ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN
INGEGNERIA ELETTRONICA, TELECOMUNICAZIONI
E TECNOLOGIE DELL'INFORMAZIONE

CICLO XXXIV
SETTORE CONCORSUALE 09/F2
SETTORE SCIENTIFICO DISCIPLINARE ING-INF/03

AUTOMATED SERVICE PROVISIONING IN PROGRAMMABLE NETWORK INFRASTRUCTURES

Presentata da
DAVIDE BORSATTI

Supervisore
Prof. WALTER CERRONI

Coordinatore Dottorato
Prof. ALDO ROMANI

ESAME FINALE ANNO 2022

Contents

Abstract **i**

1 Introduction **1**

1.1 Programmable Network Infrastructures 1
1.2 Virtualized Network Service Provisioning 3
1.3 Network Automation 7

2 Programmable network infrastructure **9**

2.1 SFC with Openflow 9
2.1.1 The OpenStack SFC extension 9
2.1.2 Experimental setup 13
2.1.3 Experimental results 17
2.2 SFC with Segment Routing 22
2.2.1 The Network Scenario 24
2.2.2 Test bed Implementation 27
2.2.3 OpenSource MANO 28
2.2.4 Experimental Results 30
2.2.5 Comparison with Openstack SFC 34

3 Network Service Provisioning **37**

3.0.1 Physical testbed validation with OSM 37

4 5G Network Slicing **41**

4.1 Network architecture and system components 41
4.1.1 The MC server 43
4.1.2 The mobile access network 44

4.1.3 Data center management infrastructure deployment . . . 45
4.2 A Network Slice for MC communications . . . 45
4.2.1 Actors and Roles . . . 45
4.2.2 Network Slice Architecture and Characteristics . 47
4.2.3 Network Slice Delivery and Lifecycle management 50
4.3 Experimental Results . . . 51

5 Enabling Industrial IoT as a Service with Multi-access Edge Computing 56
5.1 Introduction and related works . . . 57
5.2 Reference Scenario for IIoT as a Service . . . 59
5.3 Features and Components in a MEC-enabled IIoTaaS Framework . . . 62
5.4 Proof-of-Concept Implementation . . . 65
5.5 Evaluation . . . 70
5.6 MEC011 Extension . . . 75

6 Intent Based Networking 78
6.1 Preliminary work . . . 78
6.2 Formal definition of IBN . . . 84
6.3 Future research tracks . . . 98

7 Conclusion 100

Acronyms 102

Bibliography 104

Abstract

Modern networks are undergoing a fast and drastic evolution, with software taking a more predominant role. Virtualization and cloud-like approaches are replacing physical network appliances, reducing the management burden of the operators. Furthermore, networks now expose programmable interfaces for fast and dynamic control over traffic forwarding. This evolution is backed by standard organizations such as ETSI, 3GPP, and IETF. This thesis will describe which are the main trends in this evolution. Then, it will present solutions developed during the three years of Ph.D. to exploit the capabilities these new technologies offer and to study their possible limitations to push further the state-of-the-art. Namely, it will deal with programmable network infrastructure, introducing the concept of Service Function Chaining (SFC) and presenting two possible solutions, one with Openstack and OpenFlow and the other using Segment Routing and IPv6. Then, it will continue with network service provisioning, presenting concepts from Network Function Virtualization (NFV) and Multi-access Edge Computing (MEC). These concepts will be applied to network slicing for mission-critical communications and Industrial IoT (IIoT). Finally, it will deal with network abstraction, with a focus on Intent Based Networking (IBN). To summarize, the thesis will include solutions for data plane programming with evaluation on well-known platforms, performance metrics on virtual resource allocations, novel practical application of network slicing on mission-critical communications, an architectural proposal and its implementation for edge technologies in Industrial IoT scenarios, and a formal definition of intent using a category theory approach.

Chapter 1

Introduction

In the last years, the way networks are controlled and managed is undergoing radical changes. Software is taking a predominant role in modern networks to help reach the stringent requirements of current services. The title of this thesis underlines three important trends in the evolution path modern networks are undergoing:

- Programmable Network Infrastructures;
- Virtualized Network Service Provisioning;
- Network Automation.

1.1 Programmable Network Infrastructures

The first one of these evolution trends revolves around having network resources that could be programmed dynamically through software. The most famous paradigm for that is Software Defined Networking (SDN). SDN was standardized in 2011 by the Open Networking Foundation (ONF), which is *a user-driven organization dedicated to the promotion and adoption of SDN, and implementing SDN through open standards [...] necessary to move the networking industry forward*[B1]. ONF is also the organization behind the development and distribution of OpenFlow, one of the most used switch protocols for providing an

open interface for controlling connectivity and flows within that connectivity in a Software Defined Network [B2].

The main aim of SDN is to support a scalable and dynamic variation of communication resources employing a unified, open, and programmable interface for controlling network switches from different vendors. SDN achieves this by decoupling the control plane (i.e. the part of the system that takes decisions on how to forward packets) from the data plane (i.e. the part of the system that physically receives, stores, and forwards the packets) [B3]. A typical application of this paradigm is Service Function Chaining (SFC). The concept of Service Function Chaining has been introduced to describe the deployment of composite networked services obtained by a concatenation, i.e., a *chain*, of one or more basic services, or *Service Functions* [B4]. Equivalently, a *Service Function Path* (SFP) is defined as the series of service functions that network traffic must traverse for a given service to be correctly delivered. SDN principles are typically adopted to enable dynamic traffic steering across the data plane interconnecting service functions [B5]. SDN concepts can also be used to enhance the efficiency and flexibility of the control and management planes of SFP deployments: in fact, the SDN architecture can be taken as a reference scenario for the definition of the SFC Control Plane, i.e., the Service Plane [B6]. The main idea behind SFC has been conceived as a possible answer to the need for improved and flexible management of middleboxes and service deployment that network operators and service providers have been facing in the past decade [B7], [B8]. Recently, SFC has become a hot topic in the research community [B9], and is part of a standardization initiative by the IETF [B4]. SFC makes use of a service-specific overlay that creates the required service topology, possibly spanning multiple technological or administrative domains [B10]. Such a service plane lies between the application and control planes and includes all the processes that allow the infrastructure to provide services to users and maintain the state of those services, relying on control and management plane functions to suitably program the data plane.

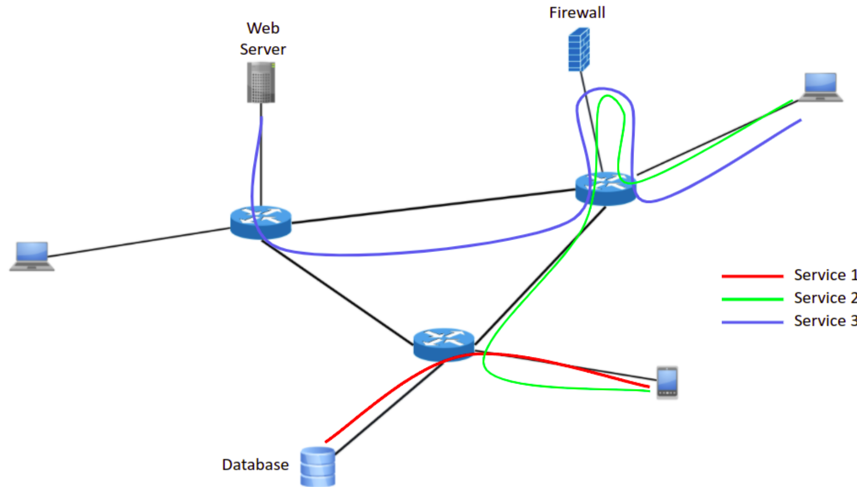


Figure 1.1: Simplified representation of three Service Function Paths over the same network topology.

Among the several research issues concerning SFC currently under investigation, it is worth mentioning service orchestration [B11], physical resources allocation to data plane components [B12], a trade-off between optimized performance and resource cost in SFP deployments [B13], service function overload and failure management [B14]. From a theoretical perspective, recent studies include the analysis of the algorithmic complexity of traffic steering in SFC, which is a particular case of the more general waypoint routing problem [B15], as well as performance modeling using network calculus concepts [B16].

1.2 Virtualized Network Service Provisioning

Virtualization has taken an important role in computing. Cloud computing has opened new business opportunities for different market sectors. Telecommunications is one of these, where the Network Function Virtualization (NFV) paradigm is decreasing the operational cost of infrastructure management while increasing its flexibility, thanks to the virtualization of physical network appliances. An Industry Stan-

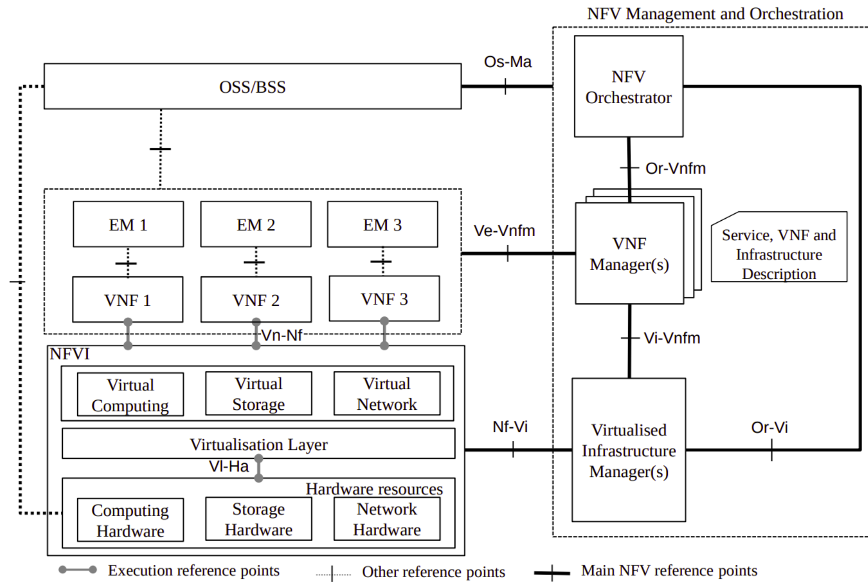


Figure 1.2: NFV reference architectural framework [B17].

dardization Group (ISG) of the European Telecommunications Standards Institute (ETSI), NFV, has defined an architecture to manage and orchestrate this type of system.

In [B17], the NFV-MANO (NFV Management ad Orchestration) architectural framework is presented Fig. 1.2. The most important components of this architecture are:

- NFV Orchestrator (NFVO): in charge of orchestrating the NFV infrastructure, triggering the actions to be taken to deploy the service required by the user;

- NFV Infrastructure (NFVI): the totality of all hardware and software components that build up the environment where Virtual Network Functions (VNFs) are deployed. It contains the physical resources that the virtualization layer can use to provide the virtual resources needed to host the required service;

- VNF Managers (VNFM): responsible for VNF lifecycle management (e.g., instantiation, update, query, scaling, termination);

- Virtualised Infrastructure Managers (VIM): provides the functionalities needed to interact with the NFVI.

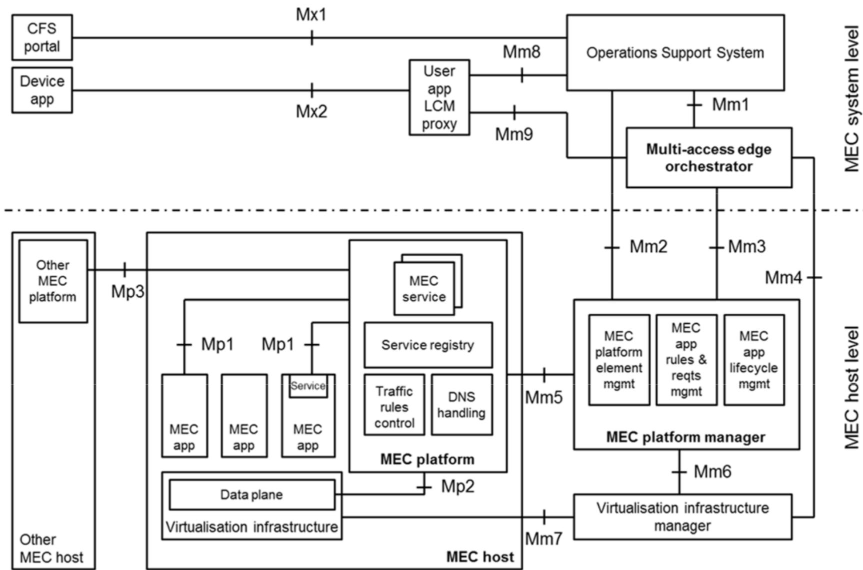


Figure 1.3: Multi-access edge system reference architecture [B19]

As a reference implementation of this system, I used Open Source Mano (OSM) [B18] for all the works requiring a MANO system. More details on this project will be given in the following chapters.

Another relevant evolution in this field is Multi-access Edge Computing (MEC). It is an ETSI ISG aiming at providing computing capabilities for IT and networking services at the edge of the network. Thanks to that, it is possible to host applications requiring high bandwidth and reduced latency. Furthermore, the applications could consume radio network information to better tune their behavior to the current working conditions. As for the NFV ISG, the group produced a reference architecture for these systems Fig. 1.3 [B19]. It follows similar principles as the NFV-MANO architecture, with an additional key component: the MEC Platform (MEP). The MEC Platform is an entity in charge of "helping" running applications by exposing a set of different Application Programming Interfaces. For example, the MEC 011 API [B20] enables MEC applications to register the services they are exposing to the Service Registry maintained by the MEC Platform, allowing other applications to discover these services and consume them by querying the registry. Furthermore, the MEC Plat-

form handles the Domain Name System (DNS) name resolutions for the applications. Finally, the MEC Platform can also expose other services to the applications, using standardized interfaces (e.g., MEC 013 for device localization) or custom ones.

Of course, these concepts bring new challenges to be addressed. Research activities tried to solve the open problem of deciding where the Virtual Functions should be deployed, i.e., which combination of available computing resources is the best one to host the required service. These decisions must consider the current status of the available infrastructure and the requirement of the desired service. Of course, these problems become even harder to solve with the increased geographical distribution of the computing resources (e.g., edge and extreme edge) and the variety of requirements demanded by modern types of services. At the same time, it's essential to understand which are the most common techniques and tools employed to build these systems. Consequently, their performance and limitations need to be studied and addressed to push forward their adoption. Furthermore, given the novelty of these paradigms, well-defined tools for them do not always exist, thus opening up the opportunity for researchers to propose new implementations advancing the state-of-the-art.

Regarding the platform choice for operating telecommunication services, it's worth recalling the initiative called CloudiNfrastructureTelcoTask Force (CNTT) [B21]. It was started by the Linux Foundation and GSMA, and it is now backed by a community of global telco operators and vendors. The main idea behind this task force is to define what a "telco cloud" is (i.e., a cloud infrastructure for NFV-based telco applications), by identifying a robust infrastructure model, selecting a set of architectures coherent with the model, and providing tools for their validation. In detail, they identified two main tools to operate this type of system: Openstack for virtual machine-based deployments and Kubernetes for container-based ones.

A perfect example showing where these new concepts take a pivotal role is 5G. Firstly, 5G adopts a cloud-native-oriented approach for the deployments of its internal components. Specifically, it relies

on a Service-Based Architecture (SBA), in which its principal functionalities (e.g., Session Management Function, Access Management Function, etc.) are built as independent software components talking through standardized APIs. Furthermore, 5G introduces a new

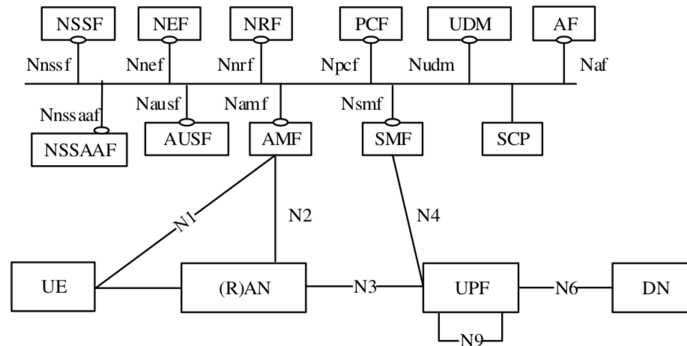


Figure 1.4: SBA 5G System architecture (Figure 4.2.3-1 [B22])

important paradigm that allows the coexistence of multiple logical networks on top of the physical one, enabling the support of different types of services with their specific requirements. This new concept, called network slicing, builds on top of the technologies presented in this chapter (i.e., network programmability and dynamic service provisioning). In detail, the three main network segments (i.e., Access, Transport, and Core) are sliced into different logical ones thanks to the aforementioned technologies. For example, on top of the transport network connecting the access and the core segments, several overlay networks can be deployed thanks to network programmability tools (e.g., SDN), each with their target QoS. Alternatively, the core segments could host many virtualized services managed by an NFV orchestrator based on the users' needs.

1.3 Network Automation

Both network virtualization resources and programmable network facilities offered by infrastructure providers should be considered as interconnected components of a holistic system, which allows service

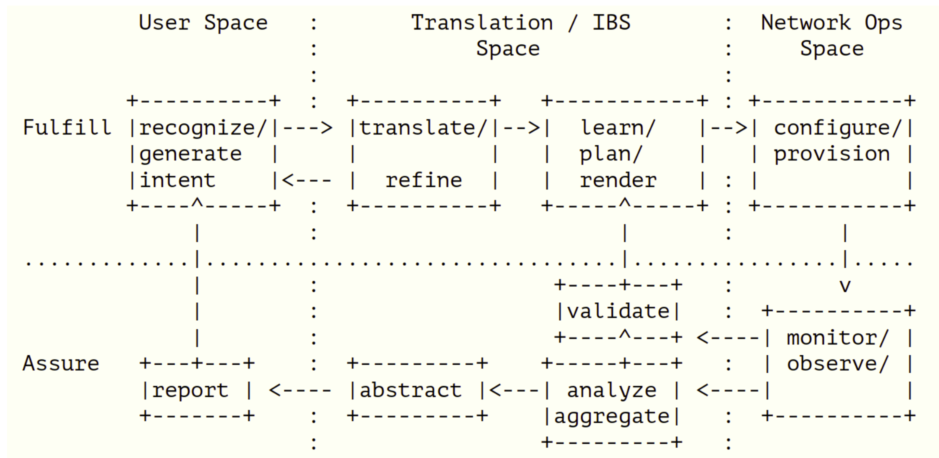


Figure 1.5: Intent Lifecycle [B23]

providers and/or customers to express service requirements, as well as to manage service lifecycle, by using high-level specifications that are independent of the particular solutions and technologies adopted. A step further in terms of network abstraction could be considered, to ease the network management burden, thus decreasing the overall operational cost. A declarative network service specification (i.e., an intent) is thus considered a promising approach to achieve the required level of abstraction, as it allows to declare the requested service in terms of "what" must be achieved and not "how" to achieve it. Specifically, in [B23], the Network Management Research Group (NMRG) defined an intent as:"A set of operational goals (that a network should meet) and outcomes (that a network is supposed to deliver), defined in a declarative manner without specifying how to achieve or implement them". In the same document, they also propose the lifecycle an intent should undergo inside an Intent Based Networking (IBN) system, Fig. 1.5.

After its generation, the intent is then translated into the set of actions required to fulfill the user's demands. The IBN system will then interact with the underlying infrastructure to provide the required service. Furthermore, it will continue to monitor it to verify the requirements are satisfied during the whole intent life span. Relevant works in the field of Intent-driven networks can be found in [B24].

Chapter 2

Programmable network infrastructure

In this chapter, the results obtained regarding network programmability will be presented. The first part of the chapter will describe how SFPs are created inside an OpenStack cluster, which technology they use, and how they perform. Therefore, it offers a benchmark for future works employing this technology and highlights some of its limitations. The second part will show a practical evaluation of dynamic traffic steering using Segment Routing and IPv6, extending the built-in features offered by OSM. These results are also published in [0] and [0]

2.1 SFC with Openflow

2.1.1 The OpenStack SFC extension

OpenStack is an open-source software platform consisting in a rich set of services allowing the deployment and management of public and private cloud infrastructures [B25]. Taking advantage of different kinds of virtualization technologies, ICT resources such as storage, CPU, RAM, and network are decoupled from a variety of vendor-specific implementations and exposed as abstractions to multiple tenants. OpenStack defines a consistent set of Application Programming Interfaces

(APIs) to manage those virtual resources as discrete pools, with which administrators and users can interact directly by means of standard cloud management tools, e.g., RESTful HTTP clients. The OpenStack platform can rely on a quite large set of extensions, including one dedicated to SFC [B26], which provides an API to support SFP creation and deployment in Neutron, the well-known OpenStack's network virtualization component.

The documentation of the OpenStack SFC extension defines a service function as a virtual or physical machine that perform a specific network function such as firewall, content cache, packet inspection, or any other function that requires processing of packets in a flow exchanged between two endpoints in the network. The extension allows for the creation of SFPs, it natively supports interaction with Open vSwitch (OvS), it implements a flow classification mechanism (i.e., the ability to classify traffic based on service-level characteristics), and it provides a vendor-neutral API.

The SFC extension defines and deploy a SFP according to a four-step approach. The first element to be instantiated is the *Flow Classifier* (FC), which is needed to classify the incoming traffic based on predefined policies (e.g., header matching rules), in order for the flow to be properly steered through the required set of service functions. In other words, the FC contains the set of matching criteria that will be used to determine whether a specific traffic flow must traverse the associated SFP or not. Those criteria can be specified by various parameters, spanning from data link to transport layer header fields, as well as OpenStack metadata, such as the port ID assigned by Neutron to the source and destination ports.

As a second step, *Port Pairs* (PPs) must be created. A port pair represents a service function instance that includes an ingress and egress port. In other words, it represents a single hop of the SFP, specifying the network ports, as defined by Neutron, attached to a given service function instance. A port pair can be either uni- or bi-directional, depending on how the associated service function can be traversed by the flow. Also, a port pair may have a *weight* associated

to it, to be used to perform load balancing over the SFP.

A *Port Pair Group* (PPG), whose definition is the third step in the creation of a SFP, is a collection of one or more port pairs. If a port pair represents entry and exit points of a particular service function instance, a port pair group can be considered as a list of different instances implementing the same service function. The definition of a port pair group enables load balancing for the corresponding service function: in fact, each new traffic flow traversing a given chain will be forwarded to one of the port pairs belonging to the same group according to a weighted round-robin policy, based on the specific weight that was assigned to each port pair at creation time. Another interesting feature is that a port pair group can be configured with a *tap* attribute. The port pairs belonging to this kind of group will not play an active part in the chain, as they will simply receive a copy of the flow traversing the SFP, without the need of forwarding it to the next hop. This functionality can be useful for those VNFs, such as a packet analyzer, that do not perform any action on the traffic passing though the chain but that just need to receive information about the incoming flows.

Finally, a *Port Chain* (PC) is instantiated as a binding between one or more flow classifiers and an ordered list of port pair groups, thus defining and implementing the actual SFP. All incoming traffic flows matching the rules of the flow classifier(s) specified in the port chain will have to traverse a port pair of each port pair group associated to the port chain, according to the specified order. A port chain may be uni- or bi-directional as well. In the former case, the chain will only be traversed by flows matching the criteria specified in the flow classifiers, whereas, in the latter case, the chain will be traversed also in the opposite direction by flows matching the “symmetric” version of those matching criteria. A visual representation of the different components of an SFP inside Openstack is depicted in Fig. 2.1. Moreover, the SFC extension allows linking together different port chains, creating the so called *Service Function Graph*.

After a SFP (i.e., a port chain) has been defined and deployed, the

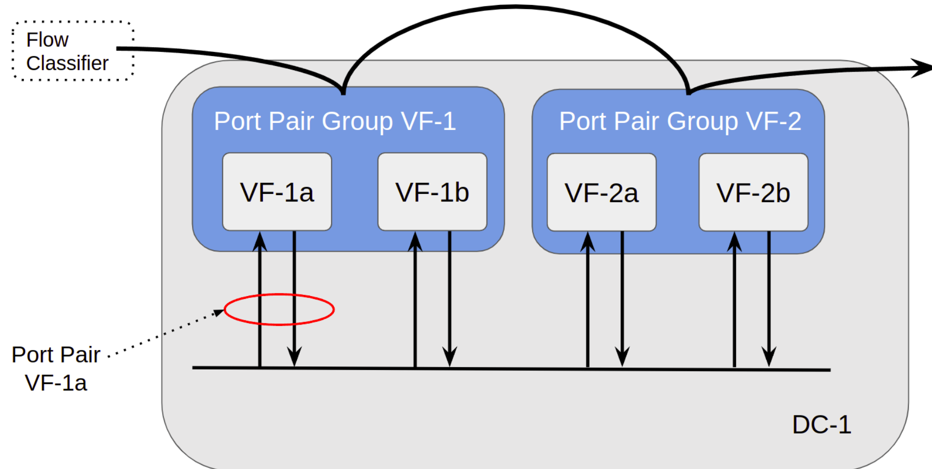


Figure 2.1: Visual representation of the different components of an SFP inside Openstack, namely Flow Classifier, Port Pair, and Port Pair Groups.

relevant traffic steering policies are installed in the OpenStack network subsystem by adding suitable OpenFlow rules to the OvS-based virtual switches found in each compute node. In particular, the integration bridge (`br-int`), to which all instances running in a given physical node are connected, and the tunneling bridge (`br-tun`), from which tunnels depart toward other physical nodes, are configured with internal and external traffic steering rules, respectively. In particular, when the SFP involves instances running on multiple compute nodes, correct forwarding operations on the physical network infrastructure require to identify which hop of which SFP each packet is currently traversing. To this purpose, each packet must carry a *Service Path Identifier* (SPI) and a *Service Index* (SI), and then be encapsulated using a suitable data plane transport technology. The current implementation of the OpenStack SFC extension supports either Multi-Protocol Label Switching (MPLS) [B27] or Network Service Header (NSH) [B28] encapsulation.

