-2020 slice lifecycle management and orchestration plane
IMT-2020 slice lifecycle management and orchestration
Su
Slice lifecycle management customer care support
Slice capacity planning and optimization
Slice provisioning
Inter-slice orchestration
Slice fault management
Slice security management
Slice charging management
Slice resource monitoring and analytics
Slice resource repository
Se
Si
External management entity support
Y.3111(17)_F03

Figure 7. Network slice lifecycle management and orchestration functional components (Source: Recommendation ITU-T Y.3011).

Moreover, it is considered that the management and orchestration architecture in IMT-2020 is also required to deal with two levels: network slice life-cycle management, as well as in each network slice instances – Instances 1 and 2, respectively. Given this architectural approach, Recommendation ITU-T Y.3011 specifies a network management and orchestration framework for IMT-2020 in order to accomplish the goals set above.

According to this ITU recommendation, network slice orchestration functionalities are specified in the functional elements: slice capacity planning and optimization, slice provisioning (SP), and inter-slice orchestration, while the management functionalities are specified in the functional elements slice fault/security/charging management, slice resource monitoring and analytics and resource repository, working together to achieve the slice lifecycle management objectives.

Figure 7 shows this set of functional components.

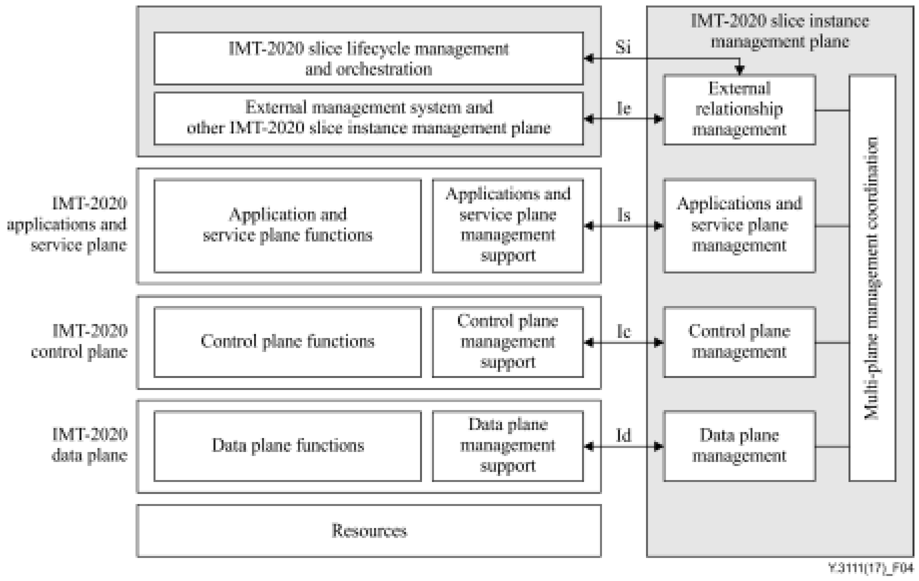
IMT-2020 slice lifecycle management and orchestration
Si
IMT-2020 slice instance management plane
External management system and other IMT-2020 slice instance management plane
Ie
External relationship management
IMT-2020 applications and service plane
Application and service plane functions
Applications and service plane management support
Is
Applications and service plane management
Multi-plane management coordination
IMT-2020 control plane
Control plane functions
Control plane management support
Ic
Control plane management
IMT-2020 data plane
Data plane functions
Data plane management support
Id
Data plane management
Resources
Y.3111(17)_F04

Figure 8. Network slice instance management functional architecture (Source: Recommendation ITU-T Y.3011).

Similarly, ITU-T Y.3011 also specifies the network slice instance management functional architecture and components, as well as its relationships and interfaces with slice lifecycle management and orchestration functional component and external management systems, as illustrated in Figure 8.

Recommendation ITU-T Y.3011 Section 11 presents the slice lifecycle management procedure, as stated below in brief:

- The IMT-2020 customer requests a slice to be provisioned with its specified service requirements;
- IMT-2020 slice lifecycle customer care support (SLMCCS) functional element receives the customer's request and carries it to the slice capacity planning and optimization functional element (SCPO). SCPO then determines an optimal slice plan based on the available resources which matches the customer's request;
- Once the provisioning policy is determined, SCPO requests provisioning to slice provisioning (SP) functional element. SP then performs the requested slice provisioning task. Upon completion of the provisioning process, SP sends a provision reply message to the customer via SLMCCS. At the same time, it sends a provision status slice resource monitoring and analytics functional element to initiate the collection and monitoring of the provisioned resources. It also sends the status update to slice resource repository (SRR) to store the provisioned resource information;
- Slice Resource Monitoring and Analytics (SRMA) performs collection, monitoring, and analysis tasks of the provisioned slice resources. Data and information collected and analysed is then stored in SRR for further processing by other functional elements;
- When SRR receives any resource status updates, it stores them in the repository and, at the same time, it emits notification to all functional elements that are listening to the status updates (slice fault management - SFM, slice security management - SSM, slice charging management - SCM, and SCPO);
- When SCPO receives the notification, it updates available resource status and determines if re-optimization is needed upon status updates;
- SP, upon receiving the provisioning update requests, performs re-provisioning tasks for the provisioned slices, generating provision status reports to the related functional elements.

In April 2018, ITU-T has released Draft Recommendation ITU-T Y.3112 (Y.IMT2020-MultiSL) - Framework for the support of Multiple Network Slicing. As presented in its summary, this Recommendation describes the concept of network slicing and use cases of multiple network slicing, enabling a single device to simultaneously connect to different network slices. The use case describes the slice service type for indicating a specific network slice and the slice user group for precisely representing the network slice in terms of performance requirements and business models. Finally, it also specifies the high-level requirements and high-level architecture for multiple network slicing in IMT-2020 network.

As far as it can be seen, APIs for network slicing in clouds is still a pending issue for ITU-T IMT-2020 future evolvements.

2.1.4 Open Grid Forum

The Open Grid Forum (OGF) is committed to the evolution and adoption of advanced applied distributed computing, such as cloud, grid, and networking, through a highly involved open community. The OGF aims at developing and promoting innovative scalable techniques, applications, and infrastructures in order to increase productivity in both enterprise and academy communities. The open community consists of thousands of individuals spread out in industry and research, representing more than 50 countries, over 400 organizations.

The work is carried out through community-initiated working groups that collaboratively develop standards and specifications with other leading standards organizations, software companies, and future users. Its organizational members, including technology companies and research institutions in academia and government, are responsible for funding the OGF. Several events to further develop grid-related specifications and use cases are hosted by OGF each year.

Currently, the OGF working groups have been studying several proposals, and the most relevant to NECOS is the Open Cloud Computing Interface (OCCI), which focuses on the cloud computing IaaS based model. OCCI is a protocol and API that aims to enable the development of interoperable tools for common tasks, including deployment, autonomous scaling, and monitoring.

As shown in Figure 9, the OCCI interface is a boundary protocol and API that acts as a service front-end to a provider's internal management framework, and it is placed in a provider's architecture.

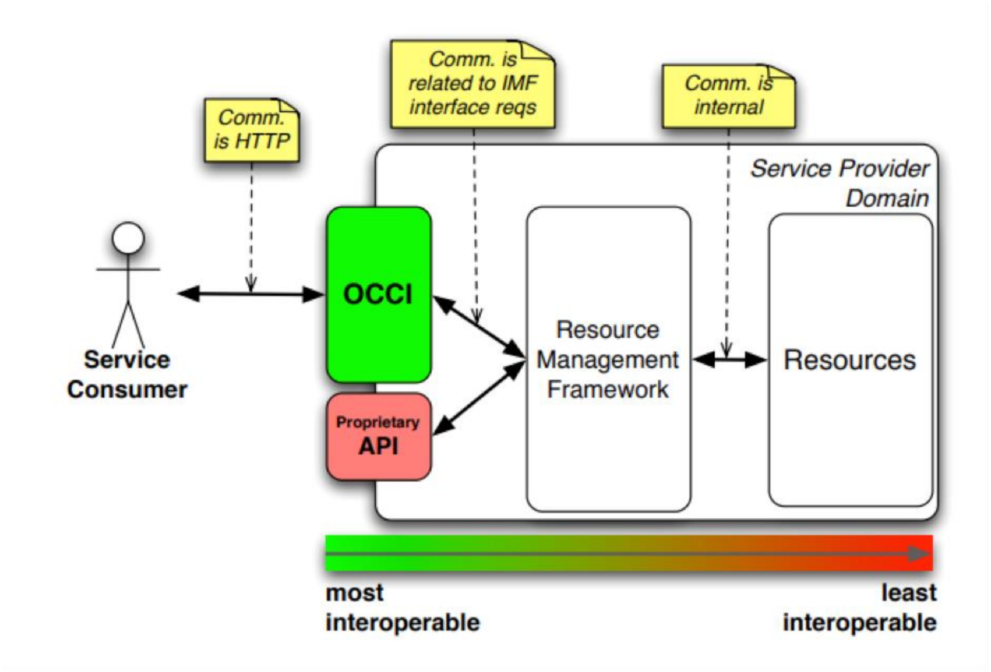

Figure 9. OCCI Interface (Source: OCCI, http://occi-wg.org/about).

End-users and other system instances can be seen as service consumers, and OCCI is suitable for both cases. As a key feature, it can be used as a management API for all kinds of resources, while at the same time maintaining a high level of interoperability.

In summary, OCCI is able to abstract and generalize methods or call specific functions of a particular VIM (or any other management software). It does not have an API to directly instantiate or monitor a slice. Some close features can be found at the OpenStack OCCI Interface implementation (https://github.com/openstack/ooi), as shown in Figure 10, which is capable of invoking OpenStack standard commands in a generic way.

In a general way, with the exception of OCCI, OGF does not have standards that may contribute to the specification of both Client-to-Cloud and Cloud-to-Cloud APIs. That is, the OGF still has no efforts aimed at slicing or networking slice. There are OCCI implementations that leverage communication with some VIMs, such as OpenStack, OpenNebula and CloudStack; but with a very specific focus on cloud computing.

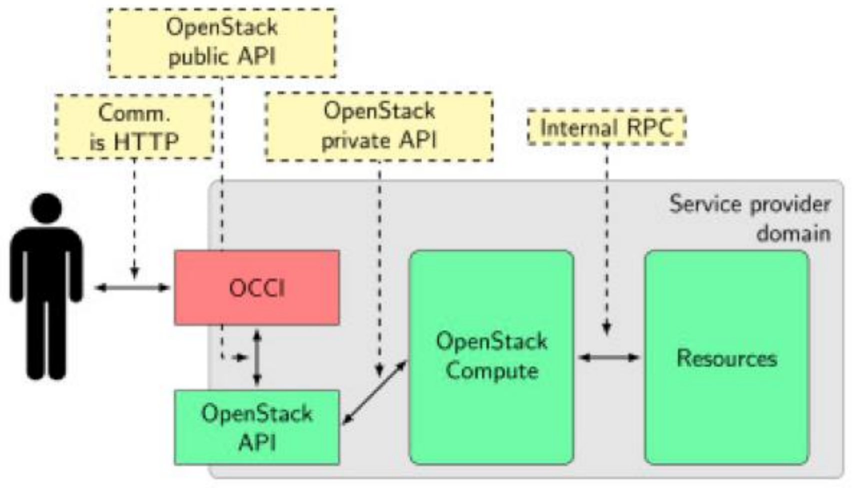

Figure 10. Openstack OCCI interface implementation
(Source: OCCI, http://occi-wg.org/tag/openstack).

2.1.5 Initiatives in Other Standards Development Organizations

Some other Standards Development Organizations (SDOs), as well as industrial associations are looking at the network slice concept from different angles and perspectives [CONTRERAS18]. From the provider's point of view, there is a risk of fragmenting the conceptual approach to network slices, since small differences can provoke incompatibilities among the different approaches. It is, therefore, necessary to reach consensus on common terms, definitions, rationale, ideas, and goals to properly normalize the concept of network slicing.

The NGMN Alliance [NGMN] has provided a primary description of the network slice concept as mentioned in the introductory section. The NGMN view is that of a 5G slice as a composition of a collection of 5G network functions and specific Radio Access Technology settings that are combined for the specific use case or business model, while leveraging NFV and SDN concepts. The network slice concept is organized in a layered manner [NGMN], differentiating the service instance layer, comprising the end-user of business services; the network slice instance (NSI) layer, as a set of functions forming a complete instantiated logical network; and the Resource layer, consisting of both physical and logical resources. In this layered view, the NSIs can be potentially shared among multiple service instances.

3GPP [3GPP] differentiates among network slices and network slice instances. On one hand, a network slice represents a logical network providing specific network capabilities and network characteristics. On the other hand, a network slice instance is defined as a deployed network slice, that is, a specific set of network function instances and associated resources.

ETSI NFV [NFV] specifies network operators' perspectives on NFV priorities for 5G, network slicing support with ETSI NFV architecture and an E2E network slicing framework. Another recent development within ETSI Zero Touch Network and Service Management Industry Specification Group (ZSM ISG) is specifically devoted to the standardization of automation technology for network slice management [GOTO18]. Within the ETSI Multi-access Edge Computing (MEC) group, a new work item called "MEC support for network slicing" [MEC] seeks to identify the necessary support for network slicing, evaluating the gaps from MEC features and functions, and identify the new requirements.

The BBF [BBF] is also approaching network slicing by augmenting the previous management functions by defining new and complementary ones, such as Access Network Slice Management, Core Network Slice Management, and Transport Network Slice Management. Each one of them is intended to take care of the slice lifecycle management of each particular network slice subinstance (*i.e.*, access, core, or transport).

2.2 Information models

2.2.1 NOVI

The FP7 project NOVI [novi2015] defined an architecture for supporting federation of infrastructures. NOVI has a Service Layer that allow users to have a unique interface to access and use resources in different testbeds. Access to the testbeds, authorization policies, monitoring information and selection of resources are integrated among platforms and implemented in the NOVI layer. The project identified the definition of a common Information Model (IM) as the essential element to achieve the federation goals. Their model was intended to support virtualized resources and context-aware resource selection; to be vendor independent; to support monitoring and measurement concepts, and to support management policies.

NOVI uses Web Ontology Language (OWL) for modelling virtualization explicitly for both, computing and networking devices using Web Ontology Language (OWL). It is vendor-agnostic, modular, and composed of three main ontologies: (i) resource ontology, (ii) monitoring ontology and (iii) policy ontology.

Resource Ontology

Figure 11 partially depicts NOVI's resource ontology. Besides the ontology-based substrate, it is a more or less common representation of resources and services.

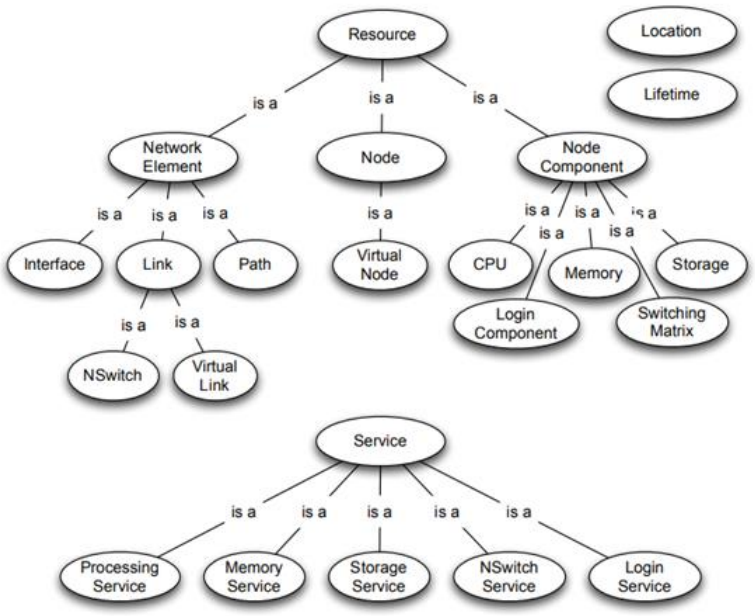

Figure 11. NOVI ontology for modelling resources and services (Source: [novi2015]).

