ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

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

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

PROGRAMMABILITY AND MANAGEMENT OF SOFTWARE-DEFINED NETWORK INFRASTRUCTURES

Presentata da
GIANLUCA DAVOLI

Supervisore
Prof. WALTER CERRONI

Coordinatore Dottorato
Prof.ssa ALESSANDRA COSTANZO

ESAME FINALE ANNO 2021

One ~~Ring~~ thing to rule them all,
one ~~Ring~~ thing to find them,
one ~~Ring~~ thing to bring them all
and in the ~~darkness~~ network bind them.

Contents

Abstract **i**

1 Introduction **1**
1.1 The evolution of traditional networks 1
1.2 The software revolution 2
1.2.1 Software-defined Networking 4
1.2.2 Network Function Virtualization 6
1.2.3 Cloud computing 7
1.2.4 Service Function Chaining 8
1.2.5 Intent-based Networking 10
1.3 Motivation and contributions 12

2 Service Function Chaining over SDN Domains **16**
2.1 SFC over IoT and Cloud domains 17
2.1.1 Reference network architecture 17
2.1.2 Intent-based northbound interface 20
2.1.3 OpenFlow and Cloud domains 23
2.1.4 Experimental validation 29
2.1.5 Performance evaluation 30
2.1.6 Remarks 32
2.2 Latency-aware SFC over SDN infrastructures 33
2.2.1 Reference architecture and testbed 33
2.2.2 Experimental results 36
2.2.3 Remarks 38

3 Service Function Chaining over non-SDN Domains **39**
3.1 Towards a SFC-aware control plane 39

3.1.1 Service Function Chaining architecture 40
3.1.2 OpenFlow-based NSH control plane 43
3.1.3 Experimental validation 45
3.1.4 Remarks 49
3.2 SFC over IoT, Cloud, Fog and non-SDN tranport domains 50
3.2.1 Remarks 56

4 Resource monitoring in SDN/NFV environments 58
4.1 Monitoring challenges and system architecture 58
4.2 Prototype implementation 61
4.2.1 Testbed based on container technology 64
4.3 Experimental validation 65
4.3.1 Combined network and resource monitoring . . 66
4.3.2 Monitoring-based traffic steering 67
4.3.3 Impact of sFlow parameters 70
4.4 Remarks 73

5 Augmenting software-defined infrastructures 80
5.1 Fate sharing and out-of-channel communication 80
5.2 Sonifying the network 82
5.3 Sonification architecture overview 84
5.4 Management object model and sonification workflow . 86
5.5 Protocol design 87
5.5.1 Connection setup 88
5.5.2 Physical technology adaptation 90
5.5.3 Close 91
5.6 Sonification of network management applications . . . 91
5.6.1 TraceSound 91
5.6.2 Heavy-Hitter Detection 92
5.6.3 DDoS Monitoring 92
5.6.4 Callbacks for network management application programmability 93
5.7 Testbed and implementation 93
5.8 Evaluation 95
5.9 Limitations and open questions 96

5.10 Haptic networking 98
5.11 Remarks 99

6 Flexible service provisioning in Fog scenarios 101
6.1 Motivation and challenges 102
6.2 Fog computing system architecture 103
6.3 A use case 106
6.4 Testbed implementation 108
6.5 Proof-of-concept evaluation 110
6.6 Remarks 111

7 Conclusion 112

Acronyms 114

Bibliography 116

Publications 125

Acknowledgements 128

Abstract

In a landscape where software-based solutions are evermore central in the design, development and deployment of innovative solutions for communication networks, new challenges arise, related to how to best exploit the new solutions made available by technological advancements.

The objective of this Thesis is to consolidate and improve some recent solutions for programmability, management, monitoring and provisioning in software-based infrastructures, as well as to propose new solutions for service deployment, management and monitoring over softwarized domains, along with working implementations, validating each point with punctual experimental validations and performance evaluations.

The treatise starts by introducing the key concepts the research work is based upon, then the main research activities performed during the three years of PhD studies are presented. These include a high-level interface for network programmability over heterogeneous softwarized domains, an implementation of a protocol for service function chaining over non-programmable networks for multi-domain orchestration, a modular system for unified monitoring of softwarized infrastructures, a protocol for the employment of unused channels to augment the capabilities of the softwarized infrastructure, and a XaaS-aware orchestrator designed to operate over Fog computing scenarios.

Chapter 1

Introduction

Software is the centerpiece of modern communication networks, the common thread binding network components together, the leader of innovation in network infrastructures. Advancements in software-based solutions and virtualization technologies have been revolutionizing the telco scene, allowing infrastructures to meet increasingly stringent requirements imposed by the great demands of modern applications and services.

1.1 The evolution of traditional networks

Over the past decades, networks have experienced significant changes in the way services were required of them, due to the development of numerous new applications, also causing a growing demand of ubiquitous connectivity for devices. Different applications have different needs in terms of throughput, delay, jitter, and loss, among other things, all of which can be classified as Quality of Service (QoS) parameters. Satisfactory performance is only a side of the requirements network services need to meet, with other crucial aspects being security, adaptability, reconfigurability, and so on. Generally, deploying a service in modern networks is a complicated task. Over the years, this complication has been decomposed into smaller problems, each one being easier to tackle. This gave way to the development of a variety of specialized network functions, traditionally implemented on

closed, proprietary and expensive hardware, that became known as *middle-boxes*, and met exploding popularity. As early as 2012, the number of middle-boxes deployed over commercial networks was comparable to the number of routers [B1]. This required a sizeable investment from network tenants, both on the devices themselves, and on management and maintenance work needed to keep this valuable equipment operational and tuned. Operating these devices needed specialized personnel with experience in the usage of hardware from the specific manufacturer of the equipment. Moreover, making devices from different vendors interoperate could prove to be a burdensome task, encouraging tenants to buy all of their network supplies from the same manufacturer. Last, but not least, updating a device often meant purchasing new specialized hardware from the same vendor. These factors lead to the so-called *vendor lock-in effect*, causing on the broader scale the phenomenon known as *network ossification*. In other words, the deployment of new features in the network started to be discouraged by economic and technical reasons alike, in contrast with the ever-growing need for continuous renovation and accommodation of new functionalities.

1.2 The software revolution

The inspiration for the the software revolution in the telco world came from the evolution observed decades before in the computing realm, where vertically integrated, closed, proprietary hardware was replaced over time by general purpose hardware offering open programming interfaces, able to run a plethora of modular applications that defined the behavior of the device. In a similar fashion, one can decouple the network mechanisms and functionalities from the hardware realizing them, obtaining, on the one hand, non-specialized hardware that is relatively cheap and reconfigurable, as opposed to the specialized one, and on the other hand, software applications and building blocks that can be combined and modified at will to achieve any desired behavior. These software bits are the driving forces behind the paradigm shifts

towards Software Defined Networking (SDN) and Network Function Virtualization (NFV) that have been revolutionizing communication networks. Employing software solutions running on general purpose hardware leads to a significant reduction both in deployment and operational costs for tenants, as well as an unprecedented level of adaptability to changing requirements.

In performing this decoupling, the separate roles of different abstract *planes* is highlighted. In this context, a *plane* is a classification of network components by the responsibility they have. One of the most widespread classifications is the partitioning of network entities into control, management and data plane components. In short, the decision power resides in the management plane, while the control plane is in charge of the coordination among devices, and the data plane is where the actions on user data are performed. These functionalities coexisted in traditional network equipment. To put this in real world terms, in a traditional router, the management plane includes the processes taking care of the interaction among routers in a network via shared protocols, and taking routing decisions based on the common knowledge, while control plane processes have to translate those decisions into forwarding actions, and communicate them to data plane processes, which are in charge of performing them on the data packets the device receives. By softwarizing the network components, the mutual independence of these planes is accentuated, and each can run as a software entity either on the same device or in a separate location, fully benefiting from the modularity of the approach. The control and management planes are sometimes considered as a single entity, in scenarios where the difference is not critical to the performed application or functionality.

The following sections give an overview over fundamental concepts the rest of the contents of this Thesis relies on, starting from the cardinal paradigms and going through enabling technologies and abstractions that make modern networks more efficient, resilient, and adaptive.

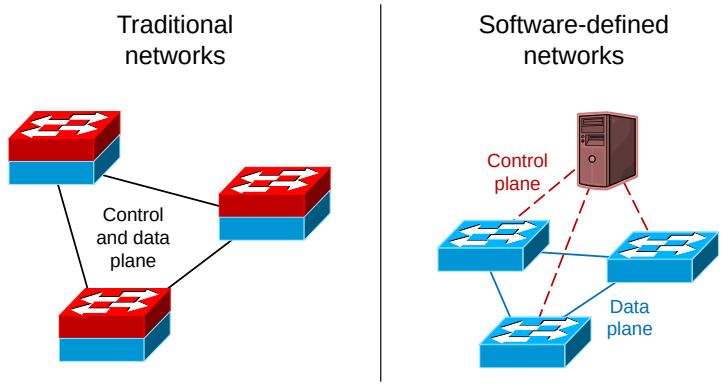


Figure 1.1: Comparison between the traditional and the software-defined network architectures.

1.2.1 Software-defined Networking

In traditional networks, traffic forwarding decisions are taken in a distributed way, thus delegating decision power to packet forwarding devices. These devices make use of standardized protocols to communicate and converge on a common knowledge of the network and related forwarding plans, then perform the proper packet forwarding accordingly. In doing so, these devices are acting both on the control/management plane and in the data plane. On the other hand, in the SDN architecture, all decision-making processes are executed in a logically centralized entity, known as the SDN controller. Network devices only need to be able to forward packets based on the instructions received from the SDN controller, on a logically (and, usually, also physically) separated interface. This way, separation (i.e., *decoupling*) of the control/management plane and the data plane is achieved. The traditional and SDN approaches to networking are compared in Figure 1.1.

The SDN controller, as shown in Figure1.2 exposes two interfaces:

– the Northbound Interface (NBI), which provides APIs to application developers as well as a level of abstraction to hide lower level details, e.g., how the forwarding devices are programmed, from the applications;

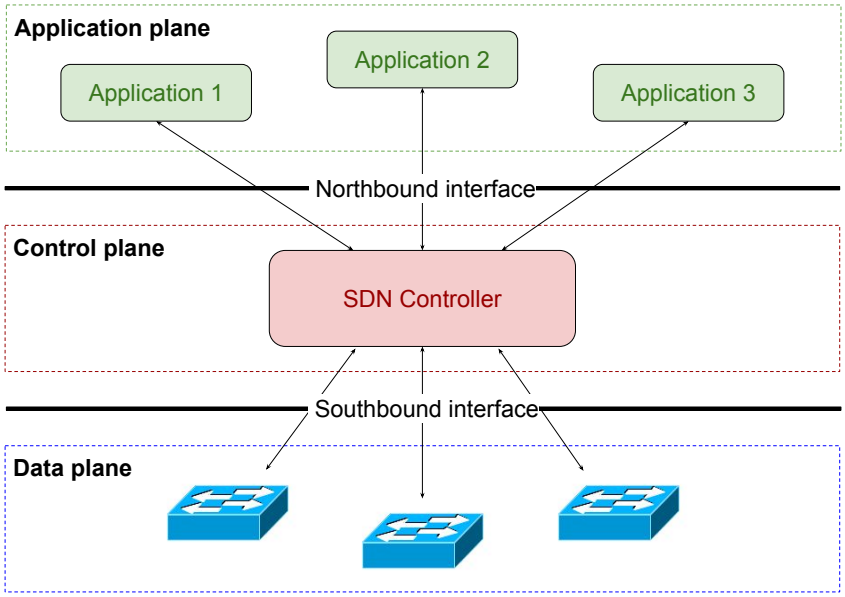


Figure 1.2: Basic SDN architecture.

– the Southbound Interface (SBI), which handles the interaction between controller and data plane devices.

Both interfaces are open to the implementation of protocols that best suit the needs of the applications and/or of the forwarding devices. There is a plethora of solutions for the implementation of both NBI and SBI communications. However, one protocol, OpenFlow [B2], has risen as the *de-facto* standard for SBI-related interactions. This protocol provides an abstraction of the forwarding behavior of underlying data plane devices, characterizing it by means of the so-called *flow table*. Entries to this table are in the form of match-action rules, each associated with a counter for statistics gathering. This way, the traffic can be partitioned into flows, with each flow being defined by features such as source/destination MAC/IP address, network or transport protocol type, transport-layer source/destination port, and so on. Each data plane device will act on the traffic based on the flow classification given by the SDN controller before traffic starts (proactive approach) or while traffic is flowing, sending every unrecognized packet to the SDN controller for classification (reactive approach).

Summarizing, the SDN approach introduces a great simplification

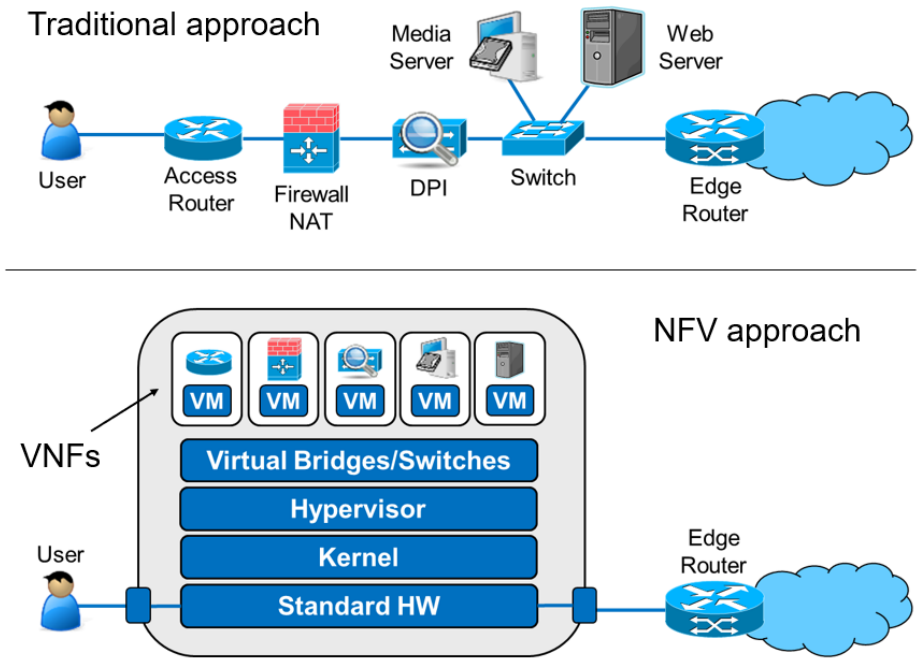


Figure 1.3: Comparison between the traditional and the NFV approaches.

in network management and network policy enforcement, reducing the obstacles to the development of new network protocols and applications. These improvements are brought about by the abstractions provided by the SDN controller to network applications, allowing for the underlying infrastructure to be perceived as an abstracted entity that can be programmed in a flexible and dynamic way.

1.2.2 Network Function Virtualization

In traditional networks, specialized hardware devices (i.e., the *middleboxes* mentioned in Section 1.1) are deployed along the path that packets traverse to get from their source to their intended destination. These devices often consist of specialized and proprietary solutions, which require vendor-specific configuration and management actions. Leveraging the advancements in virtualization technologies, as well as the availability of increasingly powerful general purpose comput-

ing hardware, in the NFV approach network functions are deployed as software components over non-specialized servers, decoupling the functionality of the component from the hardware that implements it. This allows network functionalities to achieve hardware independence, leading to a facilitated and scalable deployment of network functions. The traditional and NFV approaches are compared in Figure 1.3.

Summing up, the NFV approach facilitates flexible and efficient provisioning of network functions, reducing equipment costs and power consumption, and reducing time-to-market by allowing software and hardware to evolve independently.

1.2.3 Cloud computing

The Cloud computing paradigm has established itself during the past decade, reaching outstanding levels of pervasiveness. The Cloud offers on-demand provisioning of computing resources, real-time processing of data, and a increasingly growing number of other services offered over the Internet. The larger the coverage of a cloud service provider, the more likely it is for these services to be deployed in the network over geographical-scale domains, including large data centers as well as relatively tiny edge servers. In this multitude of devices and requirement for different services, the SDN and NFV paradigms can fuel the growth of cloud infrastructures, and in turn be fueled by it. The features offered by these paradigms are perfectly suited for the need of the Cloud. For example, NFV is ideal for the support of hardware resource sharing among multiple concurrent software instances inside of the same physical machine while providing the required mutual isolation, while SDN can greatly facilitate the complex traffic steering required to perform compositions of services across multiple domains.

Services in the Cloud computing domain are offered according to a number of models, one of the most popular ones being *Everything-as-a-Service* (XaaS), whose principal incarnations are Infrastructure-, Platform-, and Software-as-a-Service (IaaS, PaaS, and SaaS, respectively). The objective of these sub-models is to offer an increasing level of abstraction, in order to best suit the needs of the user. Following

the same rationale, a growing variety of resources is being offered in a "aaS" format.

Although the Cloud is designed for efficiency and ubiquity, sometimes responsiveness is not its strong suit, especially in meeting the latency requirements of innovative network services. Fog computing was designed to face this issue.

Fog computing

In scenarios where latency is critical, such as those depicted by the evolution towards new generation networks, Cloud computing may not be responsive enough, due to the possibly remote location of servers (from the perspective of the user). Therefore, the Cloud could benefit from resource relocation, along with computation offloading, allowing for service deployment with reduced latency and improved scalability. This is precisely what Fog computing targets: it helps in bringing services closer to the end user, reducing both service time and load on the Cloud infrastructure. The Fog acts as intermediate layer between users and the Cloud, focusing on the needs of microservices and modular applications, offering replicas of Cloud services, or serving Fog-specific applications. Similarly to its larger counterpart, Fog computing can adopt the XaaS model to help flexibly allocating and managing resources and provide for different needs of the end users. The inherently dynamic nature of the Fog poses an additional set of provisioning and management challenges, that need to be handled properly.

1.2.4 Service Function Chaining

SDN and NFV are mutually independent, as either one can be implemented in network deployments without the other being applied. However, their combination can bring convenient advantages to network tenants, both in operational and economical terms. Therefore, a continuous effort towards the definition and refinement of reference SDN-NFV architectures has to be performed, directed at making the paradigms cooperate to have the network perform complex tasks and

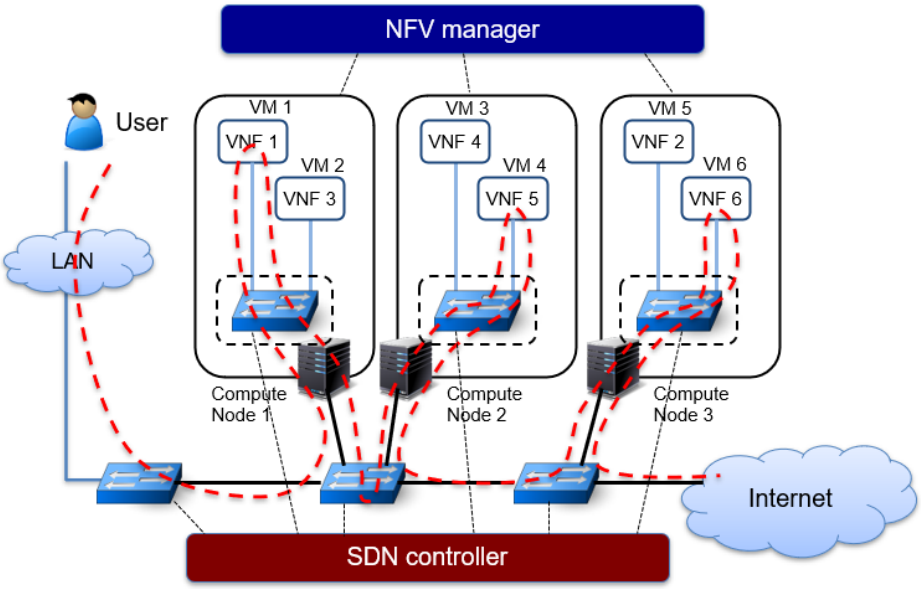


Figure 1.4: An example of a Service Function Chain.

provide consistent distributed services. The latter is one of the main goals of SDN-NFV architectures, and to that end, the heterogeneity of the infrastructures, the high dynamicity of services and the geographical distribution of network functions pose demanding challenges in terms of resource control/management capabilities, adaptive usage of multi-technology resources, and fulfillment of end-to-end latency requirements considering the impact of both processing and network delays [B3, B4]. A depiction of a SFC obtained by joints SDN-NFV efforts is shown in Figure 1.4.

One of the most remarkable offsprings of the cooperation of the SDN and NFV approaches is the redefinition of the way services are deployed over a network. The expression *Service Function Chaining* (SFC) is generally used to describe the deployment of composite services that are obtained from a concatenation, i.e., as a *chain*, of one or more basic services. Consequently, a Service Function Chain is the series of service functions that a packet or flow must traverse. By deploying (virtual) network functions over the network according to the NFV paradigm, and making use of SDN principles to steer the traffic through them, services can be deployed in the network in an evermore

profitable and adaptable way [B5].

SFC makes use of a service-specific overlay that creates the required service topology. Therefore SFC inherently defines a Service Plane, that is an intermediate plane between Application and Control Planes. The Service Plane includes all the processes that allow the infrastructure to provide services to users and maintains state on those services, relying on Control and Management Plane functions to suitably program the Data Plane.

1.2.5 Intent-based Networking

Even in the general reference architecture for management and orchestration proposed in [B6], the NBI of the SDN controller is one of the most critical interfaces, due to the amount of responsibilities it administers. Although a standard NBI definition is continuously under discussion, a commonly accepted approach is to adopt a so-called *intent-based* interface, which allows declaring service outcomes and high-level operational goals rather than specifying detailed networking mechanisms. In other words, according to the Intent-based Networking (IBN) paradigm, the consumer of a service only need to specify *what* it wants the service to perform (i.e., the intent), leaving the decision on *how* to do it completely up to the service provider. The service request should only include information relevant and intrinsically known to the consumer, without any references to the infrastructure of the provider, its operational methodologies, or its constraints. The policies of the consumer (i.e., *why* to perform the service) and those of the provider (i.e., *how* to perform the service) should be completely independent of each other. The information contained in the service request should either be naturally comprehensible to the service provider, or it should become so through a form of lookup the provider can perform, and it may include modifiers that add constraints or details to the desired service. The provider should notify the user on any issue with the request, be it related to the service itself or any of the specific modifiers.

In general, the IBN paradigm can be considered as a new method-

ology to achieve *declarative* network programmability, as opposed to a rather *imperative* approach typically offered by the APIs of previously-existing NFV orchestrators and SDN controllers. A discussion held a few years ago among key players brought to a first common definition of an intent-based interface [B7]. Meanwhile, as part of their efforts to foster the vision of SDN as a network operating system providing generalized network control plane APIs, the Open Networking Foundation (ONF) made a first step toward the standardization of an intent-based northbound interface [B8]. Their approach is based on the consumer-provider paradigm described above. The intent-based interface concept further evolved into the idea of a whole IBN system, defined as a life-cycle management software for networking infrastructures [B9]. More recently, new standardization activities were initiated toward a better understanding of IBN systems. In particular, in the framework of the Internet Research Task Force (IRTF), the Network Management Research Group (NMRG) is currently working on two Internet drafts. One is focused on defining the intent concept, clarifying the related characteristics and functionality, and discussing the main differences with policy-based network management and service models [B10]. The other draft discusses different ways to classify the concept of intent considering the multiple stakeholders involved, and proposes a related taxonomy [B11]. Several other initiatives for the definition and implementation of IBN systems are currently being developed at different levels, in terms of both academic research and industrial product innovation. Also, a number of open-source software projects focusing on the intent approach were started. As a flexible and promising technology, IBN can play a key role in several application scenarios that span over the vast outreach of communication networks.

1.3 Motivation and contributions

The paradigms and technologies described in the previous sections pave the way for massive improvements, but also inevitably introduce new issues and open questions that needs to be addressed. They can mostly be categorized into four areas, each represented by a theme:

- ***programmability***, i.e., how to program the behavior of the network using well-defined interfaces;
- ***management***, i.e., how to define policies that the infrastructure is able enforce in an automated way;
- ***monitoring***, i.e., how to gather information on the utilization of the infrastructure;
- ***provisioning***, i.e., how to allocate the resources of the infrastructure to users.