Two different kinds of traffic steering rules are added to the virtual switches by the SFC extension when a new port chain is deployed. First, each compute node includes matching rules that follow the crite-

ria as defined by the corresponding flow classifier. These rules are used to intercept packets transmitted by source nodes or service functions, which are typically unaware of the underlying traffic steering technology being used, and to push the additional headers required by the chosen encapsulation method. Second, additional steering rules are included, with matching conditions that depend on the encapsulation technology and refer to the specific switch ports to which the relevant service functions are attached.

All the interactions with the OpenStack SFC extension can be carried out either by using the native command line utilities, or through its REST API. The latter is a more general approach that enables service chain management and orchestration from authorized third-party applications through a standardized interface. Therefore, in this section, we choose to evaluate the SFC extension response time via the REST API, testing the creation of increasingly long SFPs. This result can be used as a benchmark to assess the overall time needed from an Openstack environment to deploy the required SFP. We also show that, after the deployment of an SFP, traffic is correctly steered through specific instances that would not otherwise be crossed with traditional forwarding, thus validating the SFC plugin functionalities.

2.1.2 Experimental setup

To measure the performance of the SFC extension, we installed OpenStack (Rocky release via Devstack) on a test bed consisting of 7 baremetal servers from the CloudLab facilities [B29], as shown in Figure 2.2. Out of those servers, 4 are used as OpenStack compute nodes (`contr01`, `comp02`, `comp03`, and `comp04`), including one acting as a controller (`contr01`). These nodes are physically connected through separate management and data networks. Out of the remaining servers, 2 of them (`ovs01` and `ovs02`) host instances of OvS, thus implementing an SDN-enabled data network[1], while the last one (`ext`)

[1] Although in this chapter we do not take advantage of the SDN capabilities of the physical data network and rely only on the SFC extension to the OpenStack networking service, we plan to use the same test bed setup in the near future

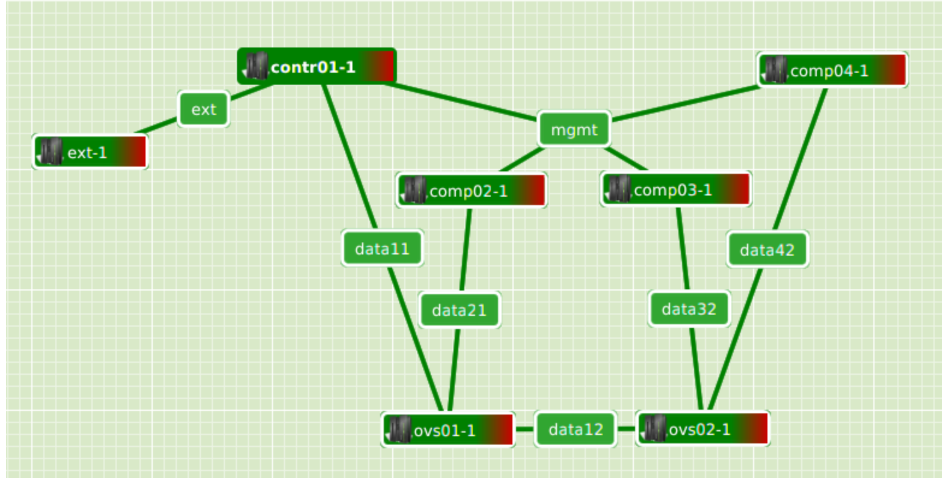


Figure 2.2: Physical setup of the OpenStack test bed deployed in CloudLab, as displayed by the jFed experiment management tool.

is used as access gateway to reach the instances. Multi-tenant traffic isolation over the data network is operated via VXLAN tunneling. Since the operating system kernel does not support NSH, we adopted MPLS for SFC encapsulation [B30]. The packet overhead introduced by VXLAN tunnels plus MPLS encapsulation is such that we had to reduce the MTU of all instance interfaces to 1418 bytes. All physical servers are equipped with 10 Gigabit Ethernet interfaces.

Virtual machines and networks were instantiated in the OpenStack cluster, yielding the logical network topology shown in Figure 2.3. With the chosen topology, instances representing endpoints (i.e., customer nodes) reside on the `customerVxlan` network, which is also shared with service function instances. The latter are also connected to a dedicated `internalVxlan` network, used for intermediate communication along the SFP. Having two distinct networks in the SFC test bed enables the emulation of realistic scenarios where a single service function may need to have interfaces connected to two separate networks, e.g., as in the case of WAN Accelerator or NATs. While this topology opens the possibility to create SFPs with asymmetric

to investigate a possible integration of the SFC extension with an external SDN controller in charge of programming the physical network

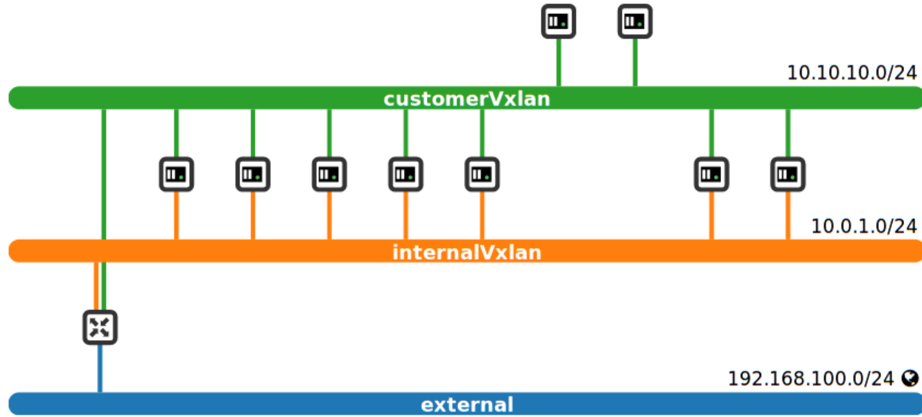


Figure 2.3: Virtual network topology and deployed instances. The boxes in the image represent the computing resources (i.e., VMs) deployed in the Openstack cluster.

connectivity, the actual direction of the traffic crossing a given service functions can still be configured when creating the corresponding port pair. The placement of the VNF instances over the four OpenStack compute nodes is depicted in Figure 2.4, where each block represents a physical node and the circles inside it symbolize the virtual machines (i.e., VNFs, source and destination) running on it. Since we are interested in evaluating the performance of the traffic steering mechanism implemented by the SFC extension, the deployed VNFs do not apply any specific processing or conditioning action to the traffic flows traversing them, apart from simple packet forwarding.

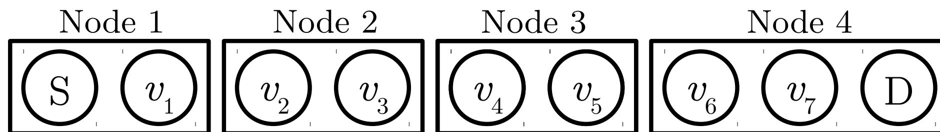


Figure 2.4: Instance placement over the physical nodes of the test bed.

In order to carry out the performance assessment tests, a REST client for the SFC extension API was developed, in the form of a Python script. The client operates calls to the mentioned REST API in order to define and deploy a predefined service chain. The details of

the requested SFP are provided to the client by a user-defined JSON file, in which the user can specify the endpoints of the SFP as well as flow classifying parameters and the ordered sequence of service functions to be traversed. The structure of the JSON file is as follows:

```
{
  "source": "node_name",
  "destination": "node_name",
  "match_fields": {
          "match_field": "field_value",
          ...
  },
  "chain": [
    {
      "name": "vnf_name",
      "direction": "direction_value"
    },
    ...
  ]
}
```

where: `source` and `destination` represent the endpoint nodes of the service chain, expressed as names of instances known to OpenStack; `match_fields` is a list of key-value pairs identifying matching fields (e.g., `protocol`) and values (e.g., `udp`) to be specified in the flow classifier criteria; `chain` is the ordered list of service functions to be traversed, each specified by the `name` of the corresponding VNF instance; for each service function, the `direction` attribute is used to specify whether the service function must forcibly eject the output traffic on the internal network (`in`), on the customer one (`out`), on the same one it received it from (`same`), or whether the VNF instance must simply receive a copy of the flow (`tap`).

After parsing the JSON file and retrieving the service chain specifications, the SFC client executes the four steps of definition and deployment of a SFP that were described in Section 2.1.1. The sequence of operations carried out by the client and the corresponding timings

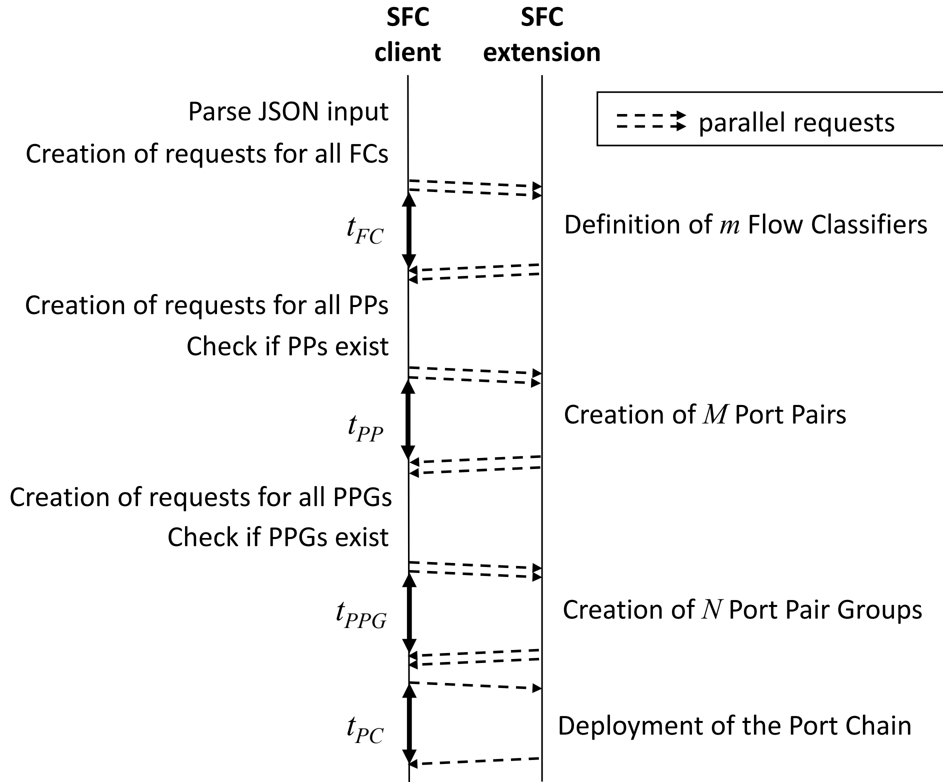


Figure 2.5: Execution flow and related timings of the SFC REST client application for a single port chain deployment.

are sketched in Figure 2.5, considering the general case of the complete deployment of a single port chain, including m flow classifiers, M port pairs, and $N \leq M$ port pair groups.

2.1.3 Experimental results

To evaluate the response time of the SFC extension REST API, we timed the execution of each of the four requests listed in Figure 2.5 needed to deploy a single SFP. We measured the response time at the client side for an increasing number N of traversed service functions, with a single flow classifier ($m = 1$) and a single port pair per port pair group ($M = N$). We considered each of the four steps separately, so as to be able to determine how each operation scales with the chain length. We repeated the tests ten times for statistical significance. It

is relevant to mention that these operations can't be parallelized. Only the Flow Classifiers can be defined while creating the Port Pairs or the Port Pair Groups. The reason is that most of these steps depend on previous ones. For example, the Port Pair Groups creation depends on the identifiers assigned to the Port Pairs created in the prior step.

Figure 2.6 shows the average response time for each of the four REST endpoints, as well as the average total time needed to define and deploy a SFP via the REST interface provided by the SFC extension. Our experiments show that a few seconds are needed to deploy a SFP, with a linear increase of approximately half a second for each service function added to the chain. The timing of client-side operations performed before and after each request are not shown here because their duration is negligible with respect to the SFC extension response time. We verified experimentally that the REST API of the SFC extension responds only after executing the request and not immediately after receiving it as an acknowledgment. Therefore, the measured time includes the time needed for the SFP to be actually deployed. That was verified by monitoring the flow tables of the Openstack Switches (e.g., `br-int`) and by checking that the required flow rules were added to the tables before the REST API response..

The time needed to define the flow classifier, t_{FC}, appears to be unaffected by the increase in SFP length. This is reasonable, not only because we assumed $m = 1$, but also because to define a flow classifying policy the OpenStack controller does not need to interact with the compute nodes involved in the SFP yet. More precisely, for the first three steps the controller must only update its internal database with the information received from the SFC client. In fact, it is possible to see that t_{PP} and t_{PPG} do increase with N, but not as much as t_{PC}. This can be explained by the increasing number of entries that the controller has to introduce inside the database when creating port pairs and port pair groups. On the other hand, the step which is most affected by the increase in SFP length is the actual deployment of the port chain (t_{PC}). This is reasonable too, as the deployment of the port chain requires the controller to install flow rules on virtual

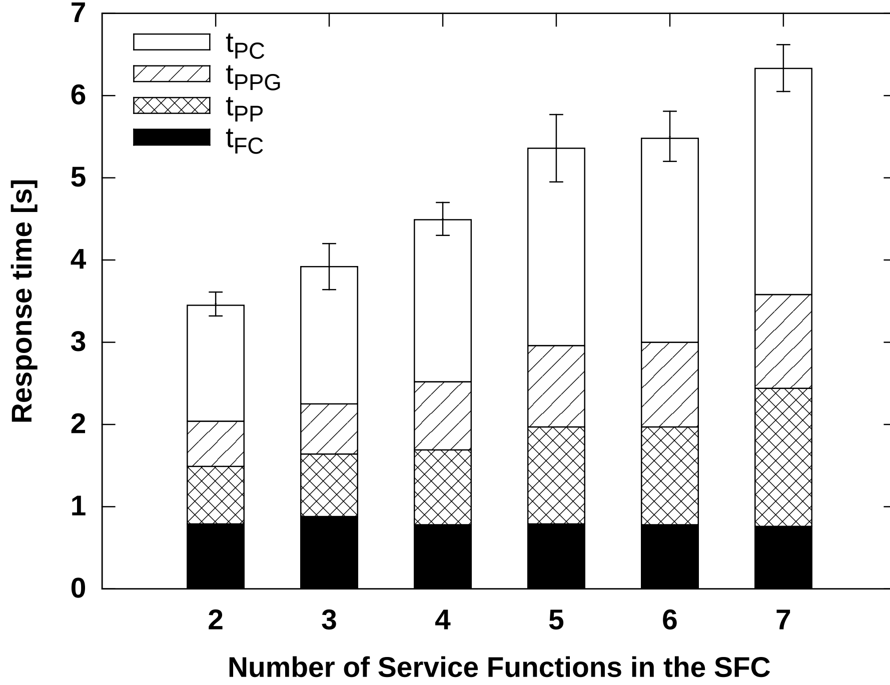


Figure 2.6: Average response time (with 95% confidence intervals) of the SFC extension REST API as a function of the length of the SFP. The histogram shows the contribution of each of the four steps required to deploy a SFP.

switches residing in all physical compute nodes involved in the SFP.

In order to prove the correctness of the traffic steering operated by the SFC extension, a proof-of-concept test was carried out. We defined the following 3-hop SFPs and deployed them:

$$\text{SFP}_1 : S \rightarrow v_1 \rightarrow v_4 \rightarrow v_7 \rightarrow D \tag{2.1}$$

$$\text{SFP}_2 : S \rightarrow v_2 \rightarrow v_4 \rightarrow v_6 \rightarrow D \tag{2.2}$$

$$\text{SFP}_3 : S \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow D \tag{2.3}$$

An `iperf3` session was launched between the two endpoints for each SFP using different UDP port numbers. The first session (1 Mbps, over SFP_3) lasted for 30 seconds, the second one (2 Mbps, over SFP_2) for 60 seconds, and the third one (3 Mbps, over SFP_1) for 90 seconds.

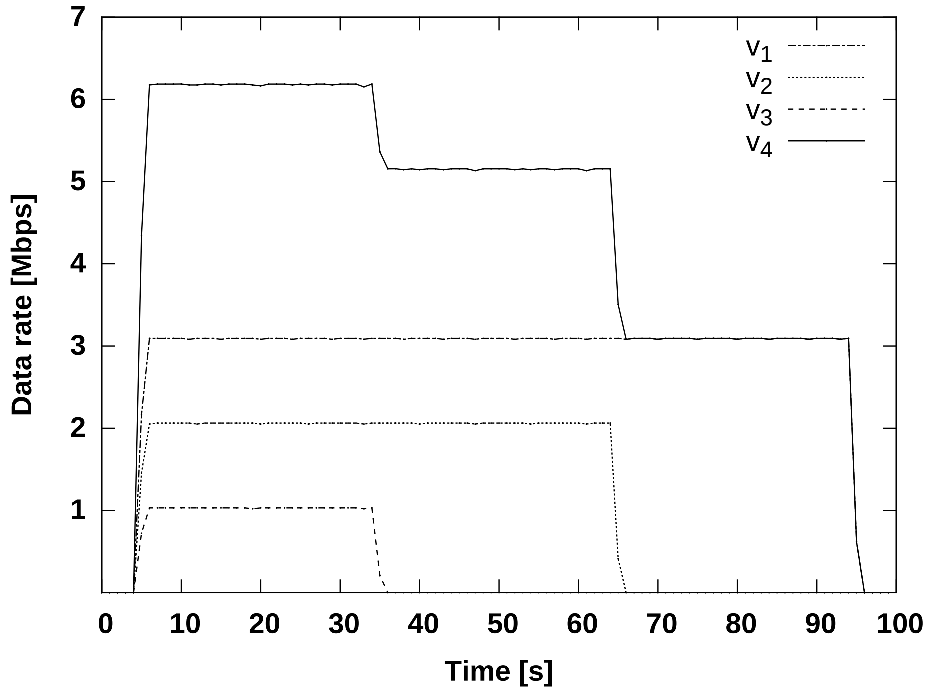


Figure 2.7: Throughput measured at the egress interface of some relevant service function instances.

The throughput measured at the egress port of some of the involved service functions is shown in Figure 2.7. v_4, which is crossed by all chains, sustains the combined traffic of the three SFPs. When the `iperf3` session over SFP_3 ends, i.e. when the throughput measured at v_3 goes to zero, the remaining active traffic keeps traversing v_4 as well as the other service functions of SFP_1 and SFP_2. In the last 30 seconds, only traffic over SFP_1 is still active, therefore the overall traffic crossing v_4 is coincident with the traffic crossing v_1. This outcome proves the correct deployment of the requested SFPs in our test bed.

We then carried out another set of experiments to evaluate the impact of the introduction of a service chain of increasing length on the TCP throughput measured between two endpoints. Table 2.1 reports the list of SFPs that were deployed to perform the TCP throughput test, where chain C_i includes i service functions. The throughput measured for the first chain C_0 is used as a reference, as it is the

maximum achievable value obtained from source S to destination D without any VNF in between. Then the order of the VNFs chosen for the other chains is not random. From C_1 to C_4 the chain length is increased by adding one VNF from each physical node according to the placement shown in Figure 2.4, starting from the nodes where S and D reside. For the last two chains, the order of the two additional VNFs is chosen as to maximize the number of transitions through different compute nodes over the physical network. This can be considered as the worst-case scenario in terms of throughput.

The throughput values of TCP sessions generated with `iperf3` over the chains in Table 2.1 are reported in Figure 2.8. While the maximum achievable throughput for C_0 is above 8 Gbps, as expected the use of SFC affects the performance of the network, reducing the throughput for each VNF added to the chain. Interestingly, a significant penalty is caused by the transition through different physical nodes, as in the case of chains C_2 to C_6. So the location of a VNF can drastically change the throughput achievable between source and destination. This is also easy to see from the results presented in Table 2.2, which reports the throughput measured with just one VNF between S and D and placed in three different positions: co-located with S, co-located with D, or in a separate physical node. From these results it is clear that the transit through the physical network, which involves MPLS labelling and VXLAN encapsulation, as well as forwarding through the OvS switches implementing the SDN-enabled data network, introduces a significant processing overhead that limits the throughput between the endpoints of the chain. The adoption of packet processing acceleration technologies could provide a possible solution to this performance problem [B31].

Lastly, a brief consideration is due on the port pair group tap attribute mentioned in section 2.1.1, as implemented by the SFC extension. After some tests we found that this functionality actually works only if the VNF that is supposed to act as a tap is running on the same physical node as the previous hop of the chain. In other words, if the mirrored flow has to go through the OpenStack tunneling

Table 2.1: List of service chains used for TCP throughput measurements

Chain hop sequence
$C_0 = S \rightarrow D$
$C_1 = S \rightarrow v_1 \rightarrow D$
$C_2 = S \rightarrow v_1 \rightarrow v_6 \rightarrow D$
$C_3 = S \rightarrow v_1 \rightarrow v_2 \rightarrow v_6 \rightarrow D$
$C_4 = S \rightarrow v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_6 \rightarrow D$
$C_5 = S \rightarrow v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow v_6 \rightarrow D$
$C_6 = S \rightarrow v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_3 \rightarrow v_5 \rightarrow v_6 \rightarrow D$

Table 2.2: Single VNF performance for different locations

VNF location	Same as S	Same as D	Other node
Throughput (Gbps)	7.01	3.39	1.99

bridge, then it will not reach the tap VNF. This is probably caused by a bug in the SFC extension version we tested, which does not install the necessary OpenFlow rules in the tunneling bridge to properly mirror the flow. Furthermore, another bug of the SFC extension related to the NSH was identified. The OpenFlow version employed couldn't process flow entries requiring an NSH encapsulation and decapsulation action on the same virtual switch, thus failing the creation of an NSH-based SFC composed of two SF hosted in the same physical node (i.e., served by the same integration bridge).