In particular, it is worth noting the classes "Location" and "Lifetime". The former is an approximate geographical location to describe which resources share the same location. It can also be extended with properties such as GPS coordinates. The latter is used to describe the time dimension of a reservation, but it can also be used to describe the availability of nodes, *e.g.*, that a node is not available during a maintenance period.

The Service class allows the user to express the desired service-level. This allows the user to decouple the service request from the actual physical implementation.

Figure 12 shows how network elements are connected in paths and nodes with unidirectional links. The authors justify this decision saying that "has been a conscious choice to follow the Network Markup Language (NML) [NML] model, which follows the philosophy that a unidirectional model can describe a bidirectional model, but not vice versa".

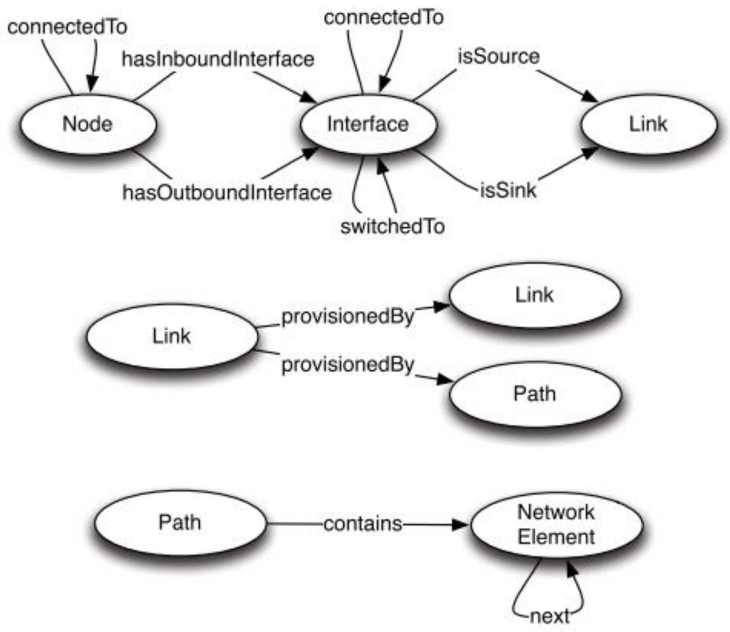

Figure 12. Network connectivity properties defined in the NOVI resource ontology (Source: [novi2015]).

Monitoring Ontology

Figure 13 depicts NOVI's Monitoring Ontology. An interesting aspect is the inclusion of modules, such as the Query model or the Statistic Model, which are semantic bridges between data consumers and data generators.

This model is designed to store static and dynamic information. By static information, NOVI refers to constant characteristics of the resource, or those that may change very infrequently (*e.g.*, number of CPUs in a server). Dynamic information encompasses attributes, such as the utilization of a CPU core. In NOVI, the description of resources is carried out at two abstraction levels: physical resources, and virtual resources. At the physical resource level, when the user requests a virtual testbed (a Topology), it may contain runtime, dynamical constraints, such as CPU or main memory. At the virtual level instead, there is the monitoring support for the virtual testbed. Given that the user successfully acquires a virtual topology, NOVI offers services to keep track of its certain temporal variables. For instance, a user is interested in the evolution of the round trip delay in certain links of his topology. These two complementary abstraction levels split between substrate monitoring and slice monitoring.

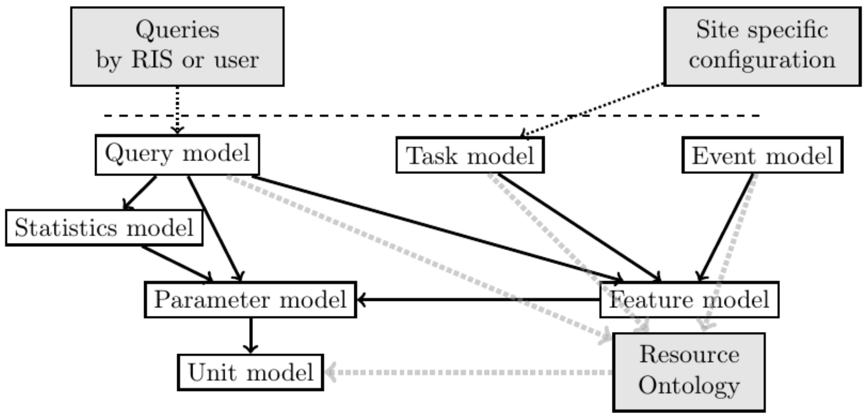

Figure 13. NOVI's modular ontology for modelling monitoring data and tasks (Source: [novi2015]).

It is also interesting to see the existence of the unit model where the fundamental concepts of the Monitoring Ontology are laid down; these are definitions of levels, dimensions, units and unit prefixes. This yields clear semantics to the data stored in the monitoring ontology.

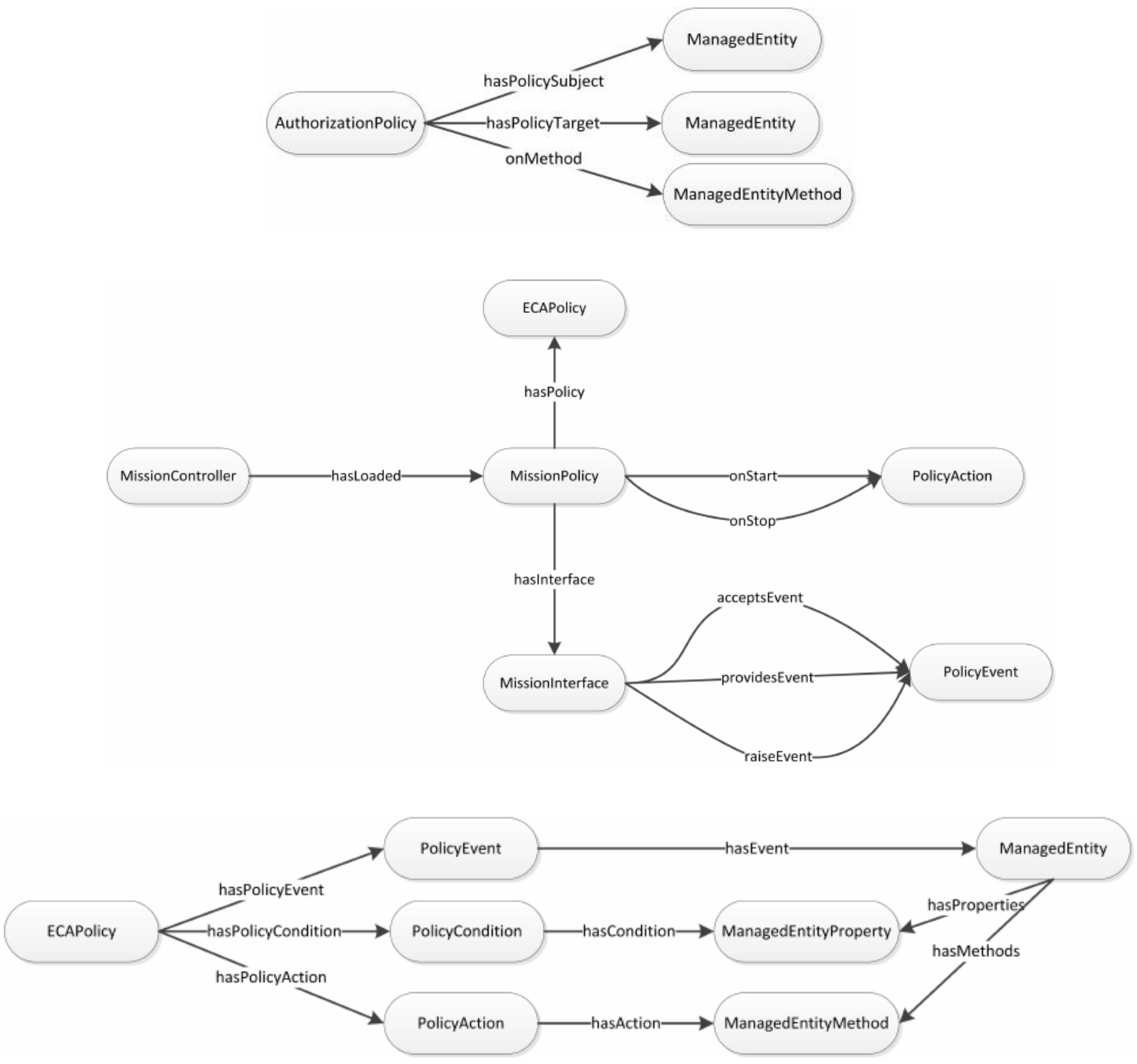

Figure 14. NOVI policy ontology types (Source [novi2015]).

Policy Ontology

NOVI's Policy Ontology, as depicted in Figure 14, includes the classic authorization policies and ECA policies. In addition to them, NOVI models an interesting entity from the NECOS' point of view, the Mission Policies, used to define inter-platform duties, *i.e.*, the management obligations that a platform must fulfil against its peer platform in a NOVI federation.

2.2.2 ETSI NFV MANO

The European Telecommunications Standards Institute (ETSI) has defined a framework for Network Functions Virtualization (NFV) and Management and Orchestration Architectures (MANO). More specifically, MANO, as a top-down approach, consists of three main software layers:

- NFV Orchestrator (NFV-O) is responsible for network service management, such as to create virtual function instances to meet service requirements. NFV-O responsibilities include the onboarding of new Network Service (NS) and the NS lifecycle management. Other functionalities include the global resource management (topology of the connected VNFs and PNFs), the authorization of NFV infrastructure resource requests and the policy management for NS instances.
- VNF Manager (VNF-M) manages the lifecycle of the components and services. The VNF-M supervise the management of the VNF instances, *i.e.*, VNF starts from VNF-M descriptor and is managed by VNF-M, also VNF-M determines the health of the VNF.
- Virtualized Infrastructure Manager (VIM) manages NFV infrastructure resources in a single domain. VIM controls and manages the NFV infrastructure resources in one operator's infrastructure sub-domain. Moreover, the VIM is responsible to collect and forward the network performance measurements.

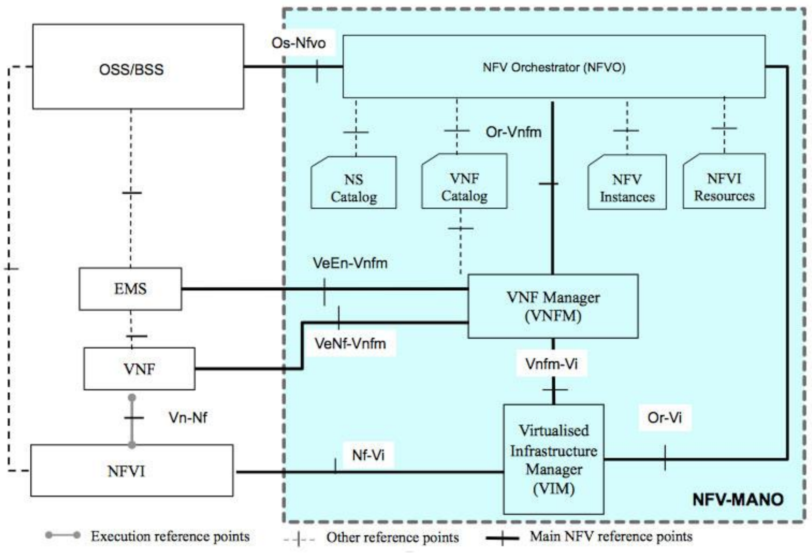

Figure 15. NFV MANO reference architecture (Source: ETSI NFV MANO WI document).

As shown in Figure 15, apart from the aforementioned 3-layers, the NFV MANO consists of 4 types of data repositories (databases that keep different types of information):

- The NS catalog, which is a set of pre-defined templates that define how services may be created and deployed; the same repository stores the connectivity parameters through virtual links for future use.

- The VNF catalog, which is a set of templates that describe the deployment and operational characteristics of available VNFs.

EUB-01-2017

- The NFVI resources repository, which maintains information about available/allocated NFVI resources.

- NFV instances repository, which maintains information about all function and service instances throughout their lifetime.

MANO Information Model

ETSI OSM is delivering an open source Management and Orchestration (MANO) stack aligned with ETSI NFV Information Models that focuses on network service orchestration. Information in a network service (NS) is structured into information elements, which might contain a single value or additional information elements that form a tree structure. Information element are classified as one of the following types: leaf element (single information element), reference element (information element that contains a reference to another information element) and sub-element (information element that specifies another level in the tree). The information elements can be used in two different contexts: as descriptors or as run-time instance records. A descriptor is defined as a configuration template that defines the main properties of managed objects in a network.

The network service descriptor (NSD) is the top-level construct used for designing the service chains, referencing all other descriptors that describe components that are part of that network service. The NSD consists of static information elements that describe deployment flavors of the network service. The NSD is used by the NFV orchestrator to instantiate a network service.

