H2020 5G-Crosshaul project
Grant No. 671598

D3.2: Final XFE/XCI design and specification of southbound and northbound interfaces

Abstract

This report presents the consolidated design of two core components of 5G-Crosshaul, namely the 5G-Crosshaul Forwarding Element (XFE) in the data plane and the 5G-Crosshaul Control Infrastructure (XCI) in the control plane.

Document Properties	
Document Number:	D3.2
Document Title:	Final XFE/XCI design and specification of southbound and northbound interfaces
Document Responsible:	NOK-N
Document Editor:	NOK-N
Editorial Team:	Carla Fabiana Chiasserini (POLITO), Thomas Deiß (NOK-N), Giada Landi (NXW), José Núñez-Martinez (CTTC), Charles Turyagyenda (IDCC)
Target Dissemination Level:	Public
Status of the Document:	Final
Version:	1.1
Reviewers:	Miguel Berg (EAB), Daniel Cederholm (EAB), Carla Fabiana Chiasserini (POLITO), Shahzoob Bilal Chundrigar (ITRI), Thomas Deiß (NOK-N), Jose Enrique Gonzalez Blazquez (ATOS), Beatriz López Herraiz (ATOS), Nuria Molner (UC3M), Antonio de la Oliva (UC3M)

Document History:

Revision	Date	Issued By	Description
0.1	26.9.17	NOK-N	Version for 1st review
0.2	10.10.17	NOK-N	Version for 2nd review
0.3	25.10.17	NOK-N	Revision after 2nd review
1.0	26.10.17	NOK-N	Final version
1.1	3.11.17	NOK-N	Final version, document status corrected, numbering in Table20 corrected

Disclaimer:

Table of Content

List of Contributors

Partner No.	Partner Short Name	Contributor's name
P01	UC3M	Antonio de la Oliva, Nuria Molner, Sergio González Díaz, Jaime García Reinoso
P02	NEC	Andrés García Saavedra, Xi Li, Xavier Salvat
P03	EAB	Chenguang Lu, Miguel Berg, Daniel Cederholm
P04	TEI	Paola Iovanna
P05	ATOS	Beatriz López Herraiz, Jose Enrique Gonzalez Blazquez
P06	NOK-N	Thomas Deiß, Dieter Knüppel, Ole Reinartz
P07	IDCC	Charles Turyagyenda, Alain Mourad
P09	TI	Andrea Di Giglio, Antonia Paolicelli, Roberto Morro
P13	NXW	Giada Landi, Francesca Moscatelli, Elian Kraja, Marco Capitani
P14	CND	Alberto Diez, Jakub Kocur
P17	CTTC	Ramon Casellas, Arturo Mayoral, Ricard Vilalta, Raul Muñoz, Ricardo Martinez, Josep Mangues-Bafalluy, José Núñez-Martinez, Jorge Baranda
P18	CREATE-NET	Leonardo Goratti, Domenico Siracusa
P19	POLITO	Claudio Casetti, Carla-Fabiana Chiasserini, Senay Tadesse
P21	ITRI	Shahzoob Bilal Chundrigar

List of Tables

List of Figures

List of Acronyms

Acronym	Description
A-ROF	Analogue Radio over Fibre
ACTN	Abstraction and Control of Transport Networks
AF	Adaptation Function
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
BH	Backhaul
BIER	Bit Indexed Explicit Replication
CA	Certificate Authority
CCM	Connectivity Check Message
CDN	Content Delivery Network
CIDR	Classless Inter-Domain Routing
COP	Control Orchestration Protocol
CPRI	Common Public Radio Interface
CPU	Central Processing Unit
C-RAN	Cloud Radio Access Network
CRC	Cyclic Redundancy Check
CRUD	Create, Read, Update, Delete
CU	Central Unit
DB	Data Base
DEI	Discard Eligible Indicator
DU	Distributed Unit
E2E	End to end
ECMP	Equal Cost Multipath

EMMA	Energy-Management and Monitoring Application
eNb	Evolved NodeB
ETSI	European Telecommunications Standards Institute
FAPI	Femto Application Platform Interface
FEC	Forward Error Correction
FH	Fronthaul
GBR	Guaranteed Bit Rate
GMPLS	Generalized Multi-Protocol Label Switching
gPTP	Generalized Precision Time Protocol
GRE	Generic Routing Encapsulation
HTTP	HyperText Transfer Protocol
IETF	Internet EngineeringTask Force
IP	Internet Protocol
IT	Information Technology
JSON	JavaScript Object Notation
LAN	Local Area Network
LBM	Loopback Message
LDP	Label Distribution Protocol
LLDP	Link Layer Discovery Protocol
LMS	Local Management Service
LTE	Long Term Evolution
LSP	Label Switched Path
LTM	Link Trace Message
MAC	Media Access Control
MANO	Management and Orchestration

MIP	Mixed Integer Programming
MMA	Mobility Management Application
MME	Mobility Management Entity
MPLS	Multiprotocol Label Switching
MPLS-TP	MPLS Transport Profile
MTA	Multi-Tenancy Application
MVNO	Mobile Virtual Network Operator
NAS	Non-access Stratum
NBI	NorthBound Interface
NFV	Network Functions Virtualization
NFVI	Network Function Virtualization Infrastructure
NFV-O	Network Functions Virtualization Orchestrator
nGBR	non-GBR
NGFI	Next Generation Fronthaul Interface
NIC	Network interface card
NS	Network Service
NSD	Network Service Descriptor
OAM	Operation and Maintenance
ODL	OpenDayLight
ONF	Open Networking Foundation
ONOS	Open Network Operating System
OS	Operating System
OTN	Optical Transport Network
OTT	Over-The-Top
PBB	Provide Backbone Bridging

PBSS	Personal Basic Service Set
PCP	Priority Code Point
PCP	PBSS Control Point
PDCP	Packet Data Convergence Protocol
PDV	Packet Delay Variation
PHB	Per Hop Behaviour
PKI	Public Key Infrastructure
PNF	Physical Network Function
PoC	Proof of Concept
PTP	Precision Time Protocol
PW	Pseudowire
QoS	Quality of Service
RAM	Random Access Memory
RAN	Radio Access Network
REST	Representational State Transfer
RFC	Request For Comments
RLC	Radio Link Control
RoE	Radio Over Ethernet
RU	Remote Unit
SBI	SouthBound Interface
SDN	Software-Defined Networking
SFC	Service Function Chaining
SID	Service Identifier
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol

SP	Service Provider
SRLG	Shared Risk Link Group
TAS	Time Aware System
TE	Traffic Engineering
TLS	Transport Layer Security
TTL	Time to live
TVBA	Television Broadcasting Application
UCA	Use Customer Address
UE	User Equipment
UNI	User-to-Network Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VDU	Virtual Deployment Unit
vEPC	virtual Evolved Packet Core
VIMaP	Virtual Infrastructure manager and Planner Application
VIM	Virtual Infrastructure Manager
VLAN	Virtual LAN
VM	Virtual Machine
VN	Virtual Network
VNE	Virtual Network Embedding
VNF	Virtual Network Function
VNFC	Virtual Network Function Component
VNFD	Virtual Network Function Descriptor
VNFFG	Virtual Network Function Forwarding Graph
VNFM	Virtual Network Function Manager

VPN	Virtual Private Network
VTN	Virtual Tenant Network
VXLAN	Virtual Extensible LAN
WP	Work Package
XAF	5G-Crosshaul Adaptation Function
XCF	5G-Crosshaul Common Frame
XCI	5G-Crosshaul Control Infrastructure
XCSE	5G-Crosshaul Circuit Switching Element
XFE	5G-Crosshaul Forwarding Element
XML	Extensible Markup Language
XPFE	5G-Crosshaul Packet Forwarding Element
XPU	5G-Crosshaul Processing Unit

Executive Summary

This document details the activities carried out and results obtained within the scope of Work Package (WP) 3 of 5G-Crosshaul. This document presents the consolidated design of two fundamental components of 5G-Crosshaul control and data plane, namely the 5G-Crosshaul Control Infrastructure (XCI) and 5G-Crosshaul Forwarding Element (XFE), and the interfaces between data plane and XCI (Southbound interface, SBI) as well as between XCI and applications plane (Northbound interface, NBI). Refinements of the initial design based on experience gained when developing the prototypes and proof-of-concept systems are reflected in the consolidated design.

The key technical achievements described in this document are summarized as follows:

- Description of the information model, workflow and design of Application Programming Interfaces (APIs) exposed by XCI services towards the application-plane through a NBI. The services include Topology and Inventory, Provisioning and Flow actions, Information Technology (IT) infrastructure and Inventory, Statistics, Network Function Virtualization Orchestration (VNF-O), Virtual Network Function Management (VNFM), Analytics for Monitoring, Local Management Service (LMS) and Multi-tenancy.
- The design of the XCI, building on the initial design described in D3.1 [1]. We present the services exposed within the XCI towards the application-plane, and a set of software components that are part of proof-of-concept prototypes developed by 5G-Crosshaul partners. We describe the integration between Software Defined Network (SDN) controllers and Management and Network Orchestrator (MANO) components in the XCI to establish paths in the 5G-Crosshaul network and to connect the Virtual Network Functions (VNFs) in the data centers to deliver the Network Services (NSs). Also, the interaction between child and parent controllers in a hierarchical setup is described in detail.
- As input to energy saving for virtualization technologies, we present models of the power consumption of hypervisor- and container-based virtualization.
- The design of the 5G-Crosshaul data plane, including an updated design of XFEs, 5G-Crosshaul Common Frame (XCF), and Adaptation Functions (AFs). We describe an OpenFlow pipeline to provide XCF encapsulation and forwarding. The bootstrapping interaction between XFEs and SDN controllers is described in detail.
- The implementation of enhanced Nodes b (eNbs) with different fronthaul splits. This allows to test the 5G-Crosshaul data plane with traffic streams having real latency and jitter requirements.
- Besides the actual transport of data, the network has to provide synchronization to remote radio heads and baseband units. We describe packet-based synchronization and related synchronization technologies to be used for 5G-Crosshaul and how its accuracy depends on the existence of other traffic and its priority. We also describe Operation and Maintainance (OAM) functionality to manage the network.
- We specify the Key Performance Indicators (KPIs) that are addressed by the data plane and control plane design, the metrics retrieved for their evaluation, and we summarize the results.

Compared to the previous document D3.1 [1], the major updates of this document are the description of the interaction of SDN controllers with the MANO components as well as with the XFEs. The network optimization model has been updated. The power consumption model has been added, as well as the description of synchronization, OAM and bootstrapping, and the XPFE pipeline. The implementation of different fronthaul splits has been described. The description of KPIs relevant for WP3 and their evaluation has been added to the document.

1 Introduction

The main objective of this document is to describe the design of two essential components of the 5G-Crosshaul project, namely the XCI and the XFE including the design of a unified frame format to forward both fronthaul and backhaul traffic. D3.1 [1] introduced and presented the first version of the design of these two components. The 5G-Crosshaul forwarding network is comprised of XFEs, which include both packet switching elements (XPFEs) and circuit switching elements (XCSEs), and carry data using a common frame format defined as XCF. In the control plane, the XCI provides control and management functionality to manage all the available resources that build the 5G-Crosshaul infrastructure, including XFEs and 5G-Crosshaul Processing Units (XPUs) which carry out the bulk of the computational burden required by the different services provided in 5G-Crosshaul.

The design of these 5G-Crosshaul components closely follows the 5G-Crosshaul system architecture designed in WP1 within the System Architecture task (see D1.1 [2]). Importantly, we design a control-plane management and orchestration architecture that can be easily implemented with different and possibly heterogeneous software platforms, both open-source and proprietary. Our design follows the architectural guidelines fixed by the European Telecommunications Standards Institute (ETSI) [3] and Open Networking Foundation (ONF) [4] standard bodies. Namely we defined an SDN/NFV-based platform (the XCI) that provides decoupled data and control plane, logically centralized control, and exposure of abstracted resources and state to external applications. Indeed, Section 3 shows that our design is a suitable enabler for modularized and independent implementations because it shows how proof of concept of different components of the XCI can be demonstrated using different software platforms. The only constraint that such software platforms should comply with the interfaces defined by the interaction with the applications (through an NBI; see Section 2 and the data plane (through a SBI; see Section 6. In Section 2, we present the information models and APIs used to describe the NBI with some detailed examples. The remaining definitions are described in the appendix. The design of the XCI is complemented with a model for power consumption of virtualization techniques and for optimization of forwarding paths in Section 4. This supports the evaluation of the KPIs related to energy saving. Secondly, Section 5 reflects the status of the design of the XFE. As mentioned in D2.1 [5] and D3.1 [1], the XFE design consists of a multi-layer switch comprised of a packet- and a circuit-switching layer (XFPE and XCSE, respectively). In more detail, we describe the XCF, packet based synchronization, Quality of Service (QoS) management and OAM for the network. In Section 6 we motivate the choice of OpenFlow as the protocol for the SBI of the XCI. We describe an OpenFlow pipeline for XCF encapsulation and forwarding and the bootstrapping interaction between the XFEs and the SDN Controllers. We consider forwarding packets through the eNb also as part of the data plane. The implementation of different fronthaul splits for eNbs are described in Section 7. Eventually, in Section 8 we define the KPIs that are addressed by WP3, the metrics to evaluate the different components designed in this WP, and we summarize the evaluation.

2 Northbound interface design

The goal of this section is to provide the final specification of the XCI controller interfaces at the Northbound, namely NBI. In particular, the reference protocol mostly used for the specification of the XCI Northbound services is based on Representational State Transfer (REST). The REST protocol is one of the most used paradigms for specifying Northbound services for both SDN and IT controllers (e.g., Open DayLight (ODL) [6], Open Network Operating System (ONOS) [7], and OpenStack [8]). Note that this does not preclude the use of other protocols such as proprietary ones or the use of WebSockets [9]. In particular, the use of WebSockets can be of primal importance for certain modules to provide, for instance, asynchronous notifications of network or compute events from XCI services towards 5G-Crosshaul applications.

The specification of NBI is not standardized yet, though organizations such as the ONF NBI Working Group are working towards this goal. We leave open the choice whether the REST APIs will be fully compliant with RESTCONF protocol [10]. In this way, there is no need to define a YANG data model associated to the REST APIs.

The the set of XCI NBI services is based on the Northbound functionalities identified in D3.1 [1] and the requirements of the 5G-Crosshaul applications, which are identified in D4.1 [11]. Note that the details reported in this section have been used as input for the subsequent development of the 5G-Crosshaul NBI. The XCI NBI services have been implemented internally in the SDN controller and the IT controller. The latter, which is mapped to the compute and storage controller in the 5G-Crosshaul architecture, exposes NBI services to the NFV MANO components, namely: the NFV-O, the VNF manager, and the 5G-Crosshaul Virtual Infrastructure Manager (VIM) referred to as the Virtual Infrastructure Manager and Planner Application (VIMaP).

The following subsections provide the NBI services exposed by the XCI towards the 5G-Crosshaul applications or towards other XCI modules. In particular, for each NBI service, we provide the APIs, the more relevant information data models associated with this NBI service and a workflow illustrating the use of this service by a generic 5G-Crosshaul application or by an internal module inside the XCI that, in turn, can expose an NBI. As for the APIs, we include a table indicating the protocol, the Uniform Resource Identifiers (URIs), the operation, and the input/output parameters associated with the operation. As for the data model, we provide a Unified Modelling Language (UML) diagram specifying the most relevant classes involved in the NBI service. Finally, we include a workflow example, in the shape of a sequence diagram, which illustrates some examples of each NBI service. Note that the low-level implementation details of the APIs, workflows and information data models are beyond the scope of this deliverable.

For the sake of brevity, we include a full description of Topology and Inventory (in Section 2.1) and the NFV-O (in Section 2.6) NBI services, as representative examples of NBI services abstracting network and IT resources. For the rest of NBI services, we provide a limited description that does not include the details regarding APIs, workflows, and information data models. The full information is provided in Section 11 (Appendix I).

2.1 Topology and Inventory

The Topology service maintains information regarding all created networks. It abstracts the learned physical topology information that may be collected by some automated process also involving the network elements (e.g., using protocols like Link Layer Discovery Protocol (LLDP)). This abstraction offers the possibility of creating subnets and/or to enhance the topology with other kind of information (e.g., Traffic Engineering (TE) topology including TE metrics, Shared Risk Link Groups (SRLG), etc.) providing the REST APIs to create, remove networks (i.e., a subnet which is different concept from that of tenant), add/remove node links to/from a network.

On the other hand, the Inventory component provides query network inventory services. In particular, it provides REST APIs to query inventory data from the inventory database (DB). In this way, the network node and port capabilities can be obtained from this service.

2.1.1 APIs

The Northbound APIs in the following table give access to the network topology stored and maintained by the SDN controller (i.e., the one formed by XFEs). These APIs also provide primitives to create, delete, and modify the networks in the physical network infrastructure. The APIs detailed below also expose network inventory resources, such as the list of nodes and their capabilities. In the following, we provide a description of the APIs offered by the Topology and inventory services.

Table 1: Topology and Inventory API.

Prot.	**Type**	**URI**	**Parameters**	
REST	GET	*../topology/default* Retrieve the whole physical network infrastructure. *../ topology/{network_id}* Retrieve the specified network topology with identifier network_id.	Input	*network_id* (optional)
			Output	*network_object*
REST	POST	*../topology/{network_id}* Register a new network.	Input	*network_id*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	PUT	*../topology/{network_id}* Add subnetwork with id *network_id* to the physical network, the specified by	Input	*network_id* *network_object*
			Output	Success: Status Code

		network_object.		of normal end
				Failure: Error code
REST	PUT	*../topology/{network_id}/{link_id}* Add new link *link_id* to *network_id*.	Input	*network_id* *link_id*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	DELETE	*../topology/{network_id}* Delete an existing network with identifier *network_id*.	Input	*network_id*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	GET	*../topology/default/nodes/* Retrieve all the nodes. *../topology/{network_id}/nodes* Retrieve the specific nodes in subnet with *network_id*.	Input	*network_id* (optional)
			Output	*node_list*
REST	GET	*../topology/nodes/node_id* Retrieve node information details of *node_id.*	Input	*node_id*
			Output	*node_object*
REST	GET	*../topology/{src_node_id}_{dst_node_id}* Get the shortest path in terms of number of hops between two infrastructure devices.	Input	*src_node_id* *dst_node_id*
			Output	*path_object*
WebSockets	SUBSCRIBE	../topology/*{network_id}_{event}*	Input	*network_id* *event*
			Output	Success: Status Code of normal end
				Failure: Error code

WebSockets	ASYNC	notification event	Input	*event*
			Output	N/A

2.1.2 Information Model

We consider the notion of a physical topology that is comprised of a list of networks. These networks are subnets from the physical network. Each of the multiple topologies has the notion of multiple nodes. Each node has multiple ports. Each network has multiple links, where each link connects two ports. Based on the technology, the object “Port” can have different technologic-specific attributes.

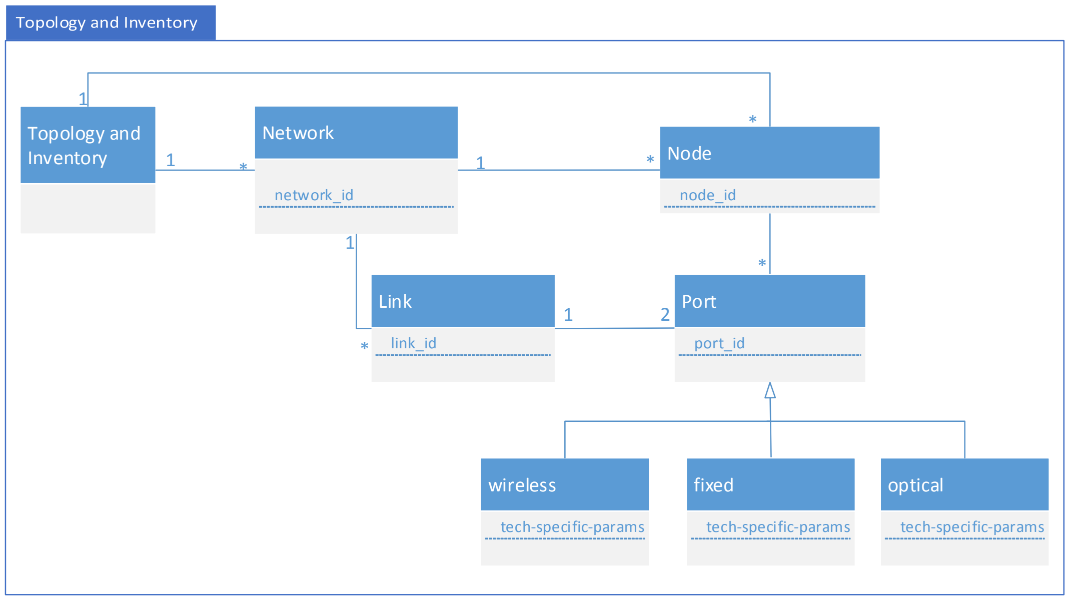


Figure 1: Topology and Inventory Information model.

A more detailed description of the attributes of some relevant information models associated with the Topology and Inventory services is described next.

Table 2: Topology and Inventory Information model.

Parameters	Type	Description
network_id	String	Identifier of the network.
network_object	Set<Links>	Object describing the network as a set of links in Javascript Object Notation (JSON) or Extensible Markup Language (XML) format. Each link has the following attributes: • EndpointA; String; Node • EndpointB; String; Node • Bandwidth; Integer; Bandwidth of the link. • State: Enum, whether the link is active or not

		• Technology; Enum; Indicate type of the link (e.g., mmWave, optical)
network_event	Enum	The specific set of asynchronous network events considered. {node_up, node_down, link_up, link_down, port_up, port_down}.
node_id	String	Identifier of the node.
node_object	Node	Contains the identifier of the node and type of node. Object describing the node as a set of properties in JSON or XML format. Parameters (variable; type; description): • Identifier: String; identifier of the node • Type: String; type of node • Name: String; name of the node • Number of Ports; Integer, number of ports
path_object	Object	Object describing the path as set of links between two endpoints (in XML or JSON format).

Request/response media types can be in JSON and/or XML format. It is important to note that the URIs specified in the request by the consumer shall be independent of the chosen representation in the implementation. For instance, in what follows we illustrate a part of the response body in JSON format of a network topology corresponding to a link connecting a node with a switch. In particular, the request will have the following form: GET *../topology/n1,* where *n1* is the identifier of the network.

A part of the output of the physical network topology corresponding to subnetwork n1 follows:

Table 3: network topology of subnetwork n1

```
"network":[
  {
          “id”:”net_example”,
          “links”: [
              {
                    “edgeId”:”1”,
                    “edgeType: “wireless_edge”,
                    "EndPointA":{
                              "id":"00:00:00:00:00:00:00:01",
                    },
                    "EndPointB":{
                              "id":"00:00:00:00:00:00:00:51",
                    },
                   "name": {
                              "value": "s1-eth1",
                    },
                    "state": {
                              "value": 1,
                    },
                    "bandwidth": {
                              "value": 100; // Data rate in Mbps
                    }
              }
              …//more links can be specified
          ]
  }
]
```

2.1.3 Workflow

In what follows, we provide a message exchange sequence to illustrate the use of the service by an application referred to as the consumer. This consumer can be an application located on top of the XCI. The goal is to illustrate the use of the Topology and Inventory services. The workflow in Figure 2 illustrates the creation of a physical subnetwork forming part of the physical network topology. As shown in Figure 2, this process can consist of the major steps:

1. The consumer application gets the physical network inventory from the Topology and Inventory service.
2. The consumer application decides to register a new network. A physical subnetwork is specified. Register a physical subnetwork from the consumer to the Topology and Inventory service.
3. A physical subnetwork is specified for the previously registered network identifier. A physical subnetwork from the consumer to the Topology and Inventory service.
   3.1. Selection of a private or public subnetwork based on the range of defined IPs.
   3.2. Selection of Virtual Local Area network (VLAN) associated to the registered physical subnetwork.

4. Subscription to an event of interest in this physical subnetwork (e.g., link failure). A Uniform Resource Locator (URL) is returned as a result of a successful subscription.
5. Creation of a WebSockets to receive asynchronous notifications from the Topology and Inventory service of the event to which the consumer application is subscribed.
6. Asynchronous notifications of the previously subscribed event by the Topology and Inventory service to the Consumer.
7. The consumer application decides to deallocate the physical subnetwork.

The entities involved in this process are the consumer application and the topology and inventory service

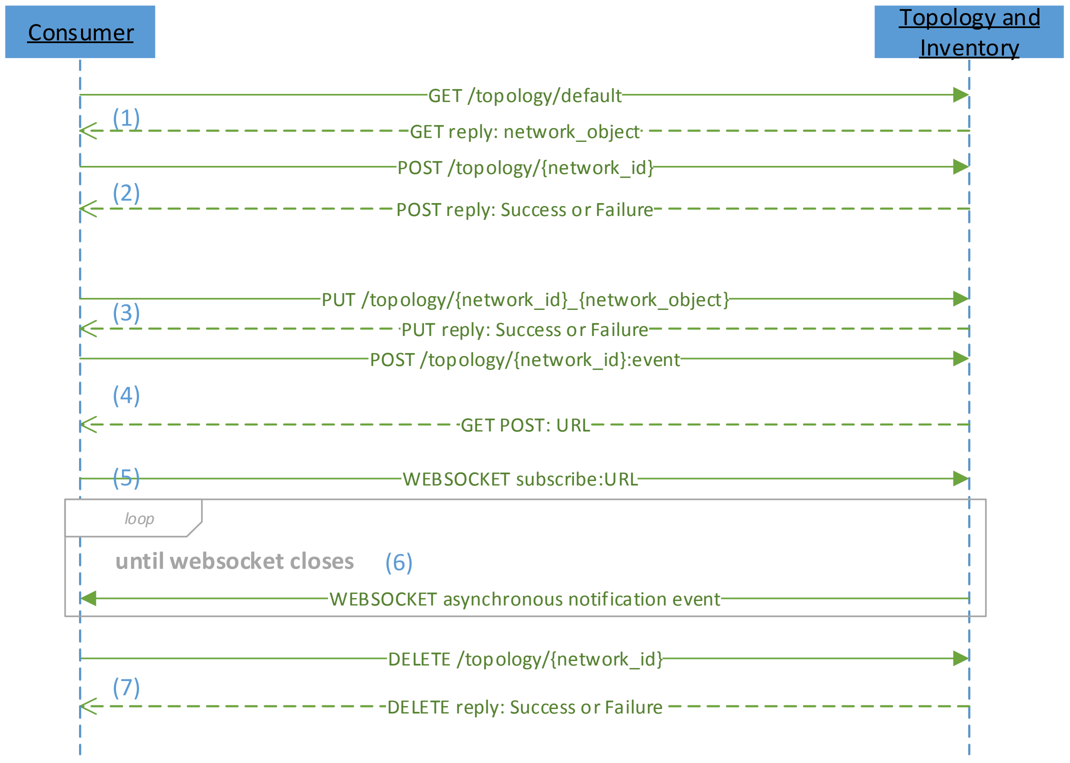

Consumer
Topology and Inventory
GET /topology/default
(1)
GET reply: network_object
POST /topology/{network_id}
(2)
POST reply: Success or Failure
PUT /topology/{network_id}_{network_object}
(3)
PUT reply: Success or Failure
POST /topology/{network_id}:event
(4)
GET POST: URL
(5)
WEBSOCKET subscribe:URL
loop
until websocket closes
(6)
WEBSOCKET asynchronous notification event
DELETE /topology/{network_id}
(7)
DELETE reply: Success or Failure

Figure 2: Topology and Inventory workflow example.

2.2 Provisioning and Flow Actions

This service provides the flow programming service for 5G-Crosshaul applications. In particular, this service is in charge of offering to the north (applications or certain modules inside the XCI) an API that allows them to conduct queries related to the forwarding rule installation and removal of network nodes. It is important to note that this module acts prior to the actual flow installation rule in the 5G-Crosshaul network nodes. A more detailed description including the APIs, information models and workflows involved can be found in Section 11.1.

2.3 IT infrastructure and inventory

The IT infrastructure and inventory NBI is based on the REST protocol. In particular, we use the NBI of several OpenStack [8] modules (e.g., nova, neutron), as the NBI offered by the IT infrastructure and inventory service. These modules will form part of the IT controller inside the XCI in the form of plugins.

To offer such an API, the IT infrastructure and inventory service must comprise several modules. For instance, the set of instantiated Virtual Machines (VMs) must be maintained by an IT infrastructure and inventory database. Also, an IT infrastructure and inventory scheduler is needed to determine where the VM must be instantiated. A more detailed description including the APIs, information models and workflows involved can be found in Section 11.2.

2.4 Statistics

D3.1 [1] describes two services tailored to the collection of monitoring information (in particular Section 7.1.4 for network-related statistics and Section 7.2.1.2 for IT-related statistics). We updated the design to a unified common service, namely Statistics service, which integrates the functionality of the two. Note that this API is based on OpenStack's Ceilometer [12].

This service shall offer to any consumer (or client) a network- computing- and storage-related statistics service on a per tenant basis, including metering, alarm, and collection of samples. A more detailed description including the APIs, information models, and workflows involved can be found in Section 11.3.

2.5 Virtual Infrastructure Manager and Planner

The NBI of the VIMaP service is based on the REST protocol. This module is basically in charge of conducting CRUD (Create, Read, Update, Delete) operations for the NS layout (a set of VMs) for the ETSI NFV architecture. The VIM also offers an API to conduct CRUD operations for the network slice or virtual infrastructure concept (i.e., a set of not only VMs, virtual switches, and virtual routers) associated to the creation of virtual infrastructure for the Mobile Virtual Network Operator (MVNO) use case. As for the former, it corresponds to the deployment of NSs as defined within the ETSI MANO architecture. In particular, it is in line with the ETSI use case #4 VNF Forwarding Graphs in [13]. As for the latter, it tackles the instantiation of virtual infrastructures with ultimate user control composed by a coherent set of network, compute, and storage infrastructure. In this case, the infrastructure is completely provided to the tenant (e.g., XFEs, cards, ports) including XPU resources. In fact, the VIMaP main goal is to offer to the consumer the services offered by the SDN and IT (compute and storage) controller in a unified manner. A more detailed description including the APIs, information models and workflows involved can be found in Section 11.4.

2.6 NFV-O

The NFV-O offers an NBI that allows the orchestration of NSs to 5G-Crosshaul applications. A NS is composed by multiple VNFs or Physical Network Functions

(PNFs), which are interconnected through the specification of a VNF Forwarding Graph (VNFFG).

2.6.1 APIs

The NFVO provides mechanisms to create, retrieve and remove NSs, as described in the following table.

Table 4: NFV-O API.

Prot.	Type	URI	Parameters	
REST	POST	*../nfvo/ns* Create a new NS for a given tenant.	Input	*NS Id* *ServiceDeploymentFlavour* *NS Tenant Id*[1]
			Output	*NS Id*
REST	GET	*../nfvo/ns/ns_id* Retrieve information about the given NS.	Input	*NS Id* *NS Tenant Id*
			Output	*NS record*
REST	DELETE	*../nfvo/ns/ns_id* Remove an existing NS.	Input	*NS Id* *NS Tenant Id*
			Output	Success: Status Code of normal end
				Failure: Error code

2.6.2 Information Model

The main entity managed by the NFVO is the NS, a chain of VNFs interconnected through a VNFFG. The characteristics of a NS are defined according to a standard template, called NS Descriptor (NSD), which defines:

- NS generic information, like vendor, version, human readable description, etc.
- The VNFs, which compose the NS, identified through their VNF Descriptors (see Section 2.7).
- The PNFs, which compose the NS (optional).
- The VNFFGs, which interconnect the VNFs (and the PNFs if available), identified through their VNFFG Descriptor. (Both VNFDs and VNFFGDs are stored in repositories at the NFV-O level).

[1] A tenant is one or more NFV MANO service users sharing access to a set of physical, virtual or service resources. An NS tenant is a tenant to which NSs are assigned. (See [14])

- The dependencies between the VNFs.
- The scripts and configuration parameters to be launched at the various stages of the NS lifecycle.
- The monitoring parameters.
- The criteria and the constraints for automated and on-demand scaling of the NS.

The VNF Descriptor information model is reported in Figure 67 within Section 11.5.2.

VNFDs are stored in the VNFD Database (DB), a shared DB which is accessed by both VNFM and NFV-O. Suitable management APIs are usually exposed by the NFV-O to load new VNFDs in the repository. This procedure is defined by ETSI NFV MANO standard and it is out of the scope of this document. During the instantiation of a VNF, the VNFD is specified in the request through its unique identifier.

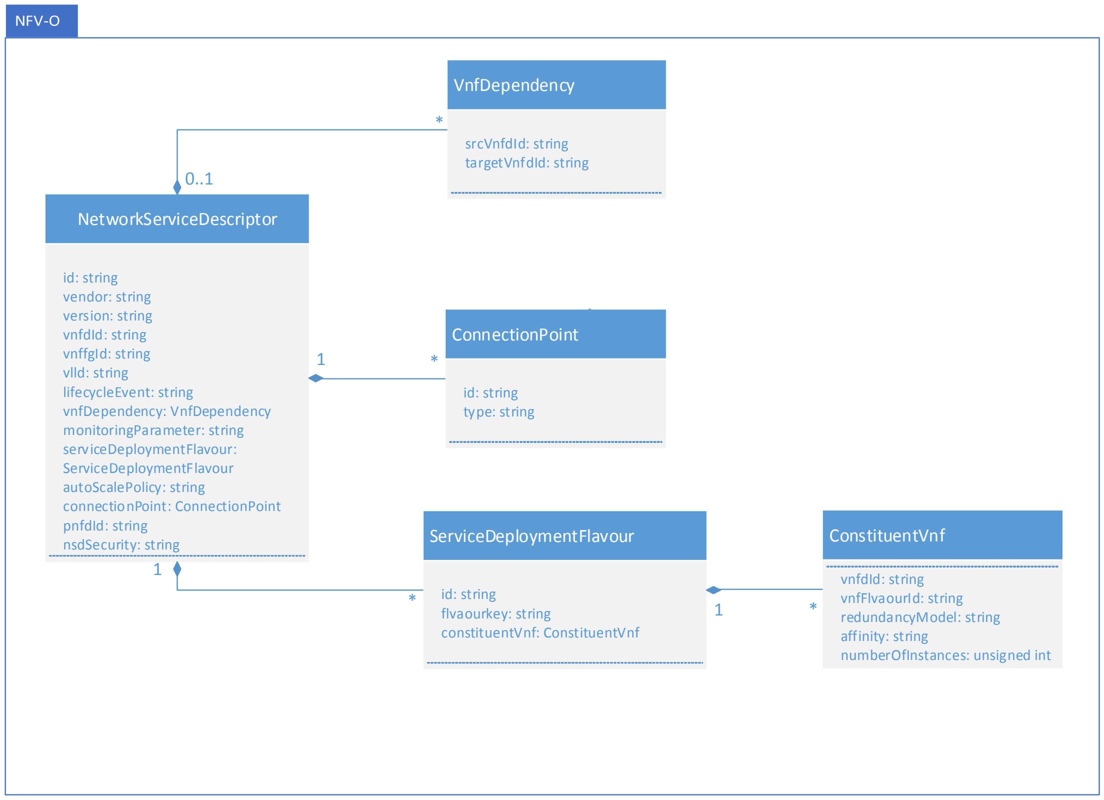


Figure 3: NFV-O information model.

Next, the main data objects are presented in more detail:

Table 5: NFV-O information model.

Parameter	Type	Description
NS Id	String	Descriptor of the NS to be instantiated
ServiceDeploymentFlavour	Object	Defines the size of the NS to be instantiated

NS record	Object	Description of the instantiated NS, its VNF instances, its status and its parameters

2.6.3 Workflow

Figure 4 and Figure 5 show the workflows to create and terminate a NS when triggered by a generic NFV-O client. In the two figures, we have assumed a simplified scenario where the NS includes only VNFs, without any PNF. If this requirement is not met, an additional interaction between the NFV-O and the SDN controller responsible for the physical network infrastructure would be required to enable the interconnection between the VNFs and the PNFs at the physical network level (which is not managed by the VIM). Moreover, we are also assuming the following:

1. The VNF Managers are already up and running.
2. No preliminary check of resource availability or resource reservation is performed before allocation (these actions are considered as optional in ETSI NFV specifications, but are usually not supported at the VIM level in state-of-the-art cloud platforms).

The detailed workflows for the instantiation and termination of VNFs are described in Section 2.7.

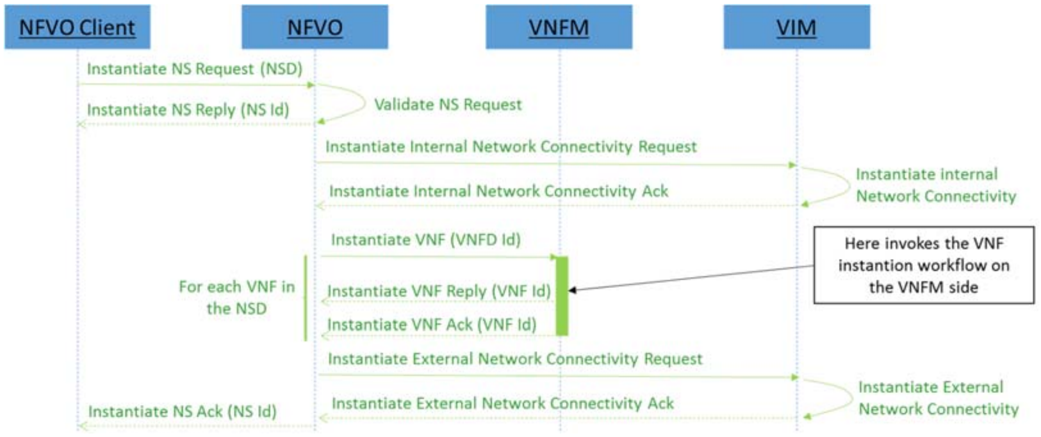

NFVO Client
NFVO
VNFM
VIM
Instantiate NS Request (NSD)
Validate NS Request
Instantiate NS Reply (NS Id)
Instantiate Internal Network Connectivity Request
Instantiate internal Network Connectivity
Instantiate Internal Network Connectivity Ack
Instantiate VNF (VNFD Id)
Here invokes the VNF instantion workflow on the VNFM side
For each VNF in the NSD
Instantiate VNF Reply (VNF Id)
Instantiate VNF Ack (VNF Id)
Instantiate External Network Connectivity Request
Instantiate External Network Connectivity
Instantiate NS Ack (NS Id)
Instantiate External Network Connectivity Ack

Figure 4: NFV-O workflow example: Instantiate NS.

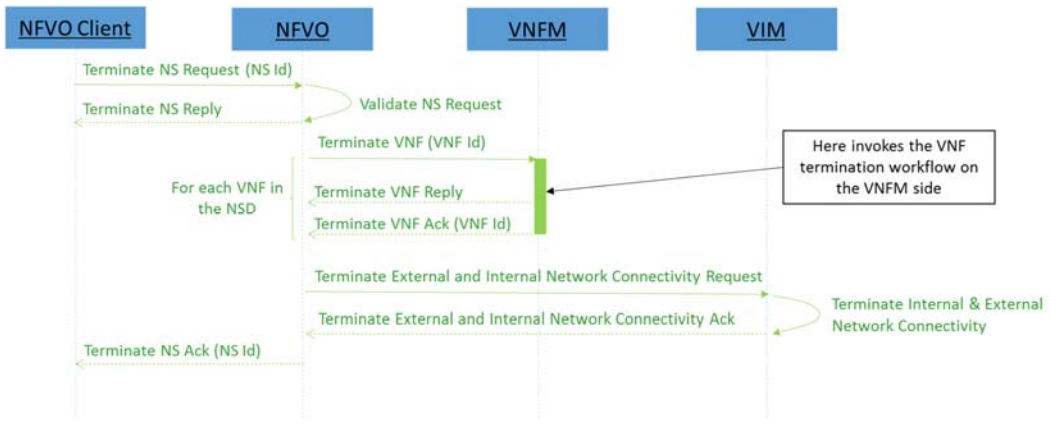

NFVO Client
NFVO
VNFM
VIM
Terminate NS Request (NS Id)
Terminate NS Reply
Validate NS Request
Terminate VNF (VNF Id)
Here invokes the VNF termination workflow on the VNFM side
For each VNF in the NSD
Terminate VNF Reply
Terminate VNF Ack (VNF Id)
Terminate External and Internal Network Connectivity Request
Terminate Internal & External Network Connectivity
Terminate External and Internal Network Connectivity Ack
Terminate NS Ack (NS Id)

Figure 5: NFV-O workflow example: Terminate NS.

The basic workflows presented above can be extended as needed to integrate the allocation of 5G-Crosshaul network resources specifically reserved to serve the traffic of a given NS. This approach involves further interactions between the NFVO and the SDN controller to setup the required network connectivity in the physical infrastructure. Further details about these procedures are provided in Section 3.2.4.

2.7 VNF Manager

The VNFM exposes NBI services to manage single VNF instances. This module offers an API to the NFV-O to conduct CRUD operations in VNFs. The role of this service is aligned with the role specified by ETSI [3]. An open source implementation compliant with the ETSI NFV specification from the functional architecture perspective can be found, for instance, in OpenBaton [15], even if based on different information models in terms of VNF Descriptors and VNFM APIs. A more detailed description including the APIs, information models and workflows involved can be found in Section 11.5.

2.8 Analytics for Monitoring

This service is in charge of offering to the consumer elaborated information obtained from the processing of the network and computing statistics gathered by the stats module specified in Section 2.4. This elaborated information could be, for instance, used by the Energy Management and Monitoring Application (EMMA) to determine whether it is convenient to change the power status of some XFE or XPU in the infrastructure. A more detailed description including the APIs, information models and workflows involved can be found in Section 11.6

2.9 Local Management Service

The LMS offers to Northbound applications the possibility of managing the status of the 5G-Crosshaul XFEs and XPUs. By status in an XFE we refer, for instance, to the capability of reconfiguring the properties of a port, or a set of ports. On the other hand, the reconfiguration of XPU status (e.g., XPU on or XPU off) and its associated properties is also considered as a potential feature offered by this service. Consequently, this module requires a DB related to the status of network and IT components that can be potentially modified by a 5G-Crosshaul application. A more detailed description including the APIs, information models and workflows involved can be found in Section 11.7.

2.10 Multi-tenancy

The Multi-tenancy service allows the Multi-tenancy Application (MTA) application or VIMaP to enforce solutions to the Virtual Network Embedding (VNE) problem [16]. It provides REST APIs to map virtual components such as virtual L2/L3 forwarding elements and virtual links that belong to a Virtual Network (VN) to the physical substrate. A more detailed description including the APIs, information models and workflows involved can be found in Section 11.8.

3 Control plane architecture

In this section, we present the design of the XCI platform. Its design is based on the guidelines described in D2.1 [5]. Figure 6 illustrates the baseline 5G-Crosshaul system architecture presented in D2.1. We divide the control plane into two layers: a top layer for external applications (see D4.1 [11]) and the XCI below. The XCI is our 5G Transport MANO platform that provides control and management functions to operate all available types of resources (networking, computing, storage). The XCI is based on the SDN/NFV principles and provides a unified platform that can be used by upper layer applications via a NBI to program and monitor the underlying data plane by a common set of core services and primitives. As mentioned in D1.1 [2], the XCI interacts with the data plane entities via a SBI in order to:

1) Control and manage the packet forwarding behavior performed by XFEs across the 5G-Crosshaul network;

2) Control and manage the PHY configuration of the different link technologies (e.g. transmission power on wireless links); and

3) Control and manage the XPUs computing operations (e.g. instantiation and management of VNFs via NFV).

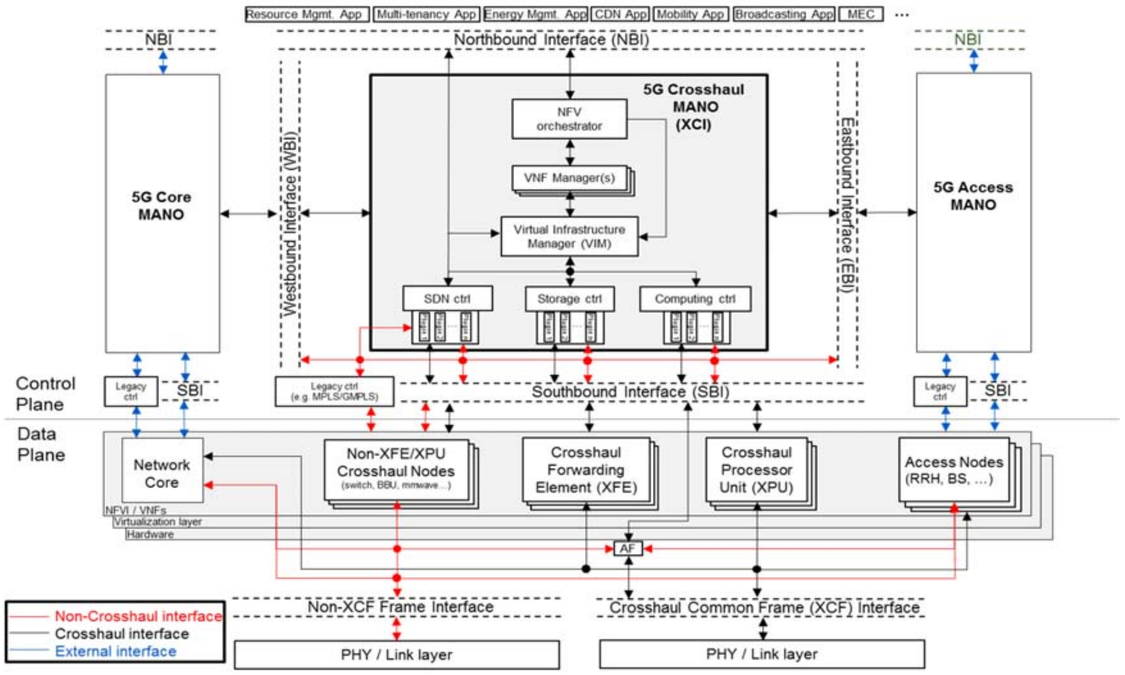


Figure 6: 5G-Crosshaul architecture illustration

3.1 High-level architecture (5G-Crosshaul MANO)

In the following, we describe in detail the 5G-Crosshaul main architecture building blocks of the control plane briefly introduced above. The XCI provides the logic controlling the overall operation of the 5G-Crosshaul. The XCI part dealing with NFV comprises three main functional blocks, namely: NFV-O, VNFM(s) and VIM (following the ETSI NFV architecture, [3]):

- The **NFV-O** is a functional block that manages a NS lifecycle. It coordinates the VNF lifecycle (supported by the VNFM) and the resources available at the NFVI (supported by the VIM) to ensure an optimized allocation of the necessary resources and connectivity to provide the requested virtual network functionality.
- The **VNFMs** are functional blocks responsible for the lifecycle management of VNF instances (e.g. instance instantiation, modification, and termination).
- The **VIM** is a functional block that is responsible for controlling and managing the NFVI computing (via *Computing ctrl*), storage (via *Storage ctrl*) and network resources (via *SDN ctrl*).

In addition to these modules, managing the different VNFs running on top of the 5G-Crosshaul, the XCI includes a set of specialized controllers to deal with the control of the underlying network, storage and computation resources:

- SDN Controller: This module is in charge of controlling the underlying network elements following the conventional SDN paradigm. 5G-Crosshaul extended current SDN support of multiple technologies used in transport networks (such as micro-wave link[2]) to have a common SDN controlled network substrate which can be reconfigured based on the needs of the network tenants.
- Computing/Storage Controllers: Storage and Computing controllers are included in what we call a Cloud Controller. A prominent example of this kind of software framework is OpenStack.

Note that the **SDN/Computing/Storage controllers** are functional blocks with one or multiple actual controllers (hierarchical or peer-to-peer structure) that centralize some or all of the control functionality of one or multiple network domains. We consider the utilization of legacy network controllers (e.g. Multiprotocol Label Switching/ Generalized Multiprotocol Label Switching (MPLS/GMPLS) to ensure backward-compatibility for legacy equipment.

3.1.1 Multi-domain control

The multi-domain transport control is a relevant aspect to consider in 5G-Crosshaul both to enable the interaction of SDN with legacy control and to support the case where several SDN controllers have to interwork. We describe in Section 3.3 the different deployment models of control interaction (e.g. peer and hierarchical), providing their comparison in different network scenarios. In this section, we report some key requirements for the extensions of the 5G-Crosshaul MANO architecture to include multi-domain transport. Multi-domain transport control is actually a relevant topic that it is currently discussed within the main standardization bodies, although a conclusion has not been reached yet. For example the Internet EngineeringTask Force (IETF) Abstraction and Control of Transport Networks (ACTN) BoF [17] proposes a hierachical architecture for the multi-domain transport but it is focused on the transport aspects with very limited description about the interaction with the virtualization functions. In [3] instead, most of the work is concentrated to the virtualization of the

[2] ONF is actively working on the definition of a SBI for micro-wave links: http://5g-crosshaul.eu/wireless-transport-sdn-proof-of-concept/

functions considering the transport as available Point to Point resources, without considering the constraints related to the interaction among different domains. In most cases, the interaction between multi-domain transport and virtualization are solved considering a sort of overprovisioning of the networking resources, but that could be a solution not viable in future due to the need to reduce the cost per bit of the transport. A practical solution to fix such issues is to define a layered architecture that assures a complete decoupling between the multi-domain transport and the virtualization layers. According to this principle, the multi-domain transport is in charge to optimize the network resources, to assure a suitable interworking among the heterogeneous transport domains and to provide a suitable exposition of the transport resources to the virtualization layer (e.g. to the NFV-O). The virtualization layer, instead, operates on the network resources using a suitable abstract view of the transport. In this model, the multi-domain transport should provide the resource exposition according to Service Level Agreements (SLA) parameters hiding the technological details of the different domains and simplifying the tasks of the virtualization layer. Moreover, the model allows independent evolution of several transport domains from legacy to SDN on virtualization functions handling and allows the XCI architecture to be quite general and applicable to concrete scenarios where operators can move towards pure SDN architecture smoothly.

The decoupling between the multi-domain transport and virtualization prevents any dependency of the general architecture of the XCI on the specific transport technology and simplifies the interaction also with the legacy control. Anyway, some issues must be solved to make the solution efficient. For example, the virtualized view provided by the multi-domain transport should be quite stable over time limiting the variation of parameters and facilitating the task of the NFV-O. Again, this could be in contrast with the resource optimization techniques applied to the transport layer where continuous change of information and data could be necessary.