2.2 SFC with Segment Routing

Another important technology adopted to realize dynamic traffic steering and achieve adaptive service composition is Segment Routing. Segment routing (SR) is based on the well-known source routing concept coupled with the SDN paradigm. By inserting an ordered list of Segment IDs (SIDs) identifying the VNFs to be traversed, it is possible

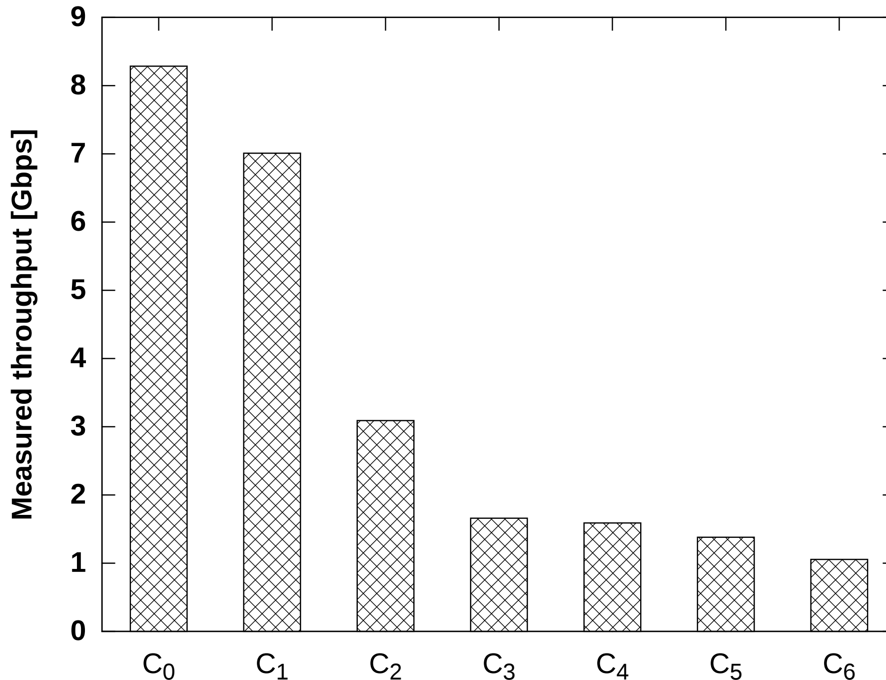


Figure 2.8: TCP throughput measured with increasing chain length (95% confidence intervals are not shown due to their negligible width).

to ensure an end-to-end connection that crosses the desired VNFs between the ingress and egress routers of the network. The general segment routing architecture has been recently standardized [B32], laying the basis for both the MPLS and IPv6 implementation versions. On top of that, the required data plane functionalities have been defined in order to achieve service programming [B33][2]. Furthermore, this architecture has been extended to support network programmability using SRv6 (i.e., Segment Routing with IPv6) [B34], defining the list of functions needed to enable advanced networking functionalities to support service programming. An example of these functionalities is the support of overlay networks. Thanks to these extensions, the segment routing architecture has become one of the most relevant solutions to realize SFC. A first solution to integrate SRv6 inside an NFV infrastructure has been proposed in [B35]. However, the solution that we

[2] The IETF Draft is now in expired and archived state

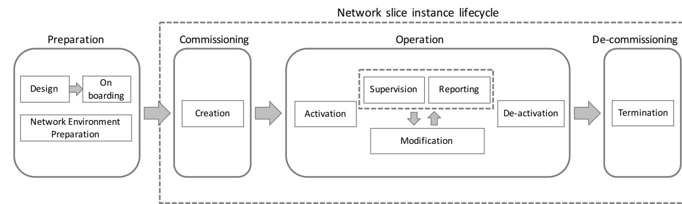


Figure 2.9: Schematic of the 5G network slice lifecycle.

are presenting here goes beyond that, as we properly define the operations to be performed by the different NFV-MANO components. We also propose a possible implementation with well known open source software platforms. Several metrics have been introduced to measure the performance of different southbound API solutions to configure SRv6 devices, both in terms of time required to send a command for a new SRv6 route and in terms of CPU utilization [B36]. The impact of an insertion of a new steering rule on packet loss is also evaluated. An open source project that grants the possibility of deploying a Linux Node acting as an SFC proxy [B4] for unaware service functions inside a segment routing enabled infrastructure has also been developed [B37]. For the sake of clarity, following the definition given in [B4], an unaware service function is a service function in a given SFP that can't decode the mechanism used in the chain to steer traffic (e.g., it can't understand the NSH header). For this reason, an SFC proxy is needed to decode and re-encode the packets of the SFP before and after the unaware service function.

2.2.1 The Network Scenario

The lifecycle of a 5G network slice is decomposed in a number of sub-phases [B38], as sketched in Fig. 2.9. In this work we focus on: preparation, creation, activation, and modification. In this section we describe how to integrate NFV-MANO and segment routing to implement those phases. This integration would allow a method to create dynamic SFP with segment routing using the interfaces granted by the NFV-MANO framework. Without loss of generality, the network

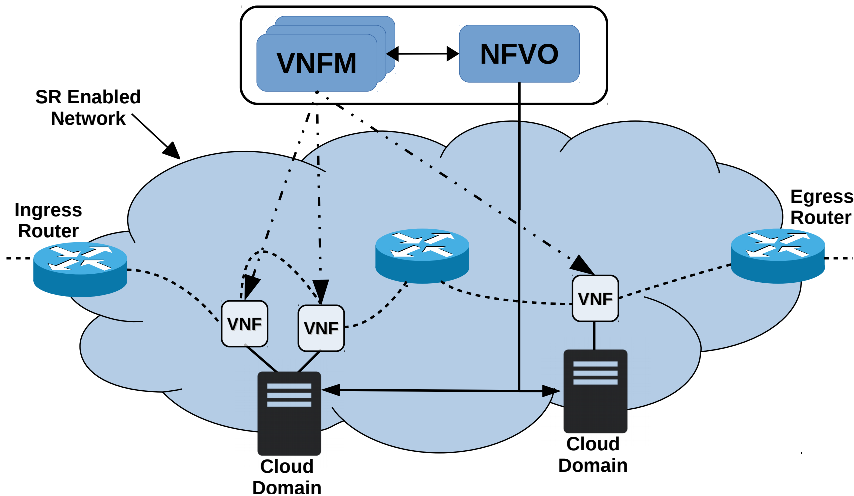


Figure 2.10: Network scenario for NFV-MANO and segment routing integration.

scenario considered here is depicted in Fig. 2.10. The NFV-MANO framework deploys the Virtual Network Functions (VNFs) composing the network service, possibly across different cloud domains. The VNFs are interconnected by a segment-routing-enabled network.

Segment Routing

With reference to Fig. 2.10, the ingress router classifies incoming packets and inserts in their header the SIDs corresponding to the ordered list of cloud domains where the VNFs needed for the particular service are deployed. The ingress router can obtain this information by interacting with the orchestration layer (i.e., the NFVO). Once the packets reach a given cloud domain, all the forwarding operations inside that domain are performed by inserting an additional stack of SIDs representing the ordered list of VNFs that need to be traversed. For instance, the ingress router of the SR network may classify the flow corresponding to a given service as `<CD1,CD2,ER>`, meaning that the required VNFs are hosted in “Cloud Domain 1” and “Cloud Domain 2”. This does not require the knowledge of the exact SIDs associated to the single VNF, which can be kept local to each cloud domain.

Then at the ingress of each cloud domain, after a local classification step (e.g., based con the traffic characteristics), an additional stack of SIDs is added to incoming packets, containing the list of VNFs the packets have to traverse inside of the cloud domain, for instance `<F1,F2>`. This secondary list is then removed before the packets leave the cloud domain.

The described stacking operations require that each cloud domain is equipped with SR ingress and egress routers that work similarly to the ones in the SR-enabled network. Several solutions exist to achieve this. For example, it is possible to use a single physical router at the edge of the cloud domain acting both as an ingress and egress router for all services. Alternatively, a VNF dedicated to a given service (or class of services) can be used as an ingress and/or egress router. In the latter case, the operations of inserting and removing the local SIDs can be easily and dynamically controlled by the relevant VNFM, thus enhancing the flexibility of the proposed solution. This solution can be further improved by using more than one SID for each cloud domain: reserving a SID to each tenant (or customer) of the cloud domain allows to keep their SID lists separated, enabling differentiated SFC management.

NFV-MANO Operations

To deploy and dynamically reconfigure the SFC within each cloud domain, the NFV-MANO components must perform a number of configurations. We describe these operations adopting the *Day 0/1/2* terminology commonly used in network automation. *Day 0* configurations are those related to the initial state of the VNF instance, including information such as the image to be used, its computing characteristics (e.g. RAM, storage, CPUs), and the initial network configuration. *Day 1* configurations include the sequence of operations to be performed immediately after launching the instance. For example, configuring additional network features, enabling system parameters, installing packages and applications setup. Finally, *Day 2* configurations relate to any additional reconfiguration made during

the lifecycle of the instance.

With reference to our network scenario, *Day 0* configurations are made jointly by NFVO and VIM that select the required VNFs and their characteristics based on the service requirements. The NFVO is able to get this information from the Network Slice Template (NST) derived from the NEtwork Slice Type (NEST), which is defined according to the Service Level Agreement (SLA) required by the customer of that particular service, as expressed by GSMA [B39].

Day 1 configurations instead can be performed by either the VNFM(s) or the VIM depending on the specific solution adopted. In this specific scenario it is required to enable the processing of SR packets in each VNF. An alternative *Day 1* configuration consists in the insertion of the first SIDs of an SFC that the user wants to deploy immediately.

Lastly, *Day 2* configurations are typically performed by the VNFM(s), that can add, remove or modify the list of SR functions present in each Service Function Path, as well as act on the routing table of the node, if necessary.

Mapping with 5G network slice lifecycle

All the operations described before can be easily mapped inside the Network Slice Infrastructure (NSI) lifecycle [B38]. Referring to Fig. 2.9, *Day 0* configurations are part of the *Creation* step in the *Commissioning* phase, whereas *Day 1* and *Day 2* are the *Activation* and *Modification* steps inside the *Operation* phase, respectively.

2.2.2 Test bed Implementation

In this section we describe how we implemented the network slice lifecycle management using open-source software tool currently available, such as Open Source MANO and OpenStack. We applied our configurations on the vanilla versions of those tools, without any change to the source code.

2.2.3 OpenSource MANO

For this work we made use of Open Source MANO (OSM) [B18], an ETSI-hosted open source project that allows to develop an Open Source NFV Management and Orchestration (MANO) software stack compliant to the ETSI specifications [B17]. OSM consists of different functional blocks.

The most important ones, shown in Fig. 2.11, are the NBI (Northbound Interface), the LCM (Life Cycle Management), the RO (Resource Orchestrator) and the VCAs (VNF Configuration Adapter).

The NBI is in charge of receiving requests from the user, from either the GUI or the command line, checking whether they are compliant with the Information Model (IM) of OSM, and passing them on to the LCM. The LCM is the component that supervises the whole process of creation, management and deletion of the different network services. It takes the requests from the NBI and it interacts with the other functional blocks to serve them. The RO is the component that interacts directly with the VIMs (e.g. OpenStack, AWS or VMware vCD) requesting or freeing up the resources needed by the network service. Finally, the VCAs are the components used to perform *Day 2* configurations on the VNFs of the service. More specifically, these components are implemented leveraging JUJU proxy charms. JUJU is an open source project backed by Canonical which aims at simplifying the deployment and configuration of applications over different types of infrastructure [B40]. To interact with the different components of an application, JUJU uses *charms*, which are a collection of actions (i.e., on-demand functions) employed to perform *Day 1* and *Day 2* configurations for the application. JUJU supports different types of charms, however OSM uses only one of them, the *proxy charms*. This type of charms can work both with Physical and Virtualized Network Functions. Moreover the whole charm logic runs in separate Linux Container Daemon containers, one for each VNF, running on the machine hosting OSM. The required commands needed for *Day 1* and *Day 2* configurations are exchanged between the LXD container and the relative VNF through Secure SHell.

In OSM, composition of network services is obtained by using two types of descriptors, both written in the YAML format. The first one is the VNF descriptor [B41] which defines the characteristics of the virtual function, including quantity and type of interfaces, name of the image for the virtual machine, *Day 1* configuration files (e.g., cloud-init [B42]) and a list of possible actions that can be launched when needed (i.e., *Day 2* configurations). The other type of descriptor is the Network Service Descriptor [B43] which defines the list of VNFs composing the service, and their interconnection.

With the JUJU tools it is possible to build the proxy charm package containing all the actions a VNF needs. In addition, it is possible to define the set of commands (e.g., bash commands) that has to be launched in the VNF whenever a specific event takes place.

The actions and their parameter defined in the proxy charm must be included in the descriptor of the VNFs where we want to use them. The syntax is structured as follows:

```
vnfd:vnfd-catalog:
  vnfd:
  ...
    vnf-configuration:
      juju:
        charm: CHARM_NAME
      config-primitive:
      - name: ACTION_NAME
        parameter:
        - name: PARAMETER_1
          data-type: STRING
          default-value: ''
          ...
```

The `CHARM_NAME` must correspond to the one of the charm included inside the package of the VNFD. The same applies also to `ACTION_NAME` and its parameter, that must match the ones in the `actions.yaml` file of the proxy charm folder. It is important to notice that this definition is completely independent from the characteristics chosen

for the Virtual Deployment Unit of the VNF. After the definition of the VNFD it is possible to use it to compose Network Service Descriptors. For this particular implementation we defined a VNFD able to support all the available SRv6 functionalities [B44].

Hardware configuration

To recreate the scenario described before, we used three bare-metal servers hosted by the CloudLab facilities [B29]. Two of them were used both to host an Openstack node (Stein release via Devstack) acting as controller and compute node. The last server was used to run OSM Release SIX, an open source solution compliant with the ETSI MANO framework. The same node was also used to emulate the network interconnecting the two different cloud domains. Finally, for simplicity we placed the ingress and egress router inside Cluster1 and Cluster2 respectively. Each one of the virtual machines deployed in the Openstack clusters used a clean image of Ubuntu 18.04.2 LTS with kernel version 4.15.0-50-generic, which supports the creation of SRv6 functions using iproute2.

2.2.4 Experimental Results

Validation

To prove the feasibility of the proposed solution, we present a simple scenario composed by two services with different priorities. Both consist of three different VMs, one acting as source of the traffic, another one as destination, and the last one as an example of a possible VNF required from the service. A single VNF was used to ease the readability of the experimental validation. However, the approach would not change with a larger number of VNFs. Recalling the scenario described in Section 2.2.2, the first two VMs can be considered as the ingress and egress point of the whole cloud domain, respectively. Nonetheless, our solution would also work in deployments with just one ingress and egress point shared by all services hosted by the cloud domain. For this demonstration, we choose to have specific ingress and

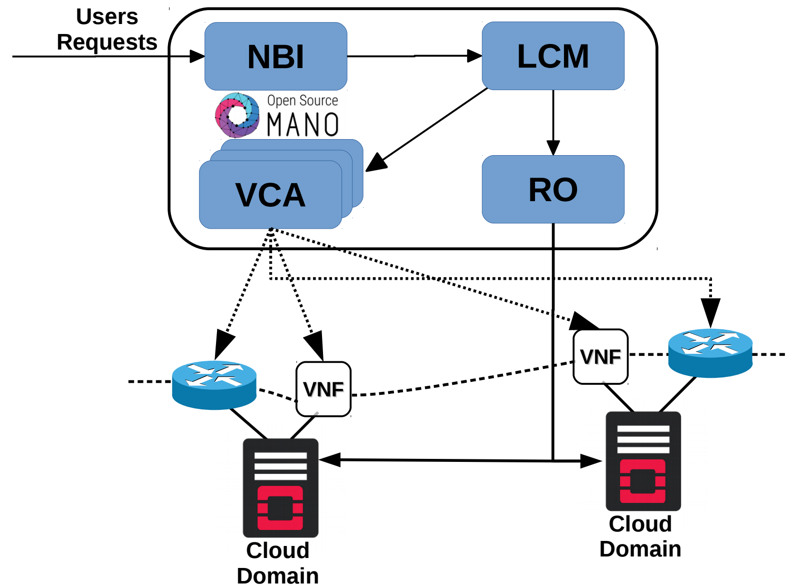


Figure 2.11: Proposed Scenario.

egress nodes for each service to ease the traffic classification phase (i.e., each service is identified univocally with its ingress point).

In our test scenario, the bandwidth of the service with lower priority must be reduced when the one with higher priority is instantiated. This can be achieved simply by adding to the segment list of the service with lower priority another VNF, acting as a Traffic Shaper, which can then be removed when the other service finishes transmitting its traffic. This way it is possible to validate both the setup of new SFCs and their dynamic reconfiguration. Figures 2.12 and 2.13 show the behavior in terms of bandwidth of the low priority service and of the high priority one, respectively. The former reports measures obtained from the destination of the packets (LP-D), the VNF of the service (LP-VF) and the VNF acting as a traffic shaper (LP-TS), which is active only when the higher priority service is present. The latter reports measurements obtained from the destination (HP-D) and from the VNF (HP-VF) of the higher priority service. For the first 50 seconds, the lower priority service is the only one active, therefore it is

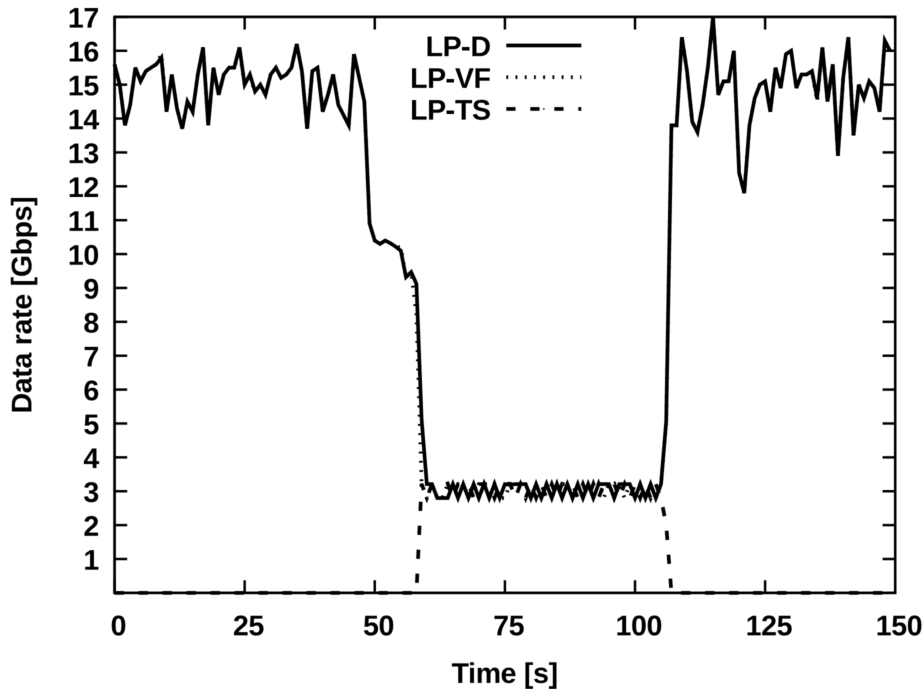


Figure 2.12: Throughput achieved by the lower priority service.

saturating all the available bandwidth. Then the higher priority service starts and the two services share the channel for about 5 seconds, which is the time needed from the system to reconfigure the segment list of the lower priority service in order to include the Traffic Shaper, which limits the throughput to 3 Gbit/sec. A zoomed-in version of this process can be seen in Figure 2.14. After the end of the higher priority service, the other one can go back to using the whole available bandwidth. This is accomplished by updating the segment list of the chain again, and this also explains the delay between the end of the higher priority service and the rise in achieved throughput of the lower priority one.

Performance Evaluation

We evaluated the amount of time needed to deploy chains of increasing length inside a single cloud domain. It is important to recall the different steps required by OSM to setup the instances of the service

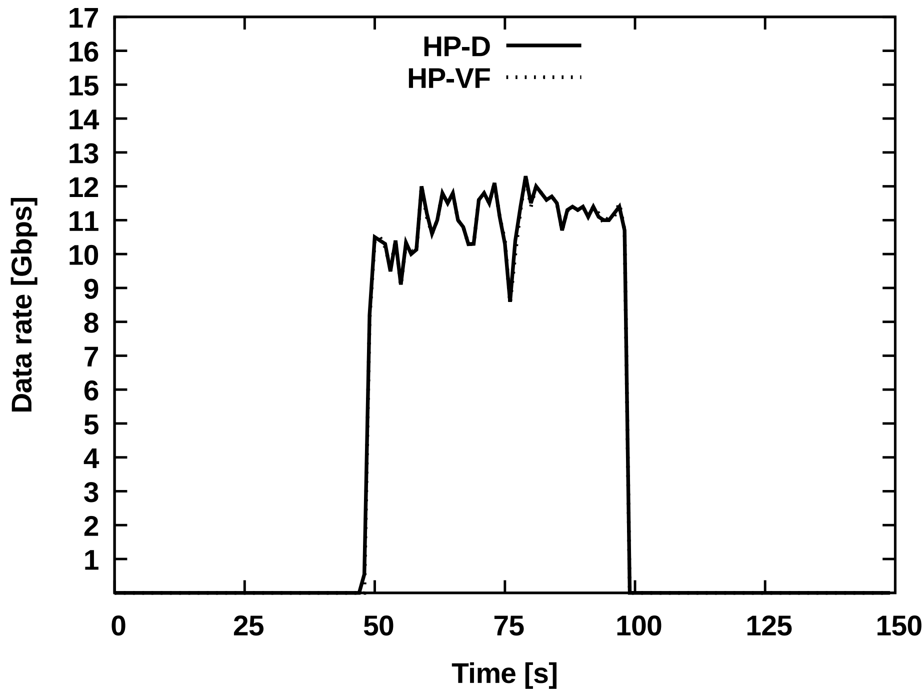


Figure 2.13: Throughput achieved by the higher priority service.

and their VCA container. The LCM of OSM first instantiates each one of the Ubuntu-based containers (t_1) that will run the VCA in charge of controlling the VNF of the service. Then, it proceeds with the installation of the proxy charms components (at t_2) and the installation of an ssh key inside each one of them (at t_3). Then the LCM contacts the RO (at t_4). Finally, the VCA containers obtain addresses on the management network, and the LCM verifies their reachability, the correctness of the SSH parameters defined in the VNF descriptors, and oversees the application of the desired *Day 1* configurations (at t_5).

By analysing the results we can see how much the setup of the LXC containers impacts on the time needed to deploy the overall service. However, it is important to highlight the fact that this affects only the deployment phase, as the additional configurations do not require to go through the whole process again. In fact, after the deployment, all the configurations will be made through commands launched from

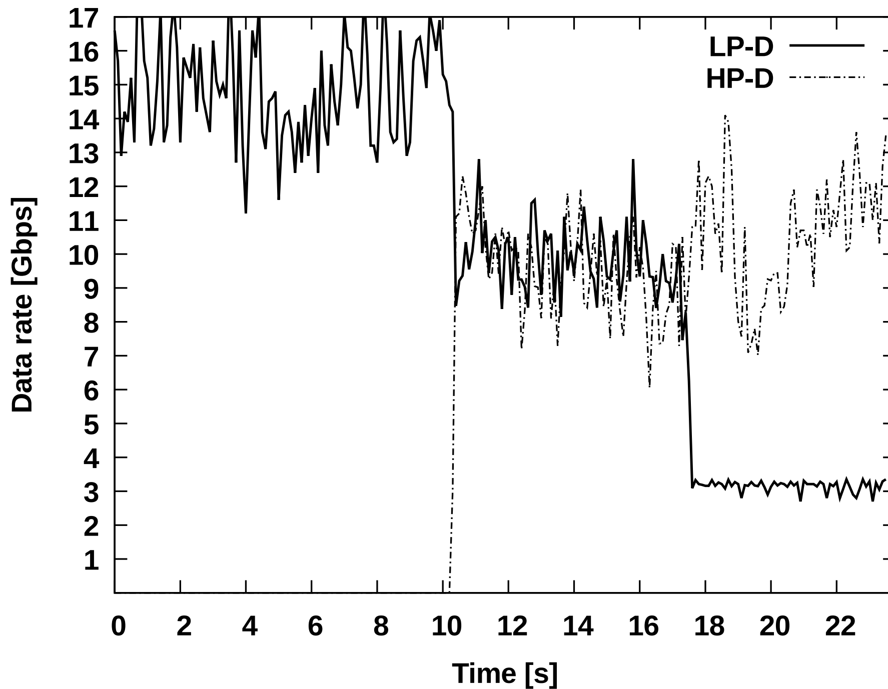


Figure 2.14: Throughput achieved by the two services during the update of the segment list.

the VCA container to its related VNF via SSH, therefore the only factor will be the time needed by the packets traveling between the two entities. In the test bed, the server hosting OSM and the one hosting Openstack were located inside the same data center, and the round trip time between the two was around 0.4 ms. Bearing this in mind, we measured the total time needed by OSM to perform an action, and the average measured time was 5.28 s, which is in line with the behavior shown in Fig. 2.12.

2.2.5 Comparison with Openstack SFC

The solution for deploying SFC natively supported by OSM leverages directly the “SFC-plugin” [B26] of Neutron (the networking component of OpenStack), but with some limitations. These include the fact that OSM only allows to instantiate chains that make use of the Network Service Header (NSH) as encapsulation method, therefore lacking

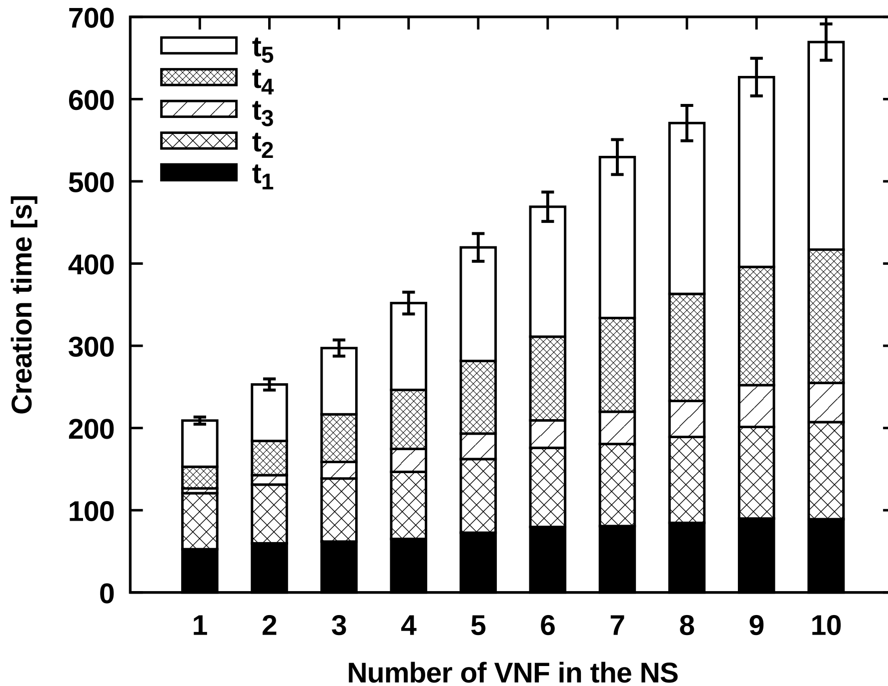


Figure 2.15: Average creation time (with 95% confidence intervals) of a NS as a function of the number of VNF in it. The histogram shows the contribution of each of the five steps required to setup the NS.

the support for MPLS encapsulation, and does not allow the instantiation of SFC-unaware VNFs. Contrarily, the solution proposed here can be easily expanded to support SFC-unaware VNFs, for example by creating an image, at VIM side, containing a modified Linux Kernel that supports SR-proxy [B37] and by adding the actions needed to configure it to the charm. We recall from Section 2.2.4 that the time needed to deploy a service increases by adding the framework for proxy charms. By using the native solution for SFC this overhead would not be present, therefore decreasing the overall deployment time. In other words, we would notice only the time marked as t_4 in Fig. 2.15, plus an additional time contribution needed from OpenStack to insert the required OpenFlow rules in its internal switches. Assessments of this new contribution can be found in [B45]. Nonetheless, the use of proxy charms allows to dynamically reconfigure the chains, a feature the na-

tive solution does not have, requiring the re-instantiation of the whole network service. Moreover, once the proxy charm framework is up and running, it can be also used for other application-specific configurations, simply by implementing the required actions. Therefore, the adoption of proxy charms introduces an overhead in the instantiation of the service, but at the same time it grants the possibility of realizing SFP with SRv6 inside different cloud domains, making this an alternative solution to automate the creation of SFC with OSM as an orchestrator.