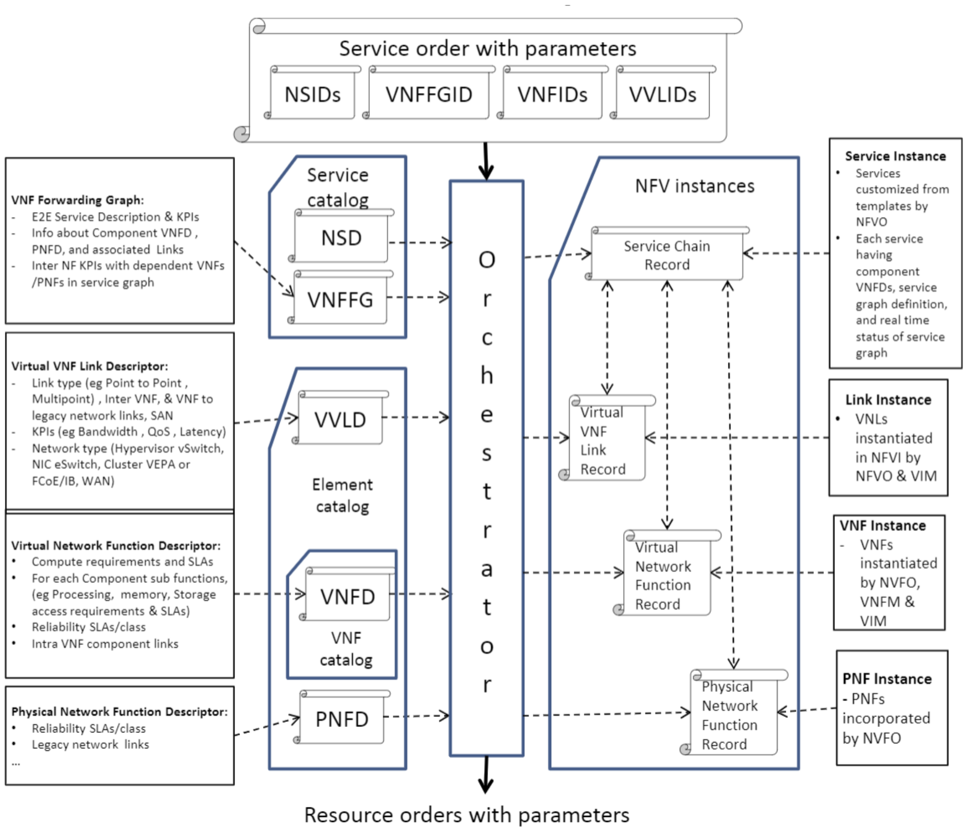
Service order with parameters
NSIDs
VNFFGID
VNFIDs
VVLIDs
VNF Forwarding Graph:
- E2E Service Description & KPIs
- Info about Component VNFD, PNFD, and associated Links
- Inter NF KPIs with dependent VNFs /PNFs in service graph
Virtual VNF Link Descriptor:
- Link type (eg Point to Point, Multipoint), Inter VNF, & VNF to legacy network links, SAN
- KPIs (eg Bandwidth, QoS, Latency)
- Network type (Hypervisor vSwitch, NIC eSwitch, Cluster VEPA or FCoE/IB, WAN)
Virtual Network Function Descriptor:
• Compute requirements and SLAs
• For each Component sub functions, (eg Processing, memory, Storage access requirements & SLAs)
• Reliability SLAs/class
• Intra VNF component links
Physical Network Function Descriptor:
• Reliability SLAs/class
• Legacy network links
...
Service catalog
NSD
VNFFG
VVLD
Element catalog
VNFD
VNF catalog
PNFD
Orchestrator
NFV instances
Service Chain Record
Virtual VNF Link Record
Virtual Network Function Record
Physical Network Function Record
Service Instance
• Services customized from templates by NFVO
• Each service having component VNFDs, service graph definition, and real time status of service graph
Link Instance
• VNLs instantiated in NFVI by NFVO & VIM
VNF Instance
- VNFs instantiated by NVFO, VNFM & VIM
PNF Instance
- PNFs incorporated by NVFO
Resource orders with parameters

Figure 16. MANO information model (Source: ETSI NFV MANO WI document).

The following four information descriptors are defined apart from the top-level network service (Figure 16):

- Virtual network function (VNF) information element, which is a deployment template that describes the attributes of a single VNF.

- Physical network function (PNF) information element, which describes a physical (legacy) network function and includes only the interconnections (connection points and virtual links). The PNF descriptor is needed if the network service includes a physical device to support network evolution.

- Virtual Link (VL) information element, which describes the resource requirements needed for a link between VNFs, PNFs and end-points of the network service, which could be met by various link options that are available in the NFVI.

- VNF forwarding graph (VNFFG) information element, which is a graph, specified by a network service provider, of bi-directional logical links that connect network function nodes, where at least one node is a VNF through which network traffic is directed.

Software that provides VNFs can be structured into software components, the implementation view of a software architecture. These components can then be packaged into one or more images, the deployment view of a software architecture. These software components are called Virtual Network Function Components (VNFCs). VNFs are implemented with one or more VNFCs, where each VNFC instance generally maps 1:1 to a VM image or a container, as defined in the VDU.

Reference Points

MANO has multiple reference points that appear as interconnection points between the functional blocks, as shown in Figure 15, *i.e.*, *Or-Vi*, *NF-Vi*, *Or-Vnfm*. Designed with open, standards-based APIs, such as NETCONF and REST, and common information models, such as YANG, the *Os-ma-nfvo* interface is exposed through open, standards-based interfaces, such as REST. This design enables upper-level orchestrators, such as Business Process Orchestrators or Service Orchestrators, to automate the entire service bring-up process.

2.2.3 4WARD

The 4WARD project [4WARD] designed an architecture for the provisioning and management of service-tailored virtual networks across multiple administrative domains (Figure 17). 4WARD relies on a centralized coordinator, namely Virtual Network Provider (VNP), for virtual network deployment. In particular, the VNP receives virtual network topology requests from a client, and subsequently partitions the request across the participating infrastructure providers. Then, each infrastructure provider receives his corresponding virtual network segment, which he embeds onto his physical topology. Finally, the topology segments are stitched together to form a virtual network requested by the clients.

In this context, the project has specified an information model for the description of the infrastructure and the virtual network resources. The information model is used by all actors to specify and exchange virtual/physical resource information during resource advertisement, assignment, monitoring, and allocation of virtual networks. The model uses the abstraction “Network Element” to describe nodes, interfaces, links, and paths. Each resource element has a unique identifier and a set of attributes. The information model ensures the binding of different elements, such as the binding of links with paths, interfaces with nodes, and so forth.

Figure 18 shows an UML diagram that expresses the relationships between virtual resources. UML is a modeling language that separates the conception phase from the implementation phase and provides a generic description that is implementation independent. 4WARD has particularly selected XML for the implementation of the information model, represented with UML.

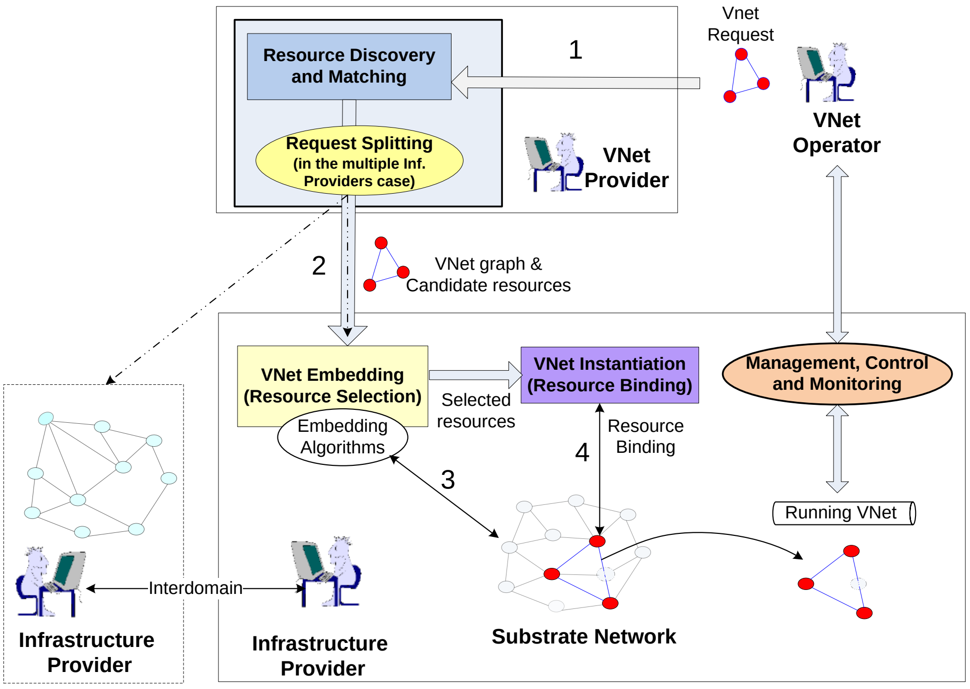
Vnet Request
1
Resource Discovery and Matching
Request Splitting (in the multiple Inf. Providers case)
VNet Provider
VNet Operator
2
VNet graph & Candidate resources
VNet Embedding (Resource Selection)
Embedding Algorithms
Selected resources
VNet Instantiation (Resource Binding)
Resource Binding
4
3
Management, Control and Monitoring
Running VNet
Interdomain
Infrastructure Provider
Infrastructure Provider
Substrate Network

Figure 17. The 4WARD approach to network virtualization (Source: 4WARD project presentation)

The 4WARD information model provides a detailed description of infrastructure and virtual resources to meet the requirements of virtual network provisioning. The different levels of abstractions offered by the model comprise an additional benefit. However, in the context of network slicing, the specific model exhibits considerable limitations. First, there are not sufficient elements and attributes in the model to express service elements (e.g., vNFs) as well as vNF graphs. Hence, this model cannot support the slice specification requirements in all NECOS modes (*e.g.*, *Mode 3* which is associated with service-oriented slice requests). Furthermore, the 4WARD information model does not include descriptors for dedicated infrastructure elements, such as switches, routers, Wi-Fi access points, and base stations. Another limitation of the model is the lack of support for extended platform awareness (EPA), *i.e.*, the exposure of certain capabilities of the infrastructure to the tenant for slice deployment. EPA is a significant feature which needs to be incorporated into information models, given the diversity of features available in modern commodity servers and cloud platforms.

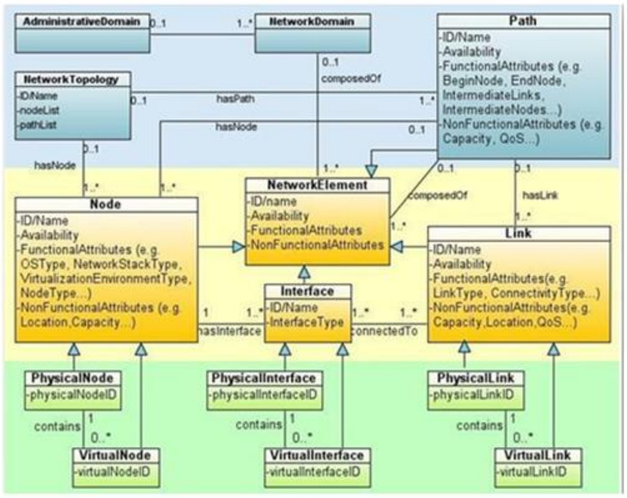

Figure 18. UML diagram of 4WARD information model (Source: [MEDHIOUB2011).

2.2.4 COMS

The Common Operation and Management of network Slicing (COMS) [ietfcoms2018a] aims at providing a comprehensive approach for the overall operation and management of network slicing, for both network slice operators and network slice tenants. Working on the top network orchestrator inside Transport Network region which directly communicates with the network slice provider, COMS enables technology-independent network slice management [ietfcoms2018b]. In this context, COMS provides a technology-independent information model for transport network slicing [ietfcoms2018c].

The COMS idea of a general information model serves the need to fill the gap between technology-agnostic network slicing service requirements, usually desired by the tenants, and technology-specific slices' implementation, typically supported by the service providers. Such a model describes the entities that a network slice consists of, along with their properties, attributes, operations and the way they relate to each other, whilst it remains independent of any specific repository, software, protocol or platform.

The COMS information model uses the data model for network topologies as a base [ietfdata2018] and enhances it with new slice-specific attributes under the "netslice" namespace. COMS uses the YANG data modeling language [RFC7950] to make a technology independent representation of the transport network slicedata model. This information model includes, among others, the following elements, which are represented in Figure 19:

- **connectivity resources**: refer to nodes and links that represent virtual nodes and links exposed to the slice user. The COMS augments these two elements with further new attributes, compared to the model [ietfdata2018], in order to represent requirements, configuration and statistics associated with a node (*i.e.*, sent/received-packets) and QoS information associated with a link (*i.e.*, link-bandwidth-agreement).

- **storage resources**: the location attribute describes the location of the storage unit, and other interesting for NECOS' modelling attributes include access-mode (public or dedicated) and read-write-mode with read-only and read/write options.

- **compute resources**: the location attribute describes the location of the compute unit, and other interesting for NECOS' modelling attribute is access-mode (shared or dedicated).

- **service instance based on predefined function blocks**: some general features can be grouped into function blocks in advance, such as load-balancer, firewall.

- **network slice level attributes**: defines a set of attributes directly applicable to a network slice, such as service-time-start/end and lifecycle-status (*i.e.*, construction, modification, activation, deletion).

COMS is a simple model that allows for extension and covers some of NECOS' modelling needs regarding its networks domain. It is interesting to note that the model defines a set of operations that must be supported for the complete network slice. However, apart from the network slice as a whole, each element insides a network slice should also be able to be operated individually. Operations defined by the COMS are:

- **construct**: construct a network slice,
- **delete**: delete a network slice,
- **modify**: modify a constructed network slice,
- **set_element_value**: set the value of an indicated element in a network slice,
- **get_element_value**: get the value of an indicated element in a network slice,
- **monitor**: monitor the status of a network slice, and
- **enable_report**: enable the active report to the subscribes/management system when the monitored status changes beyond expectation.

The aforementioned operations map to NECOS slicing model but they must be extended to accommodate its different slicing modes.

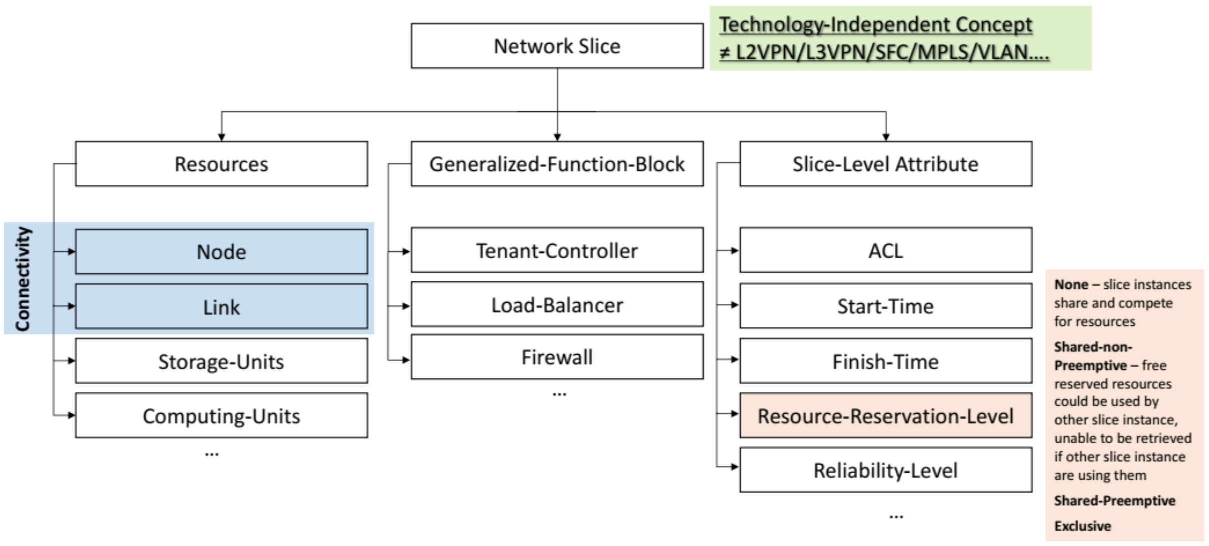
Network Slice
Technology-Independent Concept
≠ L2VPN/L3VPN/SFC/MPLS/VLAN….
Resources
Generalized-Function-Block
Slice-Level Attribute
Connectivity
Node
Link
Storage-Units
Computing-Units
…
Tenant-Controller
Load-Balancer
Firewall
…
ACL
Start-Time
Finish-Time
Resource-Reservation-Level
Reliability-Level
…
None – slice instances share and compete for resources
Shared-non-Preemptive – free reserved resources could be used by other slice instance, unable to be retrieved if other slice instance are using them
Shared-Preemptive
Exclusive

Figure 19. COMS tree of attributes for a slice (Source: [ietfcoms2018b]).

2.3 Summary

In summary, the context of Slice-as-a-Service, as promoted by NECOS, requires the specification of an Information Model that will describe both the infrastructure resources/elements along with their properties, as well as the slice components and the service elements deployed on top of them. The NECOS information model should exploit significant features of state-of-the art models, while overcoming limitations mainly related to network slicing and multi-domain physical infrastructures.