Figure 7 shows the extension of the XCI control architecture, see Figure 7, with multi-domain transport control. In the diagram we highlighted the border between the virtualization and transport layer represented by the abstract view of the transport resources, this border is within the end-to-end (E2E) abstract exposition of the transport resources. The multi-domain transport control is responsible for providing the E2E abstract view based on SLA parameters and to guarantee that such values are met; while the virtualization layer utilizes the E2E abstract view as networking resources to be combined with storage and computation according to procedures defined in [3].

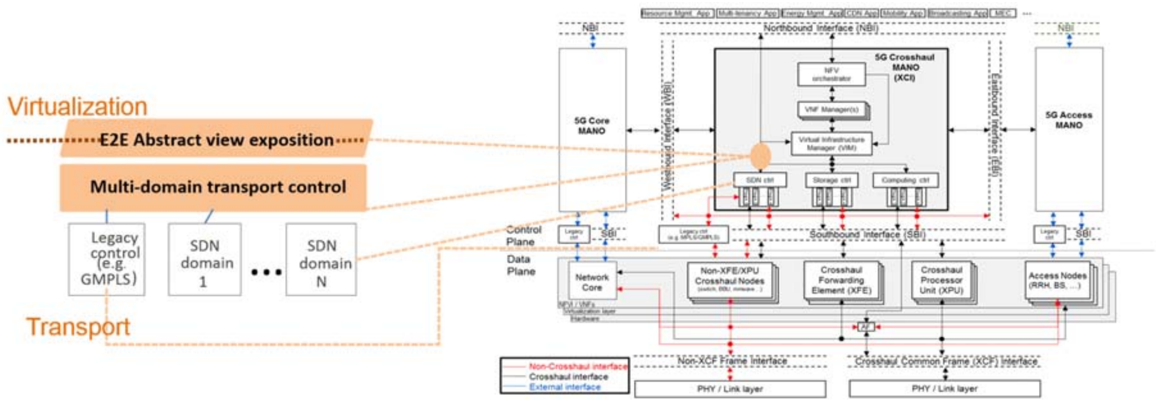


Figure 7: Multi-domain XCI architecture.

3.1.2 Multi-tenant design (XCI recursion)

The main discussion and agreements on the design of a hierarchical recursive XCI have already been presented in Section 3.3 of D1.1 [2].

3.2 XCI design

Figure 8 provides an update of the XCI design from D3.1 [1], also showing its interactions with the 5G-Crosshaul SDN applications and VNFs, which are out of WP3 scope (represented in the green boxes).

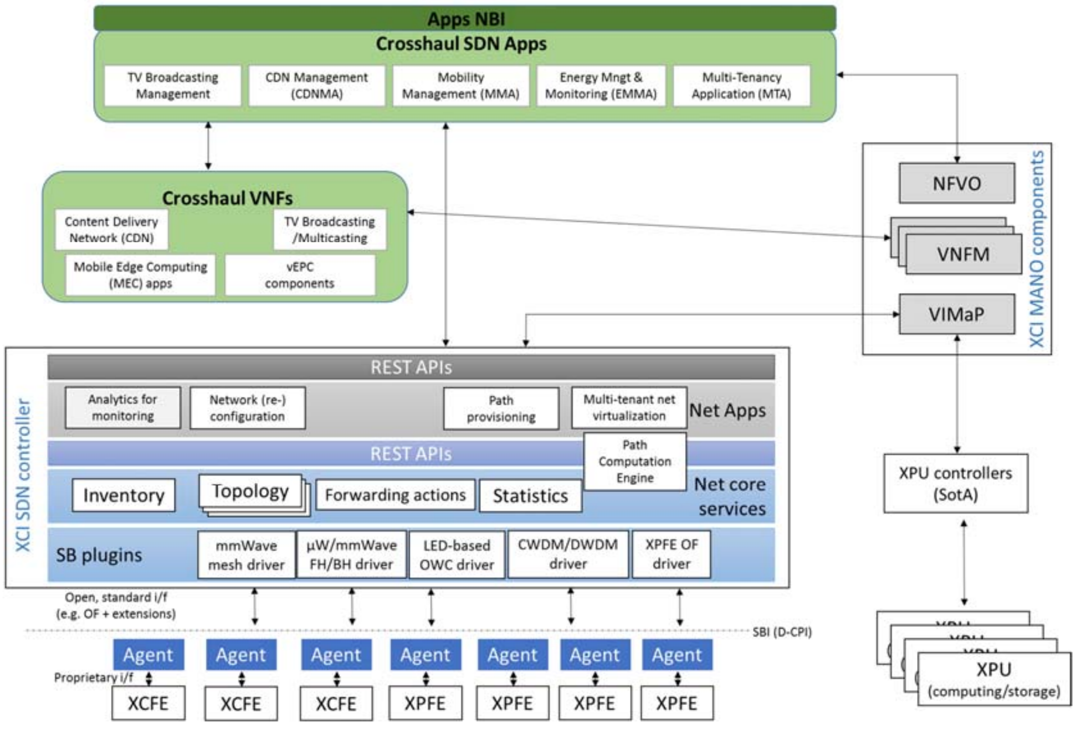


Figure 8: XCI design.

The picture highlights the two macro-modules of the XCI:

- The XCI MANO components, responsible for the instantiation, orchestration and management of VNFs and NSs (in ETSI NFV terminology, a NS is a collection of VNFs interconnected through a VNFFG and it can be considered as the equivalent of a Service Function Chain (SFC));
- The XCI SDN controller, responsible for the configuration and management of the network infrastructure.

As explained in the previous section, the XCI MANO components include the NFVO, the VNFM associated to the different 5G-Crosshaul VNFs, and the VIM. In 5G-Crosshaul, the VIM concept is extended with planning algorithms, which take efficient decisions about VMs placement and network configuration, towards integrated VIMaP functions. The controllers, in particular, the SDN controller on the network side and the XPU controllers, including both storage and computing, perform the enforcement of VIMaP decisions. XPU controllers rely on State of the Art components (e.g. OpenStack NOVA for computing controllers) and are out of scope for 5G-Crosshaul.

It should be noted that in this section we are assuming a single network domain, operated by a single SDN controller. In the case of a network infrastructure deployed in multiple domains, the general considerations described in Section 3.3 should be applied. In particular, the role of the SDN controller should be decomposed in several “child” controllers operating at each domain and abstracting the internal details of the local resources, with a hierarchical “parent” controller in charge of computing and allocating end-to-end and inter-domain connections. This is done through the coordination of the lower controllers’ actions, which are the final responsible of the actual, technology-dependent resource configuration. Further details about the applicability of the SDN hierarchical approach is provided in section 3.3, where different deployment models are investigated and compared.

The XCI SDN controller has the same macro-architecture described in D3.1 [1], with three internal layers: *SBI drivers* for the interaction with heterogeneous devices at the data plane; *network core services* implementing basic monitoring, configuration and inventory functionalities; and *network applications* implementing the network level logic. Both network core services and network applications provide NBIs which can be used by the XCI MANO components (mainly the VIMaP) and the 5G-Crosshaul applications to program the network.

The main updates from the initial design are related to the network applications, because some of the functionalities previously included at the SDN controller level have been entirely moved to the 5G-Crosshaul application level. The following network applications are embedded in the SDN controller:

- Analytics for monitoring: an advanced monitoring service used to correlate raw network monitoring data originally provided by the statistics service. It can be used, for example, to elaborate monitoring information related to a VN based on statistical data on flows and physical ports. This functionality offers an NBI to applications, which is detailed in Sections 2.8 and 11.6.
- Network (re-)configuration: used to re-configure specific network elements, mainly for management purposes. It is also used by the EMMA application to change the status of the devices for energy saving issues. This functionality is embedded in the Local Management service detailed in Sections 2.9 and 11.7.

- Path provisioning: service which establishes a network path between a source and one (or more) destination end-point(s). It can take as input several constraints, including the specification of the path itself. This option allows upper layer applications to implement their own allocation algorithms and use the path provisioning service as a sort of “configuration arm”. If no path is included in the input parameters, the internal path computation engine core service is invoked. This functionality offers an NBI detailed in Sections 2.2 and 11.1.
- Multi-tenant network virtualization: service which builds isolated and virtualized networks over the shared physical infrastructure. The multi-tenancy logic (i.e. the mapping between a Virtual Infrastructure –VI– and its tenant) can be kept at the SDN controller level, but it may be also implemented at the upper layers e.g. at the VIMaP, at the NFVO level or at the application level (MTA). The support for resource allocation and isolation are needed for all involved networking and computing elements, and a coherent management is required for unifying the concepts of infrastructure virtualization. The MTA application provides such management, it uniformly wraps and complements the infrastructure elements’ (SDN controllers, cloud management systems, network elements, etc.) capabilities to provide multi-user and resource isolation support, offering uniform and abstracted views to tenants. The mandatory features of the network virtualization service are the consistency between the VI description and the VI exposed at the SDN controller NBI level and the isolation between coexisting VIs. This functionality offers an NBI summarized in Sections 2.10 and 11.8.

The network core services are the same as defined in D3.1 [1], while the SBI drivers have been specialized according to the latest outcomes of WP2, which has identified the different technologies of the 5G-Crosshaul data plane. Each SBI driver implements mechanisms for receiving inventory and monitoring data from the devices, including technology specific parameters, and for configuring some of their management parameters and their forwarding behavior.

The following sections match the components identified in the XCI design and possible software frameworks and platforms to be used for their implementation.

3.2.1 Alternatives for NFVO/VNFM/VIM

Some open source software alternatives for the XCI MANO components have been analyzed in D1.1 [2]. The XCI architecture does not mandate any specific software platform; the same XCI functions can be developed starting from different open source or proprietary projects, with the unique constraint of being compliant with the interfaces defined at the NBI and SBI of each component in order to support the proper workflows and interactions.

The proof of concept prototypes of different 5G-Crosshaul partners are based on the components described in the following table:

Table 6: Software platforms used in XCI MANO prototypes.

Functional Component	**Software baseline**	**Features / Use case**	**5G-Crosshaul partner**
NFVO	proprietary	Orchestration of virtual Evolved Packet Core (vEPC).	NXW
	proprietary	Orchestration of Content Delivery network (CDN) nodes.	ATOS
VNFM	proprietary	Management of vEPC VNFs lifecycle.	NXW
	proprietary	Orchestration of CDN nodes VNFs lifecycle.	ATOS
VIMaP	OpenStack [8]	Provisioning of vEPC VNFs and their interconnections with QoS and energy constraints.	NXW
	OpenStack	Provisioning of CDN origin and replica servers on XPUs and SFC configuration.	ATOS
	OpenStack + proprietary	Allocation and management of VMs and their interconnections.	CTTC

3.2.2 Alternatives for storage and computing control

Storage and computing controllers will be based on state-of-the-art components, in particular on the corresponding OpenStack modules. No further extensions are required.

3.2.3 SDN controllers

As in the XCI MANO components case, the architecture does not impose any specific choice on the SDN controller software platform for the reference implementation. An analysis of possible alternatives has been provided in D3.1 [1] and D2.1 [5].

The table below reports the software baselines used for the implementation of proof-of-concept prototypes.

Table 7: Software platforms used in XCI SDN controller prototypes.

Functional Component	**Software baseline**	**Features / Use case**	**5G-Crosshaul partner**
SBI driver	ODL [6]	XPFE forwarding control via OF. Collection of energy consumption parameters from XPFE.	NXW

		Configuration of XPFE device state.	
	Ryu [18]	Configuration and control of mmWave nodes, based on OF for forwarding and REST for retrieval and configuration of port parameters.	CTTC
	ODL	Configuration, control and management of mmWave mesh nodes. Forwarding control via OF for mmWave mesh nodes.	Interdigital
	Ryu	Configuration and control of emulated mmWave nodes, based on OF for forwarding and REST for retrieval and configuration of port parameters.	Interdigital
Network core services (inventory, topology, statistics, flow actions)	ODL	Extensions to topology & inventory modules for power-consumption information.	NXW
	Ryu	Core services for mmWave technology.	CTTC
	ODL	Core services for mmWave mesh nodes.	Interdigital
	Ryu	Core services for emulated mmWave platform.	Interdigital
Path computation	ODL	Path computation for energy-efficient network path.	NXW
	-- (analytical algorithms)	Network optimization.	UC3M
	-- (analytical algorithms) ODL	Algorithms for optimal path computation and service embedding in multi-technology transport network (ETH + mmWave).	CREATE-NET

	Floodlight [19], analytical algorithms	Computation of path between RUs, DUs, and XPUs.	NEC
	Ryu/ABNO	Per-domain and multi-domain path computation (mmWave/WiFi and Optical).	CTTC
Path provisioning	ODL	Path provisioning for energy-efficient network path.	NXW
	ODL	Provisioning of SFC networking in support of CDN infrastructure.	ATOS
	Ryu/ABNO	Path Provisoning of per-domain and multi-domain.	CTTC
	ODL	Configuration and control of analogue radio over fibre (A-RoF) nodes, based on real-time mobility of the High-Speed Train to save energy.	ITRI
Network re-configuration	ODL	NBI methods to change the status of XPFE (to support EMMA)	NXW
Analytics for monitoring	ODL	Elaboration of energy-related monitoring information for paths and virtual infrastructures.	NXW
	Ryu	Configuration and maintenance of multiple Tenants in a shared environment. Onboard and on land controllers work independently.	ITRI
Parent SDN controller (see section 3.3)	Proprietary	ABNO-based parent controller with Control Orchestration Protocol (COP) (proprietary protocol) interaction with child SDN controllers.	CTTC

3.2.4 Integration between SDN controller and MANO components

So far, we have analyzed possible solutions for NFV tools able to deploy sets of VNFs organized in NSs and for SDN controllers responsible for the configuration of the network connectivity in the 5G-Crosshaul physical infrastructure. However, these two entities (NFV tools and SDN controller) need to operate in a joint and coordinated manner to deliver the virtual resources associated to the NSs with a consistent set of underlying connections that enable the network traffic.

In general, 5G NSs are virtual entities composed of VNFs interconnected through logical network topologies (i.e. the VNF Forwarding Graphs) represented with virtual links and connection points in the ETSI MANO terminology. At the VIM level, virtual links are mapped on virtual networks, while connection points are mapped to port resources. However, all these virtual resources are just logical representations that, in order to deliver traffic between the VMs running the VNF software, need a mapping to the underlying network, with coherent connectivity at the physical layer. The configuration of the physical connectivity should be performed as-a-Service, as an integrated procedure in the provisioning of the NS logical infrastructure. In other terms, when the VIM (e.g. OpenStack) creates the VMs and the related VNs, in parallel the SDN controller should configure the physical data plane to provide a coherent transport service, complying to the physical network capabilities and the QoS requirements specified by the end-users. The whole procedure has to be coordinated, so that the physical configuration is dynamically adapted to the traffic that will be generated by the VMs. For example, the traffic flows handled at the SDN controller will have to be classified according to the specific kind of network segments defined at the VIM configuration (e.g. VLAN, VXLAN, GRE) and based on segment IDs, VMs MAC and IP addresses selected at the VIM itself. Moreover, source and destination of flow connections depend as well on the VMs physical placement, which is decided at runtime and managed by the MANO components.

As a consequence, the XCI has to include an application logic able to handle the coordination between NFV MANO and SDN controller actions, providing a higher layer of resource orchestration for the end-to-end service delivery.

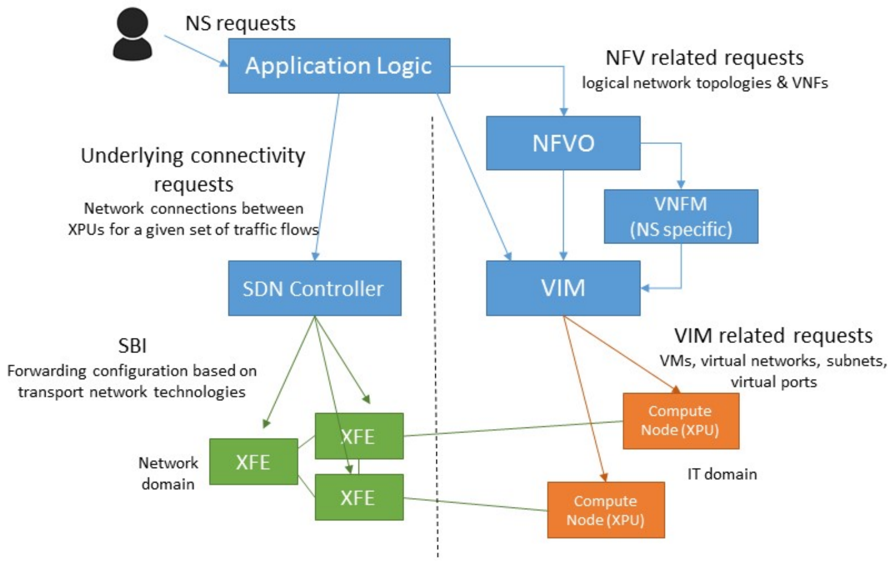

NS requests
Application Logic
NFV related requests
logical network topologies & VNFs
NFVO
VNFM (NS specific)
Underlying connectivity requests
Network connections between XPUs for a given set of traffic flows
SDN Controller
VIM
SBI
Forwarding configuration based on transport network technologies
VIM related requests
VMs, virtual networks, subnets, virtual ports
Compute Node (XPU)
XFE
XFE
XFE
Network domain
IT domain
Compute Node (XPU)

Figure 9: Functional architecture for a generic application requesting NSs

Figure 9 presents a possible architectural solution for the integration between the SDN controller and the MANO components. Here the Application Logic functional block implements the generic logic of a NS provisioning workflow and coordinates SDN controller and NFV MANO actions for service lifecycle and resource management. At the implementation level, with reference to the 5G-Crosshaul applications, the Application Logic may also implement additional functionalities or integrate algorithms that are specifically related to the target NS. For example, customized resource allocation algorithms may be implemented for each 5G-Crosshaul application (e.g. energy-aware algorithms for EMMA algorithms, multicast tree algorithms for Television Broadcasting Application (TVBA) and specific actions on physical and virtual resources may be needed in the workflow (e.g. power status configuration in EMMA). It should be noted that the Application Logic is a functional element that, regarding implementation, can be integrated into different architectural components; for example, it can be embedded in the applications or integrated as part of the NFVO. In the last case, the orchestrator should be extended with application-specific scheduling algorithms and drivers to trigger the underlying network configuration via SDN controller. Typically, for applications without specific requirements in terms of resource allocation (i.e. allocation driven by application-level criteria), the implementation of the application logic functions at the NFVO level is more efficient. In fact, it limits the interaction between applications and XCI for the collection of topology and resource capabilities data. This interaction is usually time consuming and, depending on the dimension of the network, requires the exchange of several data via REST Hyper Text Transfer Protocol (HTTP) messages to describe the full topology. Moreover, the implementation within the NFVO enables the usage of internal interfaces and direct queries to databases which are faster.

A typical workflow for requesting the deployment of a Network Service includes the following steps:

1. Computation of resource allocation solution: this step takes decisions about the physical resources to be allocated for the given NS. It can be handled by the Application Logic in a centralized manner, taking joint decisions about servers and network paths. Otherwise, delegating this computation independently to the underlying components may lead to suboptimal solutions, due to the disjoint computation where e.g. the VMs placement is decided by the VIM, and the network paths are computed at the SDN controller.
2. Creation of NS Virtual Links: handled at the MANO side under the coordination of the NFVO, the VIM creates a set of virtual networks corresponding to the virtual links in the logic network topology. At this stage, no connections are configured on the physical network.
3. Creation of the VNFs: handled at the MANO side, through the cooperation of NFVO, VNFM, and VIM. In this step, the VIM creates network ports (corresponding to the connection points of the VNFs) and VMs placed in the Compute Nodes (i.e. the XPUs).
4. Setup of the underlying network connectivity: this step is handled by the SDN controller, which configures the physical infrastructure and establishes the network paths that interconnect the XPUs via XFEs. This network path will carry the traffic among the VMs.

3.2.4.1 Reference PoC implementation

In the 5G-Crosshaul project, the integrated MANO-SDN solution has been implemented in the XPFE and XCI PoC prototype developed in WP3. The implementation is based on a reference scenario with Lagopus-based XPFEs as network data plane, OpenStack as VIM and ODL as SDN controller (Figure 10). At the NFVO level, the developments have adopted different Orchestrators and VNFMs as software baseline, i.e. proprietary NFVO/VNFM components developed by partners in the consortium. The usage of specific NFVOs and VNFMs depends on the specific application scenario (i.e. NSs composed of VNFs with vEPC, CDN and TV Broadcasting components – see [11] for further details on the demonstrators' deployment) and related resource allocation algorithms and VNFs' configuration. In the scenarios presented below, related to the EMMA application for the proviosioning of vEPC services, the application logic has been implemented as internal extensions of the NFVO.

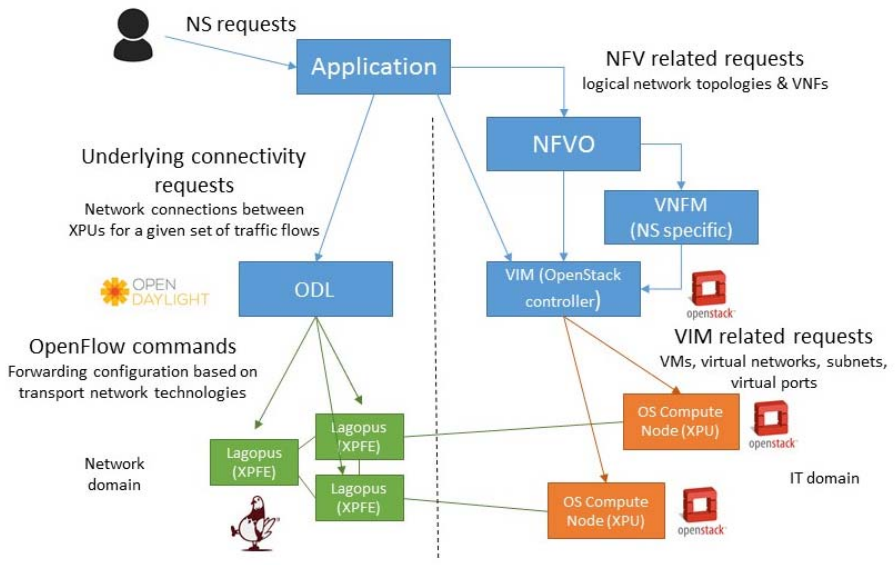

NS requests
Application
NFV related requests
logical network topologies & VNFs
NFVO
VNFM
(NS specific)
Underlying connectivity requests
Network connections between XPUs for a given set of traffic flows
ODL
OPEN DAYLIGHT
VIM (OpenStack controller)
openstack
VIM related requests
VMs, virtual networks, subnets, virtual ports
OpenFlow commands
Forwarding configuration based on transport network technologies
Lagopus (XPFE)
Lagopus (XPFE)
Lagopus (XPFE)
OS Compute Node (XPU)
OS Compute Node (XPU)
Network domain
IT domain

Figure 10: 5G-Crosshaul reference deployment

In the following, we present an example workflow for the provisioning of a simple NS composed by 2 VNFs interconnected by a virtual link. For each step (1-4) we provide a brief description of the interactions between the architecture components and the results of the actions.

Step 1 - Computation of resources allocation

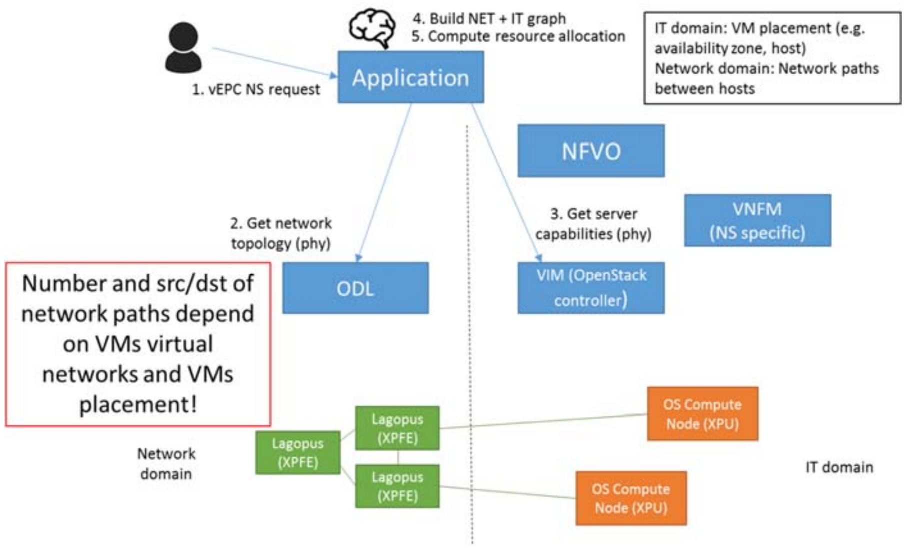

4. Build NET + IT graph
5. Compute resource allocation
IT domain: VM placement (e.g. availability zone, host)
Network domain: Network paths between hosts
1. vEPC NS request
Application
NFVO
2. Get network topology (phy)
3. Get server capabilities (phy)
VNFM (NS specific)
Number and src/dst of network paths depend on VMs virtual networks and VMs placement!
ODL
VIM (OpenStack controller)
Lagopus (XPFE)
OS Compute Node (XPU)
Network domain
IT domain

Figure 11: Computation of resources allocation workflow

1. The end-user requests the provisionig of a vEPC NS using the APIs exposed by the Application.
2. The Application interacts with the NBI of the SDN controller (ODL) to retrieve the physical network topology through the REST APIs of the Topology Manager service.
3. Similarly, the Application interacts with the VIM to get the information about XPUs' physical capabilities and available resources.
4. Making use of the information retrieved in steps 2 and 3, the application builds a global network graph which includes both network and computing nodes.
5. Based on the graph elaborated in step 4, the application algorithms compute the resource allocation solution, which includes the compute nodes where the VMs need to be deployed and the network paths to interconnect this compute nodes (see Figure 12).

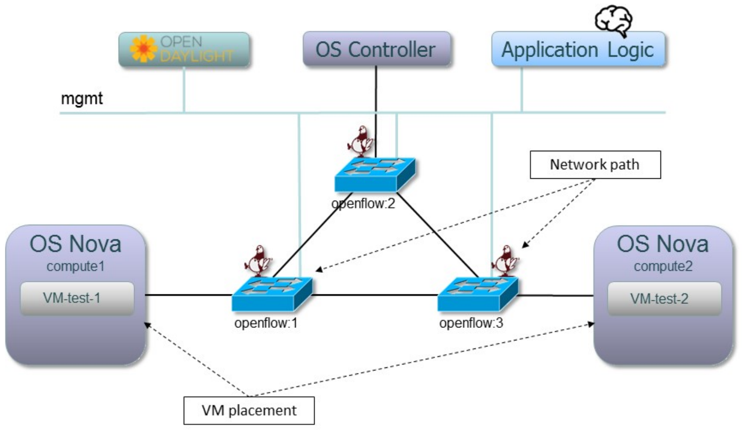


Figure 12: Computed VMs placement and Network path

Step 2 - Creation of NS Virtual Links

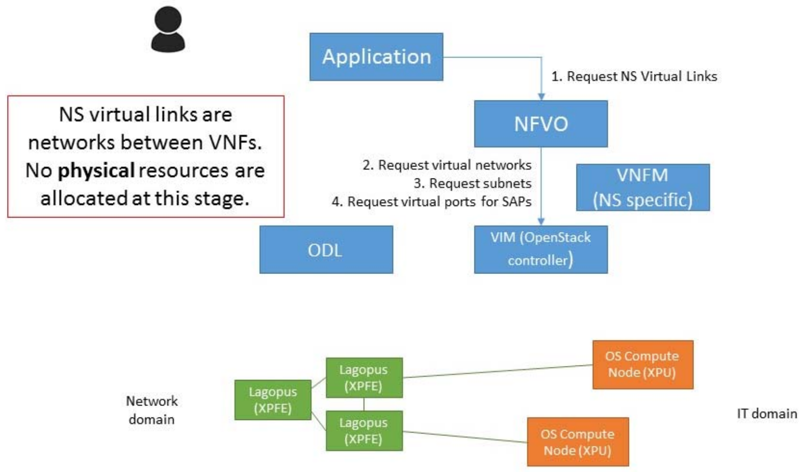

Application
1. Request NS Virtual Links
NFVO
NS virtual links are networks between VNFs. No **physical** resources are allocated at this stage.
2. Request virtual networks
3. Request subnets
4. Request virtual ports for SAPs
VNFM (NS specific)
ODL
VIM (OpenStack controller)
Lagopus (XPFE)
Lagopus (XPFE)
Lagopus (XPFE)
OS Compute Node (XPU)
OS Compute Node (XPU)
Network domain
IT domain

Figure 13: Creation of NS Virtual Links

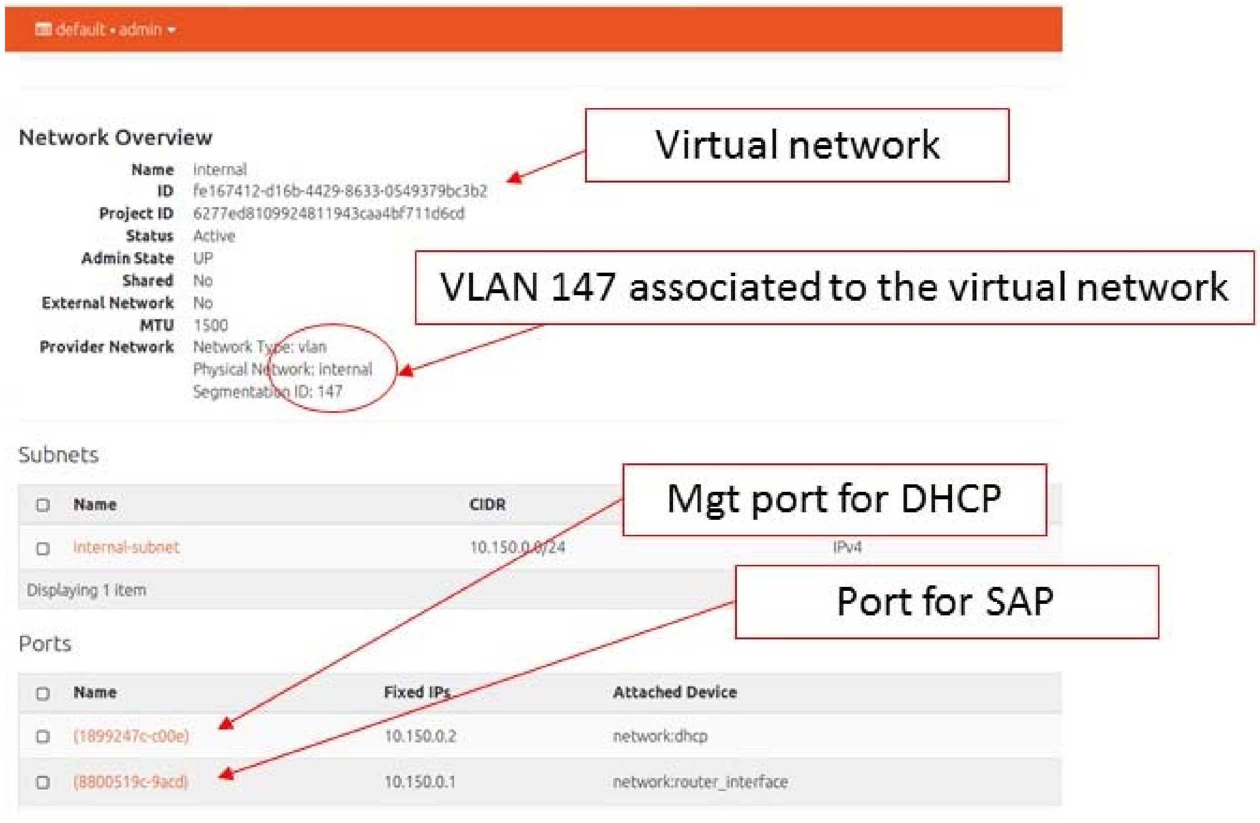
1. In order to create the logical network topology for the NS, the Application requests the creation of the NS Virtual Links to the NFVO.

2-4. The NFVO translates each Virtual Link to the corresponding resources at the VIM level. In particular, the NFVO asks the OpenStack VIM to create three types of resources: VNs, subnets and virtual ports for the Service Access Points (i.e. the external connection points that enable the interaction between the NS and external networks).

At this stage, OpenStack has created the network-related resources shown in Figure 14 for the specific virtual link of our example. Here, OpenStack has associated the segmentation ID (i.e. the VLAN ID) 147 to the network: this means that the traffic generated by the VMs on the ports connected to this network are tagged with VLAN ID 147 when exiting from the physical servers. This parameter is used during the creation of the underlying network connectivity to specify the classifier of the corresponding traffic flow.

The NFVO requests also the creation of a subnet to enable L3 traffic on the given network. Since the subnet is specified with DHCP enabled, a port is implicitly created on the network for management purposes and it is used for the DHCP traffic with the Neutron service.

The second port, instead, is created explicitly on-demand and corresponds to a Service Access Point of the NS. In order to enable the interaction with external entities, the port is attached to a VIM router (that we assume pre-existing) connected to an OpenStack external network.



default • admin ▾

Network Overview

Name	internal
ID	fe167412-d16b-4429-8633-0549379bc3b2
Project ID	6277ed8109924811943caa4bf711d6cd
Status	Active
Admin State	UP
Shared	No
External Network	No
MTU	1500
Provider Network	Network Type: vlan Physical Network: internal Segmentation ID: 147

Subnets

Name	CIDR	
internal-subnet	10.150.0.0/24	IPv4

Displaying 1 item

Ports

Name	Fixed IPs	Attached Device
(1899247c-c00e)	10.150.0.2	network:dhcp
(8800519c-9acd)	10.150.0.1	network:router_interface

Figure 14: Creation of OpenStack resources for NS Virtual Links

Step 3 - Creation of VNFs

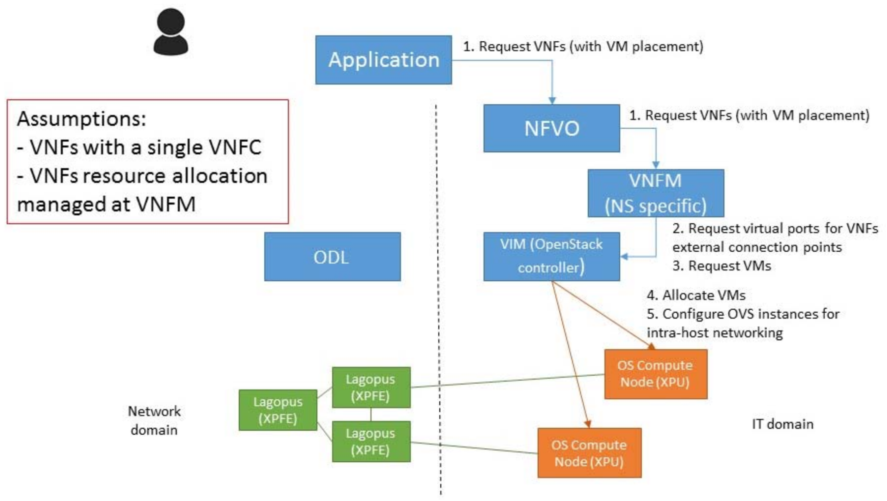

Application
1. Request VNFs (with VM placement)
Assumptions:
- VNFs with a single VNFC
- VNFs resource allocation managed at VNFM
NFVO
1. Request VNFs (with VM placement)
VNFM (NS specific)
ODL
VIM (OpenStack controller)
2. Request virtual ports for VNFs external connection points
3. Request VMs
4. Allocate VMs
5. Configure OVS instances for intra-host networking
Lagopus (XPFE)
Lagopus (XPFE)
Lagopus (XPFE)
OS Compute Node (XPU)
OS Compute Node (XPU)
Network domain
IT domain

Figure 15: Creation of VNFs

Assuming each VNF is composed of a single VNF Component (VNFC) and the VNFs resource allocation handled by the VNFM, the workflow for the creation of the VNFs is the following:

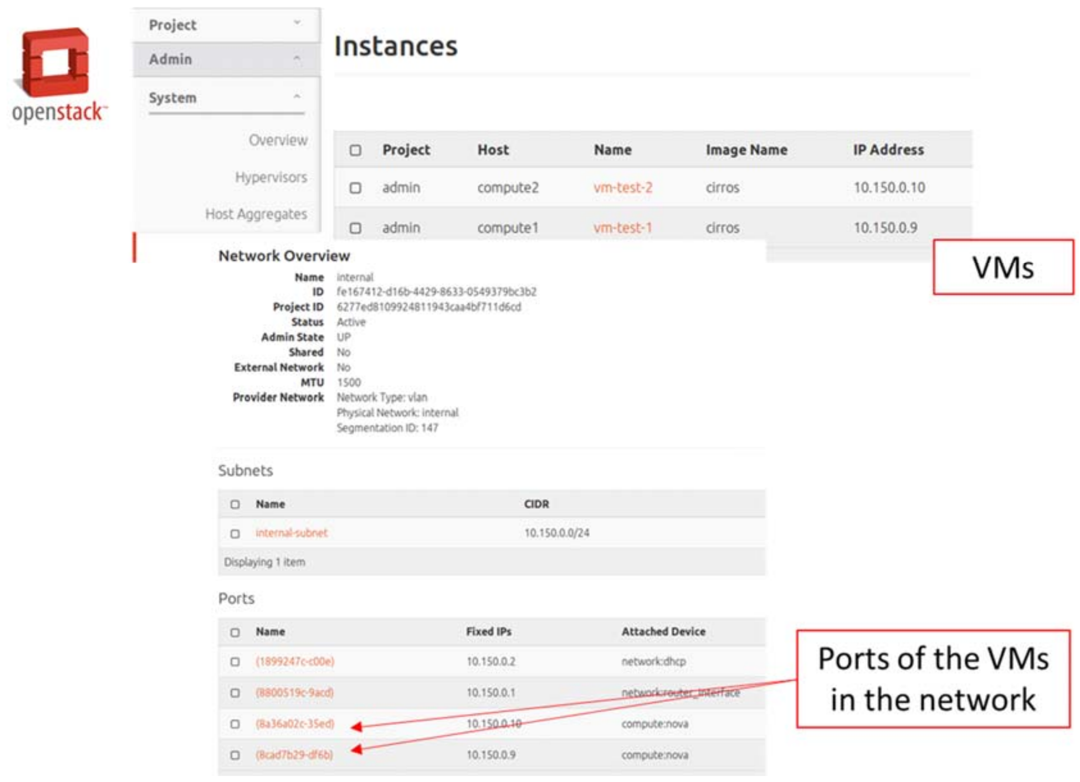
1. The Application requests the VNFs creation, specifying the VMs placement computed in Step 1, using the APIs exposed by the NFVO. The NFVO delegates the VNFs deployment to the VNFM, which is a VNF-specific entity.

2-3. The VNFM asks the VIM to create the virtual resources to instantiate each VNF. In particular, for each VNF, the VIM must create a virtual port for each VNF external connection point and a VM for each VNFC. Each virtual port must be connected to the virtual networks created in Step 2.

4. The VIM instantiates the VMs on the OpenStack compute nodes, according to the desired VMs placement. In this case, the VM deployment is forced to use two different hosts, in order to show the configuration of the physical network between the compute nodes.

5. The VIM configures the OpenVSwitch instance in each compute node for intra-host network traffic, i.e. for the traffic between VMs located on the same compute node or between VMs and the compute node's NIC for traffic towards other hosts.

The OpenStack resources created in this phase are shown in Figure 16. The two VMs vm-test-1 and vm-test-2 have been created on the hosts compute1 and compute2 respectively. Two additional ports have been created on the network instantiated in Step 2, each of them attached to a VM. The IP addresses, assigned automatically by OpenStack on the Classless Inter-Domain Routing (CIDR) of the subnet created in Step 2, are shown in the VMs table. As depicted in Figure 17, the traffic generated by the VMs and exiting from the hosts is tagged with VLAN ID 147, i.e. the segmentation ID assigned to the VN where the two ports are attached.



Instances

	Project	Host	Name	Image Name	IP Address
☐	admin	compute2	vm-test-2	cirros	10.150.0.10
☐	admin	compute1	vm-test-1	cirros	10.150.0.9

Network Overview

Name	internal
ID	fe167412-d16b-4429-8633-0549379bc3b2
Project ID	6277ed8109924811943caa4bf711d6cd
Status	Active
Admin State	UP
Shared	No
External Network	No
MTU	1500
Provider Network	Network Type: vlan Physical Network: internal Segmentation ID: 147

Subnets

	Name	CIDR
☐	internal-subnet	10.150.0.0/24

Displaying 1 item

Ports

	Name	Fixed IPs	Attached Device
☐	(1899247c-c00e)	10.150.0.2	network:dhcp
☐	(8800519c-9acd)	10.150.0.1	network:router_interface
☐	(8a36a02c-35ed)	10.150.0.10	compute:nova
☐	(8cad7b29-df6b)	10.150.0.9	compute:nova

Figure 16: Creation of OpenStack resources for VNFs

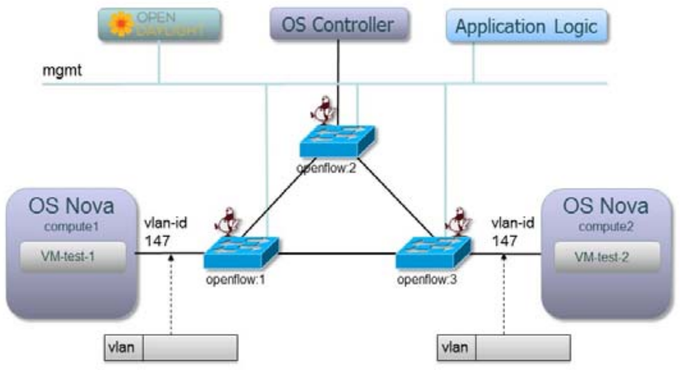


Figure 17: Tagging of traffic between compute nodes

Step 4 - Creation of the underlying connectivity

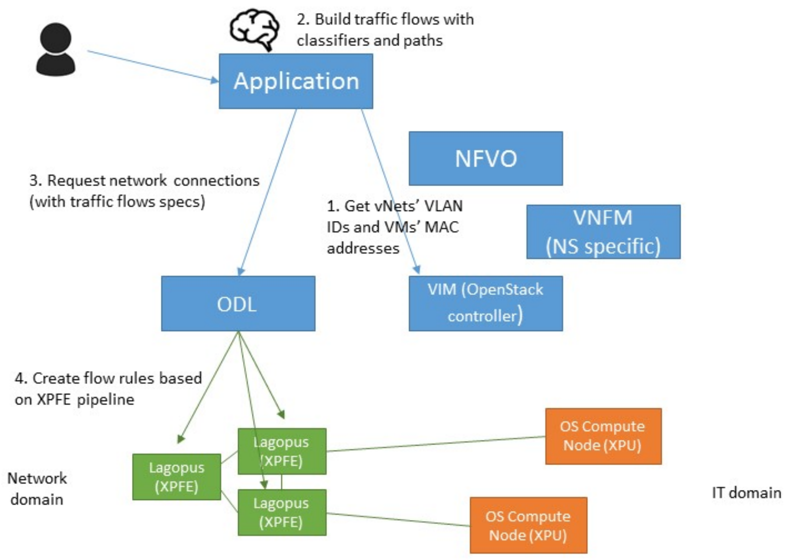

2. Build traffic flows with classifiers and paths
Application
NFVO
3. Request network connections (with traffic flows specs)
1. Get vNets' VLAN IDs and VMs' MAC addresses
VNFM (NS specific)
ODL
VIM (OpenStack controller)
4. Create flow rules based on XPFE pipeline
Lagopus (XPFE)
Lagopus (XPFE)
Lagopus (XPFE)
OS Compute Node (XPU)
OS Compute Node (XPU)
Network domain
IT domain

Figure 18: Creation of the underlying connectivity

The last step is the creation of the underlying connectivity on the physical network, which is handled mostly at the SDN controller level.

1. The Application queries the VIM to retrieve the network-related information needed to build the traffic flow requests for the SDN controller. In particular, the VLAN ID used as segmentation ID for the virtual networks and VMs MAC addresses are needed to define the classifiers of each traffic flow.
2. The Application builds the specification of the traffic flows to be installed in the data plane, taking into account the VMs placement (i.e. source and destination of traffic flows), traffic classification (MAC addresses and VLAN IDs), resource allocation solution for the network paths as computed in step 1, and requested QoS.
3. The Application invokes the REST API exposed by the Provisioning Manager module of ODL and requests the network connections setup, based on the calculated paths and classifiers.
4. The Provisioning Manager, making use of the mechanisms implemented by the OpenFlow Plugin of ODL, creates the suitable flow rules on the Lagopus-based XPFEs, following the XPFE pipeline format. In particular, the traffic is received at the source edge XPFE tagged with the VLAN ID determined in step 1. On the edge nodes of the network path traffic is further tagged and un-tagged with an outer VLAN ID decided by the SDN controller and the forwarding within the network domain is based on this outer VLAN ID only (see Figure 19) (the initial VLAN ID becomes the inner VLAN ID). The XPFEs along the target network paths are now able to forward the traffic between the VMs.

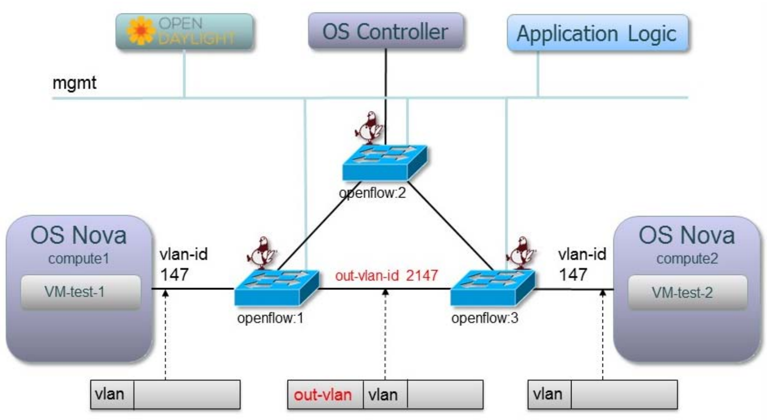


Figure 19: Configuration of underlying network connectivity at the XPFE data plane

3.2.4.2 Customization of NS provisioning workflows

The reference scenario, as well as the general workflow for the provisioning of a NS, can be customized to better fit the needs of a specific NS or an Application. In this section, we discuss two different options for the workflow customization: the integration of specialized algorithms for resource management and the introduction of additional functions to manage physical resources.

Algorithms for Resource Management

The decisions related to resource allocation and management can be performed using different kind of algorithms, able to apply specific constraints and objective functions.

In general, resource allocation algorithms can take decisions either about IT and network resources (i.e. VMs allocation at XPUs and network paths in XFE domains) in a single joint step or just about VMs placement in a first round and, after that, about network paths in a second round. The former joint approach is adopted, for example, in the EMMA application. In this case, the algorithm is typically integrated at the application level or in the NFVO (option 1 in Figure 20) since it needs information about the availability of IT resources, network topology and capabilities of network links, which can be retrieved from the combination of VIM and SDN controller.

The approach based on a disjoint, per domain resource management can involve algorithms integrated into different components. For example, the IT computation can be placed at the application level, while the computation of the network paths to interconnect the VMs can be delegated to the SDN controller (option 2 in Figure 20). Another possibility is represented by option 3 in Figure 20. Here the application logic is only responsible for the workflow coordination, while all the resource allocation decisions are taken at the XCI level, in particular at the NFVO for the IT resources and at the SDN controller for the network resources (in this case, the network related computation if performed by the internal Path Computation Engine component – see Figure 8.

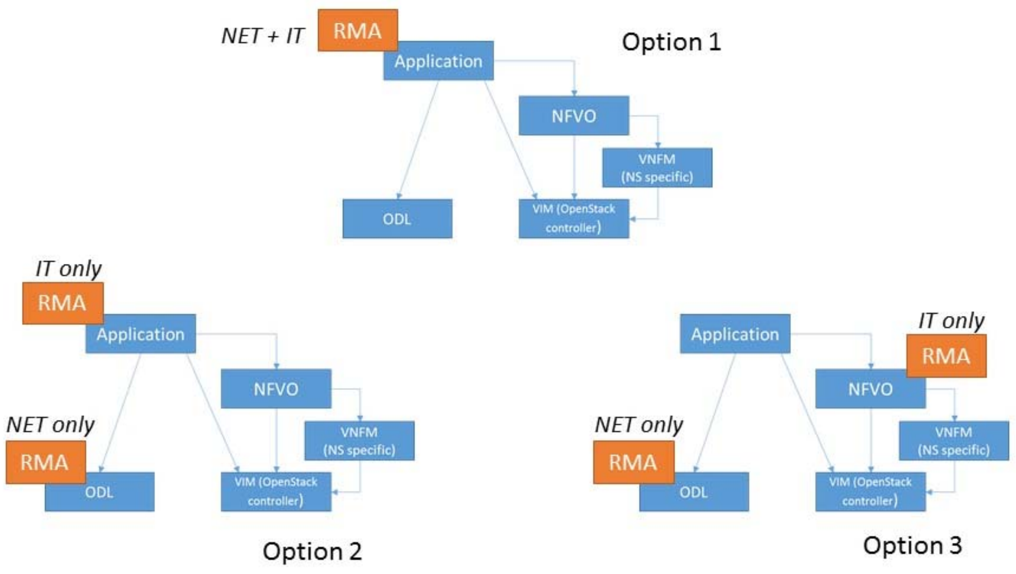


Figure 20: Options for resource management integration

Additional functions in provisioning workflow

At the higher orchestration level, the NS provisioning workflow can be modified also to support specific functionalities required by applications. In the case of EMMA, for example, the general workflow has been modified to configure dynamically the power-state of XPUs or XPFEs and switch on the devices only when needed to instantiate the virtual resources (see Figure 21). The rationale for this is to enable a joint planning of XPUs and XFEs switching on/off, with optimal resource placement. A disjoint planning may instead lead to suboptimal solutions or even XPUs that cannot be interconnected at all. In particular, the EMMA application has been implemented with specific drivers to modify the power-state of XPUs and XPFEs via OpenStack and ODL APIs. The corresponding functions have been implemented at the XCI as follows:

- Specific methods are exposed at the ODL controller NBI for XPFE's power-state configuration and monitoring. Moreover, a specific Simple Network Management Protocol (SNMP) protocol driver has been implemented to enforce the related XPFE configuration for the power-state and read power consumption values.
- On the MANO side, power consumption monitoring and power-state management is handled through a plugin within the OpenStack control node, and a driver in each OpenStack compute node. The implementation of the compute node driver depends on the specific type of hardware and the mechanisms provided by the server to handle power states (e.g. DELL iDRAC).