These themes are often mutually integrated, and addressed simultaneously in the literature.

To begin with, the deployment of SFC over heterogeneous domains poses both programmability and management questions. The 5G initiative encouraged the softwarization of modern network infrastructures such as sensor networks [B12, B13], IoT domains [B14, B15], inter-data center transport networks [B16] and flexible wide area network interconnections [B18, B17]. In this scenario, features such as independence from the underlying forwarding technology and latency awareness play a key role. The problems of SFC orchestration [B19] and physical resource allocation [B20] have been addressed, as well as trade-offs between performance and cost [B21]. On top of that, a comprehensive architecture including the mentioned domains and a standard OpenFlow-based SDN infrastructure is required in order to achieve complete interoperability among these heterogeneous components. Such architecture is the focus of Section 2.1 in Chapter 2, whereas Section 2.2 presents experimental validation results related to a latency-aware SFC deployment system over real-world hardware compliant with 5G directives. Additionally, in Chapter 3, Section 3.1 introduces a transport-independent solution to multi-domain SFC deployment, while Section 3.2 presents an architecture supporting het-

erogeneous forwarding technologies, while still maintaining end-to-end SFC deployment capabilities.

In complex scenarios such as those of next-generation networks, monitoring solutions that are able to cope with the new mechanisms are needed in order to ensure proper verification of network configuration and performance. Some works in the literature propose approaching SDN and NFV entities separately [B22, B23, B24], partly due to the challenges of integrating both worlds in a single monitoring solution [B25]. In order to exploit the join potential of SDN, NFV and Cloud, a monitoring solution should be aware of the different paradigms, independent of the technology, and easy to deploy and maintain. A possible design and implementation of such a solution is presented in Chapter 4.

Concurrently, the management, as well as the monitoring plane, should be protected from data and control plane failures, in order to avoid chain effects and overall disruptive consequences [B26, B27]. However, these entities are typically intertwined, to the point that often their mutual dependence is taken for granted [B28]. Making them independent requires expensive enhancements [B29] or yields solutions that are limited to specific applications [B30, B32, B33, B31, B34]. Based on the work of [B35], a general-purpose sonification protocol is proposed in Chapter 5, to facilitate management and monitoring tasks while preserving the mentioned independence among planes.

As already mentioned, latency plays a key role in next-generation networks, and infrastructures need to keep up with increasingly stringent response time requirements. Resources and services offered in the Cloud are sometimes not responsive enough, partly due to geographical reasons. On the other hand, Cloud infrastructures pioneered service models that offer unprecedented flexibility, allowing almost anything to be offered as-a-Service. A modern service provisioning system should provide highly responsive services that can be deployed as dynamically as possible based on the available resources. A promising location for such a system is the Fog layer, that promises to cut on response time, albeit posing new orchestration challenges [B36]. A

number of solutions for service orchestration in the Fog domain exist, but they are either specific to an application scenario such as the IoT [B37] or they only implement a specific class of service [B38], and do not include practical implementations. In Chapter 6, an original orchestrator system is presented, providing general-purpose architecture and use cases, and a prototype implementation.

This Thesis intends to consolidate the state of the art and propose improvements and solutions for the programmability, management, monitoring, and provisioning of services in softwarized infrastructures. Using these four themes as keys for classification, the publications produced by the efforts behind this Thesis are presented in Figure 1.5. As previously mentioned, these themes often blend with each other, resulting in blurred separation lines between them. However, this can also be a positive aspect in terms of coexistence of integration of multiple solutions in the same environment. Although, for the sake of clarity, the boundaries were kept as defined as possible, there is nothing against integrating, for instance, any of the SFC management solutions with the SDN/NFV monitoring one, or the Fog provisioning one. The implementation of each solution was conducted in the most modular and interoperable way possible given the specific circumstances.

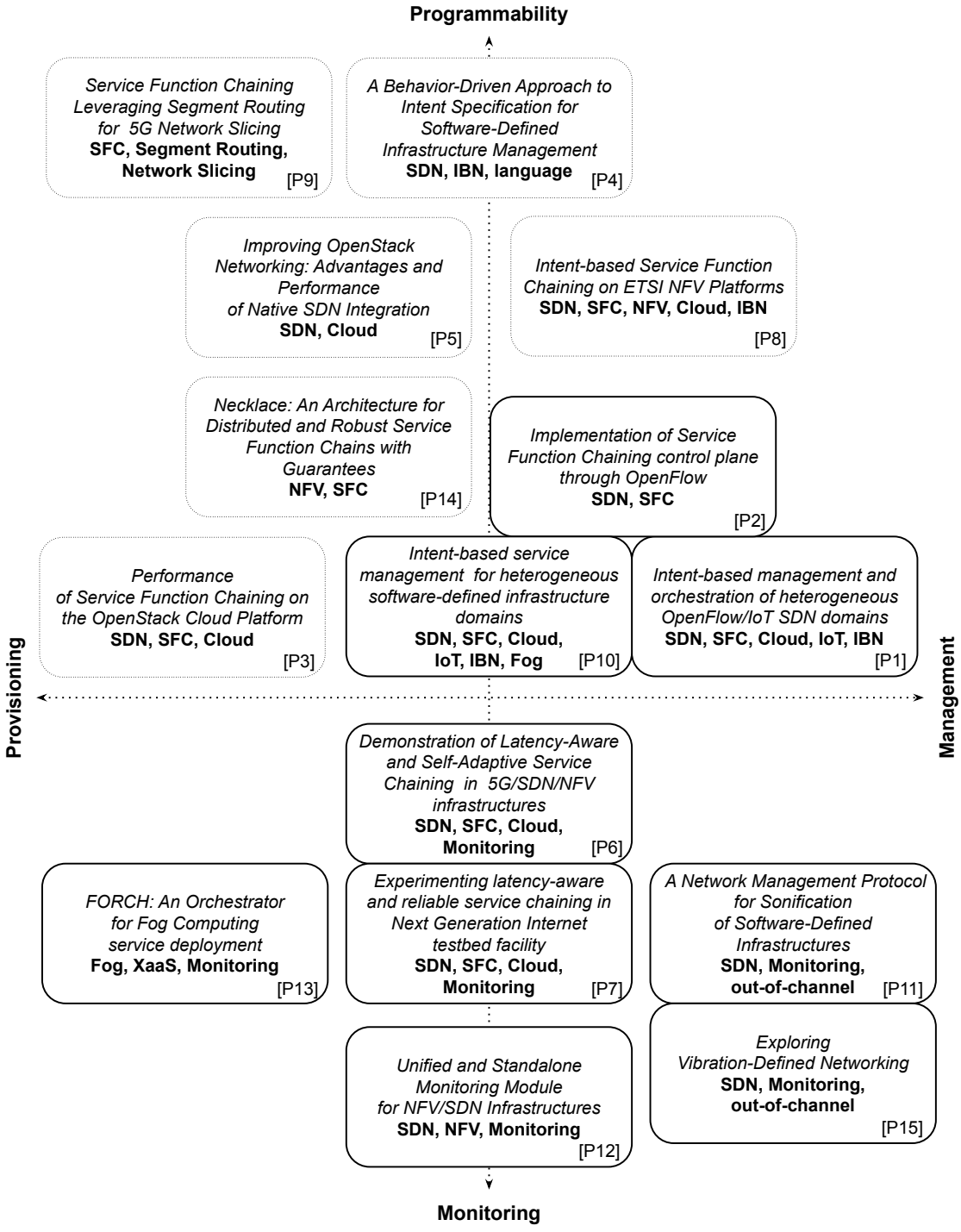


Figure 1.5: Publications arranged by theme; those presented in this thesis are encircled in solid lines, while other complementary works are encircled in dashed lines.

Chapter 2

Service Function Chaining over SDN Domains

End-to-end services provided to customers are typically delivered across different network administrative and/or technological domains. Therefore, guaranteeing certain levels of service has always been a challenging task in multi-domain environments. This is even more complex in case of services spanning multiple SDN/NFV domains, because of the more advanced control features provided by these new paradigms [B39, B40, B41]. A very critical aspect to achieve unified management and orchestration of end-to-end services across multiple domains is the definition of an open, vendor-agnostic, and inter-operable northbound interface (NBI), through which applications are allowed to control the underlying heterogeneous NFV and SDN infrastructures and take advantage of dynamic service chaining. A commonly accepted approach is to make use of an *intent-based*, such as the one introduced in Section 1.2.5, which allows to declare high-level service policies rather than specify detailed networking mechanisms.

This chapter presents an intent-based NBI for end-to-end service management and orchestration across multiple technological domains, considering two use cases. Section 2.1 describes an Internet of Things (IoT) infrastructure deployment and the corresponding cloud-based data collection, processing, and publishing services with QoS differentiation, with relevant experimental validation and performance evalua-

tion, whose results have been published in [P1, P10]. The IoT scenario is meant to be considered as a use case for an architecture that could be applied to different use cases as well, such as that of a software-defined wireless sensor network, without loss of generality. Then, Section 2.2 presents a system for dynamic service chaining orchestration on top of geographically distributed Edge/NFV clouds interconnected through SDN, with results published in [P6, P7]. Although these contributions address different needs and parameters, they are united by the common objective of realizing end-to-end service function chaining across heterogeneous SDN domains.

2.1 SFC over IoT and Cloud domains

In line with the multiple software-defined infrastructure scenarios foreseen by the 5G initiative, the considered IoT domain is inspired by existing work aimed at extending the SDN concepts to wireless sensor networks (WSNs) [B12, B13], and providing convincing motivations for the extension of the SDN paradigm to IoT domains in general. In [B14] and [B15], a solution for separating the data and the control plane of an IoT network is proposed, in order to virtualize the IoT domain, allowing the IoT controller to program the network with the aim of guarantee a specific QoS requested by the consumer. Here, these works are extended by integrating the framework with an OpenFlow-based SDN infrastructure. The proposed architecture is validated via experimental results obtained from a heterogeneous testbed consisting of IoT, OpenFlow and Cloud domains.

2.1.1 Reference network architecture

The considered reference multi-domain SDN/NFV architecture is shown in Figure 2.1. Although the approach to intent-based orchestration could be generalized to any SDN/NFV technology domain, the reference architecture is specialized for the use case considered here, where data collected from sensor and actuator devices of a software-defined IoT domain are dispatched across a wired SDN infrastructure to reach

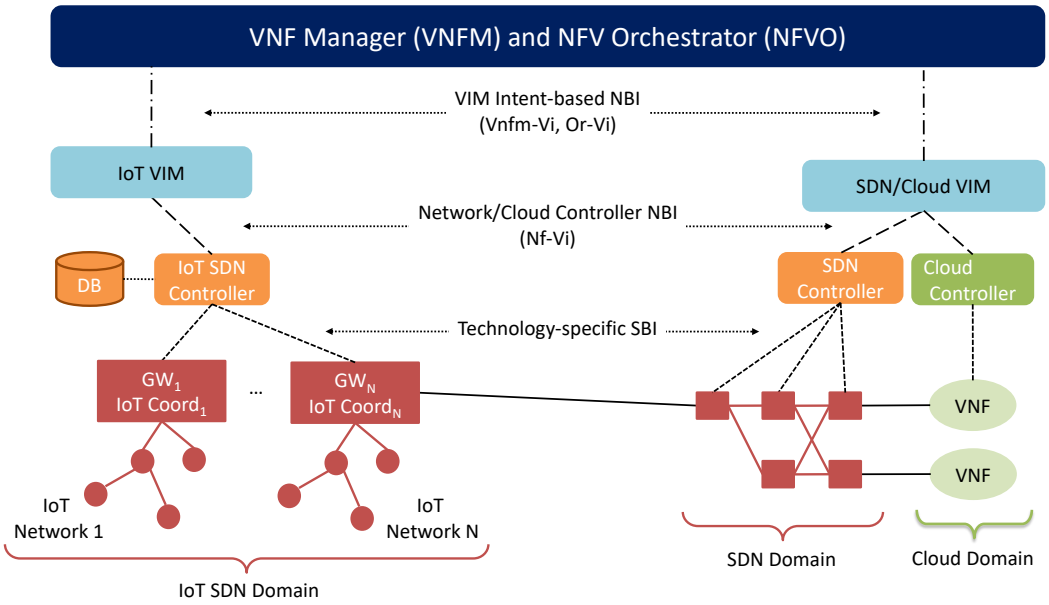


Figure 2.1: Reference multi-domain SDN/NFV architecture, specialized for the use case of IoT data collection and related cloud-based consumption.

a set of suitable consumers, implemented by means of virtual network functions (VNFs) and deployed within a Cloud computing domain. Considering the purpose of the study and the nature of the orchestration features it intends to cover, the reference architecture is inspired by the ETSI NFV specifications, with particular reference to the Management and Orchestration (MANO) framework [B6], although the presented approach considers an end-to-end service perspective. The rationale behind this choice is that, on the one hand, the proposed architecture has the advantage to be consistent with the most relevant NFV standard initiative to date; on the other hand, the architecture itself can be seamlessly extended to include any further SDN/NFV domain and technology as part of the underlying virtualized infrastructure.

Each SDN/NFV domain in Figure 2.1 consists of a technology-specific infrastructure, including:

– data plane components, such as IoT nodes and gateways, SDN

switches, virtual machines running in Cloud computing nodes, physical and virtual interconnecting links; these components provide the network, compute, and storage resources to be orchestrated;
- control plane components, such as SDN and Cloud controllers with related data stores and interfaces; these components are responsible for proper VNF deployment and traffic steering across VNFs and domains;
- management plane components, such as Virtualized Infrastructure Managers (VIMs) specialized for managing resources in the IoT-based SDN infrastructure, the wired SDN infrastructure, and the Cloud infrastructure; based on the available implementations, some of these components could be in charge of multiple domains, as in the case of the SDN/Cloud VIM in Figure 2.1.

The overarching VNF Manager (VNFM) and NFV Orchestrator (NFVO) components are responsible for programming the underlying VIMs and infrastructure controllers in order to implement and maintain the required service chains in a consistent and effective way, for both intra- and inter-domain scenarios. While technology- and domain-specific northbound and southbound interfaces are used inside each domain to efficiently control and manage the relevant components, the design of the overarching VNFM and NFVO should be as technology-agnostic as possible, so that a service chain to be deployed can be specified by a customer using a high-level, intent-based description of the service itself. This would also allow the proposed architecture to be more general and capable of being extended to different SDN technologies and domains.

In order to achieve such generality in the high-level management and orchestration components, the idea is that *the act of decoupling service abstractions from the underlying technology-specific resources should be performed mainly by the VIMs*. Therefore, the NBI offered by the VIMs should be defined as an open and abstracted interface, independent of the specific technology used in the underlying domains, extend the concept of interactions based on intents. This approach

could also allow different administrative domains to expose only service abstractions without disclosing sensitive details related to the underlying infrastructures.

2.1.2 Intent-based northbound interface

As anticipated in Section 1.2.5, the definition of an open, vendor-agnostic, and inter-operable interface can foster improved and standardized procedures for customer service specification to the underlying multi-domain NFV and SDN platforms. In particular, the powerful abstraction level offered by an intent-based NBI allows to specify policies by taking advantage of formalisms that are close to the customer's natural language - as better investigated in [P4]. Therefore, in the reference architecture it is assumed that some kind of intent-based interface is offered to the customer by the overarching VNFM and NFVO components.When a given service request is received, the high-level management and orchestration functions must convert that request into a set of suitable service chains and pass them to the relevant VIMs in charge of the underlying infrastructures and domains involved in the service composition. Then each VIM must coordinate the respective controllers in order to:

- verify availability and location in the Cloud infrastructure of the VNFs required to compose the specified service, instantiating new ones if needed;
- program traffic steering rules in the network infrastructure to deploy a suitable network forwarding path.

The NBI exposed by the VIMs should allow an abstracted yet flexible definition of the service chain, without knowledge of the technology-specific details such as devices, ports, addresses, etc. This means that a request sent to the VIMs should specify not only the sequence, but also the nature of the different VNFs to be traversed, which is strictly related to the service component they implement, as well as other peculiar characteristics of the service itself, such as quality of service (QoS) metrics and thresholds. In particular, the NBI should allow an abstracted representation of the QoS features for the requested service

and the topological characteristics of each VNF to be applied in the service chain.

A possible definition of the VIM NBI is presented here, considering the listed service and function abstractions.

- A QoS feature is defined in qualitative terms relevant to the specified service, e.g. guaranteed bit rate or limited delay.
- A QoS threshold can be specified for the metric of interest, e.g. a minimum bit rate or a maximum delay value.
- A VNF can be terminating or forwarding a given traffic flow. For instance, a deep packet inspection (DPI) function usually terminates a mirrored copy of a given flow, whereas a network address translator (NAT) forwards incoming flows.
- A forwarding VNF can be port-symmetric or port-asymmetric, depending on whether or not it can be traversed by a given traffic flow regardless of which port is used as input or output. For instance, a NAT is port-asymmetric, because it must receive inbound and outbound traffic from a port connected to a public and private network, respectively. A basic IP routing function can be considered port-symmetric, as it forwards packets based on the destination address.
- A VNF can be path-symmetric or path-asymmetric, depending on whether or not it must be traversed by a given flow in both upstream and downstream directions. For instance, an intrusion detection system (IDS) is typically path-symmetric, because it needs to analyze packets in both directions of a given flow. A traffic shaper can be considered path-asymmetric if it must limit only outbound traffic.

In order to implement the aforementioned abstractions, a service function chaining template is defined, adopting the well-known JSON format. This template should be coupled with other deployment templates defined by the ETSI MANO specifications in order to complete service provisioning. However, in this work the focus is only on the service function chaining aspects of the NBI. A service chain is therefore defined as

```
{
    "src": "node_value",
    "dst": "node_value",
    "qos": "qos_type",
    "qos-thr": "qos_value",
    "vnfList": [vnf],
    "dupList": [dup]
}
```

where: `src` and `dst` represent the endpoint nodes of the service chain, either global or limited to a given VIM domain; `node_value` is a text string that contains a high-level unique identifier of a node known to both orchestrator and VIMs, e.g. by means of some form of mapping mechanism as defined in [B42]; `qos` represents the QoS feature to be provided with the service chain; `qos_type` is a text string that contains a high-level unique identifier of a QoS metric known to both orchestrator and VIM; `qos-thr` represents the QoS threshold to be applied to the specified metric; `qos_value` is the actual value assigned to the threshold; `vnfList` is the ordered list of VNFs to be traversed according to the specified service; `dupList` is the list of VNFs towards which the traffic flow must be duplicated.

Each VNF is described in terms of its topological abstractions with the following template:

```
vnf ::= {
    "name": "node_value",
    "terminal": "bool_value",
    "port_sym": "bool_value",
    "path_sym": "bool_value"
} | ϵ
```

where `bool_value` is a text string representing either a Boolean or a null value, and the ϵ symbol indicates the possibility that `vnf` is an empty element. Considering that some network functions (e.g., DPI, IDS) require traffic flows to be mirrored, the (possibly empty) list of VNFs towards which the traffic flow must be duplicated is specified with the following template:

```
dup ::= {"name": "node_value"} | ϵ
```

The NBI offered by VIMs can be implemented through the mechanisms of a REST API, and should provide the following methods:

- define a new service chain;
- update an existing service chain;
- delete an existing service chain.

These actions are essentially in line with the operations foreseen by the ETSI MANO specifications, with reference to the interface between NFVO and VIM. It is worth highlighting that the NBI description given above is indeed based on the concept of intent. QoS metric, VNFs and service chains are specified in a high-level, policy-oriented format without any knowledge of the technology-specific details. A non-intent-based description of a service chain, e.g. using the OpenFlow expressiveness to steer traffic flows and compose the network forwarding path, would require the customer to specify multiple flow rules in each forwarding device for each traffic direction, involving technology-dependent details such as IP and MAC addresses, device identifiers and port numbers. The NBI defined above is used to specify an IoT data gathering service crossing two different SDN domains and an NFV chain, as per the architecture in Figure 2.1. For the use case considered here, the high-level QoS features offered by the SDN/NFV platform include "delay sensitive" and "loss sensitive" services, with the possibility to specify a threshold for the relevant metric. Although the above intent-based NBI definition is common to all VIMs considered in this use case, the orchestrator must specify different content for each VIM depending on the specific resources to be programmed and the specific segment of the service chain to be deployed in each domain.

2.1.3 OpenFlow and Cloud domains

Assume that both the wired SDN domain and the Cloud computing domain depicted in Figure 2.1 are managed by a single SDN/-Cloud VIM. This assumption is necessary in order to only focus on the validation of the abstractions offered by the proposed architec-

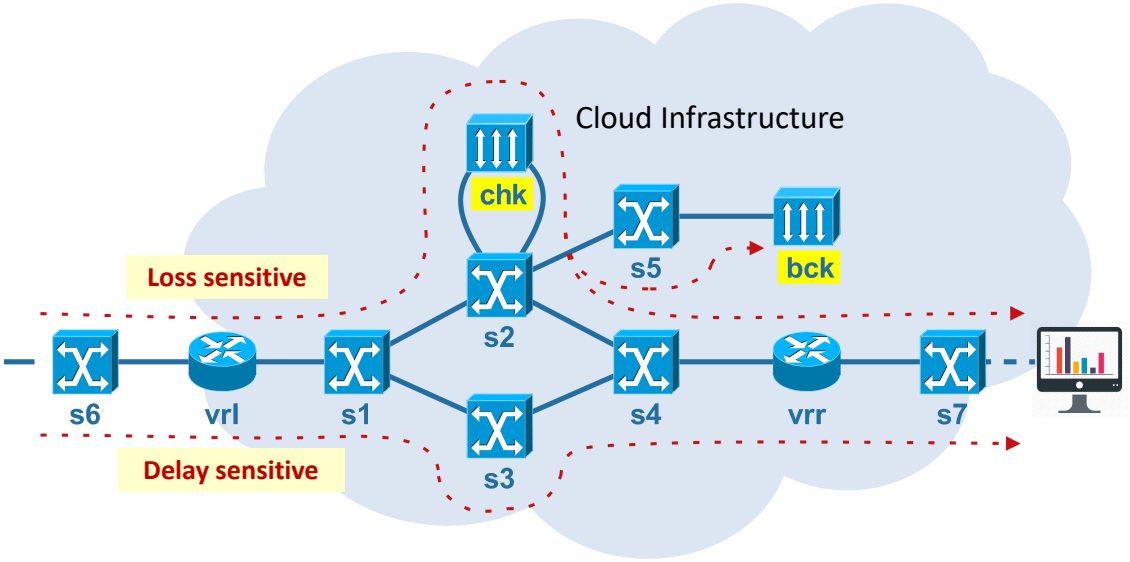


Figure 2.2: Data plane topology of the OpenFlow and Cloud domains considered for the use case.

ture without introducing impediments due to communication over geographical domains, a point that will be thoroughly addressed in Section 3. The data plane topology for the considered use case is shown in Figure 2.2. An OpenFlow-based SDN infrastructure is assumed to be in charge also of the connectivity within the Cloud domain, thus providing programmable traffic steering functionality to VNF chains. All the switches included in the topology (`s1`, `s2`, ..., `s7`) are OpenFlow-enabled devices and are governed by an SDN controller (e.g., ONOS [B43]), whereas the computing infrastructure is managed through a Cloud platform (e.g., OpenStack [B44]). Switch `s6` is an edge device connecting the IoT gateways in the IoT SDN domain to the Cloud network. Router `vrl` is the (virtual) edge router of the (virtual) tenant network responsible for the connectivity within the Cloud domain of the requested IoT data collection service. Switches `s1` to `s5` are either physical or virtual switches used by the tenant network for VNF connectivity. Two VNFs are deployed in the cloud: `chk` performs integrity and sanity check on the collected data for im-

proved reliability, whereas `bck` is used to store backup copies of the collected data. Router `vrr` is the (virtual) edge router of the (possibly different) tenant responsible for the IoT data collection, processing, and publishing services. Switch `s7` is a (virtual) switch in the latter tenant's network, providing layer-2 connectivity to the server `ServP` where collected data are processed and published.