The NOVI information model provides descriptors for resources and services, but it lacks of support for slice specification. Hence, it could not be adopted in the NECOS context. However, some of its classes, *e.g.*, the *Location* and *Lifetime* are useful and can be exploited by the NECOS information model to indicate and satisfy geo-location constrains, and specify time-related requirements for the creation and decommission of a slice. Limitations regarding the slice description and attributes to express service elements also exist in the 4WARD information model. The 4WARD model is primarily designed to specify objects and attributes of the infrastructure and the virtual network resources. The most suitable information model for network slicing is the COMS model. It provides a set of useful slice-specific attributes for NECOS, such as the starting and ending time of a service, the slice lifecycle status, as well as attributes related to slice requirements, *e.g.*, reliability levels, throughput threshold, latency or jitter agreement. The MANO information model also provides useful features with the EPA support being the most notable. In particular, the notion of EPA hides the heterogeneity among providers and exposes certain infrastructure features to the tenant. Host, hypervisor, VIM, vSwitch, interface and service-end are some of the EPA attributes incorporated in the NECOS information model. Overall, the NECOS model incorporates features from the 4WARD, COMS and MANO models.

Along with the provisional information model described in the following section, cloud API are further defined exploiting the infrastructure and network slicing features exposed by the information model. Regarding the APIs, the NECOS takes advantage of the 5GEx proposal, which defines three main network interfaces: (i) between the customer and the multi-domain orchestrator for service exposure (B2C), (ii) for multi-domain orchestrators' interaction (B2B) and (ii) for the interaction between the multi-domain orchestrator and the domain orchestrators within the same administrative domain. In accordance with the aforementioned scheme, NECOS has defined *Client-to-Cloud APIs* and a set of interfaces composing the *Cloud-to-Cloud APIs*, which will be enhanced with slice-related methods for slice provisioning and run-time management. An API for the deployment, scaling and monitoring of infrastructure resources has been also specified by the OGF, (*i.e.*, OCCI); however, this API does not support network slicing.

3 NECOS Information Model

This section presents the first version of the NECOS information model used for slice provisioning and run-time management. The main objectives of the information model are: (i) the detailed description of all infrastructure resources/elements and their properties, and (ii) the description of slice components and service elements that could potentially be deployed within slices. To meet these requirements, we introduce an information model for network slicing, which provides resource descriptions at different levels of abstraction.

Initially, we consider three abstraction levels which are equivalent to the *Slice Database*, the *Tenant* and the *Infrastructure Provider* viewpoints (see deliverable D3.1). In Section 3.1, we discuss a high-level representation of the whole model, which currently resembles the *Slice Database* view, since the latter component keeps track of all required information for the slice operation. In Section 3.2, we describe in detail the slice specification model, *i.e.*, the objects and properties of the information model required to specify/request slices and address certain slice components or service elements deployed within slices; this comprises the *Tenant*'s view of the model. In Section 3.3, we provide a detailed specification of the model's objects and properties for the infrastructure description. This is essentially the *Infrastructure Provider*'s view, which is detailed, and its limited representation is communicated through the *Slice Agent* and *Slice Controller* components to the NECOS architecture (see D3.1), due to competition purposes, *i.e.*, *Infrastructure Providers* are not willing to share detailed information about their infrastructure. In Section 3.4, we consider a web load-balancing slice example and highlight key features of the NECOS information model using simple YAML descriptions.

At this point of investigation, we work towards defining the unified NECOS information model. Such complete definition requires the input of the Proof-of-Concepts implemented at the end of the first year, which will exercise the first model definition. In the following, we discuss in model detail the three views of the model.

3.1 Information model overview

A model instantiation very close to the unified NECOS information model is at the heart of the NECOS platform, the *Slice Database* which is tightly coupled with the *Resource Orchestrator*. Such database keeps track of all information on the deployed and operating slices. This model representation bridges the network slice specification defined from the *Tenant* with the resource representation defined from the involved data-centre and WAN providers. As discussed above, the *Slice Database* stores a limited-view of the infrastructure that provides resources for the slice. The term *“limited”* reflects the limited information disclosure exercised by infrastructure providers on third parties. A high-level representation of the unified NECOS information model is shown in Figure 20.

In particular, a slice description contains a Network Slice Specification and a Slice Infrastructure Description. The former is generated based on a service graph, which consists of service functions (*fn*) and service links. Service functions are further decomposed to service elements. The service functions represent the specifications for functional entities to be instantiated and the service elements their actual instantiation.

The *Slice Infrastructure Description* contains elements that reflect the slice infrastructure graph, *i.e.*, DC and Network slice parts, which may be deployed on top of separate cloud or network domains. These parts are linked to the service-elements that they host, as depicted in Figure 20. For instance, a “dc-slice-part” contains fields reporting provider-specific information, *e.g.*, which slice provider hosts the specific part, a VIM or a DC Controller. In addition, each slice part has one or more references to the service elements it will host. Obviously such information will be available after the successful completion of the resource discovery process. Examples of YAML messages regarding the Slice Infrastructure Description are provided in Section 4.

In the following subsections, we elaborate on the two primary aspects of the NECOS information model, the Network Slice Specification and the Infrastructure Description. As an outcome of our extensive literature research (*i.e.*, Sections 2.2 and 2.3), the NECOS model is influenced from the MANO, NOVI and COMS information models.

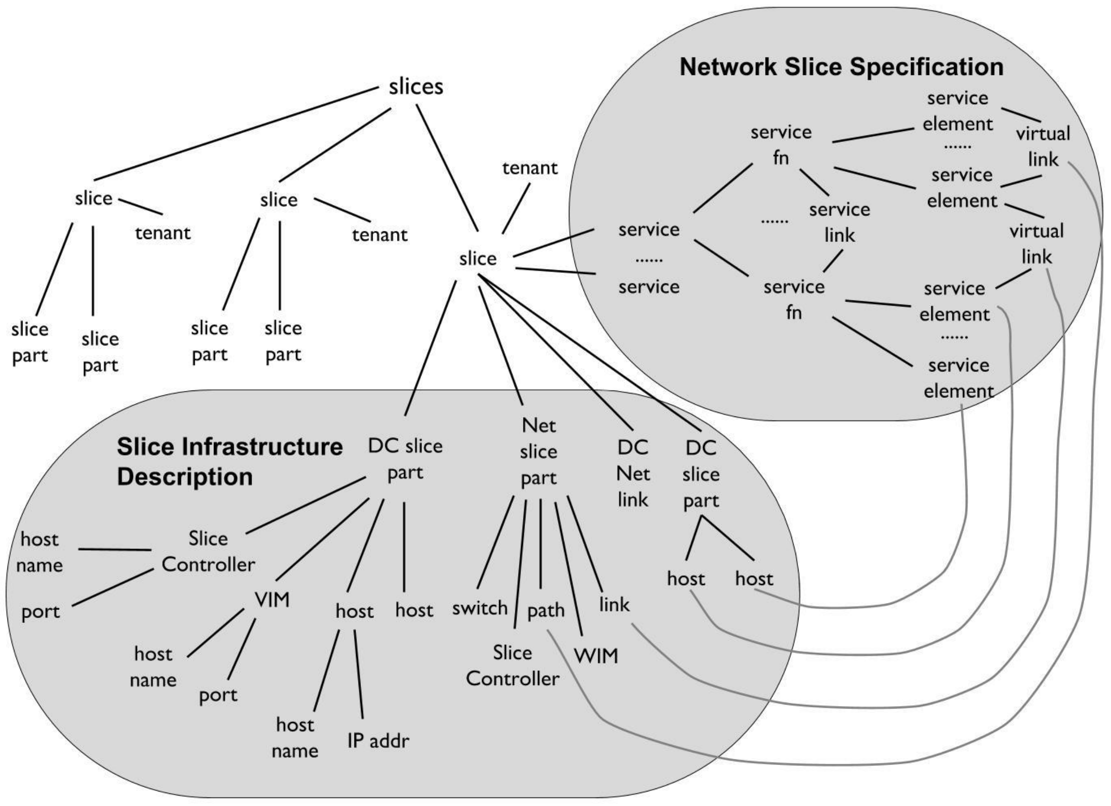

Figure 20. Overview of the NECOS information model.

3.2 Network slice specification

As NECOS deals with the provisioning, configuration, and run-time management of network slices, the NECOS information model needs to encompass all information required by the *Tenant* in order to specify slice and address slice components as well as service functions deployed within slices. Essentially, the information model will be used in different occasions by the *Tenant*, *e.g.*, when a *Tenant* submits a request for slice creation, when a *Tenant* wishes to submit a request on an existing slice for the modification of certain slice components or the scaling of the slice.

One of the challenges posed in terms of slice specification is that a slice may represent different views and/or may serve different purposes. For example, a slice may simply correspond to a subset of the physical infrastructure, essentially comprising a set of infrastructure elements, such as cloud servers. In this case, a *Tenant* will be granted with such slice and may later decide which services or applications he/she wishes to deploy. However, it should be also possible for a NECOS system to provision and manage service-tailored slices, *i.e.*, slice specifications that contain a set of service functions, such as VNFs. In the NECOS approach, this diversity in terms of slice request will be dealt with the translation of service demands into resource demands, facilitating resource assignment and allocation for slice creation. Nevertheless, in terms of slice specification, NECOS introduces a versatile information model that meets the requirements of all these aforementioned slicing levels, i.e., by completing different sets of attributes.

The slice viewpoint of the NECOS information model is illustrated in Figure 21. The model associates slices with services, which are in turn, associated with services functions and corresponding service elements (i.e., instantiated service functions). The information model further provides specifications of the VIM (which will be instantiated on demand in *Mode 0*). This allows the tenant to express preferences for the VIM (*e.g.*, request the instantiation of OpenStack). This is facilitated by the notion of Extended Platform Awareness (EPA), which will be explained in more detail below.

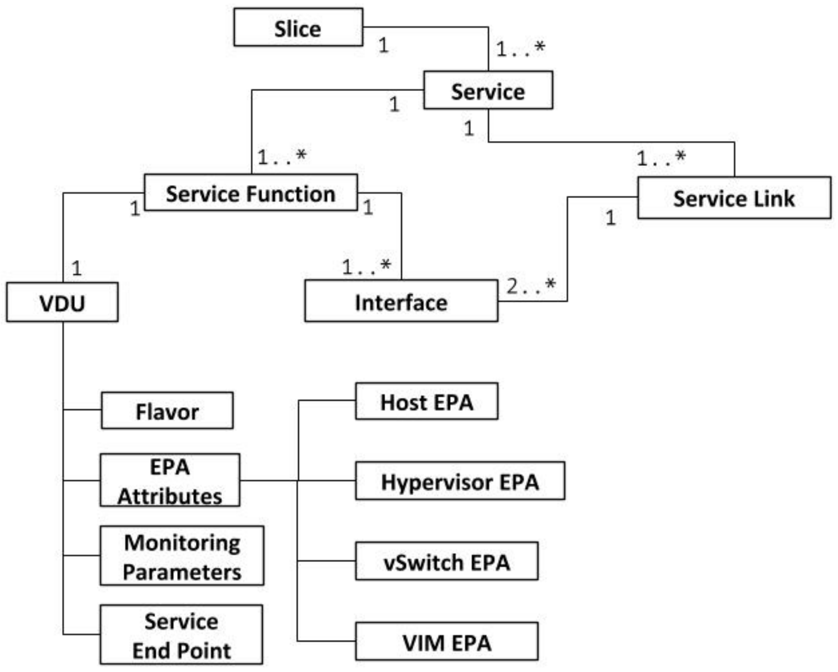

Figure 21. Slice specification with the NECOS information model.

In the following, we elaborate on the objects and attributes of the information model for the slice specification:

Slice: The *Slice* object provides a general description of the slice, which is further associated with services, as shown in Figure 21. This object includes the following attributes:

- Slice ID
- List of service deployed within the slice
- General service requirements, for example: cost model, slice lifetime (*e.g.*, start-time and end-time), geographical or other slice constraints (e.g., maximum number of slice parts), etc.

Service: The ***Service*** object provides a general description of a service, which is associated with ***Service Functions*** as well as ***Service Links***. This object includes the following attributes:

- Service ID
- Service description
- List of service functions
- List of service links

Service Function: The ***Service Function*** object is a descriptor for a service function, which is associated with ***Virtual Deployment Units (VDU)*** and interfaces. This object includes the following attributes:

- Service function ID

EUB-01-2017

- Location preference, which can be specified in the form of coordinates or with the name of a location (e.g., *city*)
- Distance tolerance from preferred location
- Number of interfaces
- Placement group (*e.g.*, isolation, co-location), which can be used to enforce the co-location or isolation among a group of service functions

Service link: This object describes a link and is associated with interfaces. The object has the following properties:

- Service link ID
- Bandwidth, which specifies the amount of required bandwidth
- Delay, which specifies an upper bound on delay for this link
- Jitter, which specifies an upped bound on jitter for this link

VDU: The ***VDU*** object binds a service function with resource requirements, a range of EPA attributes, and monitoring parameters, providing great flexibility in the specification and monitoring of service functions. This object includes the following attributes:

- VDU ID
- VDU name
- Number of service function instances
- Flavor
- Software image

Flavor: The ***Flavor*** object specifies resource demands for services functions and has the following attributes:

- Flavor ID
- Number of CPUs
- Amount of main memory
- Amount of storage space

EPA: Inline with ETSI NFV MANO, we employ the notion of EPA to assist tenants in expressing preferences for the instantiation of service functions and VIMs. In a similar manner, we have further extended the application of EPA to virtual switches and hypervisors.

Host EPA: This object specifies the following EPA attributes for the host at which a service function will be deployed:

- CPU Model (Required / Preferred), *e.g.*, Westmere, Sandybridge
- CPU Architecture (Required / Preferred), *e.g.*, x86, x86_64, i686, ARM, etc.
- CPU Instruction sets (Required / Preferred), such as AES
- Acceleration (Required / Preferred)
- Acceleration technique, *e.g.*, DPDK, Netmap, etc.

Hypervisor EPA: This object specifies the following EPA attributes for the hypervisor deployed in a cloud domain:

- Hypervisor, such as Xen or KVM

- Version, which indicates a preferred version of the requested hypervisor

vSwitch EPA: This object specifies the following EPA attributes for virtual switches that can be deployed in the slice:

- Acceleration (Required / Preferred)
- Acceleration technique
- Hardware Offloading (Required / Preferred)

VIM EPA: This object specifies the following EPA attributes for the VIM, which will be set up to manage and control the slice:

- VIM, such as Openstack, OpenVIM, etc.
- Version, which indicates a preferred version of the requested VIM
- Dedicated VIM (Required / Preferred)

Monitoring parameters: This object contains a wide range of parameters for monitoring the performance and reliability of service functions. Examples of such parameters include various counters (*e.g.*, number of packets sent/received, number of bytes sent/received) as well as the monitoring interval. Some of the parameters may be associated with Service Level Agreements (SLA), helping the client assess the achievable SLA level of his cloud service. In this deliverable version, we refrain from presenting an exhaustive list of monitoring parameters for the NECOS information model. Nevertheless, we will provide further details about the monitoring parameters in D4.2.