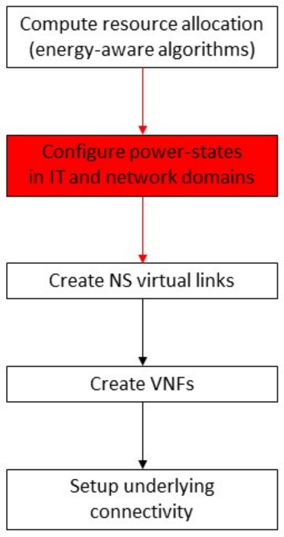

Compute resource allocation (energy-aware algorithms)
Configure power-states in IT and network domains
Create NS virtual links
Create VNFs
Setup underlying connectivity

Figure 21: EMMA workflow

3.3 Deployment models of XCI

In this section, we will discuss different deployment models that fit within the XCI architectural design. The following summarizes the different models to control and manage networking, compute, and storage resources.

It is commonly accepted that deploying a single, integrated controller for a large or complex infrastructure including network, compute, and storage resources presents scalability issues, or may not be doable at all in practice. In particular:

- The infrastructure size, in terms of controllable elements, has a direct impact on the controller requirements regarding e.g. the number of active and persistent TCP connections on top of which control sessions are established, memory requirements to store in memory e.g. a data structure representing the network graph that abstracts the network and Central Processing Unit (CPU) requirements for processing message exchange, or implement control logic
- The infrastructure complexity in terms of multiple deployed technologies (such as a packet-switched layer for Layer2/Layer3 transport over a circuit-switched optical layer, each having intrinsic and non-generalizable parameters and attributes) has an impact on functionalities and protocols to be implemented by the controller. For example, at the south-bound interface the controller needs to implement protocol extensions depending on the specific network layer of the controlled elements. Moreover, an inter-layer coordination function is also needed to deal with end-to-end connections and associated inter-layer technology adaptation, increasing the complexity of such unique controller.

To address such shortcomings, a current trend within SDN control plane design is to consider the deployment of multiple controllers, arranged in a specific setting, along with inter-controller protocols. Such architectures apply both to heterogeneous and homogeneous control (different or same control plane and data plane technologies within the domain of responsibility of a given controller). As detailed next, a straightforward scheme to arrange the controllers is either in a flat (peer) or hierarchical setting, but we will later qualify and challenge such simple model.

It is important to state that nothing precludes the deployment of two or more SDN controllers for a given set of controlled network elements. For example, the OpenFlow protocol supports the notion of primary or main and backup or secondary controllers. Two or more controllers can cover the same, overlapping or disjoint sets of network elements (this is a straightforward deployment choice for high availability reasons, where usually only a controller, e.g. the master one, has control over a given resource at a given time). For simplicity, from now on we assume a single SDN controller covering or spanning a set of controlled network elements. Having two or more controllers (e.g. for redundancy purposes) adds additional considerations such as inter-controller synchronization, and whether such controllers are synchronized e.g. using a dedicated protocol between them or via/by virtue of obtaining the information from the same set of network elements.

Regarding the management of platforms including resources that go beyond the network (i.e., the compute and storage resources), as is the case of 5G-Crosshaul, the deployment model of choice to handle scalability issues or multi-domain is unclear.

A potential deployment is a flat model on a per-domain basis, where a single MANO instance is responsible for both managing the network slices (i.e., a logical isolated network originated from the partition of a physical network) and the services running on top of these network slices for each domain. An open challenge in this kind of deployment model is how to orchestrate the provisioning of services that require a multi-domain SFC. We define a multi-domain SFC as a complex SFC that can be divided into smaller SFCs and thereby instantiated into different domains controlled by different MANO instances.

An alternative model is a recursive MANO deployment model. In this model, there is one MANO instance directly interacting with the physical network, compute, and storage substrate underneath and managing the different network slices. These network slices are exposed to additional MANO instances. The additional MANO instances orchestrate the services running on top of their particular owned slice. The degree of control and visibility that each of the additional MANO instances has over their slice (full or partial) is an open question.

In the following, we focus our discussion mainly on deployment and interconnection models of SDN controllers, since the network component is the main focus point when investigating deployment models in the context of 5G-Crosshaul.

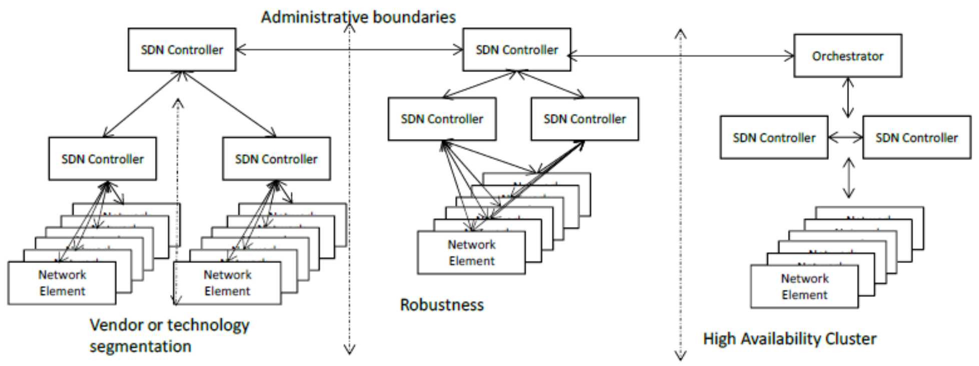


Figure 22: Common SDN controller arrangements, combining hierarchical and peer models.

3.3.1 Basic SDN controller interconnection models

In a basic approach, SDN controllers can be arranged in two canonical models: a peer or flat model and a hierarchical model (Figure 22).

3.3.1.1 Peer or flat model

This model corresponds to a set of controllers, interconnected in an arbitrary mesh, which cooperate to provide end-to-end services. In this setting, we can often assume that the mesh is implicit by the actual (sub)domains connectivity. The controllers hide the internal control technology and synchronize state using e.g. East/West interfaces. Further, the controllers manage detailed information of their own, local topology and connection databases, as well as abstracted views of the external domains and the East/West interfaces should support functions such as network topology abstraction, control adaptation, path computation and segment provisioning.

3.3.1.2 Hierarchical model

In this model, controllers are arranged in a tree-like topology, with a given root being the top-most controller. For a hierarchy level, a centralized “controller of controllers” or orchestrator (referred to as parent controller) handles the automation and has a number of high-level functions, while low-level controllers (referred to as children) cover low-level, detailed functions and operations. A recurring example is a 2-level hierarchy in which a parent SDN controller is responsible for connectivity provisioning at a higher, abstracted level, covering inter-domain aspects, while specific per-domain (child) controllers map the abstracted control plane functions into the underlying control plane technology. Proper interfaces and protocols are needed to enable this interaction between child and parent controllers; more generic interfaces and protocols enable a wider applicability of the architecture to an arbitrary number of hierarchy levels.

3.3.2 Generalizing hierarchical SDN controller interconnection models

In technological contexts such as the one defined by 5G-Crosshaul, several considerations challenge the simplistic hierarchical models. The following is a non-exhaustive list, noting that the challenges are also strongly inter-dependent, for example, different network segments may belong to one or more administrative domains/operators who internally arrange the network in technological domains, which, in turn, are commonly provided by different vendors.

- **Network segment splitting.** Defining an SDN control architecture for a network that encompasses multiple network segments (such as access, aggregation, metro, core) may be constrained regarding the feasibility to deploy a hierarchical SDN control or not.
- **Vendor constraints.** Arranging controllers in a specific setting depends on the available interfaces and protocols (ideally open and standard) and the corresponding level of support/implementation for a given vendor. It is reasonable to expect that arranging SDN controllers from the same vendor in a hierarchy will be straightforward if the vendor provides such functionality and there will be a high level of expected inter-operability in that case.
- **Redundancy, high availability, robustness.** It is accepted that deployments of SDN control in carriers' networks will be strongly constrained by the expected requirements regarding robustness and high-availability. Best common practices consider deploying multiple elements and synchronizing state between them. This is sometimes referred to as fat-trees, or forests, e.g. where the parents communicate with each other.
- **Widest definition and scope of "domain".** Strongly related to the previous ones, the term domain has multiple definitions and sometimes applies to the arrangement of network elements by their technology but also by their vendor or network segment, or even administrative or geographical domains.
- **Fitness for purpose.** A specific controller arrangement may not fully correspond to the intended logical SDN controller relationships, which sometimes can be better mapped to a client/server or master/slave model. While master/slave can correspond to a hierarchical setting, other relationships, functional splits, and responsibilities may fit better to a flat model.
- **Administrative domains, control, and ownership.** An often-recurring critique of pure hierarchical models comes from the issue of top-most parent ownership. Unless there are clear function definition and demarcation points, business arrangements and inter-connection models are based on a peer relationship in which no entity is under the control or supervision of a higher-level entity.
- **Confidentiality.** This applies to either peer or hierarchical models, although depends on the specifics of the northbound and west/east interfaces.
- **Domains of applicability.** The initial designs for hierarchical models addressed the problem of arranging SDN controllers considering only e.g. the networking and data communications for NS provisioning aspects. When considering, as, in 5G-Crosshaul, the need to offer 5G services that involve heterogeneous resources beyond the network (i.e. also storage and computing resources), the adoption of a hierarchical, peer or hybrid model is not clear. A set of network or cloud controllers may be under the control of an ETSI/NFV VIM, or the VIM may include a cloud controller that includes a network controller, and combinations hereof.
- **Provisioning workflows.** Intended provisioning workflows may also affect the choice of hierarchical or peer models. For example, "end user driven", "data driven" or "event-driven" provisioning services may be better suited to a peer model (e.g. a request from the RAN to the core) while a "operator-driven" pre-provision action may be better suited to a hierarchical model.

A main, direct consequence of the previous consideration and analysis, is that, in general, a given deployment of a carrier class SDN-based control architecture for 5G services (which combine heterogeneous networking/cloud resources over an infrastructure spanning multiple network segments and/or technological domains and vendors) will not correspond to a pure hierarchical or flat but will present a combination of centralized/distributed and hierarchical/flat/peer models constrained by the identified requirements and actual implementation choices (see Figure 23*).*

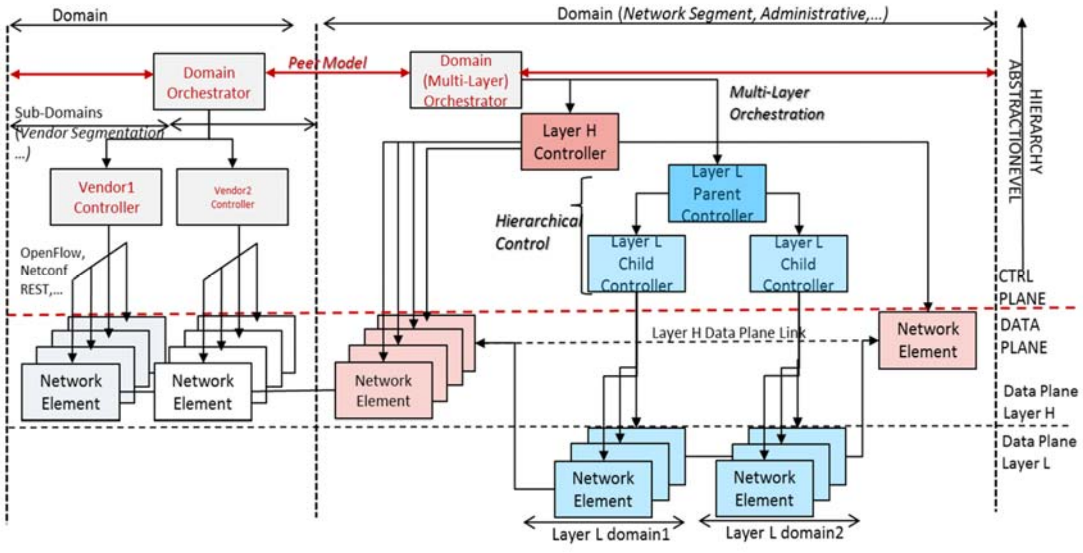


Figure 23: Example of SDN control within a mixture of administrative, technological and vendor domains, showing different peer and hierarchical modes.

3.3.2.1 Hierarchical SDN approaches based on the definition of domain

As a summary of the previous section, constraining aspects that may condition the deployment of hierarchical SDN controllers are mainly defined by the widest definition of domain and the associated requirements of confidentiality, functional split and inter-connection and business agreements. The following table summarizes and illustrates the main cases.

Table 8: Main use cases for hierarchical SDN controllers

Case examples **Deployments**	**Remarks in Hierarchical SDN architectures.**
Same vendor within a given technology and administrative domain	• Best-fit model depends on vendor defined criterion; e.g. peer or hierarchy and abstraction support provided by the SDN vendor. • Straightforward and interoperable hierarchical SDN setting. • Straightforward and interoperable peer SDN setting. • Hierarchy introduced for scalability reasons. • Within a hierarchical setting, peer-models may be used

	for robustness, or high-availability reasons (e.g. backup controller, distributed systems acting as a logically centralized entity). • Peer models commonly used when adhering to a distributed approach (e.g. controller mesh, redundancy or high availability solutions). • Hierarchical or peer architectures work at low-level interfaces with binary encodings and byte-level protocol. High degree of interworking, when applicable, proprietary extensions used. • Hybrid approaches complementing peer/hierarchical.
Different vendors within a single technology and administrative domain	• Peer models available (e.g. GMPLS controllers) but with strict constraints in interoperability. Issues in real deployments where operators chose to scope and segment into vendor islands. • Hierarchy as a means to orchestrate multiple vendors, scope inter-operability to a limited subset of interfaces and protocols, minimize risks. • Hierarchy done by means of "plugins" constrained to what is available or offered by vendors NBIs. Adopt an "NBI" standard if available and agreed upon.
Multiple network technologies within single administrative domains	• Peer models very constrained, requiring complex implementations and frameworks (e.g. GMPLS Multi-layer and Multi-region networks). • Peer model not adapted to a market where vendors cover mostly a horizontal technology or network segment. • Suitable hierarchical models in which an "orchestrator" coordinates topology management and service provisioning (e.g. IP over optical). • Hierarchy done by means of "plugins" constrained to what is available or offered by vendors NBIs. Adopt an "NBI" standard if available and agreed upon. The NBI is less straightforward since it needs to cover multiple technologies applicability.
Heterogeneous technologies and resources within single administrative domains	• High-level orchestration of e.g. cloud / storage / network controllers based on high-level requirements and systems behaviour. • Ad-hoc developments.
Different administrative	• Peer models adopted due to business and peering agreements, trust models.

domains	• Confidentiality and security issues. • Issues of Ownership and subordination. • *Forests models*. Commonly abstracts hierarchy within the administrative domain.
Wider scope infrastructures spanning multiple domains and network segments	• Hybrid approaches combining centralized, distributed elements and architectures. • Hybrid SDN-based architectures combining hierarchy and peer models, depending on inter-operability requirements, orchestration models and feasible choices. • Heavy use of abstraction and aggregation in hierarchies. • Instances of hierarchical SDN architectures for low-level interfaces (e.g. within vendor islands) and instances of hierarchical SDN architectures for high-level orchestration. • Constrained by inter-connection agreements between providers and operators. • Peer models for "data-triggered" or "event-triggered" provisioning across multiple segments.

3.3.2.2 Hierarchical SDN approaches based on API classes

It is also possible to use API classes as a criterion to identify potential SDN hierarchical architectures. For example, at a given level, a hierarchical relationship may be based on a high-level API and framework, relying on e.g. Intent based operation, control or orchestration. On the other hand, another hierarchical relationship may apply at a low-level interface, in which, macroscopically, the operation of children and parent (or sibling controllers) is fundamentally similar and the portioning is motivated by scalability, confidentiality and robustness reasons.

Table 9: Hierarchical SDN Controllers classified by APIs

Uses	**Remarks**
Low-level APIs	• Often tied to a specific low-level byte protocol. • Difficult interoperability, often only reasonable if within the same SDN controller vendor. • Strongly dependent on the hierarchy support provided by the SDN controller vendor. • Implemented to scale by combining homogeneous small-size systems into bigger ones in "stages". • Theoretical support for a constrained or arbitrary number of hierarchy levels. • ~ *"By design"*.

High-level APIs	• Commonly associated to high-level operations using high-level frameworks and constructs (e.g. REST). • APIs "exported" by the SDN controllers and "consumed"/"used" by orchestration systems. It does not preclude the use of a common, standard protocol within an implementation agreement or standard. • Often relies on implementing dedicated plugins at the orchestration (parent) entity, drives the specification of (standard) implementation agreements for APIs. • Number of hierarchy levels limited (e.g. two in most common deployments). • Quite adapted to common uses of orchestration of heterogeneous systems or vendors. • ~ *"By agreement"*.

3.3.2.3 Integration of Child and Parent Controllers

Based on the theoretical considerations of the previous sections on the challenges of a hierarchical SDN controller deployment, this section presents a practical deployment of the 5G-Crosshaul model for hierarchical control orchestration. Such model aims at achieving the following functionalities:

Centralized network orchestration: A logically centralized entity exists on top of and across the different network domains and is able to drive the provisioning (and recovery) of connectivity across heterogeneous transport networks, dynamically, and in real time.

Technology abstraction: The introduction of a new interface and protocol that abstracts the particular control plane technology of a given domain. In this sense, the proposed architecture applies the same abstraction and generalization principles that OpenFlow/SDN have applied to data networks

With these goals in mind, we have used the COP[3] to interconnect per-technology domain child controllers to a technology agnostic centralized orchestrator (see Figure 24), referred to as the parent SDN controller, which corresponds to the High-Level API approach described in section 3.3.2.2. COP is used for the interaction of the COP client plugin installed in the parent SDN controller with the COP server installed in each of the per-technology child SDN controller. However, the API between the per-technology child SDN controllers and the COP server requires a customized plugin dependent on the NBI API offered by each of the child SDN controllers. In Figure 24 there are three child SDN controllers: Wireless SDN Controller A, Optical SDN controller, and Wireless SDN Controller B. The specific plugin in each of the per-technology child SDN controllers translates the REST-based request received by the COP server to the specific per-technology SDN controller NBI API implemented by the SDN controller.

[3] https://github.com/5G-Crosshaul

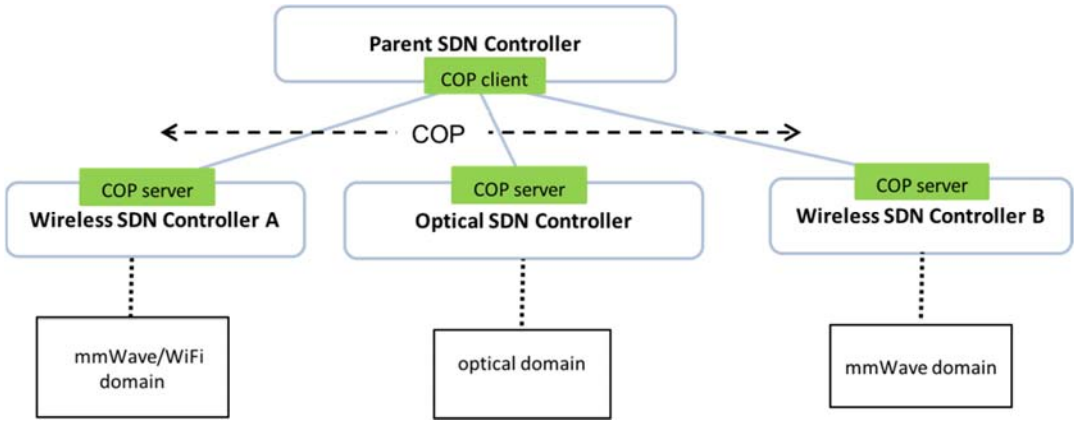


Figure 24: The hierarchical 5G-Crosshaul model: Integration of per-technology child SDN controllers with a centralized parent SDN orchestrator.

The parent SDN controller assumes that each domain is composed of a data plane controlled by an instance of a specific control plane technology for the transport network, but transport and/or control plane technologies for each domain can be different. The main functionalities of the SDN parent controller are network abstraction so that the resulting global network view is not technology related. Thus, this control plane abstraction must enable the provisioning of NSs using the underlying configuration technology. In the following, we provide a description of the services that COP provides for the interaction amongst parent and child SDN controllers.

Topology Services: Upon request through the COP client, the parent SDN controller can retrieve a child SDN network topology. This service is aligned with the specification in Section 2.1. The COP definition covers the topological information about the network, which must include a common and homogeneous definition of the network topologies (based on the concept of nodes and edges). Note that the definition of a topology may be flexible enough to include the different capabilities of each per-technology transport network domain.

Path Computation Services: The path computation service provides an interface to request (by the COP client in the parent SDN controller) and return Path objects (by the COP server in the child SDN controller) which contain the information about the route between two service endpoints. Path computation is highly related to the call services. Note that in the call object, the connection object has been designed to have the possibility of containing explicit information about the flow match/action rules along the traversed path. The path model should be the same in both, the service Call and at the service Path Computation.

Call Services: Using Call objects, the parent SDN controller can request the provisoning of end-to-end connectivity services across multiple transport domains. A Call object must describe the type of end-to-end connection service to be requested or served (e.g., Ethernet, MPLS). The Call object is formed by a list of connection objects, including the service endpoints. A connection object is used for a single per-technology transport network node domain. It includes the path or route across the transport network traffic traverses. By design, these routes may be fully described (explicit) or abstracted (delegated to the child SDN controller) depending on the orchestration/control schemes used amongst parent and child SDN controller. In the specific implementation, we have opted for detailing the explicit route from the parent to the child SDN controller, hence assuming a high degree of control by the parent SDN controller. These routes in particular include the corresponding matching rules and actions. The abstracted route description would be more appropriate in deployments with a substantial bigger number of domains and more complex inter-domain connections, i.e, mesh, to trade-off route determination complexity between parent and child SDN controllers.

On the other hand, it is important to note that each connection must be associated with a single control plane entity (e.g. a per-technology child SDN controller) responsible for the configuration of the data path. Note that this also requires an interface between the COP client and COP server to request the computation of a path in a single technological domain.

From a high-level view, in the following we show a simple example detailing the workflow between the COP client, COP server, and the child SDN controller to serve a call request attempting to install flow entries in their forwarding switches.

1. The COP client sends a call request to the COP server with call *callid*, to establish a connection between two endpoints in a technological transport domain.
2. The COP server conducts these two tasks:
    a. The COP server saves the *callid* object and starts iterating over all the connections that compose this call *callid*.
    b. The COP server processes these connections, saves their state, and prepares the info regarding each of the connection to be sent to the child SDN controller in a way compliant to the API offered by the child SDN controller (the API offered by Ryu, ODL, etc).
3. The child SDN controller conducts these two tasks:
    a. The child SDN controller receives all the information encoded in a connection, i.e. the match policy and action rules action for a given connection.
    b. The child SDN controller installs the match and action rules in the corresponding data plane element.

COP is based on REST HTTP-based technology. The REST paradigm is convenient for the COP implementation due to the need of stateless communication among SDN controllers and the SDN orchestrator. It is also convenient because of the flexibility, scalability and commodity for practical implementation. The objects defined by COP to provide useful orchestration mechanisms between parent and child SDN controller in a wide range of multi-domain network orchestration are defined using the YANG data modelling language.

In the proof of concept, the parent SDN controller jointly with the mmWave/WiFi SDN controller A and optical controller is located in one testbed site whereas the mmWave SDN controller B is located in a different testbed site. To interconnect both testbed sites, we have established different OpenVPN tunnels for both data and control plane networks. Details on the setup and inter-site performance measurements amongst the two sites will be found in D5.2 [20] (i.e., around 43ms of RTT and TCP performance of around 70Mbps in average). Thanks to the COP protocol and the established Virtual Private Network (VPN) tunnels, the parent SDN controller is able to establish end-to-end connections between different sites to make the data to flow between any pair of endpoints present in the deployment of Figure 24.

4 Control Plane Algorithms

In this section, we present models for control plane algorithms, Firstly, the power consumption of two different virtualization technologies, namely, Docker containers and VMs is investigated. In particular, a power consumption model is presented for each of them, leveraging the results of our experiments [21][22]. Secondly, a model on optimizing the forwarding behavior of networks is presented. Both models have been implemented, the first one as part of energy management algorithms, the second one within Matlab.

4.1 Power Consumption Model of Docker Containers

Containerization has recently emerged as a lightweight alternative to VMs. Applications run within containers, which share the same OS, kernel, and process space as the host machine, while keeping file systems isolated. File systems contain whatever applications need to run: code, runtime, and system libraries. Containers offer a lower level of isolation than VMs, and are in general a less mature technology. On the positive side, they imply only a fraction of the overhead associated with VMs: they can start (almost) instantly, and use a negligible amount of additional memory.

Docker is the current de facto standard for containerization, and the first commercially successful solution. A Docker daemon, equivalent to a VM hypervisor, runs several containers, each executing a different application on a separate file system. It is worth noticing that the OS operating system kernel is shared among containers (i.e., all applications run on the same OS and OS version as the host), and that data has to traverse the kernel network stack when traveling from a container to another.

4.1.1 Testbed and Applications

In the experimental tests, we used a common desktop, namely, HP Compac 8000 Elite CMT business PC with a dual-core 2.66 GHz processor and 2 GBytes of RAM. It runs the Ubuntu 14.04.5 LTS OS, with kernel 4.4.0-31-generic. Both the processor and the OS support hyper-threading, which raises the number of tasks that can be run at the same time to four. RCE PM600 power meter is used to read real-time power statistics. All tests are run for 2 minutes, and power samples are read from the power meter.

CPU test: The first test is a CPU-intensive task, continuously performing matrix products with the numpy library. Because numpy is single threaded, this application will employ at most one CPU core, even if more are available.

Network test: We consider a typical network transfer scenario: a client container uses *iperf* to send an uninterrupted, constant-rate flow of TCP data to a server container; neither container performs any other operation. The purpose of this test is to assess whether the data rate impacts the power consumption, and how.

Network test with multiple client containers: In this test, we activated different number of containers to send constant-rate flow of TCP data using *iperf*. The data rate is kept constant, and it is divided among the client containers. The test highlights the impact of varying the number of containers on the CPU utilization and power consumption.

4.1.2 Experimental results and power consumption model

Power consumption vs. CPU utilization. Figure 25 summarizes the results of the CPU test. Each data point corresponds to a different number of containers running the test: when there are up to three containers running, each consumes almost exactly 100% of the available CPU, i.e., one core out of four. The rightmost point, corresponding to the test with four containers, shows a CPU load of around 370%. The remaining 30% is used by overheads, including that of the Docker daemon and of the OS processes. The linear fitting is very good, and the linear regression law is as follows:

$$P_{CPU} = 0.1065z + 12.4411$$

where P_{CPU} is the power (in Watt) and z is the percentage of used CPU (e.g., 200 for two cores).

Power consumption vs. data rate. Figure 26 presents the results of the network test; specifically, the plot displays the power consumption as the data transfer rate varies. The dependence is more complicated than in the case of the CPU test. The best model is a second order polynomial fitting:

$$P_{data} = -17.7368d^2 + 37.2859d + 3.8210$$

where the power consumption is expressed in Watt, as measured by power meter and d is the data rate in Gbits/s.

This behavior can be explained by looking, in Figure 27, at how the CPU load changes with the data rate. We can clearly see an almost-linear growth until a rate of 0.6 Gbit/s, and then a basically constant CPU usage after 0.8 Gbit/s. This is due to the fact that, between 0.6 and 0.8 Gbit/s, we hit the maximum rate at which data transfers can occur: we can set a rate of, say 1 Gbit/s in *iperf*, but only a fraction of that data gets to the destination.

Power Consumption vs. number of containers: Figure 28 presents the results of the network test when multiple client containers are sending data to the server container. The plot displays the power consumption as the number of clients varies. In this test, the total data rate is kept constant to 600Mb/s. We can see the power consumption is having limited variation as the number of client containers change. The best model for this case is logarithmic fitting:

$$P = 19.1632 + 5.3975 log_{10} n$$

where the power consumption is expressed in Watt, as measured by power meter and n is the number of containers.

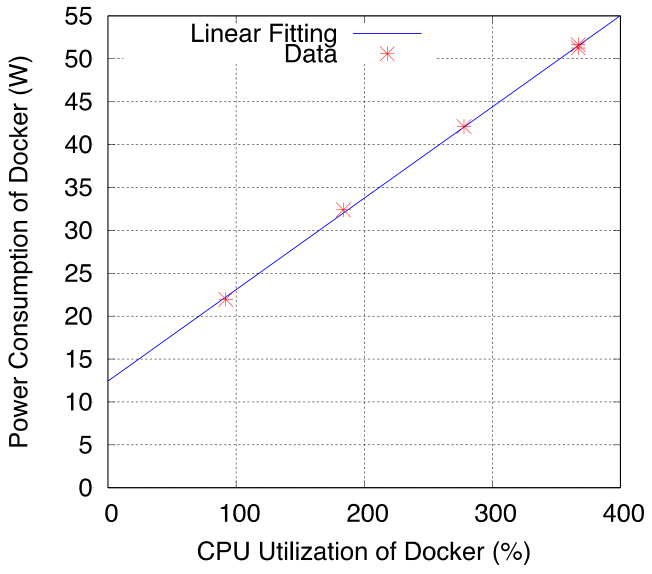


Figure 25: CPU test: Power consumption vs. CPU utilization

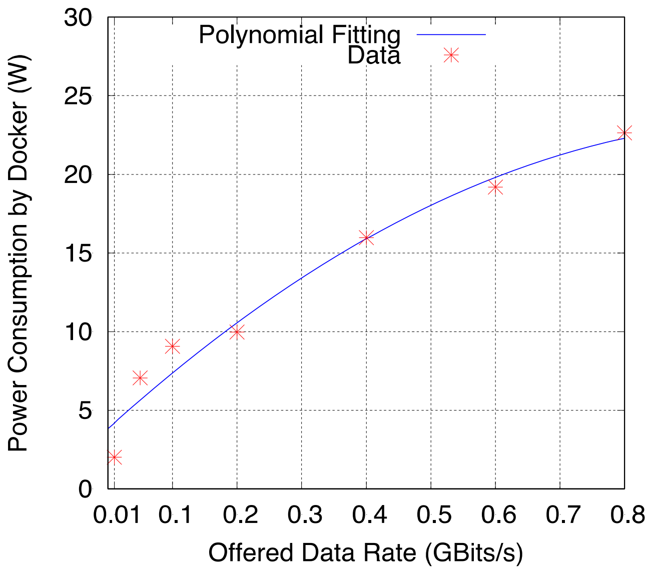


Figure 26: Network test: Power consumption vs. data rate

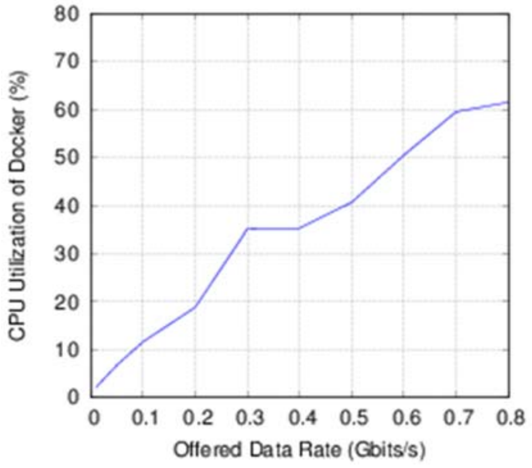


Figure 27: Network Test: CPU utilization vs. data rate

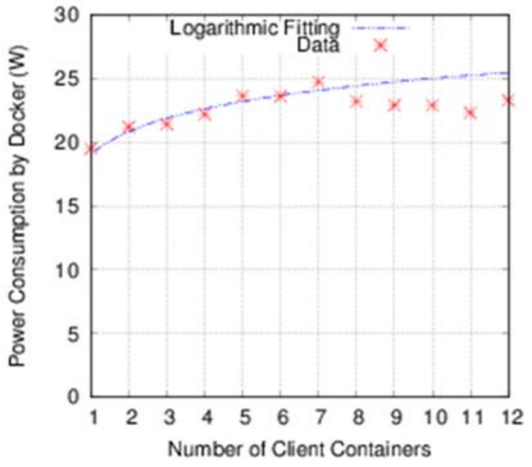


Figure 28: Network test: Power consumption vs. number of containers

4.1.3 Comparing Docker Containers and VMs

In order to compare Docker containers to VMs, we used a laptop, namely, HP EliteBook 820 G3 with a dual-core 2.3 GHz processor and 8 GBytes of RAM. It runs the Ubuntu 16.04.5 LTS OS, with kernel 4.8.0-46-generic. Both the processor and the OS support hyper-threading, which increases the number of tasks that can be run at the same time to four. VMs run in Virtualbox version 5.0.32. We remark that using a laptop, instead of a desktop, implies that the power consumption measured through the power meter has the same behavior as shown in the previous section, however the value of the coefficients varies. For sake of completeness, we report below also the models of power consumption fitting the experiments run through a laptop.

Overhead test: This test involves Docker containers and VMs (from 1 to 4) executing an application that does not consume any memory or computing resource, thus the CPU consumption observed during the test is only due to the virtualization approach overhead. Figure 29 highlights the advantage of containers over VMs, in terms of power consumption due to the Docker containers and VMs overhead, as a function of the number of deployed entities: the Docker power consumption is much lower than that of VMs and becomes negligible as the number of running entities increases.

CPU test: The CPU test is done with the CPU-intensive task, performing matrix products with the numpy library which employs at most one CPU at a time. In this test, we varied the number of containers and the number of VMs till the available capacity is saturated. Figure 30 shows that Docker containers and VMs equally perform under CPU intensive applications. Below we report the linear fitting for Docker containers:

$$P = 0.0166z + 5.0444$$

and VMs:

$$P = 0.0175z + 4.9091$$

where *P* is the consumed power (in watt) and *z i*s the CPU utilization in percentage.

Network test: In this test, the *iperf* application runs as a server in one Docker container and as a client in another Docker container. A constant rate TCP data flow is sent from the client container to the server container. The test results show the resource utilization, hence the power consumption, when VMs and Docker containers are used to implement applications involving network transfers. Figure 31 depicts the power consumption due to data transfer versus the rate of the data generated by the iperf application. In this case, the coefficients of the second order polynomial model of power consumption turned out to be, for Docker containers:

$$P = -1.6605r^2 + 3.7237r + 0.5822$$

and for VMs:

$$P = -12.8815r^2 + 22.2558r + 3.2318$$

where *P* is the consumed power (in watt) and *r* is the data rate in Gbits/s.

Memory Test: The test makes use of a java application consuming mainly memory resources. Again, the application is run in Docker containers and in VMs in order to highlight the different power consumption. As shown by Figure 32, varying the memory utilization has negligible effect on the power consumption in both Docker containers and VMs. However, due to their high overhead (see Figure 29), VMs imply a much larger power consumption than Docker containers. In this case, the power consumption (in watt) is simply given by *P*=0.31 and *P*=1.1 for Docker and VMs, respectively.

Disk IO Test: This test is used to investigate the power consumption and the latency of disk io operations. *Flexible IO (fio)* command line tool is used to perform a random reading task of different block sizes. Figure 33 specifically depicts the relationship between the block size in random reading of 10 GB file and the associated power consumption, which is essentially the same for both virtualization approaches, except for the values observed at the lower block sizes.

For the disk IO test, the power consumption exhibits an exponential fitting as the block size varies, for both Docker containers (top equation) and VMs (bottom equation):

$$P = 4.806exp(-0.1014x) + 2.848exp(0.0007232x)$$

$$P = 1.857exp(0.0009083x)$$

where *P* is the consumed power (in watt) and *x* is the block size per time unit in KBytes.

Additionally, Figure 34 shows the difference in the time to complete the disk IO task as the block size changes. The completion time in Docker is considerably smaller than in VMs; for both virtualization approaches, however, the observed behavior for the latency is second order decaying exponential. In particular, for Docker (top equation) and VMs (bottom equation), we have:

$$L = 209.3exp(-1.076x) + 5.151exp(-0.015x)$$

$$L = 7.174exp(-0.07104x) + 4.496exp(-0.0073x)$$

where *L* is the latency (in minutes) and *x* is the block size in KBytes.

Finally, Table 10 presents the level of accuracy of the models obtained for different tests. To determine the models' accuracy, a cross-validation method is applied, i.e., we have used points that have not be exploited for the fitting process. The average percentage error is determined by evaluating the percentage absolute difference between the value obtained from the model and the corresponding experimental result. From the table, we observe that the CPU model is extremely accurate, and in most of the cases we have an accuracy of about 10%.

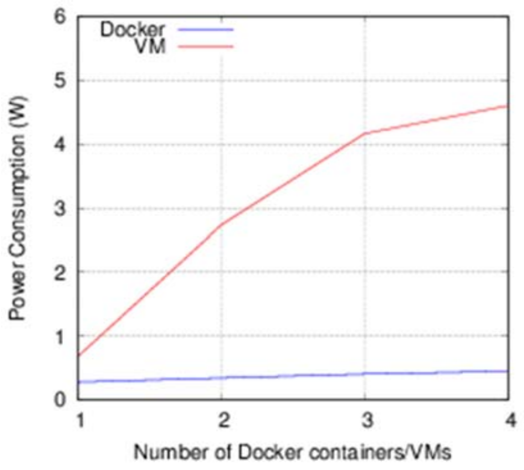


Figure 29: Overhead test: power consumption vs. number of containers/VMs

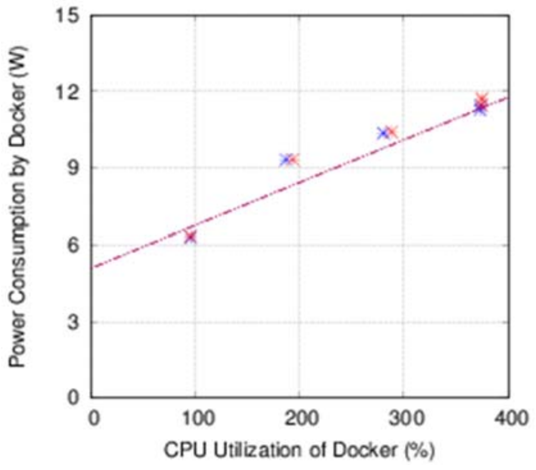


Figure 30: CPU test: power consumption vs. number of containers/VMs

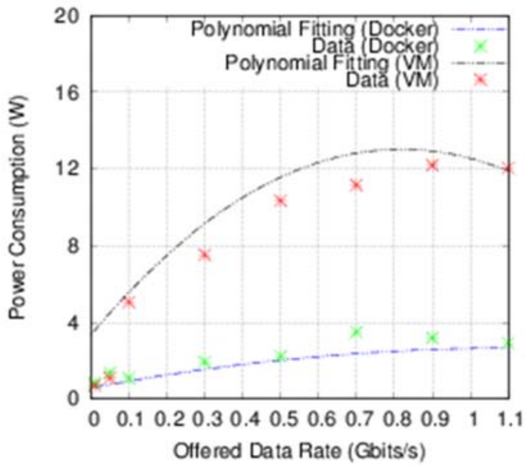


Figure 31: Network test: power consumption vs. data rate

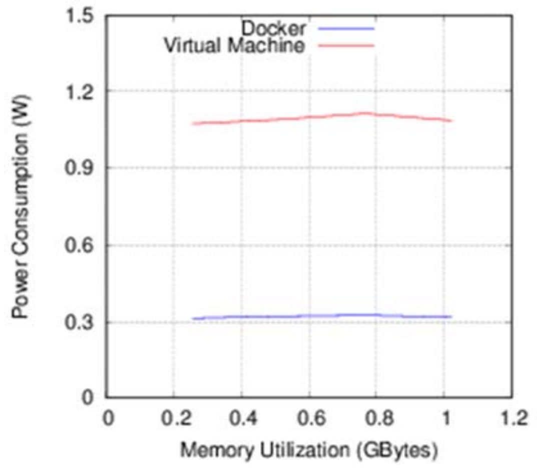


Figure 32: Memory Test: power consumption vs. memory utilization

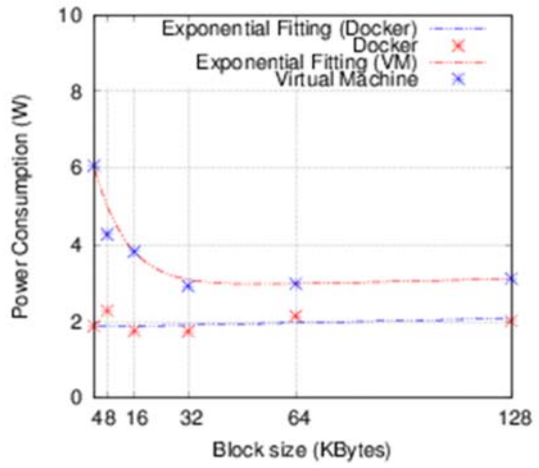


Figure 33: Disk IO test for 10 GB file size: power consumption vs. block size

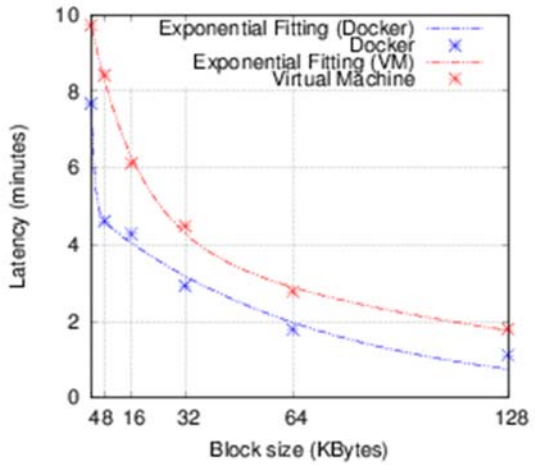


Figure 34: Disk IO test for 10 GB file size: latency vs. block size

Table 10: Models accuracy

Test	Figure	Average Validation Error (%)	Training Error
CPU test	Figure 25	0.23	0.4354 (std dev)
Network test - single client	Figure 26	8.89	2.4124 (std dev)
Network test - multiple clients	Figure 29	3.42	1.419 (std dev)
VM vs. Docker - Network test	Figure 31	23.87 - Docker	0.2746 (std dev)
		8.78 - VM	2.3699 (std dev)
VM vs. Docker - Disk-IO test	Figure 33	13.9 - Docker	0.3624 (std dev)
		11.41 – VM	0.3257 (std Dev)

4.2 Network Optimization Model

This section introduces an algorithm to optimize the forwarding behavior of a 5G-Crosshaul network comprised of multiple backhaul and fronthaul flows. Fronthaul traffic is constrained to pass through an XPU first, processing Base Station (BS) raw data, and it is then forwarded towards the core network. Backhaul (BH) traffic does not necessarily have to be relayed by an XPU before being forwarded toward the Internet. This model is an extension of the one described in D3.1 [1], using the same scenario as an example.

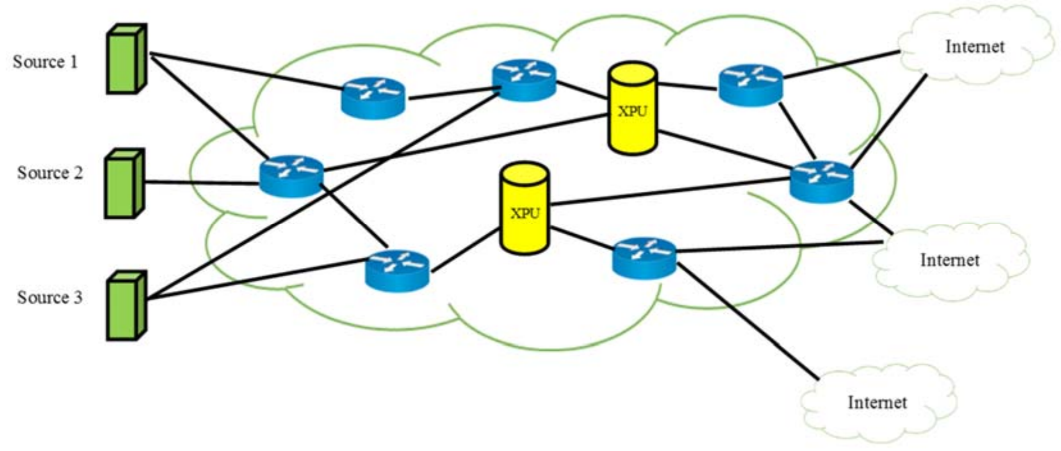


Figure 35: General scenario to optimize

We consider a scenario like the one shown in Figure 35. This scenario contains source nodes generating backhaul traffic (eNBs), source nodes generating fronthaul traffic (RUs), intermediate nodes that simply forward traffic, XPUs that contain Central Unit (CU) functionality to process fronthaul traffic, Internet gateways, and network links with specific capacities connecting different nodes. As mentioned, fronthaul traffic must pass through at least one XPU providing CU functionality, while backhaul traffic has no such constraint. We assume that an XPU may provide multiple CU functions (up to a defined maximum), and links can use any layer-2 forwarding technology, providing a specific capacity on this link. To avoid an overly complex model, we also assume that each CU can process at most one fronthaul flow.

The algorithm presented in [1] set the paths or routes from sources (generating either fronthaul or backhaul flows) to one gateway to the Internet, having defined the type of sources a priori. The extension of the algorithm presented here, instead, addresses the previous problem of defining the optimal routes for the traffic, but now the sources are not defined a priori. The algorithm indeed determines which of the sources will produce backhaul traffic (eNBs) and in which ones it will be better to place a RU and separate the CU function to place it in the appropriate XPU place inside the network. This XPU is shared among other RUs to increase the performance of all of them. If the XPU is not shared, there is no such gain in the performance, thus, it is more expensive to place an RU and an XPU only for it than placing an eNB (backhaul source). In summary, the algorithm determines the appropriate number and place of RUs and eNBs, and includes the placement of the XPUs that are necessary for the RUs that have been deployed. Then it determines the optimal path that the traffic from the sources (backhaul or fronthaul flows depending on the type of source) must follow to reach the corresponding XPU and after that, the destination, which is an Internet gateway.

It is important to highlight that to place the resources and determine the traffic routes, we take into account the propagation delay of each type of traffic and the related delay constraints. The goal of the problem is to determine the optimal place and number of RUs, eNBs, XPUs and the paths to be followed by the traffic flows so that the number of RRHs that share an XPU is maximized while the number of XPUs is minimized.

Problem formulation: The problem includes binary and real variables, yielding a Mixed Integer Programming (MIP) problem.