According to the QoS features of the use case considered here, the connectivity service offers two different paths in the OpenFlow domain. One path is characterized by minimum latency, where switches are configured with small buffers being continuously monitored by the SDN controller for possible congestion, and such that no VNF processing is performed, which could introduce additional delays. The other path is dedicated to highly reliable traffic flows, where switches have large buffers to reduce losses, and data are processed by `chk` and duplicated at switch `s2` in order to be stored in `bck`. Therefore, depending on the QoS feature requested by the customer, the high level management and orchestration functions can specify two different service chains. In a real-world scenario, the monitoring operated by the SDN controller would need to be tuned according to the capability of the infrastructure in terms of computation power and link capacity. To this end, the mentioned buffers can be tuned until a working trade-off between monitoring accuracy and useful granularity is reached. Assuming that, based on the interaction between the orchestrator and the IoT VIM, incoming data will be collected from IoT network `k` and then forwarded to `ServP`, according to the JSON format specified in Section 2.1.2 the intent-based request to the SDN/Cloud VIM NBI could be

```
{
    "src": "IoT-GW[k]",
    "dst": "ServP",
    "qos": "Max delay",
    "qos-thr": "10 ms",
    "vnfList": "null",
    "dupList": "null"
}
```

for the low latency QoS feature, or

```
{
    "src": "IoT-GW[k]",
    "dst": "ServP",
    "qos": "Reliability",
    "qos-thr": "99%",
    "vnfList": [chk, bck]
    "dupList": [bck]
}
```

with

```
chk ::= {
    "name": "chk",
    "terminal": "false",
    "port_sym": "true",
    "path_sym": "false"
}
```

and

```
bck ::= {
    "name": "bck",
    "terminal": "true",
    "port_sym": "null",
    "path_sym": "false"
}
```

for the loss sensitive QoS feature. The SDN controller must implement a data plane monitoring service to make sure that, in the former case, the delay sensitive path guarantees the requested maximum delay of 10 ms, whereas in the latter case the VNFs inserted in the service chain and the loss sensitive path ensure the required 99% accuracy.

The VIM for the SDN/Cloud domains was developed as an application running on top of the ONOS platform. It is important to remark that ONOS already provides a built-in, intent-based NBI that can be used to program the SDN domain and deploy the required network forwarding paths. However, in order to specify ONOS intents, some knowledge of the specific data-plane technical details is

required, whereas the aim here is to expose only high-level abstractions to the orchestrator. Therefore, one of the main functions of the VIM is to implement new, more general and abstracted intents that can be expressed according to the NBI specification given above. The VIM then takes advantage of the network topology features offered by the SDN/Cloud controllers in order to discover VNF location in the Cloud and relevant connectivity details, and eventually it is able to compose native ONOS intents and build more complex network forwarding paths. The VIM can be instantiated as an ONOS service called *ChainService*, which provides the capability of dynamically handling the VNF chains through the abstracted NBI defined in Section 2.1.2. To achieve extensibility and modularity, the implementation of *ChainService* is delegated to a module called *ChainManager*, which is in charge of executing all the required steps to translate the high-level service specifications into ONOS-native intents. The input to *ChainManager* can be given through either the ONOS command line interface (CLI) or a REST API. The latter is preferable, because it allows remote applications to use standard protocols (e.g., HTTP) to access resources and configure services. In this implementation, the REST API provides the following service endpoints:

```
POST /chaining/{action}/{direction}
DELETE /chaining/flush
```

In the former endpoint, the `action` variable indicates the operation that the orchestrator intends to perform on a specified service chain (`add`, `update`, or `delete`), while the `direction` variable (`forth`, `back`, or `both`) is used in case of an `update`, and it defines whether the modified chain specification refers to the existing forwarding path from `src` to `dst`, the opposite way, or both directions. The basic operations of this endpoint are specified in the following list.

- If the `add` action is given, this will result in defining a new service chain, based on the JSON specification included in the message body. This means that a forwarding path will be created for traffic flowing from `src` to `dst` and another one in the opposite direction. Note that the two paths are not necessarily symmet-

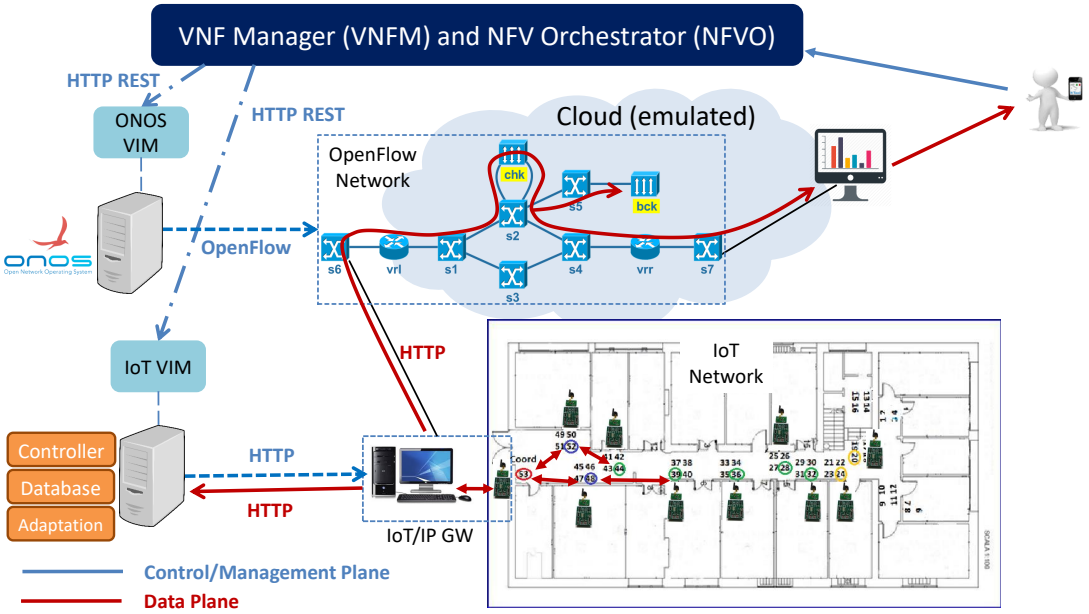


Figure 2.3: The NFV/SDN testbed setup developed to demonstrate multi-domain SDN/NFV management and orchestration.

ric, based on the topological abstractions defined by the NBI.

– If the `update` action is given, then the direction is taken into account and the forward path, backward path, or both paths of the specified existing service chain are changed. In fact, a user may be interested in changing only a segment of the forwarding path and only in one direction, to reduce the control plane latency and limiting the impact that a path change can have on the existing traffic flows.
– If the `delete` action is given, then both forwarding paths of the specified existing service chain are removed. *ChainService* provides also the flush operation through another endpoint, thus offering the possibility of deleting in a single step the forwarding paths of all the service chains previously created.

2.1.4 Experimental validation

As a demonstration of the feasibility of the proposed multi-domain SDN/NFV management and orchestration solution, a testbed was developed, implementing the reference architecture of the cloud-based IoT data collection service with quality differentiation illustrated in Figure 2.1. The complete testbed setup is shown in Figure 2.3. The customer on the top-right corner requests the service to the high-level management and orchestration functions, specifying the desired QoS feature. The orchestrator then forwards the request to the VIM REST NBIs of the relevant domains using the JSON format described in the previous sections. Each VIM performs the operations required in the respective domain and programs the underlying controllers according to the requested service and QoS feature. Data generated by the IoT devices is sent by the relevant gateway via HTTP POST to the collecting/processing/publishing server in the cloud, where the customer can retrieve it (the case of loss sensitive QoS feature is shown in the figure).

In the testbed, the OpenFlow SDN domain and the Cloud domain were emulated using Mininet [B45] running in a virtual machine. The data plane topology in Figure 2.2 was built with a customized Mininet script specifying the required OpenFlow switches, as well as routers and VNFs as separated network namespaces. Additional virtual machines were instantiated to deploy the data collection/processing server and the ONOS platform components. In order to provide the two paths with different latency, `chk` was configured to introduce an additional random delay uniformly distributed between 25 and 35 ms, with 25% correlation between consecutive samples.

Regarding the IoT domain, an IoT network was setup using the EuWIn platform and, in particular, the flexible topology testbed (Flextop) facility [B46]. The testbed is composed of a number of SDN-enable wireless sensor devices located in boxes hung on the walls of a corridor at the University of Bologna. The map with the corresponding identifiers of nodes is shown in the bottom-right part of Figure 2.3.

Due to the technological heterogeneity of the testbed, the perfor-

mances of each domain were evaluated separately. However, the results obtained allow to reasonably infer the characteristics of the end-to-end service. The results concerning the SDN/Cloud domain are reported in Section 2.1.5, while for further details on the IoT testbed and relevant results, refer to [P1, P10].

2.1.5 Performance evaluation

The performance evaluation within the emulated Cloud network is referred to the case where the customer requests the service by choosing between two traffic classes - the previously mentioned "delay sensitive" and "loss sensitive" services - according to the QoS features offered by the OpenFlow SDN domain. One-way latency in the emulated Cloud network was measured by comparing timestamps of each packet captured at switches `s6` and `s7`. The capture was performed in the server hosting the Mininet virtual machine, so the same reference clock was used for the sake of accuracy. The measurements were made by averaging over 10,000 requests. Results are reported in Table 2.1, in terms of average and standard deviation of the data plane one-way latency.

QoS feature	**Average latency**	**St. dev.**
Delay sensitive	0.3 ms	0.28 ms
Loss sensitive	31.7 ms	2.41 ms

Table 2.1: Average and standard deviation of data plane (DP) one-way latency computed at the emulated Cloud network.

The numbers prove the correct behavior of the OpenFlow domain with respect to the requested QoS feature: very limited delays were measured in the delay sensitive case, whereas in the loss sensitive case no packets were lost and `bck` successfully stored a copy of the entire data set transmitted by the IoT GW.

The last evaluation focused on the the NBI response time at the VIM implemented in ONOS, i.e., the time required by the VIM to process a JSON service chain specification. To assess the scalability of the NBI, an increasing number of requests (from 5 to 200) was

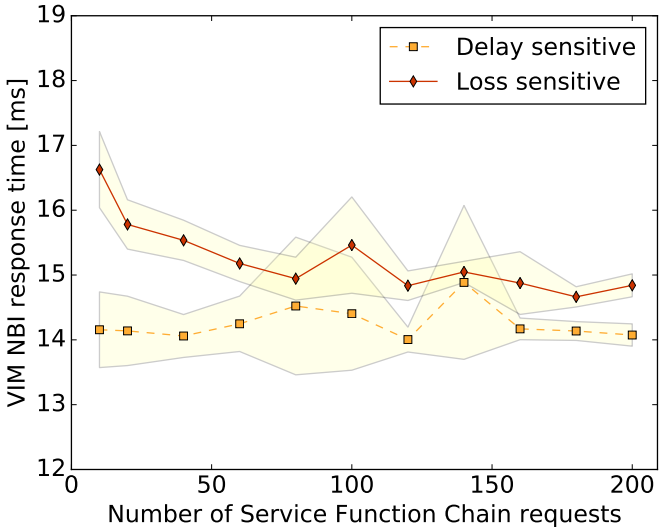


Figure 2.4: Average NBI response time and 95% confidence interval at the SDN/Cloud VIM with increasing number of service chain requests. This response time is only marginally susceptible to the difference of service.

generated and sent in a batch to the VIM. Each measured response time was obtained as an average over 20 runs with the same number of requests. Figure 2.4 shows the average NBI response time with 95% confidence intervals. The numbers show that the VIM is very responsive, in the order of tens of milliseconds. The setup of loss sensitive service chains takes slightly longer than the delay sensitive ones because of the relatively more complex service chain to be processed.

As previously mentioned, the VIM for the data center and Cloud domains was developed as an application running on top of the ONOS platform and taking advantage of its connectivity-oriented, intent-based NBI. This means that the operations performed by the VIM (i.e., parsing and processing a request received through its service-oriented, intent-based NBI; connecting to the ONOS NBI; programming the relevant intents) are decoupled from the ONOS-based operations (i.e., installing the requested intents in its core modules and translating them into actual OpenFlow rules to be added to the con-

trolled SDN switches). Therefore, the response time reported in Figure 2.4 does not include the time needed by ONOS to complete the flow rule setup. Since the latter depends on the specific SDN control technology adopted, it has been kept separate from the VIM response time.

However, for the sake of completeness, Table 2.2 reports the time needed by ONOS to execute the intent and flow installation for the two QoS classes under different virtual machine resource configurations in terms of number of CPUs. The results, obtained from the average over 100 SFC requests, show how the ONOS response time decreases when more resources are dedicated to it, keeping the network programming time in the order of a couple of seconds. This also proves the correct behavior of the data center and Cloud domain control plane from the functional point of view. A complete functional validation of the proposed NBI and the underlying control plane was performed on a very similar experimental environment in [B47].

No. of vCPUs	**Delay sensitive**	**Loss sensitive**
2	3321.4 ms	3468.9 ms
4	2071.7 ms	2984.7 ms
8	1617.9 ms	2866.6 ms

Table 2.2: Average response time of the ONOS controller to execute the intent and flow installation in the data center SDN network.

2.1.6 Remarks

The reported validation results demonstrate that the proposed NBI can contribute to a viable solution for effective service deployment under constraints in real environments, such as the heterogeneous OpenFlow/IoT SDN testbed employed here, extended in Section 3.2, where more comprehensive conclusive remarks are presented, too.

2.2 Latency-aware SFC over SDN infrastructures

This section presents experiments on a latency-aware dynamic service chaining orchestration, performed on top of the Fed4FIRE infrastructure provided within the Fed4FIRE+ Horizon 2020 Project, which offers a federation of open, accessible and high-available Next Generation Internet (NGI) testbeds to support a wide variety of different research and innovation activities, including 5G-related experiments [B48].

The presented orchestration system supports latency-aware and reliable network service chaining on end-to-end basis, including dynamic virtual function selection and intent-based traffic steering control functionalities through heterogeneous SDN control systems.

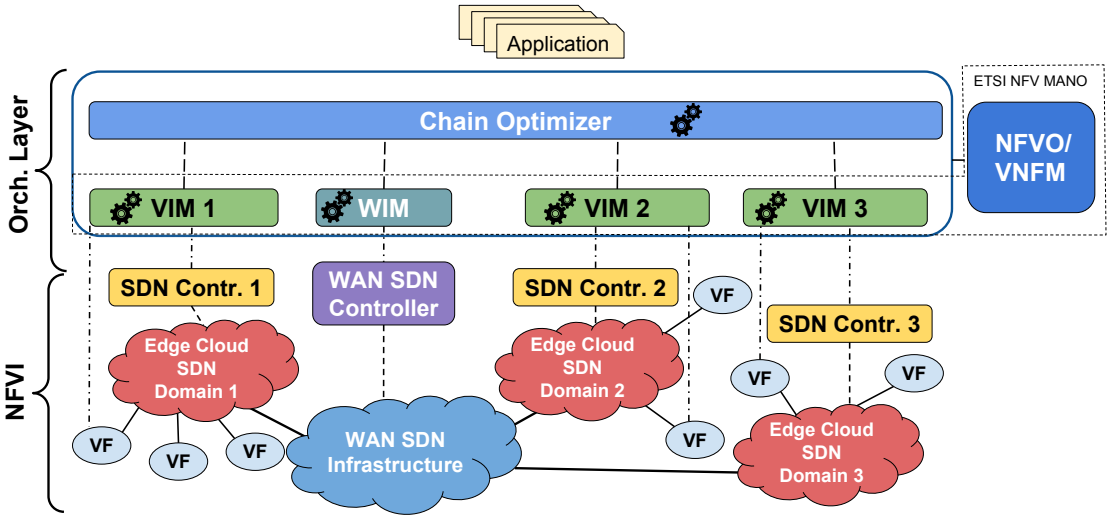
In this section, a portion of the results collected in [P6, P7] is presented, specifically the one related to the validation and performance evaluation of the proposed system in deploying service chains that meet predetermined latency requirements.

2.2.1 Reference architecture and testbed

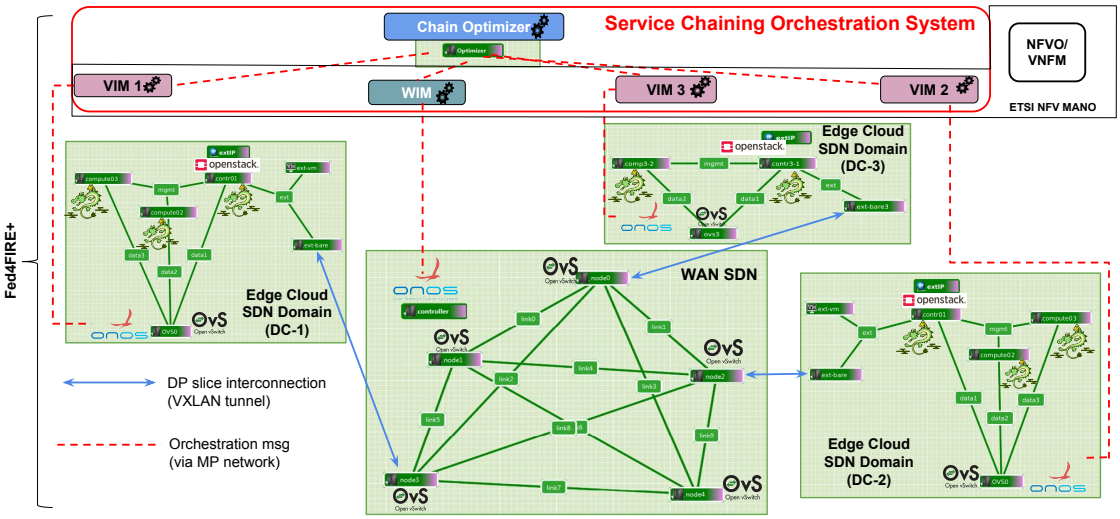
The reference architecture is shown in Figure 2.5a, and the related deployment of the service chaining orchestration system with the setup on top of the Fed4FIRE+ experimentation platform is shown in Figure 2.5b.

The ***Chain Optimizer*** is a service chaining engine running an optimization algorithm that, upon a service chain request, selects VNF instances available from different data centers (DCs) to minimize an estimated end-to-end latency calculated considering both VNF processing delays and inter-DC network delays information.

The ***SDN WAN slice*** includes a SDN network topology consisting of of five physical nodes running virtual SDN switches controlled by an instance of ONOS, hosted on a dedicated node. The *WAN Infrastructure Manager (WIM) Orchestrator* implements the orchestration logic for the WAN SDN domain slice on top of the ONOS controller,



(a) Reference architecture, including all management and control plane components.



(b) Testbed deployment, where multiple experiment slices are interconnected at both data plane (blue solid lines) and orchestration plane (red dashed lines) levels.

Figure 2.5: Orchestration system reference architecture and deployment on the Fed4FIRE+ platform.

exposing the programmable provision of service chain paths across the WAN. In line with [B49], the WIM orchestrator also offers reliable service chains by adapting (i.e., redirecting) service paths, or a segment thereof, to recover from network congestions events detected by periodically collecting statistics from the SDN controller and deriving up-to-date switch link throughput data. Finally, the WIM is responsible for the collection of network latency information (i.e., inter-DC delays) that are made available to the Chain Optimizer for computing a minimum-latency service graph.

The three ***SDN DC slices*** host small Edge Cloud deployments based on OpenStack. Each DC slice includes two or three compute nodes, where virtual machine instances are deployed over a QEMU-KVM hypervisor. All OpenStack nodes are connected to another physical node running an instance of OvS, representing the data plane SDN infrastructure of the DC, which is controlled by an instance of ONOS running locally. The same physical node hosts also the *Virtual Infrastructure Manager (VIM) Orchestrator*, which implements an SDN-enabled DC/Cloud domain orchestration logic providing advanced network management capabilities in Cloud computing environments. The VIM orchestrator exposes an intent-based northbound REST interface that allows to specify a service chain in the form of intents. This makes it suitable to manage different DC domains in a multi-technology environment, e.g., leveraging different SDN controllers. The VIM orchestrator is also capable of dynamically applying changes to an existing service chain without having to delete and re-deploy it from scratch. This allows to dynamically adapt service chains to the current context of users or services (e.g., location of users in a mobility scenario) or to varying needs of the service provider (e.g., resource management policy), and, ultimately, to avoid or prevent SLA violations. Furthermore, the REST API provided by the VIM orchestrator allows the Chain Optimizer to collect information about the currently deployed VNFs and their estimated processing latency, computed based on the current workload. The established DC slices and the WAN slice interact at the data plane level by ex-

changing packet data traffic by means of VXLAN tunnels, and at the orchestration plane level by exchanging control messages between Chain Optimizer, WIM and VIM orchestrators.

2.2.2 Experimental results

The correct operations of the orchestration system were validated by generating `create` and `delete` service chain requests to the Chain Optimizer, with different lengths and requirements in terms of bandwidth and maximum latency. The Chain Optimizer handles each request, computes a latency-optimized solution and sends the corresponding forwarding instructions to the relevant VIM and WIM orchestrators through their respective northbound interfaces. Then, each VIM/WIM interacts with the SDN controller in its domain in order to setup the relevant flow entries.

After the switches are configured and the chain is correctly established, data traffic is injected across the VNF instances implementing the chain (e.g., by using `iperf` to generate traffic at 1 Mbit/s).

Figure 2.6 shows the sequential time diagram of the throughput measured at the VNF instances involved in the deployment of the following service chain sequence:

i) VNF-1 $\rightarrow$ VNF-7 $\rightarrow$ VNF-9
ii) VNF-1 $\rightarrow$ VNF-9
iii) VNF-1

At time $t = 0$ the three chains have already been successfully deployed. According to the initial placement, instances of VNF-1 and VNF-7 are deployed in DC-1, whereas instances of VNF-9 are deployed in DC-2. At $t = 9$s traffic starts flowing through instances involved in the first chain, i.e., VNF-1, VNF-7 and VNF-9 (throughput equal to 1 Mbit/s). When traffic is sent through the second chain, at $t = 40$s a second flow is measured at VNF-1 and VNF-9 instances (throughput equal to 2 Mbit/s). Finally, when the third chain is loaded with traffic, throughput equal to 3 Mbit/s is measured at VNF-1 instance at $t = 70$s. At the end of the experiment ($t = 100$s), the measured throughput drops to zero due to the deletion of the three service chains.

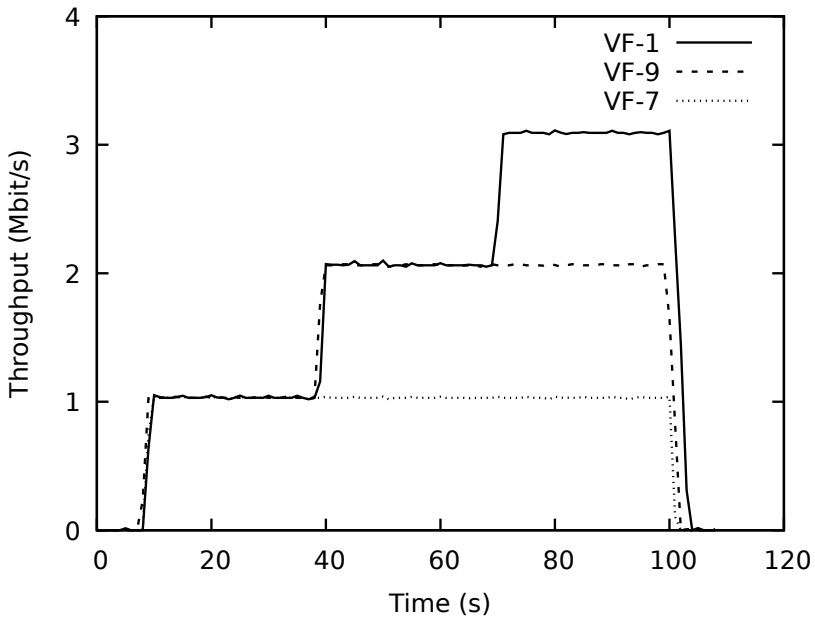


Figure 2.6: Sequential time diagram of the throughput measured at VNF instances involved in a service chain deployment sequence.

This demonstrates the correct deployment and deletion of the service chains across the involved domains.