Service end-point: This object specifies an end-point for services, augmenting the binding of service with other applications or services. This object includes the following attributes:

- Service end-point name
- Type, *i.e.*, ingress or egress
- Interface
- Application
- Port number
- Protocol, *i.e.*, TCP or UDP

3.3 Infrastructure description

The physical infrastructure spans datacenters (that provide computing resources) and wide-area (or transport) networks that provide connectivity between the datacenters from which resources will be allocated for the slice instantiation. The NECOS information model aims at capturing the main resource and network elements available in datacenter and transport networks, such as servers, routers, switches, controllers, links, etc. Each object in the information model is associated with a set of properties that represent certain attributes for the infrastructure element. The range of properties for each object is certainly not exhaustive but can be easily extended with additional properties, if needed.

In the following, we present the main objects that are supported by our information model, including their main properties.

Infrastructure: The ***Infrastructure*** object provides a general description of the whole infrastructure, on top of which, network slices will be instantiated. This object of the information model encompasses the following attributes:

- Infrastructure ID
- List of domains comprising the infrastructure

Domain: The ***Domain*** object provides a general description of an infrastructure domain, which, in the case of NECOS, may be either a datacenter network (which corresponds to a traditional cloud), an edge cloud, or a WAN that provides connectivity among different datacenters. This object has the following attributes:

- Domain ID
- Provider, *i.e.*, the infrastructure provider for this domain
- Type, *i.e.*, datacenter for traditional clouds, (mobile) edge cloud, or WAN (see the respective telco cloud and MEC use cases in the deliverable D2.1)
- List of the hosts in the corresponding network domain
- List of the network elements in the network domain
- List of the links available in the network domain

Network Element: The ***Network Element*** object provides an abstraction of network elements, such as routers, switches, and Wi-Fi access points (AP). A network element includes the following attributes:

- Network element ID
- Availability
- Type, *i.e.*, the type of network element, such as router, switch, or AP.
- Number of ports
- Forwarding Information Base (FIB) size

Router: This object describes a router and includes the following attributes:

- Router ID
- Role, *i.e.*, edge or core router
- Packet types that can be processed, such as IP, MPLS, Ethernet
- Monitoring parameters for the router

Switch: This object describes a datacenter network switch and includes the following attributes:

- Switch ID
- Role, *e.g.*, Top-of-the-Rack, Aggregation, or Core switch
- Packet types that can be processed, such as IP, MPLS, Ethernet
- Monitoring parameters for the switch

Access Point: This object describes a Wi-Fi access point and includes the following attributes:

- Access Point ID
- Availability
- MAC, *i.e.*, MAC specifications supported, such as 802.11n
- Monitoring parameters for the AP

Host: This object describes hosts and encompasses the following attributes:

- Host ID
- Hostname
- Availability

- Location
- CPU, which contains a pointer to a separate object, namely CPU
- Memory, *i.e.*, the amount of available main memory
- Storage, *i.e.*, the amount of available storage capacity
- Number of ports
- Monitoring parameters for the host
- Other service-specific host capabilities, e.g., energy-measurement hardware, SAS disks optimized for storage nodes, etc.

CPU: This is a dedicated object for CPU-related specifications. The *CPU* object is associated with the *Host* object and contains the following attributes:

- Cycles, which specifies the number of available CPU cycles per core
- Number of cores
- Model, *e.g.*, Westmere, Sandybridge
- Architecture, *e.g.*, x86, x86_64, i686, ARM
- Instruction set, *e.g.*, AES

Controller: This object is associated with hosts, providing additional properties for hosts that serve as DC or network controllers. The ***Controller*** object contains the following attributes:

- Controller ID
- Role, *i.e.*, DC or network controller
- Configuration protocol, *i.e.*, the protocol used to configure the DC or the network (*e.g.*, SNMP, YANG, OpenFlow)
- Configuration IP address

Link: This object describes network links with the following attributes:

- Link ID
- Availability
- Type of the link, *e.g.*, point-to-point, point-to-multipoint
- Capacity
- Delay
- Jitter

Port: This object describes network ports and include following attributes:

- Port ID
- Availability
- Capacity
- List of the queues that may have been configured in the port, in the case of hardware multi-queuing (e.g., SR-IOV)
- IP address
- MAC address

Queue: This object describes queues in network ports that could be potentially configured, when there is support for hardware multi-queuing. The ***Queue*** object has the following attributes:

- Queue ID
- Availability
- Capacity

Path: This object describes network paths that encompass a set of links. The *Path* object has the following attributes:

- Path ID
- Availability
- List of links that comprise the network path
- Capacity, which expresses the overall capacity of the path and corresponds to the minimum capacity of all links that comprise the path
- Delay, which expresses the total delay incurred along the path
- Jitter, which expresses the overall jitter incurred along the path
- Disjoint links, which requires that all links comprising the path are disjoint

The UML diagram in Figure 22 illustrates the objects of the information model for the infrastructure description, as well as the relations between these objects. We note that some of the objects have been omitted in the UML diagram for clarity. One such object is the Wi-Fi access point, which is connected to the ***Network Element*** object in a similar way with the other respective elements (*i.e.*, router, switch). This UML model can be easily implemented through more specific description languages or schemas, such as XML, RDF, and YAML.

In the final version of this deliverable (*i.e.*, D4.2), we will provide a refined version of this information model, taking into account feedback from the ongoing NECOS system implementation.

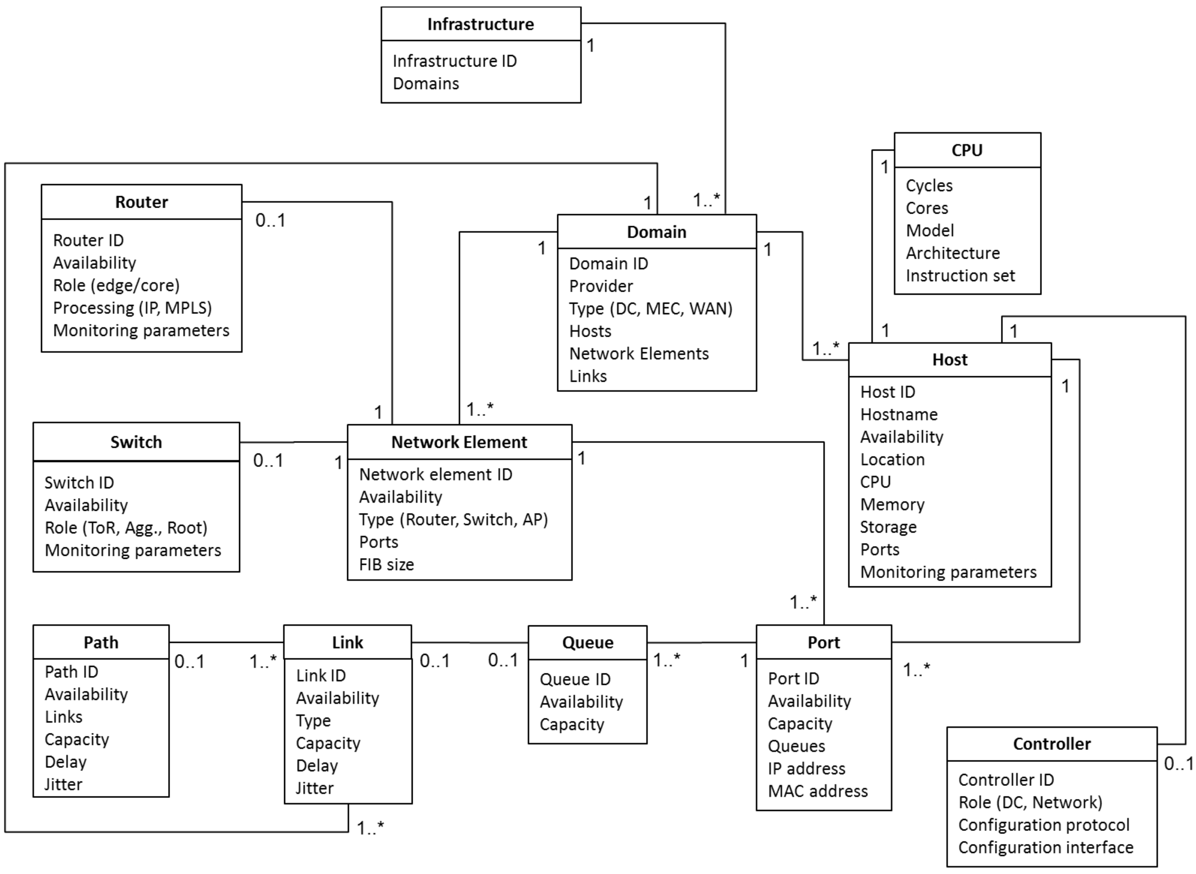

Figure 22. Infrastructure description with the NECOS information model.

3.4 YAML Description Example

Hereby, we present a YAML example based on the NECOS information model. In particular, we consider a simple slice that consists of a web server cluster and a load balancer, namely ***"Web Load Balancing Slice"***. This example is closely related to the Touristic Content Distribution scenario presented in D2.1, albeit simpler. The tenant (Metropolitan Touristic Center – MTC) wishes to deploy a CDN service that consists of three service functions, *i.e.*, a load_balancer, an orchestrator and a cluster of web servers **(*"web_server_VM"*)** in a slice that contains 2 DC slices parts and the link between them. The section of the YAML specification first provides general information regarding the slice (Figure 23), including geolocation slice constraints ***("location: EUROPE")***, the number of the requested DC and network slice parts, as well as slice requirements such as elasticity. For reasons of clarity, some sections of the YAML description have been omitted.

```
slices:
   sliced:
      id: TouristicCDN_sliced
      name: TouristicCDN_sliced
      …
      slice-constraints:
         location: EUROPE
         dc-slice-parts: 2
         net-slice-parts: 1

      slice-requirements:
         elasticity: true
      …
      slice-lifecycle:
         description: lifecycle status
         status: construction
      cost:
         dc-model:
            model: COST_PER_PHYSICAL_MACHINE_PER_DAY
            value-euros: {lower_than_equal: 10}
         net-model:
            model: COST_PER_LINK_PER_DAY
            value-euros: {lower_than_equal: 50}

      slice-timeframe:
         service-start-time: {100918: 10 pm CET}
         service-stop-time: {101018: 10 pm CET}

      # at least one slice component and one VDU should be defined
      service:
         - service-function:
              …
         - service-function:
              …
         - service-link:
            …
```

Figure 23. YAML top-level descriptions.

The top-level slice description contains a section regarding the cost for the DC and WAN slice parts, and the time-frame the slice is requested for (***"slice-timeframe:"***). The service section of the YAML description includes the service specification, as discussed below. The "web servers" service element specification is depicted in Figure 24. Such a service function description contains all the necessary information for the NECOS system to create the slice hosting the service. For example, the specification contains the number of instances of the "web server" service element, which is expressed as a range of integers (***1-75, instance-count: key***), and other placement preferences, such as the fact that this specific service element should be equally divided between two distinct slice parts (***slice-part-count: {equal: 2}*** and ***slice-part-ratio: {equal: 0.50}***), etc. The rest of the sections include EPA attributes.

```
service:
 - service-function:
        # defining web server cluster VDU
        service-element-type: vdu
        vdu:
           id: web_server_VM
           ...
           flavor:
              vcpu-count: 1
              memory-mb: 128
              storage-mb: 100
           vdu-image: 'web-server'
           instance-count: {in_range: [1, 75]}
           hosting: SHARED
           slice-part-count: {equal: 2}
           slice-part-ratio: {equal: 0.50}
           clustering: true
           epa-attributes:
              host-epa: ...
              hypervisor-epa: ...
              VIM-epa: ...

              vswitch-epa: ...
              ...
           interface:
           monitoring-parameters: ...
```

Figure 24. YAML specification of a service function.

For the specific service element, the associated EPA attributes are shown in Figure 25, where all resource demands and constraints for this element are described.

```
host-epa:
  cpu-model: PREFER_CORE2DUO
   cpu-arch: PREFER_X86_64
   cpu-vendor: PREFER_INTEL
   cpu-number: 2
   storage-gb: 2
   memory-mb: 4096
   host-count: {in_range: [10, 15]}
   max-host-count: undefined
   os-properties:
      # host Operating System image properties
      architecture: {equal: x86_64}
      type: linux
      distribution: ubuntu
      version: 16.04
   image-type: EMULAB
```

```
      host-image: ubuntu_linux_16.04_xen

   hypervisor-epa:
      type: XEN
      version: '4.5'

   VIM-epa:
      type: XEN-SERVER
      version: '7.5'
      vim-shared: true
      vim-federated: false
      vim-ref: undefined

   vswitch-epa:
      type: openvswitch
      accellaration: PREFERRED
      offload: PREFERRED
```

Figure 25. EPA attributes specification.

The specification of the service function interfaces defined by a corresponding YAML key as shown in Figure 26. It should be noted here that we differentiate between service external and service internal interfaces: the former are endpoints for users to access the MTC service offered by the slice, whereas the latter are interfaces for connecting to other service parts. Thus, the internal interfaces are associated with service-links, as discussed below.

```
   # defining web-server cluster's interfaces
interface:
   - service-external-interface:
        name: wsc-eth0
        virtual-interface:
           internal-name: eth0
           type: VIRTIO
           bandwidth: '0'
           vcpi: '0000:00:0a.0'
           ip: undefined
   - service-internal-interface:
        name: wsc-eth1
        virtual-interface:
           internal-name: eth1
           type: VIRTIO
           bandwidth: '0'
           vcpi: '0000:00:0b.0'
           ip: undefined
   - service-internal-interface:
        …
```

Figure 26. Service function interface description.

This eventually leads to the service-link YAML description (Figure 27). A link has ***"link-end-references"*** (***link-end-ref: orc-eth1 and link-end-ref: wsc-eth1***) where the values are internal interface ids of DC slice parts. The rest of the fields contain information necessary to allocate appropriate network resources for the link.

```
- service-link:
    service-element-type: link
    link:
      name: orc-to-wsc
      type: MULTIPLEXED
      ends:
        - link-end-ref: orc-eth1
        - link-end-ref: wsc-eth1
      requirements:
        bandwidth-GB: 1
      constraints:
        hops: {lower_than_equal: 2}
      reservation-protocol: undefined
```

Figure 27. Service link description.

This concludes this example for service specification. In the following section, the same example will be used to exemplify the ***Resource Discovery*** phase and illustrate better the information exchange that takes place.

4 Slice Discovery Framework

The resource discovery framework, as described in the NECOS architecture work package (*i.e.*, WP3), is responsible to locate the appropriate resources that compose a slice, *i.e.*, slice components that correspond to service functions and service links, according to the information model. A **Partially Defined Template (PDT)** message defines the general slice requirements and acts as the input to the resource discovery framework. This message is created by the *Slice Specification Processor* and passed to the *Slice Builder*. The *Slice Broker* is responsible to locate resources from both DC and WAN providers to fulfil the slice requirements and prepare a corresponding response, namely a **Slice Resource Alternatives (SRA)** message. In practice, the SRA message annotates the PDT message with alternative slice component options.

This process involves the following three architectural functional components:

- *Slice Builder* (Builder), which is responsible for forming an appropriate request to the Broker (*i.e.*, a PDT message), and selecting the most appropriate slice components, among alternatives returned by the Broker in the form of an SRA message.

- *Slice Broker*, which receives requests from the Builder and replies with alternative responses that fulfil the request, *i.e.*, creates and responds with an SRA message for each PDT message it receives.

- *Slice Agent*, which resides in the DC/WAN provider domain, replying to resource queries from the Slice Broker. The *Slice Agent* receives the slice component requirements, checks the local resource availability through communicating with its own *DC/WAN Controller* and responds with one or more resource options.