To conclude, I developed a method to deploy SFPs using Segment Routing through the functionalities offered by OSM. This method might increase the overall time required to deploy a Network Service (see Fig). However, this overhead might be reduced in future versions of OSM, optimizing the deployment of the Juju components required to perform these actions. Nonetheless, this approach overcomes the limitations suffered by the OSM built-in method for the creation of SFCs. Firstly, allowing dynamic SFP creations/modifications/deletions, while the built-in one allows the creation of SFPs only during the Network Service deployment phase. Secondly, this method eases the integration with service functions not directly managed by OSM or outside the Openstack cluster in which the virtual functions are deployed.

Chapter 3

Network Service Provisioning

Chapter 1 introduces the resource allocation problem for network services composed of many VNFs over distributed environments. Likewise, I mentioned the need of evaluating how these algorithms perform over real testbeds with well-known open-source tools to understand their limitations and drive their development. To these ends, [0] proposes a prototype of an architecture for robust service function chain instantiation with convergence and performance guarantees. To establish the practicality of this solution, the system performance was evaluated via simulations and through a prototype implementation. This chapter will focus on the latter since I dealt mainly with that evaluation step, showing how the deployment time required by OSM scales with different numbers of hosting nodes.

3.0.1 Physical testbed validation with OSM

Necklace is an allocation algorithm for resilient service chain instantiation proposed in [0]. It does so thanks to a fully Distributed Asynchronous Chain Consensus Algorithm (DACCA) that maps the required service chain to the current state of the underlying infrastructure interaction enabled through a southbound interface. This section will report the validation results obtained by applying this algo-

rithm to a real testbed. The testbed is managed by the OpenSource MANO (OSM) orchestrator (Release 7). The orchestrator controls three OpenStack clusters (Stein Release), each including two compute nodes, for a total of six individually-addressable servers on which the service chain can be mapped. Moreover, the physical testbed is composed of full-fledged servers, making their performances comparable to those that would be observed in a production environment.

We defined a format for the exchange of information between the allocation algorithm and the hardware orchestrator, using the JSON formalism:

```
{ "sfc-id": "id_value",
  "vnfs": [
    {
      "type": "type_value",
      "node": "node_value"
    }, ...]
}
```

where: `sfc-id` is a unique identifier associated with the specific chain, in the form of a text string `id_value`; `vnfs` is the ordered list of service functions forming the chain; each service function is described by fields `type` and `node`, with `type_value` and `node_value` being strings identifying a predefined Virtual Network Function Descriptor in the orchestrator, and the physical node the service function has been mapped onto, respectively.

The allocation algorithm sends these messages to a custom-built Python script exposing a REST API. This script takes these messages and constructs the Network Service Descriptors composed of the VNFs specified in the JSON files, with each type mapping to a different VNF Descriptor. Then, it onboards them on OSM and triggers their deployment, selecting for each VNF of a Network Service the compute node chosen by the allocation algorithm. In other words, the Python script acts as a "translation layer" between the southbound interface of

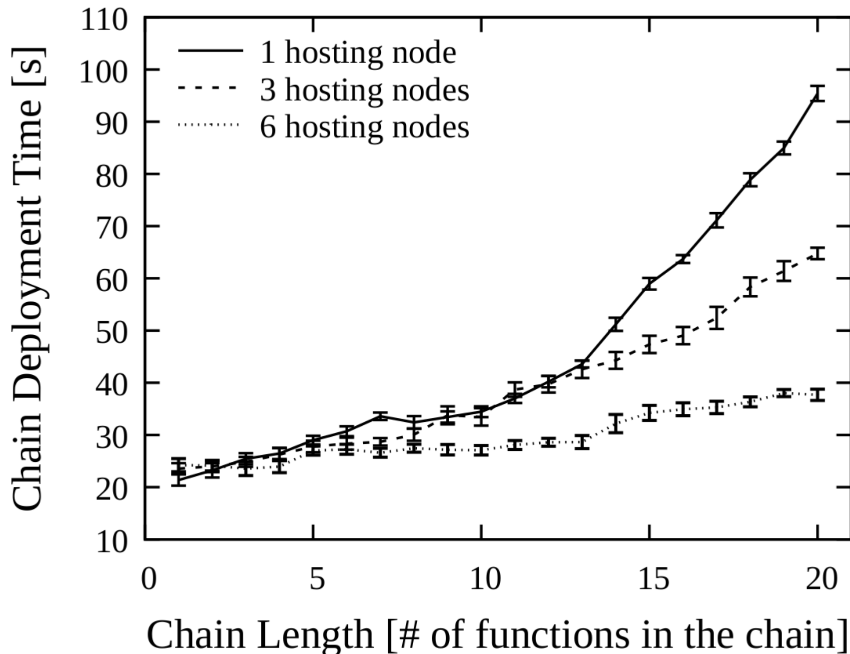


Figure 3.1: OpenSource MANO (OSM) deployment time as a function of chain length and number of hosting nodes.

the allocation algorithm and the orchestrator, mapping the algorithm decisions to OSM directives.

In the physical experimental testbed using OSM, a progressively larger number of chain deployment requests was injected, allowing service functions to be mapped onto one, then three, then all six physical nodes. For each case, I submitted requests for twenty different values of chain length, ranging from 1 to 20. I measured the deployment time of each chain, then removed the chain before running the next experiment, to have independent measurements. I performed 30 measurements per chain length and plotted the average deployment time with confidence intervals. Each measurement campaign took anywhere between 8 and 14 hours. As Figure 3.1 shows, having more hosting nodes results in better chain resource management, as service functions can be allocated on more nodes thus balancing loads, therefore resulting in a shorter chain deployment time. While this is an expected result, it demonstrates that the proof-of-concept implementa-

tion with OSM is functional, hence Necklace can be adopted to deploy service chains on real systems within reasonable time scales (tens of seconds). This means that new services could be deployed on-demand promptly and autonomously, thus proving the effectiveness of these new network paradigms since in past networks the services and their interconnection were man-managed and realized through specialized physical equipment. For clarity's sake, these results differ from the ones presented in Chapter 2 for several reasons. Firstly, these deployment times account for both VNFs instantiation and Service Chain deployment, while the others only account for Service Chain deployment. Secondly, this testbed is composed of three Openstack clusters while the other of a single one, so there is actual load distribution. Unfortunately, during this performance campaign, I identified a bug on the OSM platform that was preventing the SFC creation with Openstack. To solve this problem, I developed a patch fixing it that was then merged into the master distribution of OSM.

Chapter 4

5G Network Slicing

As stated in the introduction, Network Slicing is a pivotal paradigm for 5G networks. This chapter reports a real implementation of network slicing applied to Mission-critical (MC) communications. In this work, the architecture for an MC slice was designed, having in mind its possible distribution among different cloud infrastructures and keeping a separation between the infrastructure and the service providers. This slice was then deployed and tested using open-source tools over physical servers. The result presented here can also be found in [0].

4.1 Network architecture and system components

The implementation scenario considered in this work is in line with the current trends and uses:

- Cloud computing, allowing virtualization of computing resources in data centers equipped with general purpose hardware;
- Network Function Virtualization (NFV), that fosters flexible and cost-effective service orchestration through the deployment of virtualized network functions;
- Software Defined Networking (SDN), that decouples software-based network control and management planes from the hardware-

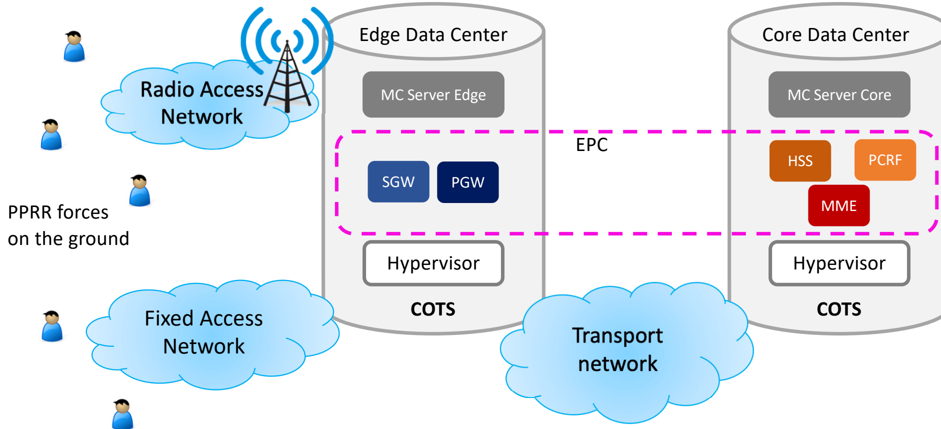


Figure 4.1: General network slicing architecture for MC communications.

based forwarding plane, turning traditional vendor locked-in infrastructures into communication platforms that are fully programmable.

The general network architecture considered here is depicted in Fig. 4.1. To resemble a general networking scenario both mobile and fixed access are considered. The network building blocks are all implemented as VNFs located in two data centers interconnected by a transport network, the Edge Data Center (E-DC) and the Core Data Center (C-DC). The E-DC emulates the access network, with its processing capabilities, close to the fronthaul of the mobile network, therefore more suitable to support functionalities with stringent latency requirements. The C-DC is in the backhaul of the mobile network and will be devoted to more data-intensive applications with less critical latency constraints.

The Mission Critical (MC) communication network is deployed as a NS that includes all the logical components of the mobile network, as well as the server for the MC communication support, compliant with the 3GPP specifications [B46].

4.1.1 The MC server

Leonardo MCX (Mission Critical Services) is part of the Leonardo CSP (Communications Service Platform) product family [B47]. It extends the portfolio of standard solutions for Public Protection and Risk Reduction (PPRR) communications, ranging from Digital Mobile Radio (DMR) to Terrestrial Trunked Radio (TETRA) technologies, with next generation broadband capabilities. It is a complete Mission Critical solution compliant with 3GPP standards on MCX. It includes features from MC Push-to-Talk (PTT), MC Video and MC Data, providing PPRR users with the next generation platform for critical communications over 4G/5G networks. The full solution for MCX is made of the following components:

- an Android Client designed for on-field operations, with a complete set of functionality, that can be installed in off-the-shelf smartphones as well as on ad-hoc terminals;
- a Web based dispatcher, providing control, monitoring and management of the operations of the teams;
- a Management interface for the management and monitoring of the platform KPIs;
- a Session Initiation Protocol (SIP) Core for IP Multimedia Subsystem (IMS)-less scenarios and that can inter-operate with external IMS.

The MCX server can be deployed in a distributed fashion, with a sharing of roles. In particular the media servers, i.e. the SIP servers that will manage and deliver the media streams, can be de-coupled from the registration server used for signalling. This is the feature that was exploited in our experiment, with the goal to keep the media servers as close as possible to the final users and guarantee optimal performance.

4.1.2 The mobile access network

The mobile access network is fully virtualized exploiting well known open-source software components. The focus is on the Evolved Packet Core (EPC), assuming that the Radio Access Network (RAN) will be deployed already on the ground either with dedicated resources or by sharing the resources of the public mobile network.

In our experiment the RAN was simulated. Both user equipment (UE) and eNodeB were simulated with the L2 network Functional Application Platform Interface (nFAPI) Simulator provisioned by OpenAirInterface [B48]. This simulator does not require any specific hardware and simulates L2 and above stack layers, short-cutting the physical layer. Furthermore, it gives the possibility to simulate multiple UEs with a single instance. The EPC was implemented with the NextEPC platform [B49]. NextEPC implements a fully functional LTE EPC in a similar way to other platforms, such as for instance OpenAirInterface. We opted for NextEPC because of its flexible modular architecture that has been designed already to be deployed in a virtualized environment. The NextEPC software suite is composed of 5 modules (*nextepc-mmed*, *nextepc-sgwd*, *nextepc-pgwd*, *nextepc-hssd* and *nextepc-pcrfd*) that can be individually installed as packages in several Linux distributions and can be managed as daemons with the respective native system and service managers. Each module provides one or more dedicated configuration files that must be modified according to the actual set-up of the data plane and control plane interfaces compliant with the 3GPP standards. In addition, this software suite gives the possibility to install a Web User Interface that allows to add in the Home Subscriber Server (HSS) database the information related to users and service subscriptions, and ease their further management.

Although NextEPC does not yet provide the Control and User Plane Separation (CUPS), its modular architecture allows a deployment of the various components in different data centers. We exploited this feature to implement ad-hoc a partial CUPS, as will be described in the remainder of this manuscript. CUPS is a design choice made for 5G network architecture aiming at separating the control plane com-

ponents from the user plane ones, thus reducing the network overhead due to control traffic on the data plane and dividing the management effort.

4.1.3 Data center management infrastructure deployment

In our experiment, OpenStack was used as a cloud management platform [B18]. OpenStack represents the VIM used for the resource management in each data center. The OpenStack installation was carried out via Kolla-Ansible, which allows to quickly get a production-ready container-based OpenStack environment. Each component (i.e., Nova, Neutron, Cinder, etc.) is deployed inside a separate Docker container, thus granting a separate working environment for each one of them. The installation strictly follows the official guide and only the most common components were used (only the traditional software update and upgrade was carried out on bare metal machines prior to the installation process).

4.2 A Network Slice for MC communications

4.2.1 Actors and Roles

Network slicing is a process that involves three main actors:

- Infrastructure Provider (IP): the owner of the infrastructure providing all the infrastructural management actions, in the specific example a network provider acting at a local or national scale operating a private network to support the MCX services;
- Network Slice Provider (NSP): the provider of the communication service implemented with the network slice, in this specific case the governmental agencies that provide the MCX support and/or third parties under contract to provide this kind of service;

- Network Slice Customer (NSC): the user of the communication service, in this specific case the PPRR forces that will use the MC network during operations (police, firefighters, hospital ER, etc.).

These actors must have rights according to their respective roles, with IP and NSP having specific management roles to keep the infrastructure up and running. Therefore the slice architecture must be defined in such a way that allows a seamless co-existence of these actors and provides all of them with the required functionalities.

An important characteristic of the NS under investigation is that it is not bound to a single data center, but is basically split into 4 logical sections:

1. Mobile and fixed access network;
2. Edge Data Center (E-DC) virtualizing the access part of the EPC and the edge MCX server;
3. Core Data Center (C-DC) virtualizing the core part of the EPC and the core MCX server;
4. Interconnection network between the DCs, that could be either a public network or a private geographical interconnection.

Moreover, the NS must be designed to satisfy the following main characteristics:

- interconnection with the outside of the DC using two logical networks, the former dedicated to inter-DC connectivity, the latter used to connect to the outer world;
- capability to establish tunnels and/or specific routing policies on the external networks;
- VNFs must be manageable objects as required by the NFV-MANO architecture;

Table 4.1: Example of some NEST parameters for the MC communications network slice

ATTRIBUTE	VALUE
Coverage	Local (Outdoor)
Guaranteed Downlink Throughput per Network Slice	391600 (391.6Mbps, band 3, channel 20MHz(100RB), 256QAM, 4x4MIMO)
Mission Critical Support	1. mission critical
+ Mission-Critical Capability Support	1: Inter-user prioritization, 2: Pre-emption, 3: Local control
+ Mission-Critical Service Support	1: MCPTT, 2: MCData, 3: MCVideo

- VNFs must be protected, meaning that their interfaces must not be directly exposed on the interconnection network to the outside of the data center;
- separate management must be guaranteed for the IP and for the NSP;
- specific management console for the NSP must be reachable from outside the data centers, to keep NS management fully transparent to the IP.

In the following we will explain the principles and the instruments that we used for the slice design and deployment.

4.2.2 Network Slice Architecture and Characteristics

According to [B50] the NS specifications are described with the NEtwork Slice Type (NEST), a set of parameters with associated values that are defined using a generalized dictionary (Generic Slice Template or GST) but referring to a specific service or set of services. A possible NEST for the network slice here considered is presented in Table 4.1.

The GST acts as a template for the NEST and the NEST provides QoS and/or functional specifications for the NS. None of them says how the NS should be implemented. The specific implementation of the NS is usually called the *Blueprint*, i.e. the collection of all the technical details that are necessary to implement that particular NS. As we will see in the following, the NS implemented in this work is rather complex and its deployment was split into several steps, to make configuration and debugging easier and more controllable.[1]

Every section of the slice is specified by means of several NFV-MANO descriptors, including one that describes how to put together the various components. Some of these descriptors are common to the various slice sections and can be re-used. The full set of these descriptions and related configuration files represents the NS Blueprint.

Figures 4.2 and 4.3 show the deployment architectures for the two sections of the slice to be hosted in E-DC and C-DC. The section in the E-DC (Fig. 4.2) will host the two gateways of the EPC, namely the Serving Gateway (SGW) and the Packet Data Network Gateway (PGW). Together with them, we added two other components needed to guarantee full slice functionalities, mainly addressing slice security and management plane connectivity. These additional components are:

- Network Slice Provider management console, connected to the various slice components for management purposes;

- Network gateway, providing the network functionalities required for correct traffic routing between the slice components and the external networks, thus also providing the required traffic isolation for security purposes.

In this scheme the SGW and the PGW will carry data traffic to the Internet, basically being devoted to the user data plane. Control

[1] It is also worth underlining that the slice architectures presented here are a graphical sketch. The actual implementation in OpenStack is even more complex since many VNFs are made of two VMs, one for production and one for management, with an additional network in between to connect them.

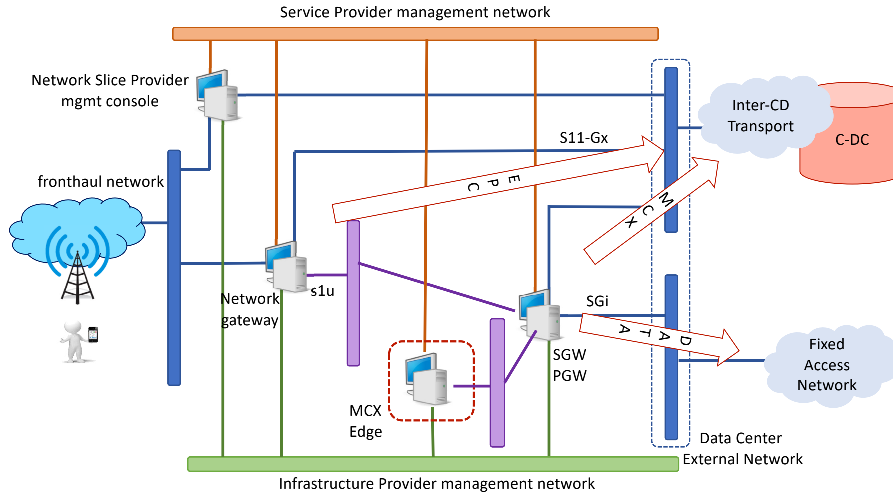


Figure 4.2: Architecture of the access section of the slice (E-DC), with SGW and PGW still shared according to the LTE architecture.

plane traffic will be routed directly to the control plane components in the C-DC by the network gateway. In this way we achieve a basic CUPS, that can be extended gradually to a fully fledged 5G compliant architecture. In this schematic we assume the eNodeB is also hosted directly in the E-DC and connected to the gateway via an external network of the DC. This is not mandatory in general: in case one or more eNodeBs are implemented with dedicated hardware outside the data center, the interconnection will be exactly the same and therefore the slice blueprint would not need any variation. Together with the components of the mobile network the E-DC will also host the edge MCX server, that will be responsible mostly for the data traffic among end users.

The section in C-DC (Fig. 4.3) will host the control plane components, i.e. Mobility Management Entity (MME), HSS and Policy and Charging Rules Function (PCRF) as well as the MCX core server. The slice section also includes a NSP management console and network gateway. The control plane traffic will be routed to the slice components by the network gateway via an internal network, according to the proper addressing configured at the eNodeB. Similarly, the interconnection between the MME and the SGW will be guaranteed.

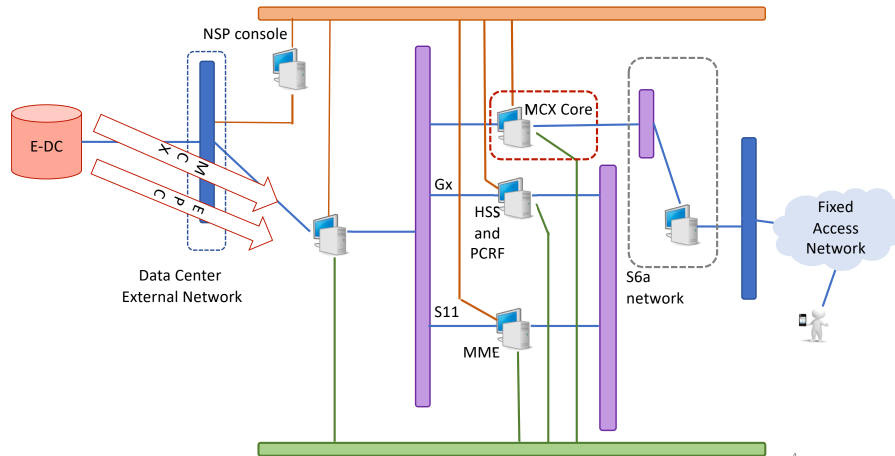


Figure 4.3: Architecture of the core section (C-DC) of the slice with control plane components.

A general comment is related to the external network interconnecting the two DCs. In our architecture it is split in two logical sections, the former devoted to DC interconnection, the latter devoted to WAN connectivity. However, this splitting has the aim to show that these could be two different infrastructures, as well as just a single infrastructure with two logical roles, maybe mapped on different IP networks.

4.2.3 Network Slice Delivery and Lifecycle management

In [B51] the various steps implementing a full NS lifecycle management are defined and described as in Fig. 2.9. All these steps have been implemented in the test-bed described in this work. The preparation phase includes the NS description and the environment preparation.

The NS description consists in creating the OSM descriptors that, according to ETSI MANO approach, provide all information regarding:

1. the VNF packages to be run in the slice;
2. the interconnections between them (Virtual Links in NFV-MANO

terminology), described in the Network Service Descriptors (NSD) and Virtual Link Descriptors (VLD);

3. the Network Slice Template (NST) as a combination of Network Service Descriptors (NSDs);

4. the details of the VIMs where the NS has to be instantiated;

5. the VNF Forwarding Graph Descriptor (VNFFGD), specifying the traffic path from one VNF to another, which has to be implemented in the NS.

The preparation phase includes the setup of that part of the infrastructure which is not NS specific. In this particular case it refers to the networks in the cloud platform that must be shared between slices and must exist before the NS is started. Three such networks were set up by the administrator of OpenStack (acting as IP):

- the management network of the IP, that will be connected to the parts of the NS that the IP has to control in case of some emergency event, collaborating with or overriding the management actions from the NSP;
- the inter-DC interconnection network;
- the external networks that will be used to connect to the access networks, either mobile or fixed.

4.3 Experimental Results

All the experiments were run in a private data center, with two separate OpenStack clusters for the E-DC and C-DC, respectively. Each one of them is composed by two physical servers, equipped as follows: 64 GB of RAM; 40 CPUs; 1.2 TB of disk; 1 Gbit/sec interfaces; Ubuntu 18 LTS as OS.