Table 2.3 compares the end-to-end latency obtained as the sum of retrieved VNFs processing latency and the inter-DC latency measurements, used by the Chain Optimizer to compute the service chain path (i.e., end-to-end latency at Chain Optimizer), with the end-to-end latency actually experienced by data while flowing in the established service chains (i.e., end-to-end latency at established chains). For each run, a sequence of 10 service chain requests is sent for every given chain length. The actual measured values and the ones estimated by the CO are fairly close, proving the robustness of the latency-awareness feature and of the computation process of the orchestration system, that allows for an efficient selection of the DCs and of the VNFs instances.

Chain Length	**Latency at Chain Opt. [s]**	**Latency at est. chains [s]**
2	56.24	76.7
3	72.38	77.2
4	103.86	112

Table 2.3: End-to-end latency as predicted by the Chain Optimizer vs. as measured after establishing the chains.

2.2.3 Remarks

The end-to-end orchestration system presented in this this section was validated using a realistic and composite SDN/NFV deployment, realizing latency-aware and reliable service chaining over geographically distributed SDN-based Cloud DCs interconnected through a SDN WAN. Possible improvements include the integration of this orchestration functionalities with a complete implementation of a MANO orchestrator, as well as perform extensive evaluation in comparing different VNF selection approaches.

Chapter 3

Service Function Chaining over non-SDN Domains

In this chapter, a possible implementation of SFC-aware control plane is proposed, inspired by the concepts discussed in [B50], starting from the discussion of architectural aspects, then moving on to the introduction of the proposed implementation, and its application to a known scenario. In Section 3.1.1, the SFC architecture is discussed, and the proposed implementation of a NSH-aware control plane is presented in Section 3.1.2 and validated in Section 3.1.3. In Section 3.2 the proposed approach is integrated in the architecture described in Section 2.1 and the related testbed, extending it, and achieving multi-domain orchestration of SFC over both SDN and non-SDN domains. These contributions are linked by the common goal of making the mentioned SFC orchestration feasible and to prove that the approach can be applied to use cases supported by different heterogeneous technologies.

3.1 Towards a SFC-aware control plane

Several aspects of SFC are currently being investigated by the research community. SFC Orchestrators designed to deploy SFCs as well as control their activity and make adjustments are introduced in [B19]. The problem of allocating physical resources to data plane

components of a SFC is addressed in [B20], while a solution for the trade-off between optimized performances and resource cost in SFC deployments is presented in [B21].

A very important problem in the implementation of the SFC Orchestrator arises when the chain spans several network domains with non homogeneous forwarding technologies. This problem was addressed by the Internet Engineering Task Force (IETF) in [B51], where it is suggested that the service-specific overlay can be obtained by applying packet encapsulation. One option being considered by IETF is the *Network Service Header* (NSH) [B52], which intends to provide a flexible, dynamic, and transport-independent SFC solution for the data plane. The NSH standard focuses on data plane aspects only, and very little has been said about a possible SFC control plane solution.

3.1.1 Service Function Chaining architecture

The SFC architecture [B51] introduces some important concepts that are briefly mentioned in the following. The *Service Function Path* (SFP) is a specification of the path to be followed by packets assigned to a certain SFC. It is an abstraction of the sequence of nodes the packets requiring a given service will traverse. On the other hand, the *SFC encapsulation* (SFC-En) always provides SFP identification and can optionally provide further information. It is used by the SFC-aware functions to realize the Service Plane functionalities, but it is not used for packet forwarding through the underlying network topology. Carrying the SFC-encapsulated traffic is the task of the chosen network transport protocol.

The main components of the SFC Service Plane are:

- *SFC Classifiers* (SFC-Cl), which classify the incoming traffic based on predefined policies, in order for the flow to be steered through the required set of network service functions; the main task for the SFC-Cl is to add the SFC-En, which is then removed by the last node in the SFP, or by a SFC-aware function that consumes the packet;
- *Service Functions* (SF), which are the basic elements of a chain,

and are responsible for a specific treatment of received packets; they can act at different levels of the protocol stack, and they can be implemented either as virtual elements hosted by a server, or as physical equipment with specialized hardware; a SF can be either SFC-aware (i.e., able to act on SFC-encapsulated packets) or SFC-unaware (i.e., it must receive only packets without SFC encapsulation);

- *Service Function Forwarders* (SFF), which are responsible for forwarding traffic to one or more connected SFs according to information carried in the SFC-En; they can also terminate the SFP;
- *SFC Proxies* (SFC-Pr), which remove and insert SFC-En on behalf of SFC-unaware SFs, before and after their action, respectively.

The reference architecture of the SFC Control Plane (SFC-CP) described in [B50] defines the following interfaces to communicate with Data Plane components:

- interface *C1*, between SFC-CP and SFC-Cl, used to manage SFC classification rules in classifiers;
- interface *C2*, between SFC-CP and SFF, used for exchanging required information for SFC forwarding decision-making, collect state information on SFPs, etc.;
- interface *C3*, between SFC-CP and SFC-aware SF, used, for example, to collect output information resulting from the processing of packets in the SF;
- interface *C4*, between SFC-CP and SFC Proxies, used to communicate SFC instructions and to retrieve state information.

The deployment of SFCs must take into account complex aspects that must be handled carefully, as reported in [B53]. Such aspects include topological dependence, consistent ordering of SFs, and dynamic SFC classification. Moreover, end-to-end SFCs are typically deployed across multiple network administrative and/or geographical domains.

The SFC Architecture can be implemented by making use of NSH, which defines a Service Plane protocol, specific for the creation of dy-

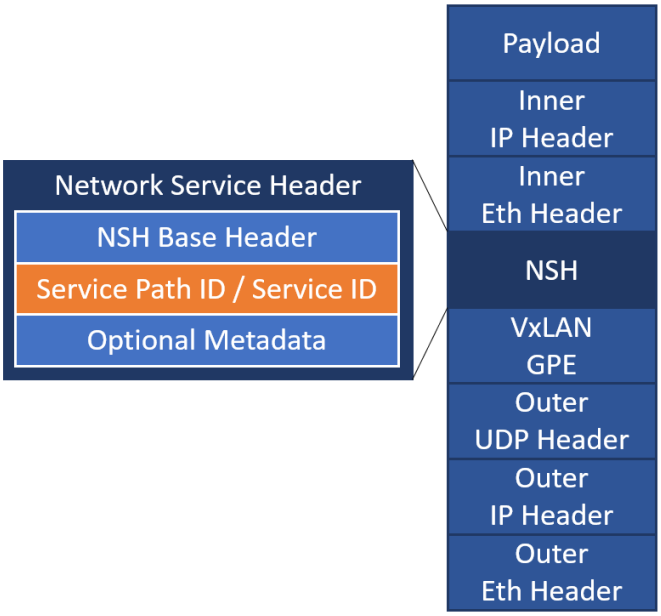


Figure 3.1: Structure of a NSH-encapsulated packet using VXLAN for transport.

namic SFCs. It provides SFP identification, transport-independent chaining, and packet-based network and service metadata. NSH is designed to be easy to implement across a range of devices, both physical and virtual, including hardware platforms. The NSH header is summarized and given context among the other fields of a packet in Figure 3.1.

The two most important fields in the NSH header are the *Service Path Identifier* (SPI) and the *Service Index* (SI). The SPI is a 24-bit integer number assigned to packets by the first SFC-Cl in the SFP, and all nodes taking part in that SFP must use the same SPI consistently. The SI, an 8-bit integer number, is used to identify the location within the SFP. The SI must be set by the initial SFC-Cl either to its maximum value (i.e., 255) or to a value related to the length of the SFP, and it must be decremented by one unit by all SFC-aware SFs and SFC Proxies the packet traverses in the SFP.

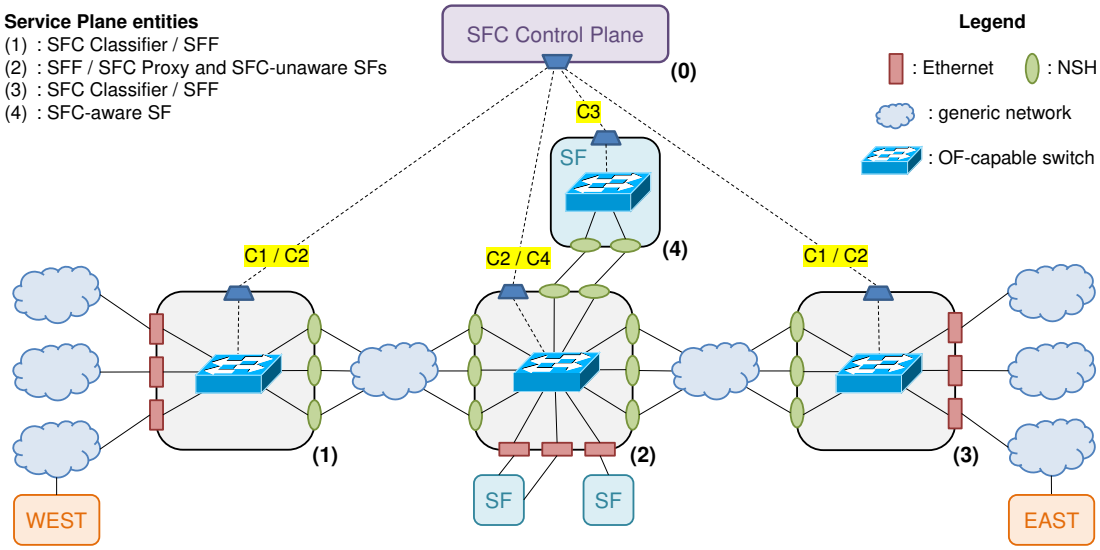


Figure 3.2: Reference scenario: the role of Nodes (1) to (4) is shown in the upper left corner.

3.1.2 OpenFlow-based NSH control plane

The reference scenario for the proposed NSH control plane is shown in Figure 3.2. It is composed of a SFC-CP entity, a pair of SFC-Cls, an intermediate node serving as both SFF and SFC-Pr towards SFC-unaware SFs, a SFC-aware SF, and two SFC-unaware SFs. In the reference implementation, it is assumed that each Service Plane entity is built around an OpenFlow-capable switch (OF-S). Then, all SFC entities are interconnected by means of a tunneling technology (e.g., VXLAN) through an underlying network infrastructure, controlled by one or multiple network operators through a generic control plane paradigm. The network infrastructure can use either SDN or non-SDN control, but this does not matter because the proposed SFC-CP is separate from the network control plane. Therefore, *service providers and network providers can act as completely independent entities, each adopting its favorite control plane approach.*

Mapping a SFP to the transport network requires to define a relationship between a given position in the SFP (i.e., a SPI/SI pair) and a certain next-hop in the underlying network. While the former

information belongs to the Service Plane, the latter depends on the network's topology and technology, as it must point to an existing location in the underlying network, typically expressed as an address (e.g., IP or MAC). How to implement this mapping is not a matter of standardization and different solutions may be adopted. The suggested mapping strategy is based on a rather straightforward idea: *mapping the SFP-to-transport relationship onto the ports of the employed OF-S.*

In the proposed solution, the SFP-to-transport relationship is actually mapped onto the ports of the employed OF-S. In fact, each NSH interface, corresponding to a specific SPI/SI pair, is bridged to a port on the node's internal OF-S. Through the association of SPI/SI pairs to ports on a OF-S, it is possible to have the node acting as a NSH Service Plane component while controlling it through the OpenFlow protocol from an SDN Controller, which takes the role of SFC Control Plane entity (SFC-Co) running applications that enforce Service Plane policies.

It is possible to add multiple NSH interfaces to each node, bridging them to ports of a OF-S, and program the Service Plane actions of the node through the OpenFlow protocol. The NSH mapping tables are therefore implemented in the form of flow tables inside the OF-S. For example, assume port N of the OF-S is bridged to interface `nshM` of the node. Instructing the switch (via a flow table action) to send traffic out of port N will result in the node sending NSH-encapsulated traffic out of interface `nshM` with the corresponding SPI/SI values.

Therefore, depending on what kind of flow rules are installed in the internal OF-S, a SFC node can be programmed to perform different Service Plane entity functions. With reference to Figure 3.2, the entities are mapped to the nodes in the following way:

- Node (0) hosts the SFC-Co.
- Node (1) is responsible for adding the NSH tag to packets coming from *WEST* hosts and forwarding NSH-encapsulated packets to the first SFF in the SFP: in this role, it acts as SFC-Cl. Additionally, this node is also responsible for removing the NSH

tag from packets assigned to a SFP which ends at Node (1), such as packets destined to *WEST* hosts, thus acting as SFF. Following this approach, the SFC classification is as expressive as OpenFlow matching is.

- Node (2) is responsible for handling the NSH encapsulation on behalf of SFC-unaware SFs, as well as for forwarding the NSH-encapsulated packets to the following SF or SFF in the SFP. In those two tasks, Node (2) acts as SFC-Pr and SFF, respectively.
- Node (3), similarly to Node 1, acts both as SFC-Cl and SFF for the traffic exchanged with *EAST* hosts.
- Node (4) acts as a SFC-aware SF, as it is able to receive NSH-encapsulated packets from the SFF and process them, before sending them back to the SFF after updating the SI.

3.1.3 Experimental validation

Testbed setup

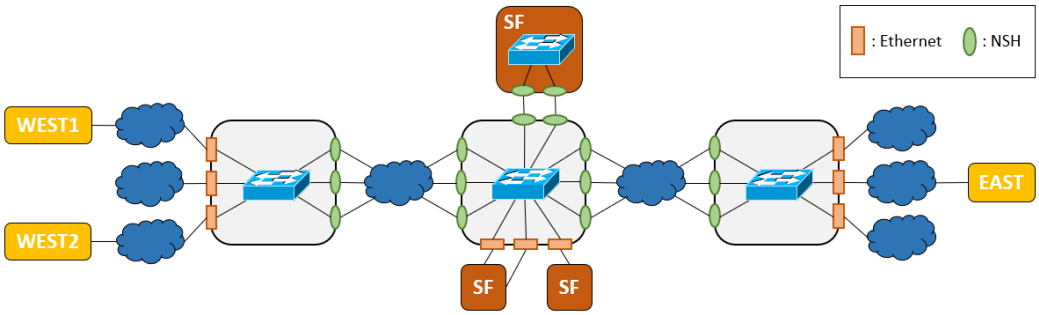
As a proof of concept, the proposed solution was implemented in a testbed, based on the reference scenario illustrated in Figure 3.2. The testbed comprises a total of five Virtual Machines (VMs) and the interconnecting virtual networks. The VMs are deployed on a single physical server, and virtualization is managed through libvirt/KVM. One of them hosts an instance of the SDN Controller ONOS [B43], while the remaining four VMs implement NSH-capable nodes. The choice of ONOS as SDN Controller (therefore, in this testbed, as SFC-Co) is motivated by its availability of Java and REST APIs, along with a well-documented Command Line Interface (CLI) and Graphical User Interface (GUI), allowing for easier monitoring of the controller's activities. However, this choice does not affect the generality of the implementation. Depending on the required virtual topology, some nodes also host virtual OpenFlow-capable switches, deployed using Open vSwitch (OvS) bridges, or a few additional terminal hosts, obtained by means of Linux network namespaces. One of the virtual networks serves as control and management network, and connects

the SFC-Co with the NSH nodes. The remaining virtual networks emulate the underlying network infrastructure. The open-source NSH kernel module [B54] was installed on each NSH node. This open-source NSH implementation allows to define logical network interfaces capable of encapsulating Ethernet traffic into NSH data units with specified SPI/SI values and transport technology.

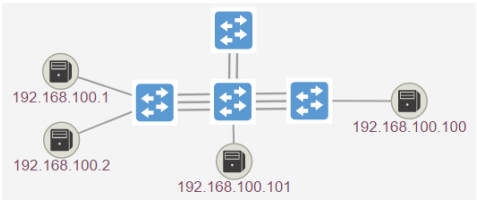
Each NSH interface was assigned a SPI/SI pair, and was mapped to a transport-level next-hop (i.e., an IP address), instructing the node to use VXLAN as encapsulation protocol to obtain the overlay topology. This is equivalent to adding an entry in the NSH-to-transport mapping table specifying that all traffic addressed to the endpoint with that SPI/SI should be encapsulated in VXLAN packets and sent to the specified remote IP address. Similarly each NSH interface was made aware of the inbound SPI/SI values it is meant to receive. Thus the mapping was achieved for outgoing and incoming traffic.

The transport network infrastructure will be traversed by as many VXLAN tunnels as the number of SPI/SI pairs defined. Each packet sent out by the VMs over one of their NSH interfaces will be intercepted by the NSH kernel module and encapsulated in a NSH/VXLAN packet, obtaining the SPI/SI pair assigned to the NSH interface. Similarly, when a packet is received on one of the NSH interfaces, the kernel module will intercept it and remove the NSH/VXLAN encapsulation, before handling the packet to the traditional IP forwarding module of the VM.

As previously mentioned, the NSH nodes were created employing OvS bridges as internal OF-S, programmed by the SDN Controller/SFC-Co. The NSH logical interfaces were attached to the OvS ports. The *WEST* and *EAST* hosts, as well as the SFC-unaware SFs, were implemented as logically isolated virtual entities by means of Linux network namespace technology. Thus, the full set-up depicted in Figure 3.2 was obtained. A *Deep Packet Inspector* (DPI) and a *Traffic Controller/Shaper* (TC) were deployed as SFC-unaware SFs, the latter configured with two Layer-2 interfaces (inbound and outbound). The SFC-aware SF is an *Integrity Checker* (IC).



(a) Logical topology with key physical components.



(b) Topology perceived by the SDN Controller/SFC-Co

Figure 3.3: Testbed topology from different perspectives.

The overall topology of the testbed is shown in Figure 3.3, where Figure 3.3a highlights the involved VMs and transport domains, while the topology as it is perceived by the SFC-Co is shown in Figure 3.3b. The SFC-Co is only aware of the overlay topology, i.e., the set of nodes and interconnections belonging to the Service Plane. Although there are two SFs, only one of them is reported by the topology manager of the SFC-Co.This is due to the fact that one of the SFs is interconnected to the rest of the testbed with two interfaces lacking an IP address, and therefore not addressable at network level, making it impossible to find for the discovery module of the SDN Controller/SFC-Co.

The *WEST* hosts represent users wishing to communicate with the *EAST* hosts. Different (categories of) users must be assigned different priorities, and the following service policies must be enforced:

- traffic coming from user WEST1 should be first checked by the DPI and then copied in the IC;
- traffic coming from user WEST2 should be first checked by the

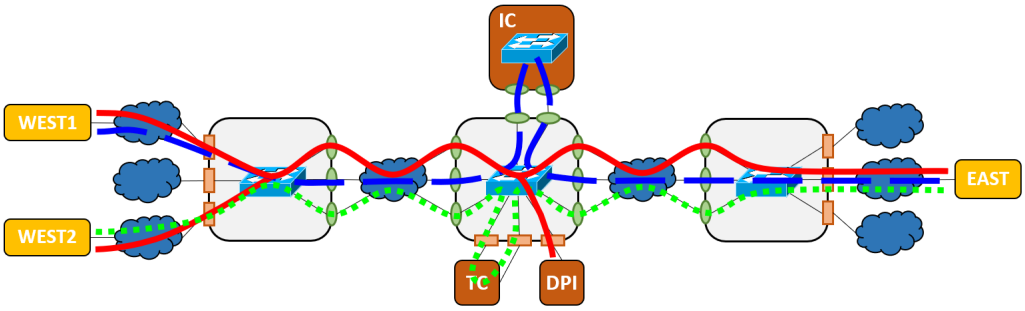


Figure 3.4: The considered SFCs and related SFPs on the deployed testbed topology; SFC1 in red solid line, SFC2 in blue dashed line, SFC3 in green dotted line.

DPI and then limited in bandwidth by TC.

Therefore, three possible SFCs are needed:

- SFC1, from any WEST user to the destination EAST user, duplicating the traffic towards the DPI;
- SFC2, from a high-priority WEST user to the destination EAST user, passing through the IC;
- SFC3, from a low-priority WEST user to the destination EAST user, passing through TC for bandwidth limitation.

To each SFC corresponds a SFP, and those related to the decribed SFCs are depicted in Figure 3.4.

Proof-of-Concept validation

A basic orchestrator (implemented as a script emulating an orchestrator's interaction with ONOS) was deployed in the SFC-Co node, in order to accomplishes the desired dynamic SFC behavior. The orchestrator installs proactive flow rules in the OF-S internal to relevant SFC entities, so as to apply chain SFC1. Then, it waits for any WEST user to start a flow of traffic towards the destination EAST user. When the flow starts, the orchestrator starts the DPI, and after a small time period, it retrieves information from it. If the inspected traffic contained data from WEST1, the script installs rules applying SFC2, otherwise, if the traffic contained data from WEST2, the script installs rules ap-

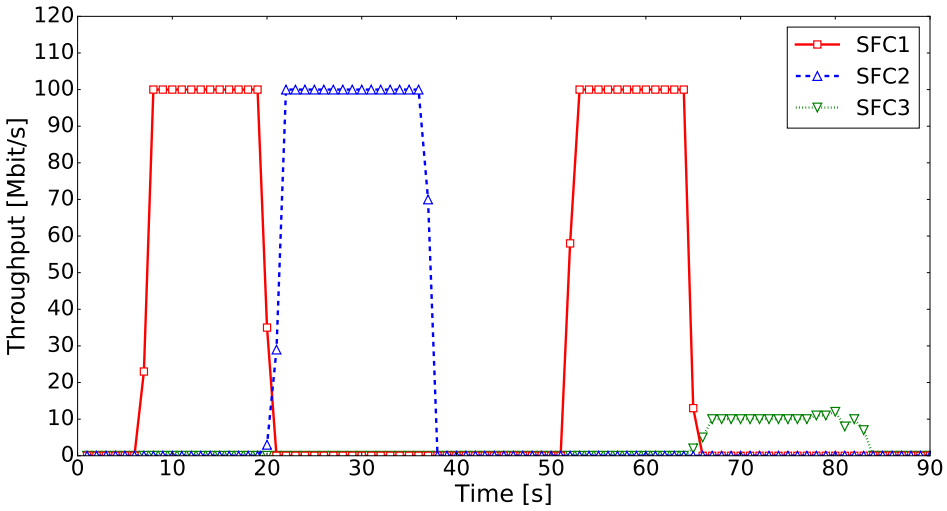


Figure 3.5: WEST-to-EAST throughput measured at the OF-S within Node (2) while applying dynamic SFC.

plying SFC3. It should be noted that traffic flows are steered to a different SFP without stopping them, thus achieving dynamic SFC.

The WEST-to-EAST throughput measured at the OF-S within Node (2) while applying the different SFCs is shown in Figure 3.5. At first, SFC1 is applied to traffic from WEST1 (from $t = 8$s to $t = 20$s), then after inspection SFC2 is applied (from $t = 21$s to $t = 38$s). Later on, traffic from WEST2 is subject to SFC1 (from $t = 53$s to $t = 65$s), then after inspection SFC3 with shaping is applied (from $t = 66$s to $t = 84$s). This outcome proves the correct implementation of dynamic SFC in the testbed.

3.1.4 Remarks

The SFC Control Plane solution proposed in this paper is based on the SDN paradigm. However, the SFC-Co and the SDN controller remain logically separated entities. In the proposed approach, assuming SFC entities that are built around an OpenFlow-capable switch, one can take advantage of the inherent dynamicity and programmability of SDN also in the Service Plane, while keeping it independent of the underlying network infrastructure. Therefore, network providers and

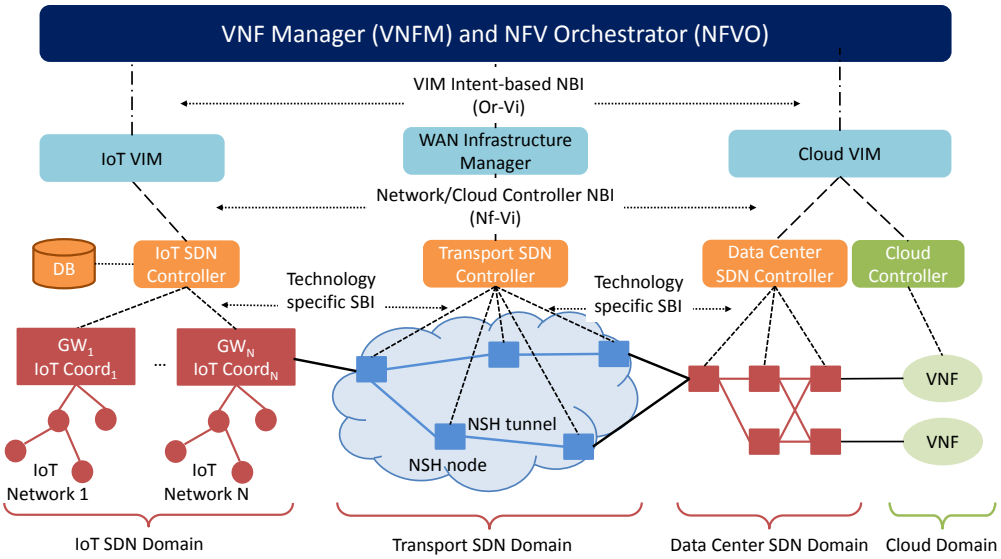


Figure 3.6: Reference multi-domain SDN/NFV architecture. Three different technological domains are displayed here, including an IoT domain, a data center and Cloud domain, and a geographical transport network domain.

service providers can adopt completely separate Control Plane solutions. The results gathered are promising and encouraging, as they prove that the proposed approach is feasible and effective.