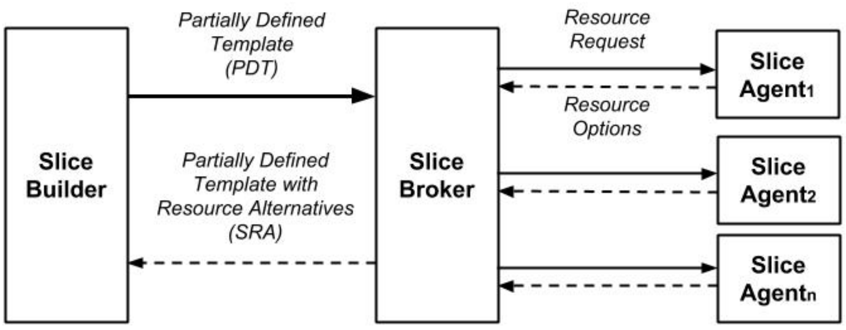

Figure 28. Overview of the resource discovery workflow.

Although the above description implies a query/answer model, in which data regarding availability of resources is dynamically collected for each request, a different model in which providers "push" information to the *Broker Agent* might as well be used, with minor modifications in the flow described below.

In the following, we highlight a **basic slice resource discovery workflow**. We assume that the builder has already prepared a PDT message, which includes the preferable number of slice components, their main resource requirements and the desirable connectivity among them. The basic steps of the workflow are as follows:

1. The *Builder* sends to the *Broker* a request in the form of a partially defined slice template.
2. The *Broker* proceeds with the incoming slice request processing through the following steps:
   2.1. Firstly, it queries the *DC Slice Agents* about the DC resources that are requested in the slice template.

EUB-01-2017

2.2. Subsequently, the *Broker* processes the DC resource responses received from the *Slice Agents*. For sets of resource components that match the template, the broker identifies potential network resource providers, and sends queries according to the connectivity demands indicated in the template.
2.3. Finally, it collects available information, in the form of alternative resources for each template's slice component and conveys the response to the *Builder*.

3. The *Builder* receives the above information in the form of an SRA message and decides on the final slice specification, completing the slice specification template and instantiating slice components with the allocation of the respective resources.

In the following, we elaborate further on the form of messages exchanged between the components stated above.

4.1 Partially Defined Template

The creation of the Partially Defined Template is the work of the *Slice Specification Processor*, which is responsible for creating an initial template based on user service specification.

In the simplest case, the template has a complete specification of each component (resource) required and only matching resources are returned in the responses. A more flexible setting involves a template component to be annotated with general slice or resource-specific requirements, so that resource providers can respond in a more flexible manner.

The PDT template must include both the desired slice topology, which encompasses the desired slice-parts, along with any resources' constraints on them and their connectivity (*i.e.*, the slice graph). For instance, below we present an abstract view of a simple exemplary template, represented in YAML (Figure 29):

```
slice:
   # definition of DC slice parts
   - dc-slice-part:
        name: dc-slice1
        vdus: …
   - dc-slice-part:
        name: dc-slice2
        vdus: …
        …
   - …
   # definition of WAN slice parts
   - net-slice-part:
        name: extrernal_ip_slice1-to-external_ip_slice2
        links:
           - dc-part1: dc-slice1
           - dc-part2: dc-slice2
        type: interaction
        …
```

Figure 29. Slice topology description in YAML.

Figure 29 indicates that the slice topology is clearly reflected in the YAML message. The two dc-slice parts (*i.e.*, the ***dc-slice1*** and ***dc-slice2***) should be connected via a network link, represented by the ***external_ip_slice1-to-external_ip_slice2*** inside the net-slice-part. We consider the former as nodes in the slice graph, while the later corresponds to edges.

Each part carries information regarding the desired characteristics that should be met when instantiating the component. Figure 30 depicts a more detailed specification of the ***dc-slice1*** part of Figure 29.

```
- dc-slice-part:
     name: dc-slice1
     dc-slice-controller:
        dc-slice-provider: undefined
        ip: undefined
        port: undefined
     VIM: undefined
     vdus:
        # defining load balancer VDU
        - dc-vdu:
             id: load_balancer
             name: load_balancer
             description: load balancer for elastic CDN deployment
             host-count-in-dc: 1
             max-host-count-in-dc: 1

        # defining web server cluster VDU
        - dc-vdu:
             id: web_server_VM
             name: web_server_VM
             description: web-servers for elastic CDN deployment
             host-count-in-dc: {equal: 5}
             max-host-count-in-dc: {equal: 8}
```

Figure 30. Slice part description in YAML

In Figure 30, each dc-slice part:

- has an ***undefined*** dc-slice-controller section that will be filled with the appropriate information, once the Builder selects one of the candidate providers' offers returned by the *Broker*.
- lists the VDUs of the service specification and the hosts that are being designated to. Each ***dc-vdu*** section identifies the corresponding VDU from the service description and is annotated with a set of requirements or constraints. Such constraints can be either in the form of a value (e.g., ***max-host-count-in-dc*: 1**) or expressions as relational constraints that the value must satisfy. Figure 30 demonstrates alternative ways of expressing constraints: can either be constants (i.e.,

numeric values) or relational expressions, as in the case of ***{equal:5}.*** Obviously, other relational constraints can be used, such as ***{greater_than:5}***, etc.

Undefined and constrained values justify the term “partially defined” of the PDT message, since they act as “variables” in the slice structure. As such, it contains sufficient information for the rest of the components to discover resources along with undefined/constrained fields to be completed later in the discovery process.

A similar design vein is followed for the representation of the ***net-slice-part*** (Figure 31):

```
- net-slice-part:
           name: extrernal_ip_slice1-to-external_ip_slice2
           wan-slice-controller:
              wan-slice-provider: undefined
              ip: undefined
              port: undefined undefined

           WIM: undefined

           links:
              - dc-part1: dc-slice1
              - dc-part2: dc-slice2
           type: interaction
           accommodates:
              - service-element: orc-to-wsc
              - service-element: wsc-to-orc-monitoring
           link-ends:
              link-end1-ip: undefined
              link-end2-ip: undefined
```

Figure 31. Net slice part description in YAML.

Similarly, the sections regarding the WAN-Slice-Controller and link-ends are left undefined, whereas the “accommodates” section defines the service elements that correspond to the links that this network connection should accommodate.

The information depicted in the *slice* block outlined in Figure 31 is not sufficient for the *Broker* to query *Slice Agents* for resources. Thus, the above is sent together with the service description, as mentioned previously. The service description contains a plethora of requirements and constraints that must be met by the resources to be matched. We decided not to duplicate that information here, for reasons of representational economy.

4.2 Queries Addressed to Resource Providers

The *Broker* decomposes the template it receives from the *Builder* and creates a different query for each slice component. Given the structure of the PDT message, such decomposition can be performed easily, since each slice component corresponds to a different resource provider. The *Broker* has all the necessary information to form query messages that contain all the constraints/preferences/resources needed for the component request message. For instance, such a message is depicted in Figure 32:

```
dc-slice-part:
   name: dc-slice1
   slice-constraints:
      geographic: EUROPE

   slice-requirements:
      elasticity: true
      reliability:
         description: reliability level
         enabled: true
         value: none  # {path-backup, logical-backup, physical-backup}

   slice-lifecycle:
      description: lifecycle status
      status: construction  # {modification, activation, deletion}

   cost:
      dc-model:
         model: COST_PER_PHYSICAL_MACHINE_PER_DAY
         value-euros: {lower_than_equal: 10}

   slice-timeframe:
      service-start-time: {100918: 10 pm CET}
      service-stop-time: {101018: 10 pm CET}

   dc-slice-controller:
      dc-slice-provider: undefined
      ip: undefined
      port: undefined

   vdus: …
   vim: …
```

Figure 32. Broker to DC Slice agent query message.

As shown in Figure 30, the message contains a number of sections that need to be filled by the *Slice Agent* when responding positively. For instance, information regarding the dc-slice-controller needs to be completed with its own information and the cost model section. Instantiated values act as constraints with respect to the slice part (*e.g.*, elasticity). Constraints regarding the specific hosts that will host the VDUs are described in section ***vdus*** (Figure 33).

```
     # defining load balancer VDU
      - dc-vdu:
           id: load_balancer
           instance-count: 1
           hosting: DEDICATED

           epa-attributes:
              host-epa:
                 cpu-model: PREFER_CORE2DUO
                 cpu-architecture: PREFER_X86_64
                 cpu-vendor: PREFER_INTEL
                 cpu-number: 2
                 storage-gb: {in_range: [2, 4]}
                 memory-mb: {greater_or_equal: 4096}
                 # host Operating System image properties
                 os-properties:
                    architecture: {equal: x86_64}
                    type: linux
                    distribution: ubuntu
                    version: 16.04
                 image-type: EMULAB
                 host-image: 'dns_load_balancer'

      # defining web server cluster VDU
      - dc-vdu:
           id: web_server_VM
           …

      # defining orchestrator VDU
      - dc-vdu:
           id: orchestrator

   vim:
      name: cdn-xen-vim
      type: XEN-SERVER
      on-demand: true
      host-count: 5
      max-host-count: 8
      image: 'ubuntu_linux_16.04_xen_server'
      hypervisor:
         type: XEN
         version: '4.5'
      vswitch:
         type: openvswitch
         acceleration: PREFERRED
         offload: PREFERRED
```

Figure 33. VDU specification in YAML.

As depicted in Figure 33, for each ***dc-vdu*** the ***host-epa*** attributes section contains information regarding the host characteristics required and any additional constraints. For instance, in the specific case the preferred CPU architecture is ***X86_64 (cpu-architecture: PREFER_X86_64)***, and the storage capacity has to be in the range of 2 to 4 GB (***storage-gb: {in_range: [2, 4]}***). The latter demonstrates how constraints on resources are communicated via relational constraint expression.

A similar message is sent to the *WAN providers* (Figure 34) that contains the necessary information for allocating the link between DC slice parts. Messages to WAN providers are sent after processing of the DC Providers' replies.

```
net-slice-part:
   # Same information as in the DC-above
   slice-constraints: …
   slice-requirements:
   slice-lifecycle: …
   cost: …
   slice-timeframe: …
   wan-slice-controller:
      wan-slice-provider: undefined
      ip: undefined
      port: undefined
   WIM: undefined
   links:
      - dc-part1:
           name: dc-slice1
           dc-slice-point-of-presence:
              # Needs more attributes
              pop-name: defined-previous-step-by-broker
```

```
                ip: defined-previous-step-by-broker
                router-type: defined-previous-step-by-broker
                reservation-protocol: undefined
                requirements:
                   bandwidth: 1 GB
        - dc-part2:
              name: dc-slice2
              …
   type: interaction
   constraints:
      bandwidth: 1 GB
   link-ends:
      link-end1-ip: undefined
      link-end2-ip: undefined
```

Figure 34. Link specification in YAML.

It should be noted that the same scheme can be alternatively implemented by sending complex queries to the *Slice Agents* in order to report availability on multiple components of the slice. The advantages of this alternative approach will be further investigated in the future.

4.3 SRA Message

The *Broker* collects all alternative responses to the messages above, and sends the response to the Builder. Responses for each alternative slice-part (both dc-slice parts and network-slice parts) are, in fact, lists of alternative resources originating from the *Providers*' *Slice Agents*. Since each *Slice Agent* supplies references to the offered slice parts, along with cost and other information, the *Builder* is in position to select the configuration of the slice that best matches the client's needs. Mechanisms for slice part selection and allocation will be described in the deliverable D5.1.

This concludes the initial presentation of the slice resource discovery workflow. Next steps include the *Slice Builder* instantiating slice parts to resources offered by providers and finally passing this information to the *Slice Orchestrator* to complete the slice creation according to the deliverable D3.1.

5 NECOS Cloud API Specifications

This section presents the cloud APIs specified by NECOS for slice request, creation, configuration, and run-time management. These cloud APIs have been classified into: (i) *client-to-cloud APIs*, which include API methods invoked by the *Tenant* for slice request, control and run-time management, and (ii) *cloud-to-cloud APIs*, which are associated with interactions between NECOS system components residing in different domains (*e.g.*, in the case of federation), such as the *Slice Resource Orchestrator*, the *Slice Builder*, *Slice Broker*, the *Slice Agents*, and the *Slice Controllers*, as shown in Figure 35. In Section 5.1, we elaborate on the *client-to-cloud API* specifications, whereas Section 5.2 describes the *cloud-to-cloud APIs*.

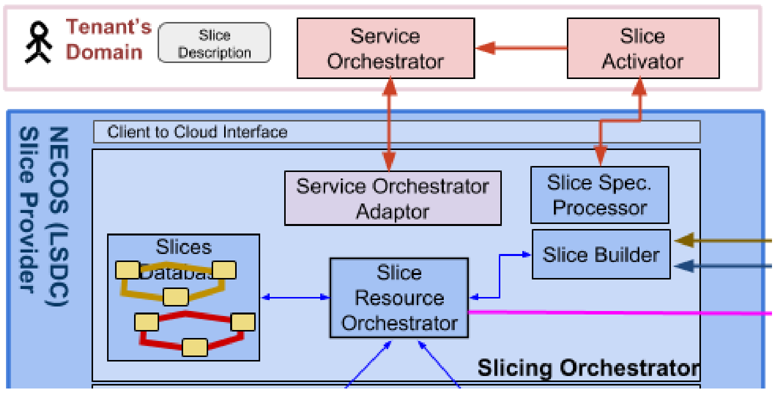

Figure 35. NECOS Client-to-Cloud Interface / API

5.1 Client-to-Cloud API

5.1.1 Slice (and Service) Management

NECOS provides different means to a *Tenant* to request the instantiation of a service and / or slice to a *NECOS (LSDC) Slice Provider*, as detailed in this section. The tenant has the option to initiate the creation of a slice with one of the following specification types, reflecting different levels of abstraction: (i) the Slice Specification, the lowest abstraction level focusing on resource aspects, (ii) the Slice Requirements, specifying the general slice requirements and leaving the *Slice Builde*r with the assistance of the *Slice Specification Processor* to determine the slice details, and (iii) the Service Specification, detailing the service to be deployed. As discussed in Section 3, the information model representation for the *Tenant* covers all these aspects and the above three types of slice requests complete different sets of attributes in the model. Subsequently, the *Slice Specification Processor* prepares the input of the *Slice Builder* and the latter translates the Slice Requirements or the Service Specification to an equivalent *Slice Specification*, in the form of the Partially Defined Template (PDT).

In particular, NECOS supports the following API methods for slice instantiation and management:

create_slice (Slice Specification, [Start Time], [End Time]): Slice ID

When this API call is invoked, the *NECOS Slicing Orchestrator* receives an explicit slice specification (also including slice requirements) as an input from a *Tenant*. This is the lowest level of abstraction that can be used by a Tenant to interact with the NECOS Slice Provider via the *Client-to-Cloud Interface*.

The API method **create_slice** shall be invoked by the *Tenant* (*e.g.*, via the *Slice Activator* in Figure 35) providing an explicit description of the slice topology to be instantiated (using the NECOS information model presented in Section 3, including *e.g.*, required resources such as number of cores, type of VIMs/WIMs, geographical constraints of the infrastructure elements, etc.) along with potential expected slice requirements (*e.g.*, delay constraints, rules of compliance, etc.). As an example, when using this

API call, the *Tenant* may explicitly define the number of (physical) hosts to be part of the slice, a particular type of VIM to be used to manage a slice segment, as well as slice access points to be used later on to anchor service instances to that slice (*e.g.*, hooks to the VIM or SSH connectivity for remote configuration tools like ansible).