Input parameters:

- f^l fronthaul flow from source l before using a CU
- f^l_{CU} fronthaul flow from source l after using a CU
- b^l_k backhaul flow k from source l
- c_{ij} maximum capacity for link (i, j)
- XPU_r maximum number of CUs XPU r can allocate
- D_{f^l} delay constraint for fronthaul flow l before using the XPU
- $D_{f^l_{CU}}$ delay constraint for fronthaul flow l after using the XPU
- $D_{b^l_k}$ delay constraint for backhaul flow k from source l
- L_{ij} length of link (i, j)
- v_l light speed

Variables:

- β_l binary variable to determine if source l is an RU or not (if not, it is an eNB)
- $x_{ij\,f^l}$ binary variable to determine if (i, j) link is used for the f^l flow before using the XPU.
- $x_{ij\,f^l_{CU}}$ binary variable to determine if the (i, j) link is used for the f^l flow after using the XPU.
- $x_{ij\,b^l_k}$ binary variable to determine if the (i, j) link is used for the b^l_k flow.
- δ_r binary variable to determine if the r-th XPU is used or not.
- $z_{r\,f^l}$ binary variable to determine if the r-th XPU is used for the f^l flow.
- d_{f^l} delay for fronthaul flow l in the whole network
- $d_{f^l_{CU}}$ delay for fronthaul flow l before using the XPU
- $d_{b^l_k}$ delay for backhaul flow k from source l

Objective function: the goal of the problem is to maximize the performance of the sources, thus if most of the sources can be RUs and share the XPU its traffic uses, they obtain a gain in the objective function, because the costs will be lower since the XPU is shared. On the other side If one source cannot share the XPU, this source is better to be an eNB that does not need to use an XPU. These ideas are gathered in the objective function:

$$max \sum_r (gain \cdot (\sum_l z_{r\,f^l} - \delta_r) - \delta_r)$$

Constraints:

- to determine if source l is an RU (we consider the traffic leaves the source through a single path): $\beta_l = \sum_j x_{l\,j\,f^l} \quad \forall\, f^l_k$ flow

- to determine if source l is an eNB (we consider the traffic leaves the source through a single path): $1-\beta_l=\sum_j x_{l\,j\,b_k^l}\ \forall\, b_k^l$ flow
- to impose that flows in a link cannot exceed the capacity of the link:

$$\sum_l f^l\cdot x_{i\,j\,f^l}+\sum_l f_{CU}^l\cdot x_{ij\,f_{CU}^l}+\sum_{k,l} b_k^l\cdot x_{ij\,b_k^l}\ \le\ c_{ij}\qquad \forall\,(\mathrm{i,j})\ \text{link}$$

- to impose that all flows in the network reach the Internet (we denote the internet gateway r as Int_r):

$$\sum_{k,l}(b_k^l\cdot(1-\beta_l))\ +\sum_l(f^l\cdot\ \beta_l)\ =\sum_{r,i}(\sum_{k,l} b_k^l\cdot x_{i\,Int_r\,b_k^l}+\sum_l f^l\cdot x_{i\,Int_r\,f^l})$$

- to impose that all fronthaul flows in the network reach an XPU:

$$\sum_l \beta_l\ =\sum_r\sum_l z_{r\,f^l}$$

- the XPU is used by the fronthaul flow l if the traffic arrives by one of the incoming links:

$$z_{r\,f^l}\le\sum_i x_{i\,XPU_r\,f^l}\qquad \forall\ \text{XPU r},\ \forall\, f^l\ \text{flow}$$

- the XPU can accommodate at maximum a predefined number of flows:

$$\sum_i x_{i\,XPU_r\,f^l}\ \le\ XPU_r\qquad \forall\ \text{XPU r},\ \forall\, f^l\ \text{flow}$$

- to determine if a XPU is used or not:

$$z_{r\,f^l}\ \le\ \delta_r\qquad \forall\ \text{XPU r}$$

$$\delta_r\ \le\ \sum_l z_{i\,f^l}\qquad \forall\ \text{XPU r}$$

- a fronthaul flow that transits by a node without being processed by the XPU implemented therein, it maintains its status of fronthaul flow. A flow that instead reaches a node and is processed by the XPU implemented therein, it becomes a backhaul flow:

$$\sum_i x_{i\,XPU_r\,f^l}=\sum_i x_{XPU_r\,i\,f_{CU}^l}\cdot z_{i\,f^l}+\sum_i x_{XPU_r\,i\,f^l}\cdot(1-\ z_{i\,f^l})$$
$$\sum_i x_{i\,XPU_r\,f_{CU}^l}\ =\ \sum_i x_{XPU_r\,i\,f_{CU}^l}\cdot(1-\ z_{i\,f^l})$$

- at any intermediate node r, all incoming traffic must equal the outgoing traffic:

$$\sum_j x_{jr\,f^l}=\sum_i x_{ri\,f^l}\qquad \forall\ \text{node r},\ \forall\, f^l\ \text{flow}$$

$$\sum_j x_{jr\,f_{CU}^l}\ =\sum_i x_{ri\,f_{CU}^l}\qquad \forall\ \text{node r},\ \forall\, f_{CU}^l\ \text{flow}$$

$$\sum\nolimits_j x_{jr\,b_k^l} = \sum\nolimits_i x_{ri\,b_k^l} \qquad \forall \text{ node r}, \forall\ b_k^l \text{ flow}$$

- single path for each fronthaul flow before using the XPU:

$$\sum\nolimits_j x_{i\,j\,f^l} \le \beta_l \quad \forall \text{ node i}, \forall\ f^l \text{flow}$$

- single path for each fronthaul flow after using the XPU:

$$\sum\nolimits_j x_{i\,j\,f_{CU}^l} \le \beta_l \quad \forall \text{ node i}, \forall\ f_{CU}^l \text{flow}$$

- single path for each backhaul flow:

$$\sum\nolimits_j x_{ij\,b_k^l} \le 1 - \beta_l \quad \forall \text{ node i}, \forall\ b_k^l \text{ flow}$$

- delay constraints: we compute the propagation delay the packets suffer when using the links of the network and this delay has to be lower than the maximum established:

$$d_{f^l} = \sum\nolimits_{i\,j} \frac{L_{ij}}{v_l} \cdot\ x_{i\,j\,f^l} + \sum\nolimits_{i\,j} \frac{L_{ij}}{v_l} \cdot\ x_{i\,j\,f_{CU}^l} \ \le\ D_{f^l}$$

$$d_{f_{CU}^l} = \sum_{i\,j} \frac{L_{ij}}{v_l} \cdot\ x_{i\,j\,f^l} \ \le\ D_{f_{CU}^l}$$

$$d_{b_k^l} = \sum\nolimits_{i\,j} \frac{L_{ij}}{v_l} \cdot\ x_{i\,j\,b_k^l} \le\ D_{b_k^l}$$

- binary variables:

$\beta_l\ \in\ \{0,1\}$	$\forall\ l$ source
$x_{i\,j\,f^l}\ \in\ \{0,1\}$	∀ (i,j) link, ∀ f^l flow
$x_{ij\,f_{CU}^l}\ \in\ \{0,1\}$	∀ (i,j) link, ∀ f_{CU}^l flow
$x_{ij\,b_k^l}\ \in\ \{0,1\}$	∀ (i,j) link, ∀ b_k^l flow
$z_{r\ f^l}\ \in\ \{0,1\}$	∀ XPU r, ∀ f^l flow
$\delta_r\ \in\ \{0,1\}$	∀ XPU r

The two non-linear constraints in the model can be easily linearized by adding additional variables and changing the products by summations bounding the variables of interest above and below. Once these constraints are linearized we obtain a mixed integer linear programming problem, which can be solved by maximizing the objective function under the given constraints. The problem determines the number and placement of RUs and eNBs that can be accommodated in the network, while solving the placement of CUs to XPUs, and the route each flow from the sources has to follow to reach its destination. Our goal is to minimize the number of used resources while maximizing the initial bandwidth of the network prioritizing FH traffic over the backhaul traffic. This model takes in account the delay the flows experience to reach the resources since it is one essential requirement of the traffic we deal with, and even more important in the case of FH traffic.

This problem is NP-hard, it is not scalable as an optimization problem. For simulations, we have implemented the problem in Matlab and used and tested the model in anenvironment with a network of up to 31 nodes, 31 nodes require more than 19000 variables. Also, we performed simulations in a smaller network of 16 nodes with 8 sources of traffic changing the parameter XPU_r of the maximum number of CUs we can accommodate in each XPU. The results obtained are shown in Figure 36, where we can observe the variation of the number of XPUs required to satisfy the RUs traffic.

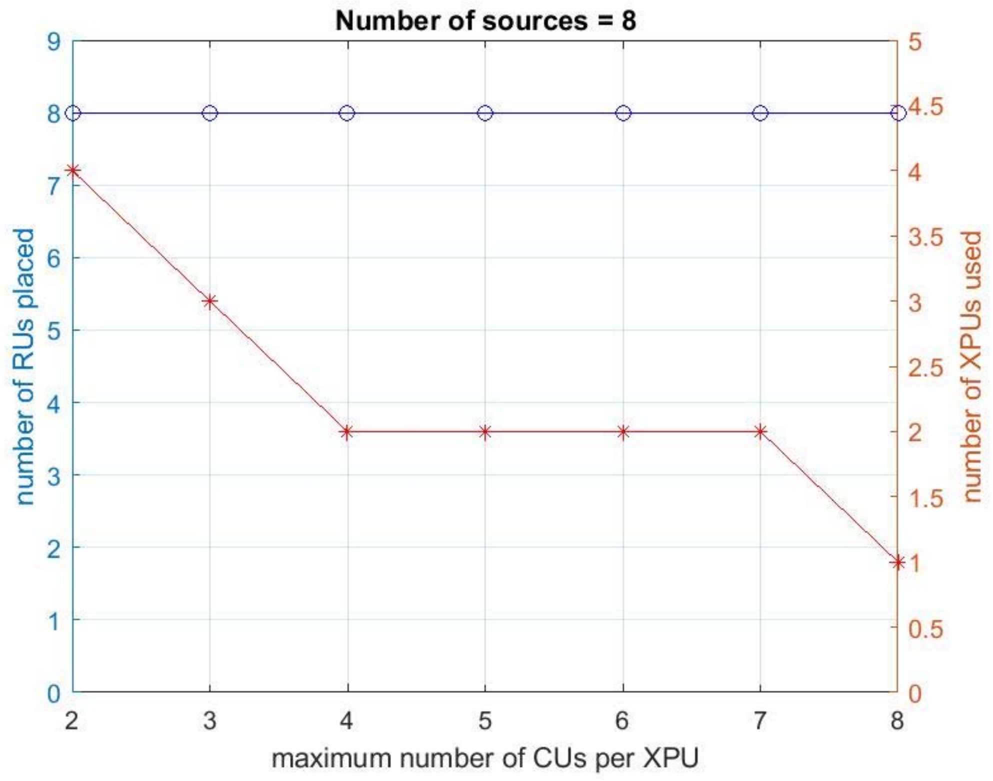


Figure 36: XPUs to satisfy RU traffic

5 Data plane architecture

In this section, we summarize the architectural framework for the data plane. Based on this description, we present further detail regarding QoS, Operations, Administration and Management (OAM), and synchronization.

5.1 Architectural Framework

The 5G-Crosshaul data plane architecture is illustrated in Figure 37.

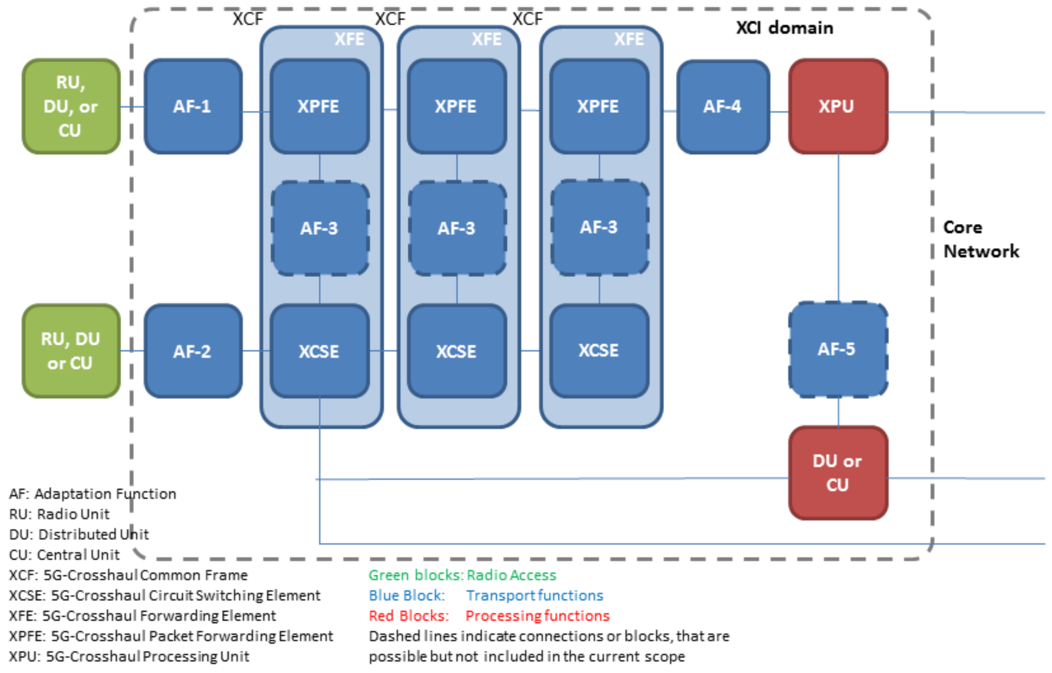

XCF
XCF
XCF
XCI domain
XFE
XFE
XFE
RU, DU, or CU
AF-1
XPFE
XPFE
XPFE
AF-4
XPU
AF-3
AF-3
AF-3
Core Network
RU, DU or CU
AF-2
XCSE
XCSE
XCSE
AF-5
DU or CU
AF: Adaptation Function
RU: Radio Unit
DU: Distributed Unit
CU: Central Unit
XCF: 5G-Crosshaul Common Frame
XCSE: 5G-Crosshaul Circuit Switching Element
XFE: 5G-Crosshaul Forwarding Element
XPFE: 5G-Crosshaul Packet Forwarding Element
XPU: 5G-Crosshaul Processing Unit
Green blocks: Radio Access
Blue Block: Transport functions
Red Blocks: Processing functions
Dashed lines indicate connections or blocks, that are possible but not included in the current scope

Figure 37: 5G-Crosshaul data plane architecture [5].

The fundamental building block of the data plane architecture is the XFE. The XFE is a multi-layer switch constituting a 5G-Crosshaul Packet Forwarding Element (XPFE) and a 5G-Crosshaul Circuit Switching Element (XCSE). The adaptation functions (AF-x) perform media adaptation and translation of various frame formats used by RUs, DUs, CUs, and XPUs into the 5G-Crosshaul common Frame (XCF) format. The XPFEs communicate with each other by exchanging frames according to XCF format. The frame format is transparent to the XCSEs. Section 7 of D2.1 [5] provides an in-depth description of the XFE.

5.2 XFE design

The overall design of the XFE as a multi-layer switch consisting of a packet- and a circuit-switching layer has not changed since D3.1 [1]. Additional detail is provided in this section, focusing on the packet switching layer.

5.2.1 Circuit Switching (XCSE)

The circuit-switching part, i.e. the XCSE consists of a classical circuit switch, i.e. an Optical Transport Network (OTN) switch, and a purely optical switch. For further detail see D2.1 [5], Section 7.

5.2.2 Packet Switching (XPFE) and the XCF

OpenFlow has been used as the SBI protocol to control the forwarding behavior the XPFEs. The XPFE provides a common switching layer for packet switched traffic. This common switching layer employs the XCF format to support: various traffic types (i.e. fronthaul variants and backhaul) and the various link technologies in the forwarding network [1]. The XCF is a frame format which defines the structure of the frames in the XPFE common switching layer. The corresponding control of the forwarding behavior of the XPFE is defined in the XCI in line with the SDN approach, where all control aspects are moved to a logically centralized controller.

The XCF has to support a large variety of services such as, IP based fronthaul variants arising from different functional splits [24] and the Long Term Evolution (LTE) BH traffic [23] that is dependent on the physical topology and different protocols of the backhaul network. The XCF should contain enough information in the frame headers to enable the XPFEs to fulfill their task as a common switching layer for both fronthaul and backhaul traffic. In addition to supporting backhaul traffic and different fronthaul functional splits, multiple tenants should also be supported as well. Table 43 in Section 13 provides a detailed list of requirements that the XCF should support including: OAM support, interaction with legacy devices, efficient use of available bandwidth, etc.

5G-Crosshaul adopted the MAC-in-MAC XCF, presented in subsection 5.2.2.1, as the baseline while the MPLS-Transport Profile (MPLS-TP) XCF, presented in subsection 5.2.2.2, has been considered as a suitable alternative. Within 5G-Crosshaul MAC-in-MAC was preferred to maintain alignment of developments among partners. From a technical perspective, no clear advantages or disadvantages of MAC-in-MAC or MPLS-TP as an XCF were identified.

5.2.2.1 MAC-in-MAC XCF as baseline

In order to ensure interoperability with legacy devices and to benefit from previous research, an existing packet format i.e. MAC-in-MAC was adopted as the XCF. This XCF provides sufficient information for the XFEs to forward the packets towards their destination while satisfying the requirements on latency and jitter.

Starting from the ubiquitous availability of Ethernet, its widespread deployment in datacenters, and recent developments for Radio over Ethernet (RoE) [24], the XCF was based on Ethernet. To better support multi-tenancy MAC-in-MAC or Provider Backbone Bridging (PBB) was chosen as the baseline format for the XCF. MAC-in-MAC allows hiding the tenant (MAC) addresses from the provider network, such that changes to tenant addresses do not cause reconfigurations of the packet forwarding within the provider network. Only the edge of the provider network needs to be reconfigured.

The MAC frames of a tenant are encapsulated with up to two additional headers; they are transported themselves unchanged across the provided network. First, there is the actual MAC-in-MAC header and second there is an (optional) F-Tag (Flow-Filtering Tag) to support equal cost multi-path (ECMP).

The MAC-in-MAC header contains a new Ethernet header with MAC addresses, a B-TAG (Backbone VLAN Tag) to support VLANs, and an I-TAG (Backbone Service Instance Tag) to support further service differentiation. The frame format is shown in Figure 38.

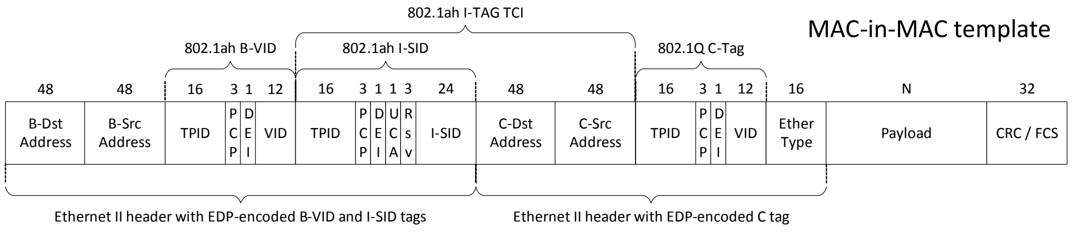


Figure 38: MAC-in-MAC header

The outer MAC addresses are used to address the XPFEs. The destination B-MAC address is the MAC address of the XPFE to which the tenant device, identified by C-Dest address, is connected. The B-VLAN tag contains the VLAN-ID in the provider network as well as the Priority Code Points (PCP) used to prioritize the packets appropriately. The PCP and Discard Eligible Indicator (DEI) values of the I-Tag are redundant to the ones in the B-Tag and are not used in 5G-Crosshaul. The Use Customer Address (UCA) is used to indicate whether the addresses in the inner header are actual client addresses or whether the frame is an OAM frame.

In the 5G-Crosshaul System Architecture, the concept of Multi-tenancy is of primal importance to support several tenants with their own traffic circulating through the network. Specifically, it is necessary to differentiate between tenants because each one has a different portion of a virtual slice of the physical network. Moreover, there are several kinds of services, and each tenant can offer some of them but not necessarily all of them, so it is also important to differentiate services. Finally, there are different types of traffic which have to be differentiated to give more importance to those having more priority in the network. Therefore, the labels of each packet must contain fields to express all these requirements.

The first label to identify a tenant is the B-VID tag contained in the field VID that has 12 bits of length and allows to identify 2^12 = 4096 different tenants. Based on the instructions received from the control plane, the AF configures the PCP and DEI fields to preserve traffic isolation and support SLAs within the 5G-Crosshaul transport network, and a Credit/Time shaper might be associated to each traffic class, identified by the PCP. To differentiate the service used, the I-SID tag is introduced in the I-SID field as a Service ID. There are two possible ways to proceed: the I-SID scope is general, and their values are shared among all the tenants or the I-SID scope is separate per tenant. In both cases, the I-SID is defined by the infrastructure owner. ECMP can be supported by providing a value per end user flow. This value can be used to calculate over which of several paths a frame should be forwarded. If individual flows have different values, this allows distributing flows to different paths while keeping all packets of one flow on the same path. ECMP is not essential for carriers because it might present difficulties for real time services and complicates the management and localization of failures. Such a value is contained in the F-Tag for flow filtering, see [25], clause 44.2 and Figure 39.

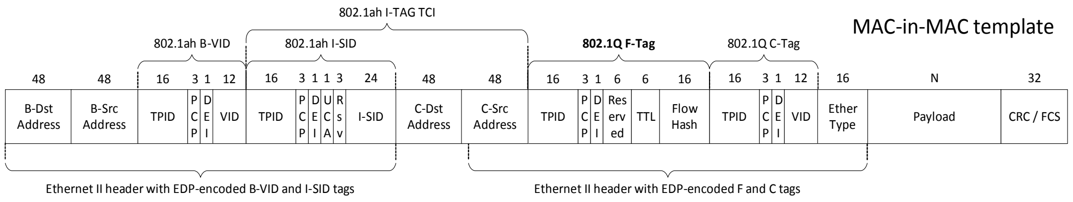


Figure 39: F-tag

The F-Tag contains PCP and DEI fields, which are redundant and are not used within 5G-Crosshaul. The Time to live (TTL) can be used to prevent forwarding loops. But as the F-Tag is considered optional within 5G-Crosshaul this mechanism was not be used. The flow hash field in the F-Tag is the one relevant to support ECMP. MAC-in-MAC as baseline XCF satisfies all the 5G-Crosshaul requirements (Table 44) with the exception of energy efficiency that was deemed not applicable to a frame format.

5.2.2.2 MPLS(-TP) as XCF

In this subsection, we present MPLS Transport Profile (MPLS-TP) as an alternative XCF format. As presented in [26], the physical topology and already installed services have an impact on the choice on the forwarding technology. Other techniques and their corresponding frame formats, such as MPLS-TP, can take the role of MAC-in-MAC as in described before. Both, for MAC-in-MAC as well as MPLS-TP, forwarding decisions would still be done by each XPFE based on forwarding information stored at the XPFEs.

In MPLS-TP, the forwarding of frames is based on MPLS, but the necessary configuration is done via management commands, not via routing protocols such as the Label Distribution Protocol (LDP). Typically, nodes are connected via point-to-point pseudo-wires. Multipoint-to-multipoint networks have to be implemented by a mesh of pseudo-wires.

MPLS-TP has a typical frame structure with an outer label for a label switched path (LSP), an inner label for a pseudo-wire (PW), and an optional PW control word. Figure 40 shows an MPLS-TP header, using Ethernet as data link layer technology. As usual in MPLS, each label contains the actual label, a 3-bit traffic class (TC), a 1-bit indication whether the bottom of the label stack has been reached (EOS), and an 8-bit time to live field (TTL).

The label in the LSP label allows distinguishing different tenants; it would be a task of the XCI to keep track of the relation among tenants and labels per XPFE. Prioritization of different services can be based on the TC bits in the same way as on the PCPs for MAC-in-MAC. Also, traffic flows could be assigned to scheduling classes based on their label. A PW can be used to transport different services, both packet and circuit-oriented.

A label in the LSP label has 20 bits of length and has a local scope so theoretically there are no limits to the number of tenants. Scalability is limited by equipment restrictions (forwarding table dimensions) rather than by the number of available labels. The SDN controller must keep track of the association between the tenant and the complete sequence of labels used for the LSP.

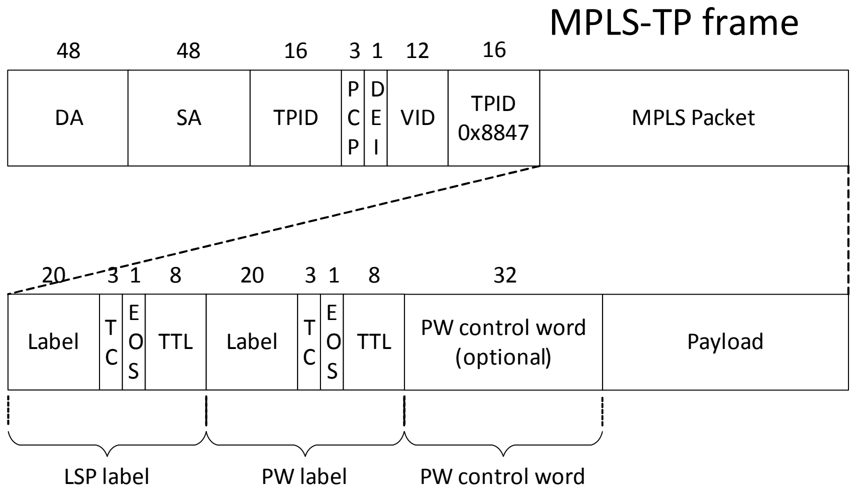


Figure 40: MPLS-TP headers

MPLS(-TP) support two kinds of traffic differentiation:

- E-LSP: EXP-inferred PHB (Per-hop behaviour) scheduling class LSP. In this case, the TC field, originally named EXP, of the MPLS header is used by the XPFE to determine the PHB to be applied to the packet. This includes both the PHB scheduling class and the drop preference. A maximum number of eight scheduling classes is possible.
- L-LSP: Label-only-inferred PHB scheduling class LSP. In this case the PHB scheduling class is explicitly assigned at the time of label establishment. In principle, an unlimited number of scheduling classes is supported.

Different LSPs can be provisioned to transport different flows. Edge nodes or adaptation functions are in charge of classifying the incoming traffic and assigning it to the correct LSP. Intermediate nodes need to analyse only the LSP label (instead of many fields) of the incoming packets and use the forwarding behavior associated with the LSP kind (E-LSP or L-LSP as described before).

As such, multiple tenants can be supported over the same network without interfering their traffic by adding the previous information in the tags to differentiate the traffic of the tenants.

On Ethernet links, MPLS-TP is compatible with synchronization protocols such as synchronous Ethernet or IEEE 802.1AS. MPLS-TP is equally compatible with security mechanisms such as IEEE 802.1X and IEEE 802.1AE. The fulfillment of the 5G-Crosshaul requirements, by MPLS-TP, is summarized in Table 45 in Section 12. MPLS-TP as XCF satisfies the XCF requirements in almost the same way as MAC-in-MAC as XCF does. MPLS-TP also provides a rich set of OAM functionality, but there is no impact on the XCF and is considered an aspect orthogonal to the XCF.

5.2.2.3 Data Plane Implementation

Within a 5G-Crosshaul network some of the XPFEs might be implemented in software. We expect this to be the case for virtual switches on servers aggregating the traffic of VMs or switching traffic among VMs as well as for XPFEs with a low number of interfaces. XPFEs might have their switching functionality implemented in software as OpenFlow capable Application Specific Integrated Circuits (ASICs) are targeting switches with a large number of ports.

We have been investigating how to reduce latency and jitter of a software switch and checking for which fronthaul splits this would be suitable. We based the investigations on the open source switch Lagopus [27] as it provided the MAC-in-MAC support for the XCF. Similar to other high-performance software switches, it is using DPDK [28], a framework for packet processing on general purpose processors. Lagopus processes packets in three stages, namely; reading packets from the Network Interface Cards (NICs) (receive), applying the OpenFlow rules to the packets (worker), and finally transmitting the packets to the NICs (transmit). The three stages are decoupled by ring buffers as shown in the architecture in Figure 41.

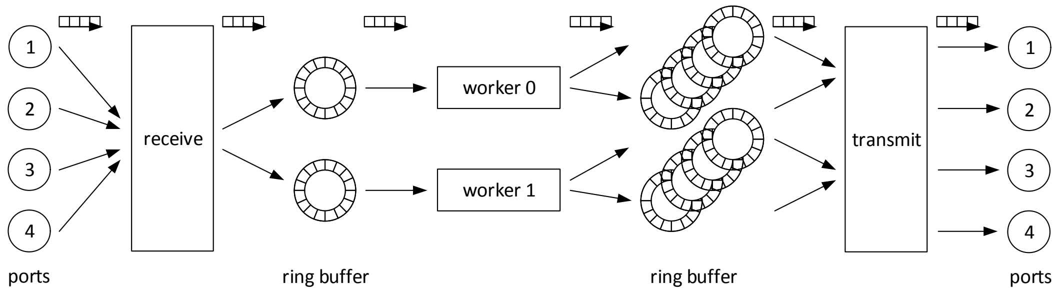


Figure 41: Lagopus internal strucuture

Each of the tasks can be mapped to one or several processing cores. One processing core may also handle several tasks. In the diagram above there is one receive and one transmit core; with two cores for processing the packets in parallel, to forward packets among four ports.

Lagopus is processing packets in bursts, it tries to read a burst of packets from a NIC, workers read a burst of packets from a ring buffer and processes them one after the other before enqueuing them to a ring buffer in one operation. Similarly, the transmit task takes a burst of packets from the ring buffers and tries to write these packets to a NIC for transmission. This approach minimizes the overhead for accessing common data structures and increases throughput, but it increases jitter of the packet processing.

The transmit task provides scheduling functionality in case a port is congested. This scheduling allows to prioritize e.g. fronthaul over backhaul traffic, thus reducing latency and jitter for fronthaul traffic.

To ensure fronthaul traffic is prioritized in case the switch itself becomes congested, we introduced additional ring buffers to allow the worker and transmit cores to process packets according to priority, see Figure 42.

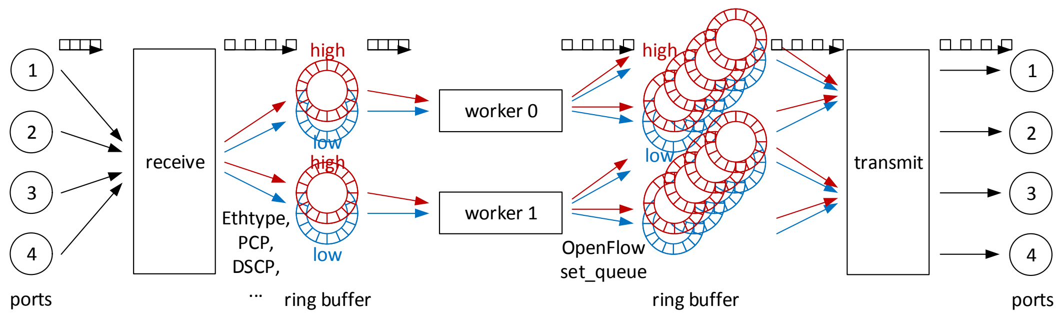


Figure 42: Lagopus enhanced with additional ring buffers

The receive task can classify packets to the ring buffers based on L2 or L3 priority information in packet headers. The worker tasks classify the packets to ring buffers based on OpenFlow set-queue commands.

Additionally, packets are forwarded to the next task as quickly as possible, i.e. packets are inserted to the ring buffers one by one and are also transmitted to the NICs one by one. Receiving packets from the NICs and dequeuing them from the ring buffers for OpenFlow processing is still done in bursts. This difference is indicated at the top Figure 42 by dense and sparse sequences of packets.

As expected, these changes allow to prioritize fronthaul traffic over backhaul traffic and correspondingly reduce the latency of fronthaul relative to backhaul traffic. As an example, the average latency for forwarding 64B packets increases from about 5.3µs for both fronthaul and backhaul traffic in a lightly loaded system to about 7.7µs for fronthaul traffic and several tens of µs for backhaul traffic on a heavily loaded system. When overloading the system, the latency for fronthaul traffic remains at about 7.7µs, whereas latency of backhaul increases to hundreds of µs and packets of backhaul traffic are dropped. In these measurements, Lagopus was executed on a server system, operating at a processor frequency of 1.7GHz and using 10Gbps Ethernet interfaces.

5.2.3 Adaptation function

The general view of the 5G-Crosshaul Adaptation Function (XAF) is depicted in Figure 43. It is important to note that an XAF does not have to provide all of the indicated mapping functions, it may provide an adaptation between just two ports.

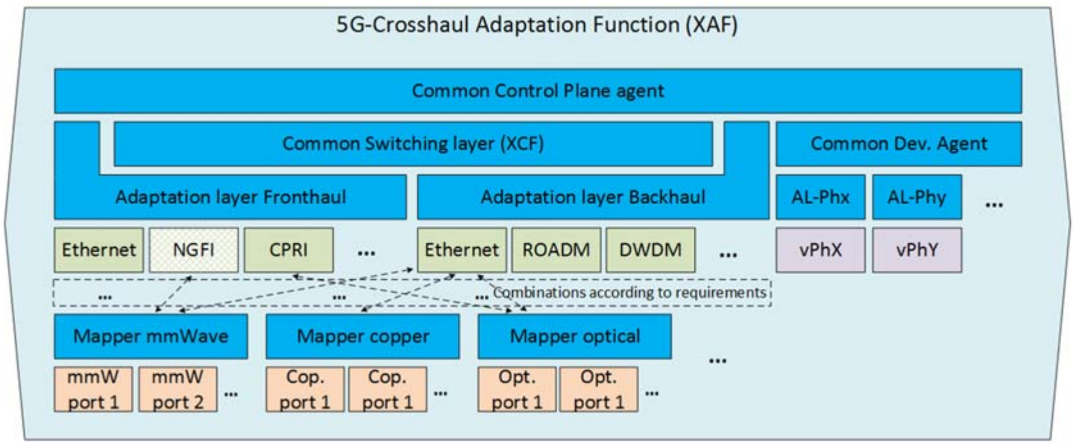


Figure 43: 5G-Crosshaul Adaptation Function [1]

Frames exchanged between the two ports are en/decapsulated with a header according to XCF. Further functionality can be implemented as well by an XAF, e.g. an XAF can provide a de-jitter buffer.

Viewing an XAF as an OpenFlow switch with two physical ports, it can be depicted as in Figure 44.

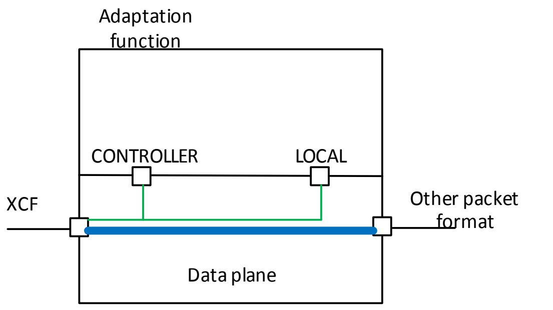


Figure 44: XAF as OpenFlow switch

XAF handles mainly the traffic between the two physical ports; but some simple control is needed, e.g. to set up the flows for en/decapsulation or for managing the XAF, and therefore there is traffic to the LOCAL and the CONTROLLER port for in-band control.

5.3 Quality of Service

Diverse types of FH and BH traffic have different latency and jitter requirements. Overprovisioning transport links, such that no congestion can occur is a traditional way of solving the QoS problem, but it is too expensive. On links with varying bandwidth, such as microwave links, a minimum bandwidth can be guaranteed only, but most of the time more bandwidth is available. This bandwidth variation has to be considered in a QoS aware manner. XPFEs have to distinguish different types of traffic and schedule frames for transmission according the QoS requirements of those frames.

The XCF provides 3 bits to encode priority information, allowing XPFEs to distinguish 8 different traffic classes. In turn, XPFEs provide 8 queues per port, one per each traffic class, to prevent head of line blocking by low-priority traffic. Based on these traffic classes, the 5G-Crosshaul network can provide four major service classes: ultra-low latency, control, low latency and regular. The latter two services are divided further into subclasses for GBR (Guaranteed Bit Rate), nGBR (non-GBR) premium, and nGBR best effort traffic. Table 11 presents a mapping of traffic types to priorities code points. Additional traffic classes would provide even more fine granular control, but the 3 types of service classes for end user traffic are considered sufficient.

Table 11: Traffic Classes

PCP	**Traffic class**	**service class/priority**	**Comment**
7	RoE, eCPRI [29] IQ data, Synchronization	Ideal	May preempt other frames, committed bit rate, ensure there remains sufficient bandwidth for control.
6	Control (network control, FH radio control, BH radio control)	Near/sub ideal/GBR high	Traffic volume not sufficient to starve other traffic classes, split bandwidth further for the different types of control traffic.
5	FH data GBR, mission critical	Near/sub ideal/GBR high	Committed bit rate. Priority of mission critical traffic over GBR traffic can be ensured by admission control on application level.
4	BH GBR, mission critical, tactile, voice, video	Non-ideal/ GBR high	N/A
3	FH nGBR premium, mission critical	Near/sub ideal/nGBR high	Just a part of the bit rate may be committed.
2	FH nGBR best effort	Near/sub ideal/nGBR low	Just a part of the bit rate may be committed.
1	BH nGBR premium, mission critical	non-ideal/nGBR high	Just a part of the bit rate may be committed.
0	BH nGBR best effort	non-ideal/nGBR low	Just a part of the bit rate may be committed.

The service classes – ideal, near/sub ideal, non-ideal – are taken from [26]. To prevent lower-priority traffic classes being starved by the higher-priority ones, the available bandwidth should not be overbooked with high-priority traffic.

The traffic class with highest priority may even preempt the transmission of other frames to reduce jitter [29], if this is considered necessary. Instead of having to wait for the complete transmission of a frame, a high priority frame could be sent immediately after waiting for the minimum fragment size. CPRI-like data could be prioritized using frame preemption applicable to copper, fiber, Ethernet and 802.11 based transport links. For instance, the 802.11ad based Fast Forward technology (FF), described in D2.2 [31], performs frame preemption by using two queues, namely a high priority or FF queue and a normal queue. Packets are classified based on their Ethernet MAC header fields, then they are timestamped, get a Cyclic Redundancy Check (CRC) appended and placed into separate buffers i.e. FF or normal. Packets from the high priority queue are processed by the PHY layer first.

The traffic classes could also be mapped in different ways to the priorities and different service classes could be provided. The specific mapping in this section just shows that 8 PCPs are sufficient to differentiate several different service classes and subclasses.

5.4 Operations Administration and Management (OAM)

Both OAM and SDN protocols define abstractions, functions, and interfaces which need to be implemented on the switch. However, the OAM and SDN models are based on two conflicting assumptions:

- OAM defines *stateful* mechanisms that must be executed on the switch;
- SDN defines *stateless* forwarding model for the switch and delegates stateful logic to the controller;

This causes a mismatch between OAM and SDN. On the one hand, OAM requires the execution of complex tasks on the switch while on the other hand, SDN aims at removing complexity from the switches. Moreover, SDN could in principle provide better network management. However, it is non-trivial to realize enhanced management without a full integration of OAM procedures into SDN paradigm.

The implementation of common packet-based OAM mechanisms such as Connectivity Check (CCMs), Loopback (LBM) and Link Traces Messages (LTM), as defined in the IEEE 802.1ag and ITU-T Y.1731 requires either to: (a) be triggered by the controller or (b) to use SDN-compliant tools to define OAM procedures locally at the switches. Since SDN switches can exclusively perform stateless forwarding, the only way of implementing OAM protocols is on top of the network controller. Such an approach clearly complies with the SDN paradigm and provides the corresponding benefits but shows several drawbacks in OAM operations effectiveness as described in the following:

1) *Connectivity Check:* this protocol requires the switch to generate periodic heartbeat messages, however there is no SDN OpenFlow API for performing such operation. Thus, the network controller needs to overcome such shortcoming and implement the CCM mechanisms.
2) *Loopback:* unlike CCMs, Loopback messages are administratively initiated and stopped. Nevertheless, the Loopback protocol still requires the switch to generate and send specific messages over the data plane and, like in the CCM case, the messages need to be generated in the network controller. One of the main goals of Loopback protocol is to measure the delay of a link. Clearly, such approach prevents to truly measure the delay between two switches on the data plane because of the additional delay introduced by the control plane.

3) *Link Trace:* enables the tracking of a certain path hop-by-hop. To identify each hop, a series of Link Trace Messages (LTM) are sent over the network with incremental Time-To-Live (TTL) values. The L2 destination address of these messages is multicast, the real destination is inside the CFM header, to forward this message and to generate the reply the message need to be processed by the network controller. The main drawbacks of such approach are hence: *i)* the additional delay in the tracing procedure due to the controller-switch distance, and *ii)* the overloading of the controller and control channel.

The centralization of this OAM procedures might cause an overload of the controller due to the periodic message generation (e.g., CCM), this overload together with the delay in the control plane will degrade the performance of the protocols. In particular, loopback metrics obtained by a centralized procedure could be completely useless due to the control plane delay.

We propose the utilization of a set of SDN-compliant tools that allows the definition and configuration of OAM procedures in a local manner on the switches. It provides a better accuracy and performance and allows an offloading of the control plane due to the non-existence of delay because of the local execution. These tools executed in the switch would have an abstract interface to the network controller maintaining the huge benefits of SDN.

5.4.1 In-band control

SDN switches in a data-center can be expected to have out-of-band control, i.e., the switches have a dedicated port used to connect to the management network to which also the SDN controllers are connected. The flow-table modifications to the switch are communicated via this dedicated network. Also, the management of the switch in terms of administratively enabling/disabling ports, updating software, etc., is done via this dedicated network.

Only some XPFEs in a 5G-Crosshaul network are expected to have such a dedicated management network. Other XPFEs deployed in the access network have to be configured and managed in-band, i.e., the management and control traffic is sent on the same links as the actual data traffic. OpenFlow supports in-band control via the LOCAL port, which is a reserved port representing the switch’s local network and management stack (see [32], Section 4.5). Incoming packets from a physical port can be matched by flow-table entries and forwarded as an action to the LOCAL port. Vice versa, messages received from the LOCAL port can be forwarded to the SDN controller or to the management system. The LOCAL port is optional in OpenFlow, but XPFEs with in-band control have to support it.

In-band control requires some kind of bootstrapping, initial flow-table entries are needed to exchange packets between the LOCAL and some physical port where the SDN controller can be contacted. The control functionalities in an XPFE and an SDN controller communicate with IP packets over plain or VLAN-tagged Ethernet frames. In case the configuration and management traffic on the 5G-Crosshaul network conforms to the XCF, then also flow-entries to en/decapsulate the management packets are needed. These flow entries to adapt the management traffic would have to be created before the connection to the SDN controller can be established, i.e. these flow entries would have to be established independent of the SDN controller.

Bootstrapping in case of in-band control is a general problem of SDN-controlled switches and is already under study in more generic setups than 5G-Crosshaul, see e.g. [33] which proposes a method that is tested on an example network, but without going into detail on how to establish a secure control network in an otherwise untrusted network. The OF-CONFIG specification [34] enables applications to manage networking device configurations remotely through secure connections using NETCONF [35] as transport protocol. This provides a secure framework for automated management and control of SDN networks. At the time of writing this deliverable, it does not provide recommendations for the bootstrapping method, but assumes preconfiguration in the switches before they are being added to the network. But such preconfiguration increases complexity in rollouts. In addition, [36] proposes several methods to establish a secure channel. [37] proposes an LTE connection for out-of-band control of OF-switches in a wide area network. Although this would avoid separate physical links, it increases configuration effort and cost as the LTE devices need to be purchased and configured. Additionally, it requires availability of an LTE connection which cannot be assumed in general. [38] proposes extensions to current protocols such as DHCP-SDN, but also does not cover the security challenges in in-band control networks.

In the following we present a procedure for securely bootstrapping XPFEs using minimal assumptions on preconfiguration. Thereafter, we extend this procedure to bootstrap wireless nodes with securely setting up the wireless links.

5.4.1.1 Secure Bootstrapping

The secure bootstrapping process of an XPFE can be separated into several phases:

A. Establish connectivity to the control network and retrieve connection identifiers (IP address and so forth) of an SDN controller and of elements of a Public Key Infrastructure (PKI)
B. Optionally authenticate to the CA (Certificate Authority) and create, sign and download an operator specific certificate
C. Establish secure connection to the SDN controller (e.g. through Transport Layer Security (TLS))
D. Register at the SDN controller by instantiating an OpenFlow session

Each newly added XPFE floods all its neighbors with DHCP messages until it receives a response from a DHCP server. If one of the neighbors is

1. A DHCP server, it replies to the XPFE.
2. The SDN controller, it forwards the messages to the DHCP server towards which it knows the path
3. An XPFE connected to the SDN controller, the device will forward the messages to the SDN controller

If none of the above applies, the messages are dropped.

Once an IP address has been assigned and also other required parameters have been transported as DHCP payload to the new XPFE, it (optionally) accesses the CA. After that it establishes a TLS session with the SDN controller and perform the registration procedure.

The XPFEs should start in fail-secure mode [32] preconfigured with two flow entries:

• Any traffic received on a physical port should be forwarded to the LOCAL port.

• Any traffic from the LOCAL port should be flooded to all physical ports, i.e. to the ALL port.

The LOCAL port is optional for OF switches in general, for our bootstrapping procedure we require this port to exist. No L2 loop can be created by adding a new XPFE to the network as no traffic is forwarded directly among physical ports. The two initial flow entries are the same for all types of XPFEs and therefore do not cause additional administration effort for deploying different OF switches.

The method proposed here relies on TLS to prevent any intervention in the control channel between XPFEs and their SDN controllers. To ease the operation and thereby support broader market support, vendors could install certificates signed by their own CA as part of the production process. Devices could then initially register at operator networks using this ‘vendor certificate’ if the operator network would trust these certificates. Operators can then decide whether they want to deploy their own ‘operator certificates’ in addition, possibly implemented in a fully automated process.

5.4.1.2 Bootstrapping phases

In this section, we present in more detail the bootstrapping of XPFEs, starting from the first XPFE in a network. XPFE by XPFE a control network will be created to which the XPFEs and the SDN controller are connected. In our case, the control network will have a specific outer VLAN address and PBB service identifier (SID).

A) Establish connectivity to the control network and Retrieve connection identifiers of a SDN controller and of elements of a PKI

Initially, the XPFE scans all physical interfaces for active links, and then floods DHCP DISCOVER messages on these interfaces on plain Ethernet in regular intervals until it receives a DHCP OFFER. This DHCP OFFER should be generated by a DHCP server with the support of the SDN controller. The DHCP server needs to be configured with

- An adequate range of IP addresses to assign to the XPFEs
- Additional parameters to share with XPFEs such as IP addresses of SDN controllers and CAs

The path to the DHCP server needs to be known to the SDN controller. This path could be a preconfigured one or the SDN controller and the DHCP server could be located on the same node. If the DHCP server is connected to the XPFE directly connected to the SDN controller, this path can also be learned as part of the bootstrapping procedure.

The first switch on the path from a new XPFE to the SDN controller encapsulates the messages from the XPFE in Packet_In messages and sends them via its own control connection to the SDN controller.

The OFFER message is expected to contain one or more DHCP Option 43 entries (vendor-specific option) containing the IP address of the SDN controller to connect to and the IP address of the CA (optional). When such a message is received, the DHCP procedure continues with the XPFE exchanging REQUEST and ACKNOWLEDGE messages with the DHCP server. Eventually, the OF switch has an IP address assigned to it and is configured with the IP address of a remote SDN controller and optionally the IP address of a CA, all to continue the bootstrapping process.

B) Authenticate to CA and update certificate

Optionally, operator certificates can be enrolled automatically to new XPFEs. A network operator would not need to preconfigure each XPFE with a valid own certificate before adding it to the network, but can instead rely on the vendor certificate for the initial handshake. If used, this procedure, needs to be performed before the XPFE establishes a secure connection with the SDN controller. There are several options to implement such a procedure. The one summarized here is described in more detail in [39].

As a precondition for the operator certificate enrolment the XPFE needs to be preconfigured with a switch-vendor provided private/public key pair and with the related certificate signed by a vendor CA. During phase A it also needs to have received connection identifiers (i.e. IP address) of the CA server of the operator of the network. The operator CA needs to be preconfigured with a certificate of the vendor and its own certificate(s). The protocol being used for the enrolment is CMPv2 [40].

To begin the procedure the XPFE generates a new private/public key pair to be enrolled in the operator CA. It then starts interaction with the CA by generating the Initialization Request (ir) and sending it to the CA. The CA verifies the digital signature on the ir message against the vendor root certificate. It also verifies the proof of the possession of the private key for the requested certificate. Then it generates the operator certificate for the XPFE. This certificate is sent back to the XPFE in the Initialization Response (ip). The XPFE extracts and installs the new certificate generated for it by the operator, and if necessary also the operator root certificate. The XPFE confirms receiving of the ip message by generating and sending a Certificate Confirm (certconf) message to the CA, which the CA responds with a Confirmation message (pkiconf). Thereafter, the XPFE possesses a certificate trusted by the operator network. The message exchange is shown in Figure 45.

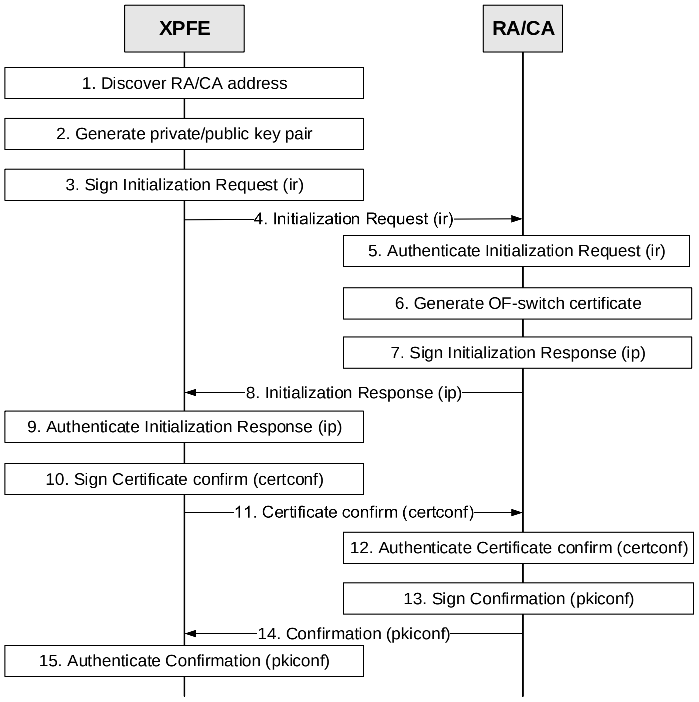

XPFE
RA/CA
1. Discover RA/CA address
2. Generate private/public key pair
3. Sign Initialization Request (ir)
4. Initialization Request (ir)
5. Authenticate Initialization Request (ir)
6. Generate OF-switch certificate
7. Sign Initialization Response (ip)
8. Initialization Response (ip)
9. Authenticate Initialization Response (ip)
10. Sign Certificate confirm (certconf)
11. Certificate confirm (certconf)
12. Authenticate Certificate confirm (certconf)
13. Sign Confirmation (pkiconf)
14. Confirmation (pkiconf)
15. Authenticate Confirmation (pkiconf)

Figure 45: Example message flow for OF switch enrolment [39]

C) Establish secure connection to the SDN controller

The XPFE establishes a secure connection with the SDN controller (see [32]) by firstly resolving its MAC address, we propose the control network is a L2 one. Thereafter, the XPFE sets up a TLS session with the SDN controller, using the certificates of the SDN controller and the XFPEs to generate the keys for the symmetric encryption and to prove to the SDN controller that the XPFE indeed possesses the private key corresponding to its public key certificate.

An exchange of TLS Finished messages, already sent encrypted, completes the procedure.

D) Register at the SDN controller by instantiating an OF session

After the secure connection has been established, the new XPFE and SDN controller need to send a OF-Hello message to each other, containing the OF protocol version set to the highest version supported by the sender. Next the SDN controller sends a Feature Request to the XPFE which is being replied by a Feature Response containing the list of features the XPFE supports. Based on this data the SDN controller continues to configure the XPFE and to discover the extended topology. Note that, up to this time, the OF switch has flooded all frames sent from its LOCAL port on all its physical ports.

Eventually, the SDN controller installs flow entries in the new XPFE to connect to the control network and the next XPFE to extend the control network to the new XPFE. The flow entry changes for the new XPFE have to be sent by the SDN controller in one message as the changes will cause a temporary interruption of the control connection. If not all information is available on the new XPFE to re-establish the control connection, it will remain disconnected. From this point onwards, no more Packet_In or Packet_Out messages are used to communicate with the new XPFE. The new XPFE is integrated into the network and is ready to be configured by the SDN controller for forwarding production traffic.

A detailed example is shown in Section 12. The procedure described above is a basic procedure to integrate new XPFEs into a network. Once the XPFE is connected to an SDN controller, further steps can be taken to enhance reliability and robustness of the control network.

5.4.1.3 In-band signaling for wireless nodes to join SDN network

The procedure described above can be used as well for XPFEs with wireless links only. As an additional step for a wireless node based on IEEE 802.11ad [41], the establishment of the wireless links requires already some kind of authentication.

This is shown in the example network in Figure 46, with 4 wireless sector interfaces on each of 4 nodes. Some of the sectors are used to connect the node as a station (STA) to another node, in other sectors the node acts as a Personal Basic Service Set (PBSS) Control Point (PCP) according to [41]. Node 1 is the new node joining the network. Node 2 is the peer node to which Node 1 is connecting to and acts as a relay node. Node 3 is a gateway node connected to the wired network via a service provider (SP) GW from which the SDN controller is reachable via IP routing. Finally, Node 4 is a leaf node on the network connected to Node 2 on the same sector interface as Node 1 creating a point-to-multipoint topology.