3.2 SFC over IoT, Cloud, Fog and non-SDN tranport domains

The implementation of the SFC-aware control plane presented up to this point was employed in [P10] to extend the work on dynamic chaining over SDN domains presented in Chapter 2, particularly the part covered in Section 2.1. The architecture shown in Figure 3.6 is obtained by including a non-SDN transport domain in between the IoT and Cloud domains shown in Figure 2.1. The testbed reflects this addition too.

The role of the transport domain is to provide inter-domain con-

nectivity between IoT and data center/Cloud domains across a general geographical network, as required by the service chain to be instantiated. Although the SDN concept has recently been extended to inter-data center transport networks [B16] and to flexible wide area network (WAN) interconnections [B18, B17], the implementation presented here is independent of the control capabilities offered by the transport network. The rationale behind this is that an overlay approach allows to deal with heterogeneous forwarding technologies in the transport domain.

In order to keep service provisioning operations separate from and independent of the underlying transport infrastructure, the NSH approach is adopted, using the OpenFlow-based implementation described in the first part of this chapter. As detailed in Sections 3.1.1, when used in conjunction with a tunneling technology (e.g., VXLAN), NSH can be seen as a way to implement a network overlay enabling service function chaining on top of legacy transport networks. Moreover, an SDN-like solution for implementing the NSH control plane enables a seamless integration of the NBI of the transport infrastructure manager with the NBI adopted in the IoT and data center/Cloud domains, as well as the ability to dynamically adapt traffic flow forwarding to the requirements of the SFC being deployed.

Finally, the transport domain was implemented on a legacy physical network, on top of which NSH encapsulation and VXLAN tunneling between pairs of NSH-capable nodes were enabled. The NSH endpoints serve as SFC-Cls, as introduced in Section 3.1.1. An instance of the SDN controller Ryu [B55] implements the SDN controller responsible for steering the traffic in the transport domain. It does so by means of NSH encapsulation and dynamic SPI/SI allocation.

In order to validate the adaptive traffic steering capabilities of the NSH-based transport domain, three NSH endpoints were deployed as ingress/egress nodes exchanging traffic with other domains. One endpoint was connected to the IoT domain gateway located at the University of Bologna premises. A second endpoint was connected to the VMs previously mentioned, where the data center SDN domain

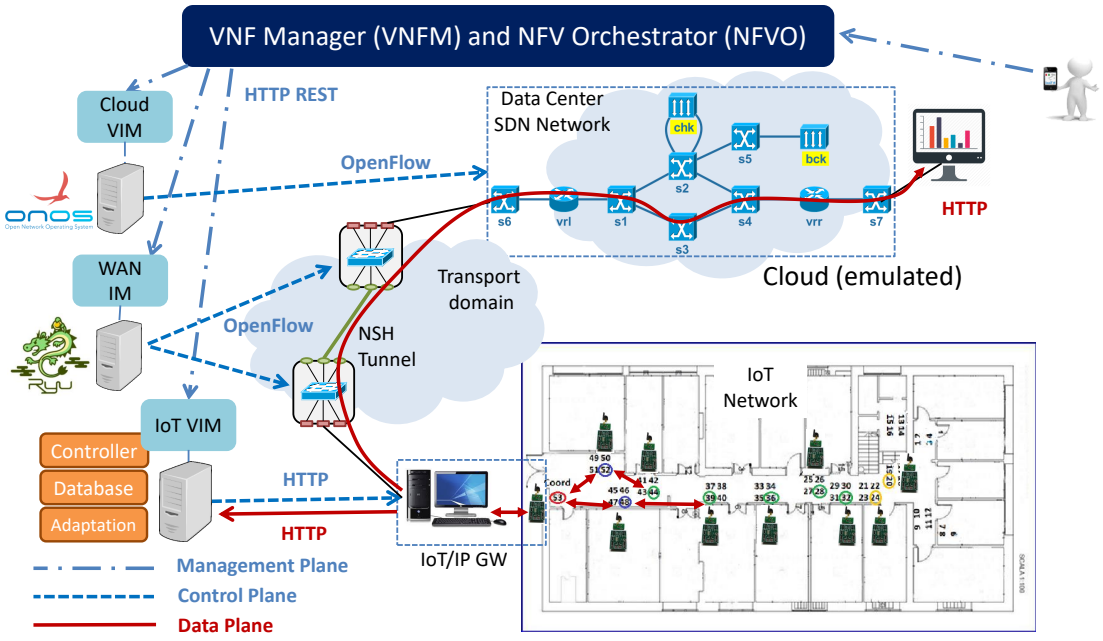
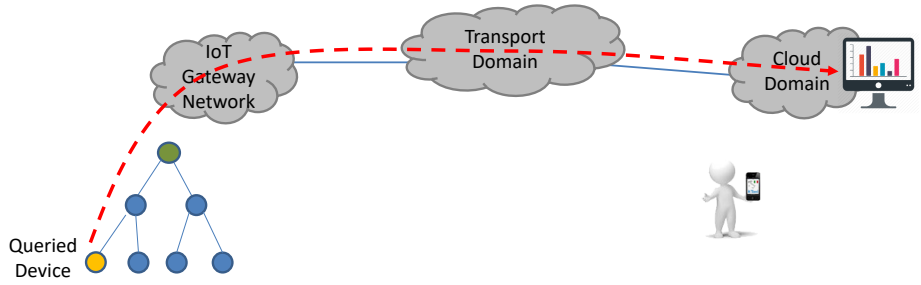


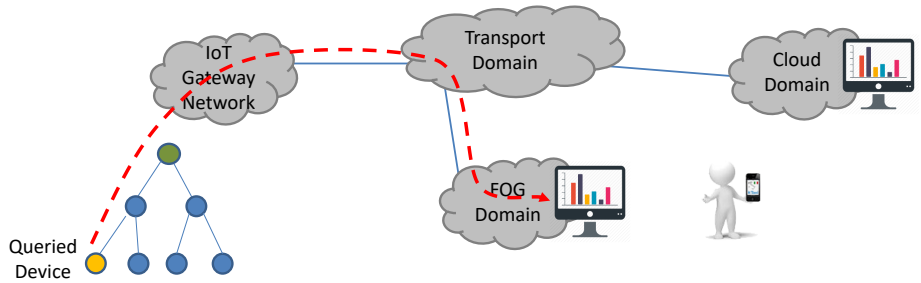
Figure 3.7: The NFV/SDN testbed setup developed to demonstrate end-to-end multi-domain service management.

and the Cloud domain were emulated with Linux namespace technology. Those VMs were deployed on a physical server located in a research-oriented computing facility in Belgium, belonging to the same federated experimental facility (i.e., Fed4FIRE+) mentioned in Section 2.2.

A set of experiments were run after instantiating the data collection/processing server also in a virtual machine located at the University of Bologna and connected to the third NSH endpoint. The latter setup was employed to emulate the scenario where the required service is discovered in an edge or Fog computing domain located closer to the IoT domain with respect to the remote Cloud domain. In this case, the edge/Fog node offering the service may not be continuously available, due to the limited and variable (e.g., due to mobility) number of resources available in such kind of computing environments. However, when the required resources can be found in a local edge/Fog domain, it is preferable to take advantage of them so that a delay-sensitive service can be delivered with a reduced data plane latency, resulting



(a) Deployment across the IoT, transport, and data center/Cloud domains.



(b) Deployment when the required resources are available in a Fog domain located closer to the user or IoT domain.

Figure 3.8: End-to-end service deployment in different conditions, depending on the availability of Fog resources.

also in a reduced traffic load in the transport network. The adaptive traffic steering capabilities of the NSH-based transport domain allow to dynamically change the end-to-end service deployment from the Cloud-based scenario to the edge/Fog-based one, as sketched in Figure 3.8.

The validation of the transport domain focused on the data plane latency between the NSH endpoint connected to the IoT domain and the NSH endpoint connected to the domain where the "data consumer" is located. To functionally validate the adaptive traffic steering capabilities of the SDN control plane adopted for the NSH-based overlay, a delay-sensitive service was first deployed in the remote Cloud , then at some point it was assumed that suitable resources were dis-

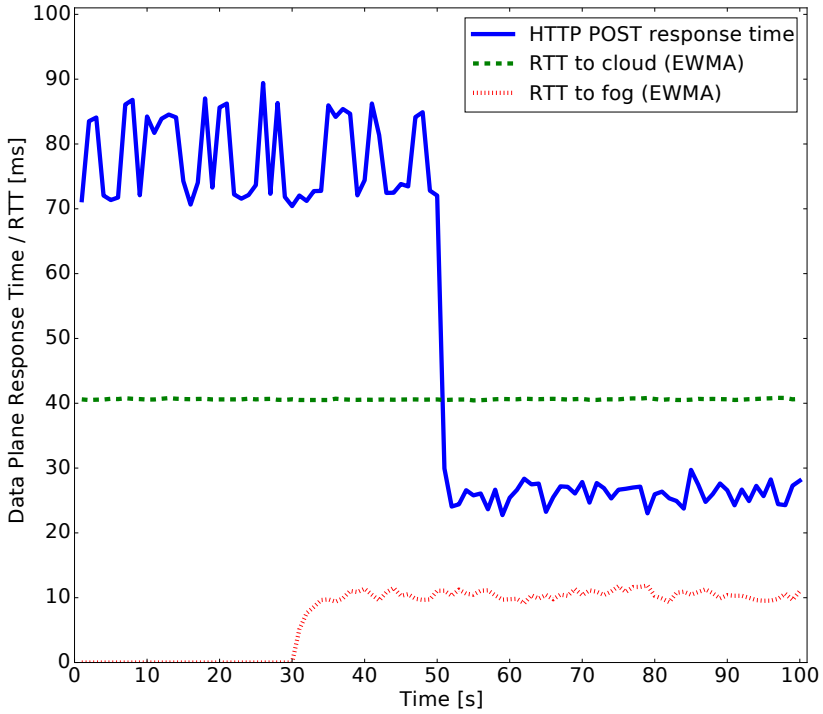


Figure 3.9: Temporal evolution of the transport data plane response time for HTTP POST requests and corresponding measured RTT values (EWMA with weight $\alpha = 0.5$). When the server in the fog domain becomes available and traffic is steered towards it, the overall response time improves significantly.

covered in a Fog domain located closer to the IoT domain. A fully fledged resource discovery mechanism was not implemented, as this is out of the scope of this validation, that resorts to `ping` responses to detect when the VM, representing the resource located at the edge/Fog domain, becomes active. This very simple resource discovery mechanism is deemed sufficient to demonstrate the correct behavior of the traffic steering in the NSH-based transport domain. To assess RTT, `ping`-based periodic measurements between each pair of NSH endpoints (IoT-to-Cloud and IoT-to-Fog) were employed, so as to choose the target domain offering the minimum data plane latency. In order

to stabilize the RTT measurements, the exponential weighted moving average (EWMA) of the collected RTT samples was considered, with weight $\alpha = 0.5$. In the implemented testbed, it was impossible to accurately assess single-way latency, as the source and destination NSH endpoints resided in different and remote physical machines, with non-synchronized clock sources.

As a realistic estimation of the response time in the transport network data plane, the time needed to complete a series of HTTP POST requests from endpoint to endpoint was measured, taking into account TCP session setup, HTTP POST message request, and `200 OK` response. The POST messages were generated and sent by the node serving as NSH endpoint connected to the IoT domain, and were received and acknowledged by the node serving as the NSH endpoint connected to either the Cloud or the Fog domain. A total of 100 POST requests were generated, and sent at intervals of 1 second. In Figure 3.9, the temporal evolution of the transport data plane response time for HTTP POST requests is represented by the blue solid line, while the network-level EWMA of the RTT is represented by the green dashed line for the Cloud domain and by the red dotted line for the Fog domain. At the beginning, the HTTP traffic is sent towards the server in the Cloud domain, with a quite steady RTT moving average of about 40 ms. From $t = 0$s to $t = 50$s, the traffic actually reaches the Cloud domain (located in Belgium), and the fluctuations in the measured response time are mainly caused by application-level delays. Meanwhile, at $t = 30$s the periodic ping measurement detects that the Fog node has become available, with a RTT moving average of about 10 ms, significantly lower than the RTT measured toward the Cloud . After a resource/service discovery period, assumed to be completed at $t = 50s$, the transport domain SDN controller steers the traffic coming from the IoT endpoint toward the Fog domain, achieving overall better latency performances. This validates the correct behavior of the transport domain control plane from the functional point of view. The difference between the RTT values and HTTP POST response times is due to the additional overhead included in the HTTP POST

transaction, with respect to a simple echo request/reply (*ping*) packet exchange.

As a final validation, the actual end-to-end service deployment time across the multi-domain scenarios shown in Figure 3.8 was evaluated. For this analysis, it is worth remarking that the service deployment response time is due to the response time of the management plane, consisting of the VIMs orchestrating the service implementation via the NBI, the delay in the network control plane, implemented by the SDN controllers, i.e. the IoTC/ONOS/Ryu platforms in this specific testbed, and the data plane latency required by the data traveling the network once the SFC is deployed.

In this case, the measurement focused on the time needed for the user's request containing the intent-based service specification to reach the VIMs in the different domains, the generation of data in the IoT domain, its transmission through the transport domain to the destination server in the Cloud/Fog domain, and the final acknowledgment. The time needed to actually program the network control plane was not included, as it was already presented in Table 2.2.

The measured average values, computed over 100 samples and shown in Table 3.1, are all in the proximity of 0.5 s, and the largest contribution to is given by the service management plane (orchestration, intent-based request set and processing, etc.), with just about 10% given by the network data plane latency. Nonetheless, the reduced network latency is evident when the service is "re-routed" to the Fog domain. This is very important because, for all the data posted after the service set-up, the network delay would be the only component (the time needed by the management plane being needed just at set-up) and therefore they would experience an improvement in response time of almost 100%.

3.2.1 Remarks

The reported validation results demonstrate the feasibility of the approach and the potentials of the NBI applied in real environments over a heterogeneous OpenFlow/IoT SDN testbed with Fog comput-

QoS feature	Cloud scenario	Fog scenario
Delay sensitive	532.3 ms	511.8 ms
Loss sensitive	554.0 ms	530.1 ms

Table 3.1: Average end-to-end service deployment time, for different QoS features and Cloud or Fog domain scenarios.

ing options. The latency values measured at both data and control/-management planes allowed to get a first insight to the performance levels of the overall system, resulting in reasonable response times for service setup and QoS requirement satisfaction. Scalability tests also gave promising results. The reported use case represents a working example of a more general approach to properly define high-level interfaces and develop the related control and management components to unify orchestration capabilities across multiple SDN/NFV domains. As a future direction, the performance can be tested after generalizing the proposed intent-based NBI in order to encompass different service scenarios that may involve multiple domains, such as 5G network slicing or multi-access edge computing. Anothering interesting direction is that of the development of a mathematical formulation of the intent mapping problem and an intent specification interpreter based on natural language.

Chapter 4

Resource monitoring in SDN/NFV environments

This chapter discusses typical monitoring issues in SDN/NFV infrastructures, and describes a module for unified resource monitoring over them, as introduced in [P12].

4.1 Monitoring challenges and system architecture

In infrastructures where SDN, NFV and Cloud computing cooperate, it is of utmost importance to have an adequate monitoring solution, able to integrate metrics from the network domain with ones regarding computing features [B5]. Such monitoring system should be well integrated with other network components, to the point where its deployment should be as similar as possible to that of any other item. A number of features of a softwarized infrastructure, including global view over combined resources and on-demand resource provisioning, can be leveraged to face the aforementioned requirements, moving towards the design of a unified and standalone monitoring tool, such as the one presented in [P12]. The monitoring module introduced there is designed as a standalone VNF and it is compatible with a generic SDN/NFV infrastructure deployed in a cloud environment.

The module is separated from the existing control plane components, this way sparing critical elements of the infrastructure from additional workload, yet still being able to collect measurements for different assets. This design enables flexible integration with various control plane components or measurement tools, without the need for modifications in the existing software or hardware solutions, thanks to the usage of well-known and widely implemented protocols and interfaces only. Moreover, the process of updating the monitoring module is transparent from the point of view of the other components.

This standalone solution could be used to provide monitoring information to a service orchestrator operating over network and computing resources (e.g., [B56]) as well as to investigate the limitations of virtualized infrastructures whose characteristics are outside of the control of the tenant. The proposed system has been implemented, deployed, and validated in three different scenarios, namely a public cloud platform, a virtualized scenario built over physical machines using container technology, and a public cloud environment, to demonstrate the portability of the solution.

Rather than focusing on solutions for distinct monitoring of SDN and NFV entities, such as the ones reported in [B22, B23, B24], the challenges this work focuses on are among those pointed out in [B25]. For more related work, see [P12].

To fully benefit from the union of SDN, NFV and Cloud potentialities, the monitoring module needs to be able to conduct unified measurements over distributed heterogeneous resources, combining all the metrics and inferring conclusions on resource usage. To accomplish control plane decoupling, the module should also be independent of any controller in the infrastructure. It should be possible to consider the monitoring tool as a special VNF, inheriting the same ease of deployment and usage typical of virtualized functions, and make use of well-known protocols to allow seamless integration with existing infrastructures. Such a tool can, among other things, assist the processes of on-demand deployment of new VNFs as well as of dynamic traffic steering across the domain(s), in order to enforce a given

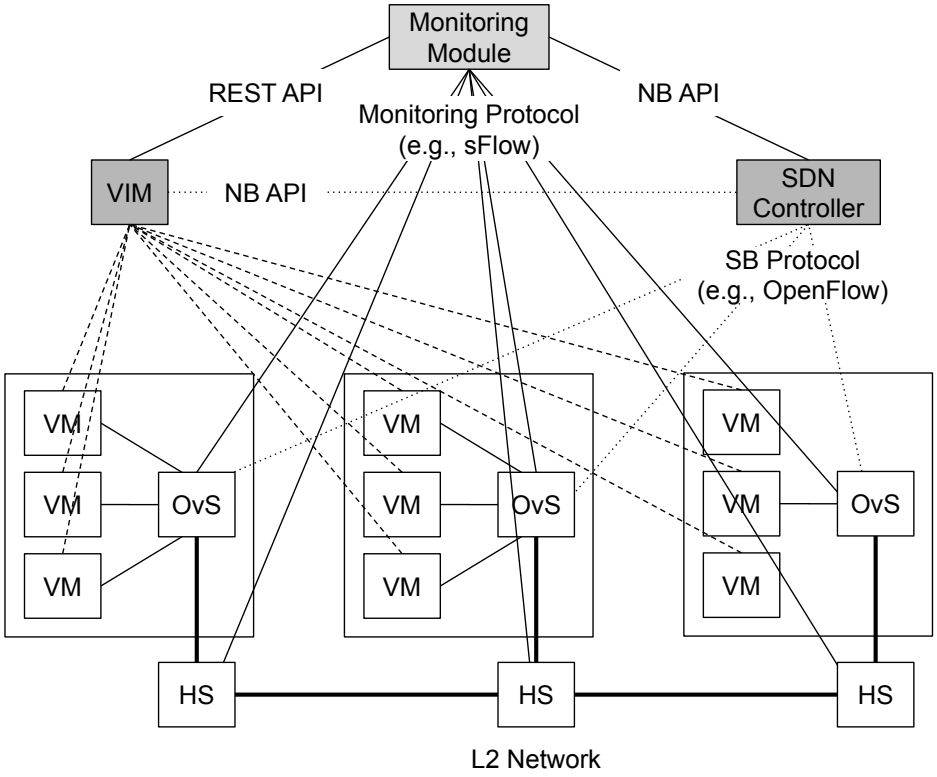


Figure 4.1: Architecture of the system, including data plane elements (white), control plane elements (dark gray), and the monitoring module (light gray). VIM - Virtualized Infrastructure Manager, NB - Northbound, SB - Southbound, HS - hardware switch. Solid, dashed and dotted lines are used to denote communication related to the monitoring module, SDN controller (NB and SB), and VIM, respectively.

service policy. In the current version, the proposed solution focuses on passive monitoring.

The architecture of the proposed unified monitoring system is shown in Figure 4.1. The monitoring module can collect information from various sources, including the VIM and the SDN controller, through their APIs, as well as directly from the network infrastructure components, using existing and widely available protocols, such as sFlow [B57]. The modular nature of the proposed monitoring system allows to count on certain amount of redundancy, by collecting information on the same aspects from different sources. While ensuring a level of fault

tolerance and protection against denial-of-service attacks, this also allows to tune the granularity of the monitoring data. Moreover, the monitoring module can also be employed as a trigger for corrective actions in the monitored domain, by suggesting to the network controller or infrastructure manager which corrective actions might be needed.

4.2 Prototype implementation

The monitoring module itself has been developed as a software application written in Python, due to the suitability of the language for prototyping purposes and ease of implementation of a REST API. It was designed to collect and aggregate information gathered from multiple sources. In the prototype, the module implements data collection from Openstack Ceilometer/Gnocchi as well as flow sampling directly from the network devices by means of a custom implementation of an sFlow collector. The sFlow protocol was selected among a number of flow statistics gathering tools due to its widespread native support by network devices, including virtual switches. While sFlow is suitable for the purpose of this prototype, it may not be sufficient to perform detailed real-time monitoring tasks. Thanks to the modular architecture of the monitoring system, the sFlow protocol can always be replaced with a different data plane monitoring solution [B58].

The main parameters that can be used to tune the behavior of the sFlow data collection are the ***sampling ratio*** and the ***sample aggregation interval***.

The sampling ratio N denotes that, on average, one out of N packets handled by the node will be sent by the sFlow agent to the sFlow collector for the purposes of statistics gathering. The higher the value of N, the lower the communication and computational overhead. However, by increasing N the measurements are available with a higher delay and worse sensitivity. By default the sFlow agent only sends the first B bytes of the sampled packet to the sFlow collector, so as to include the packet header, along with an indication of the total packet size. This way the sFlow protocol overhead is reduced,

without impacting the statistical accuracy.

The sample aggregation interval C denotes the interval over which individual sFlow samples are aggregated, causing that all samples received within a time window of C seconds are combined and treated as single measurement entry. This aggregation aims to attribute a group of asynchronous samples to a given moment in time, thus achieving a tunable "quantization" of the time axis. Similarly to the sampling ratio, increasing the value of the sample aggregation interval C reduces the monitoring overhead and deteriorates the monitoring capabilities in terms of measurements sensitivity and time needed to obtain the collected monitoring data.