The submitted input parameters are processed by the *Slice Specifications Processor*, which takes care of merging the requested slice topology description with the slice requirements, also verifying their full compliance. As soon as the above check is completed, the final slice specification is eventually produced and provided as an input to the *Slice Builder* component of the *NECOS Slicing Orchestrator*. The operation flow continues according to the steps required for building a (multi-provider) slice via the *Cloud-to-Cloud API*. The *Slice Builder* takes also into account the received slice requirements to generate a set of rules to be used by the *Slice Resource Orchestrator* to trigger slice lifecycle events – parameters of the slice are being adjusted at run-time according to monitoring information collected, while the slice is up and running using the *Cloud-to-Cloud API*. At the end of the Slice creation process, a **Slice ID** associated to the new created slice is returned to the *Slice Activator* in the *Tenant*'s Domain and is being used by the *Tenant* to deploy service instances on that particular slice via its own *Service Orchestrator*.

Alternatively, a *Tenant* may provide the description of a slice to be instantiated by the *NECOS Slice Provider*, at a given future instance in time, using the optional **Start Time** and **End Time** parameters in the signature. The operation flow is identical to the previous **create_slice** API call but the *Slice Builder* is not actually activating the slice until the specified time instant is reached (*i.e.*, specified by the **Start Time** parameter) and the slice remains until the time instance specified by the **End Time** parameter is reached. However, resource reservation might be requested on the potentially involved domains via the *Slice Marketplace Cloud-to-Cloud interface*. **Start Time, End Time** and other relevant parameters are part of the Slice Specification as well, but the signature definition overrides the equivalent attributes in the specification.

create_slice (Slice Requirements): Slice ID

In this case, the *Slice Resource Orchestrator* receives a set of Slice Requirements from a *Tenant*: a *Tenant* is able to request the allocation of a given set of resources, in the form of slice, for its services to be instantiated. However, the *Tenant* is aware of the services requirements and uses them as an input for the novel *NECOS Slice-as-a-Service* feature. As such, the *Tenant* is not providing a full specification of the slice elements, as in the previous API call. For instance, a requirement of the slice might be associated with delay bounds between two arbitrary elements of the slice to be created, *e.g.*, for delay sensitive services, the expected delay must be lower than a given critical threshold specified as part of the desired QoS. That delay information might be factored in during the creation of the slice in order to *e.g.*, map the slice connectivity related part on the proper network links interconnecting the computation and storage resources.

In this case, a *Tenant* invokes the **create_slice** method and provides as parameter a description of the expected slice requirements. The requirements are in the form of contract, an equivalent to Service Level Agreement for a Slice (*i.e.*, a Slice Level Agreement), consisting of one or more slice objectives (expressed as specific constraints on some KPIs of interest for the slice) to be fulfilled throughout the whole lifecycle of the slice. This method invocation results again in a request being sent to the *Slice Specification Processor* of the *NECOS Slicing Orchestrator*, which processes the requirements and generates a Slice Specification for the *Slice Builder*. The workflow continues as in the description of the previous API call. In the case of request rejection, (*e.g.*, due to insufficient resources), the API method will return a Slice ID set to 0, which will indicate to the tenant the outcome of his request (*i.e.*, rejection).

create_slice (Service Specification): Slice ID

In this case, a *Tenant* uses the NECOS *Client-to-Cloud Interface* to provide a high-level specification of a service (including also additional service related requirements / KPIs objectives) that are being translated into the Slice Specification required to run that particular service. The service may be defined as a forwarding graph that contains service elements (*e.g.*, VNF instances, VM / Container image types) available from different infrastructure providers / domains. This Service Specification is being considered together with the provided service requirements, at the slice instantiation time.

In this case, the *Slice Activator* in Figure 35 receives the *Tenant*'s Service Specification, in the same way with a Slice Specification and invokes the API call using that specification as an input parameter. This has the effect of propagating the Service Specification to the *Slice Specification Processor* within the *NECOS Slicing Orchestrator*. A Slice Specification related to that service is generated from the *Slice Specification Processor* and eventually processed by the *Slice Builder* to start the slice creation process. From that moment, the workflow is the same, as described for the previous API calls. As an additional step, once the slice creation is completed, the *Slice Activator* triggers the actual service instantiation on the new slice via the *Service Orchestrator* in the *Tenant*'s Domain.

delete_slice (Slice ID)

This API call is exposed to a *Tenant* to allow the deactivation of a slice that is no longer required. The invocation of this call triggers the decommission of the slice (identified by the **Slice ID**) via the interaction with the *Slice Resource Orchestrator*, which takes care of releasing the allocated resources in the corresponding Resource Domains (via the *Cloud-to-Cloud API*). In case there is a service running on the slice, the **delete_slice** method triggers the termination of the particular service.

get_slice_parts (Slice ID): Slice Part ID [ ]

As soon as a Slice is successfully instantiated, and its related **Slice ID** has been generated and returned to the requesting *Tenant*, further operations might be carried out on the slice in order to, *e.g.*, adjust its configuration and / or perform life-cycle operations on the slice. The *Tenant* may retrieve additional information about his allocated Slice using this API call by providing the ID of the Slice of interest, as an input parameter. This call returns a set of IDs associated to the different *Parts of the Slice* that can be used to perform fine-grained configuration / operational tasks on each element of the allocated slice. At this point, we decided to allow a lower-level view of the slice abstraction to the *Tenant* (i.e., have access to the slice parts), to allow better fine-tuning in the provided slice. This option may be omitted in the refined deliverable, since the advanced high-level abstractions we plan to build can make this obsolete.

start_service ([Service Specification, Service Name], Slice ID): Service ID

This API call is invoked by a *Tenant* to request the instantiation of a service on the slice (identified by the **Slice ID**) that was previously created by the *NECOS Slice Provider*. The service can be identified by its name, in the case a Service Specification was used for the slice creation, or by a new Service Specification. In the latter case, this process includes a validation step for the consistency of the Service Specification with the Slice Specification of the particular slice. When this call is invoked by the *Service Orchestrator* in the *Tenant*'s Domain, the Service Specification is passed to the *Service Orchestrator Adaptor* (in the *NECOS Slicing Orchestrator*), which, in turn, performs any required adaptation before requesting the embedding of the desired service on the slice. The call returns the ID of the instantiated service.

stop_service (Service ID)

This API call is used by a *Tenant* who previously submitted a service instantiation request (for a service identified by the **Service ID**) to stop the corresponding service elements, deployed on a slice.

get_service_info (Service ID): Service Status Information

This API call is invoked by a *Tenant* who previously submitted a service instantiation request to retrieve information about the runtime status of that particular service (identified by **Service ID**). This information can include, *e.g.*, the status of the service elements (such as VNFs, containers, links) in the case where the service monitoring process was delegated to NECOS (according to the level of abstraction in the interaction *Tenant / Provider*). More advanced monitoring options will be provided from the evolution of the IMA components.

5.1.2 Slice Configuration

After a successful slice provisioning requested by a *Tenant* using the API calls described in Section 5.1.1, the client should have the ability to configure and operate his slice. For example, the client may request the scaling of an existing slice, *e.g.*, the addition of new hosts to increase the compute / storage resources, links, or network elements (*e.g.*, virtual switches, routers, etc.). In this respect, NECOS further provides *Client-to-Cloud API* methods that are associated to the slice configuration and its lifecycle management.

In terms of slice configuration, the *Client-to-Cloud API* provides different methods to be used according to the level of abstraction (i.e., on the slice request or slice mode) that a *Tenant* wishes to use when interacting with a *NECOS Slice Provider*, as detailed in the two following subsections.

5.1.2.1 Lowest abstraction level

NECOS provides additional *Client-to-Cloud API* methods to *Tenants* that wish to exercise various operations (*e.g.*, control, configuration, life-cycle management) to a slice that was previously allocated by a *NECOS Slice Provider*. In the following, we present a set of such API methods that allow the *Tenant* to interact with the different slice parts and to the different elements related to them using the lowest available level of abstraction. In this Section, we refer to element as a generic instantiated entity, representing, *e.g.*, a *Host*, a *Path*, a *Switch*, a *Router* (this list is not be exhaustive).

get_slice_part_infrastructure_management_handle (Slice Part ID): Slice Part Management URL

This API method allows a *Tenant* to provide the ID of a slice part and to retrieve (when available) the URL that should be used to interact with the management interface of the VIM / WIM running on that slice part. This can be, for instance, the URL of the GUI that should be accessed to customise an Openstack instance that was deployed (on demand) in the slice part identified by the **Slice Part ID**.

get_slice_part_elements (Slice Part ID): Slice Part Element ID []

A *Tenant* that requires performing low-level configuration and management operations on the elements composing a slice part, uses this API call to retrieve the related references to them. When invoking this call using a given **Slice Part ID**, a list of **Element IDs** for that slice part are returned.

get_element_handle (Element ID): Element Management URL

This API call provides the abstractions to retrieve a proper Management entry point associated to the element of a slice part identified by an **Element ID**. For example, if the element is a physical host, a URL may be returned to access that particular host, e.g., for VNC or SSH.

add_element (Slice Part ID, Element Specification)

A *Tenant* uses this API call to dynamically modify the topology of an already allocated slice. This call provides a *Tenant* with a relatively abstracted way to add a new element to the slice part identified by the **Slice Part ID**. The NECOS components take care of reconfiguring the topology of the end-to-end slice to fulfil the *Tenant*'s request. The Element Specification contains information on the element, such as an image for the physical host, router configuration, etc. The *NECOS Slicing Orchestrator* forwards the **add_element** requests to the relevant *Slice Controllers*. The latter instantiate elements matching the element specifications and include them in the corresponding slice parts. In case the particular slice part does not have enough resources, this may trigger either a negative response or a slice elasticity process.

delete_element (Element ID)

As described for the previous API call (**add_element**), a *Tenant* is able to delete one of the elements from a slice part by providing the corresponding **Element ID**. The NECOS system abstracts the required management and configuration operations to the *Tenant* and the topology of the end-to-end slice is automatically modified to fulfil the *Tenant*'s request.

5.1.2.2 Intermediate abstraction level

In a slightly different scenario of interaction between the *Tenant* and a *NECOS Provider*, the former might need to modify / adjust the topology of an already existing slice without being aware of the related implementation details. In this case, the *Tenant* should consider using the API calls described in this subsection. In this abstraction level, the *Tenant* manages the slice as a whole, in a seamless way with respect to the consisting slice parts.

get_slice (Slice ID): Slice Specification

The *Tenant* uses this API method to retrieve the description (*i.e.*, the representation of the slice using the NECOS information model in Section 3) of a slice identified by its **Slice ID**.

update_slice (Slice ID, [Slice Specification], [Service Specification])

A *Tenant* that requested the allocation of a slice and received the associated **Slice ID** after its successful instantiation, is able to modify the slice topology by providing an updated Specification for it (either via the Slice or Service Specification parameter). The API call is processed by the NECOS system that transparently adjusts the arrangement of the slice parts (and elements) according to the delta between the existing and the old slice specifications. This abstraction level does not allow direct manipulation of service elements (i.e., addition or removal), because this may trigger inconsistencies in the Service Specification, so the **update_slice** method with a refined Service Specification should be used instead.

add_resources (Slice ID, Resource Descriptor)

This API method allows a *Tenant* to dynamically modify the topology of his own slice (identified by the **Slice ID**) by adding new (virtual) resources via the specification of a Resource Descriptor to be attached to the already existing slice. This is slightly different from the **update_slice** method, as in this case the explicit specification of the resource elements to be attached to the slice are being provided, instead of a new global slice description. The Resource Descriptor is the subset of the Slice Specifications that corresponds to one or a set of resources.

delete_resources (Slice ID, Resource Name)

A *Tenant* uses this API to explicitly delete resources from the slice identified by the **Slice ID** and described by the **Resource Name**, which is an attribute of the Resource Descriptor specified through the previous call. The NECOS system processes the request and abstracts the operations related to the modification of the existing slice. The **Resource Name** should be unique within a particular slice.

5.2 Cloud-to-Cloud APIs

As soon as a *Tenant* completes the submission of a slice instantiation request to one of the entry points of the NECOS platform (using any of the methods of the *Client-to-Cloud API* described in Section 5.1.1), the involved *NECOS Slice Provider* processes that request and starts the instantiation of the corresponding end-to-end slice.

Considering the interaction workflow described for the API call **create_slice (Service Specification): Slice ID** in Section 5.1.1, when a *Tenant* wishes to deploy the slice via the specification of a (type of) service that should be running on the slice, he should first describe that particular service. This process may involve an interaction with a *Service Marketplace* offering different service parts, however this step is out of the scope of this document. Once the service to be deployed is defined by the *Tenant* using an arbitrary combination of the logical service elements, and after the invocation of the **create_slice** call, a slice descriptor is generated by the *Slice Specifications Processor* and sent to the *Slice Builder*.

A similar (simplified) workflow is also executed when the *Tenant* submits the request for a slice instantiation via the **create_slice (Slice Specification, [Start Time], [End Time])** or **create_slice (Slice Requirements)** API calls, described in Section 5.1.1. A Slice descriptor is eventually generated also in this case, and provided again as input to the *Slice Builder* (by the *Slice Specifications Processor*) to actually start off the slice instantiation process via the *Cloud-to-Cloud API*.

The *NECOS Cloud-to-Cloud* API consists of 4 different interfaces as depicted in Figure 36, *i.e.*, the *Slice Request Interface*, the *Slice Instantiation Interface*, the *Slice Marketplace Interface* and the *Slice Runtime Interface*. Details related to each of the above interfaces will be provided in the remainder of this Section.

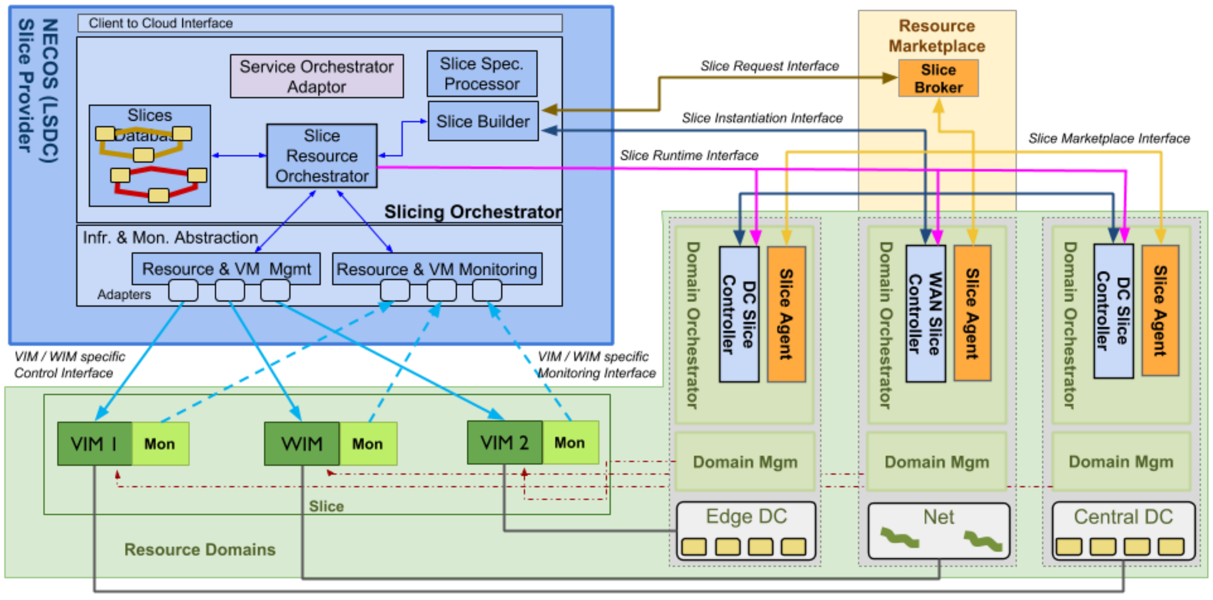

Figure 36. NECOS Cloud-to-Cloud API.