Before the generic procedure can start, Node 1 (new node), establishes a temporary association with neighbour Node 2. While establishing the temporary association, Node 2 informs the controller, via a New Node Link Report, that Node 1 is attempting to associate to Node 2. The controller authenticates the identity of Node 1 (new node) e.g. by using the vendor certificate of this node. Once authenticated, Node 1 can start the generic procedure as described above. At this point, the controller is aware of the Node 1 sector MAC address that is associating to Node 2.

The new node joins the SDN controlled wireless network by sending a DHCP message over an established wireless connection. The procedure continues with the exchange of IP addresses, optional enrolment of an operator certificate, and eventually the establishment of the control channel among node1 and the SDN controller. Note the operator certificate is not available for the initial establishment of the wireless link. Only the vendor certificate can be used for this establishment. The operator certificate can be established only once the first links have been created.

The New Node Link Report contains information on bridge and link MAC addresses of both ends of the physical link and can be used for topology discovery as an alternative for LLDP or similar protocols. Note, the Packet_In messages used in the procedure contain information on the used port for the XPFE where a new XPFE is connected, but not which port the new XPFE is using to connect.

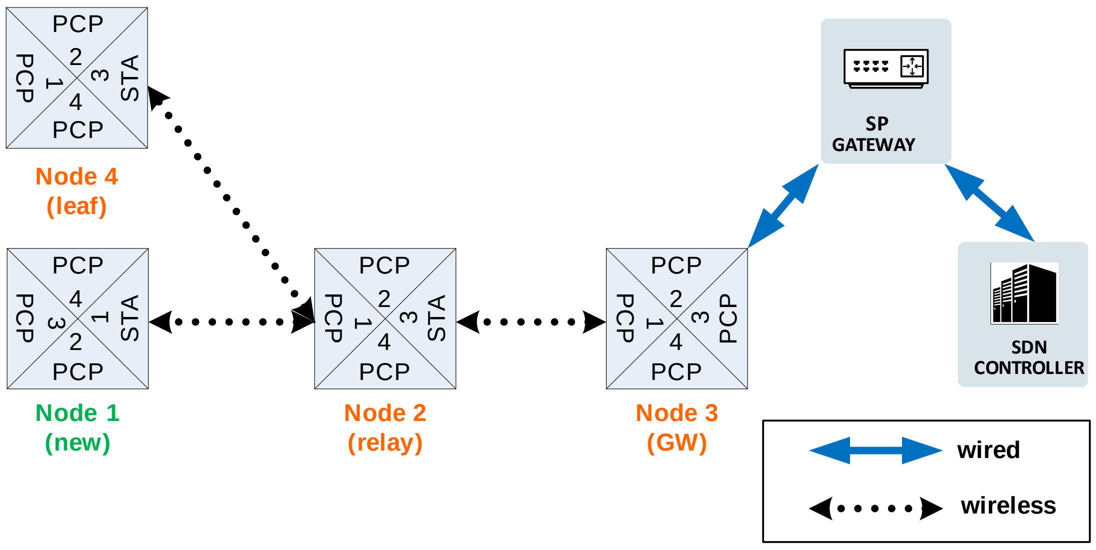


Figure 46: 4 Node SDN-Controlled network

5.5 Synchronization

All mobile systems require synchronization between RUs in order to support handover and to minimize interference. Also, the CU and RU need to share a precise clock in order to fulfil stringent regulatory requirements. Often, a Global Navigation Satellite System (GNSS) like the Global Positioning System (GPS) is used to provide synchronization to a radio base station. With traditional time division multiplexing (TDM) based transport like CPRI fronthaul, frequency synchronization in RUs could then be derived from the physical layer of the fronthaul transport technology between CU and RU. However, with the move towards packet-based transport like eCPRI or higher layer functional splits in general, the ability to distribute synchronization is changing and other synchronization distribution methods are needed. Further, for basic operation of FDD mobile systems, frequency synchronization is sufficient but evolving wireless networks with time division duplexing (TDD) operation, carrier aggregation, mobile positioning, Multimedia Broadcast/Multicast etc. require accurate time and phase in addition to frequency. For the XFE in 5G-Crosshaul, Synchronous Ethernet (SyncE) and/or Precision Time Protocol (PTP) (or a combination of the two) are relevant:

- Synchronous Ethernet is an ITU-T standard for computer networks that allows the transmission of clock signals (frequency sync) over Ethernet. It includes three recommendations, namely ITU-T Recommendations G.8261, G.8262 and G.8264. While the IEEE 802.3 standard specifies Ethernet clocks to be within ±100 ppm, the accuracy of Ethernet Equipment slave Clocks (EECs) in SyncE must be within ±4.6 ppm. SyncE allows nearly immediate frequency lock but requires that all traversed network nodes have SyncE support. Links using IEEE 802.11 (WLAN) technology do not support SyncE.
- IEEE 1588-2008, PTP Version 2 [42], is a hierarchical master-slave architecture for clock distribution across packet networks. PTP encapsulations exist for

various protocols such as UDP, IP, and Ethernet. Hardware time-stamping of packets allows time precision down to tens of nanoseconds. Not all the options and features in the standard are needed for all applications and therefore IEEE 1588 introduced the concept of profiles. IEEE 1588 defines only the default profile for industrial-automation applications while other profiles are specified elsewhere. PTP profiles contain both required and prohibited options, as well as ranges and defaults for configurable attributes, to meet specific application requirements.

IEEE 802.1AS [43], approved in February 2011, is a PTP profile for Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks. More precisely, the IEEE 802.1AS standard, defines a mechanism to ensure that synchronization requirements are met for time-sensitive applications across Bridged and Virtual Bridged Local Area Networks consisting of LAN media where the transmission delays are fixed and symmetrical. IEEE 802.1AS can be applied to provide synchronization between the XFEs in the 5G-Crosshaul network in IEEE 802.3 Ethernet networks but not for IEEE 802.11 based links. The latter needs another solution, e.g. a native timing mechanism. 802.1AS is plug-and-play, that is, the Grand Master clock is selected automatically. After this, a clock tree reconfigures automatically, whereby bridges in the tree propagate time towards the leaves. The master periodically broadcasts the time reference to the other clocks (up to 10 messages per second are permitted in PTPv2).

To support all features of modern mobile systems, ITU-T has developed a PTP profile specifically to address the stringent requirements of telecom applications, the so called “Telecom Profile” [44] where recommendations G.827x from the ITU-T address the actual profile (G.8275.1), network requirements (G.8271, G.8271.1), and clock specification (G.8272, G.8272.1, G.8273, G.8273.2, G.8273.3) for synchronous networks. Recommendation G.8275.1 specifies a profile with full timing support from the network while G.8275.2 specifies a profile with only partial timing support from the network. The latter could be of interest when interfacing legacy networks but better performance is expected with the full timing support option. Finally, ITU-T G.8265.1 specifies a PTP profile for applications that only need frequency synchronization. One-way messaging is sufficient for frequency distribution but for phase and time distribution, two-way messaging must be used. By using networking equipment with hardware time-stamps and timing support in intermediate nodes (boundary clock or transparent clock), it should be possible to achieve timing errors down to a few tens of nanoseconds, supporting even the most stringent timing alignment requirements in 3GPP, e.g. for Carrier Aggregation and MIMO. Another advantage is that the Telecom Profile supports operation over UDP/IP, which means that it is independent of the layer 2 protocol. These properties make it a suitable choice for packet transport synchronization in 5G-Crosshaul.

In PTP, phase and time synchronization between master and slave clocks require the exchange of four messages (see Figure 47) with time-stamps for both distributing the local time and estimating the round-trip to improve accuracy. Two important conditions must be satisfied to achieve good performance: The first condition is that the time offset between master and slave must be approximately constant during the exchange of the above mentioned four messages. If there is a frequency offset between master and slave, the time offset is only approximately constant during a short period. This problem can be handled by imposing accuracy requirements on the free-running clocks, by using syncE to remove frequency offset, or by model-based approaches taking an unknown frequency offset into account. Packet Delay Variation (PDV) creates similar problems as frequency offsets but are more random in nature and thus more difficult to mitigate. The second condition is that the forward and backward transit times should be equal, i.e. the network should be symmetric.

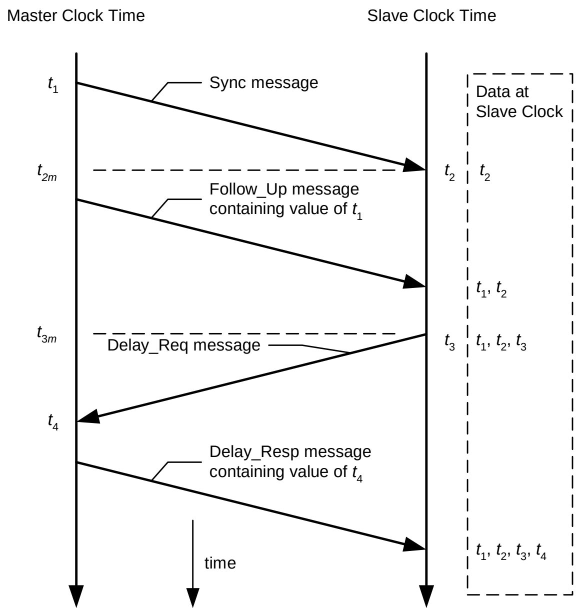


Figure 47: Synchronization with PTPv2.

Once the slave clock received the values of t_1, t_2, t_3 and t_4, and if the above conditions are met, the slave can estimate the clock offset and propagation delay to the master as follows:

$$\text{Clock offset} = [(t_2 - t_1) - (t_4 - t_3)]/2$$

$$\text{Propagation time} = [(t_2 - t_1) + (t_4 - t_3)]/2 = [(t_4 - t_1) - (t_3 - t_2)]/2$$

If other traffic (e.g. time-sensitive FH traffic) has higher priority than PTP through the XFEs, there may be significant jitter in the clock offset and propagation time estimation due to high PDV during the PTP message exchange. Even if PTP has the same or higher priority than interfering traffic, PDV may be high unless pre-emption is implemented.

The problem with high PDV during sync message exchange can be mitigated, at least to some extent, by proper strategies in the clock offset and propagation delay estimation process. To identify the most significant problems for synchronization in 5G-Crosshaul, an FPGA-based testbed was built and phase noise measurements were performed on the recovered clock for a use case where PTP is only implemented in endpoints [45]. Further work has been performed to highlight practical difficulties and potential solutions of selection and filtering techniques to achieve low time error in the presence of PDV [46]. Results for the simplified topologies studied so far, show that it is possible to fulfil relevant 3GPP radio requirements even when PTP is only implemented in the endpoints. These results could be relevant for a case where legacy equipment is connected to 5G-Crosshaul XFEs via adaptation functions. However, it is expected that more complicated topologies with many hops and varying traffic mixes may still need timing support from intermediate nodes (XFEs) to achieve desired performance. Further evaluations on the testbed using network switches with timing support (according to ITU-T Telecom Profile), yielded timing errors with a standard deviation as low as 15 ns. Also, with timing support from the network, timing error is affected less by network topology.

6 Southbound interface design

Openflow [32] has been adopted at the SBI to control the XPFEs. In this section we describe which OpenFlow actions are used by the adaptation functions and the XPFEs and we describe the XPFE OpenFlow pipeline. We also describe how the SDN controllers retrieve topology information from the XPFEs, especially information on ports. Extensions of Openflow to control microwave devices have been described already in [5], section 9.2.

6.1 OpenFlow and Adaptation Function

The Adaptation function encapsulate and decapsulate the customer fronthaul/backhaul traffic as shown in Section 6.3. Openflow supports MAC-in-MAC since version 1.3 by defining the `PUSH_PBB` and `POP_PBB` actions for encapsulation and decapsulation.

The `PUSH_PBB` header action logically pushes a new PBB service instance header onto the packet (I-TAG TCI), and copies the original Ethernet addresses of the packet into the customer addresses (C-DA and C-SA) of the tag. The PBB service instance header should be the outermost tag inserted, immediately after the Ethernet header and before other tags. The customer addresses of the I-TAG are in the location of the original Ethernet addresses of the encapsulated packet. This action adds both the backbone MAC-in-MAC header and the I-SID field to the front of the packet. The `PUSH_PBB` header action does not add a backbone VLAN header (B-TAG) to the packet, this can be added via a `PUSH_VLAN` header action after the `PUSH_PBB` header operation. After this operation, regular `set-field` actions can be used to modify the outer Ethernet addresses (B-DA and B-SA).

A `POP_PBB` header action logically pops the outer-most PBB service instance header from the packet (I-TAG TCI) and copies the customer addresses (C-DA and C-SA) in the Ethernet addresses of the packet. This action removes the backbone MAC-in-MAC header and the I-SID field from the front of the packet. The `POP_PBB` header action does not remove the backbone VLAN header (B-TAG) from the packet; it should be removed prior to this operation via a `POP_VLAN` header action.

The F-Tag is considered optional within 5G-Crosshaul as described in Section 5.2.2.1. The OpenFlow 1.5.1 specification does not support the F-Tag in `PUSH_VLAN` and `POP_VLAN` actions. Therefore, in case of optional employment of the F-Tag within 5G-Crosshaul, OpenFlow extensions must be defined. Despite the F-Tag, OpenFlow supports since version 1.3 all the required functions envisioned for the Adaptation Function in case of having MAC-in-MAC as XCF baseline.

6.2 OpenFlow and XPFE

XPFEs forward the XCF traffic based on the information contained in the MAC-in-MAC header according to the OpenFlow forwarding model. The first operation performed by an XFE for XCF forwarding is to match the incoming packets against the flow entries of the flow tables. Table 12 reports the MAC-in-MAC header fields that can be matched by using different OpenFlow versions.

Table 12: OpenFlow support for MAC-in-MAC fields: match and set-fields

Field	**OF version**	**Comment**
Backbone Destination Address	OF-1.0	Same as Ethernet Destination Address
Backbone Source Address	OF-1.0	Same as Ethernet Source Address
Backbone VID: TPID	OF-1.1	Same as S-TAG Ethertype: 0x88a8
Backbone VID: PCP	OF-1.1	Same as S-TAG PCP
Backbone VID: DEI	N/A	Not supported
Backbone VID: VID	OF-1.1	Same as S-TAG VID
Instance SID: TPID	OF-1.3	Ethertype: 0x88E7
Instance SID: PCP	OF-1.3	
Instance SID: DEI	N/A	Not supported
Instance SID: UCA	OF-1.4	
Instance SID: I-SID	OF-1.3	
Optional: F-TAG: TPID	N/A	Not supported
Optional: F-TAG: PCP	N/A	Not supported
Optional: F-TAG: DEI	N/A	Not supported
Optional: F-TAG: TTL	N/A	Not supported
Optional: F-TAG: Hash	N/A	Not supported

Even the latest OpenFlow version (1.5.1 at the time of writing this report) does not support the matching and the configuration of the DEI field. Such field may be used separately or in conjunction with the PCP to indicate frames eligible to be dropped in the presence of congestion. Therefore, OpenFlow still has to be extended to support matching and configuration of the DEI field.

A flow entry is composed of a match and an action set. The action set of the incoming matched XCF packets will contain the output instruction, or the `DROP` action alternatively, including the port and queue the XCF frame should be forwarded to. XCF frames are directed to one of the queues based on the packet output port and the packet queue id, set using the `OUTPUT` action and `SET_QUEUE` action respectively. XPFEs provide eight queues per port, one per each traffic class, to prevent head of line blocking by low-priority traffic. A specific PCP value is associated to each queue as reported in Section 5.3.

In general, an OpenFlow switch provides limited QoS support. A switch can optionally have more than one queue attached to a specific output port, and those queues can be used to schedule packet transmission. Packets mapped to a specific queue will be treated according to that queue's configuration. Queue processing happens logically after all OpenFlow pipeline processing. Packet scheduling using queues is not defined by the OpenFlow specification and it is switch-dependent; in particular, no priority between queue IDs is assumed. Hence, queue configuration takes place outside the OpenFlow switch protocol, either through a command line tool or through an external dedicated configuration protocol. The 5G-Crosshaul XPFEs use a hard-coded queue configuration or read the queue configuration from a file.

6.3 XPFE Flow Pipeline

The XPFEs are based on an OpenFlow pipeline to forward XCF frames as well as to encapsulate tenant frames into XCF frames and to decapsulate them vice versa. Encapsulation and decapsulation is based on the L2-L3-ACL example defined in [49]. As the XCF is based on PBB we omitted the L3 part from this example completely. The pipeline consists of three different paths as shown in Figure 48.

- forward XCF frames to another XPFE
- decapsulate XCF frames and deliver them to directly connected hosts,
- handle frames received from directly connected hosts, either by sending them to another directly connected host or by encapsulating them and sending them to another XPFE.

Some of the tables in the pipeline are replicated per tenant. These tables are numbered *t,n* in Figure 48. In case the total number of tables exceeds the OpenFlow limit of 256 tables per XPFE, the traffic of several tenants has to be handled by the same set of tables. This can be achieved by extending each flow rule with a match against a service id recorded in the frame metadata.

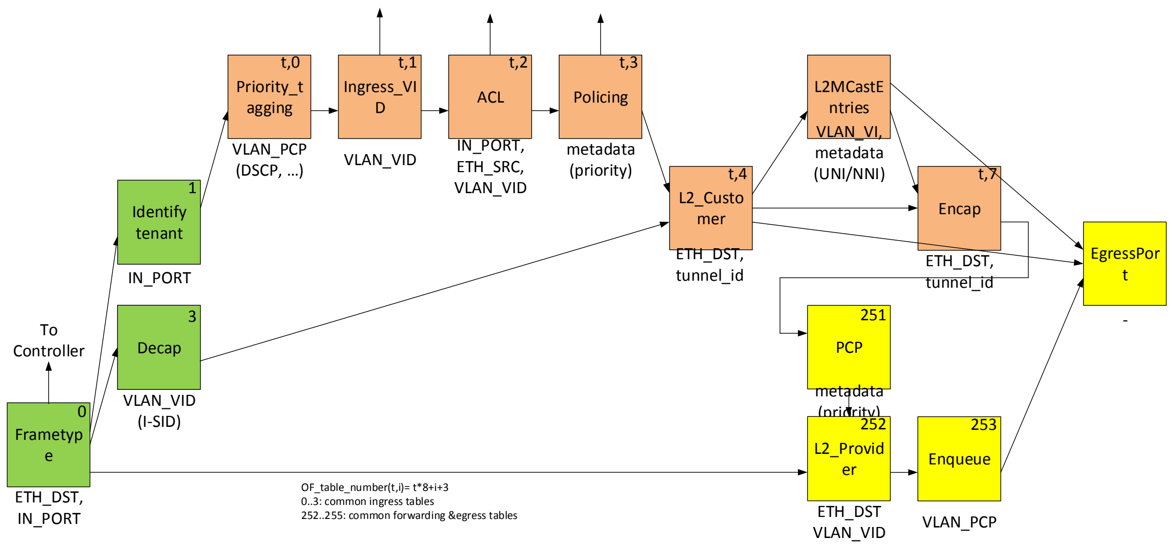


Figure 48: XPFE Open Flow Pipeline

To determine the correct treatment of received frames (Table 0 in Figure 48), we divide the ports logically into two groups of User Network Interface (UNI) and Network Network Interface NNI ports. At UNI ports the hosts of tenants can be connected, assuming non-XCF, but VLAN-tagged Ethernet frames are exchanged.

At NNI ports, other XPFEs can be connected, exchanging XCF frames. There is an outer VLAN around the PBB header, therefore the outermost Ethertype is the same for both XCF and non-XCF frames. Therefore, we use the UNI/NNI port distinction to distinguish traffic from hosts and from other XPFEs and record this distinction in packet metadata. Frames received at a NNI port can still receive two different treatments. When the destination address of a received frame matches a specific MAC address of the port, acting as an XPFE identifier, then the frame is decapsulated, otherwise it is forwarded.

In the forwarding path, the egress port is determined based on the destination MAC address and the outer VLAN Id (Table 252 in Figure 48), allowing different forwarding decisions for different services or tenants. This is followed by a table mapping the frames to egress queues based on their outer PCP (Table 253 in Figure 48).

For frames to be decapsulated, a tunnel Id is recorded in metadata of the frame (Table 3 in Figure 48). The egress port is determined based on the destination address and tunnel Id (Table t,4 in Figure 48). In case the destination address is a multicast address, the frame is forwarded to all UNI ports of the service (Table t,5 in Figure 48). Both tables are shared with the encapsulation path. To implement a split horizon for multicast frames, i.e. to prevent that the frame is encapsulated again, the metadata recording UNI/NNI port reception is used. We consider the network to be tightly controlled, i.e. addresses of connected devices are known. Therefore, frames with unknown destination address are not flooded, they are silently discarded.

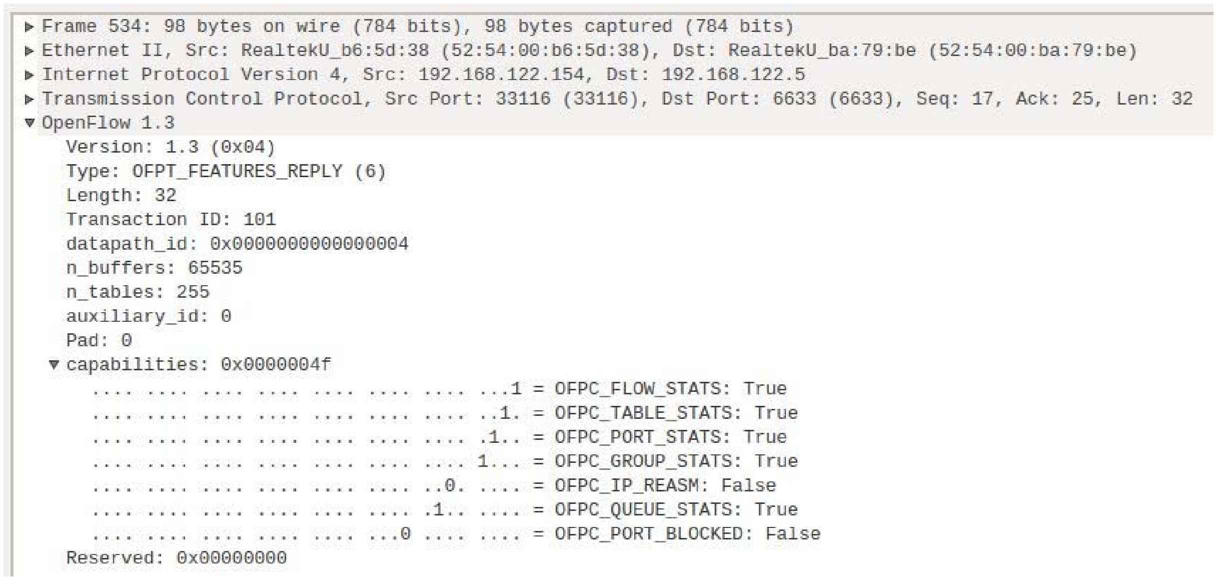
In the encapsulation path, the service is determined based on ingress port and customer VLAN (Table t,1 in Figure 48), and the priority is determined (Table t,0 in Figure 48). Both information is recorded in metadata. For the sake of simplicity, we assume that frames at a UNI port have a VLAN header, allowing to determine service and priority easily. The network operator may actually use a different priority on the frame than a tenant. Priority remarking has to be negotiated among operator and tenant. Note, the original frame is not changed, the priority is carried in the outer VLAN.

After identifying the service and priority the received frames are checked whether they may access the network (Table t,2 in Figure 48), allowing operators to enforce the SLAs with their customers. Thereafter the frame is delivered locally or forwarded via another XPFE. In the latter case, it is encapsulated (Table t,7 and 251 in Figure 48) and the same forwarding decisions (Table 252 in Figure 48) as for received XCF frames are taken.

We mapped separate functionalities to separate tables in the pipeline for the sake of clarity, although some of the tables could be combined from a technical perspective. The complexity of the pipeline is comparable to the one for MPLS as described in [50]. This MPLS pipeline contains handling of OAM frames, which is still missing from our pipeline. OAM frames could be determined in the first table based on Ethertype, the UCA bit in the PBB header, or their address, and forwarded to a local entity for further treatment or to another XPE for e2e monitoring.

6.4 XPFE Configuration

In the 5G-Crosshaul infrastructure the interaction between the SDN Controller and the network-related data plane, composed of XPFE devices, takes places using the OpenFlow switch protocol [32]. The XPFEs connect to an SDN controller with an initial OFMP_HELLO message. Thereafter the SDN controller retrieves the list of capabilities of the XPFE via OFMP_FEATURES_REQUEST/REPLY messages. The example in Figure 49 shows that the XPFE supports various statistics, but it cannot reassemble IP packets.

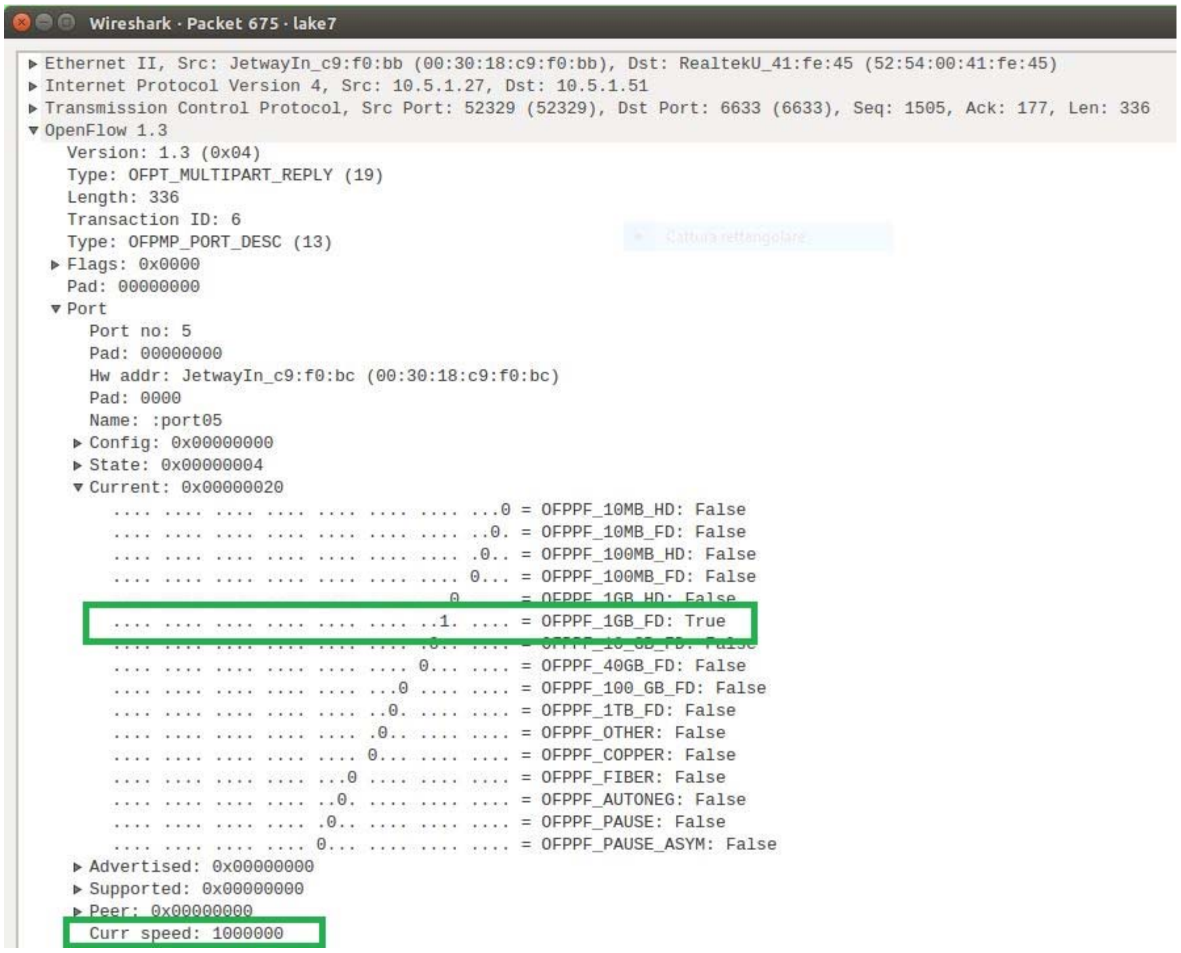


```
▶ Frame 534: 98 bytes on wire (784 bits), 98 bytes captured (784 bits)
▶ Ethernet II, Src: RealtekU_b6:5d:38 (52:54:00:b6:5d:38), Dst: RealtekU_ba:79:be (52:54:00:ba:79:be)
▶ Internet Protocol Version 4, Src: 192.168.122.154, Dst: 192.168.122.5
▶ Transmission Control Protocol, Src Port: 33116 (33116), Dst Port: 6633 (6633), Seq: 17, Ack: 25, Len: 32
▼ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_FEATURES_REPLY (6)
    Length: 32
    Transaction ID: 101
    datapath_id: 0x0000000000000004
    n_buffers: 65535
    n_tables: 255
    auxiliary_id: 0
    Pad: 0
  ▼ capabilities: 0x0000004f
      .... .... .... .... .... .... .... ...1 = OFPC_FLOW_STATS: True
      .... .... .... .... .... .... .... ..1. = OFPC_TABLE_STATS: True
      .... .... .... .... .... .... .... .1.. = OFPC_PORT_STATS: True
      .... .... .... .... .... .... .... 1... = OFPC_GROUP_STATS: True
      .... .... .... .... .... .... ..0. .... = OFPC_IP_REASM: False
      .... .... .... .... .... .... .1.. .... = OFPC_QUEUE_STATS: True
      .... .... .... .... .... ...0 .... .... = OFPC_PORT_BLOCKED: False
    Reserved: 0x00000000
```

Figure 49: OFMP_FEATURES_REPLY

The SDN controller monitors the connection to the XPFE with OFPT_ECHO_REQUEST/REPLY messages. It also retrieves further general information of the XPFE with OFPT_BARRIER_REQUEST/REPLY and OFPT_ROLE_REQUEST/REPLY messages.

The SDN controller requests more specific information of the XPFE, e.g. number and bandwidth of ports. The information provided by the XPFE might be too large to fit into a single IP packet, therefore OpenFlow multipart messages are used to exchange the information. Figure 50 shows an example of the port description of an XPFE, indicating the interface speed (Curr speed).


Wireshark · Packet 675 · lake7

```
▶ Ethernet II, Src: JetwayIn_c9:f0:bb (00:30:18:c9:f0:bb), Dst: RealtekU_41:fe:45 (52:54:00:41:fe:45)
▶ Internet Protocol Version 4, Src: 10.5.1.27, Dst: 10.5.1.51
▶ Transmission Control Protocol, Src Port: 52329 (52329), Dst Port: 6633 (6633), Seq: 1505, Ack: 177, Len: 336
▼ OpenFlow 1.3
    Version: 1.3 (0x04)
    Type: OFPT_MULTIPART_REPLY (19)
    Length: 336
    Transaction ID: 6
    Type: OFPMP_PORT_DESC (13)
  ▶ Flags: 0x0000
    Pad: 00000000
  ▼ Port
      Port no: 5
      Pad: 00000000
      Hw addr: JetwayIn_c9:f0:bc (00:30:18:c9:f0:bc)
      Pad: 0000
      Name: :port05
    ▶ Config: 0x00000000
    ▶ State: 0x00000004
    ▼ Current: 0x00000020
        .... .... .... .... .... .... .... ...0 = OFPPF_10MB_HD: False
        .... .... .... .... .... .... .... ..0. = OFPPF_10MB_FD: False
        .... .... .... .... .... .... .... .0.. = OFPPF_100MB_HD: False
        .... .... .... .... .... .... .... 0... = OFPPF_100MB_FD: False
        .... .... .... .... .... .... ...0 .... = OFPPF_1GB_HD: False
        .... .... .... .... .... .... ..1. .... = OFPPF_1GB_FD: True
        .... .... .... .... .... .... .0.. .... = OFPPF_10_GB_FD: False
        .... .... .... .... .... .... 0... .... = OFPPF_40GB_FD: False
        .... .... .... .... .... ...0 .... .... = OFPPF_100_GB_FD: False
        .... .... .... .... .... ..0. .... .... = OFPPF_1TB_FD: False
        .... .... .... .... .... .0.. .... .... = OFPPF_OTHER: False
        .... .... .... .... .... 0... .... .... = OFPPF_COPPER: False
        .... .... .... .... ...0 .... .... .... = OFPPF_FIBER: False
        .... .... .... .... ..0. .... .... .... = OFPPF_AUTONEG: False
        .... .... .... .... .0.. .... .... .... = OFPPF_PAUSE: False
        .... .... .... .... 0... .... .... .... = OFPPF_PAUSE_ASYM: False
    ▶ Advertised: 0x00000000
    ▶ Supported: 0x00000000
    ▶ Peer: 0x00000000
      Curr speed: 1000000
```

Figure 50: OFPMP_PORT_DESC

Similarly, the SDN controller requests information on flow tables, groups, and meters.

This initialization phase is followed by a periodically interaction between the SDN controller and the XPFE for the statistics polling. Statistics are retrieved for flows, ports, and groups. Again, multipart messages are used. The information received from the switches are stored in the SDN controller datastore. In the case of the EMMA application, the module in charge of doing this kind of operation is the Openflow Plugin within the ODL-based SDN controller. In particular, this module updates the Inventory datastore within ODL and then all the information related with the Openflow nodes in the data plane are reachable via a REST API.

Besides collecting information from the XPFEs, the SDN controller configure the XPFEs. Most prominently it provides the flow-entries for the various flow-tables.

The Openflow plugin behaviour is well defined through its yang interfaces, which define both the types and the available operations:

- Messages[4]
- Inventory datastore information[5]. This model augments the node connectors, for example to support statistics such as the queue statistics:

```
grouping flow-node-connector {
        description "Wrapper of openflow port. ";
        uses port:flow-capable-port;
}
augment "/inv:nodes/inv:node/inv:node-connector" {
        ext:augment-identifier
          "flow-capable-node-connector";
        description
          "Openflow port into inventory tree.";
       uses flow-node-connector;
}
```

- Flow-capable port definition[6]
- Queue-packet definition for queue statistics[7]

6.5 XCSE Configuration

The XCSE is composed of a framing, see D2.1 Section 6.2 [5], the TDM switch and the optical switch. The XCSE is configured according to a proprietary South Bound Interface, an RPC protocol [51] is used for the demo implementation.

The XCSE modeling is shown in the figure below:

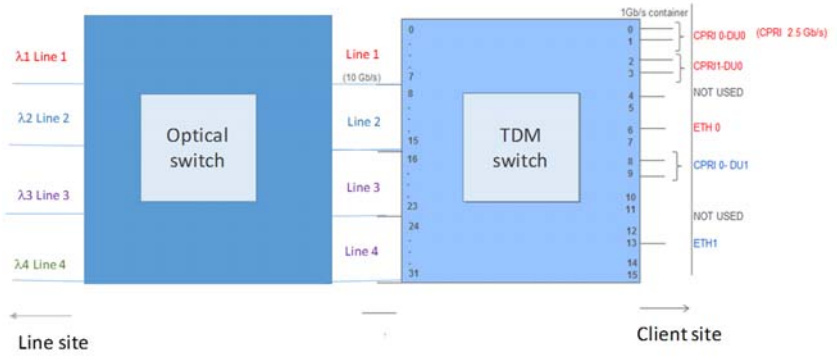


Figure 51: XCSE model

In this model, several client ports are dedicated to transport CPRI, and Ethernet traffic. The granularity of the TDM switch port is the same as the framing slot granularity (e.g.

[4] https://github.com/opendaylight/openflowjava/blob/master/openflow-protocol-api/src/main/yang/openflow-types.yang
[5] https://github.com/opendaylight/openflowplugin/blob/master/model/model-flow-service/src/main/yang/flow-node-inventory.yang
[6] https://github.com/YangModels/yang/blob/master/experimental/odp/opendaylight-port-types.yang
[7] https://github.com/YangModels/yang/blob/master/experimental/odp/opendaylight-queue-types.yang

1.25 Gb/s as slot framing granularity). Hence in case of CPRI signal with 2.5 Gb/s, two slots are considered, while in case of Ethernet client at 1 Gb/s as bit rate, 1 port is considered. Such ports are aggregated and switched to a group of output ports (e.g. 10 Gb/s) that are the ingress port of the optical switch.

There can be M ports between the TDM switch and the optical switch. The optical port can have N external ports with $N \neq M$. This is to consider the case where some optical wavelengths pass-through the optical switch without terminating in the TDM switch. Therefore, the cross-connection between TDM and optical switch can be expressed as <portX, type> - <portY, type>, where port/type identifies the termination point (TP) and type (e.g. TDM or optical switch)

The RPCs that composes the TDM interface are:

- connect(tpFrom, tpTo);
- disconnect(tpFrom, tpTo);
- protect(tpFrom, tpTo(w), tpProt(p));
- unprotect(tpFrom, tpTo);

The RPCs that compose the optical interface are:

- connect(tpFrom, tpTo);
- disconnect(tpFrom, tpTo);
- protect(tpFrom, tpTo(w), tpProt(p));
- unprotect(tpFrom, tpTo);

The other configuration parameter is the enabling of Forward Error Correction (FEC). The use of FEC is optional according the level of performance (e.g. BER) and the characteristics of the transmission channel (e.g. loss, chromatic dispersion, etc.). See the nomenclature of ITU G 698.1 for an example of application code with and without FEC.

7 Fronthaul split

The XFE and XCI are meant to forward and control the mix of both fronthaul and backhaul traffic in the 5G-Crosshaul network architecture. The demonstration of the concepts for XFE and XCI requires the ability to mix fronthaul and backhaul traffic over Ethernet networks. Backhaul traffic was available through the products of several partners but fronthaul traffic over Ethernet wasn't. Therefore, part of the work of WP3 has been evolving the tools and emulators of radio access available in the partner's portfolio to support the desired fronthaul traffic format.

As part of the evolution towards the C-RAN radio, the standard bodies work with number of possible splits (see 3GPP TR 38.801 [56] or eCPRI [29]). Among those the decision was made to focus on two solutions that had high chances to be introduced in the final version of [56], including a higher layer split in Packet Data Convergence Protocol –Radio Link Control (PDCP-RLC) layer (Option 2) and a lower split in MAC-PHY layer (Option 6).

Existing OpenEPC eNodeB software emulators provided an architecture that is possible to adapt to support the two splits and the fronthaul over Ethernet formats. This has permitted to perform experiments and demonstrations of the 5G-Crosshaul network architecture with realistic Crosshaul traffic composed of different types of fronthaul and backhaul connections.

The eNodeB software is a modular architecture in which each layer of transmission is implemented as an independent module that provides an API and communication primitives to other (higher and lower) modules. The eNodeB emulator does not support the physical layer of the LTE radio interface, instead it provides an abstraction layer over Ethernet and re-using IP tools (e.g. DHCP) which permits to connect the Client emulator to run any kind of user plane traffic through the eNodeB and the core network. The UE behavior is implemented as part of the eNodeB emulator which also terminates NAS signaling towards the core network (i.e. the MME). This layered and modular design facilitated implementing C-RAN splits in the eNodeB emulator software.

The eNodeB emulator software is implemented completely in Linux in user space without kernel dependencies, which guaranteed its portability to any kind of hardware platform, but also limited its performance. Since it has only been intended to perform experiments and functional testing it has served its purpose. Part of the 5G-Crosshaul experiments focus in showing relevant performances and measurements of throughputs and latencies in the XFE, requiring work in performance improvement at the eNodeB emulators.

7.1. High Split (PDCP-RLC)

C-RAN architectures with fronthaul over Ethernet already supported a high split between PDCP and RLC layers. This split is simple to implement since it keeps the complexities of the MAC and RLC layers, which are functionality-wise nearer to the radio, in the Remote Unit (RU). The Digital Unit (DU) includes the protocol stack from PDCP on. This approach allows to efficiently utilize available processing power for the air interface encryption and integrity protection processes that can scale based on average load across multiple sites rather than peak utilization of each eNodeB. Figure 52 represents the placement of different protocol layers within network entities.

The fronthaul interface between RU and DU can be carried over Ethernet and IP. The fronthaul protocol in this case is not standardized and there are several commercial vendors known to be using this split with proprietary solutions (e.g. Altiostar [57]).

The split between PDCP and RLC produces a fronthaul interface with a traffic profile similar to that of backhaul, since there is no significant difference between backhaul and fronthaul traffic characteristics when using this split.

The high split required adaptations of the eNodeB emulator software splitting it into two independent network functions and introducing an intermediate module in each of the functions that would implement the fronthaul protocol. Due to a lack of a standardized solution for this split, the selected fronthaul protocol is proprietary. The fronthaul protocol implemented a simple encapsulation of IP packages with an additional header indicating packet type, direction etc.

Both the theoretical analysis and the experimentation permitted to evaluate the overhead of the fronthaul protocol of approximately 1% from UE input to fronthaul packets and of 0.6% from UE input to backhaul packets. This corroborates the similarity of the fronthaul to backhaul traffic profiles.

When testing the performance of the eNodeB emulator it was necessary to perform simple optimizations to achieve throughputs which were in line with the aims in the project. The project purposes were satisfied with a downlink throughput of around 100Mbps, still far away from those of 5G New Radio. To achieve this performance it has not been necessary to modify substantially the architecture and requirements of eNodeB emulator which can still be used in Linux, in user space, without kernel module dependencies and virtualized.

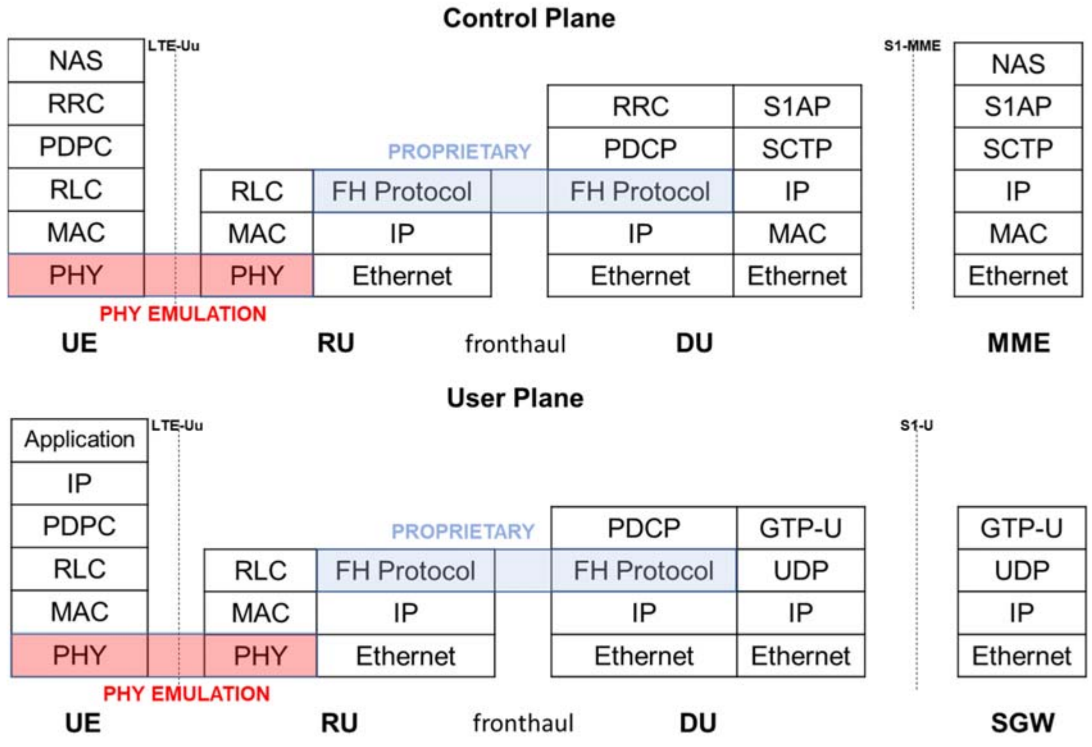


Figure 52: The protocol stacks of the High Split

7.2. Low Split (MAC-PHY)

The standardization of C-RAN has targeted a split of the eNodeB within the MAC layer for its advantages including leaving all functionality which can be done in hardware at the RU and all the software in the DU. Small Cell Forum already addressed this split in their initial studies about Small Cell Virtualization (SCF106 [58] and SCF159 [59]). Besides the work of Small Cell Forum it is also one of the splits proposed in the work of 3GPP (Option 6 in TR 38.801). This split is also considered interesting because it is nearer to the legacy CPRI split but higher in the protocol stack permitting relaxing the stringent requirements in the fronthaul which are characteristic of CPRI, and also importantly, allowing to scale the fronthaul traffic depending on the momentary user traffic.

The fronthaul when implementing the lower split can be transported over IP and Ethernet but with a traffic profile different from backhaul. Unlike the fronthaul in the higher split, for the split between PHY and MAC there is a standard from Small Cell Forum which can be followed and is called FAPI (SCF082 [60]) and more recently nFAPI. There are other standards (e.g. eCPRI [29], RoF, NGFI[8]) but nFAPI suited particularly well the requirements and objectives.

FAPI is an API which was meant for use internally in small cells. nFAPI is meant to be a C-RAN fronthaul protocol transported over IP and Ethernet. The availability of a

[8] https://standards.ieee.org/develop/wg/NGFI.html

standard permitted us to implement RUs and DUs matching best practice for the lower split.

nFAPI provides a set of primitives and message formats that while initially covering the small cells use case can be applied to any C-RAN eNodeB implementation. One of the resulting advantages of this lower split is that there are no proprietary interfaces anymore. Only the UE/RU emulation still is based on a software only setup that permits the signaling and data transfer but doesn’t implement the LTE radio air interface.

The eNodeB software emulation already included a basic stub of FAPI implementation meant to integrate with Software Defined Radio (SDR) boards available in the market (e.g. from Octasic [61]). Within the project the software has been completed to support the nFAPI layer both for RU emulator and DU network function. The extensions have been focused in implementing the basic attachment and data transfer scenarios required for demonstration and experimentation with XFE and XCI, but not the complete functionality of a C-RAN product.

The use of nFAPI as fronthaul protocol, as depicted in Figure 53, resulted in a protocol overhead from UE input to fronthaul packet of 2%, double that of the higher split.

The lower the split, the more stringent the software performance requirements; at the RU emulation the packet processing times and latencies are critical to comply with LTE-Uu timing requirements. The eNodeB was initially not architected for this split and therefore significant effort was required to adapt it and partially rearchitect it.

One of the fundamental changes in the eNodeB RU emulator with the lower split is the requirement of using a version of Linux Ubuntu 16.04 optimized for real-time operations[9]. The RU cannot longer run virtualized and will run on bare-metal. Along with that, improving the processing latency required a complete code review to identify areas for optimization looking at operations which were redundant and optimizations. The optimizations – especially extracting time-critical processing from main message pipeline and executing it on a dedicated core, reducing the number of memory allocations and deallocations, and moving lock-heavy operations to the non time-critical part of the code – permit to achieve the 100Mbps throughputs and low latencies on a host with a quadcore Intel processor operating at 2.6GHz.

[9] https://gitlab.eurecom.fr/oai/openairinterface5g/wikis/OpenAirKernelMainSetup/

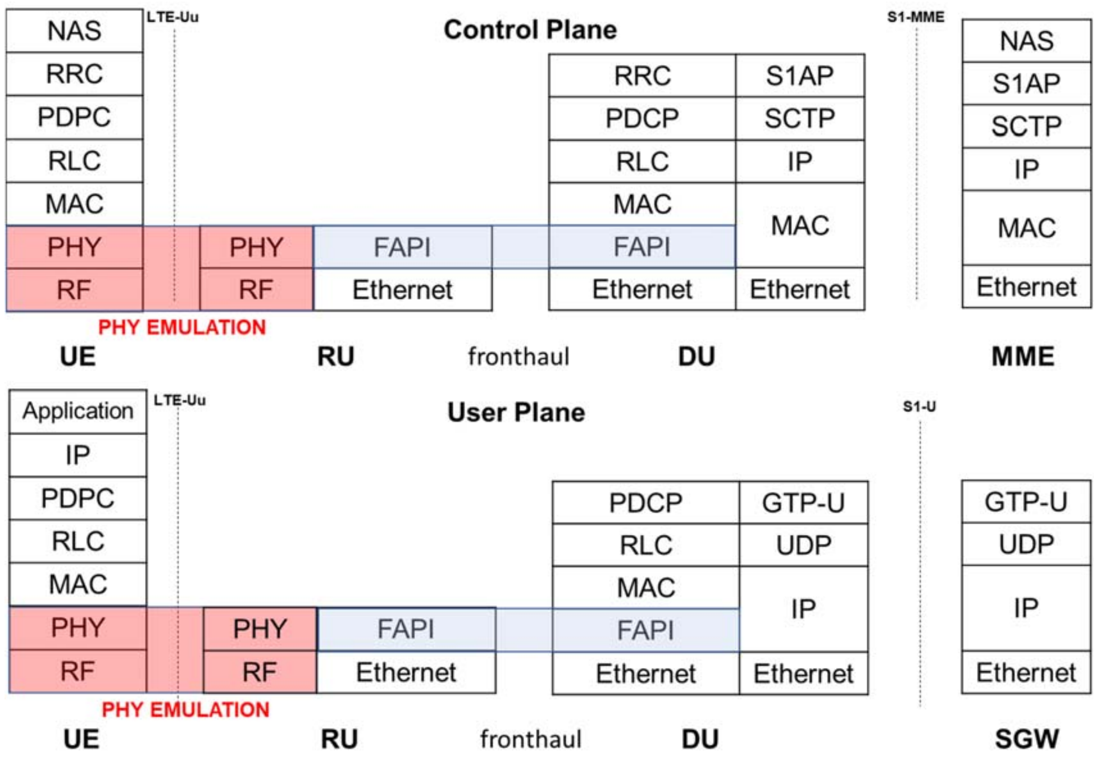


Figure 53: The protocol stacks of the low split

8 Project KPIs

A number of objectives and KPIs have been defined to determine whether 5G-Crosshaul has achieved its targets. In this section, we describe which of the KPIs are relevant for WP3, how they are measured and summarize the measurements.

8.1 Obj1: Design of the 5G-Crosshaul Control Infrastructure (XCI)

WP3 extended existing SDN controllers and introduced new mechanisms to abstract 5G-Crosshaul transport networks and to aggregate measured contextual information.

Table 13: Objective1 and KPIs within WP3

5GPPP KPI Impact	Benchmark	Measurements
Increase the number of connected devices per area by at least a factor of 10.	Limited number of devices due to separate and manual management of each technological domain.	Path Setup Time, Path restoration time after link and node failure.
Energy efficiency improvement by at least a factor of 3.	Path provisioning and VM deployment without energy consumption considerations, network elements are always on.	Power consumption of physical network nodes (XPFEs and XPUs).