A more detailed representation of the prototype monitoring module and of the interactions between monitoring components is shown in Figure 4.2. The sFlow agent in each OvS samples one packet out of N for a particular flow that the OvS forwards in the data plane network – a flow being defined by the source and destination of the traffic. Sampled packets are assembled in sets comprising two to six units, and sent to the sFlow collector in a sFlow datagram packet. The rate of arrival of sFlow datagram packets to the sFlow collector is not steady nor predictable, as it depends on the intensity of the data plane traffic through the sampling ratio N. Therefore, there is no synchronization between sFlow agent and sFlow collector. The collector aggregates sFlow datagrams over temporal windows spanning C seconds, where C is the aforementioned sample aggregation interval, generating one measurement entry every C seconds, on the closing instant t_n of the current temporal window. Figure 4.2 also visualizes how N and C parameters affect the delay after which measurements are available and sensitivity. Each measurement entry coming from the sFlow collector is the result of the combination of the most recent sFlow sample with the recent previous ones through an EWMA. Parallel to the sFlow data sampling and collection, the other main component of the monitoring module, the Ceilometer poller, periodically polls the Cloud controller to retrieve data measured from there, and generating additional measurement entries. For each flow, two metrics are collected, namely an

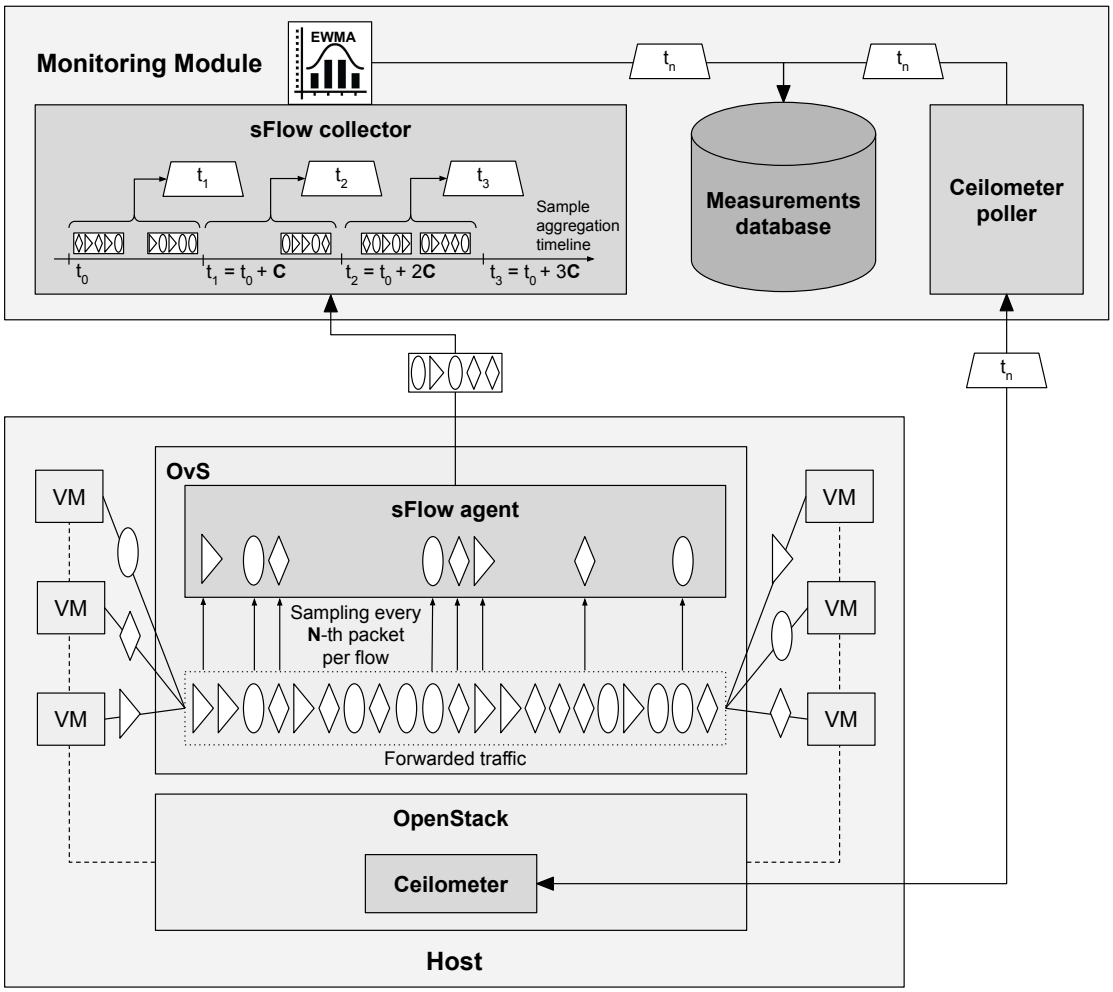


Figure 4.2: Inside view of the prototype monitoring module with a representation of the main mechanisms.

instantaneous data rate estimate value, and an Exponential Weighted Moving Average (EWMA) value according to the recursive formula

$$e_n = \alpha s_n + (1 - \alpha) e_{n-1} \tag{4.1}$$

where e_n is the EWMA value computed when the n-th instantaneous sample s_n is received and $0 < \alpha < 1$ is a weighting coefficient. The EWMA was introduced to smooth out measurements and make them robust against load fluctuations. The α coefficient expresses how fast historical measurements lose importance: the higher its value, the

heavier the weight of instantaneous samples.

It is important to quantify the overhead caused by a monitoring protocol such as sFlow, i.e., the amount of additional signaling traffic exchanged between network nodes and the sFlow collector to perform the monitoring operations. For each sFlow packet p received by the collector, the relative overhead can be defined as the ratio of the size of the sFlow packet $L_{\text{sflow},p}$ over the total amount of data to which that packet refers. The latter quantity includes the size of sFlow packet p, plus the full size of each sampled data packet carried by sFlow packet p, plus the size of $N-1$ data packets not being sampled for each sampled packet in p. It is assumed that, on average, the total size of the $N-1$ non-sampled packets is $N-1$ times the size of the sampled packet; this assumption is equivalent to the assumption made by the sFlow protocol, which considers one sample out of N as an estimation of the monitored bit rate. Thus, the relative overhead of packet p can be approximated as:

$$O_{\text{sflow},p} = \frac{L_{\text{sflow},p}}{L_{\text{sflow},p} + N \sum_{i=1}^{n_p} L_i} \tag{4.2}$$

where n_p is the number of samples carried by sFlow packet p and L_i is the full size of the i-th sampled packet.

The size of a generic sFlow packet is variable and depends on the number and size of the carried samples, including meta-data associated to each sample and carried in a sample header. Recalling that sFlow packets are transported by UDP datagrams, in an Ethernet network:

$$L_{\text{sflow},p} = h_{\text{eth}} + h_{\text{ip}} + h_{\text{udp}} + h_{\text{sflow}} + \sum_{i=1}^{n_p} (h_{\text{sample},i} + B) \tag{4.3}$$

where h_{eth} is the Ethernet header size, h_{ip} is the IP header size, h_{udp} is the UDP header size, h_{sflow} is the sFlow header size, $h_{\text{sample},i}$ is the i-th sample header size, and B is the sample data size.

4.2.1 Testbed based on container technology

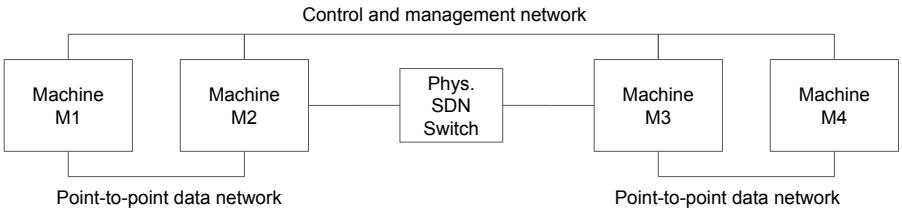
Container technology has gained immense popularity since its introduction, owing to the diverse benefits it can offer to a vast range of

applications and services. Containerization enables lightweight and scalable deployments of service functions in the network, facilitating dynamic service provisioning and management. This makes the inclusion of container technology meaningful for the thorough evaluation of the capabilities of this monitoring module. Therefore, the prototype was evaluated in a testbed based on container technology, managed by using Docker [B59], which acted as VIM, handling requests for deployment of new VNFs, managing their networking and overseeing their lifecycle.

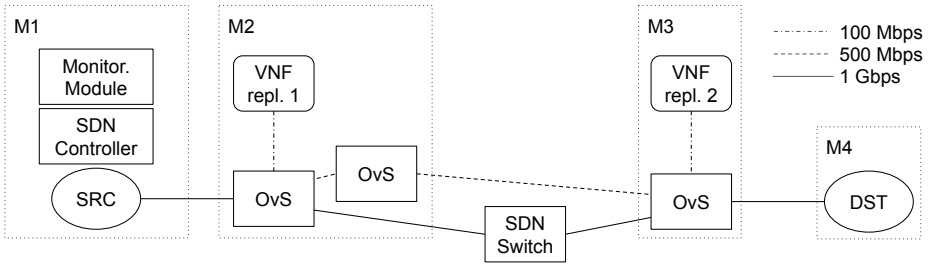
The container-based testbed consists of four physical machines, all equipped with 4 CPUs (Intel i5-4460 up to 3.2 GHz) and 8 GB of RAM. All of them are connected to the same local private network, which serves as control and management network only. Additionally, the machines are arranged in two pairs, by connecting the first two and the last two of them directly, and connecting the second and third machines via a physical SDN switch (the OpenFlow-enabled HP Aruba 2920-48G switch). The first machine hosts the running instances of the monitoring module and of the SDN controller (Ryu), as well as the source endpoint of the generated traffic flows. The second and third machines hosted the container management software (Docker), as well as the virtual switches (OvS) required to complete the desired overall logical topology. The fourth and last machine hosted the destination endpoints of the generated traffic flows. The links between each pair of machines had a physical capacity of 1 Gbit/s. However, some links in the final logical topology were limited to different values for evaluation purposes that will be addressed in the following. A depiction of the physical testbed, along with a representation of the intended logical topology, is given in Figure 4.3.

4.3 Experimental validation

Extensive data gathering campaigns and results analysis have been conducted, and published in [P12]. The following sections report only a part of them, specifically, those on traffic steering (in Sec-



(a) Physical setup.



(b) Logical setup.

Figure 4.3: Container-based testbed topology.

tion 4.3.2), plus an overview of some results on sFlow parameters impact on datarate estimation and protocol overhead (in Section 4.3.3).

4.3.1 Combined network and resource monitoring

To prove that there is a correlation between network interface and CPU load in VNFs, and that the monitoring module is able to appreciate both and correlate them, two traffic patterns were applied, and they are presented in Figure 4.4. The first scenario employs periodical traffic spikes of the same intensity, resulting from TCP sessions starting at times 5, 65, 125 seconds, lasting 30 seconds, and having unlimited throughput ("unlimited" by the application – the limit is a result of the saturation of link bandwidth). The second scenario employs traffic spikes with increasing intensity, given by TCP sessions starting at times 5, 65, 125, 185, 245, 305, 365, 425 seconds, lasting 30 seconds, and having throughput limited by the application that generates the traffic. The limits are set to 0.1, 0.2, 0.5, 1, 2, 5, 10 Gbit/s respectively to span the most typical transmission rates, while

the last TCP session is again throughput-unlimited.

Figure 4.4 presents both the number of bytes received and the CPU utilization of a selected VNF. The received bytes are measured by aggregating values found in sFlow samples, corresponding to data forwarded by the switch the VNF is attached to. The CPU load metric shows the percentage of time the CPU is busy in the specific machine running the selected VNF. As VNFs are running on separate virtual machines with dedicated resources, "CPU" here denotes the virtual CPU associated with a particular instance, that can be referred to as guest CPU.

4.3.2 Monitoring-based traffic steering

The containerized testbed was employed to perform proof-of-concept experiments on the ability of the proposed monitoring module to constructively interact with an SDN controller in order to support the dynamic steering of traffic flowing in the network, aimed at optimizing the utilization of physical and virtualized networking and computing resources. Traffic is generated with `iperf` using TCP with a specific throughput target, depending on the experiment.

Two case studies are examined. In the first one, referred to as *choose VNF*, traffic steering is performed to mitigate congestion of a particular VNF instance. This is possible thanks to the monitoring capabilities of the proposed module in a NFV domain. The desired action is to distribute traffic among different replicas of the same VNF, no matter what is their location considered from the network perspective. In the second case study, referred to as *choose path*, the aim of traffic steering is to avoid network congestion based on the measurements performed on a network node representing an SDN domain. Thus, it is critical to change the path of traffic to distribute the load between different network links and no matter if traffic destination is the same replica or not.

In both cases, the objective is to achieve the maximum overall throughput from source to destination, while respecting service policies, such as the requirement to cross a given VNF. This is achieved

in the *choose VNF* scenario by finding the VNF replica with sufficient computing resources even if the traffic is directed through the same network path. On the contrary, in *choose path* the goal is achieved by finding network links with sufficient resources even if the destination node is the same. Therefore, the monitoring module is able to trigger traffic steering based on the measurements performed in both NFV and SDN domains.

The logical topology, shown in Figure 4.3b, aims at providing a testbed that can be used to run experiments in both case studies. For these runs, the monitoring module was configured with $N = 10$, $\alpha = 0.4$ and $C = 1$ s.

Case study: choose VNF

All traffic is required to cross a replica of a given VNF. Any network function can be considered, e.g., traffic shaping, packet inspection for an intrusion detection system, or transcoding for multimedia traffic flows. As shown in Figure 4.4, the traffic load on the interface of the VNF is strongly correlated to its computational load. For the purpose of this test, the VNF does not perform any useful packet processing, but it simply forward the packet towards the intended destination. This simplification should be considered as a best case because any other function requires more computing resources for each packet received. Therefore, it is reasonable to assume that the measurements taken at network level can also give an insight of the computational burden on the VNF, justifying the actions taken to optimize the load. In these experiments, a total of four `iperf` sessions are launched from SRC to DST, all of them using TCP and aiming at a throughput of 30 Mbit/s. These sessions are activated sequentially at intervals of 30 seconds, and configured to last until the end of the experiment.

To begin with, a baseline experiment is run, in order to assess the behavior of the system when no steering is applied. All the generated traffic is crossing the same replica of the VNF. The evolution in time of the data rate of the traffic flow from the source host to the destination host is shown in Figure 4.5. As expected, after the fourth flow started,

i.e., at $t = 105$s in Figure 4.5a, the traffic saturates the capabilities of the VNF instance, and the flows needed to compete for the resources, causing the total throughput to be limited by the capacity of the link to the VNF, i.e., 100 Mbit/s. In the second experiment, this potential deterioration is avoided by taking advantage of the proposed monitoring module. The traffic is initially steered through the first replica of the VNF. The monitoring module keeps tracking of the increase in resource utilization in the VNF, verifying that the traffic is below a predetermined *warning threshold* set at 50 Mbit/s. When this threshold was exceeded, i.e., at $t = 45$s in Figure 4.5b, the monitoring module interacts with the VIM to find out the location of a second replica. Based on that, the monitoring module instructs the SDN controller to steer the traffic accordingly, in order for it to cross the second replica. This way, even when the four flows are running at the same time, they do not have to compete for the shared resources, and the full combined throughput can be achieved.

Case study: choose Path

This case study aims at highlighting the benefits of dynamic traffic steering over multiple paths in the network, in case switches are overloaded or not utilized optimally.

Similarly to the previous case, a total of four `iperf` sessions were launched from SRC to DST, all of them using TCP, this time aiming at a throughput of 200 Mbit/s. Once again, the sessions were activated sequentially at intervals of 30 seconds and configured to last until the end of the experiment. Initially, the traffic crossed the *upper path*, with reference to the logical topology in Figure 4.3b, where the link capacity is 500 Mbit/s.

Similarly to the case examined in the previous section, the temporal evolution of the data rate of source-to-destination traffic flow is shown in Figure 4.6. The baseline experiment, without traffic steering, shows that the links becomes saturated after the third traffic flow starts, at $t = 75$s in Figure 4.6a, then the combined throughput of flows is limited to the capacity of the link with smaller capacity, i.e.,

500 Mbit/s, which acts as a bottleneck. In fact, the impact of the fourth flow, starting at $t = 105$s, is practically invisible. The proposed monitoring module enables avoiding this service degradation. In the second part of the experiment, the monitoring module keeps tracking of the increase in network resource utilization, considering the load on the ports of the switch. Again, when a predefined threshold is exceeded, after the second flow starts at $t = 45$s in Figure 4.6b, the monitoring module triggers traffic steering mechanisms. This time it instructs the SDN controller to steer the traffic through the alternative path. The traffic is then directed through the not congested *lower path* and throughput of flows was no longer limited. The overall throughput reached its maximum value, equal to the sum of the throughput of the four flows.

4.3.3 Impact of sFlow parameters

First, the fundamental properties of the sFlow protocol are investigated, as a function of three parameters: the sFlow sampling ratio (N), the sample aggregation interval (C) and the α coefficient of the EWMA. The employed traffic pattern imposes a stepwise increasing load, by running three `iperf` data flows starting at times 5 s, 65 s and 125 s, respectively, with each flow aiming at a throughput of 30 Mbit/s and lasting until the end of the experiment.

Figure 4.7a shows the accuracy of the EWMA of the sFlow samples for different values of N, with $\alpha = 0.3$ and $C = 1$ s, comparing it to the generated load pattern and the instantaneous sFlow sample values collected for $N = 10$. Figure 4.7b reports the same measurements for $C = 10$ s. For $C = 1$ s, it can be seen that the EWMA shows higher variability when N is higher, because sFlow collects less samples and thus is more affected by temporal traffic fluctuations. Furthermore, the higher the generated load, the wider the fluctuations, for any value of N. Thus, the sampling ratio should be adjusted not only to the required accuracy, but also to the absolute traffic load. Changing the sample aggregation interval to $C = 10$ s reduces the number of sFlow samples, smoothing out all curves (Figure 4.7b). The obtained

measurements are therefore less sensitive to load fluctuations, but at the cost of worse responsiveness. The first general conclusion is that any tool considered for flows statistics gathering should be adjustable to the expected load and particular needs.

Based on the same results, it is also possible to draw conclusions regarding the timeliness of the collected measurements, as it could be a critical factor for some applications. It can be stated that the value of C represents a lower-bound of the delay introduced by the module, as results are updated once every C seconds at best. Some monitoring application might not need quicker results, so this configuration, although apparently quite "slow", is worth consideration.

For both $C = 1$ s and $C = 10$ s, sampling every packet ($N = 1$) provides accurate results until the third flow is injected in the network. At that point, the testbed is no longer able to measure network load accurately. This is caused by several reasons, one being the sub-optimal implementation of the sFlow collector. Moreover, $N = 1$ is not a practical value, as it causes the monitoring traffic to double the amount of data traffic present in the network, to some extent defeating the purpose of monitoring. Lastly, the testbed has some computational limits as it is run in a virtualized infrastructure. Therefore, the case for $N = 1$ is presented only for the sake of completeness, as well as to prove that sFlow is not a good choice if the objective is to carefully analyse each packet.

Evaluation of sFlow protocol overhead

The sFlow sampling ratio N affects the accuracy, as described in Section 4.3.3, and the overhead of the sFlow-based measurement process, as computed in eq. (4.2). Network nodes and collector exchange samples of packets from the overall traffic, plus the sFlow protocol header and the sample headers, as reported in eq. (4.3).

An additional parameter, namely the size B of the packet sample to be included in the sFlow packet, directly affects the overhead. The required sample size depends on the elaborations that must be carried out on the sampled data, but for the purpose of this performance

N	Measured relative overhead	Estimated relative overhead
1	15.07%	13.74%
2	8.17%	7.37%
5	3.50%	3.09%
10	1.83%	1.57%
20	0.98%	0.79%
50	0.46%	0.32%
100	0.28%	0.16%

Table 4.1: Measured and estimated relative overhead introduced by the sFlow protocol using different values of sampling ratio N.

evaluation, B is set to 128 bytes. Also, the presented results were collected with sample aggregation interval $C = 1$ s. It must also be noted that changing the α parameter does not impact the overhead of the sFlow protocol, as α affects only how samples are processed in the collector and does not change the amount of data that network nodes send to the collector. Therefore, the overhead results are presented as a function of the sFlow sampling ratio N.

The employed traffic pattern includes a single `iperf` session with 1 Mbit/s throughput ran for 60 seconds between two different virtual machines. Figure 4.8 presents the total absolute amount of sFlow signaling traffic being exchanged between network nodes and the sFlow collector. The results show how the sFlow protocol signalling decreases with the sampling ratio N.

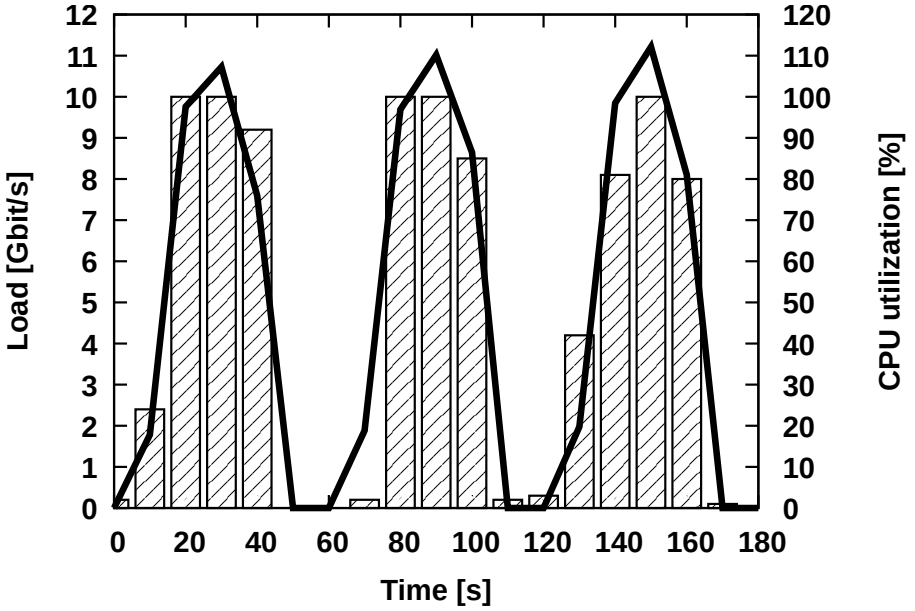
Table 4.1 reports the relative overhead as a percentage of the total traffic exchanged, for different values of N. The measured relative overhead was obtained by capturing the traffic at the output of a virtual switch and reporting the percentage of sFlow traffic over the total traffic captured. The estimated relative overhead was instead obtained by applying eq. (4.2) and considering the parameters related to the `iperf` session, that sent packets of $L_i = 1432$ bytes, $\forall i = 1, \ldots, n_p$. The small throughput chosen for this test allows to proficiently use any value of N, including $N = 1$, without overloading the system. For every sampled data packet, the first $B = 128$ bytes were sent to the sFlow collector. This implementation fo the module makes use

of sFlow version 5, which adds to each data sample some metadata that amounts to a sample header of $h_{\text{sample},i} = 88$ bytes. Therefore, each sample carried by the sFlow packet adds a total of 216 bytes to the packet size. Considering an sFlow packet carrying $n_p = 6$ samples and adding the standard protocol header sizes (i.e. $h_{\text{eth}} = 14$ bytes, $h_{\text{ip}} = 20$ bytes, $h_{\text{udp}} = 8$ bytes, and $h_{\text{sflow}} = 28$ bytes), then the size of the sFlow packet is $L_{\text{sflow},p} = 1366$ bytes. Considering eq. (4.2), it follows that $O_{\text{sflow},p} = 13.72\%$ for $n_p = 6$. While more than 90% of the captured sFlow packets carried 6 samples, the rest included from 1 to 5 samples. The estimated relative overhead reported in Table 4.1 shows the weighted average of $O_{\text{sflow},p}$ for $n_p = 1, \ldots, 6$. Conclusively, the approximate formula (4.2) comes very close but underestimates the actual relative overhead. The reason lies in the fact that the captured traffic included also some packets due to background traffic and whose size was smaller than 1432 bytes, thus increasing the actual overhead. However, the formula capture quite well the behavior of the overhead as a function of the sampling ratio, with an approximation error of 1.33% in the worst case (i.e, for $N = 1$). Thus, the expected overhead can easily be estimated, considering the trade-off against accuracy, and properly configure the monitoring tool making it suitable to the requirements of a given SDN/NFV infrastructure. Applicability and usefulness of the proposed formula are further increased by the fact that it can be adjusted to any other data plane monitoring solutions, making it possible to theoretically estimate the monitoring overhead.