The overall scenario considered is shown in Fig. 4.4. From the SIP Uniform Resource Identifier (URI) point of view the domain is simply called `test` and two UEs are registered as `user1@test` and

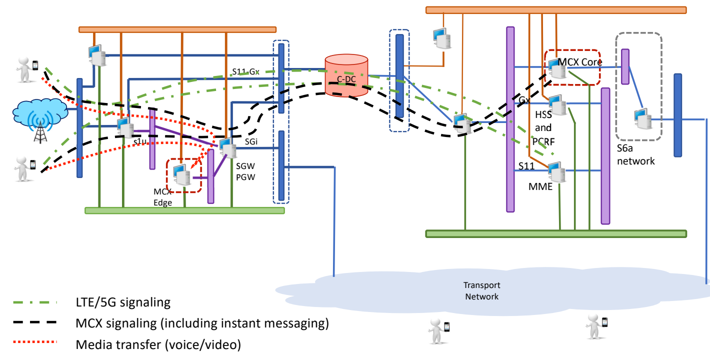


Figure 4.4: Full slice blueprint with signalling and data traffic flows, for the test where both user1 and user2 are connected via the emulated eNodeB.

`user2@test`. The paths of the signalling traffic flows are also shown in Fig. 4.4. Although the gateways (such as SGW and PGW) are not split and will carry both control and user plane traffic, according to the LTE architecture implemented in NextEPC, the Figure 4.4 shows that the slice is ready for CUPS and enables splitting the various control plane components between the E-DC and the C-DC, leaving closer to the user the components that may help providing better performance.

Coming to the experiments, at first we tested the correct functional splitting of roles of the two MCX servers according to the planned split of workload. In the considered scenario the core MCX server is dedicated to handle signalling traffic, such as SIP registration and call set-up messages, while the edge MCX server acts as media server only. Figure 4.5 shows the flow of an MCVIDEO call from the point of view of the caller (`user1@test 10.250.123.101`) to the callee (`user2@test 10.250.123.102`). The call flow is produced with Wireshark out of the traffic traces. The core MCX is located at `10.250.2.249` while the edge MCX is located at `10.250.2.35`. The MCX servers are configured in order to force the communication to go through the media server coupled with the signalling server. Figure 4.4 shows that the split of roles is correctly realized in the slice. Indeed, the call forwarded by the MCX servers to the callee shows a clear separation of the signalling from data. The SIP traffic re-

quired to set-up and close the multimedia call between the two users is routed to the core MCX server. In fact, we see that SIP messages such as INVITE, TRYING, RINGING flow between the core MCX server (`10.250.2.249`) and the callee (`10.250.123.102`). Instead, the Real-time Transport Protocol (RTP) media traffic is exchanged between the edge MCX server and the users. In particular, with reference to the reported traffic trace, the RTP packets are forwarded from the edge MCX (`10.250.2.35`) to the callee.

Then to prove the effectiveness of the CUPS approach we exploited the performance measure feature of the MC mobile app. This is an Android app that can be installed in a commercial smartphone or in an Android emulator and provides all the MC service implementations as per the 3GPP standard, in particular MCDATA, MCVOICE and MCVIDEO as required by the NST. The performance feature of the app provides a series of evaluation tools for measuring network latency and capacity as shown in Fig. 4.6. To emulate a greater latency when connecting to the core infrastructure we forced a delay of $T = 200$ ms on the inter-DC connection. This delay was forced with Linux traffic control on the outgoing interface of the PGW. We asked the app to register on both the MCX core and on the MCX edge.

Obviously the MCX core is the only one which allows the registration of a SIP user since it is the only one running the management functions. When we ask the MC app to register on the MCX edge which is acting as media server only, the registration is not successful but still the app allows the execution of the performance test, even though in a limited way. As a consequence the two screenshots are different. For the scopes of this research the field of relevance that can be compared are: `2. CONNECT TCP` and `3. HTTP PING`.

These values depend on the round trip time (RTT) of the data connection. We can see that in both cases they are approximately 200ms larger in the connection to the MCX core than to the MCX edge. This is perfectly in line with the additional latency introduced in the path towards the C-DC, that is in this experiment 200ms.

Therefore we can conclude that, in case of a real call, the RTT of

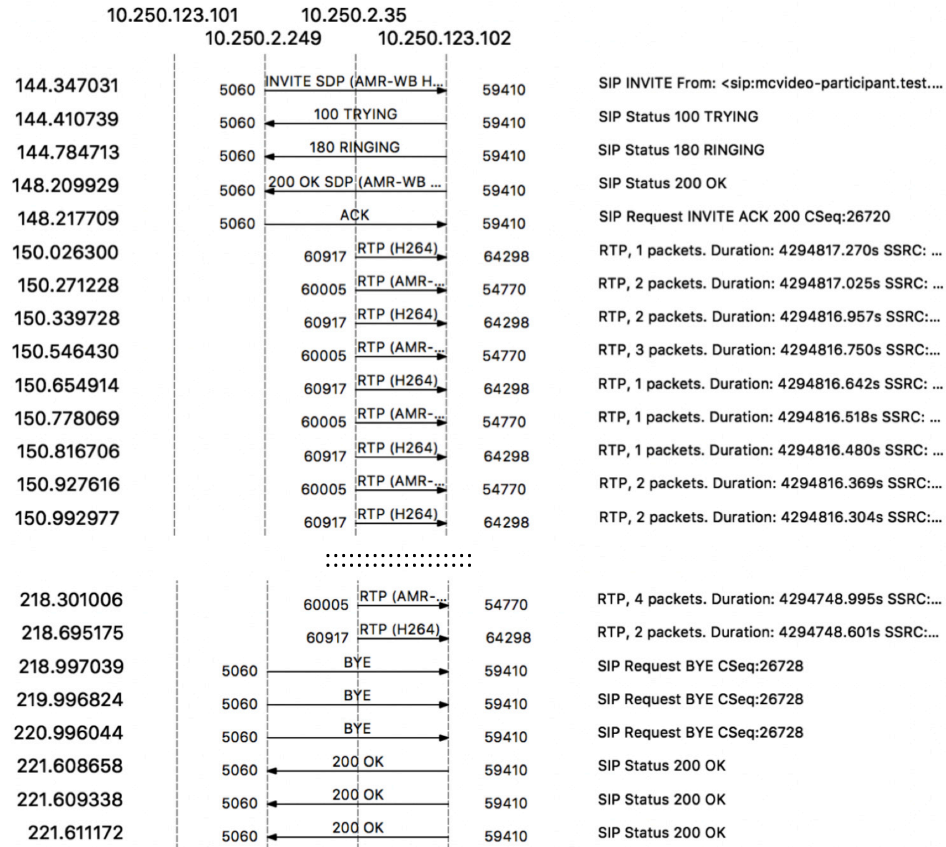


Figure 4.5: SIP flows of an MCVIDEO call obtained by capturing the traffic on the callee (`user2@10.250.123.102`).

the media flows (voice and video) would be significantly lower when compared to the RTT of the signalling towards the MCX in the core. This is one of the advantages expected by the CUPS approach.

To conclude, in this chapter, I have reported a proof-of-concept implementation of NFV orchestration and network slicing applied to mission-critical applications.

It included the definition of a network slice blueprint including both 5G and Mission Critical core network functions. Furthermore, it reported a demonstration of the complete automation of its lifecycle management, compliant with the NFV-MANO specifications, exploiting the ETSI Open Source MANO platform.

This chapter showed that a complete separation between the con-

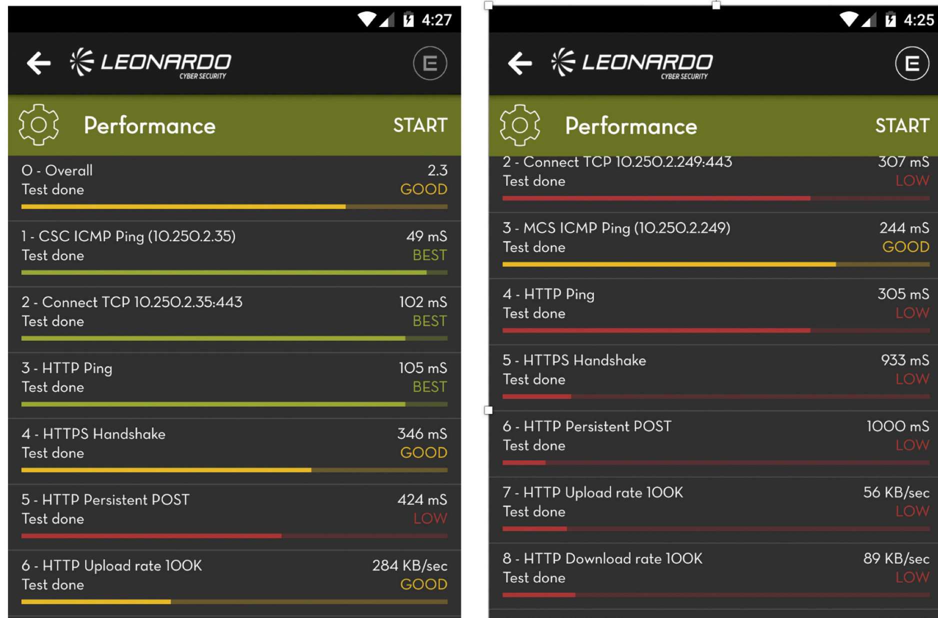


Figure 4.6: Screenshot of the MC application executing performance measurements towards the MCX in the edge and in the core.

trol and data plane could be achieved at both the network (5G) and the service (MC) levels. Some numerical examples are provided that demonstrate the effectiveness of this approach in terms of performance.

Overall, the results presented prove the effectiveness of the 3GPP architectural approach to MC communications and can be a valuable guideline for future MC network implementations at the European level.

Chapter 5

Enabling Industrial IoT as a Service with Multi-access Edge Computing

Another relevant scenario that will benefit from the technologies presented in the previous chapters is Industrial IoT (IIoT). The sheer amount of data produced from these systems and their distributed nature demand a networking infrastructure carefully thought for them. This chapter will present an architecture for IIoT systems merging the Multi-Access Edge Computing (MEC) and Fog paradigms. Furthermore, the chapter also contains a proof-of-concept implementation of this architecture developed with open-source tools. Lastly, an extension to the standardized MEC APIs is presented, helping the discovery of IoT data sources available in a MEC system. To the best of my knowledge, there are not a lot of works merging the fog and edge paradigms proposing both a high-level architecture and its practical implementation. Furthermore, the same applies to systems using the APIs standardized by ETSI MEC. The work presented here has been published in [0] and [0].

5.1 Introduction and related works

The Internet of Things (IoT) concept has evolved in the last decade, and has expanded in many fields of today's society. An increasing number of smart environments are being created, unlocking unprecedented potentialities for innovative applications and developments [B52]. One of the context attracting increasing interest for extensive IoT applications and adoption of advanced communication network technology is represented by smart industry and manufacturing, often referred to as Industry 4.0 [B53]. Smart connectivity capabilities of a large number of sensors and devices, the availability of cloud computing platforms, and the implementation of software-defined network control and management techniques bring the opportunity to support rapid material handling, efficient information sharing, fault detection time reduction, and flexible production processes in manufacturing environments.

Recently, 5G technologies have been making their way inside factories as well, enabling the capability to manage a high density of devices and different classes of service, including massive machine-type communications, enhanced mobile broadband, and ultra-reliable low latency communications [B54]. That is expected to make the integration of components easier, by improving the communication between heterogeneous devices. That is possible thanks to the flexibility offered by 5G networks. As was explained in the previous chapters, 5G network slicing allows the co-existence of multiple logical networks, with different requirements, on top of the same physical one. Therefore, diverse industrial services can be successfully supported by the 5G networks, even with very diverse service requirements. Also, Cyber-physical Systems (CPS), such as machine digital twins, which combine statistics, computer modeling, and real-time data measured on physical systems, can help in modeling the response of a system under multiple working scenarios. Industrial systems can benefit from the combination of multiple technologies and agents, in order to take real-time decisions and reach the common goal of improving the efficiency and responsiveness of production systems [B55].

As an Internet-based commodity to share computing, storage, and network resources, cloud computing can be part of the answer to the increasing need of manufacturers for data processing. New cloud-based manufacturing models can be defined, where all resources and capabilities are virtualized and offered "as a service" allocated on-demand through the cloud, leading to the concept of smart manufacturing [B56]. However, exchanging data between machines/sensors and remote cloud locations may result in delayed responses, high usage of bandwidth, and energy consumption. Besides, the long-distance communication exposes the system to the risk of external network faults and security breaches, which represent highly critical challenges.

To overcome those problems, fog and edge computing solutions have been proposed to bring compute, storage, and network capabilities closer to or even within the user premises. As a consequence, data collected from smart machines and sensors can be processed locally or at the edge, without reaching the cloud, to fulfill stringent requirements on latency, real-time responsiveness, limited network traffic, and protection of sensitive data. In particular, considering the nature of devices and equipment used in a factory, peripheral processing in support of highly reactive systems could be more effective than a typical cloud-based approach [B57].

All the discussed aspects should be considered in order to reach the original objective of Industry 4.0 of achieving much higher gains in operational efficiency with respect to the marginal improvements expected from traditional cost-cutting measures. Therefore, it is clear that the transition to Industry 4.0 will depend on the successful adoption of many new information and communications technologies, which, in addition to enhanced IoT connectivity and processing, will provide highly flexible control and management capabilities, as well as high reliability and security. All of this will be offered to the customers in a fully automated and collaborative environment through suitable programmable platforms, thus fostering the introduction of an *Industrial IoT as a Service* (IIoTaaS) model.

Despite the availability of technologies at an adequate level of ma-

turity, there still exist the need for a suitable framework in which the different communication and software technologies can operate efficiently to achieve the objectives of smart manufacturing and IIoTaaS. The main contribution of this chapter is the definition of an architecture for IIoTaaS applications which takes advantage of a multi-level computing platform, consisting of edge, fog and cloud environments. In particular, the proposed approach aims at unifying the orchestration of heterogeneous fog and edge computing resources under a single framework, which is designed to be compliant with existing standards for Multi-access Edge Computing (MEC) [B19], rather than defining a new set of interfaces. This brings all the advantages of MEC-based service management to the development and deployment of IIoTaaS applications. In this chapter, a reference operational architecture, the different components of the framework, and a proof-of-concept implementation are reported, showing how the MEC-based approach and the supporting information and communication technologies enable the automated deployment of IIoTaaS applications in a matter of seconds. Smart industry environment is considered here, representing one of the most challenging and demanding application of the proposed framework. In any case, the proposed methodology has a high potential to be reused in other fields. Indeed, the high-level architecture designed is general, thus supporting applications even outside the IIoT scope.

5.2 Reference Scenario for IIoT as a Service

In a smart manufacturing environment, production line appliances are equipped with sensor nodes that generate and exchange monitoring data over a (wireless) network, according to IoT principles [B54]. Such data then needs to be processed, to evaluate production performance and recognize faults in the procedure, as well as for other purposes. To this aim, several diverse compute and network resources are available, ranging from nearer and less powerful ones located at the edge or in

one or multiple fog clusters, to farther but more powerful ones located in a remote cloud, as depicted in Fig. 5.1.

According to its original definition, the fog domain includes compute resources offered by infrastructures located anywhere from the cloud to the edge. For the purpose of this work, however, only fog resources located in the edge domain are considered. Therefore, in the following, the term *fog resources* identifies computing resources located in the local network of the factory, including mobile devices being carried around the premises by personnel as well as devices located on production line machinery. On the other hand, the term *edge resources* identifies local datacenters, closer to the access network or within factory premises, and possibly including high-performance servers normally employed to perform specific low-latency tasks.

Considering that IIoTaaS applications can include software components that must take advantage of the proximity to the source of data, both edge and fog resources should be used to deploy them, based on their availability. Orchestrating edge and fog resources in a unified way according to the ETSI MEC framework [B19] allows to benefit from its additional features. The MEC approach can be used to facilitate the interoperability between services hosted on different domains and implemented with different technologies, towards a seamless integration of heterogeneous service components. It also allows for a direct interaction with the 5G access network, including direct knowledge of end-system position and connection quality, for which standard APIs already exist (e.g., MEC 013 for Location API, or MEC 012 for Radio Network Information API), thus enabling the support of new types of services. This reference scenario still supports a more classic cloud-based approach, in which the computational resources are located in remote datacenters, public or private, offering higher capacity than the one provided by local resources.

Recent work includes valuable examples of the adoption of fog and edge paradigms in 5G and Industrial IoT contexts. A number of such scenarios are presented in [B58], but no specific management architecture dealing with the diversified infrastructure is proposed. On this

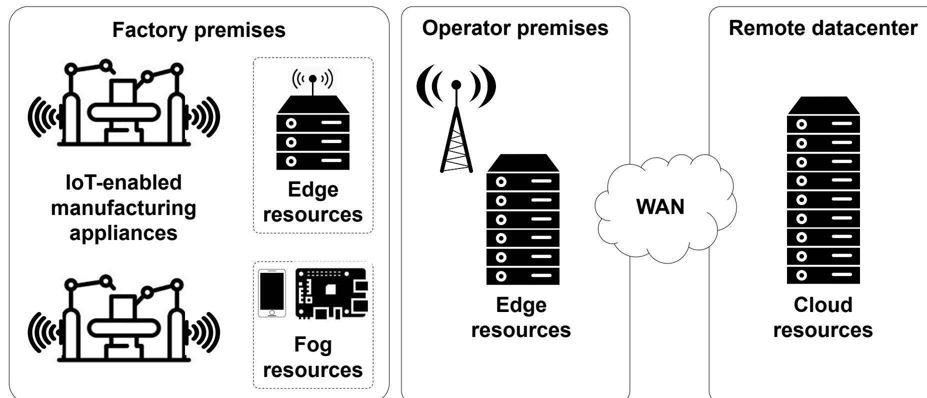


Figure 5.1: Reference scenario, where cloud, edge and fog computing resources are available; in specific cases, only fog or edge resources may be available in addition to cloud ones.

matter, some architectures integrate NFV and Fog [B59], but do not adopt standardized functionalities that can be provided at the edge. Other research efforts target a scenario that is very similar to the one presented here [B60] [B61], but consider a fog infrastructure including fixed hardware only. In contrast, this chapter proposes a management architecture based on ETSI MEC and ETSI NFV standards to supervise remote cloud, edge and fog resources, with the benefits that come with the integration of a MEC system, including service discovery, location services, and more. Also, in this work fog clusters comprise devices that are not necessarily known a-priori, considering the possibility of allocating services on them in a dynamic and automated fashion. Finally, the fog domain service orchestrator employed here is able to allocate the service according to multiple allocation models, and to choose the most efficient allocation technique available at the time of service request [B62].

In line with one of the use cases presented in [B58], a possible example of an IIoT service spanning across multiple domains could be based on data exchanged according to a publish-subscribe paradigm. A set of remote cloud resources could be allocated to store and analyze long-term data received from the factory premises. A set of edge resources may be allocated to act as brokers between the sources of

IoT data and their subscribers, being them running in the local or remote domain. Furthermore, local processing of the aforementioned data may be needed to comply with the requirements of low latency services. Finally, exploiting the proximity and mobility of fog devices, their resources may be used on demand to collect data from specific areas and/or appliances of the factory, to perform light processing on them, and to redirect pre-processed information toward the message broker and, in turn, to interested subscribers when needed. All the data exchanges just described could use typical IoT messaging protocols, whose components must then be dynamically deployed according to the specific service needs.

5.3 Features and Components in a MEC-enabled IIoTaaS Framework

A framework and reference architecture for MEC is introduced in [B19], along with the description of relevant functional elements. Based on this reference architecture and the scenario described in Section 5.2, the proposed MEC-compliant architecture for IIoTaaS application deployment and resource allocation across edge and fog computing environments is shown in Fig. 5.2. The supervising entity is the Operation/Business Support System (OSS/BSS), representing users or third-party services that manage service deployment, enforce company policies, or react to computational needs.

Service deployment in the fog computing subsystem is managed by FORCH [B62] (Fog ORCHestrator, or FO for brevity), a modular orchestrator for flexible deployment of computing services over dynamic fog computing infrastructures. The components of FORCH can be mapped onto functional blocks described in the reference MEC standard. With reference to the left side of Fig. 5.2, this mapping is described as follows:

- the functionalities of the Multi-access Edge Orchestrator are realized by the FO mediator module, which receives service re-

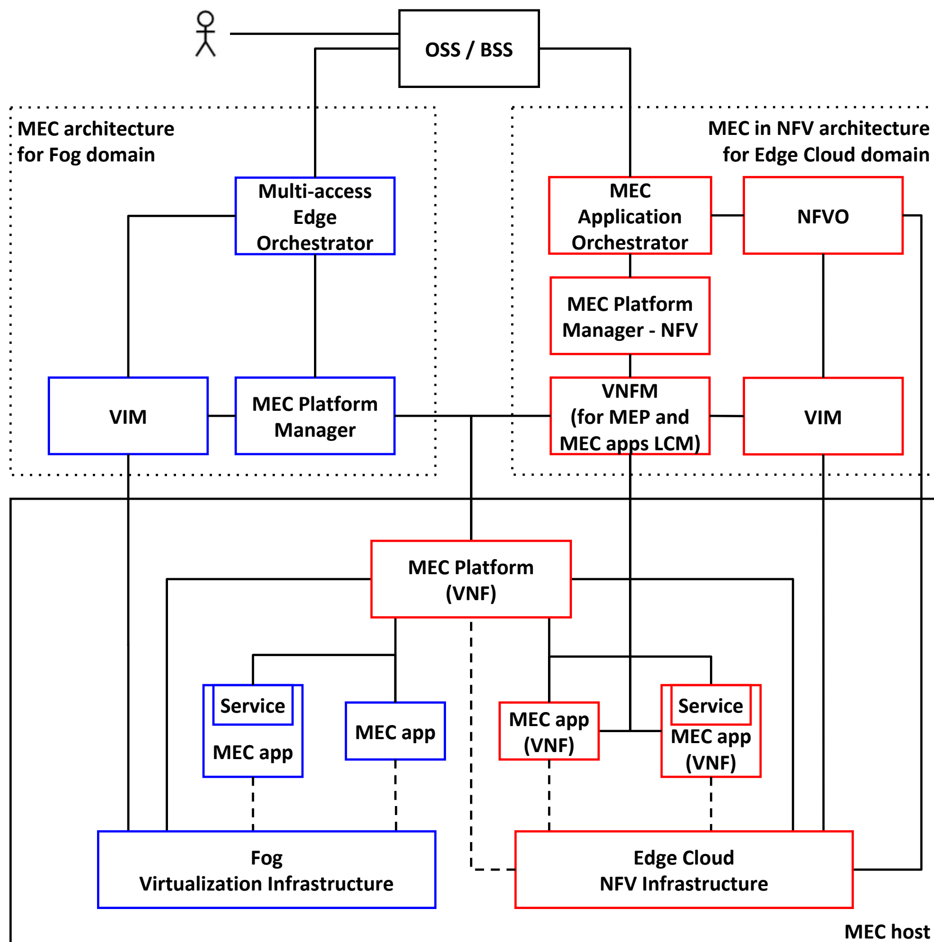


Figure 5.2: MEC-compliant proposed architecture, incorporating elements from the edge (highlighted in red) and fog (highlighted in blue) computing environments.

quests and selects the appropriate resources to be allocated to the service;

– tasks pertinent to the MEC Platform Manager are executed by the FO aggregator module, which gathers and aggregates information on service deployment and resource usage;

– the Virtualization Infrastructure Manager (VIM) defined in the MEC standard is mapped onto the FO VIM module, which manages the activation of services on fog resources, handling technology-specific details required by allocation and deployment procedures, and collecting monitoring information;

– the Fog Virtualization Infrastructure (VI) is represented by the VI offered by each of the fog nodes, which hosts activated services running on local resources, and reports monitoring information to the FO VIM module.

The latter component is meant to be configured on each of the nodes offering resources to the fog system. The resource utilization of every node is monitored, and this information is passed to the fog orchestrator, along with the set of services that the node can host, allowing the orchestrator to make informed decisions on the allocation of services on nodes. Allocation policies are configurable to meet specific needs, via a multi-tenant system, based on an approach similar to that of major cloud orchestrators.

Considering the right side of Fig. 5.2, the architecture proposed for the edge domain is essentially based on the "MEC in NFV" architecture defined by ETSI in [B19], which aims at re-using components from the Network Function Virtualization (NFV) framework to fulfill a part of the MEC management and orchestration tasks, thus allowing to instantiate both MEC applications and Virtualized Network Functions (VNFs) within a unified framework. Specifically, the functionalities offered by the Multi-access Edge Orchestrator (MEO) are split into two different functional blocks, the NFV Orchestrator (NFVO) and the MEC Application Orchestrator (MEAO). The former is in

charge of managing MEC applications as if they were typical VNF instances. The latter is in charge of the remaining MEO functions, such as enabling the instantiation and termination of MEC applications and maintaining a view of the status of the MEC system (e.g., deployed MEC hosts, available resources). The MEC Platform Manager becomes the MEC Platform Manager - NFV (MEPM-V), which is in charge of the same tasks as the MEC Platform Manager but without any Life-Cycle Management (LCM) action on the MEC Platform. These actions are instead delegated to a Virtualized Network Function Manager (VNFM), being the MEC Platform itself deployed as a VNF.