The impact on the number of connected devices is achieved indirectly by the XCI. The XCI uses standard APIs to automatically control the network nodes, this automation allows to deploy nodes at a higher density. This allows deploying denser access networks in a cost-efficient manner. Controlling multiple technological domains allows this densification even if different data link technologies are used. In turn, the denser access network allows more devices to connect to the network. Here, the relevant metrics are path setup and restoration times. Other applications, e.g. to reduce interference, and their impact on this KPI are described in WP4.

Hierarchical control in the XCI: As detailed in D5.2 [20], measured path setup times in both wireless and optical domains confirm the advantage of a hierarchical orchestration model to cope with the higher densification of deployed data plane nodes. Overall, these results exhibited an average of 3,971 seconds from the point of view of the RMA application, located on top of the SDN controller, of automated E2E path setup delay and 3.349 seconds from the point of view of the parent SDN controller, hence radically contributing to the target of lowering the multi-domain service deployment time amongst different devices located in different administrative domains. Updated numbers will be provided in D5.2 [20]. The key behind these results is the automation amongst SDN applications, parent, and child SDN controllers in contraposition with the manual procedures that need to be conducted in current deployments which may take this path setup time establishment to days. In this case, we have shown a decrease in service creation time from days to seconds.

The automation of path restoration after links and/or network failures is also subject to more details in evaluation in D5.2 [20]. In these measurements, we focus our attention on a mmWave/WiFi data plane domain and evaluate the time needed to restore an established path that suffers from wireless link failures. Path restoration times were below 300ms (below 120ms on average) under an unreliable wireless control plane (based on WiFi), and below 10ms with a reliable wired control plane. Again, the key behind these results is the automated control plane logic in the child SDN controllers so that not all control plane decisions are taken by the parent SDN controller (e.g., a control plane logic decision affecting the wireless domain might be managed by the wireless SDN controller). This operating mode would require the notification of these decisions to the upper SDN control layers.

Support for energy management in the XCI: The contribution to energy efficiency provided by the XCI, combined with the EMMA application, has been evaluated in different scenarios using both emulated networks and a real test-bed. The analysis of the results in emulated scenarios with reference topologies are provided in D4.2 [55], including a comparison with the state of art of alternative energy-efficient solutions described in literature.

The technical feasibility of the proposed solution has been verified in the 5G-Crosshaul test-bed, deploying prototypes of XCI and EMMA application over physical networks based on RoF, XPFEs and mmWave technologies. The related results will be presented in D5.2 [20]. Major savings in energy efficiency are achieved in scenarios with discontinued traffic (e.g. in the high-speed train scenario). Moreover, the additional procedures to modify dynamically the status of the network nodes and servers (starting from an idle condition) is in the order of tens of milliseconds. This generates a minimum impact on the total provisioning time of a virtual service, which is in the order of few minutes and it is mainly influenced by the time to instantiate and configure the VNFs.

Finally, to verify the scalability of the system and its applicability to realistic environments, the XCI and EMMA application prototype have been also tested over a reference network topology provided by a network operator in the consortium and representing the real regional network deployed in an area in the North-West of Italy. The topology has a total of 51 network nodes and 61 links, with 4 BBUs, two of them acting also as gateway.

The XCI has been tested to establish a number of bi-directional network paths from the edge nodes to the gateways and evaluating the total number of nodes that can be maintained in idle mode when all the flows of the expected traffic matrix are active. In particular, the XCI has established on-demand a dedicated path for the traffic generated by each of the 1497 antennas attached to edge nodes of the network. The fronthaul traffic bandwidth for each flow is based on the values provided by the network operator and varies from 0,35 Mbit/s to 473,54 Mbit/s. As shown in Figure 54, a total of 6 nodes can be maintained in idle mode (grey switches) out of 51 (around 12%). It should be considered that this value refers to the actual contribution of the EMMA algorithms at control/application plane. As analysed in WP1, to obtain a reasonable estimation of the total energy savings reachable with the deployment of the 5G-Crosshaul solution, this value must be added to the savings introduced by 5G-Crosshaul data plane technologies and by the adoption of multi-tenancy, which are in the order of 70%.

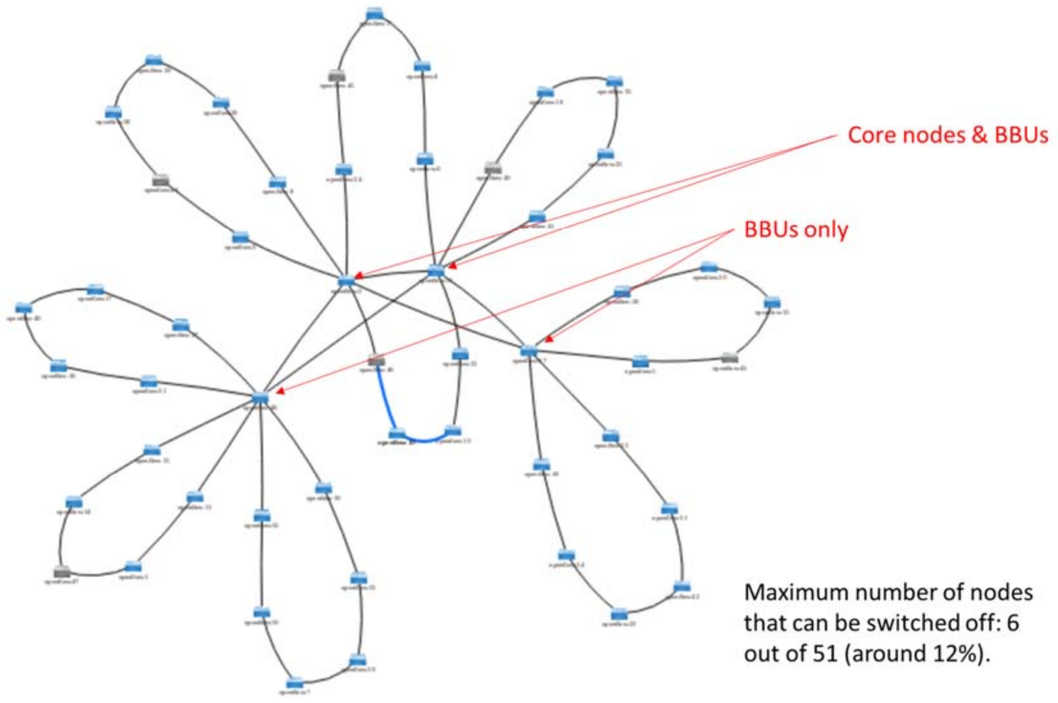

Core nodes & BBUs
BBUs only
Maximum number of nodes that can be switched off: 6 out of 51 (around 12%).

Figure 54: Active (blue) vs. idle (grey) networks nodes in a realistic regional network with full traffic matrix active

8.2 Obj2: Specify the XCI's northbound (NBI) and southbound (SBI) interfaces

WP3 defined interfaces to accelerate the integration of new physical technologies (SBI) and the introduction of new services (NBI). Instead of deploying new services manually, they can be deployed automatically and their deployment can be greatly reduced.

Table 14: Objective2 and KPIs within WP3

5GPPP KPI Impact	Benchmark	Measurements
Enable the introduction or provisioning of new services in the order of magnitude of hours (e.g., VPNs, network slices).	Manual deployment of the service, which can take months [52], [53].	service provisioning or deployment times of e.g. paths (VPNs), nodes (vEPC nodes), or services (e.g. CDN).

The measured time is the actual deployment time. The time to define a service, e.g. its business logic, is not included in the measurement.

Path provisioning for VNFs: The experimental activities in WP5 have evaluated the provisioning time for network connections (both in single and multi-domain scenarios) and Network Services with instantiation and configuration of VNFs over real test-beds deploying 5G-Crosshaul data plane technologies and software prototypes, including the

XCI components. These experiments, which will be detailed in D5.2 [20], show a provisioning time of few seconds for intra- and inter-domain network connections and few minutes for vEPC NSs with up to 4 VNFs (mostly caused by the time required by VM instantiation and booting).

The scalability of the system in terms of provisioning time for network connections has been tested using an emulated network with the topology of the reference regional network of the previous section (see Figure 54) and with the XCI SDN controller running in a VM with 4 vCPUs and 16 GB RAM. Requests for bi-directional connections between each of the 1497 antennas and the two gateways have been issued with different request rates, to evaluate how these rates impact the provisioning and deletion time in a network with a realistic dimension and topology. Each test has been repeated 20 times and the results about average, minimum and maximum time are shown in Figure 55 for provisioning and Figure 56 for deletion. In the considered range of requests' rates, the system has always been able to process and executes all the requests. For low rates (up to 1,3 requests per second) the provisioning time is in the order of 60 ms, while it increases to around 80 ms and 100 ms for rates of 2 and 2.5 requests per seconds. After these rates, the provisioning time raises quickly up to an average value of 500 ms with peaks of 850 ms for a rate of 6,7 requests per second, associated with an increasing value of the variance. Analysing in detail the components of the provisioning time, we can see that the major delay is introduced by the path computation, which takes usually 98-99% of the total provisioning time. This fact can be easily explained with the fact that in the emulated environment both the exchange of OpenFlow messages and the configuration of the virtual switches are performed locally in the machine and they introduce a minimum delay.

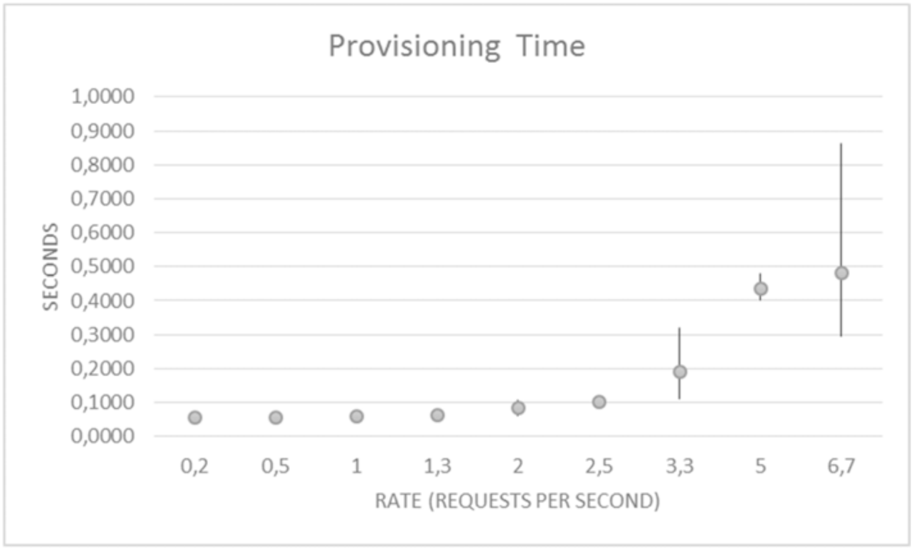


Figure 55: Provisioning time of a network connection

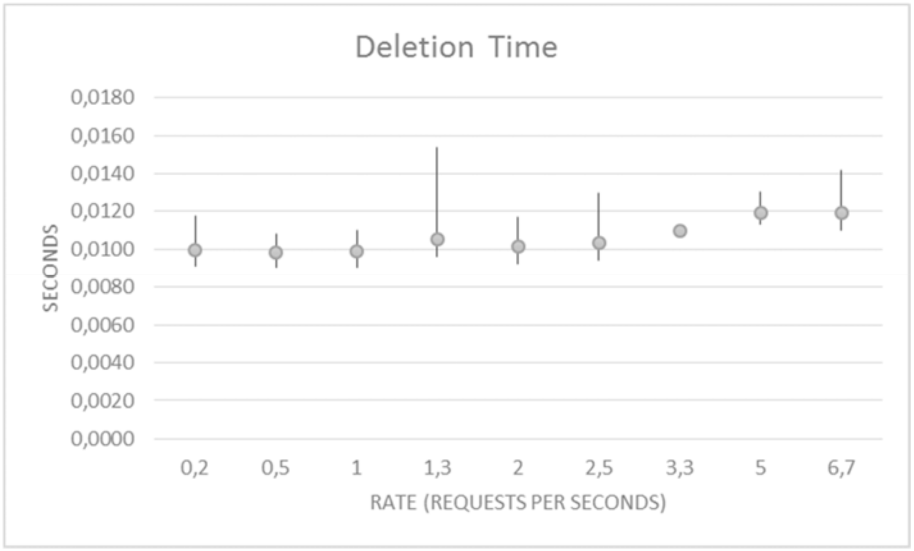


Figure 56: Deletion time of a network connection

Service deployment in CDN networks: Regarding services, such as a CDN, the integration and orchestration exploiting the capacities of the XCI allow the configuration of these services through VNFs. This would allow the application layer the automatic deployment of a complete and configured service (servers, network configurations, connectivity among the elements which build the service) in a virtualized environment, avoiding the manual configuration and instantiation of every component as hardware appliances. Therefore, the deployment time will be significantly reduced. Detailed measurements are reported in D5.2 [20].

End-to-end path provisioning support: The end-to-end path provisioning time from Application plane (Resource Management Application) to data plane via the XCI has been evaluated in a multi-domain environment in the context of WP5, illustrated in Figure 57. In the application plane, the RMA computes optimal routes between endpoints in the multi-domain data plane relying on a graph-based abstracted view of the underlying topology, provided by the XCI. The XCI SDN component is hierarchical, with a parent ABNO controller orchestrating one SDN controller per domain. Each controller uses a different protocol for inventory, management and monitoring of the respective hardware (e.g., a REST-based interface in one of the mmWave domains, and a proprietary protocol in the other one). To expose this multi-domain information in a homogeneous manner to the RMA, the parent controller interacts with the child controllers via COP, and so does the RMA with the parent controller. The RMA is deployed at a different site than the controllers and the data plane, the RTT among the two sites is about 60ms. The child controllers can make local decisions at a shorter time-scale than the RMA and the parent controller. For instance, the mmWave mesh controller can configure local paths within the mmWave mesh to use in case of failures and so ensure connectivity while RMA calculates and triggers the request for a new path provisioning.

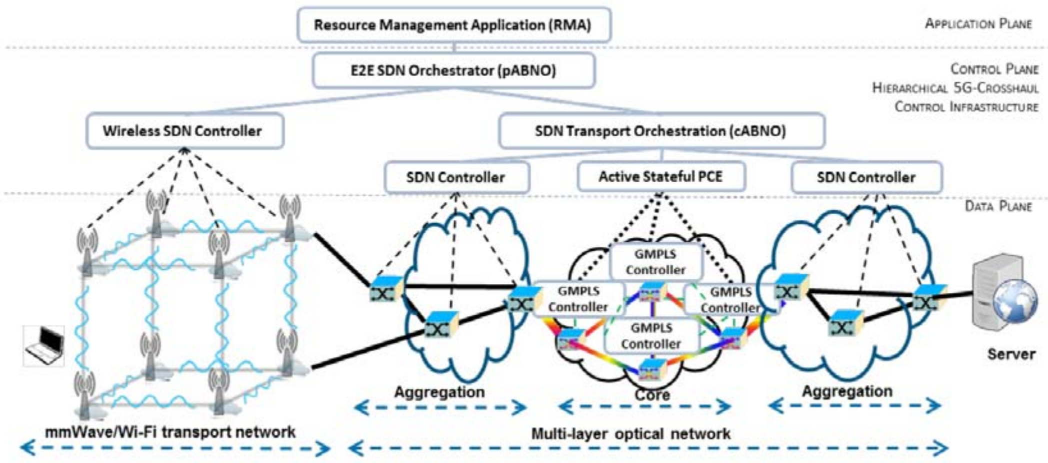


Figure 57: Experimental setup of an multi-domain data plane.

From the point of view of the parent ABNO, Figure 58 shows the histogram and CDF for the end-to-end setup delay, from the reception of the request in the ABNO NBI to the completion of the operations. The multiple peaks in the histogram are caused by the ABNO (child and parent), which process received requests every 50ms. The average setup delay increases from tens or hundreds of milliseconds in the wireless and single-layer optical domains as seen from the child controllers to seconds as seen from the parent. This is due to various factors. First, an average of 2,867s are spent in the multi-layer optical network. Second, there is the interaction and message processing between the parent ABNO and child controllers. Third, there is the sequential handling of some of the messages to set up the E2E path. As far as the application plane is concerned, Table 15 compares some statistics of the setup delay as seen from the parent ABNO and from the RMA. Recall that that the RMA and parent controller are deployed on different sites with a RTT of 60ms. However, the difference is 600ms approximately, which is due to the processing carried out at the RMA.

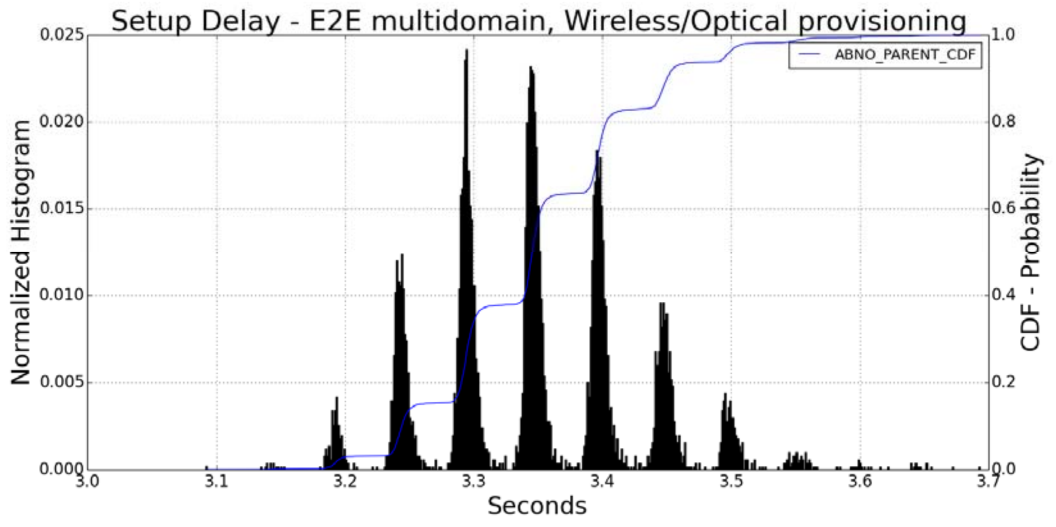


Figure 58: Histogram and CDF of the setup delay seen by the ABNO

Table 15: End-to-End Path setup Delay in a multi-domain environment.

	Average	Min.	25-percentile	75-percentile	Max.
pABNO (s.)	3,349	3,092	3.294	3.398	3.693
RMA (s.)	3.971	3.667	3.804	4.046	5.281

8.3 Obj3: Unify the 5G-Crosshaul Data Plane

WP3 – jointly with WP2 – developed the XFE and XCF as a common and flexible frame format to carry both fronthaul and backhaul traffic through the network.

Table 16: Objective3 and KPIs within WP3

5GPPP KPI Impact	Benchmark	Measurements
CAPEX and OPEX savings due to the unified data plane (25%) and multi-tenancy (>80%, depending on the number of tenants).	Latency and jitter of hardware switches.	Data plane throughput, data plane latency/jitter.
80% increased energy efficiency due to consolidation of equipment.	Energy consumption of existing networks (equipment is always on).	Power consumption is determined by a tool evaluating power consumption on a network wide level.

CAPEX and OPEX of XFEs are not measured within WP3. This is compared in the cost model of D1.2 [54] against the costs of existing hardware switches. This cost model covers energy savings as well.

It can be shown that multiple tenants are supported by the XCI and the XFEs, but its impact cannot be measured directly. The multi-tenancy feature is included in the cost model described in D1.2 [54]. The measurements of throughput, latency/jitter indicate whether the more flexible design of switches and frame format to support multi-tenancy are still able to satisfy the requirements of fronthaul and backhaul traffic.

Latency depending on load: For a lightly loaded switch, there is no significant difference in latency and jitter among traffic streams with different priority. As the load increases, latency and jitter of low-priority traffic increase significantly and will be dropped first in case of overload. The latency per hop is below 10µs for 10G interfaces. Although this is significantly higher than the latency of a hardware switch, this is still sufficiently small for fronthaul traffic. Detailed numbers will be provided in D5.2 [20].

8.4 Obj4: Develop physical and link-layer technologies to support 5G requirements

Link layer technologies have been investigated in WP2. L2 switching technologies have been investigated in WP3.

Table 17: Objective4 and KPIs within WP3

5GPPP KPI Impact	Benchmark	Measurements
Latency of < 1ms between 5G Point of Attachment (PoA) and mobile core.	Latency and jitter of hardware switches (E.g. less than 400ns, [31]).	Data plane throughput, data plane latency/jitter.

The measurements of throughput, latency/jitter indicated whether the more flexible devices are still able to satisfy the requirements of fronthaul and backhaul traffic. The measurement of data plane throughput is not needed to measure the KPI, but is used to validate that latency requirements are achieved without increasing packet loss rate.

Data plane throughput, latency, and jitter are described already for Objective 3 in Section 8.3.

8.5 Obj5: Increase cost-effectiveness of transport technologies for ultra-dense access networks

This objective is not relevant for WP3.

8.6 Obj6: Design scalable algorithms for efficient 5G-Crosshaul resource orchestration

WP3 has developed the XCI, including an algorithm for network optimization. The relevant aspect here is the scalability of the algorithms, being able to handle networks with 10 times more nodes.

Table 18 : Objective6 and KPIs within WP3

5GPPP KPI Impact	Benchmark	Measurements
Scalable management framework: algorithms that can support 10 times increased node densities.	Separate management of FH and BH networks in metropolitan areas.	CPU profiling of the control subsystem.

Resource consumption, especially CPU usage, for reference configurations is measured to show that the XCI scales up. Topology and size of reference configurations are based on the topologies described in D1.2 [54].

Further resource management and TE algorithms as well as techniques for path provisioning have been developed in WP4, their evaluation on the scalability is shown in D4.2 [55].

8.7 Obj7: Design essential Crosshaul-integrated (control/planning) algorithms

WP3, jointly with WP4, has developed applications to reduce energy consumption in the 5G Crosshaul by 30% through energy management. In the XCI, the control of optimal scheduling of equipment sleep cycles, routing and function placement is the key functional enabler for the Energy Monitoring and Management Application. The corresponding KPI – Design essential 5G-Crosshaul-integrated (control/planning) applications – is described in D4.2 [55].

8.8 Obj8: 5G-Crosshaul key concept validation and proof of concept

WP3 contributed to the joint demonstrations in the testbed providing XCI and XFE components. In general, the relevant KPIs related to these demonstrations are described in WP5. Specifically, WP3 developed self-healing mechanisms. Algorithms for resource orchestration based on traffic load and energy-aware optimization and reconfiguration are part of the applications and are described in WP4.

Table 19: Objective8 and KPIs within WP3

5GPPP KPI Impact	**Benchmark**	**Measurements**
Self-healing mechanisms for unexpected 5G-Crosshaul link failures through alternative path routing in mesh topologies.	Separate out-of-band wired control plane.	Path restoration times for link and node failures.

It has been measured whether the XCI designed in 5G-Crosshaul is able to, within an acceptable time (i.e., a few seconds), maintain the backhaul/fronthaul path amongst two end points after a failure in the current established path. Both wired (benchmark) and wireless control channels are compared.

Our findings reveal that, due to the self-healing mechanisms embedded in the control logic of the XCI and regardless of the control plane medium used, the designed XCI is able to restore data plane paths that have been affected by failures. In particular, the exhibited distribution of data plane restoration times is within the order of hundreds of ms (i.e., less than 300ms) whereas results with a wired control plane (i.e., a separate out-of-band control plane) were below 10ms.

The aforementioned results reported are independent of the kind of node/link failure. We obtain these results by evaluating different data plane failure events (e.g., a mmWave data plane node failure or a WiFi data plane node failure). More details about this set of measurements can be found in D5.2 [20].

The path restoration times are not excessively higher than those obtained with a separate wired out-of-band control plane. Results obtained with a wireless control plane are higher because of the unreliability of the wireless control plane. It is important to note that the deployment cost with a wireless control plane is significantly lower than that of an out-of-band reliable wired control plane.

On the other hand, as the XCI spans across larger parts of the network and therefore multiple technology (or per-vendor) domains, the overall network recovery will be substantially improved compared to out-of-band control of isolated technological domains due to the necessity of manually restoring the multi-domain paths. When path restoration times are within the same order or even better, the overall self-healing mechanism will provide a more optimal network recovery solution as it spans across a larger part of the network than isolated technical domains.

8.9 Use of developed components in demonstrations

The components developed in WP3 have been evaluated in the demonstrations in WP5. The usage of the components in the experiments, see Table 20, as described in D5.2 [20] is summarized in Table 21 and Table 22. The NBI and SBI definitions and the XCF are used implicitly, they are not listed here.

Table 20: List of Experiments in D5.2 [20]

Number	Name
1	Power Consumption Monitoring for XPFE physical nodes
2	Power consumption monitoring for single network paths and tenants
3	Energy-oriented network resource management in RoF domains
4	Energy-oriented network resource management in XPFE domains
5	Energy-oriented virtual infrastructure management
6	Energy-oriented network resource management in XPFE domains for on-demand provisioning of connections dedicated to fronthaul and backhaul traffic
7	EMMA resource management over mmWave mesh
8	Virtual CDN service on the 5G-Crosshaul infrastructure
9	Multicast TV service provisioning
10	Path reconfiguration when QoS degradation
11	Distribution of live content through the vCDN infrastructure on the 5G-Crosshaul infrastructure
12	Assessment of the SDN-based control of the Optical Transport Network (optical domain)
13	Assessment of the SDN-based control and data plane for the mmWave/Wi-Fi mesh domain
14	End-to-end characterization. Network Orchestration across multiple heterogeneous domains: control and data plane characterization
15	Backhaul and Fronthaul services integration through an SDN WS-WDM-PON transport network and XPFEs + Radio-over-Fibre
16	Evaluation of mixed Digital/Analogue Radio-over-Fibre
17	Integration of XCSE, XPFE, mmWave and CPRI compression data plane solutions
18	Evaluation of integrated packet-based FH/BH combining wired XPFEs and hybrid mmWave/optical wireless links

Table 21: Use of XCI components in experiments

Component	**Section**	**Experiment**
NFVO	3.2.1	5, 8, 9, 10, 11
VNFM	3.2.1	5, 8, 9, 10, 11
VIMaP	3.2.1	5, 9, 10, 11, 12
SDN controller	3.2.3	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
SDN controller/SBI driver	3.2.3	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
SDN controller/network core services	3.2.3	1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14
SDN controller/path computation	3.2.3	4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
SDN controller/path provisioning	3.2.3	2, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14
SDN controller/network reconfiguration	3.2.3	4, 5, 6, 7, 10, 12, 13, 14
SDN controller/analytics for monitoring	3.2.3	1, 2, 4, 5, 6
SDN controller/ Parent SDN controller	3.2.3, 3.3	12, 13, 14
SDN controller/analytical algorithms	4.1	5

Table 22: Use of dataplane components in experiments

Component	**Section**	**Experiment**
XPFE (enhanced Lagopus)	5.2.2.3	6, 8, 9, 10, 11, 15, 17, 18
XPFE Flow pipeline	6.3	6, 8, 9, 10, 11, 15, 17, 18
eNb with different split options	7	6, 14, 15, 18

9 Conclusions

In this deliverable, we have presented the consolidated design of the 5G-Crosshaul control plane platform (i.e. XCI) together with the design of a NBI to the applications plane and a SBI to the data plane. Further, we have provided the data plane design, particularly the design of packet switching elements (XPFEs).

We have provided the design of NBI APIs exposed by several XCI services towards the 5G-Crosshaul applications. More specifically, we have provided the API design for each NBI service, the most relevant information of each respective data model, and a workflow to illustrate the use of each service by a generic 5G-Crosshaul application or by an internal module inside the XCI that, in its turn, can expose an NBI. The NBI APIs designed in WP3 reflect the design agreements with WP4, where the requirements of 5G-Crosshaul applications have been detailed.

Furthermore, we presented the XCI design, which is in line with the 5G-Crosshaul System Architecture in D1.2. Essentially, we provided detailed deployment aspects including the selection of software components inside the XCI and possible software frameworks and platforms to be used for their implementation, along with a detailed discussion of different deployment models of the XCI (focusing on deployment and interconnection models of the SDN controllers). We further presented the details of the interaction among SDN controllers, MANO components and between child and parent controller. The interaction shows that our XCI design is a suitable enabler for modularized and independent implementations.

The algorithms in the XCI use different underlying models. We described power consumption models for virtual machines and containers. An optimization model for the forwarding decisions is described as well.

In this document, the consolidated design of the data plane based on XFE has been defined, in particular, the XPFE and the XCF. MAC-in-MAC is chosen has the baseline technology to design the XCF. Synchronization aspects for packet based networks and OAM for 5G-Crosshaul network have been further explored in this document.

The document also focuses on the interaction between XCI and the data plane at the SBI. The OpenFlow protocol has been chosen as the SBI candidate for controlling the data-plane forwarding. We presented an analysis on how OpenFlow 1.5.1 specifications can fulfill the XFE, Adaptation Function (AF) and the XCF design requirements, and a XPFE OpenFlow pipeline. This is complemented by a description of procedures to integrate XPFE nodes into an existing network.

We described implementations of different splits of the protocol stack, implying different requirements regarding latency and jitter for the fronthaul traffic.

Finally, the report has detailed the KPIs relevant for WP3 and their evaluation.

10 Bibliography

[1] 5G-Crosshaul, Deliverable D3.1, XFE/XCI design at year 1, specification of southbound and northbound interface

[2] 5G-Crosshaul, Deliverable D1.1, 5G-Crosshaul initial system design, use cases and requirements

[3] ETSI GS NFV 003 V1.2.0 (2014-11), Network Functions Virtualization (NFV); Terminology for Main Concepts in NFV. European Telecommunications Standards Institute.

[4] SDN Architecture 1.0 (June 2014 | TR-502). Open Network Foundation.

[5] 5G-Crosshaul, Deliverable D2.1, Study and assessment of physical and link layer technologies for 5G-Crosshaul

[6] OpenDaylight [Online]. Available: http://www.opendaylight.org

[7] ONOS [Online]. Available: http://onosproject.org

[8] OpenStack [Online]. Available: http://www.openstack.org

[9] WebSocket protocol. https://tools.ietf.org/html/rfc6455

[10] RESTCONF protocol. https://tools.ietf.org/html/draft-ietf-netconf-restconf-12

[11] 5G-Crosshaul, Deliverable D4.1, Initial design of 5G-Crosshaul Applications and Algorithms

[12] OpenStack Ceilometer [Online]. Available at http://docs.openstack.org/developer/ceilometer/

[13] ETSI NFV Use Cases [Online]. Available at. http://www.etsi.org/deliver/etsi_gs/nfv/001_099/.../gs_nfv001v010101p.pdf

[14] ETSI GS NFV IFA 010, v2.1.1, Network Functions Virtualization (NFV); Management and Orchestration; Functional Requirements Specification.

[15] Open Baton [Online]. Available at http://openbaton.github.io

[16] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer and X. Hesselbach, "Virtual Network Embedding: A Survey," in IEEE Communications Surveys & Tutorials, vol. 15, no. 4, pp. 1888-1906, 2013.

[17] IETF Abstraction and Control of Transport Networks BoF. https://sites.google.com/site/actnbof/

[18] Ryu controller [Online]. Available at https://osrg.github.io/ryu/

[19] Floodlight [Online]. Available at http://www.projectfloodlight.org/floodlight/

[20] 5G_Crosshaul, Deliverable 5.2, Report on experimentation results and proof of concept (under preparation)

[21] S. Tadesse, C.F. Chiasserini, F. Malandrino, "Energy Consumption Measurements in Docker", IEEE Computers, Software, and Applications Conference (COMPSAC 2017), Turin (Italy), July 2017

[22] Standard Performance Evaluation Corporation, SPEC CPU 2006, https://www.spec.org/cpu2006/

[23] NGMN, LTE backhauling deployment scenarios, white paper, 2011

[24] IEEE 1904.3, Draft Standard Radio over Ethernet Encapsulations and Mappings

[25] IEEE Standards Association, Bridges and Bridged Networks, IEEE Std 802.1Q™-2014

[26] Small Cell Forum, Small Cell Virtualization Functional, Splits and Use Cases, document 159.05.1.01, June 2015

[27] Lagopus [Online]. Available at http://www.lagopus.org

[28] Dataplane Development Kit [Online]. Available at http://dpdk.org/

[29] eCPRI 1.0 Specification, [Online], Available at http://www.cpri.info/spec.html

[30] IEEE LAN/MAN standards committee, Specification and Management Parameters for Interspersing Express Traffic, IEEE 802.3br

[31] 5G-Crosshaul, Deliverable D2.2, Integration of physical and link layer technologies in 5G-Crosshaul network nodes.

[32] Open Networking Foundation (ONF), "OpenFlow Switch Specification," March 2015. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf

[33] Sachin Sharma, Dimitri Staessens, Didier Colle, Mario Pickavet, and Piet Demeester, In-Band Control, Queuing, and Failure Recovery Functionalities for OpenFlow, IEEE Network Jan/Feb 2016

[34] Open Networking Foundation (ONF), OF-Config 1.2, OpenFlow Management and Configuration Protocol, ONF TS-016, [Online]. Available: https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2013/02/of-config-1.2.pdf

[35] NETCONF protocol, https://tools.ietf.org/html/rfc6241

[36] D. Samociuk: Secure Communication Between OpenFlow Switches and Controllers; AFIN 2015: The 7th Intl. Conf. on Advances in Future Internet https://www.thinkmind.org/download.php?articleid=afin_2015_2_30_40047

[37] R. Santos, A. Kassler, A SDN Controller Architecture for Small Cell Wireless Backhaul using a LTE Control Channel, 2016 IEEE 17th Intl. Symp. on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)

[38] Katiyar, R., Pawar, P., Gupta, A., and Kataoka, K.: Auto-Configuration of SDN Switches in SDN/Non-SDN Hybrid Network. In Proc. of the Asian Internet Engineering Conf. (2015), ACM, pp. 48–53

[39] 3GPP TS 33.310, Network Domain Security (NDS); Authentication Framework (AF) https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2293

[40] Internet X.509 Public Key Infrastructure Certificate Management Protocol. RFC 4210, https://tools.ietf.org/html/rfc4210

[41] IEEE 802.11ad, http://www.ieee802.org/11/Reports/tgad_update.htm

[42] IEEE Standards Association, "Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE 1588," 2008

[43] IEEE Standards Association, "Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks. IEEE 802.1AS," 2012
[44] J.-L. Ferrant, et al, “Development of the First IEEE 1588 Telecom Profile to Address Mobile Backhaul Needs”, IEEE Communications Magazine, pp. 118-126, October 2010.
[45] J. Paulo, I. Freire, I. Sousa, C. Lu, M. Berg, I. Almeida, and A. Klautau, “FPGA-Based Testbed for Synchronization on Ethernet Fronthaul with Phase Noise Measurements”, in Proceedings of Intl. Symp. on Instrumentation Systems, Circuits and Transducers – INSCIT, Sept. 3, 2016.
[46] I. Freire, I. Souza, I. Almeida, C. Lu, M. Berg, A. Klautau, “Analysis and Evaluation of End-to-End PTP Synchronization for Ethernet-Based Fronthaul”, in Proceedings of IEEE Global Communications Conference, Dec. 7, 2016.
[47] IETF, Deterministic Networking (detnet) working group, https://datatracker.ietf.org/wg/detnet/charter
[48] IETF, Source Packet Routing in Networking (spring) working group. https://datatracker.ietf.org/wg/spring/charter
[49] Open Networking Foundation (ONF), OpenFlow Table Type Patterns, Version 1.0, August 2014.
[50] Open Networking Foundation (ONF), MPLS-TP OpenFlow Protocol Extensions for SPTN, Version 0.9, ONF TS-029, January 2017.
[51] IETF RFC 5531, RPC: Remote Procedure Call Protocol Specification Version 2
[52] Affirmed Networks, Service Creation [Online]. Available: http://www.affirmednetworks.com/products-solutions/service-creation/
[53] Cisco, Cisco Network Services Orchestrator Enabled by Tail-f [Online]. Available: https://www.cisco.com/c/dam/en/us/products/collateral/cloud-systems-management/network-services-orchestrator/at-a-glance-c45-734640.pdf
[54] 5G-Crosshaul, Deliverable, D1.2, Final 5G-Crosshaul System Design and Economic Analysis (under preparation)
[55] 5G-Crosshaul, Deliverable D4.2, Final 5G-Crosshaul Applications and Algorithms
[56] 3GPP TR 38.801 v14.0.0, Study on New Radio Access Technology, Radio access architecture and interfaces
[57] Altiostar [Online]. Available: http://www.altiostar.com/news/press-releases/company-launch/
[58] Small Cell Forum, „Virtualization for small cells – Overview” [Online]. Available: http://scf.io/en/documents/106__Virtualization_for_small_cells_Overview.php
[59] Small Cell Forum, “Small cell virtualization functional splits and use cases” [Online]. Available: http://scf.io/en/documents/159_-_Small_Cell_Virtualization_Functional_Splits_and_Use_Cases.php
[60] Small Cell Forum, “nFAPI and FAPI specifications” [Online], http://scf.io/en/documents/082_-_nFAPI_and_FAPI_specifications.php
[61] Octasic Software Defined Radio product [Online]. Available: http://www.octasic.com/product/octbts-3000/

11 Appendix I: Northbound interface design

In this appendix, we provide the details of those APIs, which have been omitted in Section 2 for the sake of brevity.

11.1 Provisioning and Flow Actions

11.1.1 APIs

Table 23 provides the set of Northbound APIs that are available to perform different operations on the flows through an SDN controller (e.g. ONOS, ODL). These interfaces, in the form of REST APIs, allow creating, deleting, and modifying flow rules in physical nodes. In this specific context, nodes are intended as XPFEs. The response to these operations is in XML or JSON format.

Table 23: Provisioning and flow actions API: flow rules in physical devices.

Prot.	**Type**	**URI**	**Parameters**	

REST	POST	*../sdn_ctrl/flows/{node_id}*
		
		or
		
		../sdn_ctrl/flows/{node_id}/{table_id}
		
		Create a new flow rule.
			Input	*node_id*
				
				table_id (optional)
				
				flow_object

			Output	Success: *flow_id*
				Failure: Error code

REST	DELETE	*../sdn_ctrl/flows/{node_id}/{flow_id}*
		
		or
		
		../sdn_ctrl/flows/{node_id}/{table_id}/{flow_id}
		
		Delete an existing flow rule.
			Input	*node_id*
				
				table_id (optional)
				
				flow_id

			Output	Success: Status Code of normal end
				Failure: Error code

REST	GET	*../sdn_ctrl/flows/{node_id}*
		
		or
		
		../sdn_ctrl/flows/{node_id}/{table_id}
		
		Retrieve the list of all flow rules on a
			Input	*table_id*
				
				node_id

			Output	*flow_object* List of flow rules on the specified node, or out of a specific flow table

		specific node or table.		
REST	GET	*../sdn_ctrl/flows/* Retrieve all the flow rules	Input	-
			Output	List of all flow rules
REST	GET	*../sdn_ctrl/statistics/flows/{node_id}/{port_id}/* Retrieve statistics for all flows passing over a port of a node	Input	*node_id* *port_id*
			Output	Aggregate values • Bytes counter • Rate [bytes/s] Time of last statistics collection
REST	GET	*../sdn_ctrl/streams/flows/{event_id}*	Input	*event_id*
			Output	URL to the notification service (e.g. Websocket)
Websockets	SUBSCRIBE	*../sdn_ctrl/streams/flows/{event_id}*	Input	*event_id*
			Output	Success: Status Code of normal end
Websockets	ASYNC	notification event	Input	*event_id*
			Output	*event_object*

11.1.2 Information Model

Description of the parameters used for setting up flow rules in the switching elements are shown below in Table 24. The UML diagram describing the defined information model to provision flow rules in a node (i.e. switching element) is shown in Figure 59.

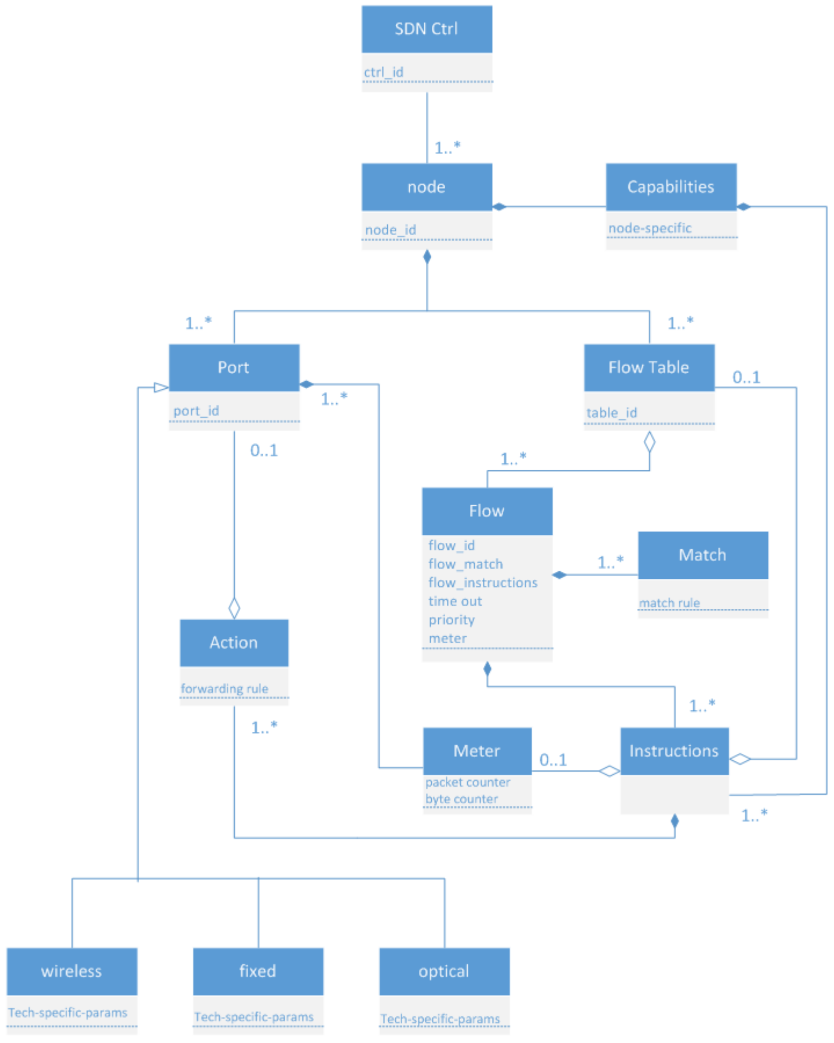


Figure 59: Provisioning and flow actions information model.

The main data objects are presented in more detail:

Table 24: Provisioning and flow actions information model.

Parameters	Type	Description
node_id	String	Identifier of the node
table_id	String	Identifier of the flow table
port_id	String	Identifier of a port
flow_id	String	Identifier of a flow
flow_object	Object	• table_id: string • flow_id: string • flow_match: string denoting the matching rule • flow_instruction: string denoting the instructions.

		• Time out: Integer denoting the time the flow rule exists. • Priority: Integer indicating the priority of a flow rule. • Meter: Integer denoting statistics such as packet and byte counters.
event_id	String	Identifier of a specific event to subscribe.
event_object	Object	Object which contains the set of information specific to the subscribed event.

11.1.3 Workflow

The workflow provided in Figure 60 illustrates the configuration of flow rules (e.g. FlowMod in OpenFlow) in a node. Any 5G-Crosshaul application (e.g. Mobility Management Application (MMA)) or even the VIM, can request the SDN controller to create, delete and modify flow rules in one or more switching elements at the data plane.

The workflow shown in Figure 60 includes the following steps:

1. A consumer application requests the creation of new flow rules in a specific device.
    a. The SDN controller creates new flow rules specifying the matching rules, instructions, priority and time out, just to name a few.
2. A consumer application decides to request the details of a specific flow rule.
    a. The SDN controller returns an object describing the parameters of the flow (i.e. flow identification, flow match, flow actions, flow priority and time out).
3. A consumer application decides to request the statistics for all the flows passing through a specific port.
    a. Upon receiving the request, the SDN controller returns related statistics such as byte counter, rate and the time at which statistics were collected.
4. A consumer application decides to subscribe to the notification of a specific event of interest (e.g. arrival of a new flow). An URL is returned in response.
5. A WebSocket is created in order to subscribe to the notification service for the event of interest.
6. Asynchronous notifications are received for the specific event of interest. An object is returned in response containing the set of information for the subscribed event.
7. A consumer application decides to remove (delete) a specific flow rule from a device.

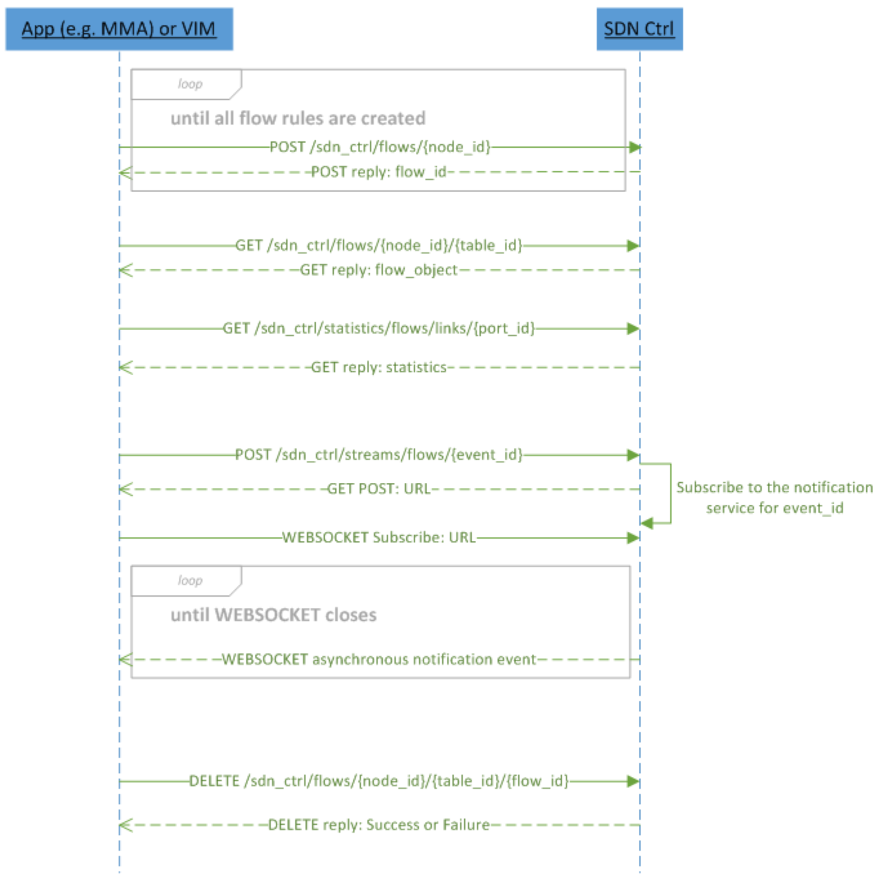

App (e.g. MMA) or VIM
SDN Ctrl
loop
until all flow rules are created
POST /sdn_ctrl/flows/{node_id}
POST reply: flow_id
GET /sdn_ctrl/flows/{node_id}/{table_id}
GET reply: flow_object
GET /sdn_ctrl/statistics/flows/links/{port_id}
GET reply: statistics
POST /sdn_ctrl/streams/flows/{event_id}
GET POST: URL
Subscribe to the notification service for event_id
WEBSOCKET Subscribe: URL
loop
until WEBSOCKET closes
WEBSOCKET asynchronous notification event
DELETE /sdn_ctrl/flows/{node_id}/{table_id}/{flow_id}
DELETE reply: Success or Failure

Figure 60: Flow actions example.

11.2 IT infrastructure and inventory

11.2.1 APIs

In the following, we provide a description of the APIs offered by the IT infrastructure and inventory services.

Table 25: IT infrastructure and inventory API.

Prot.	Type	URI	Parameters	
REST	GET	*../it/vms* *../it/tenant/{tenant_id}/vms* Retrieve all/per_tenant VM elements. *../it/vms?type="type"* *../it/{tenant_id}/vms?type="typ e"* Retrieve all VM elements of	Input	*type* (optional) *tenant_id*
			Output	*vm_list*

		specified type.		
REST	GET	*../it/vm/{vm_id}* Retrieve information of VM with identifier *vm_id*.	Input	*vm_id*
			Output	*vm_object*
REST	POST	*../it/vm* *../it/tenant/{tenant_id}/vm* Create a new VM for *tenant_id*.	Input	*vm_object* *tenant_id*
			Output	*vm_id*

REST	PUT	*../it/vm/{vm_id}*
		../it/tenant/{tenant_id}/vm/{vm _id}
		
		Update information of VM with identifier *vm_id*.
			Input	*vm_id*
				
				tenant_id
				
				vm_object

			Output	Success: Status Code of normal end
				Failure: Error code

REST	DELETE	*../it/vm/{vm_id}*
		../it/tenant/{tenant_id}/vm/{vm _id}
		
		Delete the VM (in *tenant_id*) with identifier *vm_id*.
			Input	*vm_id*
				
				tenant_id

			Output	Success: Status Code of normal end
				Failure: Error code
REST	GET	*../it/tenant/{tenant_id}/os_host* List compute/storage nodes in *tenant_id*	Input	*tenant_id*
			Output	List of compute/storage nodes
REST	GET	*../it/tenant/{tenant_id}/os_host s/{host_name}* List details of *host_name*	Input	*tenant_id* *host_name*
			Output	*host_object*
REST	GET	*../it/tenant/{tenant_id}/os_host s/{host_name}/start* Start a compute node *host_name* in tenant *tenant_id*	Input	*tenant_id* *host_name*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	GET	*../it/tenant/{tenant_id}/os_host s/{host_name}/shutdown*	Input	*tenant_id* *host_name*

		Shutdown a compute node *host_name* in tenant *tenant_id*	Output	Success: Status Code of normal end
				Failure: Error code
REST	GET	*../it/tenant/{tenant_id}/os_hypervisors* Get list of hypervisor in tenant *tenant_id*	Input	*tenant_id*
			Output	*Hypervisor_list*
REST	GET	*../it/tenant/{tenant_id}/os_hypervisors/{hypervisor_id}*	Input	*tenant_id* *hypervisor_id*
			Output	*hypervisor_object*

11.2.2 Information Model

The data model comprises the computing, storage, and network functions associated to a given tenant (or owner/user). Each tenant has associated a set of VMs and can comprehend one or more VNs. Each VN is formed by a set of virtual links defining the connection patterns between each pair of VMs.