5.2.1 Slice Request Interface

This API initiates the slice creation process. We assume that the *Slice Builder*, after receiving the invocation of the **initiate_slice_creation** call on its internal interface (this API has not fully been detailed in the current release of the document and will be further described in the final version of this deliverable), interacts with the *Slice Broker* (in the *Resource Marketplace*) to request an updated view of the resources available from the different providers that have "registered" their availability to that *Slice Broker*. The term "registered" is used here to identify whatever form of interaction between the *Slice Broker* and the underlying *Slice Agents* (the communication mechanisms are not restricted to any particular technology).

The *Slice Request Interface* should provide the methods described in the following in order to support the functionalities required by the *Slice Builder* to discover and select resources made available from (external) resource *Providers* that can be used to build the requested end-to-end slice.

locate_slice_resources (Partially Defined Template): Service Resource Alternatives

The *Slice Builder* starts the slice instantiation process by invoking the **locate_slice_resources** API call (on the *Slice Broker*). The method requires the *Tenant* input in the form of a **Partially Defined Template** (PDT), which covers both the Slice Descriptor and the Slice Requirements. The *Broker* first identifies which *Providers* are able to supply resources for the required slice parts, according to the overall view that the *Slice Broker* collected via the *Slice Marketplace Interface*. The *Broker* responds with a **Service Resource Alternatives** (SRA) message which includes a list of the Slice Controllers corresponding to the providers offering resources.

5.2.2 Slice Instantiation Interface

The next phase of the slice creation process involves interactions via the *Slice Instantiation Interface* between the *Slice Builder* and the set of *Slice Controllers* retrieved earlier via the *Slice Request Interface* (*i.e.*, after the invocation of the **locate_slice_resources** call). Based on the list of entry points of the *Slice Controllers* and on the slice topology partitioning returned by the *Slice Broker*, the *Slice Builder* submits individual requests to the relevant *Slice Controllers* to allocate resources for each single part of the end-to-end slice. The descriptors and requirements for each slice part are being produced based on the PDT message.

The *Slice Instantiation Interface* provides the following methods in order to support the functionalities required by the *Slice Controllers* to reserve, activate and release the elements of a slice part.

request_slice_part (Slice Part Descriptor, Slice Part Requirements): Slice Part ID

This API method is exposed by the *Slice Controllers*. The **Slice Part Descriptor** (which is generated based on the Slice Specification or the Service Specification) details the characteristics of either a DC or network part of the overall slice, whereas the Slice Part Requirements (which are generated from the Slice Requirements) are used to provide information about the performance requirements for that particular slice part, as derived from the initial description of the end-to-end Slice. Resources are reserved on the *Slice Controller* that receives the invocation of this API call, and a **Slice Part ID** is returned back as a response, in case of a successful interaction.

activate_slice_part (Slice Part ID): {Slice Part ID, Infrastructure Manager Handle}

This API method is also part of the interface exposed by the *Slice Controllers*. This particular method is used to actually activate a slice part (identified by the **Slice Part ID**) on that *Slice Controller* and deploy the on-demand VIM/WIM (Slicing Mode 0) or a shim object on behalf of the tenant (Slicing Mode 1). As a result of the call, a handle to the relevant Infrastructure Manager (*e.g.*, a VIM / WIM) or shim object that was allocated in the Slice Part is returned.

delete_slice_part (Slice Part ID)

This method is exposed by the *Slice Controller* to allow the deletion of a slice part. As a result of the call, the allocated resources/elements of the slice part are released.

5.2.3 Slice Marketplace Interface

This API is related to the interaction between the *Slice Broker* and the *Slice Agents* to implement mechanisms for the propagation of resource offerings between (external) resource domains. As already discussed earlier, different *Slice Agents* register their resources' availability to the *Slice Broker*. The interaction involving the *Slice Broker* and the underlying *Slice Agents* (i.e., communication mechanisms and protocols) are pluggable and not constrained to any specific implementation / technology. The API is aligned to the *NECOS Slice Discovery Framework* detailed in Section 4.

register_provider (Agent Entry Point): Provider ID, Location

This API method is exposed by the *Slice Broker* in order to select candidate providers. The Agent Entry Point describes the *Providers*' *Slice Agents*, which announce their presence to the *Slice Broker* and then interact to provide offers regarding their resources. Every *Provider*, either *DC* or *WAN*, returns to this API call its ***Provider ID*** along with its geographic ***Location***. The former information is used by the *Broker* when responding to the *Builder* by loading the appropriate fields in the Service Resource Alternatives (SRA) message, *e.g.*, ***dc-slice-controller***, ***dc-slice-point-of-presence*** (*i.e.*, response to the corresponding **PDT** message). The latter is evaluated to meet the *Tenant's* geographic slice constraints.

push_resource_offer (Resource Descriptor): Resource Element ID [], Cost

This API method is also exposed by the *Slice Broker* in order to receive a pool of offers regarding either a DC or a network part of a whole slice. The **Resource Descriptor** field specifies the kind of resource to be offered, while the **Resource Element ID** list sent by the *Slice Agent* is explicitly defining the attributes of a specific piece of resource element, *e.g.*, a Host. For example, while the **Resource Descriptor** indicates the offer of a ***dc-vdu***, the **Resource Element ID** list sent by the *Slice Agent* contains the full list of attributes corresponding to a specific ***host-epa***. The resource offer is accompanied by its corresponding **Cost** and takes part in the formation of a Service Resource Alternatives (SRA) message.

pull_resource_offer (Slice Description): Resource Element ID [], Cost

A similar API to the **push_resource_offer** is the **pull_resource_offer** API which, however, is exposed by the *Slice Agents* and returns **Resource Element ID** lists and their **Cost** as a response to a **Slice Description** request submitted by the *Slice Broker*. More specifically, once the *Broker* receives the **PDT** message, it decomposes it in different queries towards the *Slice Agents*. Each query specifies the **Slice Description** of a slice component, which defines the resource specifications of the latter. The **Slice Description** is included and further defined in the **Resource Element ID** list returned by the *Slice Agent*. In practice, while the ***dc-vdu*** fields define a set of preferences regarding the specifications of a ***host-epa***, *e.g.*, **storage_gb: {in_range: [2, 4]}**, the same fields are explicitly defined inside the *Slice Agent* response, which is the response of this API call, e.g., ***storage_gb: 4***. The responses are used in the Service Resource Alternatives (SRA) message, which is the response of the *Slice Broker* to the *Slice Builder*.

5.2.4 Slice Runtime Interface

This API implements the interface that provides functionalities to dynamically modify the resource allocation for a slice part. This is required by the *Slice Resource Orchestration* to perform lifecycle operation on the end-to-end slice according to the feedback received by each slice part via the monitoring measurements.

get_slice_part_elements (Slice Part ID): Slice Part Element ID []

This API method is exposed by the *Slice Controllers* to allow the *Slice Resource Orchestrator* to get the references to the elements of a given slice part. When invoking this call using a given **Slice Part ID**, a list of **Element IDs** for that slice part is returned to the *Slice Resource Orchestration* that invoked the call.

get_element_handle (Element ID): Element Management URL

This API method is exposed by the *Slice Controllers* to allow a *Slice Resource Orchestration* to retrieve a proper Management entry point associated to the element of a slice part, identified by the **Element ID**.

add_element (Slice Part ID, Element Specification)

This API method is exposed by the *Slice Controllers* to allow a *Slice Resource Orchestrator* to dynamically add a new element into an already allocated slice part. Upon receiving this call, the *Slice Controller* looks for an element matching the element specification and adds that element to the slice part.

delete_element (Element ID)

This API method is exposed by the *Slice Controllers* to allow a *Slice Resource Orchestrator* to dynamically remove an element from an already allocated slice part.

Note that the API methods **get_slice_part_elements**, **add_element** and **delete_element** can be invoked by the *Slice Resource Orchestrator* to provide low-level details of the slice-part elements (and fulfil the tenant's requests described in Section 5.1.2.1), as well as to provide high-level information of the slice part (and fulfil the tenant's requests described in Section 5.1.2.2). In the latter case, the *Slice Resource Orchestrator* may derive the required high-level information from the low-level one.

update_slice (Slice Part ID, Slice Part Descriptor)

A *Slice Resource Orchestrator* that requested the allocation of a slice part and received the associated **Slice Part ID** after its successful instantiation, is able to modify the elements of the slice part by providing an updated Specification for it (via the Slice Part Descriptor parameter). The API call is being processed by the *Slice Controller* that adjusts the arrangement of the slice elements according to the delta between the existing and the new Slice Part Descriptor.

start_VNFs (VNF Descriptor, Infrastructure Manager Handle): VNF_ID []

This API method is exposed by the Resource and VM Management components to allow a *Slice Resource Orchestrator* to start VNFs (described by the **VNF Descriptor** parameter) under the management of a given VIM (identified by the **Infrastructure Manager Handle** parameter**)** in a given slice part. When invoking this call, the relevant VIM instantiates the VNFs and the **VNF_IDs** are returned to the *Slice Resource Orchestrator*.

delete_VNFs (VNF_ID [], Infrastructure Manager Handle)

This API method is exposed by the Resource and VM Management components to allow a *Slice Resource Orchestrator* to delete VNFs (described by the **VNF_ID []** parameter) running under the management of a given VIM (identified by the **Infrastructure Manager** Handle parameter**)** in a given slice part.

get_VNFs_info (VNF_ID [], Infrastructure Manager Handle): VNF status information

This API method is exposed by the *Resource* and *VM Monitoring* components to allow a *Slice Resource Orchestrator* to retrieve status information on VNFs (described by the **VNF_ID []** parameter) running under the management of a given VIM (identified by the **Infrastructure Manager Handle** parameter**)** in a given slice part.

We note that we will seek the alignment of VNF-related API methods with ETSI NFV MANO, and particularly with its VNFM component. Further details on this will be documented in D4.2.

6 Conclusions

This deliverable provides the first version of the NECOS information model, from the *Tenant*, *Slice Database* and the *Infrastructure Provider* viewpoints, and a set of cloud API methods, which are either invoked by the *Tenant* during slice request, configuration, run-time management (*i.e.*, client-to-cloud APIs) or by the NECOS system for slice provisioning, control, and resource orchestration (*i.e.*, cloud-to-cloud APIs). D4.1 further elaborates on methods for resource discovery, describing the workflow and detailed examples of exchanged information in YAML. All these specifications have been made after careful inspection of SOTA, which has been analyzed in this deliverable.

Essentially, this deliverable extends the NECOS system architecture (documented in D3.1) with the necessary means to request and provision slices, enabling a new cloud computing model, namely *Slice as a Service*. The information model and the cloud APIs will be further refined within WP4, and certain extensions are foreseen, especially as the model and the APIs are integrated and tested in the NECOS proof-of-concept implementation. The final version of the information model and cloud APIs will appear in the deliverable D4.2.

7 References

[3GPP] 3GPP, http://www.3gpp.org/

[5GEX] 5GEx Deliverable 2.2, “5GEx Final System Requirements and Architecture”, December 2017, https://drive.google.com/open?id=1O8KjGolPEAWlit0wRJ_pLykvvanJ2Rk5

[BBF] BBF SD-406: End-to-End Network Slicing, https://www.broadband-forum.org/5g

[CONTRERAS18] L. M. Contreras, “Slicing challenges for operators”, Book chapter in Emerging Automation Techniques for the Future Internet, M. Boucadair and C. Jacquenet (Eds.), IGI Global, 2018.

[GALIS2017] “Network slicing terms and systems” slides. Available online: https://datatracker.ietf.org/meeting/99/materials/slides-99-netslicing-alex-galis-netslicing-terms-and-systems-02, access in May 30th, 2018.

[GOTO18] Y. Goto, Standardization of Automation Technology for Network Slice Management by ETSI Zero Touch Network and Service Management Industry Specification Group (ZSM ISG), NTT Technical Review, Vol. 16, No. 9, Sept. 2018.

[ietfdata2018] A. Clemm, et al., "A Data Model for Network Topologies", Internet Draft, expires on June 2018. Available online: https://tools.ietf.org/html/draft-ietf-i2rs-yang-network-topo-20

[ietfcoms2018a] L. Qiang, et al. "Technology Independent Information Model for Network Slicing", Internet Draft, expires on July 30, 2018. Available online: https://tools.ietf.org/html/draft-qiang-coms-netslicing-information-model-02.

[ietfcoms2018b] C. Qiang, "COMS Architectural Design Enablers & Artefacts-I, COMS Technology Independent Information Model" Slides. Available online: https://datatracker.ietf.org/meeting/101/materials/slides-101-coms-coms-architectural-design-enablers-artefacts-1-coms-technology-independent-information-model-cristina-qiang-00

[ITU-T Y.3011] Recommendation ITU-T Y.3011 (01/2012), SERIES Y: GLOBAL INFORMATION INFRASTRUCTURE, INTERNET PROTOCOL ASPECTS AND NEXT-GENERATION NETWORKS - Next Generation Networks – Future networks - Framework of network virtualization for future networks.

[ITU-T Y.3111] Recommendation ITU-T Y.3111 (09/2017), SERIES Y: GLOBAL INFORMATION INFRASTRUCTURE, INTERNET PROTOCOL ASPECTS, NEXT-GENERATION NETWORKS, INTERNET OF THINGS AND SMART CITIES - Future networks -IMT-2020 network management and orchestration framework.

[MEC] ETSI MEC, Multi-access Edge Computing (MEC), MEC support for network slicing, GR MEC 024 v 2.0.5, July 2018.

[MEDHIOUB2011] H. Medhioub, I. Houidi, W. Louati, and D. Zeghlache, Design, implementation and evaluation of virtual resource description and clustering framework, IEEE International Conference on Advanced Information Networking and Applications (AINA), 2011.

[NFV] ETSI Network Operator Perspectives on NFV priorities for 5G, https://portal.etsi.org/NFV/NFV_White_Paper_5G.pdf.

[NGMN] NGMN White Paper description of network for service provider networks, https://www.ngmn.org/fileadmin/user_upload/161010_NGMN_Network_Slicing_framework_v1.0.8.pdf

[NML] Network Markup Language Working Group, Available online: http://www.ogf.org/gf/group_info/view.php?group=nml-wg

[novi2015] J. van der Ham, J. Stéger, S. Laki, Y. Kryftis, V. Maglaris, and C. de Laat, “The NOVI information models, Future Generation Computer Systems”, 42(C), 2015, pp. 64–73, http://doi.org/10.1016/j.future.2013.12.017

EUB-01-2017

[RFC7950] M. Bjorklund, The YANG 1.1 Data Modeling Language. (M. Bjorklund, Ed.). RFC Editor, 2016, https://doi.org/10.17487/RFC7950

Version History

Version	Date	Author	Change record
0.1	11/07/2018	P. Papadimitriou	Creation
0.2	03/09/2018	P. Papadimitriou	First integrated draft
1.0	28/09/2018	P. Papadimitriou	Final version release

EUB-01-2017