The edge and fog computing environments are merged at the MEC host level. A MEC host functional block can be implemented as a single physical or virtual machine offering computing resources, or as a cluster of such machines, operating behind a single abstraction. Therefore, the set of fog nodes and edge resources can be grouped into a single MEC host, operating with a single MEC Platform (MEP). The choice of considering both the edge and fog domains as a single MEC host is motivated by the particular characteristics of fog nodes. Having assumed the set of fog resources to be mutable over time, instantiating and maintaining an active MEC Platform in this kind of scenario could be challenging. For this reason, the proposed architecture relies on a single MEC Platform for both domains, hosted in the edge, considering it to be a more stable infrastructure over time. This solution also enables the adoption of a single abstraction for heterogeneous computing environments (fog and edge) and simplifies the interoperability between the two domains since it does not require to handle the communication between different MEC Platforms.

5.4 Proof-of-Concept Implementation

In order to demonstrate the feasibility of the proposed MEC-based architecture for IIoTaaS applications and how it enables interoperability between the edge and fog domains, a proof-of-concept (PoC)

implementation[1] has been developed and tested on commercial off-the-shelf servers under a use case relevant to Industry 4.0 scenarios. The example briefly discussed in Section 5.2 is considered, where the widely adopted Message Queuing Telemetry Transport (MQTT) protocol has been chosen as a publish-subscribe solution: an MQTT broker is deployed on edge resources, multiple sensing applications are instantiated on available fog nodes and edge nodes, acting as MQTT publishers sending data to the MQTT broker, and a sink application running in the cloud acts as an MQTT subscriber receiving data from the MQTT broker. All the required software components running in the edge and fog environments are instantiated on demand using the automated procedures offered by the proposed framework architecture and detailed below. By taking advantage of the MEC-based approach, those software components are deployed and made capable of interacting with each other independently of the specific computing platform being used, resulting in an Industrial IoT application truly offered "as a service."

The interoperability among the different computing domains introduced in Section 5.2 is achieved by making use of a testbed composed of several platforms:

- OpenStack is employed in the core cloud domain to instantiate Virtual Machines (VMs). For the PoC no specific configuration was needed.
- Kubernetes orchestrates the deployment of containers that execute MEC applications in the edge domain. For this PoC, the Kubernetes cluster was configured with Docker as a container engine, CoreDNS as DNS service, Calico as container networking solution, Metallb as load balancer, and OpenEBS as persistent volume manager.
- Open Source MANO (OSM) handles the deployment of NFV Management and Orchestration services in the edge. Specifi-

[1] `https://github.com/DavideBorsatti/IIoTaaS`

cally, it is used to deploy MEC applications in the Kubernetes cluster.

- FORCH takes care of the deployment of containers that execute MEC applications in the fog cluster.

As mentioned in Section 5.3, the fog orchestrator FORCH is a Python-based original solution for fog computing service deployment recently developed at the University of Bologna [B62], composed of several cooperating modules. In this PoC, a subset of the APIs offered by the fog orchestrator is utilized to deploy MEC services in the fog domain.

The MEC applications employed to test this solution are based on the "Unibo MEC API Tester," also developed at the University of Bologna as part of the ETSI NFV&MEC Plugtests 2020 [B63]. It already implements most of the MEC 011 APIs [B20], which were integrated with a simple MQTT client service that can be exposed and consumed through the aforementioned APIs. Due to the limited scope of this PoC and constraints in the experimental platform, this setup only employs the MEC 011 APIs for service registration/discovery. However the same architecture is also capable of supporting different MEC-enabled services.

To the best of our knowledge, a working open-source software tool that implements all the functionalities of a MEC Platform is not available yet. Therefore, a Python-based custom solution has been developed to implement the set of MEC Platform features required by the PoC. This custom solution follows the directive defined by ETSI in terms of API specification.

The MEC 011 APIs are adopted to aid the interaction between different MEC applications and their services, even if they are deployed on different infrastructures. More specifically, each MEC application can register its own services to the MEC Platform through a REST API POST request to the `/applications/{appInstanceId}/services` endpoint. This request includes all the details of the specific service being registered, such as hostname or IP address, transport layer protocol, and port, and/or other endpoint information. The MEC Plat-

form receives this request and loads it into its internal database. The list of registered services can be retrieved by any other MEC application via the `/services` REST endpoint of the MEC Platform. This way, MEC applications can discover the list of available services and how to consume them. An example of the format standardized by ETSI used to describe MEC services is included below.

As specified in the ETSI MEC standard, the MEC Platform should also provide DNS resolution to all MEC applications in its domain. In the presented scenario, this would ease the communication between services deployed on the edge and the fog computing domains. It would also partly justify the choice of considering both edge and fog nodes as parts of a single MEC host abstraction. The internal DNS service of Kubernetes was configured to be exposed outside of the Kubernetes cluster, to be used also by MEC applications deployed in the fog domain. Then, two different DNS zones were defined, one for services running in the edge domain and another for those running in the fog domain. For the former, CoreDNS was configured to resolve all incoming requests related to the `mec.host` zone to external IP addresses used by the Kubernetes cluster, thus reachable by any other MEC application running in either edge or fog nodes. As for the DNS zone related to services running in the fog, the fog orchestrator adds its DNS associations to a DNS zone file that is shared with the Kubernetes DNS service, which refers to it for all the requests directed to the `fog.host` zone.

Having deployed all the necessary components, the practicability of the proposed solution is verified through an experiment that follows these steps:

1. Deployment of the MEC Platform as a Kubernetes application in the edge.

2. Instantiation of the data sink as a VM in the core cloud, which may happen before, during, or after the execution of the previous step.

3. Deployment via OSM of an MQTT broker as a MEC application

running in the edge. The MQTT broker registers its service to the MEC Platform, providing details of the exposed MQTT endpoint according to a standardized JSON format, as reported in the following extract:

```
{"serInstanceId": "Mec-Broker",
 ...
 "transportInfo":{
  ...
  "type": "MB_TOPIC_BASED",
  "protocol": "MQTT",
  "endpoint":{
   "addresses":[{
    "host" : "mec-broker.mec.host",
    "port" : "1883"}]
   } } }
```

Specifically, the `transportInfo` section contains information such as the `type` of messaging mechanism used (e.g., a topic-based message bus which routes messages to receivers based on topic subscriptions), the `protocol` used (MQTT in the example) and on which `endpoint` the service is available.

4. Subscription by the sink in the core cloud to the MQTT broker application.

5. Deployment via the fog orchestrator of an MQTT publisher as a MEC application running in the fog domain.

   The application `Mec-app` registers its service to the MEC Platform by simply exposing the REST endpoints to be used to start or stop the generation of MQTT traffic, therefore the value of the key `endpoint` will change to `uris`, which is a list of two elements `mec-app.fog.host/start-sensing` and `mec-app.fog.host/stop-sensing` respectively. Of course in this case the `type` and `protocol` fields in the service descriptor will be different, with `"type": "REST_HTTP"` and `"protocol": "HTTP"`.

6. Generation of MQTT traffic from the MEC application in the fog node, initiated by a REST call to its `/start-sensing` endpoint.

7. Deployment via OSM and Kubernetes of another MQTT publisher as a MEC application running in the edge domain. The application registers its service (start/stop sensing) to the MEC Platform.

8. Generation of MQTT traffic from the MEC application in the edge node.

9. Interruption of MQTT traffic generated by the MEC application in the fog domain, caused by a REST call to its `/stop-sensing` endpoint.

10. Interruption of MQTT traffic generated by the MEC application in the edge domain.

The described steps are represented in the sequence diagram of Fig. 5.3, limited to steps 3) to 6) for readability reasons. The deployment of any new MEC application in the fog or edge domain will follow the same steps as for the deployment of the first MEC application in the fog or the MQTT broker in the edge domain, respectively. In the experiment, multiple MQTT publishers were deployed in the form of additional MEC applications, following the steps described above.

In this PoC, MQTT was the only protocol employed to transmit sensor data. However, the system is completely agnostic to the protocol of choice. For example, OPC UA, a common industrial protocol, still uses TCP or HTTP/S as underlying transport, therefore its MEC 011 service definition would still be similar to the one described for MQTT, with `"protocol": "TCP/HTTP"` and of course with the correct host and port pair.

5.5 Evaluation

The proposed architecture implementation is evaluated by measuring the response time of the most relevant APIs and the footprint

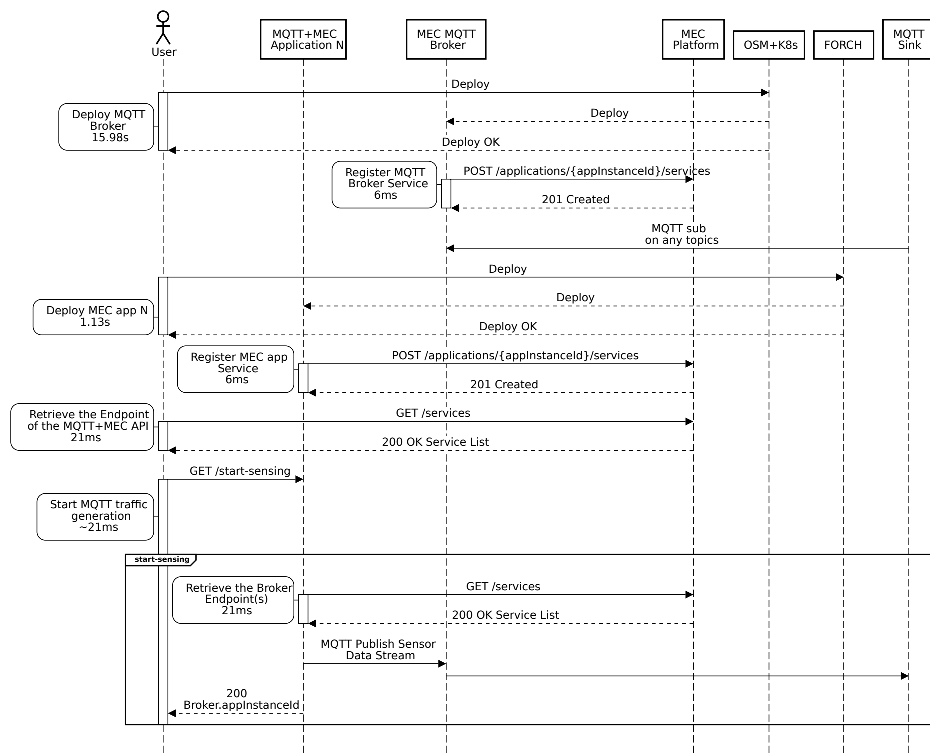


Figure 5.3: Sequence diagram of part of the described PoC evaluation, with measured response times.

of the required running components. The interoperability of different computing platforms is validated by observing the MQTT traffic generated by MEC applications in different domains. Along with the partial representation of the PoC steps, Fig. 5.3 reports the average values of the measured time required for the system to perform each individual action. This assessment shows that the time needed to deploy an MQTT MEC application and start the MQTT message flow towards a broker heavily depends on the domain chosen, ranging from slightly more than 1s for deployments in the fog domain, to almost 16s in the edge domain. It is however worth to mention that the total time required to deploy from scratch the first complete working MQTT mechanism for this PoC, including the MQTT broker in the edge and an MQTT publisher in the fog, is marginally higher than 17 seconds, proving the advantage offered by the proposed MEC-based architecture to automate the deployment of IIoTaaS applications.

A foreseeable bottleneck resides in the MEC Platform itself. As shown in Fig 5.3, MEC applications need to interact with it to register and discover services. The time required to perform this operation has been observed to grow linearly with the amount of simultaneous requests and the number of registered services. However, improved performance of the MEC Platform can be achieved by applying the scaling mechanisms offered by Kubernetes.

Table 5.1 reports the amount of storage and memory resources each node needs to support hosting MEC applications, in the two considered domains. Both Kubernetes and FORCH are configured to use the same container runtime engine (i.e., Docker), but the former platform is designed for more general and complex scenarios, thus requiring more software components running in the edge nodes to operate the cluster. As expected, the results highlight that the resource utilization on fog nodes is smaller compared to edge nodes. Furthermore, the resource consumption of the employed MEC Platform is comparable to that of MEC applications. Their container images occupy about 60 MB of storage space and require approximately 21 MB of RAM to be run.

Table 5.1: Resource utilization in different computing domains.

Domain	Disk util. [MB]	RAM util. [MB]
Edge	1366	202
Fog	124	180

In Fig. 5.4, every rising/falling edge of the curve represents the activation/deactivation of an MQTT message flow, as perceived by the only subscriber deployed, i.e., the sink application located in the cloud domain and subscribed to all MQTT topics. The line represents the amount of MQTT traffic received by said subscriber, corresponding to the sum of all MQTT data flows generated by all MQTT publishers that are active at a given time. MQTT publishers are deployed as MEC applications both in the fog and in the edge domains and are activated according to an alternating pattern. Specifically, the first publisher to be activated resided in the fog domain, the second one in the edge domain, the third one in the fog, and so on. MQTT traffic generation only lasts for a limited amount of time, after which the publisher stops generating data and remains silent. In this particular example, in order to keep the figure readable, a maximum of five concurrent MQTT clients were kept active at any given time. However, this is not a generic upper bound, which would depend on available resources. The asynchronous activation of publishers causes a variable superposition of MQTT traffic at the subscriber, resulting in a step-shaped curve. The fact that the subscriber receives MQTT traffic from all publishers, regardless of the technological domain they are deployed onto, proves the effective interoperability of the proposed solution.

In this chapter we focus on the application/service deployment process and limit our proof of concept to the management plane aspects of the proposed architecture. As for the data-plane performance, which depends on hardware capacity, adopted technology, and geographic location, there is no general consensus on typical values for the latency in the three different domains this work considers, and only generic assumptions are typically made [B64].

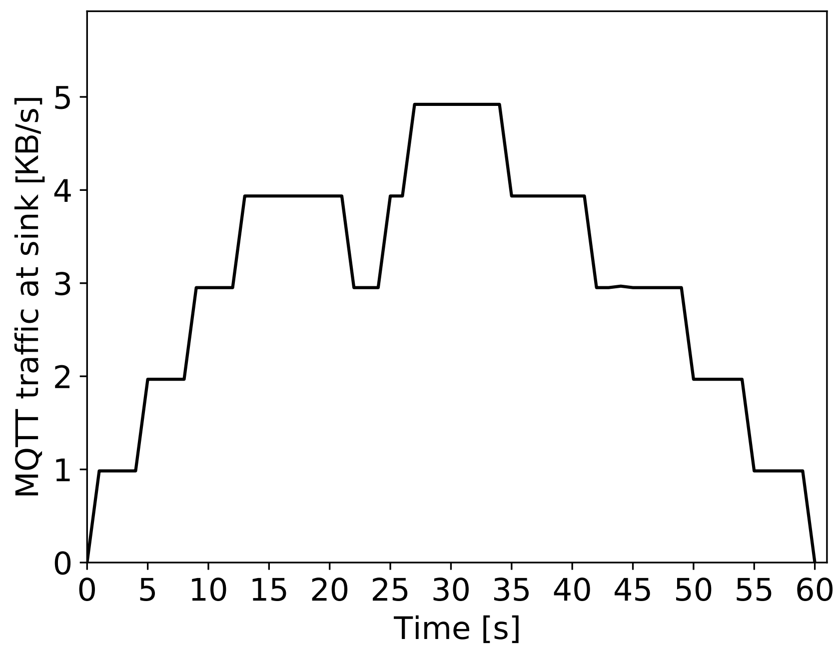


Figure 5.4: Evolution of MQTT traffic received by the MQTT subscriber (sink) running in the core cloud, while varying the number of MQTT publishers and their deployment domain.

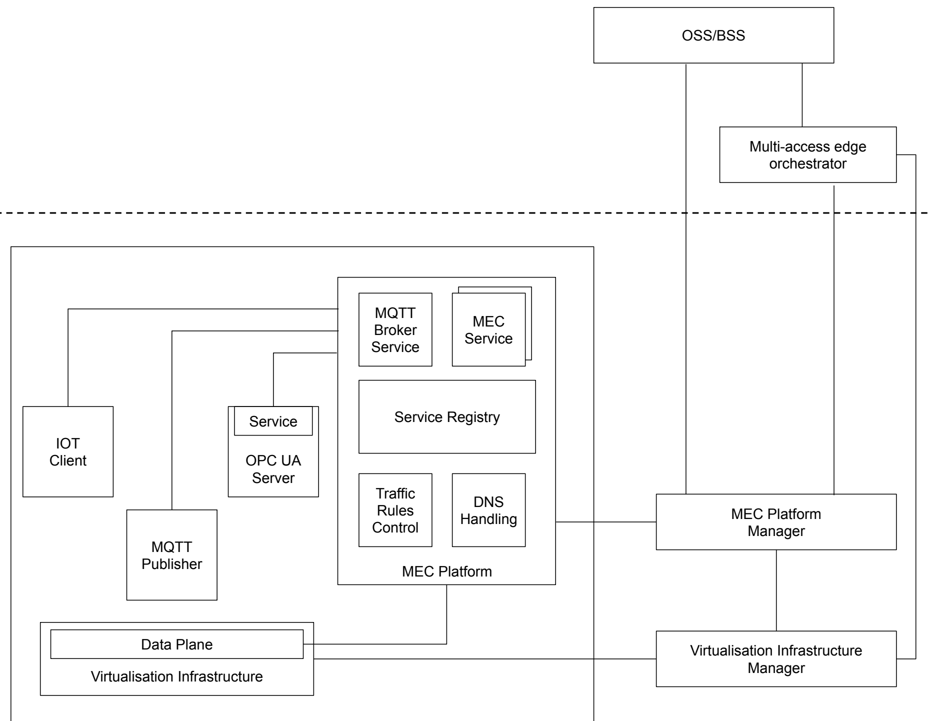


Figure 5.5: High Level MEC System Architecture

However, based on practical experience, it is reasonable to consider the data plane latency to be below 1ms for fog resources, around 5ms for edge resources, and around 50ms for core cloud resources.

5.6 MEC011 Extension

The approach presented was further developed to ease the IoT data discovery thanks to a custom extension to MEC011 APIs. Furthermore, more IIoT components were integrated inside the MEC Architecture, Fig: 5.5. In this implementation, the MQTT Broker is exposed by the MEC Platform as one of the MEC services offered with a custom interface. Other MEC applications requiring an MQTT broker (e.g., the MQTT Publisher in Fig. 5.5) can discover its endpoint using the MEC 011 API for service discovery. Since the standard service endpoint in MEC 011 contains only the URI or the IP/Port pair to use to consume the service, in our implementation we defined an al-

ternative endpoint type named `mqtt-topic`, to transport also the list of topics that the MQTT broker is serving. An example of this new endpoint definition is reported here:

```
...
    "endpoint": {
        "alternative":{
            "mqtt-topics":{
                "host" : "mqtt-broker",
                "port" : "1883",
                "topics" : [dev1/topic1]
            }  }  }
...
```

In order to add topics to this list, an additional REST endpoint was added to the MEC Platform, through which MQTT publishers can add, modify, or delete topics. This could be configured as an automated Day 1 operation performed at start-up time. Like MQTT, OPC UA is a protocol commonly used in IoT scenarios. It is a vendor-independent platform that gives the operator the possibility to design its own information model: objects, variables, and methods that are offered by the server [B65]. It is based on a publish/subscribe mechanism to create a standardized client-server architecture. When an entity requires a certain service (Client), it asks a third component (Discovery Server (DS)) for the list of services. The DS keeps track of the endpoint information of all the entities registered (Servers). In this way, the client can directly connect to the server it is looking for. Since the DS acts as a sort of catalog, in our implementation it is integrated directly inside the Service Registry of the MEC Platform. Therefore, all the OPC UA entities offering any functionality need to register their endpoints and details about their services to the MEC Platform. As for MQTT, the service registration is made through the adoption of the MEC 011 API, in particular by means of a REST API POST request to the corresponding `/applications/{appInstanceId}/services` endpoint. An OPC UA Server will automatically register its services to the MEC Platform during its start-up phase. Similarly to what was

done for MQTT, an extension to the MEC 011 service endpoint definition was defined, as reported in the following example:

```
...
  "endpoint": {
    "alternative": {
      "opcua": [  {
        "port": "4840",
        "host": "opcua-server",
        "uri": "namespace.uri",
        "objects": [  {
          "object_name": "My_Object",
          "variables": [
            "Temperature_Sensor",
            "Water_Sensor"]  }  ]
    }  ]  }  }
...
```

As can be seen from the example, in this case, the OPC UA server not only registers its host/port pair but also adds information about its internal information model. Specifically, inside the `opcua` list, we can find all the details that an external client can use to access those services, such as the fields `host` and `port`. Then the field `uri` is used for the identification of a particular "object" among the ones listed by the server. Each object is identified by its `object_name` and includes all the `variables` that a client may be interested in for the subscription.

With proper extensions to the standard interfaces defined by ETSI, it is possible to further simplify the interaction between IoT services even in a multi-vendor/multi-protocol scenario. With this addition, an IoT client could not only discover the endpoint used by the IoT data sources available but also what protocols they are using and which type of data.

Chapter 6

Intent Based Networking

6.1 Preliminary work

As stated in Chapter 1, a step further from the concepts of network programmability and virtual service provisioning is Intent Based Networking (IBN). Recalling the content of the introduction, this paradigm tries to abstract the physical detail of the underlying infrastructure offering a declarative interface for the users. Therefore, it allows easier interaction with the networking environment. A preliminary work done in this direction was [0], in which a JSON format to specify intent was proposed. The scope of this intent was limited to SFC deployment over Openstack datacenters managed by OSM. At the time of publication, there were not a lot of works proposing practical implementations of this problem. The architecture of the test bed used to test and prove this approach is depicted in Fig. 6.1. The test bed is deployed on bare-metal servers from the CloudLab facilities [B29]. OpenStack (Stein release via Devstack) is the open-source solution chosen to realize the cloud infrastructure where virtualized service components are instantiated[B25]. A small-scale cluster composed of four OpenStack compute nodes is used, including one node acting also as a controller. An additional node is used to host OpenSource MANO (Release SIX) [B18] and the Intent Layer implementation. The former is an open-source implementation of the ETSI MANO framework, officially supported by ETSI. The latter is an original software compo-

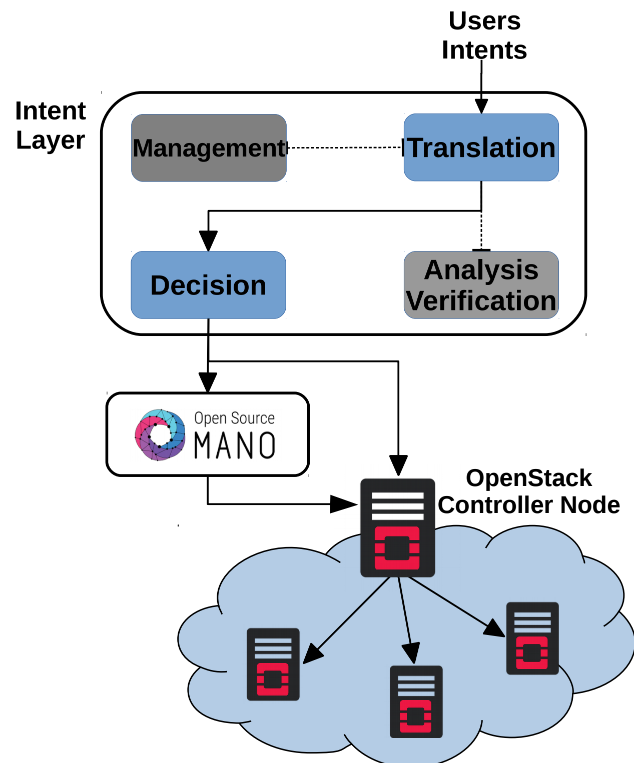


Figure 6.1: Testbed architecture.

nent we developed for the purpose of this demonstrator, implemented though a Python script which exposes a REST interface to accept users intents in a JSON format. The SFC intent specification format

is structured as follows:

```
{
  "name": "intent_name",
  "service_blocks": [
    {
      "block": "service_block_name",
      "managed": Bool:Management_Required,
      "order": int:Order_inside_SFP ,
      "symmetric": Bool:Block_On_Reverse_Path
    },
    ...
  ],
  "service_requirements": {
    ...
  }
}
```

Each `"block"` included in the intent specification represents a component of the service chain. The choice of a list of service blocks is similar to the one made in [B66], which uses a list of `middleboxes` to specify a service chain. The `"managed"` property specifies whether that component must be equipped with an interface connected to a management network; the `"order"` attribute is used to determine the sequence of blocks in the service chain; the `"symmetric"` attribute specifies whether the same component must be traversed in both traffic directions. An additional section named `"service_requirements"` can be included to specify some kind of quality of service envelope (e.g., required service level, latency, etc.) The intent specification presented above is to be considered as a *service-level* intent. The block properties were defined as the result of an abstraction process determining what a SFC must achieve, i.e. an ordered list of service components, possibly managed, to be selectively applied to one-way or two-way traffic flows. Although we believe these properties are general enough to be applied to any specific type of service, the intent specification can be extended by defining additional properties along with their mapping to the SFC configuration.