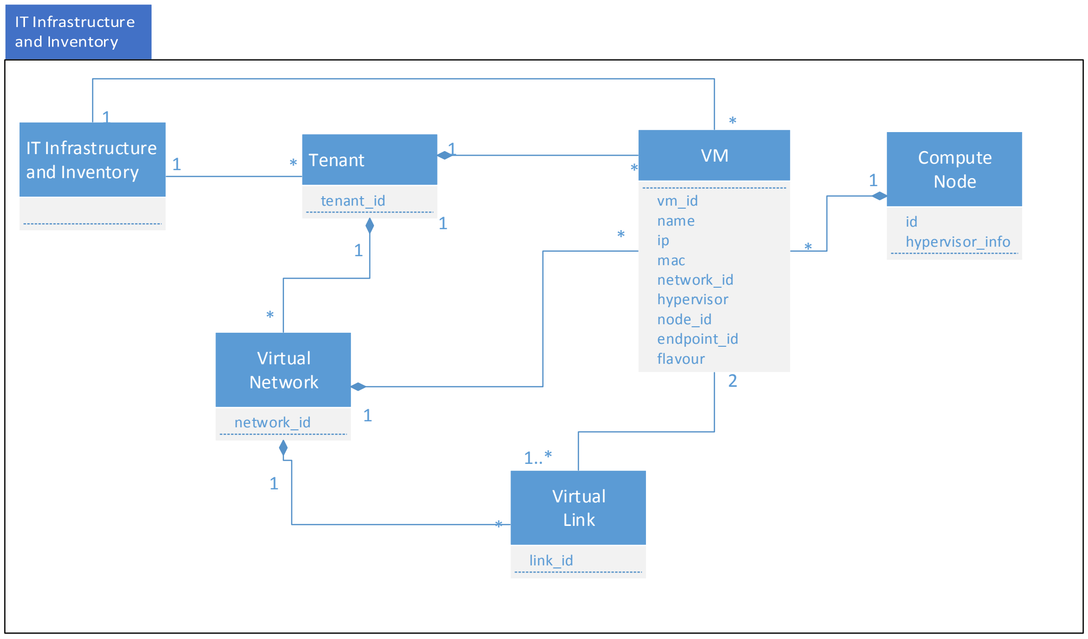


Figure 61: IT Infrastructure and Inventory information model.

A more detailed set of the attributes for the object VM is provided in the following table:

Table 26: IT infrastructure and inventory information model.

Parameters	**Type**	**Description**
vm_id	String	Identifier of the VM.

vm_object	Object	Object describing the VM as a set of properties in JSON or XML format. Parameters (variable; type; description): • Name; String; name of the VM. • Flavor**;** String; Hardware template, • Image_name; String; VM template, • NetworkId; String; L2 Network identifier. • SubnetId; String; L3 Network identifier. • Id; String; unique identifier of the VM. • MAC address; String; L2 address • IP address; Ipv4; L3 address • Hypervisor; String; Identifier of the compute node on which is deployed the VM. • Node_id; String; DpId of the switch attached • Endpoint_id; String; Network identifier of where the VM is attached.
type	String	Type of the VM (compute or storage).

11.2.3 Workflow

In the following, we provide a message exchange sequence to illustrate the use of the IT controller service by the VIMaP upon the creation and management of Over-The-Top (OTT) network services. The consumer is in this example the NFV-O and the VIMaP, both located inside the XCI. The goal is to illustrate the use of the IT infrastructure and inventory services for the ETSI NFV use case. In particular, the workflow in Figure 62 illustrates the creation of a *network service layout*, composed by a set of VMs and their direct interconnection. As shown in Figure 62, this process can consist of four major steps:

1. The NFV-O requests the instantiation of a network service layout to the VIMaP composed by a set of VMs. Note that the network service layout is associated to a tenant, which is the owner of this network service layout.
2. The VIMaP parses the template specified by the NFV-O and requests to the IT infrastructure the instantiation of the following resources:
   2.1. The creation of a VN layout, returning an identifier in case of successful creation.
   2.2. In the case of successful creation, the IT controller returns an identifier for this virtual network layout.
   2.3. The instantiation of the VMs forming the virtual network layout with the identifier previously returned by the IT infrastructure and inventory.
      2.3.1. Based on the template introduced to the VIMaP, the IT controller creates/instantiates a set of VMs.
      2.3.2. The number of instantiated VMs is based on the profile requested by the NFV-O.
      2.3.3. A zone (physical location) is selected where each VM will be geographically deployed.

3. The VIMaP (e.g., by request of the VNF Manager, the NFV-O) may decide, at any point in time of the lifecycle of the network service layout, to update its profile, adding or removing elements.

4. The VIMaP receives a request to terminate the network service layout. This implies the interaction with the IT infrastructure and inventory service for the subsequent deallocation of VMs.

The entities involved in this process are the NFV-O, the VNF manager, and the IT infrastructure and inventory.

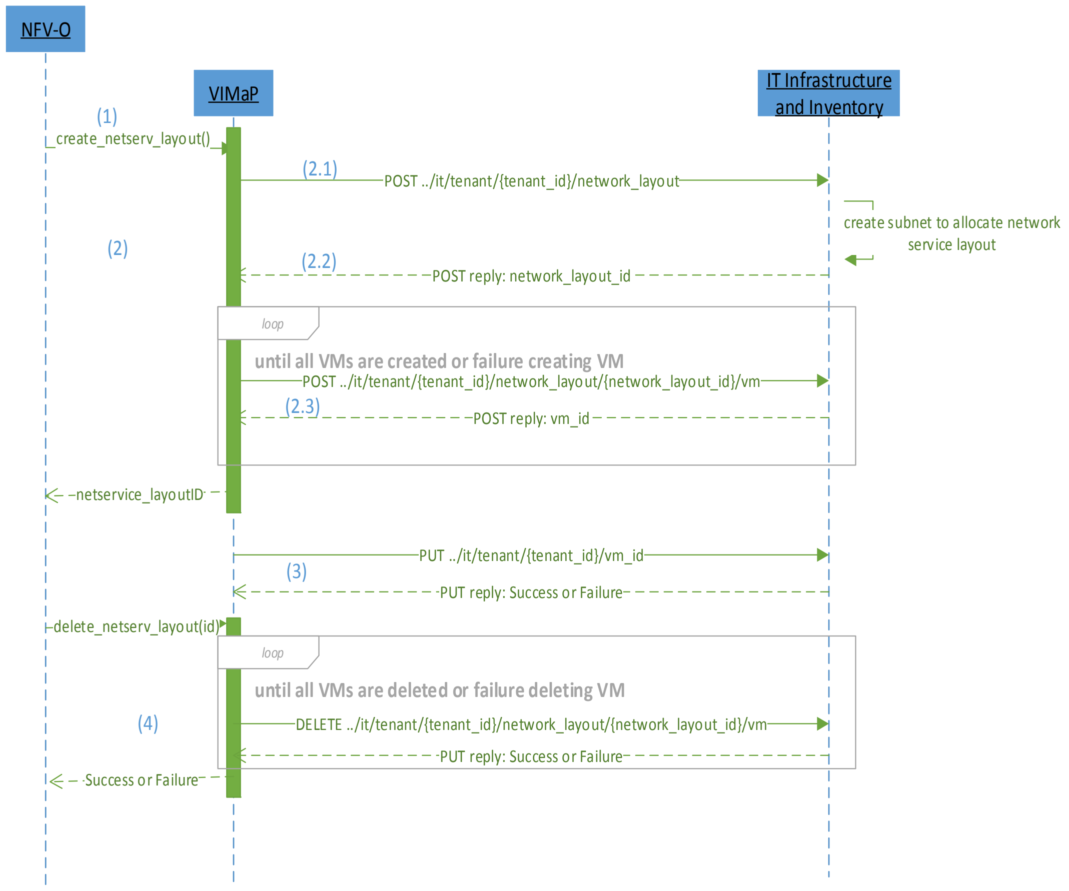

NFV-O
VIMaP
IT Infrastructure and Inventory
(1)
create_netserv_layout()
(2.1)
POST ../it/tenant/{tenant_id}/network_layout
create subnet to allocate network service layout
(2)
(2.2)
POST reply: network_layout_id
loop
until all VMs are created or failure creating VM
POST ../it/tenant/{tenant_id}/network_layout/{network_layout_id}/vm
(2.3)
POST reply: vm_id
netservice_layoutID
PUT ../it/tenant/{tenant_id}/vm_id
(3)
PUT reply: Success or Failure
delete_netserv_layout(id)
loop
until all VMs are deleted or failure deleting VM
(4)
DELETE ../it/tenant/{tenant_id}/network_layout/{network_layout_id}/vm
PUT reply: Success or Failure
Success or Failure

Figure 62: IT infrastructure and inventory workflow example.

11.3 Statistics

11.3.1 APIs

In the following, we provide a description of the APIs offered by the Statistics service.

Table 27: Statistics API.

Prot.	Type	URI	Parameters	
REST	GET	*../stats/{tenant_id}* Retrieve a list of *stats* that can be polled	Input	*tenant_id*
			Output	Success: Return *list of stats* in the response body.
				Failure: Error code
REST	GET	*../stats/{tenant_id}/ {stat_id}/samples* Retrieve samples	Input	*tenant_id* *stat_id*
			Output	Success: Return *stat* structure in the response body.
				Failure: Error code
REST	POST	*../stats/{tenant_id}/ {stat_id}/samples* Post a list of samples	Input	*tenant_id* *stat _id* *samples*
			Output	Success: Return *list of meters* in the response body.
				Failure: Error code
REST	GET	*../stats/{tenant_id}/ {stat_id}/statistics* Retrieve statistics out of samples	Input	*tenant_id* *stat _id* *stat_info*
			Output	Success: Return statistics of samples according to *stat_info* in the response body.
				Failure: Error code
REST	GET	*../stats/{tenant_id}/ {stat_id}/statistics/f unctions* Retrieve list of available aggregate functions	Input	*tenant_id* *stat_id*
			Output	*List of aggregate functions XCI can perform*
REST	GET	*../stats/{tenant_id}/ {stat_id}/alarms* Retrieve list of available alarms	Output	Success: Return list of *alarms* structure in response body.
				Failure: Error code

REST	PUT	*../stats/{tenant_id}/{stat_id}/alarms/{alarm_id}* Set alarm_id	Input	*tenant_id* *stat_id* *alarm_id* *alarm_info*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	DELETE	*../stats/{tenant_id}/{stat_id}/alarms/{alarm_id}* Delete alarm_id	Input	*tenant_id* *stat_id* *alarm_id*
			Output	Success: Status Code of normal end
				Failure: Error code
Websockets	SUBSCRIBE	*../stats/{tenant_id}/{stat_id}/alarms/{alarm_id}* Subscribe to an alarm	Input	*tenant_id* *stat_id* *alarm_id*
			Output	Alarm flag
WebSockets	ASYNC	notification event	Output	*tenant_id* *stat_id* *alarm_id* *alarm_info*

11.3.2 Information Model

The Statistics service information model supports the collection of samples of different types of information: bytes transmitted/received, bytes stored, free storage space, CPU load, etc., a simple data processing (e.g. aggregation of samples), and setting up alarms for monitoring. Note that this information is collected per *tenant_id.*

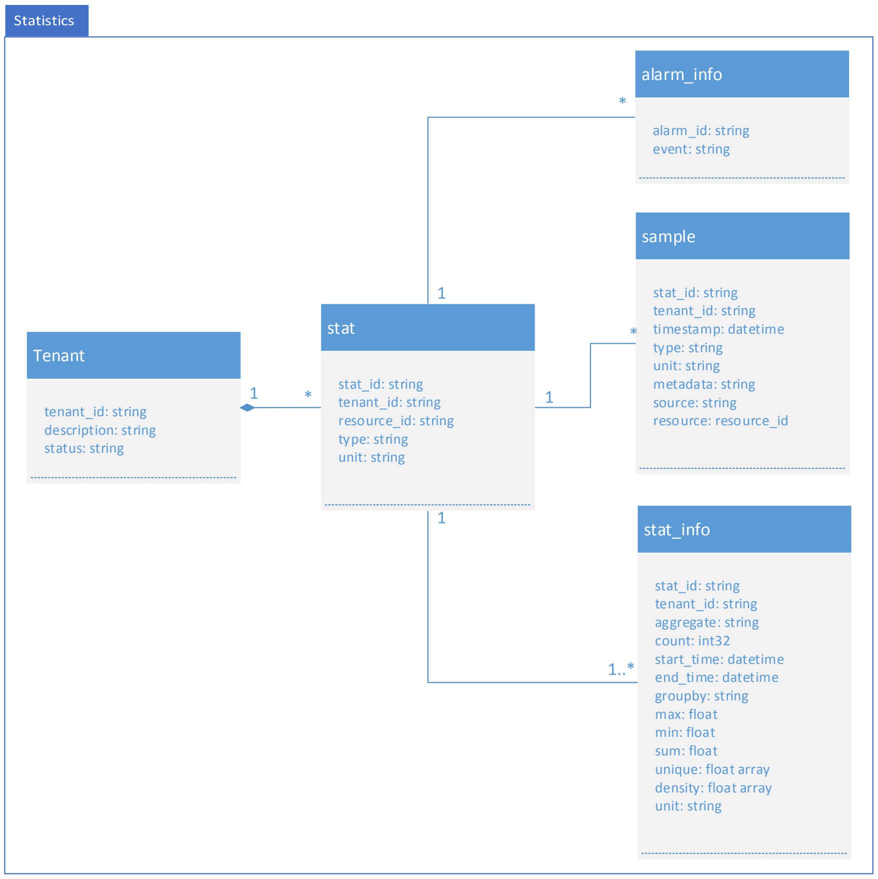


Figure 63: Statistics information model.

Next, the main data objects are presented in more detail:

Table 28: Statistics information model.

Parameter	**Type**	**Description**
stat_id	Integer	Unique identifier of a type of data (bytes transmitted/received, storage used, storage free). The identifier is unique across different tenants which removes the need to specify a *tenant_id*.
tenant_id	Integer	Unique identifier of a virtual tenant.
alarm_info	Object	It contains an *alarm_id* to uniquely identify the alarm and a description of the event that triggers the alarm, e.g. when bytes transmitted reach a threshold.
sample	Object	Object that contains a sample collected for a *stat*, including timestamps, source of the sample and other metadata.
stat_info	Object	This object is aimed to generate and store process information for a stat. This structure may contain a numeric value (e.g. maximum sample in a certain period of time), or an array of samples (number of occurrence of each unique sample). An

		important element of this information object is the *aggregate* function which lets us request personalized processing of a data shape by providing a mathematical function which XCI shall perform on such subset of data.

11.3.3 Workflow

Figure 64 illustrates how a consumer (an application like EMMA) can poll for statistics performed over a dataset:

1. Consumer requests Statistics service to aggregate a set of samples of *stat_id* according to certain parameters e.g. within a period of time, and an aggregate function, e.g. average. The statistics service replies with the processing result.
2. A consumer may set up an alarm should an event occur. To achieve this, a WebSocket is created so that the Statistics service can notify asynchronously the consumer when such an event occurs.

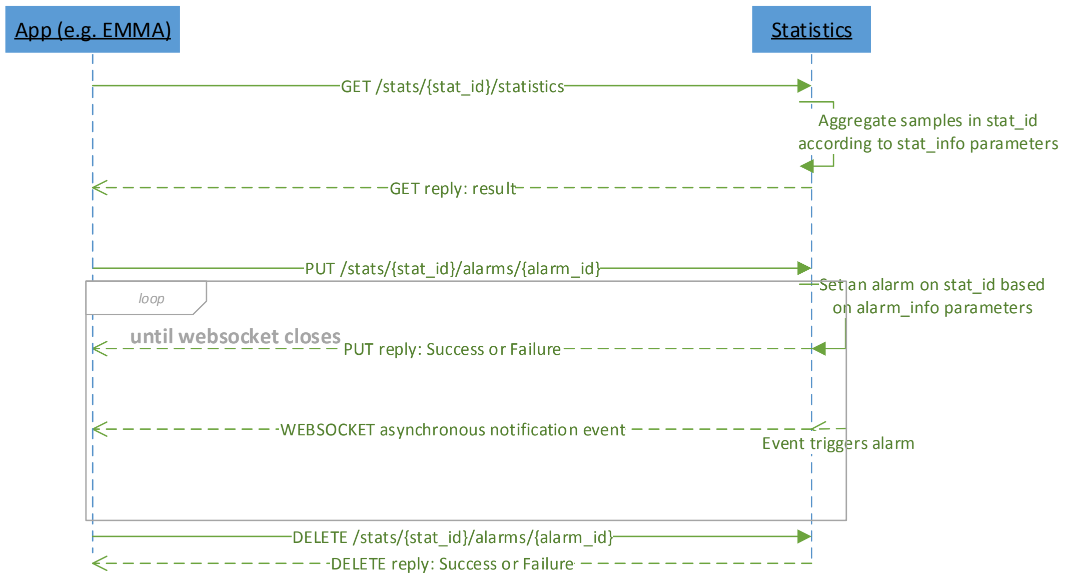

App (e.g. EMMA)
Statistics
GET /stats/{stat_id}/statistics
Aggregate samples in stat_id according to stat_info parameters
GET reply: result
PUT /stats/{stat_id}/alarms/{alarm_id}
Set an alarm on stat_id based on alarm_info parameters
loop
until websocket closes
PUT reply: Success or Failure
WEBSOCKET asynchronous notification event
Event triggers alarm
DELETE /stats/{stat_id}/alarms/{alarm_id}
DELETE reply: Success or Failure

Figure 64: Statistics workflow example.

11.4 Virtual Infrastructure Manager and Planner

The NBI of the VIMaP service is based on the REST protocol. The VIMaP is basically in charge of conducting CRUD operations for the network service layout (a set of VMs) for the ETSI NFV architecture. The VIMaP also offers an API to conduct CRUD operations for the network slice or virtual infrastructure concept (i.e., a set of not only VMs, virtual switches, and virtual routers) associated with the creation of virtual infrastructure for the MVNO use case. For operations on the network service layout, it corresponds to the deployment of Network Services as defined within the ETSI MANO architecture. In particular, it is in line with the ETSI use case #4 VNF Forwarding Graphs in [13]. For operations on network slices, it tackles the instantiation of virtual infrastructures with ultimate user control composed of a coherent set of network, compute, and storage infrastructure. In this case, the infrastructure is completely provided to the tenant (e.g., XFEs, cards, ports) including XPU resources. In fact, the VIMaP main goal is to offer to the consumer the services offered by the SDN and IT (compute and storage) controller in a unified manner.

11.4.1 APIs

In the following, we provide a description of the APIs offered by the VIMaP services.

Table 29: Virtual Infrastructure Manager and Planner API.

Prot.	Type	URI	Parameters	
REST	POST	*../vimap/tenant/{tenant_id}/vm/create_vm* Create a new VM for *tenant_id.*	Input	*vm_object* *tenant_id*
			Output	*vm_id*
REST	GET	*../vimap/tenant/{tenant_id}/vm/{vm_id}* Get information for *vm_id* in tenant *tenant_id.*	Input	*vm_id* *tenant_id*
			Output	*vm_object*
REST	DELETE	*../vimap/tenant/{tenant_id}/vm/{vm_id}* Delete VM *vm_id* for *tenant_id.*	Input	*vm_id* *tenant_id*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	PUT	*../vimap/tenant/{tenant_id}/vm/{vm_id}* Update *vm_id* information with *vm_object* in tenant *tenant_id*.	Input	*vm_id* *tenant_id* *vm_object*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	GET	*../vimap/tenant/{tenant_id}/vm* Retrieve all VM elements	Input	*tenant_id* *filter* (optional) *JSON_Object (*Object containing a key/value array with properties to

		in *tenant_id*. *../ vimap / tenant/{tenant_id}/ vm?filter={"networkId":" net1"}* Retrieve all VM elements within *net1*		filter the list)
			Output	*vm_list*
REST	POST	*../vimap/ tenant/{tenant_id}/ connectivity_service/create _call* Specify connectivity provisioning between endpoints with *call_object* in *tenant_id.*	Input	*tenant_id* *call_object*
			Output	*tenant_id* *call_id*
REST	GET	*../vimap/ tenant/{tenant_id}/ connectivity_service/{call_ id}* Get connectivity provisioning object between endpoints identified by *call_id* in *tenant_id.*	Input	*tenant_id* *call_id*
			Output	*call_object*
REST	PUT	*../vimap/ tenant/{tenant_id}/ connectivity_service/{call_ id}* Modify connectivity provisioning between endpoints in *tenant_id* specified in *call_id.*	Input	*tenant_id* *call_id* *call_object*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	DELETE	*../vimap/ tenant/{tenant_id}/ connectivity_service/{call_ id}*	Input	*tenant_id* *call_id*

		Delete connectivity provisioning between endpoints in *tenant_id.*	Output	Success: Status Code of normal end
				Failure: Error code
REST	GET	*../vimap/ tenant/{tenant_id}/ connectivity_service* Obtain connectivity provisioning information between endpoints in *tenant_id*	Input	*tenant_id*
			Output	*connectivity_service_list*
REST	GET	*../vimap/ tenant/{tenant_id}/hypervisor/{hypervsisor_id}* Get hypervisor information of hypervisor *hypervisor_id* in tenant *tenant_id.*	Input	*tenant_id* *hypervisor_id*
			Output	*hypervisor_object*
REST	GET	*../vimap/ tenant/{tenant_id}/hypervisor* Get list of hypervisors in tenant *tenant_id.*	Input	*tenant_id*
			Output	*hypervisor_list*
REST	GET	*../vimap/ tenant/{tenant_id}/ network_topology/{network_id}* Get network information for *network_id* in *tenant_id.*	Input	*tenant_id* *network_id* (sec.2.1)
			Output	*network_object* (sec.2.1)
REST	POST	*../vimap/ tenant/{tenant_id}/ Slice_provisioning_service /create_slice* Creation of a slice specified by *slice_object* in *tenant_id.*	Input	*tenant_id* *slice _object*
			Output	*tenant_id* *slice_id*
REST	GET	*../vimap/ tenant/{tenant_id}/ Slice_provisioning_service*	Input	*tenant_id* *slice_id*

		/{slice_id} Obtain information details of *slice_id* in *tenant_id.*	Output	*slice _object*
REST	PUT	*../vimap/ tenant/{tenant_id}/ Slice_provisioning_service /{slice_id}* Update information details of *slice_id* in *tenant_id* with *slice_object.*	Input	*tenant_id* *slice_id* *slice _object*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	DELETE	*../vimap/ tenant/{tenant_id}/ Slice_provisioning_service /{slice_id}* Delete *slice_id* in *tenant_id.*	Input	*tenant_id* *slice_id*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	GET	*../vimap/ tenant/{tenant_id}/ Slice_provisioning_service* Obtain information details of all slices in *tenant_id.*	Input	*tenant_id*
			Output	*slice_objects_list*
REST	POST	*../vimap/ tenant/{tenant_id}/ NetworkServiceSupportLayout/create_ netserv_layout* Create *netser_layout_object* in *tenant_id.*	Input	*tenant_id* *netserv_layout_object*
			Output	*tenant_id* *netserv_layout_id*
REST	GET	*../vimap/ tenant/{tenant_id}/ NetworkServiceSupportLayout/{ netserv_layout_id}* Obtain information details of *netserv_layout_id in tenant_id.*	Input	*tenant_id* *netserv_layout_id*
			Output	*netserv_layout_object*
REST	PUT	*../vimap/ tenant/{tenant_id}/ NetworkServiceSupportLayout/{ netserv_layout_id}*	Input	*tenant_id* *netserv_layout_id* *netserv_layout_object*

		Update *netserv_layout_id* with *netserv_layout_object* in *tenant_id*.	Output	Success: Status Code of normal end
				Failure: Error code
REST/	DELETE	*../vimap/ tenant/{tenant_id}/ NetworkServiceSupportLayout/{ netserv_layout_id}* Delete *netserv_layout_id in tenant_id.*	Input	*tenant_id* *netserv_layout_id*
			Output	Success: Status Code of normal end
				Failure: Error code
REST/	GET	*../vimap/ tenant/{tenant_id}/ NetworkServiceSupportLayout* Obtain information details of all network service layouts in *tenant_id.*	Input	*tenant_id*
			Output	*netserv_layout_objects_list*
REST/	GET	*../vimap/ tenant/{tenant_id}/ streams/ topology_update/ {network_id}* Obtain URL to subscribe to network topology events.	Input	*tenant_id* *network_id* (sec.2.1)
			Output	URL to subscribe to notification service (e.g., websocket)
REST	GET	*../vimap/ tenant/{tenant_id}/ streams/ connectivity_service_update/{call_id}* Obtain URL to subscribe to connectivity service events.	Input	*tenant_id* *call_id*
			Output	URL to subscribe to notification service (e.g., websocket)
WEBSOCKET	SUBSCRIBE	*../vimap/ tenant/{tenant_id}/ streams/ connectivity_service_update/{call_id}* Subscription to connectivity service specified by *call_id.*	Output	*call_object*

11.4.2 Information Model

The VIMaP information data model supports the concept of *tenant*. A tenant, at the VIMaP service, may be composed of different slices (virtual infrastructure use case) and different network service layouts (OTT use case). Both the network *slice* and *network_service_layout* objects are considered a VN from the point of view of the IT infrastructure and inventory service. A slice or a network service layout entails a set of calls defining the flow patterns between these components.

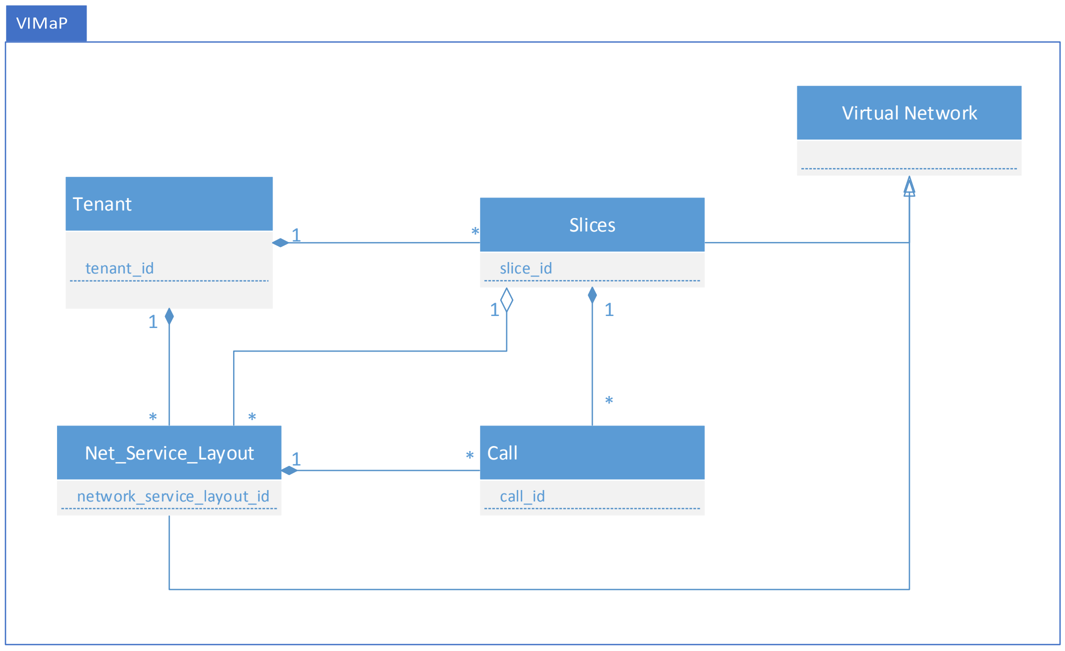


Figure 65: Virtual Infrastructure Manager and Planner information model.

In the following, we present more detailed information of the most relevant objects forming part of the VIMaP NBI service.

Table 30: Virtual Infrastructure Manager and Planner information model.

Parameters	**Type**	**Description**
tenant_id	String	Identifier of the tenant who is requesting the VIMaP service.
vm_id	String	Identifier of the VM.
vm_object	Endpoint	Object describing the VM as a set of properties in JSON or XML format: Parameters (variable; type; description): • Name; String; name of the VM. • Flavor**;** String; Hardware template. • Image_name; String; VM template. • NetworkId; String; L2 Network identifier. • SubnetId; String; L3 Network identifier. • Id; String; unique identifier of the VM. • MAC; String; L2 address • IP; Ipv4; L3 address • Hypervisor; String; Identifier of the compute

		(physical) node on which is deployed the VM. • Node_id; String; DpId of the switch attached • Endpoint_id; String; Network identifier of where the VM is attached.
call_id *call_object*	String Call	Identifier of the connectivity service (Call) Intent-based connectivity service request object described as a set of properties in JSON or XML format: Parameters (variable; type; description): • aEnd**;** Endpoint; Source connectivity service endpoint • zEnd**;** Endpoint; Destination connectivity service endpoint • transport_layer; TransportLayerType; Connectivity service description (L0, L2, L3…) • traffic_parameters; TrafficParams; QoS parameters describing the connectivity service (Bandwidth, Latency…).
Slice _object	Slice	Slice object described as a set of properties in JSON or XML format: Parameters (variable; type; description): • virtual_IT_infrastructure**;** JSON; object containing the description of the virtual IT infrastructure requested. It should contain the description of computing and storage resources, requested for the slice. • virtual_tenant_network; VTN (sec 2.10.1); abstract representation of the network slice requested. • network_service_layout**:** NetworkServiceSupportLayout; object containing a set of service endpoints (VMs) connected by a graph representation (layout). • control_stack: JSON: object describing the Software-Defined control stack for the slice. This object may contain the addressing and security parameters to access the control instances.
netserv_layout _object	NetworkServiceSupportLayout	A network service layout object described as a set of properties in JSON or XML format: Parameters (variable; type; description): • service_endpoints**;** list(Endpoint); list of service endpoints • transport_layer; TransportLayerType; Connectivity service description (L0, L2, L3…) • traffic_parameters; TrafficParams; QoS parameters describing the connectivity service (Bandwidth, Latency…). • topology_layout; Topology; graph representation of service endpoints connectivity.

11.4.3 Workflow

In the following, we provide a message exchange sequence to illustrate the use of the VIMaP service by an NFV-O when managing the creation and management of OTT network services. The consumer is in this example the NFV-O, which is located inside the XCI. The goal is to illustrate the use of the VIMaP services for the ETSI NFV use case. In particular, the workflow in Figure 66 illustrates the creation of a network service layout composed by a set of VMs and their direct interconnection.

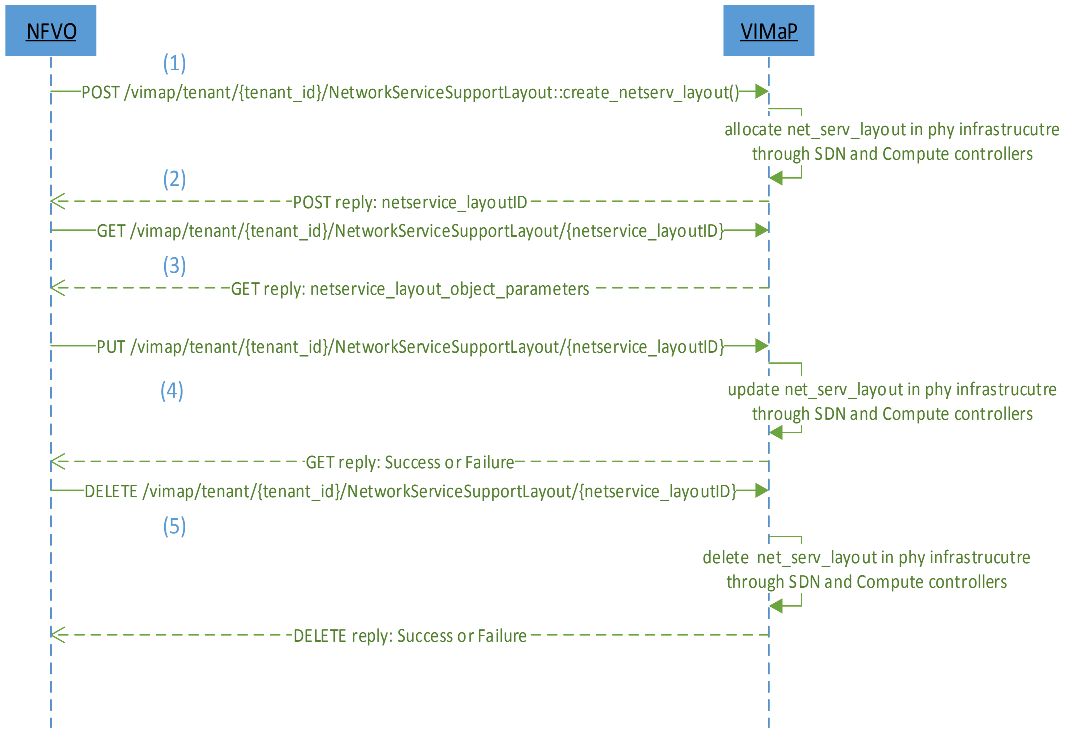

NFVO
VIMaP
(1)
POST /vimap/tenant/{tenant_id}/NetworkServiceSupportLayout::create_netserv_layout()
allocate net_serv_layout in phy infrastrucutre through SDN and Compute controllers
(2)
POST reply: netservice_layoutID
GET /vimap/tenant/{tenant_id}/NetworkServiceSupportLayout/{netservice_layoutID}
(3)
GET reply: netservice_layout_object_parameters
PUT /vimap/tenant/{tenant_id}/NetworkServiceSupportLayout/{netservice_layoutID}
(4)
update net_serv_layout in phy infrastrucutre through SDN and Compute controllers
GET reply: Success or Failure
DELETE /vimap/tenant/{tenant_id}/NetworkServiceSupportLayout/{netservice_layoutID}
(5)
delete net_serv_layout in phy infrastrucutre through SDN and Compute controllers
DELETE reply: Success or Failure

Figure 66: Virtual Infrastructure Manager and Planner workflow example.

As shown in Figure 66, this process can consist of the major steps:

1. The NFV-O creates a network service layout formed by a set of VMs.
   1.1. The VIMaP conducts the necessary operations to create and interconnect the VMs forming the network service layout.
   1.2. Note that in this case, the default algorithm used by VIMaP offers the logic for the placement of the VMs. This does not preclude the specification of the placement of VMs by other entities (e.g., the NFV-O or the MTA).
2. In case the VIMaP can allocate the indicated set of VMs (with their associated characteristics specified by a template) it returns an id of the successfully created network service layout.
3. The NFV-O can request the characteristics and status of the created network service layout.
4. At a given point, the NFV-O can update the characteristics or the template of the previously created network service layout. For instance, it can request to change the location, the characteristics of a given VM, or the way the VMs are

connected between them. This change implies the interaction either with the SDN or compute controllers, or with both entities.

At a given point in time, the NFV-O may decide to deallocate the network service layout from the physical infrastructure. This requires the interaction of the VIMaP with both the SDN and compute controllers.

11.5 VNF Manager

11.5.1 APIs

The VNF Manager provides mechanisms to create, retrieve and remove VNFs, as described in the following table.

Table 31: VNF Manager API.

Prot.	Type	URI	Parameters	
REST	POST	*../vnfm/vnf* Create a new VNF for a given tenant.	Input	*VNFD Id* *VNF Tenant Id*[10] *Deployment flavour Id* *NS Id* External virtual links Ids (List<String>): identify the external virtual links the VNF must be connected to (through its external connection points).
			Output	*VNF ID*
REST	GET	*../vnfm/vnf/vnf_id* Retrieve the information related to a given VNF.	Input	*VNF Id* *VNF Tenant Id*
			Output	*VNF record*
REST	DELETE	*../vnfm/vnf/vnf_id* Remove an existing VNF.	Input	*VNF Id* *VNF Tenant Id*
			Output	Success: Status Code of normal end
				Failure: Error code

[10] A tenant is one or more NFV MANO service users sharing access to a set of physical, virtual or service resources. A VNF tenant is a tenant to which VNFs are assigned. (See [14])

11.5.2 Information Model

The main entity managed by the VNF Manager is a VNF. The characteristics of a VNF are defined according to a standard template, called VNFD, which defines:

- VNF generic information, like vendor, version, human readable description, etc.
- The VNF Components (VNFC), which compose the VNF and their characteristics, through the definition of associated Virtual Deployment Units (VDUs).
- The internal and external Connection Points and the virtual links they are attached to.
- The dependencies between the VNFC, i.e. between VDUs.
- The scripts and configuration parameters to be launched at the various stages of the VNF lifecycle.
- The monitoring parameters at the whole VNF and single VNFCs level.
- The criteria and the constraints for automated and on-demand scaling of the VNF.

The VNF Descriptor information model is described in Figure 67.

VNFDs are stored in the VNFD DB, a shared DB which is accessed by both VNFM and NFV-O. Suitable management APIs are exposed by the NFV-O to load new VNFDs in the repository. This procedure is defined by ETSI NFV MANO standard and it is out of the scope of this document. During the instantiation of a VNF, the VNFD is specified in the request through its unique identifier.

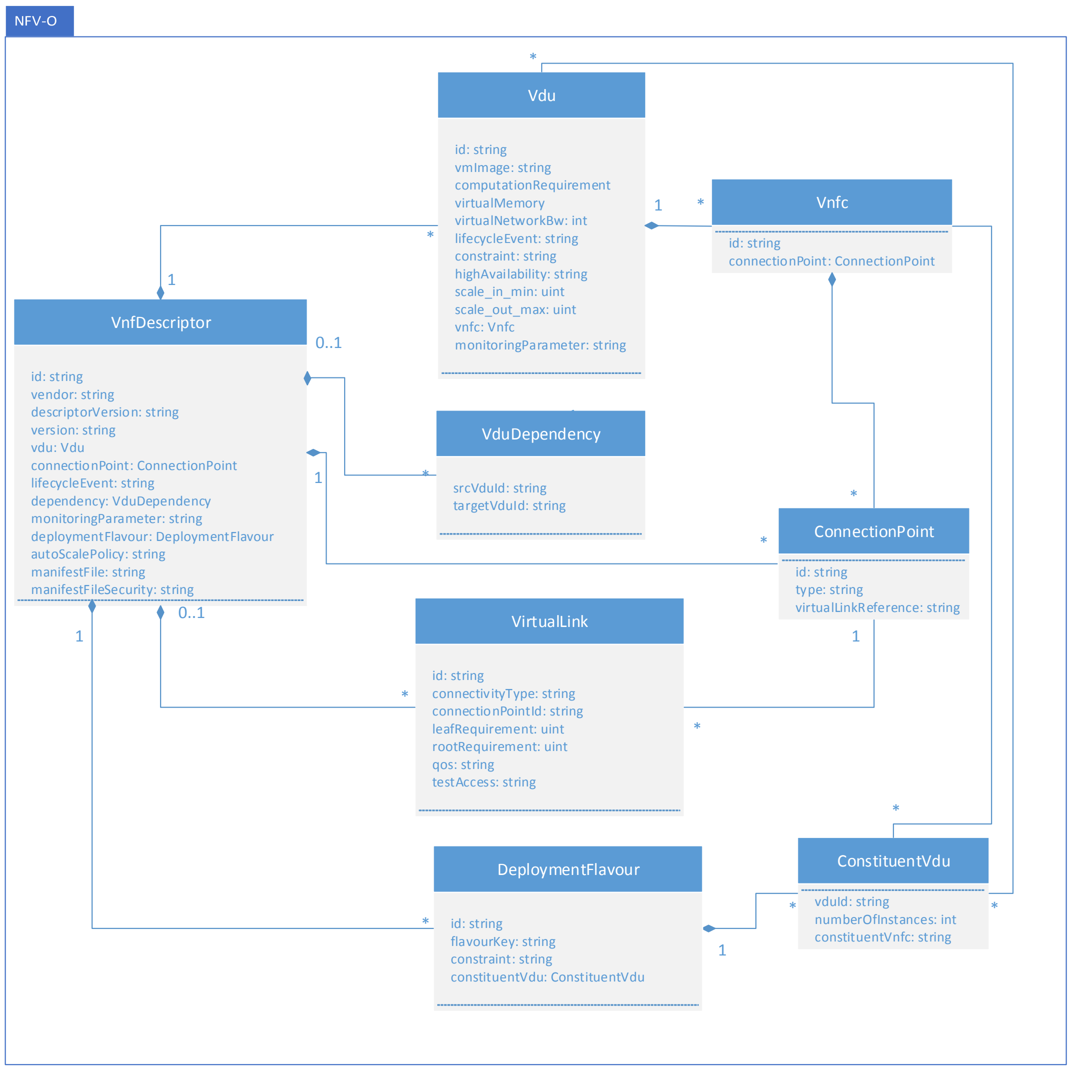


Figure 67: VNF descriptor information model.

Next, the main data objects are presented in more detail:

Table 32: VNF Manager information model.

Parameter	Type	Description
VNFD Id	String	ID of the VNF descriptor to be instantiated.
Deployment flavour Id	String	Defines the size of the VNF to be instantiated.
VNFD	Object	Description of the instantiated VNF, its VNFC instances, its status and its parameters.
Vnf record	Object	Description of the instantiated VNF, its status, and its parameters.

11.5.3 Workflow

This section describes the workflows for instantiation and termination of single VNFs, modelled according to the option of resource allocation done by the NFV-O. In other terms, while the VNFM is responsible for coordinating the whole instantiation procedure, the request for resource allocation to the VIM is mediated by NFV-O. These workflows are part of the whole Network Service instantiation and termination workflows; in this case, the VNFM client is actually the NFV-O itself.

As shown in Figure 68, the VNFM receives a request to instantiate a VNF, receiving as input the VNFD, together with other parameters which indicate the size of the VNF (i.e. its deployment flavour) and how the VNF must be interconnected to the whole Network Service (e.g. through the specification of already established external virtual links). The VNFM generates a VNF Id which is immediately returned and used as reference ID for further asynchronous notifications or requests related to the lifecycle of that VNF. The VNFM elaborates the VNFD and identifies the virtual resources (network, storage, computing) which must be allocated to build all the VNF Components (VNFCs) and the internal virtual links that compose the VNF itself. The resulting resources are requested to the NFV-O that can optionally perform some algorithms to decide the optimal resource placement and finally forwards the request to the VIM. The VIM allocates all the resources, typically starting from the network side, and when finished notifying the NFV-O which, in turns, sends an acknowledgement to the VNFM. Once the allocation procedure is finished, the VNFM takes care of the configuration of the VNF and its VNFCs and finally notifies the originating requester about the instantiation result.

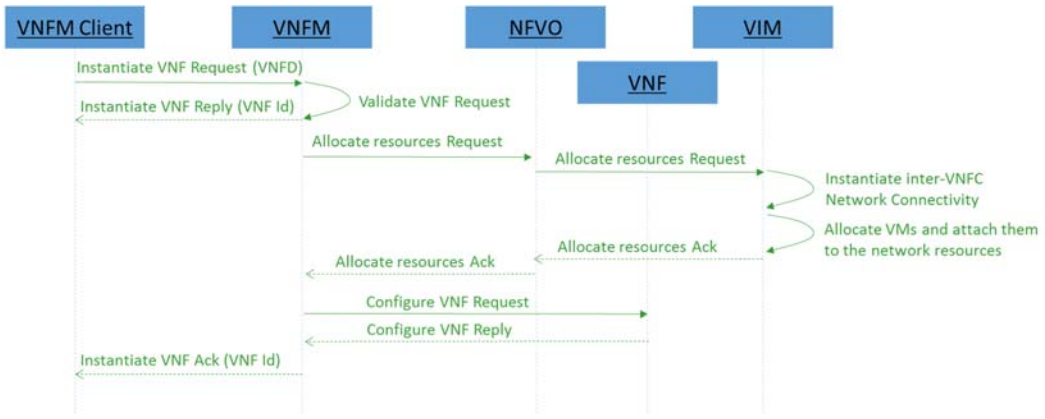

VNFM Client
VNFM
NFVO
VIM
VNF
Instantiate VNF Request (VNFD)
Instantiate VNF Reply (VNF Id)
Validate VNF Request
Allocate resources Request
Allocate resources Request
Instantiate inter-VNFC Network Connectivity
Allocate VMs and attach them to the network resources
Allocate resources Ack
Allocate resources Ack
Configure VNF Request
Configure VNF Reply
Instantiate VNF Ack (VNF Id)

Figure 68: VNF Manager workflow example: Instantiate VNF.

Figure 69 shows the termination procedure. The VNFM sends the VNF the commands required to (gracefully) shutdown the running applications, as specified in the VNFD. Then it asks the NFV-O to delete the virtual resources previously allocated, an action which is executed interacting with the VIM to remove VMs and network resources. Once all the resources are deleted, the originating requester is informed with an asynchronous message.

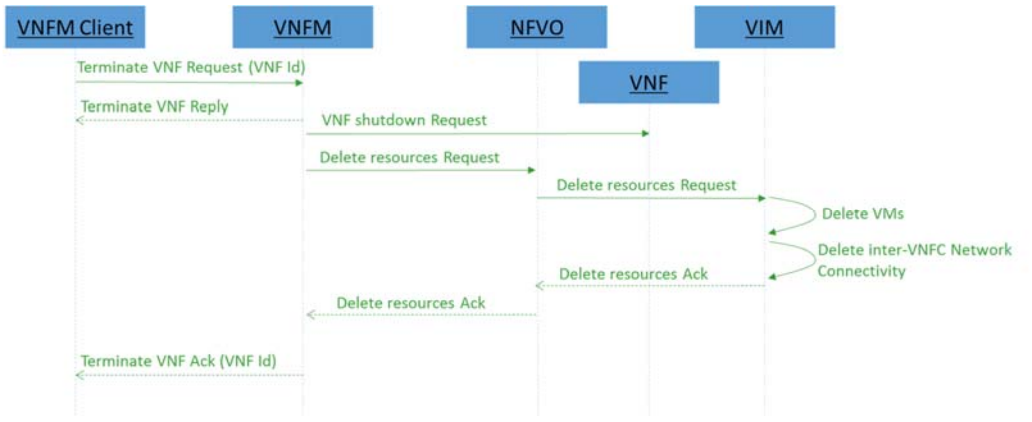


Figure 69: VNF Manager workflow example: Terminate VNF.

11.6 Analytics for Monitoring

11.6.1 APIs

This service reliably collects measurements on the utilization of the physical and virtual resources comprising the cloud infrastructure. This dataset can be used for subsequent retrieval and analysis, and triggering actions when it is necessary.

Table 33: Analytics for Monitoring API.

Prot.	Type	URI	Parameters	
REST	GET	*../analytics/resources* Retrieve the whole resources *../analytics/resources/*(resource_id) Retrieve details about one resource with the *resource_id*	Input	*resource_id* (optional)
			Output	*list(resource)* *resource*
REST	GET	*../analytics/meters* Return all known meters *../analytics/meters/*(meter_name) Return samples for the meter with the meter_name	Input	*meter_name* (optional)
			Output	list (meter) list (samples)
REST	POST	*../analytics/meters/*(meter_name) Post a new meter	Input	meter_name
			Output	Success: Status Code of normal end
				Failure: Error code

REST	GET	*../analytics/samples* Return all known samples *../analytics/samples/*(sample_id) Return a sample with the sample_id	Input	*sample_id* (optional)
			Output	*list (sample)* *sample*
REST	GET	*../analytics/event_types* Get all event types *../analytics/event_types/*(event_type) Return an event_type *../analytics/events* Return all events matching the filters *../analytics/events/*(event_id) Return an event with the event_id	Input	*event_type* (optional) *event_id* (optional)
			Output	*List (event types)* *event_type* *list (event)* *event*

11.6.2 Information Model

The information data model supported by the analytics for monitoring service is shown in Figure 70:

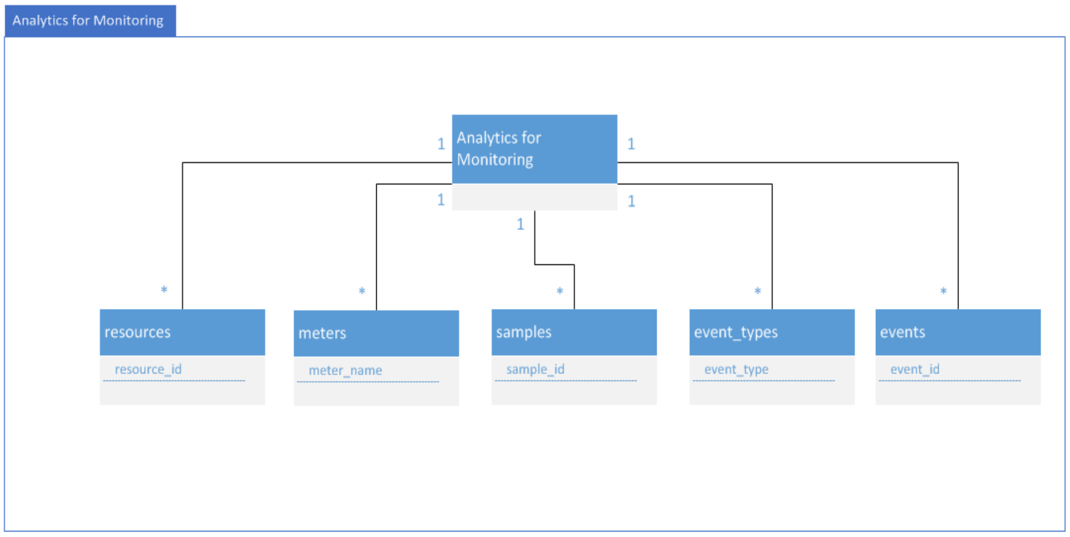


Figure 70: Analytics for Monitoring Information model.

The main parameters are described in the following table:

Table 34: Analytics for Monitoring Information model.

Parameters	Type	Description
resource_id	Unicode	Identifier of the resource
meter_name	Unicode	Identifier of the meter
sample_id	Unicode	Identifier of the sample
event_type	Unicode	Identifier of the event type
event_id	Unicode	Identifier of the event

11.6.3 Workflow

This section shows the workflow followed by an application that requires the analytics for monitoring service. In particular, Figure 71 illustrates the request of the samples for a specific meter and the request for creating a new meter for monitoring.