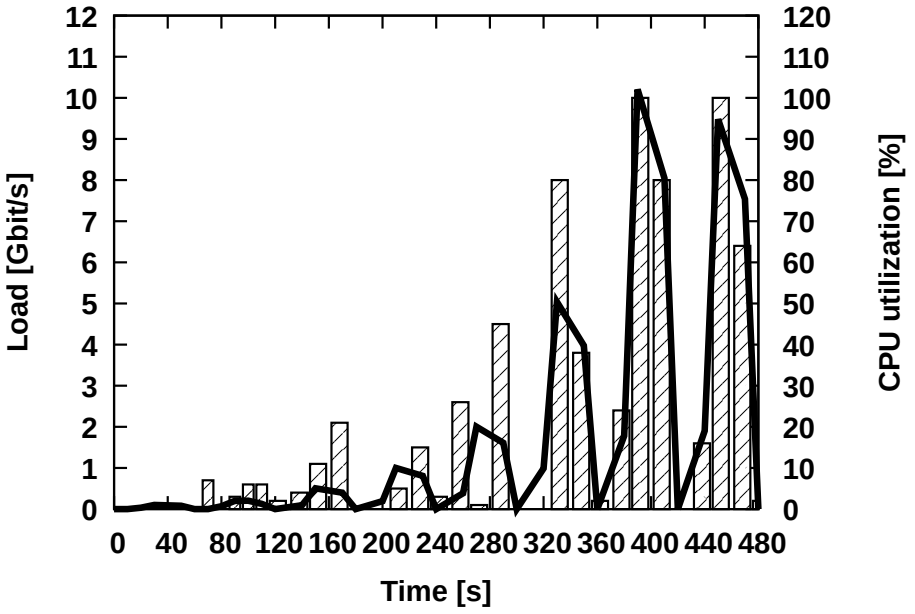
4.4 Remarks

Based on the experimental validation and collected results, the proposed module can be easily integrated with a variety of controllers as well as tools aimed at collecting metrics specific to particular assets. This approach does not impose a significant load on existing control plane components, due to the full independence of it, and the possibility of deploying the module in the form of a VNF. Although the specific case of sFlow as a network monitoring protocol was analyzed

in detail, the findings can be generalized to other specific technical solutions. The modular architecture provides significant advantages, and some limitations of the selected tools can be overcome by proper configuration or by replacing one of the loosely coupled components of the architecture. Future improvements include the deployment of selected virtual network functions, and employment of the proposed monitoring module to feed optimization algorithms aimed at improving infrastructure utilization and avoiding congestion, as well as the extension of the monitoring module to support orchestration of active measurements in software-based infrastructures.

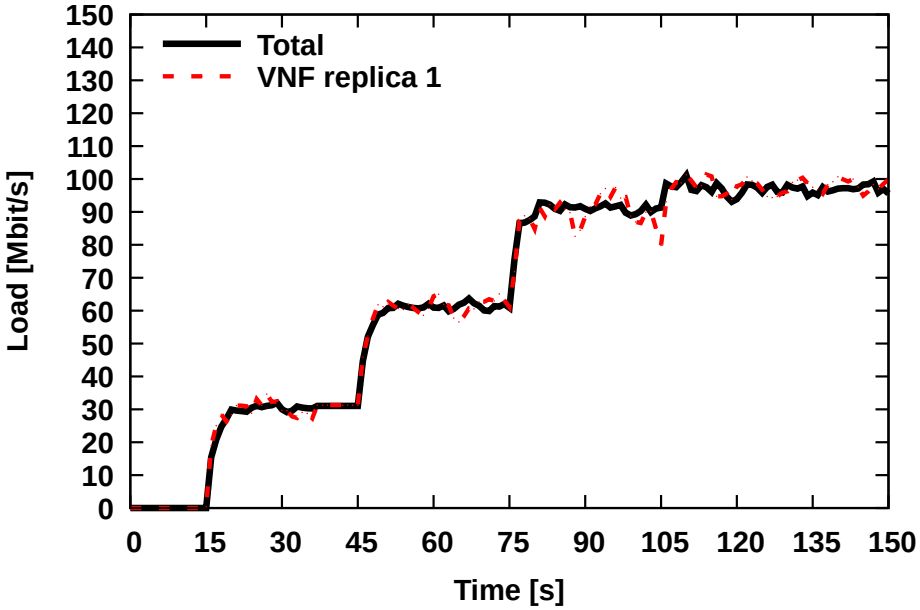


(a) VNF subject to traffic spikes of the same intensity.

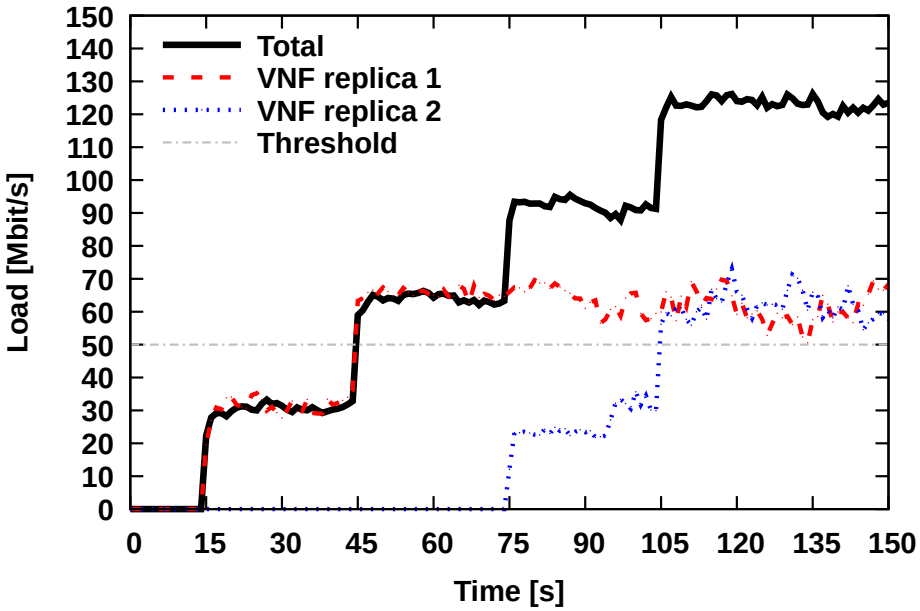


(b) VNF subject to traffic spikes of increasing intensity.

Figure 4.4: Amount of bytes received (line) and percentage of time the CPU is busy (bars) for a selected VNF. In both cases, there is a clear relationship between network and compute load.

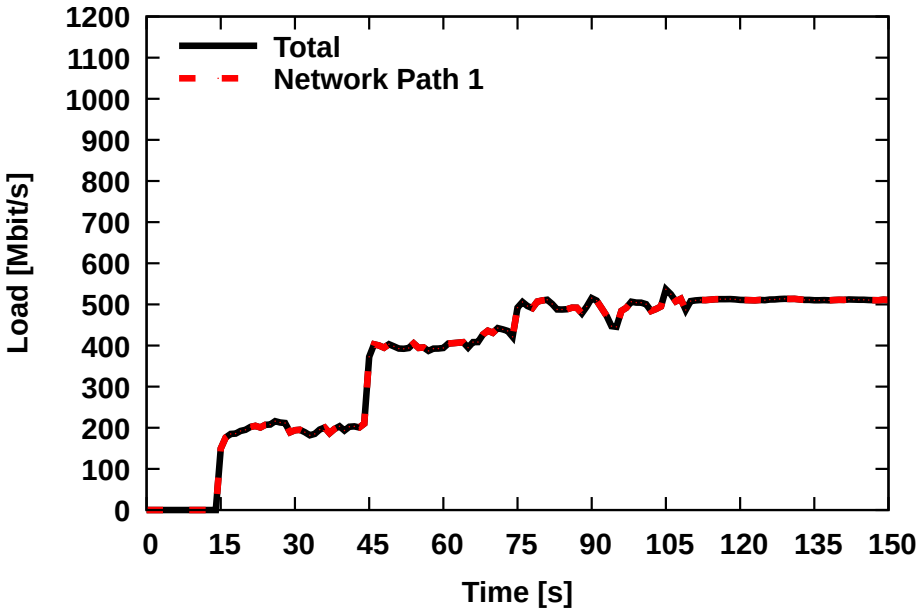


(a) Load of VNFs without traffic steering.

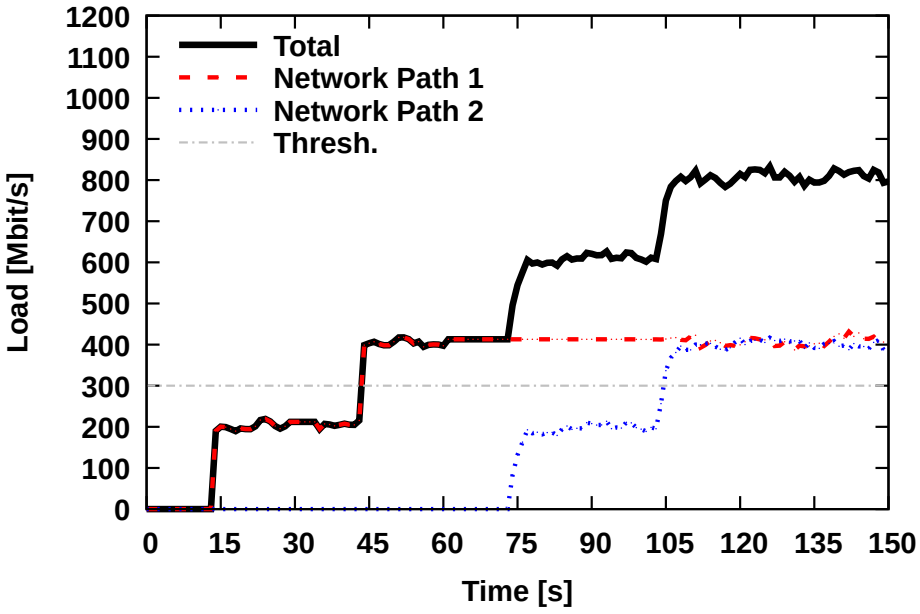


(b) Load of VNFs with traffic steering.

Figure 4.5: Case study: *choose VNF*. Load of VNFs based on the monitoring in NFV domain. Additional VNF replicas allow reaching a higher overall throughput.

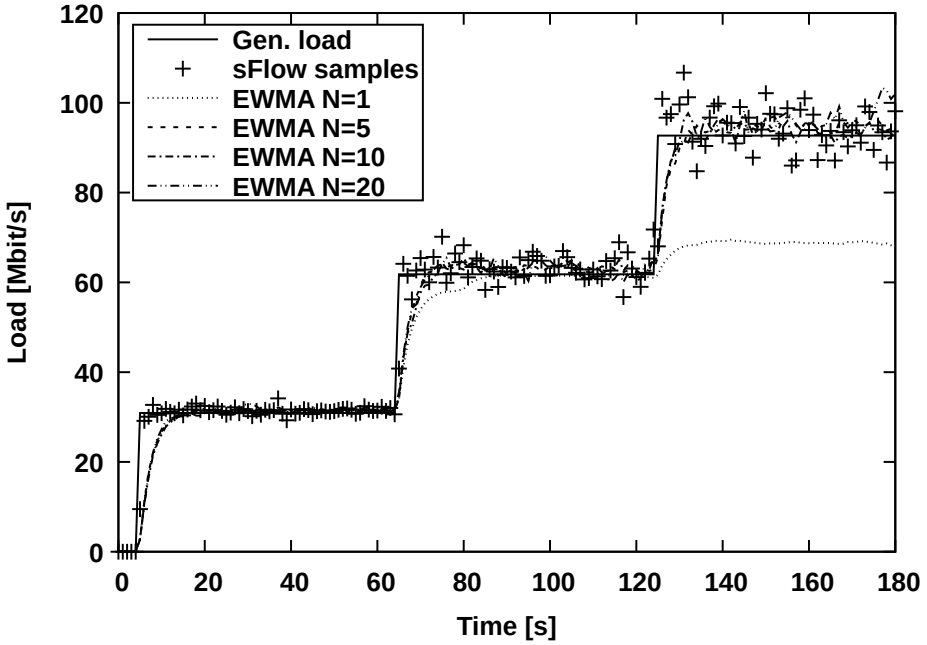


(a) Load of switch interfaces without traffic steering.

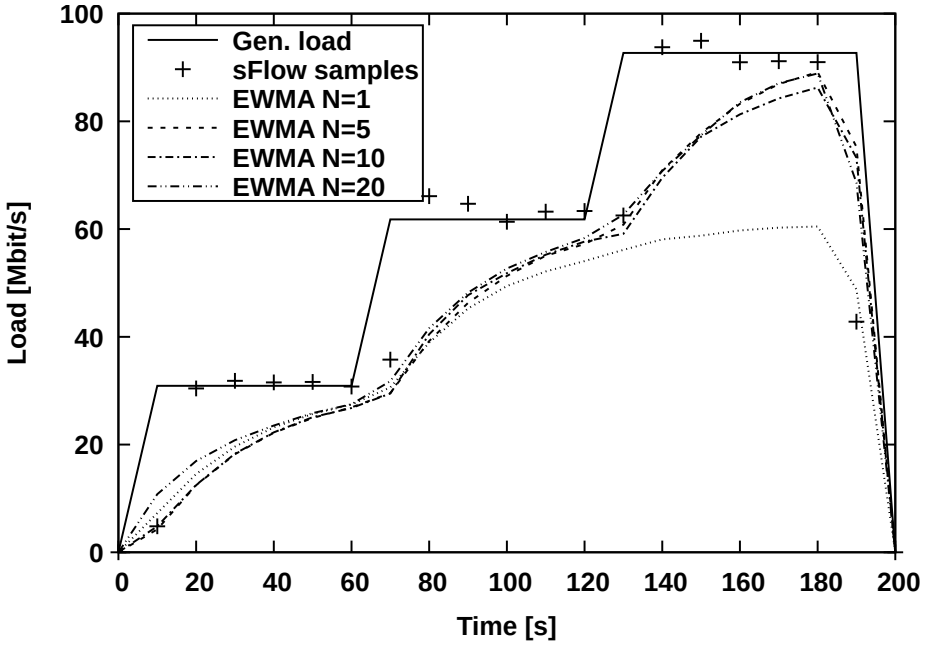


(b) Load of switch interfaces with traffic steering.

Figure 4.6: Case study: *choose Path*. Load of switch interfaces based on the monitoring in SDN domain. Additional paths allow reaching a higher overall throughput.



(a) Accuracy with $C = 1$ s.



(b) Accuracy with $C = 10$ s.

Figure 4.7: Accuracy of the sFlow EWMA under different sampling ratio values N, with $\alpha = 0.3$ and varying C, compared to the instantaneous sFlow samples and the generated load pattern.

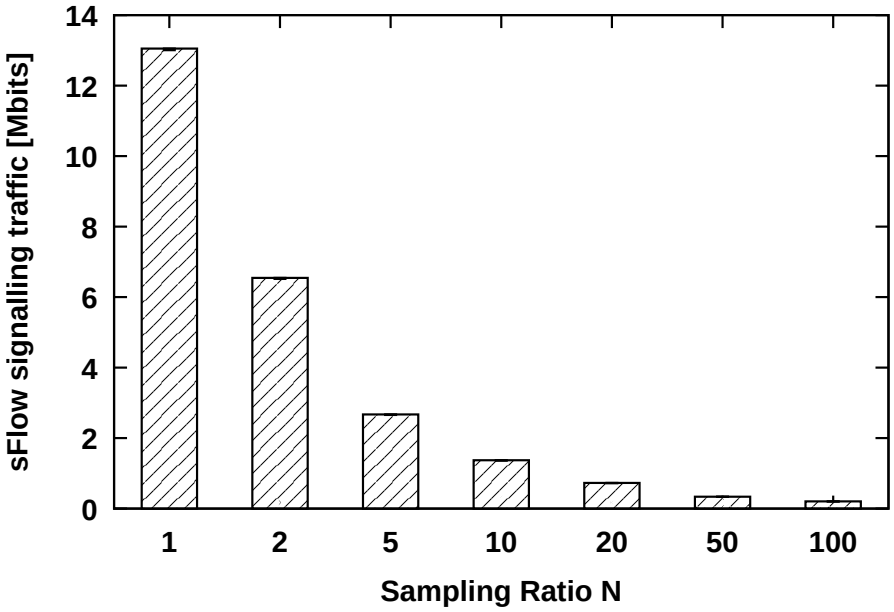


Figure 4.8: Total absolute amount of sFlow signalling traffic during a 1 Mbit/s data transmission lasting for 60 seconds. Increasing the sampling ratio decreases measurement accuracy but also decreases overhead.

Chapter 5

Augmenting software-defined infrastructures

This chapter revolves around the idea that software-based infrastructure could benefit from using currently unemployed techniques to augment their capabilities, achieving mutual independence among different planes, as well as gaining access to a set of functionalities that are inherently specific to such techniques. The ideas and results presented here have been published in [P11].

5.1 Fate sharing and out-of-channel communication

When the Internet was originally designed, the *fate-sharing* principle was used in support to the end-to-end argument for maintaining robustness to network failures: keeping the connection state at the endpoint hosts makes failures always recoverable, unless at least one the hosts themselves fails, thus sharing the fate with the lost connection [B26]. Of course, disruptive events that make the network completely partitioned cannot be recovered either, unless the partitioning is intermittent and suitable delay/disruption tolerant ap-

proaches are adopted, which seemingly do not break the fate-sharing principle [B27]. Fate sharing between data and control planes is also a fundamental principle of IP networks based on legacy distributed routing protocols: as long as two routers are able to exchange *hello* packets or other routing messages, reachability in the data plane is also guaranteed, or at least is assumed [B28].

However, with the advent of network paradigms such as Generalized Multi-Protocol Label Switching (GMPLS) and the previously mentioned Software-Defined Networking (SDN), which advocate the separation between control and data planes, fate sharing is considered as something to be avoided, as data plane operations should not be affected by reachability in the control plane, and vice versa. This is even more important considering the recent trends in network data plane programmability, where also the management plane requires frequent interactions and should not share the same fate as the data plane.

This problem is typically addressed by means of out-of-band management approaches, with network devices being equipped with additional ports to be connected to separate control/management plane networks. However, such a separation is usually applied at a logical level, e.g. using separate service queues and/or VLANs on the same physical network infrastructure. Thus fate sharing persists between the physical medium used by data plane traffic and control/management traffic. Failures in data plane equipment could then prevent also management traffic from reaching the exact network regions at fault, making it impossible to perform key management tasks, such as diagnostics and recovery.

A possible way to overcome the fate sharing problem at the physical level is to adopt physically separate channels for exchanging network management/control traffic [B60], thus shifting from out-of-band to *out-of-channel* control/management. In order to avoid cost and complexity of duplicated equipment and wiring, different *untethered* physical media could also be taken into account, where the term "untethered" is used here as a broader meaning than "wireless". Such untethered media include any wireless connection using the electro-

magnetic spectrum, which can still require expensive infrastructure enhancements [B29], or even the acoustic spectrum, which was recently proved as a feasible and cost-effective means of performing out-of-channel control/management tasks [B35].

Using physically separate channels to control and manage Software-Defined Infrastructures (SDIs) requires some form of coordination between the entities involved in the operations. In particular, a suitable protocol is needed to determine which physical channel — and possibly which sub-channel, if any — must be used to execute some given tasks. Referring to the fascinating case of adopting the acoustic spectrum, there is the need of a management protocol to establish which sounds correspond to which events or actions, or, in other words, the need to define a protocol for the *sonification* of SDI management. In this context, sonification means translating relationships in data or information into sound [B61].

5.2 Sonifying the network

The definition of the sonification protocol is carried out with the following question in mind: *what are the insights that sonified data can offer in network management?*

The most common sonification technique is based on the principle of raising the pitch when a higher level of the quantity under observation is detected [B61]. In this protocol design, the classical notion of sonification is extended and acoustics is considered only as a possible variant, allowing DevOps, network managers and researchers to potentially use different wireless spectra to communicate with out-of-channel signals, enabling network management policy programmability by using combinations of sound waves as well as combinations of other signals outside the acoustic spectrum. These combinations can be used also simultaneously in order to scale. As a use case, and to demonstrate the practicality of providing out-of-channel signals to solve the fate-sharing problem, the protocol to sonify a set of network management tasks.

The problem of obtaining an out-of-channel, non-fate-sharing control and management system has been addressed in the literature. In [B29], a low-latency facility network is proposed, which makes use of low-cost, 60GHz beamforming radio that provides communication paths decoupled from the wired network. Sounds have been used to transfer information in [B30, B32, B33, B31, B34]. Sound can be used to transfer data [B33] but with very limited data rate [B30]. Similarly to [B35], the focus of the work presented here is on the control and management planes.

Specifically, this work focuses on the design of *Music Management Protocol* (MMP), a wireless-signal-based network management protocol aimed at setting up an out-of-channel signaling mechanism with similar goals as those of ICMP and SNMP. The involvement of "Music" in the name comes from the idea of actuating said signals in the audio band. However, the protocol is expected to be able to operate over a number of different wireless channels, spacing from the aforementioned acoustic band to a selection of radio channels (e.g., ISM bands), and potentially exploiting wireless signals in other spectra as well (e.g., the visible spectrum). To demonstrate the potential of the protocol, three sonified network management applications were implemented: TraceSound (to recall a sonified version of traceroute), a Heavy-Hitter Detection application, and a Distributed Denial-of-Service attack monitoring application. A proof-of-concept prototype was implemented employing Scapy [B62], and its functionalities were tested on a virtual network testbed, as well as using real network equipment attached to microphones and speakers. The tests show that all these applications can be successfully recognized, even when their corresponding sounds are played together. Moreover, despite the small scale of the current testbed, which is limited to campus or enterprise networks, the initial tests show that sonification can be an effective technique for vertical out-of-channel management.

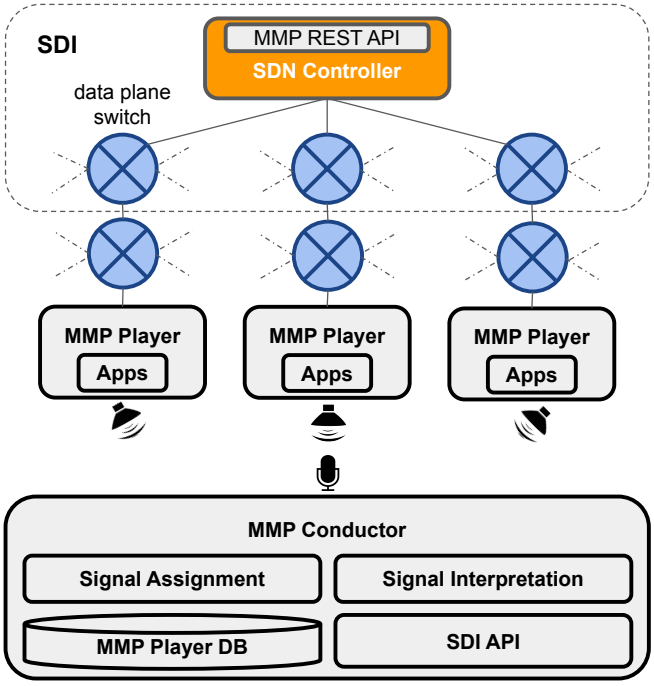


Figure 5.1: Architecture overview: The SDN controller is optional and used only to react to state variations detected by the Conductor.

5.3 Sonification architecture overview

The set of fundamental mechanisms necessary to control (or offload the control of) an infrastructure with out-of-channel signals or sonification are presented in this section. Figure 5.1 gives an overview of how MMP augments an SDI with out-of-channel vertical network management capabilities over multiple spectrum slots. The goal is to clarify what are the mechanisms (invariances) for sonification, explaining who does what, while giving some examples of possible implementations and policies.

There are two entities acting in the proposed architecture: *MMP Conductor(s)* and *MMP Player(s)*. Similarly to the choice of the name of the protocol, these names were chosen having in mind the implementation in the audio spectrum, thus the names borrowed from the music lexicon. Despite this, the sonification can be ported to other spectrum, including ones inaudible by humans.

The main (logically centralized) brain of the architecture lays with the (MMP) Conductor. This component is equipped with a receiver set in the alternative physical channel (in case of sonification, one or several microphones), as well as with a logical connection with the SDI to apply any required actions. The Conductor may, for example, merely invoke `iptables` commands, or can be connected to an SDN controller to perform complex network programmability tasks.