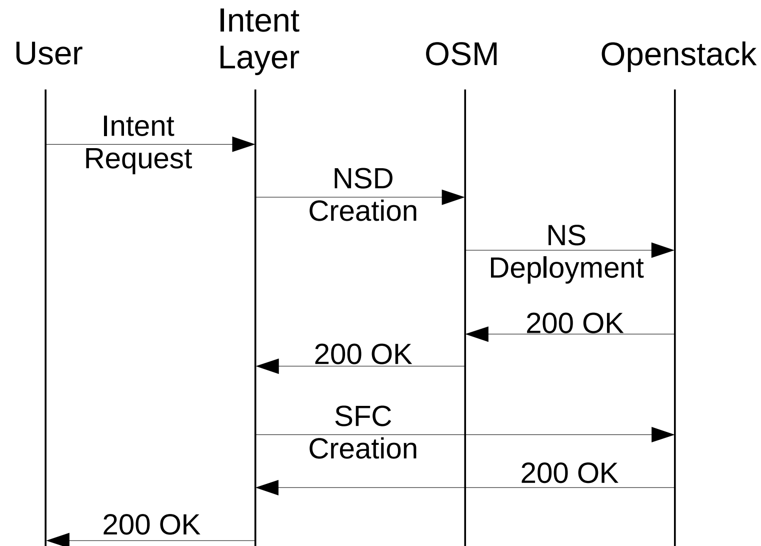


Figure 6.2: Service deployment workflow.

Following the architecture proposed in [B67][1], a valid intent received through the REST interface is translated into the required set of operations to be applied on the underlying infrastructure. More specifically, according to the list of blocks requested in `"service_blocks"`, the translator composes the Network Service Descriptor (NSD) in the OSM format [B43]. The descriptor includes the details of the Virtualized Network Functions (VNFs) needed to deploy the requested service blocks, as well as their interconnecting networks. We assume that the implementation details of the VNFs are stored in a service component catalogue that represents the knowledge base on which the translator is able to map high-level intents into implementation-specific service components. In order to allow further customization, the single generic service component can be adjusted by the translator to the service needs by taking advantage of a specific list of OSM configurations, including “Day 0” configurations (related to the initial set of computing and network resources) and “Day 1” configurations

[1]This is the updated version of the IETF draft. However, at the time of publication, the version was [B68]

(including the sequence of operations to be performed immediately after launching the VNF instance). Additional "Day 2" (i.e., run-time) configurations are possible through the management network, but only for blocks with the `"managed"` property set to true.

The NSD is then onboarded on OSM by the decision block, which triggers the instantiation of the VNFs in the OpenStack cluster by sending the correct command to OSM. The decision block is also in charge of deploying the Service Function Path (SFP) that traverses all the blocks requested in the intent specification in the given order. To do so, we chose to use directly the OpenStack SFC-plugin [B26]. The main reason is that OSM currently supports only a subset of the capabilities granted by the plugin, whereas controlling directly the OpenStack plugins increases the flexibility of the service chaining operations. In particular, the decision block is also enabled to select the encapsulation method (i.e., NSH or MPLS) used to implement the SFP, and to determine whether the VNFs along the path are aware or unaware of that. In case of so-called "SFC unaware" service functions, an SFC proxy function must be deployed [B4]. The detailed steps required to create a SFP with the SFC-plugin of OpenStack are described in [B45].

The different phases needed for a service deployment are depicted in Fig. 6.2.

To better understand the procedure of mapping an intent to an NSD, an example is given. The service intent is expressed as follows:

```
{ "name": "serviceA",
  "service_blocks": [
    {
      "block": "dpi",
      "managed": true,
      "order": 1,
      "symmetric": true
    },
    {
      "block": "firewall",
      "managed": false,
```

```
    "order": 0,
    "symmetric": true  } ] }
```

The requested service is composed by two different symmetric VNFs, a deep packet inspector (`dpi`) and a firewall (`firewall`). The former requires a connection to a management network, while the latter does not need it. The intent is then mapped by the translation component into the following NSD, represented in the YAML format used by OSM:

```
nsd:nsd-catalog:
  nsd:
  -  constituent-vnfd:
     -  member-vnf-index: 0
        vnfd-id-ref: firewall_vnfd
     -  member-vnf-index: 1
        vnfd-id-ref: dpi_vnfd
  ...
  vld:
  -  id: dataNet
     type: ELAN
     vnfd-connection-point-ref:
     -  member-vnf-index-ref: 0
        vnfd-connection-point-ref: vnf-cp-data
        vnfd-id-ref: firewall_vnfd
     -  member-vnf-index-ref: 1
        vnfd-connection-point-ref: vnf-cp-data
        vnfd-id-ref: dpi_vnfd
  -  id: mgmtNet
     type: ELAN
     vnfd-connection-point-ref:
     -  member-vnf-index-ref: 1
        vnfd-connection-point-ref: vnf-cp-mgmt
        vnfd-id-ref: dpi_vnfd
```

As can be seen, the descriptor includes the two required VNFs by calling their respective VNF descriptors (`firewall_vnfd` and `dpi_vnfd`)

taken from a catalog maintained by OSM. These two VNFs are then connected to their respective virtual networks, as specified by the virtual link descriptors (`dataNet` and `mgmtNet`).

Recalling Fig. 6.1, at the current state our Intent Layer implementation does not implement any management and analysis/verification functionality yet. In particular, the latter should monitor whether the service requirements specified in the intent are respected.

A live demo of this work was presented during the IETF 108 Hackathon coupled with the "Slice Intent" proposed by Molka Gharbaoui and Barbara Martini (CNIT, Italy). After the demo, these two types of intent were used as intent classification examples and are now part of the related IETF Draft from the NMRG group [B69].

To conclude, this work can be considered a first step in the IBN domain with a practical implementation of this type of system, even if limited. An insight gained from this work and corroborated by other related publications was the lack of a formal and general description of these concepts. Specifically, this work and others tackling the same problems (e.g., [B70]) tend to propose their own data models to describe intents for their specific, and sometimes limited, application areas. The next section of this chapter contains a formal way to formalize the intent using mathematical tools trying to overcome these limitations.

6.2 Formal definition of IBN

Intents are inherently a flexible and abstract way to express network operation. They should support composition since the network actions could be composed in different ways by different intent specifications. For this reason, category theory could be the suitable mathematical tool to reason about IBN. Since the main aim of this branch of mathematics is to provide a way to describe and work on abstract concepts, even on math itself. Furthermore, category theory has a strict connection with (functional) programming, which could lead directly from the theoretical representation of the problem to its realization. Func-

tional programming could be a good fit for the IBN world thanks to its declarative nature, conversely to the imperative one adopted by other programming languages (e.g., C).

Applied Category Theory is becoming a relevant field in research, showing how category theory can be applied to different fields outside of “pure mathematics”. For example, in [B71] Coecke, Sadrzadeh, and Clark applied category theory to natural language processing, defining a model that characterizes natural language expressions and their meaning leveraging tools from category theory. These concepts are also adopted in the area of modeling cyber-physical systems [B72] [B73]. Jacobs et al., in [B70] propose a process for intent refinement, which uses AI to process intent request expressed in natural language and transform them into an intermediate format *Nile*, that can be fed back to the operator/user to be validated before its deployment. The same formalism is used and extended in [B74] to cover a broader set of use cases, specifically ones related to traffic rerouting and service traffic protection. The formalism that will be presented in the remainder of this section would still be valid to describe the aforementioned approaches.

First of all, it is necessary to define concepts of category theory that will be use in the remainder of the text. The first of course would be the introduction of a *category*. Following the definition given by Fong and Spivak in [B75], to specify a category C:

i one specifies a collection Ob(C), elements of which are called objects.

ii for every two objects c, d, one specifies a set C(c, d), elements of which are called morphisms from c to d.

iii for every object $c \in Ob(C)$, one specifies a morphism $id_c \in C(c, c)$, called the identity morphism on c.

iv for every three objects $c, d, e \in Ob(C)$ and morphisms $f \in C(c, d)$ and $g \in C(d, e)$, one specifies a morphism $f \circ g \in C(c, e)$, called the composite of f and g.

We will sometimes write an object $c \in C$, instead of $c \in Ob(C)$. It will also be convenient to denote elements $f \in C(c, d)$ as $f : c \to d$. Here, c is called the domain of f , and d is called the codomain of f. These constituents are required to satisfy two conditions:

a *unitality*: for any morphism $f : c \to d$, composing with the identities at c or d does nothing: $id_c \circ f = f$ and $f \circ id_d = f$.

b *associativity*: for any three morphisms $f : c_0 \to c_1$, $g : c_1 \to c_2$, and $h : c_2 \to c_3$, the following are equal: $(f \circ g) \circ h = f \circ (g \circ h)$. We write this composite simply as $f \circ g \circ h$.

Another important concept to introduce is the *functor*. A functor maps all objects of a category C to objects of a category D, while preserving its structure (i.e., identities and composition). More formally [B75], let C and D be categories. To specify a functor from C to D, denoted $F : C \to D$,

i for every object $c \in Ob(C)$, one specifies an object $F(c) \in Ob(D)$;

ii for every morphism $f : c_1 \to c_2$ in C, one specifies a morphism $F(f) : F(c_1) \to F(c_2)$ in D.

Which must satisfy two properties:

a for every object $c \in Ob(C)$, we have $F(id_c) = id_{F(c)}$;

b for every three objects c_1 , c_2 , $c_3 \in Ob(C)$ and two morphisms $f \in C(c_1, c_2), g \in C(c_2, c_3)$, the equation $F(f \circ g) = F(f) \circ F(g)$ holds in D.

The link between Haskell, or functional programming in general, and category theory might not be easy to see. However, it is possible to construct a category *Hask* [B76] in which $Ob(Hask)$ contains all Haskell data types (e.g., `Int`, `Bool`, etc.) and morphisms between these are function between types (e.g., $isEven :: Int \to Bool$). This construction can be considered a category, up to some minor approximation. For example, let's consider a function `f :: A -> B` between types `A` and `B`. In Haskell, we could define an identity morphism

`id` such that `id . f = f . id = f`. However, since Haskell supports polymorphic functions this syntax is completely correct but does not directly translate into the *unitiality* condition presented. In category theory morphisms are only monomorphic (i.e., a morphism has a unique source and target objects). Therefore, by rewriting the above formula and by properly instantiating (i.e., fixing the input and output data types) each identity function, `(id :: B -> B) . f = f . (id :: A -> A) = f`, the condition is now satisfied. By considering *Hask* as a category we can of course use all the other construction defined in category theory (e.g., functors, monad, etc.). For example, a new type definition in Haskell could be seen as an endofunctor on *Hask* (a functor from *Hask* to *Hask*). Since it maps types to a new type and with its `fmap` it preserves morphisms (functions) between the starting types.

The focus of this work is trying to formalize an approach for an IBN system using a categorical approach, to be then implemented with Haskell. First of all, a category to represent intents was designed. The object of this category can be seen as a subset of all possible text in English (or even in other languages), in other words the objects are all possible well formed text related to network management that can be constructed. While morphisms can describe relationship between "similar" intent requests. Specifically, a kind of ordering can be introduced between element of this set, represented by the symbol $\leq$. For example, let $Intent_1$ be "Deploy low-latency Service X" and $Intent_2$ be "Deploy Service X", then $Intent_1 \leq Intent_2$, since the deployment of a service X with a low-latency requirement of course implies the deployment of service X. Then, it is easy to prove that this construction is actually a category, since:

- for every Intent I_i in the object set, we have $I_i \leq I_i$. This is trivial since it is clear that an Intent implies itself (identity morphism).

- for every three Intent I_1, I_2, I_3 in the object set, and $I_1 \leq I_2, I_2 \leq I_3$, then $I_1 \leq I_3$. In other words if I_1 implies I_2 and I_2 implies I_3 then of course I_1 will imply I_3 (composition rule).

Furthermore, this category is a partially ordered category. In fact, if exists a morphism between two objects then they are related as described, and that morphism is unique. Partially since there might be object not correlated with others. Alternatively, this category can be seen as the free category obtained from a partially ordered set, which is a set with an "ordering" function defined between its elements.

Inside this category a "product" operator can be defined $\otimes$ acting between its objects. This operator could be used to link different intent expression, its use could resemble a logical `and`. For example, let $Intent_1$ be "Deploy a web server" and $Intent_2$ "Deploy a Firewall", then $Intent_1 \otimes Intent_2$ would be "Deploy a web server *and* deploy a Firewall". For this reason, this operator could be used to build a structure inside the category in which complex intent request are linked to their constituent components through this operator. By defining an identity object for this operator, it is possible to prove that the "Intent category" equipped with this product is a "monoidal preorder" (i.e., a free category obtained from a preorder set equipped with a monoidal product). The identity object for this operator should represent an intent request that if multiplied, or logically linked, to any other intent the resulting intent would not change. Thus, this identity object would be something like a *Null Operation* intent. Then to prove that $\otimes$ is actually a monoidal operator, the following properties need to hold:

- **Monotonicity:** For all $x_1, x_2, y_1, y_2 \in I$, if $x_1 \leq y_1$ and $x_2 \leq y_2$, then $x_1 \otimes x_2 \leq y_1 \otimes y_2$;
- **Unitality:** Let Id be the identity object for $\otimes$, then for all $x \in I$ the left and right identities hold $Id \otimes x = x \otimes Id = x$;
- **Associativity:** For all $x, y, z \in I$ $(x \otimes y) \otimes z = x \otimes (y \otimes z)$

Let's give an example for the first property, fixing $x_1 =$"Deploy low-latency Service X"; $y_1 =$"Deploy Service X"; $x_2 =$ "Activate a firewall between 8 am and 10 pm"; $y_2 =$ "Activate a firewall". So following the definition, $x_1 \otimes x_2$ and $y_1 \otimes y_2$ would be equals to "Deploy low-latency Service X *and* activate a firewall between 8 am and 10 pm"

and "Deploy Service X *and* activate a firewall" respectively. Since $x_1 \leq y_1$ and $x_2 \leq y_2$, it's easy to see that also $x_1 \otimes x_2 \leq y_1 \otimes y_2$, thus satisfying the monotonicity property. An additional property that could be discussed and proven is **symmetry**, meaning given $x, y \in I, x \otimes y = y \otimes x$.

Another definition of this product between objects could be also obtained by means of universal construction. For any Intent I_i having two projection towards I_1 and I_2, meaning I_i implies both I_1 and I_2. It exists an object $I_1 \otimes I_2$ such that there is an unique morphism going from I_i and $I_1 \otimes I_2$ that makes the two "triangles" in Fig. 6.3 to commute.

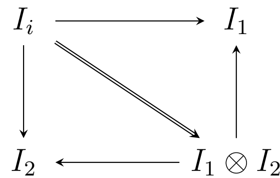


Figure 6.3: Commuting diagram for categorical product.

This could be interpreted as if objects I_i are all the possible intent specifications requiring a service composed by the ones requested by I_1 and I_2, being $I_1 \otimes I_3$ the way of describing the composite service in which its component are "easier" to identify. Here an example that may clarify this, let I_1 be "Deploy a web server" and I_2 "Deploy a firewall", then $I_1 \otimes I_2$ would be "Deploy a web server *and* a firewall". In this example the "I_is" would be expressions like "Deploy a secured HTTP server" or "Deploy a HTTP server and secure it".

Having defined the category representing the intent requests (I), the next step could be to define a category representing the "services required"($Services$), in other words what an intent is requiring. The intent category could then be linked to this new one through a functor. Which would map all object from I (i.e., the intents) to object in $Services$, while preserving the structure of the starting category. Meaning, let $F :: I \rightarrow Services$ be the functor between the two categories and $I_1, I_2 \in I$ such that $\exists f :: I_1 \rightarrow I_2$ then

$\exists F(f) :: F(I_1) \to F(I_2)$. The object of this category represent all the services that can be asked by an intent. While morphisms between these objects could embed composition rules between them.

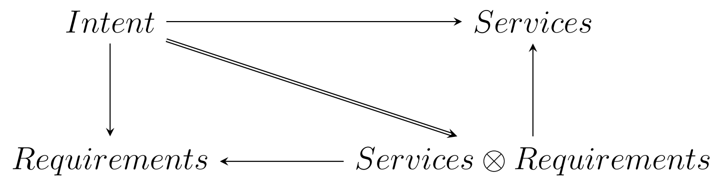
A similar approach could be followed for another category representing the "service requirements" (*Requirements*). This category will embed all the modifiers that a particular intent could ask, for example specific QoS values to satisfy (e.g., bandwidth, latency etc.) or period of time in which the intent will be active (e.g., "everyday","all Mondays", "only between 8 a.m. and 10 a.m.). Also for this category a functor from *Intent* can be defined, with the same properties as above.

To recap a category containing all possible intent requests has been defined (*Intent*), with two functors mapping it to other two categories, *Services* and *Requirements*. The former representing the services that can be required by intents. The latter embedding all possible "modifiers" that an intent could ask. Going a step forward with this abstraction, the category *Cat* can be considered. *Cat* is a large category in which its objects are categories, and morphism between these objects are functors between categories. Recalling the universal construction used for the monoidal product inside *Intent* a similar approach could be applied here with the three category defined. Specif-

$$\begin{array}{ccc} Intent & \longrightarrow & Services \\ \downarrow & & \\ Requirements & & \end{array}$$

ically we see that *Intent* has two projections, one going to *Services* and the other one to *Requirements*. Recalling the construction of product in a category, then an object "$Services \otimes Requirements$" can be defined, such that it exists a unique morphism going from *Intent* to that object.

It is important to remind that the last definition given was obtained reasoning on the category of categories. Therefore, $Services \otimes Requirements$ is actually a whole category and the unique morphism



is a functor from the *Intent* category.

The idea here is that the objects in the category $Services \otimes Requirements$ contains all the information about what an intent is asking. These objects can be seen as pairs, combining the services (or generally actions on the network) and the specific requirements from them. In other words, the object set of category $Services \otimes Requirements$ can be seen as the cartesian product of the object sets of *Services* and *Requirements*, $Ob(Services \otimes Requirements) = \{(s, r) | s \in Ob(Services), r \in Ob(Requirements)\}$. While for morphisms, for any $(s_1, r_1), (s_2, r_2) \in Ob(Services \otimes Requirements)$ then a $f : (s_1, r_1) \rightarrow (s_2, r_2)$ exists if and only if $g_1 : s_1 \rightarrow s_2$ and $g_2 : r_1 \rightarrow r_2$ exist in *Services* and *Requirements* respectively. This could be seen as a proof that given a way to map an intent to the services it is asking for and to the requirements of this request, then it exist a mapping of this intent to a couple (service, requirements), which contains all the information carried by the intent, and this mapping is unique (by universal construction).

A key aspect that it's worth highlighting is the following. No details were given on the internal structure of the "Service" and "Requirement" categories, but only to the Intent one. By leveraging the abstraction granted by the category theory's tools, it's possible to reason and define a structure at the "natural language level" (i.e., at the intent level). This structure is preserved in the linked categories (e.g., Service and Requirement) through categorical relationships (e.g., functors), without the need of "looking inside" them. In other words, we could say that the "Service" and "Requirement" categories can be seen as intermediate steps between an intent reception and its actuation on the system, the Translation/Intent-Based System (IBS) Space in the intent lifecycle [B77]. However, we don't need to define precise data

models for these intermediate steps to derive their properties since they are inherited by the structure of intents expressed in natural language.

In this view, the functors going from the Intent category to the other two can be seen as ”meaning extracting” functions, extracting services and their requirements contained in natural expressions. More than one functor could exist between two categories (i.e., two different mappings). Natural transformations are a way to introduce a relationship between two functors acting on the same couple of categories. Following Definition 3.49 given in [B75], let C and D be categories, and let $F, G : C \to D$ be functors. To specify a natural transformation $\alpha : F \implies G$, for each object $c \in C$, one specifies a morphism $\alpha_c : F(c) \to G(c)$ in D, called the c-component of α. These

$$\begin{array}{ccc} F(c) & \xrightarrow{\alpha_c} & G(c) \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ F(d) & \xrightarrow[\alpha_d]{} & G(d) \end{array}$$

Figure 6.4: Commuting diagram for the *naturality condition* (*naturality square*)

components must satisfy the following rule, called the *naturality condition*:

a for every morphism $f : c \to d$ in C, the following equation must hold: $F(f) \circ \alpha_d = \alpha_c \circ G(f)$ Fig. 6.4.

In this scenario, since the categories considered are constructed on top of preorders, we could say that given two functors F, G a natural transformation between them, $\alpha : F \implies G$, exists if for every c in the starting category $F(c) \leq G(c)$. In other words, G could be seen as a map with a “lower resolution” concerning the one done by F. Intuitively, this means that G is a “lossy” function, that loses parts of information carried by natural language intents during its map. An

extreme example of that could be a trivial functor that maps every object in *Intent* to the terminal object in *Services* (or *Requirements*).

Having defined these categories and how they relates with each other, the next step would be to look deepen in each one of them, using a Haskell-like syntax to start building the bridge between the theoretical construct and a possible implementation.

First of all, the objects of *Intent* could be described with a type defined as `data Intent = Intent String | NullOperation`. This example could also be used to explain how types can be defined in Haskell. The first "Intent" keyword here is a *type constructor* which will identify the type name. While the other two keywords on the right side of the equality sign, "Intent" and "NullOperation", are the data constructors. These specify how data of type `Intent` can be constructed. For this example, a value of type `Intent` can be constructed using either `NullOperation` without any parameter or `Intent` followed by a `String`. Here two examples, `intent1 = Intent "Deploy a firewall"` or `nullOpIntent = NullOperation`. In other words, an object of the category *Intent* is either a string (i.e., an actual intent request in natural language) or a null operation (i.e., the identity object defined for the internal monoidal product).

Then the same thing could be done for the object of the *Services* category. The related type could be defined as `data Action = Service (Phase, BasicService) | TemporalCompose [Action] | LogicalCompose [Action] | NoAction`. This is a recursive data type, a common type definition in functional programming. The new type `Action` can be a single `Service`, specified through a pair of new data types (`Phase` and `BasicService`) that will be described later, or by a composition of a list of `Action`. Specifically, two different kind of composition are defined, temporal and logical. The former represents a list of actions that need to be executed in a specific temporal order (e.g., deploy a firewall *and after* configure it). The latter describes an action that depends on a series of other logical components without a precise order, for example the deployment of a 5G core depends on the deployment of a set of other function (e.g., AMF, SMF, PCF etc.).

Lastly, the type constructor `NoAction` represents an "empty" `Action` object (i.e., an action that does nothing on the system), similar to empty list constructor `Nil` in classic recursive list definitions [B78]. The `BasicService` type is used to represent basic services or actions. The idea here is that this new type definition should contain all possible "building blocks" that can be composed (temporally or logically) together to construct a complex service required by an intent. A basic definition of this new type: `data BasicService = VnfId String | RouterConfig (String, String)| PathCreate (String,String)`. Of course, this type could be extended with other basic services that users may require (i.e., adding new data constructor), or by using more complex structures as input for them. Finally, the `Phase` type simply defines the phase of the lifecycle in which that action will take place. A basic definition of this new type: `data Phase = Add | Update | Restart | Remove`. Having defined objects in *Services*, the next step would be to define the structure of this category, in other words its morphisms. Morphisms can describe how services may be composed by other ones, for example: let $x, y \in Ob(Services)$ with $x =$ *Deploy a 5G Core Network* and $y =$ *Deploy an SMF function*, then $\exists\ f : x \to y$ meaning x is composed by y. To clarify, the definition of x and y used in the example is not completely correct, since they don't follow the definition given. However, a functor from the *Intent* and the *Services* has been defined, therefore each object in the first category is mapped to an object of the other one. For this reason, x and y can be seen as the objects that are mapped by intents *Deploy a 5G Core Network* and *Deploy an SMF function* respectively.

To better understand what the functor does between these two categories Fig. 6.5 is presented. The dashed lines connects intents to the objects in *Services*, or in other words to the service they are requiring. The same color has been used to highlight how the functor maps morphisms between the two categories.