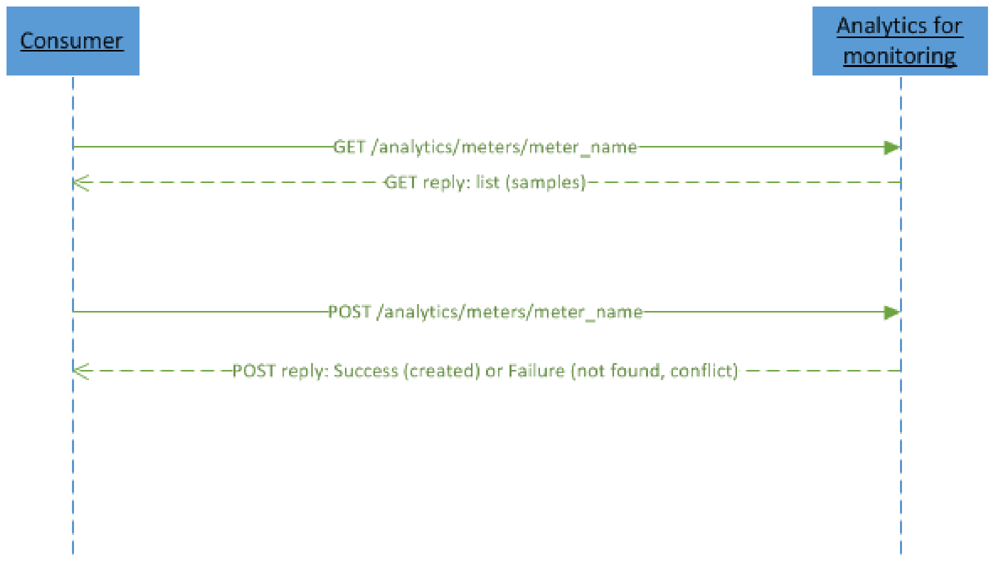

Consumer
Analytics for monitoring
GET /analytics/meters/meter_name
GET reply: list (samples)
POST /analytics/meters/meter_name
POST reply: Success (created) or Failure (not found, conflict)

Figure 71: Analytics for monitoring workflow example.

1. Gathering the monitoring data from existing services or by polling the infrastructure.
2. Configuring the kind of data gathered to meet different operating requirements.

11.7 Local Management Service

11.7.1 APIs

The Local Management Service is in charge of the modification of the status of XFEs and XPUs. For instance, it provides EMMA with REST APIs that can be used to perform the network (re-)configuration (switching on/off physical nodes) if such action leads to the minimization of the energy expenditure while ensuring an acceptable QoS.

Table 35: Local Management Service API.

Prot.	Type	URI	Parameters	
REST	GET	*../lms/{node id}/stts* Retrieve Node Status (On/Off)	Input	*Node id*
			Output	Success: *n_status*
				Failure: Error Code
REST	POST	*../lms/{node id}/stts/n_status}* Set Node Status to On or Off	Input	*Node id*
			Output	Success: *n_status*
				Failure: Error Code

11.7.2 Information model

The information data model supported by the Local Management service is shown in Figure 72, and it supports the identification of a specific node and its status.

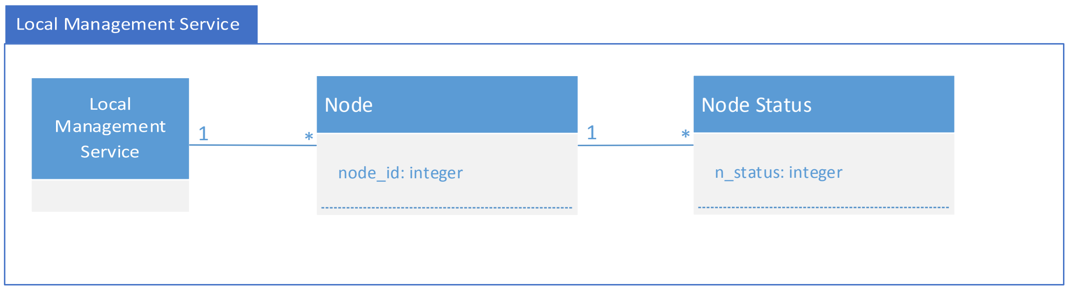


Figure 72: LMS information model

In the following, we describe the relevant parameters:

Table 36: LMS information model.

Parameter	Type	Description
node_id	Integer	Unique identifier of a physical node
n_status	Integer	Status (e.g., On/Off/Sleep) of a physical node

11.7.3 Workflow

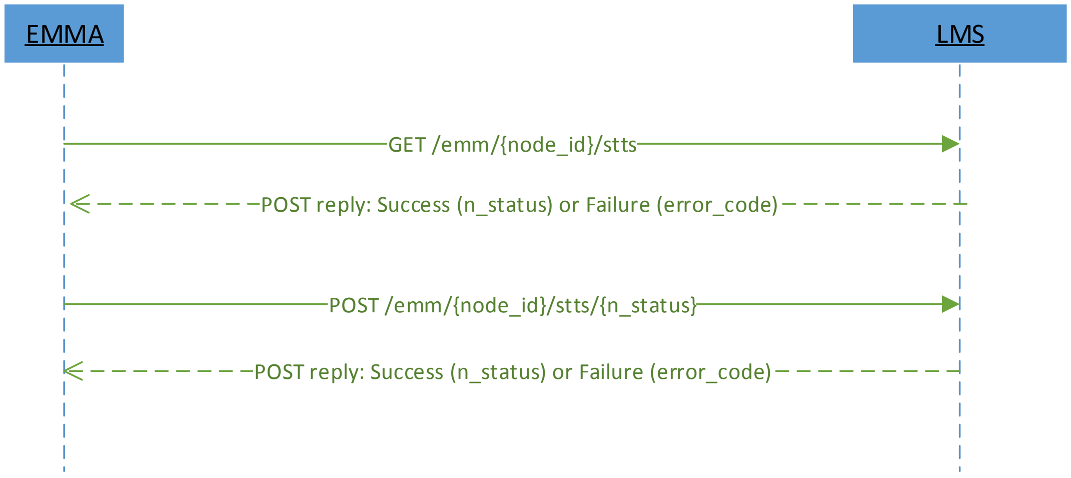

EMMA
LMS
GET /emm/{node_id}/stts
POST reply: Success (n_status) or Failure (error_code)
POST /emm/{node_id}/stts/{n_status}
POST reply: Success (n_status) or Failure (error_code)

Figure 73: Local Management service workflow example.

As shown in the first interaction of the workflow of Figure 73, the Energy Management and Monitoring Application can ask the LMS to return the status of a physical node, identified by its node ID. Possible status indicators (*n_status)* are associated to On, Off or Sleep status. This status is then used by EMMA to run its energy saving algorithms. The LMS can reply with an error code if, e.g., the node is non-existent or its status cannot be determined (if the node has been disconnected). A possible output of EMMA algorithms is the request that a physical node is turned On, Off, or set to Sleep (low energy consumption) state. This can be achieved (second interaction of the workflow of Figure 73) by asking the LMS to set the node to a specific status (On, Off or Sleep). The LMS service can reply with an error code if, e.g., the node is non-existent or its status changed is denied (different error codes can be defined to specify possible reasons why the status change has been denied).

11.8 Multi-tenancy

11.8.1 APIs

The Northbound APIs in the following table provide primitives to create, modify, delete and visualize the mapping between Virtual Tenant Networks (VTNs) and the physical infrastructure. Media types can be in JSON and/or XML format. It is important to note that the URIs specified in the request by the consumer shall be independent of the chosen representation in the implementation. Table 37 shows the most important functions to create, modify and delete a VTN.

Table 37: Multi-tenancy service API: Virtual Tenant Network (VTN) functions.

Prot.	**Type**	**URI**	**Parameters**	
REST	GET	*../mt/vtns* Retrieve list of	Input	-
			Output	Success: List of *VTN_info* structures in response body

		VTNs		Failure: Error code
REST	POST	*../mt/vtns/* Creates a VTNs	Input	-
				Request body contains VTN_info
			Output	Success: Status Code of normal end Returns *tenant_id*
				Failure: Error code
REST	GET	*../mt/vtns/{tenant_id}* Retrieve information related to a VTN	Input	*tenant_id*
			Output	Success: *VTN_info* of *tenant_id* structure in response body
				Failure: Error code
REST	POST	*../mt/vtns/{tenant_id}* Modify information of a VTN	Input	tenant_id
				Request body contains *VTN_info*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	DELETE	../mt/vtns/{*tenant_id*} Remove a VTN	Input	*tenant_id*
			Output	Success: Status Code of normal end
				Failure: Error code

Table 38 shows the most important functions to add, modify and delete virtual switches (Layer-2 forwarding elements) into a VTN.

Table 38: Multi-tenancy service API: Virtual L2 forwarding element functions (virtual switches).

Prot	Type	URI	Parameters	
REST	GET	*../mt/{tenant_id}/v_switches* Retrieve list of virtual switches that belong to *tenant_id*	Input	*tenant_id*
			Output	Success: List of *v_switch_info* structures in response body
				Failure: Error code
REST	POST	*../mt/{tenant_id}/v_switches/* Creates a virtual switch	Input	*tenant_id*
				Request body contains *v_switch_info*
			Output	Success: Status Code of normal end Returns *v_switch_id*

				Failure: Error code
REST	GET	*../mt/{tenant_id}/v_switches/{v_switch_id}* Retrieve information related to a virtual switch	Input	*tenant_id* *v_switch_id*
			Output	Success: *v_switch_info* of *v_switch_id* structure in response body
				Failure: Error code
REST	POST	*../mt/{tenant_id}/v_switches/{v_switch_id}* Modify information of virtual switch	Input	*tenant_id* *v_switch_id*
				Request body contains *v_switch_info*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	DELETE	*../mt/{tenant_id}/v_switches/{v_switch_id}* Remove a virtual switch	Input	*tenant_id* *v_switch_id*
			Output	Success: Status Code of normal end
				Failure: Error code

Table 39 shows the most important functions to add, modify and delete virtual routers (Layer-3 forwarding elements) into a VTN.

Table 39: Multi-tenancy service API: Virtual L3 forwarding element functions (virtual routers).

Prot	**Type**	**URI**	**Parameters**	
REST	GET	*../mt/{tenant_id}/v_routers* Retrieve list of virtual routers that belong to *tenant_id*	Input	*tenant_id*
			Output	Success: List of *v_router_info* structures in response body
				Failure: Error code
REST	POST	*../mt/{tenant_id}/v_routers/* Creates a virtual router	Input	*tenant_id*
				Request body contains *v_router_info*
			Output	Success: Status Code of normal end Returns *v_router_id*
				Failure: Error code

REST	GET	*../mt/{tenant_id}/v_routers/{v_router_id}* Retrieve information related to a virtual router	Input	*tenant_id* *v_router_id*
			Output	Success: *v_router_info* of *v_router_id* structure in response body
				Failure: Error code
REST	POST	*../mt/{tenant_id}/v_routers/{v_router_id}* Modify information of virtual router	Input	*tenant_id* *v_router_id*
				Request body contains *v_router_info*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	DELETE	*../mt/{tenant_id}/v_routers/{v_router_id}* Remove a virtual router	Input	*tenant_id* *v_router_id*
			Output	Success: Status Code of normal end
				Failure: Error code

Table 40 shows the most important functions to manage the mapping between physical ports (e.g. of XFEs) and virtual forwarding elements.

Table 40: Multi-tenancy service API: port mapping functions.

Prot	**Type**	**URI**	**Parameters**	
REST	GET	*../mt/{tenant_id}/v_switches/{v_switch_id}/v_ifaces* *or* *../mt/{tenant_id}/v_routers/{v_router_id}/v_ifaces* Retrieve list of interface mapping that belong to *a virtual switch or a virtual router*	Input	*tenant_id* *v_switch_id/v_router_id*
			Output	Success: List of *v_iface_info* structures in response body
				Failure: Error code
REST	POST	*../mt/{tenant_id}/v*	Input	*tenant_id*

		_switches/{v_switch_id}/ *or* *../mt/{tenant_id}/v_routers/{v_router_id}/* Creates an interface mapping		*v_switch_id/v_router_id*
				Request body contains *v_iface_info*
			Output	Success: Status Code of normal end Returns *v_iface_id*
				Failure: Error code

REST	GET	*../mt/{tenant_id}/v_switches/{v_switch_id}/{v_iface_id}*
		
		or
		
		../mt/{tenant_id}/v_routers/{v_router_id}/{v_iface_id}
		
		Retrieve information related to a virtual interface mapping
			Input	*tenant_id*
				
				v_switch_id/v_router_id
				
				v_iface_id

			Output	Success: *v_iface_info* structure in response body
				Failure: Error code

REST	POST	*../mt/{tenant_id}/v_switches/{v_switch_id}/{v_iface_id}*
		
		or
		
		../mt/{tenant_id}/v_routers/{v_router_id}/{v_iface_id}
		
		Modify information of virtual interface mapping
			Input	*tenant_id*
				
				v_switch_id/ v_router_id
				
				v_iface_id

				Request body contains *v_iface_info*
			Output	Success: Status Code of normal end
				Failure: Error code

REST	DELETE	*../mt/{tenant_id}/v_switches/{v_switch_id}/{v_iface_id}*
		
		or
		
		../mt/{tenant_id}/v_routers/{v_router_id}/{v_iface_id}
			Input	*tenant_id*
				
				v_switch_id/ v_router_id
				
				v_iface_id

			Output	Success: Status Code of normal end
				Failure: Error code

		Remove a virtual interface mapping		

Table 41 shows the most important functions to manage virtual links between virtual forwarding elements.

Table 41: Multi-tenancy service API: virtual link functions.

Prot	**Type**	**URI**	**Parameters**	
REST	GET	*../mt/{tenant_id}/v_links* Retrieve list of virtual links	Input	*tenant_id*
			Output	Success: List of *v_link_info* structures in response body
				Failure: Error code
REST	POST	*../mt/{tenant_id}/v_links/* Creates an virtual link	Input	*tenant_id*
				Request body contains *v_link_info*
			Output	Success: Status Code of normal end Returns *v_link_id*
				Failure: Error code
REST	GET	*../mt/{tenant_id}/v_links/{v_link_id}* Retrieve information of a virtual link	Input	*tenant_id* *v_link_id*
			Output	Success: *v_link_info* structure in response body
				Failure: Error code
REST	POST	*../mt/{tenant_id}/v_links/{v_link_id}* Modify information of virtual link	Input	*tenant_id* *v_link_id*
				Request body contains *v_link_info*
			Output	Success: Status Code of normal end
				Failure: Error code
REST	DELETE	*../mt/{tenant_id}/v_links/{v_link_id}* Remove a virtual link	Input	*tenant_id* *v_link_id*
			Output	Success: Status Code of normal end
				Failure: Error code

11.8.2 Information Model

The Multi-tenancy information model described in Figure 74 supports virtualization of networking resources through a mapping between virtual and physical entities.

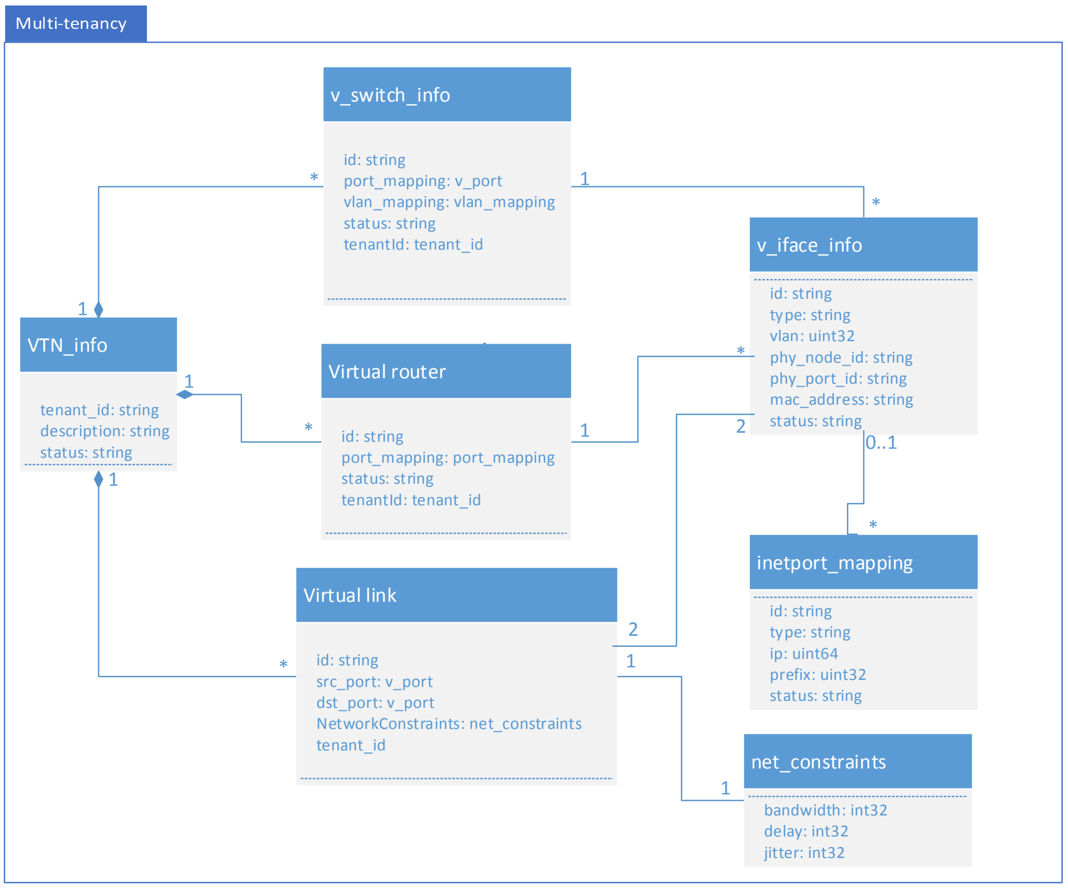


Figure 74: Multi-tenancy service information model.

See next a more detailed description of the most relevant objects:

Table 42: Multi-tenancy service information model.

Parameter	Type	Description
tenant_id	Integer	Unique identifier of a virtual tenant.
v_switch_id	Integer	Unique identifier of a virtual switch.
v_router_id	Integer	Unique identifier of a virtual router.
v_link_id	Integer	Unique identifier of a virtual link.
v_switch_info	Object	Object describing a virtual switch (e.g., status).
v_router_info	Object	Object describing a virtual router (e.g., status).
v_iface_info	Object	Object describing a virtual interface mapping (e.g.

		status, mapping to physical interfaces).
v_link_info	Object	Object describing a virtual link mapping (e.g. status, mapping to physical links/paths).

11.8.3 Workflow

Figure 75 illustrates an example usage of the multitenancy service by MTA or VIMaP, who are the consumers of the service in this case. This represents a very simple example of the creation of a VTN through a process that comprises the following steps:

1. The consumer requests the creation of a new network tenant. The multitenancy service allocates space to store information on the new tenant and generates a new identifier if successful.
2. The consumer iteratively requests the creation of virtual switches. The multitenancy service generates identifiers for each of the new entities.
3. The consumer iteratively requests the creation of virtual routers. The multitenancy service generates identifiers for each of the new entities.
4. The consumer iteratively requests a mapping between a physical port and a virtual port that belongs to a virtual switch or virtual router. In the latter case, an IP configuration is also required.
5. The consumer iteratively requests the creation of virtual links by routing pairs of virtual ports. To do so, the multitenancy service requests the SDN controller a route between two mapped physical ports given a set of network constraints (*net_constraints*).

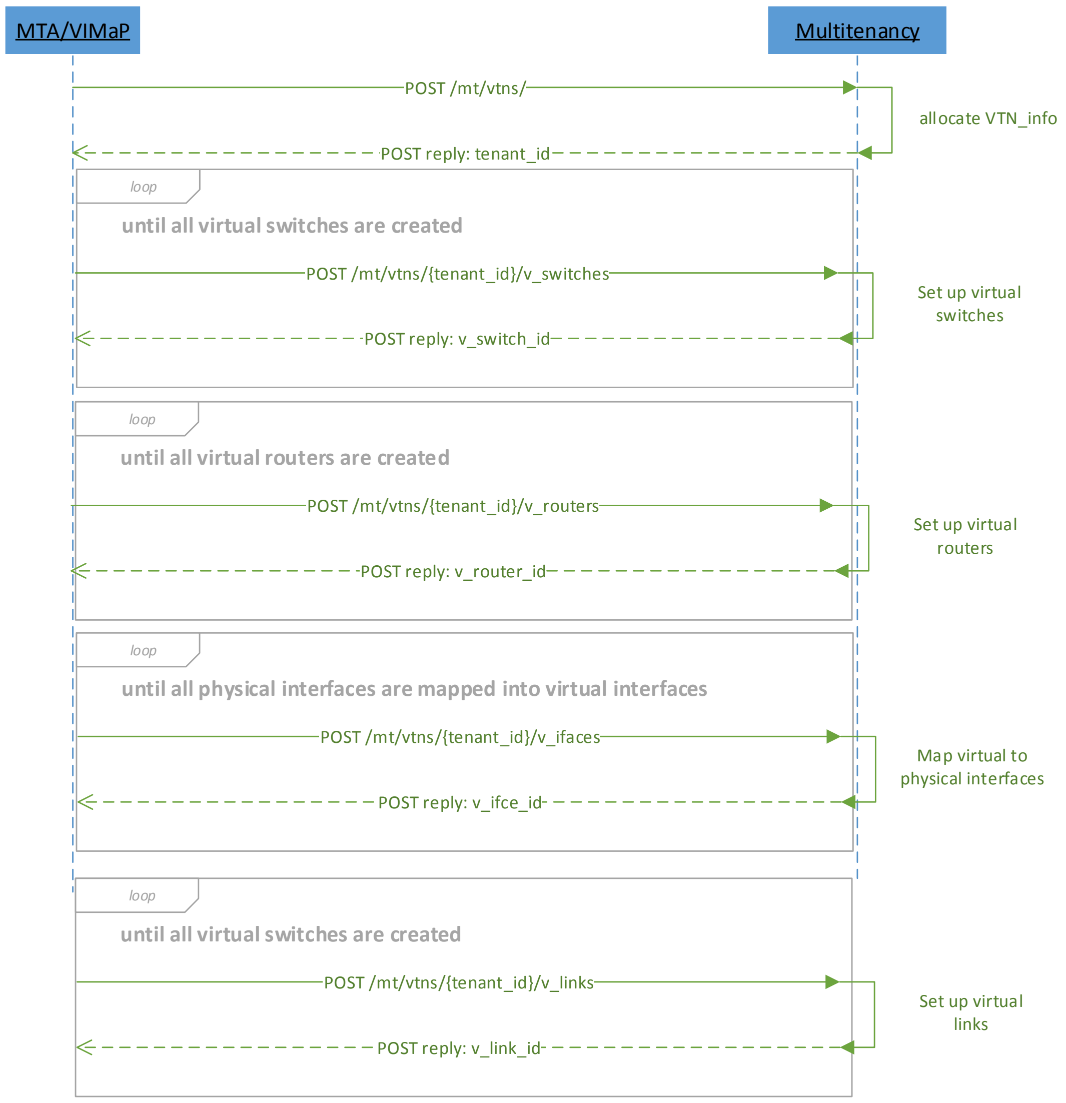

MTA/VIMaP
Multitenancy
POST /mt/vtns/
allocate VTN_info
POST reply: tenant_id
loop
until all virtual switches are created
POST /mt/vtns/{tenant_id}/v_switches
Set up virtual switches
POST reply: v_switch_id
loop
until all virtual routers are created
POST /mt/vtns/{tenant_id}/v_routers
Set up virtual routers
POST reply: v_router_id
loop
until all physical interfaces are mapped into virtual interfaces
POST /mt/vtns/{tenant_id}/v_ifaces
Map virtual to physical interfaces
POST reply: v_ifce_id
loop
until all virtual switches are created
POST /mt/vtns/{tenant_id}/v_links
Set up virtual links
POST reply: v_link_id

Figure 75: Multi-tenancy service workflow example.

12 Appendix II: Bootstrapping XPFEs: Detailed Example

In this appendix we present a detailed example of the procedure to bootstrap XPFEs as described in Section 5.4.1.1. The bootstrapping of XPFEs is explained using the example network depicted in Figure 76. In the figure, the network ports of all XPFEs are numbered. The SDN controller and the DHCP server are both directly connected to XPFE1, the DHCP server through port 2 and the SDN controller through port 1.

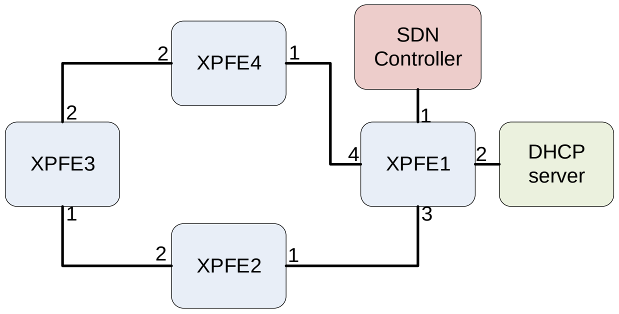


Figure 76: example topology

After their local bootup phase, all XPFEs start phase A by flooding all active interfaces with DHCP Discover messages from their LOCAL port. At this stage, all messages will be discarded except those that are transmitted by XPFE1 on its port 2. On all other ports, neighboring XPFEs are receiving these messages (and the SDN controller from port 1 of XPFE1) and as all XPFEs are forwarding incoming messages only to their LOCAL port, all of these messages are discarded.

The bootstrapping uses frames without VLAN tagging for most of the messaging. Tunneling of the control network will be configured after the connection to the SDN controller has been established. IP addresses are taken from the subnet 192.168.1.0/24, with 192.168.1.253 for the DHCP server and 192.168.1.252 for the SDN controller.

12.1 Bootstrapping XPFE1

Only the DHCP discover message of XPFE1, transmitted on its port 2, reaches the DHCP server. Upon receipt, the DHCP server replies with a DHCP offer message with the destination MAC address set to that of the LOCAL port of XPFE1. XPFE1 forwards this message to its LOCAL port where it is being processed.

The DHCP client on XPFE1 checks the vendor-specific options present in the offer, and if they are sufficient, it continues by sending the DHCP request message, which is again being flooded on all its physical ports. The DHCP server responds with a DHCP Ack which is being forwarded to the LOCAL port of XPFE1, where the DHCP client assigns the IP address granted by the DHCP server to the LOCAL port. We assume that 192.168.1.1 is used for XPFE1. This concludes phase A for XPFE1.

After the (optional) enrolment of an operator certificate, XPFE1 starts phase C by sending an ARP request message on all physical ports, of which only the messages sent on port 1 reaches the SDN controller, which responds with an ARP reply. After that, the rest of phase C (establishing a secure TLS connection) completes through that path. At this point, no control network is established, the network is depicted in Figure 77.

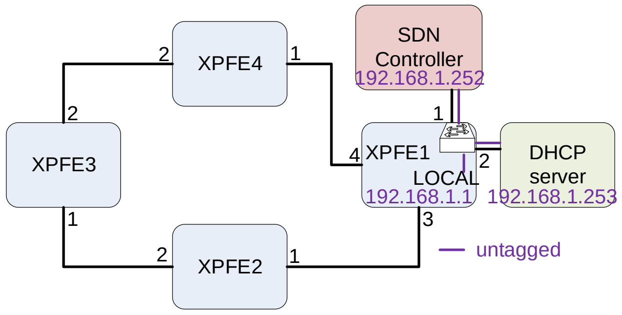


Figure 77: XPFE1 bootstrapping, end of phase C

As part of phase D XPFE1 registers at the SDN controller. The SDN controller adds the newly connected XPFE1 to its databases and configures the device.

This setup establishes a separate VN for the in-band control network, such as a control VLAN, between OF switch 1 and the SDN controller. The control network uses the XCF, i.e. has an additional MAC header and an outer VLAN. At this stage the control network is quite simple: all Ethernet frames received at ports 1 and 2 and from the LOCAL port of XPFE get the additional MAC header and outer VLAN pushed. In the opposite direction these headers are popped. This is depicted in Figure 78.

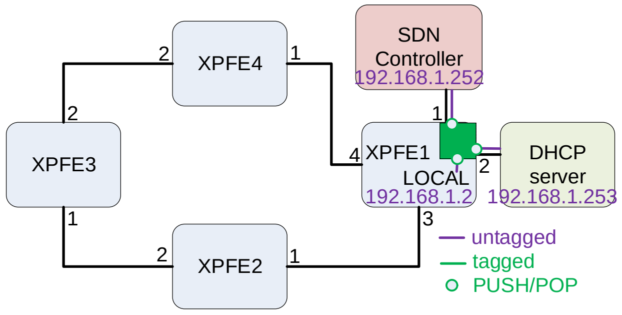


Figure 78: XPFE1 bootstrapping, end of phase D

The SDN controller sends the command to add the PUSH/POP flow entries in a single message to XPFE1. Then XPFE1 adds these flow entries to its tables. While installing the flow entries the connection between XPFE1 control and the SDN controller could be interrupted shortly. As soon as the flow entries are installed for port 1 and for LOCAL, the connection is operational again. Strictly speaking the flow entry for PUSH/POP at port 2, i.e. towards the DHCP server, could be send in a separate message as it is unrelated to the connection between XPFE1 and the SDN controller. We do not use OF scheduled bundles to synchronize the operations among the switches, as this would require having the XPFE connected to an NTP server, which would be another connection to be established in this early bootstrapping phase.

In addition, the SDN controller configures XPFE1 to trap all ingress plain Ethernet frames and forward them to the SDN controller through the control network wrapped in Packet_In messages.

12.2 Bootstrapping XPFE2 and XPFE4

After bootstrapping of XPFE 1 has completed, XPFEs 2, 3 and 4 are still in phase A. As part of connecting the second XPFE the SDN controller learns the path to the DHCP server and uses this information for future setups. After having been configured, XPFE1 wraps the DHCP discover messages it receives from XPFE 2 through its port 3 and from XPFE 4 through its port 4 into Packet_In messages and sends them to the SDN controller through its newly established OF session.

The SDN controller classifies the contents of the Packet_In messages as DHCP discover messages from (so far) unknown networking devices, and stores the MAC addresses of the local ports of both XPFEs in its database, combined with the XPFE and port through which they can be reached (XPFE1/port 3 for XPFE2 and XPFE1/port 4 for XPFE4). Thereafter, the SDN controller forwards the DHCP discover messages. As so far the SDN controller doesn’t have information on where the DHCP server is located, it forwards the messages wrapped in Packet-Out messages to XPFE 1 with the instruction to flood the packets contained on all its ports (except the ingress port). The DHCP discover messages flooded from XPFE1/port2 reach the DHCP server, which responds with DHCP offer messages. Based on its setup, XPFE1 wraps the DHCP offer packets in Packet-In messages which it sends to the SDN controller, also conveying port 2 as the ingress port. The SDN controller stores that information as the path to the DHCP server for further DHCP messages. From this time on in the bootstrapping procedure the SDN controller uses this information to directly access the DHCP server. It also extracts the DHCP offer packets contained, which it then sends again as Packet_Out messages to XPFE1. Based on the destination MAC address of the DHCP offer frame and information obtained earlier during processing of the Packet_In messages containing the DHCP discover frames, the SDN controller selects the outgoing port XPFE1 shall use.

The DHCP offer messages are being forwarded to the local ports of XPFE2 and XPFE4, which generate related DHCP request messages. These messages are forwarded along the same path as the DHCP discover messages earlier, being responded to by the DHCP server by DHCP Ack messages which are also forwarded along the same paths as the DHCP Offer messages earlier.

After completion of phase A and optionally phase B, XPFE2 and XPFE4 continue with phases C and D. All packets of XPFE2 and XPFE4 are received in XPFE1, and as the DHCP messages, are forwarded to the SDN controller wrapped in Packet_In messages. The SDN controller processes the packet stream separately based on the information in the Packet_In messages, and sends related messages back to XPFE2 and XPFE4 using Packet_Out messages to XPFE1 which unwraps the contained packets and sends them on as instructed. After phase C, the network for XPFE2 is depicted in Figure 79.

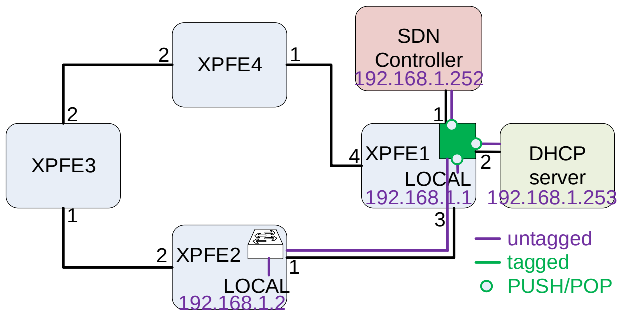


Figure 79: XPFE2 bootstrapping, after phase C

As the result, XPFE2 and XPFE4 have registered at the SDN controller through secure control channels which are tunnelled through the secure control channel of OF switch 1 by Packet_In/Packet_Out messages. The SDN controller now commands OF switch 2 and OF switch 4 to re-establish the control channel through the control network. For XPFE2, the final network is depicted in Figure 80. The SDN controller command as well XPFE1 to forward frames to the SDN controller or DHCP server depending on MAC addresses instead of using Packet_In/Packet_Out.

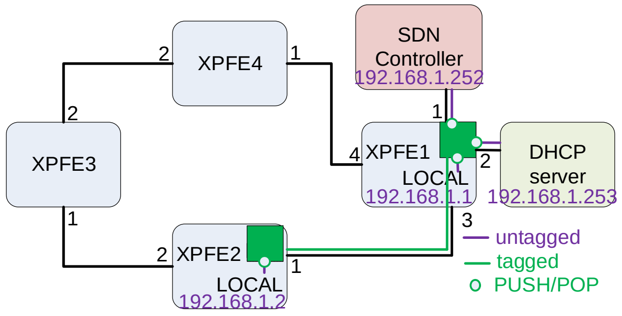


Figure 80: XPFE2 bootstrapping, after phase D

Note, there are no PUSH/POP commands for port 1 in XPFE2 nor for port 3 in XPFE1, frames for the SDN controller are exchanged with VLAN and PBB tag from this point onwards. The commands to reconfigure XPFE need to be sent in a single command by the SDN controller. The control connection between XPFE2 and the SDN controller might be interrupted shortly while the PUSH/POP flow entries are added to the tables in XPFE2 and XPFE1 is reconfigured. Only when both switches have finished their configuration, the control connection is operational again. It is important that the SDN controller reconfigures first XPFE2 and thereafter XPFE1. If done otherwise, control in XPFE2 won't be reachable anymore from the SDN controller.

12.3 Bootstrapping XPFE3

Finally, XPFE3 originated DHCP Discover frames are, based on the timing of the registration procedure of XPFE2 and XPFE4, being received by one or even both of these XPFEs. Here we assume that XPFE2 and XPFE4 completed their registration at almost the same time, so both receive the frames, and both wrap them into Packet_In messages and send them to the SDN controller through their newly established OpenFlow sessions.

The SDN controller classifies the contents of the Packet_In messages as DHCP discover messages from the (so far) unknown XPFE3, and stores the MAC address of the local port of it in its database, combined with the XPFEs and ports through which they can be reached (XPFE2/port 2 or XPFE4/port 2). In this example, the SDN controller only stores one version of this information in its database, overwriting a possibly existing entry whenever it receives a DHCP Discover message from XPFE3. Using this method, the SDN controller will later only use the path as the return path through which it last saw the DHCP Discover message. It then forwards the DHCP discover messages to XPFE1 wrapped in Packet-Out messages, with the instruction to send the DHCP Discover frames on XPFE1/port 2, based on information on the location of the DHCP server obtained earlier.

The DHCP server responds with DHCP offer messages which XPFE1 forwards to the SDN controller wrapped in Packet_In messages. The SDN controller extracts the DHCP offer frames, and selects either XPFE2 or XPFE4 as the return path dependent on through which path the DHCP Discover message has been received last. In this example, it selects XPFE2, so it sends out the DHCP offer frames wrapped in as Packet-Out messages to XPFE2 with the instruction to send the frames contained out of XPFE2/port 2.

The DHCP offer messages are being forwarded to the local port of XPFE3, which selects the first offer that contains the expected vendor specific options and generates a related DHCP request message. This message is moving along the same path as the DHCP discover message went earlier, being responded to by the DHCP server by a DHCP Ack message which is also moving along the same paths.

Thereby XPFE3 completes phase A and optional phase B, and continues with phases C and D. All frames sent out by XPFE3 are travelling through both XPFE2 and XPFE4 and so arrive twice at the SDN controller. All the protocols used in both phases are robust against packet replication, even duplicates of SYN messages. For the return path, the SDN controller can select either XPFE2 or XPFE4. In this example, it selects XPFE2.

As the result, XPFE3 has registered at the SDN controller through a secure control channel which is tunnelled through the secure control channel of XPFE2 by Packet_In/Packet_Out messages. The SDN controller now commands XPFE3 to re-establish the control channel through the control network. Thereafter the XPFE2 must extend the control network to its port2, commanded by the SDN controller. Then the SDN controller may have XPFE3 send probe messages out of its ports to detect the network between XPFE3, XPFE2 and XPFE4. Finally, the SDN controller should set XPFE3 to a state where they trap all ingress plain Ethernet frames and forward them to the SDN controller through the control network wrapped in Packet_In messages. The final configuration is depicted in Figure 81.

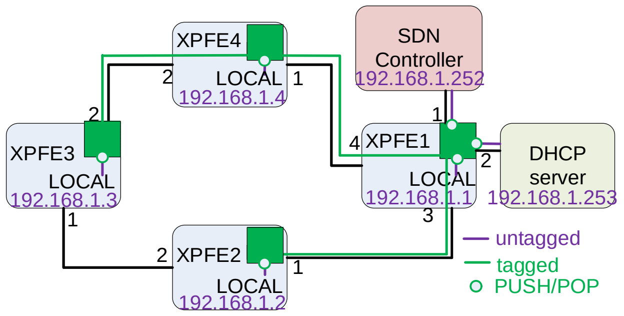


Figure 81: XPFE4 bootstrapping, after phase D

13 Appendix III: XCF Requirements

We defined requirements on the frame format, i.e. which information is contained in individual frames. The control, how this information is used in the XPFEs for the forwarding decisions, is a control-plane issue related to the XCI. It is not relevant here whether e.g. point-to-point or multipoint-to-multipoint services are established or any kind of TE is performed. These questions are important, but independent of the frame format, as long as the frames contain sufficient information to establish the corresponding services.

Table 43: XCF Requirements

ReqId	Requirement	Explanation
	Functional splits	
XCF-R1	Support multiple functional splits	The XCF has to support traffic of different functional splits of the radio protocol stack, ranging from CPRI-like fronthaul traffic to backhaul traffic
	Multi-tenancy	
XCF-R2	Isolate traffic	Provide guaranteed QoS to traffic of different tenants. Traffic of one tenant shall not impact the QoS of the traffic of other tenants.
XCF-R3	Separate traffic	Maintain the privacy of the traffic of different tenants. One tenant shall not be able to listen to traffic of another tenant.
XCF-R4	Differentiation of forwarding traffic	Traffic of different tenants may be forwarded differently.
XCF-R5	Multiplexing gain	It shall be possible to utilize statistical multiplexing gains among the traffic of different tenants.
XCF-R6	Tenant ID	The traffic of different tenants shall be identifiable.
	Coexistence	
XCF-R7	Ethernet	Compatibility with legacy Ethernet switches
XCF-R8	Security support	Security is supported for the frames themselves, i.e. encryption or authentication. Securing access to the network itself is considered a control-plane issue.
XCF-R9	Compatible with IEEE 1588v2 or IEEE 802.1AS	It shall be possible to carry synchronization information on the same links as the data traffic using XCF.
	Transport Efficiency	
XCF-R10	Short overhead	The additional headers introduced by the XCF shall be short.
XCF-R11	Multi-path	The XCF shall allow carrying traffic towards one destination on different paths, but keeping individual flows on the same path. This could be useful in meshed microwave networks.
XCF-R12	Flow	The frame format shall support to provide QoS for

	differentiation	individual flows in addition to traffic classes.
XCF-R13	Class of Service differentiation	XFEs shall support different classes of service for different types of traffic
Management		
XCF-R14	In-band control traffic (OAM)	It shall be possible to carry OAM traffic on the same links as the data traffic using XCF
Support of multiple media		
XCF-R15	802.3	yes/no
XCF-R16	802.11ad	yes/no
XCF-R17	mmWave	yes/no
Energy efficiency		
XCF-R18	Energy usage proportional to handled traffic	The XFEs shall be energy efficient by using features such as sleep modes, reduced line rates, etc.
Miscellaneous		
XCF-R19	No vendor lock in	The XCF shall be based on standards, such that no vendor lock-in may happen.

The subsequent Table 44 and Table 45 summarize how the requirements are fulfilled in the case of choosing MAC-in-MAC as XCF as well as with MPLS-TP as alternative.

Table 44: MAC-in-MAC XCF requirement fulfillment

ReqId	Requirement	Explanation
Functional splits		
XCF-R1	Support multiple functional splits	All functional splits with packetized transport can be supported by the XCF, see also XCF-R2
Multi-tenancy		
XCF-R2	Isolate traffic	QoS provided by PCP, see also Section 5.3
XCF-R3	Separate traffic	Traffic of different tenants can be distinguished by the B-Tag and the I-Tag
XCF-R4	Differentiation of forwarding traffic	N/A, to be solved in XCI
XCF-R5	Multiplexing gain	Naturally, as packet-based technology multiplexing gains can be achieved
XCF-R6	Tenant ID	B-Tag and I-Tag
Coexistence		
XCF-R7	Ethernet	Legacy Ethernet switches can forward frames based on DA and VLAN-ID of outer header
XCF-R8	Security support	Either payload has to be encrypted e.g. by IPsec or encrypted links have to be used, e.g. 802.1AE MACSec
XCF-R9	Compatible with IEEE 1588v2 or IEEE 802.1AS	IEEE 1588 packets can be carried as any other IP packet as payload. XCF frames and 802.1AS (gPTP) can be carried on the same link. See also Section 5.5
Transport Efficiency		
XCF-R10	Short overhead	B-tag + I-Tag: 22B, (optional) F-Tag: 6B
XCF-R11	Multi-path	Based on F-Tag

XCF-R12	Flow differentiation	Individual services can be identified by I-SID and classified to dedicated queues, although PCP is considered sufficient, see Section 5.2.2.1
XCF-R13	Class of Service differentiation	QoS provided by PCP, see also Section 5.3
	Management	
XCF-R14	In-band control traffic (OAM)	OAM traffic is under control of XCI. OAM traffic can be distinguished by Ethertype or by dedicated multicast addresses. See also Section 5.4
	Support of multiple media	
XCF-R15	802.3	Yes
XCF-R16	802.11ad	Yes
XCF-R17	mmWave	same frame format as 802.11ad
	Energy efficiency	
XCF-R18	Energy usage proportional to handled traffic	N/A
	Miscellaneous	
XCF-R19	No vendor lock in	Based on MAC-in-MAC standard

Table 45: XCF requirement fulfillment by MPLS-TP

ReqId	Requirement	Explanation
	Functional splits	
XCF-R1	Support multiple functional splits	All functional splits with packetized transport can be supported by MPLS-TP as XCF, see also XCF-R2
	Multi-tenancy	
XCF-R2	Isolate traffic	QoS provided by TC, same number as bits PCP of MAC-in-MAC, see also Section 5.3
XCF-R3	Separate traffic	Traffic of different tenants can be distinguished LSP label
XCF-R4	Differentiation of forwarding traffic	N/A, to be solved in XCI
XCF-R5	Multiplexing gain	Naturally, as packet based technology multiplexing gains can be achieved
XCF-R6	Tenant ID	LSP label
	Coexistence	
XCF-R7	Ethernet	Ethernet can be used as link layer technology.
XCF-R8	Security support	Either payload has to be encrypted e.g. by IPsec or encrypted links have to be used, e.g. 802.1AE MACSec
XCF-R9	Compatible with IEEE 1588v2 or IEEE 802.1AS	IEEE 1588 packets can be carried as any other IP packet as payload. XCF frames and 802.1AS (gPTP) can be carried on the same link. See also Section 5.5
	Transport Efficiency	

XCF-R10	Short overhead	Link layer header (18B) + LSP/PW/PW-ctrl (12B) = 30B
XCF-R11	Multi-path	Not supported
XCF-R12	Flow differentiation	Individual services can be identified by LSP label and classified to dedicated queues, although PCP is considered sufficient, see Section 5.2.2.2
XCF-R13	Class of Service differentiation	QoS provided by TCP, similar to PCP of MAC-in-MAC, see also Section 5.3
Management		
XCF-R14	In-band control traffic (OAM)	OAM traffic is under control of XCI. OAM traffic can be distinguished by Ethertype or by dedicated multicast addresses. See also Section 5.4
Support of multiple media		
XCF-R15	802.3	Yes
XCF-R16	802.11ad	Yes
XCF-R17	mmWave	Yes, same frame format as 802.11ad
Energy efficiency		
XCF-R18	Energy usage proportional to handled traffic	N/A
Miscellaneous		
XCF-R19	No vendor lock in	Based on MPLS-TP RFCs

14 Appendix IV Potential enhancements to 5G-Crosshaul data plane design

Although we consider the data plane architecture with the XCF as described in this deliverable as sufficient for 5G-Crosshaul, there are a number of ongoing networking standardization activities that target to solve similar problems as the data plane of 5G-Crosshaul. Two such activities are summarized here, one for deterministic networking and one for segment routing.

14.1 IETF DetNet data plane solutions

IETF DetNet (Deterministic Networking, [47]) provides a capability to carry unicast or multicast data flows for real-time applications with extremely low data loss rates, timely delivery and bounded PDV. DetNet identifies existing IP and MPLS, layer-2 or layer-3 encapsulations and transport protocols that could be considered as foundations for a deterministic networking data plane. The DetNet data plane is logically divided into two layers, namely DetNet service layer and DetNet transport layer.

The DetNet transport layer operates below and supports the DetNet service layer and optionally provides congestion protection for DetNet flows. DetNet proposes the following data plane technology alternatives for the DetNet transport layer: native IPv6, native IPv4, MPLS and Bit Indexed Explicit Replication (BIER).

The DetNet service layer provides adaptation of DetNet services. It is composed of a shim layer to carry deterministic flow specific attributes, which are needed during forwarding and for service protection. The DetNet service layer is used to deliver traffic end to end across a DetNet domain. DetNet proposes the following data plane technology alternatives for the DetNet service layer: Pseudo Wire Emulation Edge-to-Edge (PWE3), Generic Routing Encapsulation (GRE), MPLS-based Services for DetNet, MPLS-Based Ethernet VPN (EVPN) and higher layer header fields.

DetNet service layer technologies could be used together with the 5G-Crosshaul XCF to realize deterministic traffic flow configuration across the 5G-Crosshaul domain. For example, the Pseudo Wire Emulation Edge-to-Edge mechanism could be configured over a MAC-in-MAC (baseline XCF) packet switched network (PSN); to establish a PSN tunnel with deterministic characteristics.

14.2 Segment Routing

When forwarding a frame or packet through a network on the packet switched part of 5G-Crosshaul, each XPFE needs to be configured how to forward this packet. Typically, forwarding is based on the destination address. Segment routing [48] instead adds a stack of labels describing the path through the network when the packet enters the network. The network is seen as a set of segments that the packets have to traverse. There are two types of segment IDs.

- Node Segment IDs are used to describe a path to a node, i.e. which is the next node to reach.
- Adjacency segment IDs are used to describe a service at a node.

Segment routing is a type of source routing, where the source chooses the path and encodes it in the packet header as an ordered list of segment IDs. The intelligence for routing is on the source router while the rest of the routers can be kept simple. No per-flow state needs to be kept on these other routers. Nevertheless, there is no single router in a network acting as source router for all traffic flows. Many routers will be source routers for some traffic flows, and each of them has to control the routing of its flows. The source router intelligence is programmed by an external controller, which fits well with a SDN approach.

As one possibility, segment routing leverages MPLS, where each router adds a label or a label stack, pops some label, or swaps a label. The typical operation of a router within the network would be to determine the next hop based on the top most label and then pop this label.

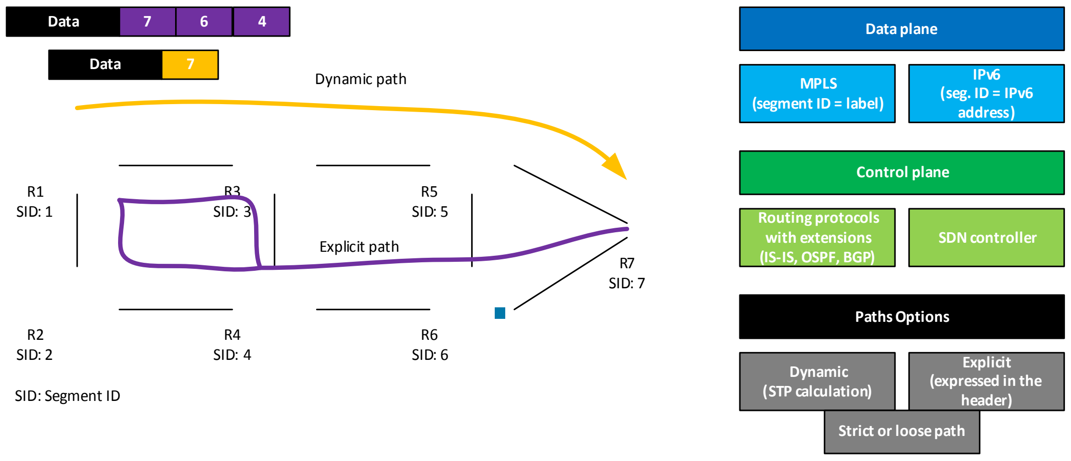


Figure 82: Segment routing.

The label stack may be incomplete; it is not necessary to list all forwarding nodes on a path. This leaves flexibility within the network to determine local paths or to provide alternative paths in case of link failures. In Figure 82, for the violet packet, the segment IDs are initially provided as label stack, describing rather precisely the path through the network, which is called an explicit path. Only within the first segment, there is some choice how to reach node 4. If the first segment is an Ethernet network, this choice could be resolved by the spanning tree protocol. For the golden packet, just the destination node is provided in the label stack and it is left to the network to find a route. This is called a dynamic path.

The choice of destination based routing, or segment routing, or any other routing, is orthogonal to the goal of 5G-Crosshaul to forward both fronthaul and backhaul traffic on the same links. Therefore, the focus within the 5G-Crosshaul was kept on MAC-in-MAC as the XCF, not extending the header with stacks of labels or addresses.