From Fig. 6.5, it is possible to gain an insight on what is necessary to satisfy an intent request by taking into account the sub-tree having as root the mapped service. In other words, by knowing how to deploy

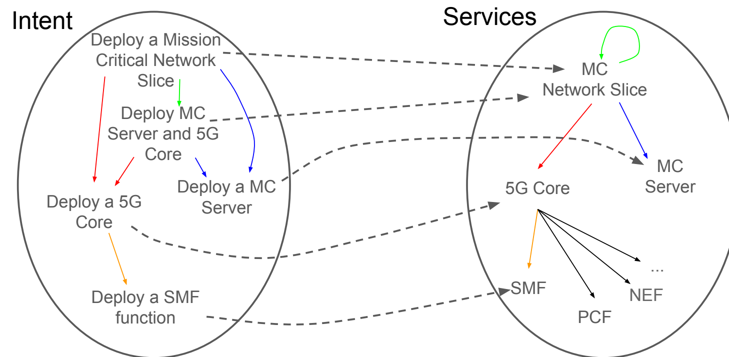


Figure 6.5: Graphical representation of the functor between the two categories *Intent* and *Services*.

the leaves of these trees and how to compose them, then it should be possible to deploy the required service.

For example the leaves in Fig. 6.5, using the type definition introduced above, can be defined as follows: `smf = Service (Add, VnfId "smfId")`, meaning variable `smf` of type `Action` is defined using its data constructor `Service` since we can consider it as a single vnf, thus not requiring any composition. This data constructor takes as argument a pair of objects. One of type `BasicService`, which was constructed using its data constructor `VnfId` followed by a string representing the identifier given to that specific function, `"smfId"` in the example. The other of type `Phase` built with its data constructor `Add`, since the intent in the example is requiring the deployment of a function. The same procedure can also be applied to the other leaves represented in the figure, of course with the correct VNF identifiers. Then these leaves can be composed together to construct, for example, the object representing a 5G core deployment: `5gCore = LogicalCompose[ (Service (Add, VnfId "smfId")), (Service (Add, VnfId "pcfId")), ...]`. Here, the data constructor `LogicalCompose` is used to identify the list of VNFs that are part of the 5G core service (e.g., SMF, PCF, etc.). The logical composition is simply grouping them together without a particular ordering in which the VNFs need to be deployed (i.e., in NFV-MANO terminology, a network service

composed by several VNFs without any specific Virtual Network Forwarding Graphs).

Of course, similar type definitions could be given for the *Requirements* category as well. In which, its objects can be seen as a composition of basic "intent modifiers", such as: Scope, to whom the intent is (e.g., network wide, single user); Time, when the intent should be active (e.g. "everyday from 2pm to 5pm", "always"); Latency Constraints Throughput Constraints. Using Haskell notation, `data SrvRequirement = Requirement BasicReq | ComposeReq [SrvRequirement] | NoReq`, with
`BasicReq = Latency _ | Bandwidth _ | Scope _ | Uptime _` . As for the other data type, also `BasiqReq` could be expanded to cover a larger set of requirements. After the definition of these new types, the ordering has to be defined. This ordering will map the relationship between two variables of the same type, embedding the concept of morphisms between objects in the categories from which the types derive. Since the categories were partial orders, the class `Ord` present in Haskell's prelude would not be the correct choice. Therefore a new class has to be defined for them. Since in a partial order, two objects may be related (e.g., less/equal/greater) or not a possible way to program this is using a construct like `Maybe Ord.Ordering`. In other words, if the relationship exists the comparison of the two variables will return `Just Ord` (e.g., LT,EQ,GT), while if it doesn't `Nothing`. An object of these new types would less than another one if it is a components of the latter.

Having defined the types describing the three categories considered, the next step would be the definition of the functors connecting them. In Haskell, this would mean defining functions taking as input objects of type `Intent` and returning `Action` or `SrvRequirement` (or directly a pair `(Action, SrvRequirement)`). For example, `extractAction :: Intent -> Action` or `processIntent :: Intent -> (Action, SrvRequirement)`. These functions should take natural language expressions and construct the trees representing the network operations they are asking for and their requirements.

To start testing these operations, I implemented a preliminary version of one actuation function, specifically, the one able to construct the Open Source Mano (OSM) Network Service Descriptor (NSD) required by a specific action. In detail, if the function is called with as input an action like `Service(Add, VnfId "vndfid")`, the Vnf identifier is added to the required fields of an OSM NSD data model, namely in the `vnfd-id` and `vnf-profile` lists. Alternatively, if the input of the function is a logical composition of a set of VNFs, then using a fold function (`foldr`) the behavior just described is applied recursively to all the elements in the list, accumulating all the results in the same network service descriptor. In a Haskell-like syntax:

```
generateNsd :: Action -> Nsd -> Nsd
generateNsd (Service (Add, VnfId vnfid))
   nsd = [...]
generateNsd (LogicalCompose actions) nsd =
   foldr generateNsd nsd actions
```

For example, assuming an input action like:

```
LogicalCompose[
        (Service (Add, VnfId "smfId")),
        (Service (Add, VnfId "pcfId"))]
```

the function constructs a valid OSM NSD containing the required VNFs. The relevant components of the descriptor generated from the previous example are:

```
nsd:
  nsd:
    df:
    - id: default-df
      vnf-profile:
      - id: pcfid
        virtual-link-connectivity:
        - constituent-cpd-id:
          - constituent-base-element-id:
```

```
          pcfid
        constituent-cpd-id: mgmt-ext
      virtual-link-profile-id: mgmtnet
    vnfd-id: pcfid
  - id: smfid
    virtual-link-connectivity:
    - constituent-cpd-id:
      - constituent-base-element-id:
          smfid
        constituent-cpd-id: mgmt-ext
      virtual-link-profile-id: mgmtnet
    vnfd-id: smfid
vnf-id:
- pcfid
- smfid
[...]
```

The NSD descriptor created with this function can then be onboarded on the OSM platform, and its deployment can be initialized. Furthermore, thanks to the type checking features of Haskell, the NSD format is automatically validated during every parsing and rendering of the YAML file, thus increasing the robustness of the code.

6.3 Future research tracks

This work has directly applied category theory concepts to IBN to build a formal representation of the intent specifications. This representation aims to help with the reasoning about this new paradigm while keeping a close relationship with functional programming and the possible implementations. I have shown a preliminary implementation of the categories for intent using Haskell. This implementation can take a definition of a service and constructs a valid OSM NSD containing the required VNFs, by generating its relevant components.

Further works can be made in both directions. For example, lenses in category theory [**spivak-lenses**] are used to describe in mathemati-

cal terms the concepts of *agent* and *environment*, thus they could serve as a suitable tool to model the interaction between the Intent System and the network infrastructure it is managing. They are also widely used in Haskell since they are one of the most powerful tools to access and modify data structures. Also, other mathematical concepts might expand and improve the proposed formalism. For instance, the intent resolution might be mapped into a Kolmogorov Problem. Therefore, works on Dialetica categories may link well with IBN. Likewise, the Yoneda lemma might also reveal interesting properties for the extraction of meaning from intent specifications.

Chapter 7

Conclusion

This thesis has presented the main evolution trends networks are undergoing today, proposing relevant works carried out for each one of them. Specifically, Chapter 3 presented two possible implementations for SFC. The former uses OpenFlow inside Openstack clusters while the latter proposes an Srv6 approach in an NFV-MANO environment. For both solutions, performance metrics are presented, showing the advantages and drawbacks of both (e.g., deployment times or the flexibility offered). Chapter 4 briefly reports deployment time results obtained using OSM. OSM is also the platform of choice for the two following chapters. Chapter 5 dealt with 5G network slicing applied to Mission Critical Communications. In this work, OSM is used to deploy a network slice composed of both the mobile core network components and the mission-critical ones over a multi-domain environment. It proves how an NFV architecture could support dynamic services with specific QoS indicators. Chapter 6 showed an application of MEC in IIoT environments. The described approach enables an Industry 4.0 infrastructure to configure and provide services with a high degree of flexibility and adaptability. As demonstrated by the proof-of-concept implementation, a sensing and data gathering service can be deployed on-demand over multiple technological domains (i.e., fog and edge cloud) in a seamless way and in a matter of tens of seconds. Furthermore, with proper extensions to the standard interfaces defined by ETSI, it is possible to further simplify the interaction

between IoT services even in a multi-vendor/multi-protocol scenario. Finally, Chapter 7 presents results in the field of network automation, specifically about intent. This chapter builds up from the previous ones, trying to move further by abstracting the concepts described in the first chapters. It presents an intent approach to deploy network resources and create the correct SFP in an NFV environment. Then, it proposes a formal model, using category theory, to describe intent. This approach could be beneficial because it allows to reason about intent-related concepts abstracting from specific, and maybe limited, implementations.

Acronyms

API Application Programming Interface. 5,

CLI Command Line Interface.

DC Data Center.

DNS Domain Name System.

DPI Deep Packet Inspection/Inspector.

IBN Intent-based Networking.

IC Integrity Check(er).

IDS Intrusion Detection System.

IETF Internet Engineering Task Force.

IoT Internet of Things.

LXD Linux Container Daemon. 28,

MANO Management and Orchestration.

MEC Multi-access Edge Computing.

NAT Network Address Translation/Translator.

NBI North-Bound Interface.

NFV Network Function Virtualization.

NFVO Network Function Virtualization Orchestrator.

NSD Network Service Descriptor. 30,

NSH Network Service Header.

QoS Quality of Service.

SDN Software-Defined Network(ing).

SF Service Function.

SFC Service Function Chain(ing).

SSH Secure SHell. 28,

TC Traffic Controller/Shaper.

VDU Virtual Deployment Unit. 30,

VIM Virtualized Infrastructure Manager.

VNF Virtual Network Function.

VNFM Virtual Network Function Manager.

WAN Wide Area Network.

WIM WAN Infrastructure Manager.

XaaS Everything-as-a-Service.

Bibliography

[B1] *Open Networking Foundation (ONF)*. URL: https://opennetworking.org/.

[B2] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM CCR* 38.2 (Mar. 2008), pp. 69–74. URL: http://doi.acm.org/10.1145/1355734.1355746.

[B3] F. Hu, Q. Hao, and K. Bao. "A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation". In: *IEEE Communications Surveys Tutorials* 16.4 (Fourth quarter 2014), pp. 2181–2206.

[B4] J. M. Halpern and C. Pignataro. *Service Function Chaining (SFC) Architecture*. RFC 7665. Oct. 2015. URL: https://rfc-editor.org/rfc/rfc7665.txt.

[B5] F. Callegati, W. Cerroni, C. Contoli, R. Cardone, M. Nocentini, and A. Manzalini. "SDN for dynamic NFV deployment". In: *IEEE Communications Magazine* 54.10 (Oct. 2016), pp. 89–95.

[B6] G. Davoli, W. Cerroni, C. Contoli, F. Foresta, and F. Callegati. "Implementation of Service Function Chaining Control Plane through OpenFlow". In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Nov. 2017.

[B7] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. "Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service". In: *ACM SIGCOMM 2012 Conference*. Helsinki, Finland: ACM, 2012.

[B8] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu. "Research Directions in Network Service Chaining". In: *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. Nov. 2013.

[B9] A. M. Medhat, T. Taleb, A. Elmangoush, G. A. Carella, S. Covaci, and T. Magedanz. "Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges". In: *IEEE Communications Magazine* 55.2 (Feb. 2017), pp. 216–223.

[B10] H. Chen, S. Xu, X. Wang, Y. Zhao, K. Li, Y. Wang, W. Wang, and L. M. Li. "Towards optimal outsourcing of service function chain across multiple clouds". In: *2016 IEEE International Conference on Communications (ICC)*. May 2016.

[B11] A. M. Medhat, G. A. Carella, M. Pauls, M. Monachesi, M. Corici, and T. Magedanz. "Resilient orchestration of Service Functions Chains in a NFV environment". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Nov. 2016.

[B12] M. T. Beck, J. F. Botero, and K. Samelin. "Resilient allocation of Service Function Chains". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2016, pp. 128–133.

[B13] T. Soenen, S. Sahhaf, W. Tavernier, P. Sköldström, D. Colle, and M. Pickavet. "A model to select the right infrastructure abstraction for Service Function Chaining". In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2016, pp. 233–239.

[B14] J. Lee, H. Ko, D. Suh, S. Jang, and S. Pack. "Overload and failure management in service function chaining". In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. July 2017.

[B15] S. A. Amiri, K.-T. Foerster, R. Jacob, and S. Schmid. "Charting the Algorithmic Complexity of Waypoint Routing". In: *SIGCOMM Comput. Commun. Rev.* 48.1 (Apr. 2018), pp. 42–48.

[B16] Q. Duan. "Modeling and Performance Analysis for Service Function Chaining in the SDN/NFV Architecture". In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. June 2018.

[B17] *Network Functions Virtualisation (NFV); Architectural Framework*. Accessed Jun. 16, 2019. The European Telecommunications Standards Institute (ETSI). 2013. URL: http://www.etsi.org/technologies-clusters/technologies/nfv.

[B18] *Open Source MANO*. URL: https://osm.etsi.org/.

[B19] *Multi-access Edge Computing (MEC); Framework and Reference Architecture*. Accessed Dec. 30, 2020. The European Telecommunications Standards Institute (ETSI). URL: https://www.etsi.org/deliver/etsi_gs/MEC/001_099/003/02.02.01_60/gs_mec003v020201p.pdf.

[B20] *Multi-access Edge Computing (MEC); Edge Platform Application Enablement*. Accessed Dec. 30, 2020. URL: https://www.etsi.org/deliver/etsi_gs/MEC/001_099/011/02.02.01_60/gs_MEC011v020201p.pdf.

[B21] *Cloud iNfrastructure Telco Taskforce*. Accessed Oct. 2021. URL: https://cntt.readthedocs.io/en/stable-kali/gov/chapters/chapter01.html.

[B22] *5G; System architecture for the 5G System (5GS) (3GPP TS 23.501 version 16.6.0 Release 16)*. URL: https://www.etsi.org/deliver/etsi_ts/123500_123599/123501/16.06.00_60/ts_123501v160600p.pdf.

[B23] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. *Intent-Based Networking - Concepts and Definitions*. Internet-Draft draft-irtf-nmrg-ibn-concepts-definitions-06. Work in Progress. Internet Engineering Task Force, Dec. 2021. 28 pp. URL: https://datatracker.ietf.org/doc/draft-irtf-nmrg-ibn-concepts-definitions.

[B24] L. Pang, C. Yang, D. Chen, Y. Song, and M. Guizani. "A Survey on Intent-Driven Networks". In: *IEEE Access* 8 (2020), pp. 22862–22873.

[B25] *OpenStack: Open Source Cloud Computing Software*. URL: https://www.openstack.org/.

[B26] *Service Function Chaining Extension for OpenStack Networking*. URL: https://docs.openstack.org/networking-sfc/latest/.

[B27] A. Farrel, S. Bryant, and J. Drake. *An MPLS-Based Forwarding Plane for Service Function Chaining*. RFC 8595. June 2019. URL: https://www.rfc-editor.org/info/rfc8595.

[B28] P. Quinn, U. Elzur, and C. Pignataro. *Network Service Header (NSH)*. RFC 8300. Jan. 2018. URL: https://www.rfc-editor.org/info/rfc8300.

[B29] *CloudLab*. URL: https://www.cloudlab.us.

[B30] A. Farrel, S. Bryant, and J. Drake. *An MPLS-based forwarding plane for Service Function Chaining*. Internet-Draft. IETF, Aug. 2018. 28 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-mpls-sfc-02.

[B31] M. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal. "Enhancing VNF performance by exploiting SR-IOV and DPDK packet processing acceleration". In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. Nov. 2015.

[B32] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir. *Segment Routing Architecture*. RFC 8402. IETF, July 2018.

[B33] F. Clad, X. Xu, C. Filsfils, D. Bernier, C. Li, B. Decraene, S. Ma, C. Yadlapalli, W. Henderickx, and S. Salsano. *Service Programming with Segment Routing [EXPIRED, ARCHIVED]*. Internet-Draft draft-xuclad-spring-sr-service-programming-05. IETF, Apr. 2019. 31 pp.

[B34] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li. *Segment Routing over IPv6 (SRv6) Network Programming*. RFC 8986. Feb. 2021. URL: `https://www.rfc-editor.org/info/rfc8986`.

[B35] A. Abdelsalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, and L. Veltri. "Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure". In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. July 2017, pp. 1–5.

[B36] P. L. Ventre, M. M. Tajiki, S. Salsano, and C. Filsfils. "SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Networks". In: *IEEE Transactions on Network and Service Management* 15.4 (Dec. 2018), pp. 1378–1392.

[B37] A. Mayer, S. Salsano, P. L. Ventre, A. Abdelsalam, L. Chiaraviglio, and C. Filsfils. "An Efficient Linux Kernel Implementation of Service Function Chaining for legacy VNFs based on IPv6 Segment Routing". In: *2019 5th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. June 2019, pp. 333–341.

[B38] *5G; Management and orchestration; Concepts, use cases and requirements*. Tech. rep. 3GPP TS 28.530 ver. 15.1.0 rel. 15. 3GPP, 2019.

[B39] *From Vertical Industry Requirements to Network Slice Characteristics*. Tech. rep. The GSM Association, 2018.

[B40] *JUJU*. URL: https://jaas.ai.

[B41] *Virtual Network Function Descriptor Information Model*. URL: http://osm-download.etsi.org/repository/osm/debian/ReleaseSIX/docs/osm-im/osm_im_trees/vnfd.html#.

[B42] *Cloud-Init Documentation*. URL: https://cloudinit.readthedocs.io/en/latest/.

[B43] *Network Service Descriptor Information Model*. URL: http://osm-download.etsi.org/repository/osm/debian/ReleaseSIX/docs/osm-im/osm_im_trees/nsd.html#.

[B44] *SRv6 - Linux Kernel Implementation - Advanced Configuration*. URL: https://segment-routing.org/index.php/Implementation/AdvancedConf.

[B45] D. Borsatti, G. Davoli, W. Cerroni, C. Contoli, and F. Callegati. "Performance of Service Function Chaining on the OpenStack Cloud Platform". In: *1st Workshop on Segment Routing and Service Function Chaining (SR+SFC 2018), 14th International Conference on Network and Service Management (CNSM)*. Nov. 2018, pp. 432–437.

[B46] *Mission Critical Services in 3GPP*. Accessed May 12, 2020. URL: https://www.3gpp.org/news-events/1875-mc%5C_services.

[B47] Leonardo s.p.a. *LTE broadband solutions*. URL: https://www.leonardocompany.com/it/security-cyber/professional-communications/.

[B48] *OpenAir Interface*. Accessed March 2020. URL: https://www.openairinterface.org.

[B49] *NextEPC*. Accessed March 2020. URL: https://nextepc.org.

[B50] GSMA. *From Vertical Industry Requirements to Network Slice Characteristics*.

[B51] *3GPP TS 28.530, Management and orchestration; Concepts, use cases and requirements, release 15.3*.

[B52] E. Ahmed, I. Yaqoob, A. Gani, M. Imran, and M. Guizani. "Internet-of-things-based smart environments: state of the art, taxonomy, and open research challenges". In: *IEEE Wireless Communications* 23.5 (2016), pp. 10–16.

[B53] T. Taleb, I. Afolabi, and M. Bagaa. "Orchestrating 5G Network Slices to Support Industrial Internet and to Shape Next-Generation Smart Factories". In: *IEEE Network* 33.4 (2019), pp. 146–154.

[B54] J. Cheng, W. Chen, F. Tao, and C.-L. Lin. "Industrial IoT in 5G environment towards smart manufacturing". In: *Journal of Industrial Information Integration* 10 (2018), pp. 10–19.

[B55] Y. Cai, B. Starly, P. Cohen, and Y.-S. Lee. "Sensor data and information fusion to construct digital-twins virtual machine tools for cyber-physical manufacturing". In: *Procedia Manufacturing* 10 (2017), pp. 1031–1042.

[B56] F. Tao, L. Zhang, Y. Liu, Y. Cheng, L. Wang, and X. Xu. "Manufacturing service management in cloud manufacturing: Overview and future research directions". In: *J. Manuf. Sci. Eng.* 137.4 (2015).

[B57] Q. Qi and F. Tao. "A smart manufacturing service system based on edge computing, fog computing, and cloud computing". In: *IEEE Access* 7 (2019), pp. 86769–86777.

[B58] G. S. S. Chalapathi, V. Chamola, A. Vaish, and R. Buyya. "Industrial Internet of Things (IIOT) applications of edge and fog computing: A review and future directions". In: *arXiv preprint arXiv:1912.00595* (2019).

[B59] P. Habibi, S. Baharlooei, M. Farhoudi, S. Kazemian, and S. Khorsandi. "Virtualized SDN-based end-to-end reference architecture for fog networking". In: *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE. 2018, pp. 61–66.

[B60] R. Vilalta, V. López, A. Giorgetti, S. Peng, V. Orsini, L. Velasco, R. Serral-Gracia, D. Morris, S. De Fina, F. Cugini, et al. "TelcoFog: A unified flexible fog and cloud computing architecture for 5G networks". In: *IEEE Communications Magazine* 55.8 (2017), pp. 36–43.

[B61] P. Pop, B. Zarrin, M. Barzegaran, S. Schulte, S. Punnekkat, J. Ruh, and W. Steiner. "The FORA fog computing platform for industrial IoT". In: *Information Systems* 98 (2021), p. 101727.

[B62] G. Davoli, D. Borsatti, D. Tarchi, and W. Cerroni. "FORCH: An Orchestrator for Fog Computing service deployment". In: *2020 IFIP Networking Conference (Networking)*. IEEE. 2020, pp. 677–678.

[B63] *Unibo MEC API Tester*. Accessed Dec. 30, 2020. URL: `https://mecwiki.etsi.org/index.php?title=MEC_Ecosystem`.

[B64] P. Habibi, M. Farhoudi, S. Kazemian, S. Khorsandi, and A. Leon-Garcia. "Fog computing: a comprehensive architectural survey". In: *IEEE Access* 8 (2020), pp. 69105–69133.

[B65] *OPCUA - Specification*. Accessed July 2021. URL: `https://opcfoundation.org/about/opc-technologies/opc-ua/`.

[B66] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville. "Refining Network Intents for Self-driving Networks". In: *SIGCOMM Comput. Commun. Rev.* 48.5 (Jan. 2019), pp. 55–63. URL: `http://doi.acm.org/10.1145/3310165.3310173`.

[B67] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. *Intent-Based Networking - Concepts and Definitions*. Internet-Draft draft-irtf-nmrg-ibn-concepts-definitions-09. Work in Progress. Internet Engineering Task Force, Mar. 2022. 29 pp. URL: `https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-concepts-definitions-09`.

[B68] Q. Sun, W. (LIU, and K. Xie. *An Intent-driven Management Framework [EXPIRED]*. Internet-Draft draft-sun-nmrg-intent-framework-00. Work in Progress. Internet Engineering

Task Force, July 2019. 13 pp. URL: https://datatracker.ietf.org/doc/html/draft-sun-nmrg-intent-framework-00.

[B69] C. Li, O. Havel, A. Olariu, P. Martinez-Julia, J. C. Nobre, and D. Lopez. *Intent Classification*. Internet-Draft draft-irtf-nmrg-ibn-intent-classification-06. Work in Progress. Internet Engineering Task Force, Feb. 2022. 47 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-intent-classification-06.

[B70] A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville. "Refining Network Intents for Self-Driving Networks". In: *Proceedings of the Afternoon Workshop on Self-Driving Networks*. SelfDN 2018. Budapest, Hungary: Association for Computing Machinery, 2018, pp. 15–21. URL: https://doi.org/10.1145/3229584.3229590.

[B71] B. Coecke, M. Sadrzadeh, and S. Clark. *Mathematical Foundations for a Compositional Distributional Model of Meaning*. 2010.

[B72] A. Speranzon, D. I. Spivak, and S. Varadarajan. "Abstraction, Composition and Contracts: A Sheaf Theoretic Approach". In: *CoRR* abs/1802.03080 (2018). URL: http://arxiv.org/abs/1802.03080.

[B73] G. Bakirtzis, C. Vasilakopoulou, and C. H. Fleming. "Compositional Cyber-Physical Systems Modeling". In: *Electronic Proceedings in Theoretical Computer Science* 333 (Feb. 2021), pp. 125–138. URL: http://dx.doi.org/10.4204/EPTCS.333.9.

[B74] M. Bezahaf, E. Davies, C. Rotsos, and N. Race. "To All Intents and Purposes: Towards Flexible Intent Expression". In: *2021 IEEE 7th International Conference on Network Softwarization (NetSoft)*. 2021, pp. 31–37.

[B75] B. Fong and D. I. Spivak. *Seven Sketches in Compositionality: An Invitation to Applied Category Theory*. 2018.

[B76] *Category Hask*. URL: https://wiki.haskell.org/Hask.

[B77] A. Clemm, L. Ciavaglia, L. Z. Granville, and J. Tantsura. *Intent-Based Networking - Concepts and Definitions*. Internet-Draft draft-irtf-nmrg-ibn-concepts-definitions-05. Work in Progress. Internet Engineering Task Force, Sept. 2021. 27 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-concepts-definitions-05.

[B78] J. Hughes. "Why Functional Programming Matters". In: *Computer Journal* 32.2 (1989), pp. 98–107.