The Conductor has two main components: signal assignment and signal interpretation. The signal assignment component keeps track of the n-to-n mapping between applications (or management tasks) and frequency set. The signal interpretation component functionalities concern the sound detection; this component also implements the actions that are optionally pushed to the managed SDI via the SDI API Component shown in Figure 5.1. Finally, the Player DB component is a logically centralized database that stores the signal assignment mapping and other states. This piece acts as a Management Information Base.

In order to augment network devices with sonification or other out-of-channel wireless capabilities, a (software or hardware) component needs to be installed on them. In the reference architecture, this component is called (MMP) Player. Each Player must be first configured by the Conductor in terms of physical channel to be used, relevant signals on that channel, and their meaning. Then, when some of the configured conditions are met, the Player needs to encode the corresponding signals and transmit them to the Conductor on the configured physical channel. This way the Conductor is alerted of the specific events/conditions happening on the device and can optionally interact with the SDI accordingly.

In the prototype implementation, a REST API was used to send commands from the Conductor to an SDN controller. Examples of states that can be transferred include network load condition, or path traversed by a specific flow, which can help the SDN controller to optimize the decision tasks on how to properly steer data traffic. For instance, when the SDI is implemented as an OpenFlow-based net-

work infrastructure, the device enhanced with an Player can be an OpenFlow-enabled switch or an OvS virtual switch [B63]. In that case, the sonification logic may be implemented as a set of OpenFlow rules to be installed/changed/removed based on application that has a speaker attached to the switch via a single-board device such as an Arduino [B64] or a Raspberry Pi [B65].

5.4 Management object model and sonification workflow

The network management literature [B67, B68, B66] reports that, historically, a management object model is composed by three elements:

i) a set of objects, whose attribute inform the management entities about what states need to be managed;
ii) an interface (for example an API) to allow object attributes change locally
iii) a management protocol, to change the object attributes remotely.

Object attributes are material for standardization documents, no details on that are reported here; also, the presented API architecture design is fairly simple, as just a simple REST API is employed. In this implementation, the communication between the Conductor(s) and the SDN controller happens via a REST interface, obtainable either as a customization of the SDN controller's pre-existent REST interface (as in the case of Ryu [B55] or ONOS [B43]), or by deploying a simple REST server as an application of the SDN controller. The details of this protocol design can be used in future contributions attempting to sonify other network management applications or to implement other sonification techniques.

The following is an overview of the main tasks involved in exchanging sonified data for network management and its workflow, i.e., the interactions among all managed entities.

The main task of the proposed protocol is to exchange management object states; an example of those states are (application or network) configuration items that are exchanged with the managed entities de-

ployed in the physically separate management plane, or *Music Protocol (MP) plane*. The configuration happens via a packet-switched network, but when the entities are configured, they will exchange information in the form of wireless signals. Conductor(s) and Player(s) act in the MP plane. In a common application, every Player is connected to a switch, and monitors a sample of the traffic passing by it. In the presented SDN-based implementation, this is achieved by mirroring a sample of the traffic transiting on data plane ports towards the ports to which the Player is connected. The Players are configured to execute a set of applications, that instruct the switches on how to react to specific observed traffic conditions, or to hardware (failure) events. Every Conductor is in charge of assigning alphabets signal to Players. It then receives and interprets signals from them, and communicates with the "primary" control plane entity (e.g., the SDN controller). The information collected by the Conductor from the Players is then used to provide the SDN controller observations that can help gain quicker insights and determine what are the best (network or security) policies to deploy, based on the current network status.

But are the Conductor and the SDN controller both necessary? The architecture of the Conductor presented in Section 5.3 is agnostic from the SDI infrastructure and hence from the presence of an SDN controller. Therefore, in general, Conductor and SDN Controller are not always simultaneously needed, but in the current implementation both are employed. The idea is not to make us of sonification as a replacement for an SDN controller, but merely as a helper, to augment or offload part of its functionality with an out-of-channel management plane. Also, as a remark, *out-of-channel* is different from *out-of-band*.

5.5 Protocol design

Since the goal of the protocol is to establish a common ground for (untethered) communication among Conductors and Players, the protocol design needs multiple phases, all described in the following sub-

Type code	Type ID	Type description
10	PLHELLO	Setup - Player Hello
11	CNHELLO	Setup - Conductor Hello
12	CHSUGG	Setup - Player Channel Suggestion
13	SIGNASS	Setup - Conductor Signal Assignment
14	ACKSIGN	Setup - Player Signal ACK
20	PHYCH	Physical Technology Change
21	ACKPHY	Physical Technology Change ACK
30	END	Close
31	ACKEND	Close ACK

Table 5.1: MMP packet types

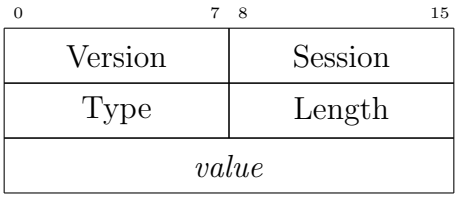


Figure 5.2: Generic MMP packet format

sections. Table 5.1 shows the set of possible message types that Conductors and Players may exchange. Each MMP message is based on some common fields (such as version of the protocol and session ID) followed by a message-specific structure in the type-length-value format, as shown in Figure 5.2.

5.5.1 Connection setup

The objective of the setup phase is to initialize the communication between a Player and a Conductor, agree on one or more physical technologies to use (for example acoustic and Bluetooth), and assign signals to the Player. Every Player maps signals to applications in the same order as they are specified by the Conductor (first signal assigned to first application, second signal assigned to second application, and so on). This, of course, is an easily editable assignment

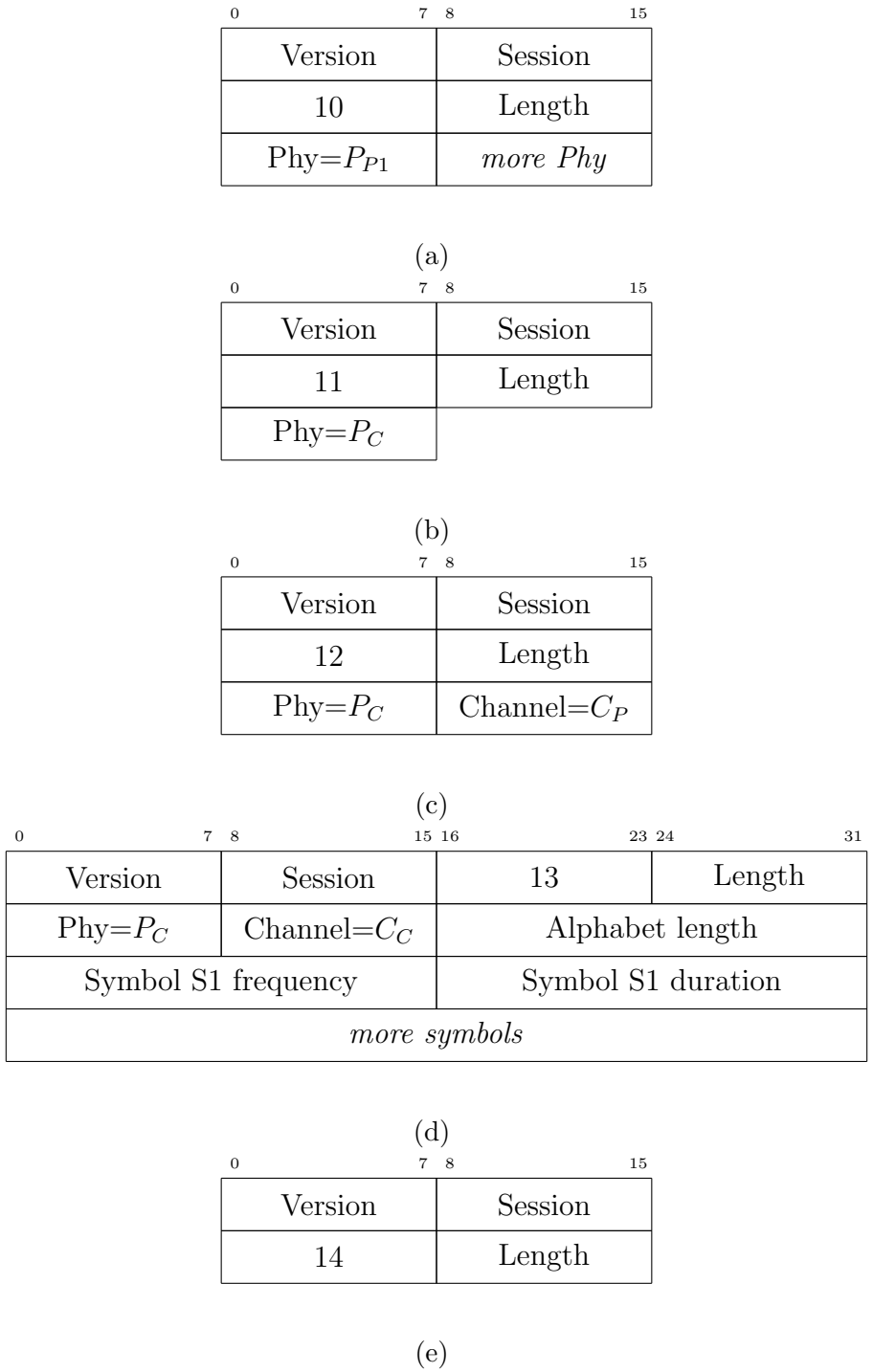


Figure 5.3: Setup phase packets: (a) PLHELLO, (b) CNHELLO, (c) CHSUGG, (d) SIGNASS, (e) ACKSIGN

policy. The design of the setup phase was inspired by the Dynamic Host Configuration Protocol (DHCP) messages. Instead of acquiring a new IP address, the "hosts" (in this case the Players) acquire a new frequency set to operate on for the duration of the sonified application.

The Player initiates the communication (Fig. 5.3a) by notifying the Conductor of its presence while also informing the latter on the version of the protocol and the physical channel technologies it supports ($P_{P1}, ..., P_{Pn}$). This is a discovery phase. The Conductor then generates a session ID for the Player, used as an identifier for the duration of the session (from setup to close). The unique session ID is present in every packet exchanged between the Conductor and the specific Player, so it could act as a sort of address field for filtering. The Conductor then responds to the Player (Fig. 5.3b) with a message informing the Player on the session ID, the agreed version of the protocol and the chosen physical layer P_C. The Player acknowledges the physical layer type choice by replying back to the Conductor with such a choice in a packet (Fig. 5.3c), specifying a channel proposal C_P to use. This proposal is based on sensing of the environment. A specific sub-channel might be more crowded, given its proximity with other devices using similar bands. The Conductor confirms the chosen channel C_C, that might differ from the initial suggestion C_P by the Player while also sending the list of signals the specific Player must use and their time duration (Fig. 5.3d). Finally, the Player acknowledges the reception of the "alphabet" to the Conductor with a message of type ACKSIG (Fig. 5.3e).

5.5.2 Physical technology adaptation

This phase of the protocol is designed to allow an Conductor to modify the physical access technique of an already registered Player without having to go through a new setup phase. The Conductor sends the Player a message with the indication of a specific physical technology P. The Player acknowledges it by replying with a message of type ACKPHY. The order in the communication is kept with the addition of a sequence number, so as to avoid confusion in understanding what

is the most updated physical technology to use.

5.5.3 Close

This phase of the protocol is designed for deallocation. With this phase, the Conductor relieves a Player of its sonification duty, instructing it to free the signals assigned to it. To do so, the Conductor sends the Player a message of type END, specifying the mapping to destroy: for example, free up channel C_E from physical layer P_E. This implementation also supports multiple deallocation with a single message by specifying a predefined code. The Player acknowledges the close command with a message of type ACKEND.

5.6 Sonification of network management applications

This section describes a few examples of network management applications that were implemented on the SDN-based sonification testbed.

5.6.1 TraceSound

The implementation of *TraceSound* was based on the concept and use of the known Linux command `traceroute`. Tracesound enables (not formal) network state verification, debugging and entitlement. With TraceSound, a Player emits a signal when an ICMP Echo Request (i.e., `ping`) traverses its monitored switch. TraceSound can be used to verify which switches are traversed by which traffic flow, or which flow is departing from a specific source and is routed to a specific destination. To obtain this information, usually an expensive log analysis is required a posteriori, or some state tracking is necessary [B69]. By starting a `ping` session between endpoints in the data plane, the Conductor receives TraceSound signals and can trace which switch was or was *not* responsible for routing a specific flow. A specific sound can be mapped to every Player allowing a path reconstruction of the sequence of switches traversed by the ping even for network debugging

purposes. This application is easily extensible to react on the transit of packets different than the ICMP Echo Request, such as BGP route advertising or headers belonging to unsafe SSL/TLS versions.

It is well known that OpenFlow flow tables are unsafe with respect to managing conflicting rules [B70, B71]. A network manager (system) may need to be informed on the percentage of traffic that traverses a specific set of devices, or rather that does not traverse another subset of devices. With complex network configurations involving large amount of SDN-enforced steering policies, a new steering rule might interfere with an existing one, causing the older policy to not be properly enforced. The TraceSound application can be used to verify an actual path before and after the injection of a new rule: the sequence of switches traversed between two customizable endpoints can play different sounds if an accidental rule change has impacted the route. Furthermore, a switch may be configured to play alarming sounds when it is called to forward a packet without being entitled to do so, given some internal BGP policies in place.

5.6.2 Heavy-Hitter Detection

Heavy-hitters (sometimes they are referred to as elephant flows) are flows that carry a large quantity of data, defined by a threshold usually related to a fraction of utilized link capacity over a fixed period of time. Identifying them is crucial to network management applications such as load balancing. As a use case, an MMP application called Heavy-Hitter Detection (HHD) was developed, to help identifying heavy-hitters. Players emit a signal when they observes a high amount of traffic (determined by a threshold, set as a parameter) going through the monitored switch, so as to make the Conductor aware that the switch may require priority to manage all the traffic.

5.6.3 DDoS Monitoring

An important task in network security is the identification of potential Distributed Denial-of-Service (DDoS) attacks. To this aim, it

could be useful to identify hosts that contact a large number of other hosts in a suspiciously short time-frame. A host that contacts at least k unique destinations over a given interval of time is called a k-superspreader [B72]. This feature was implemented in DDoS Monitoring (DDoSM) by tracking flows that have the same source but different destinations, and making a Player emit a (sound) signal when it observes the transit of more than k such flows over its monitored switch.

5.6.4 Callbacks for network management application programmability

In any software engineering context, callbacks are programming tools used to detect and respond to the occurrence of a given hardware or software event. Such callbacks could represent link failures, connection requests from a new host, connectivity between a host and a switch, or a switch reboot. With MMP, Players can be configured to emit signals when they observe such events. However, this requires Players to have a deeper point of view on the switches than the one allowed by simply monitoring the traffic traversing them. This abstraction is not implemented in the Proof-of-Concept prototype, but left for a future work, including the implementation of specific and customized sonification events. For example, a open question is how to implement support for sound-based, out-of-channel SNMP traps [B73]; a Player could emits a signal if it observes a specific event mapped to an SNMP signal: *coldStart*, *warmStart*, *linkDown*, *linkUp*, *authFailure* or *egpNeighborLoss*.

5.7 Testbed and implementation

The protocol and use case applications were tested on both a real network testbed and a virtual network testbed. The local virtual network testbed is composed by a number of data plane hosts, interconnected by a mesh network of switches, each connected to a Player.

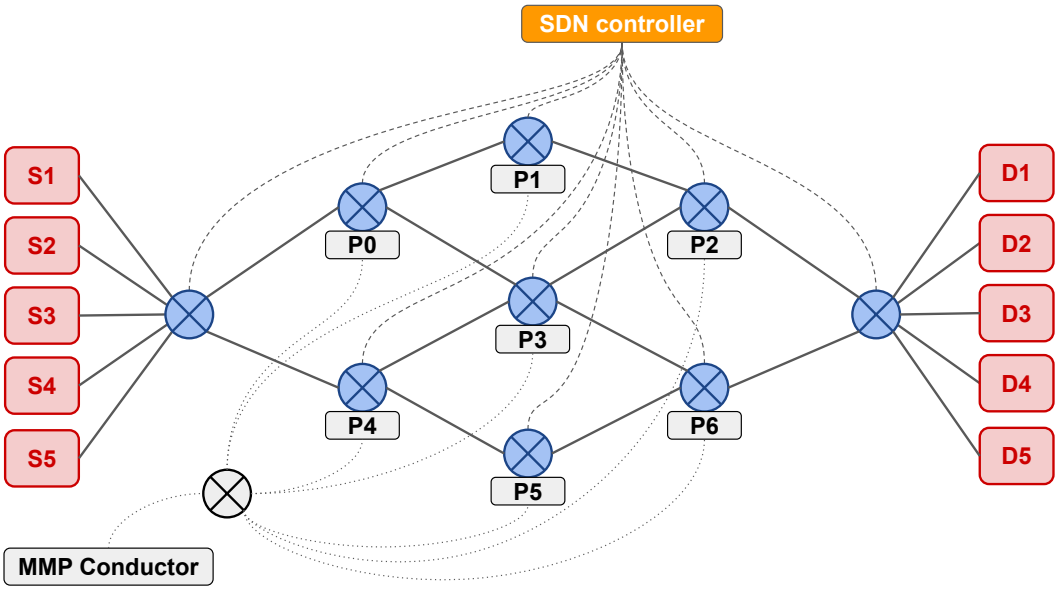


Figure 5.4: SDN-based virtual network testbed. Every switch is connected to both an Conductor and and SDN controller, managing the network over two separate independent channels.

Every Player is, in turn, connected to a Conductor via a separate dedicated network. A representation of the testbed is shown in Figure 5.4. Each host and MP plane entity is emulated by a Linux network namespace [B74], while virtual switches are deployments of OvS instances [B63], and are managed by the Ryu SDN controller [B55]. Every virtual switch is configured to mirror all traffic traversing it to the MP plane port attached to the Player. In this testbed, by design, data plane hosts are unaware of MP plane components. All protocol packets are implemented with the Python Scapy library [B62]. This testbed enables the testing of each phases of the protocol and, to a certain extent, the scalability of the approach. It is crucial to note that sonification is not envisioned for large scale datacenters, but that it can be used within a smaller-scale enterprise networks as well as in a single point-of-presence, for example in a region of a campus network.

The real network testbed is instead composed by Raspberry Pis [B65] and inexpensive speakers and microphones. The real testbed was used

to assess the feasibility of sound signaling. Specifically, an audio track was generated in the virtual testbed, then played using a speaker, so as to verify that sounds emitted by Players would be correctly recognizable by the Conductor.

5.8 Evaluation

Signal	**Application**	**Player P0**	
S1	TraceSound	400 Hz	1000 ms
S2	Heavy-Hitter Detection	500 Hz	1000 ms
S3	DDoS Monitoring	600 Hz	1000 ms

Table 5.2: Signal-to-application mapping

This section presents the proof-of-concept evaluation results obtained by deploying the protocol on the testbed described in Section 5.7. The sound traces are obtained by recording the acoustic signals from the virtual network testbed. The traces are then processed on the physical network testbed, so as to verify the correct interpretation of signals in real settings. Every Player is running three applications, namely, TraceSound (TS), Heavy-Hitter Detection (HHD), and DDoS monitoring (DDOSM).

In this evaluation scenario, every Player requires an assignment of three signals. Table 5.2 shows the assignment of signals to applications. The first signal assigned to every Player is associated to TraceSound, the second to the Heavy-Hitter Detection application and the third to the DDoS monitoring. Table 5.2 also reports the actual values of frequency and duration of the three signals assigned to Player P0 (identifiable on the top-left corner of the network of switches in Figure 5.4). Note how frequency and duration are the two characterising policies of a sound signal. Together with amplitude, such policies can be tuned for network management sonification programmability.

In general, when multiple applications are executed on a number of Players simultaneously, signals overlap in an asynchronous way. Fig-

ure 5.5a shows the magnitude of the frequency components found in the generated audio trace, while Figure 5.5b shows a mel-scaled spectrogram of the signals emitted by the Player in the virtual testbed when a variety of events occur. The Conductor is aware of the list of signals that every Player is currently using and of the signal-to-application mappings. The Conductor hence is able to determine the source and meaning of every signal upon reception. Note in Figure 5.5b the two almost-simultaneous groups of three signals (around time $t = 1.5s$ and $t = 6s$). These are caused by two running TraceSound sessions. Those signals are emitted by the three Players whose switches are traversed by the traffic between the two endpoints of the ICMP traffic. Among them, the signal emitted by Player P0 is recognizable at frequency $f = 400Hz$. The signal starting at $t = 3s$ instead is caused by the detection of a potential elephant flow in one of the switches (Heavy-Hitter Detection application or HHD). The signal that follows, starting at $t = 3s$, is generated upon the detection of a large number of different traffic sources in another switch.

Note how the system also recognized, starting at time $t = 4.5s$, the HHD signal coming from Player P0, at frequency $f = 500Hz$. Finally, starting at around $t = 7.5s$, there is the DDoS monitoring signal of P0. Table 5.3 gives an overall view of all detected signals in the frequency domain, captured using the Fast Fourier Transform (FFT).

5.9 Limitations and open questions

While the presented object model and protocol design can be generalized to the policy programmability of many wireless physical layer parameters, the implemented testbed and experiments have been limited to sonification. It is well-known that the acoustic spectrum has limited capabilities: first, sound speed is fairly slow compared to other media; second, even at a small-medium scale, interference appears to be harder to manage with respect to other signals, given the inability to modulate and protect the signal-to-noise ratio. Surely acoustics can be restricted solely to the last hop of a communication, but that

Time [ms]	Frequency [Hz]	Player	Signal	Application
1034	400	P0	S1	TS
1040	1300	P3	S1	TS
1042	2200	P6	S1	TS
3021	1600	P3	S2	HHD
3529	2400	P6	S3	DDOSM
4754	500	P0	S2	HHD
5694	400	P0	S1	TS
5650	700	P1	S1	TS
5653	1000	P2	S1	TS
7394	600	P0	S3	DDOSM

Table 5.3: Signals detected during the evaluation, as shown in Figure 5.5.

does not necessarily solve the fate-sharing problem. In summary, it is arguable that the acoustic spectrum in isolation can be considered a viable and scalable form of network management signaling only for a very specific applications that do generate low-amount of control traffic. Moreover, sound can be used very specifically on fairly small networks or applications even across multiple hosts.

There are also sound insecurity limitations. Acoustics has the same drawbacks as any other wireless signal. Messages sent out-of-channel could perhaps be considered as conveyed through a (secure because) hidden channel, assuming that attackers are unable to tamper or listen to such signals. One could also also envision the possibility of using sound as two-factor authentication system: a "proof-of-sound" could be requested to MMP-enabled processes.

An additional research open question is how to use this protocol for intent specification of a networked system. An intent, as described in Section 1.2.5, is a high-level specification that allows to declare service policies rather than a specific network mechanism. Typically, an intent is perceived as a form of specification that a user or a user-level application can formulate. Sounds or other out-of-channel means

of expressing the intents in a more natural way can be explored. For example, a higher-hierarchy entity could specify an intent by means of a sound, and the underlying structure, knowing the mapping, can carry out the required actions so as to apply that intent. In the other logical direction, a device may emit a sound meaning "this switch is overloaded", and the network controller can view that sound as inverse function of an intent declaration, expressing the need to have the traffic re-steered to relieve that particular switch.

Another interesting case study is that of vibrations, that can be employed as a form of out-of-channel signal, opening up to a set of interesting possibilities, some of which are introduced in the following section.

5.10 Haptic networking

Vibrations and haptic technologies are being explored in end-user (wearable) devices, and Tactile Internet is being used merely as a metaphor. However, with rare exceptions and for smaller scoped projects, vibration has been largely untouched as networking communication media.

A growing movement has brought focus to the need for making computer science education accessible to all individuals, particularly to those of the blind and visually impaired communities. With many graphical and visual issues within networking, these individuals are often left behind. The protocol presented in this Chapter, and most importantly the philosophy behind it, can be extended to vibrations, providing a flexible medium with potential to enable new applications, both in the direction of of out-of-channel augmentation of networking infrastructures, similar to those described in Section 5.6, and towards new ways of teaching networking to visually impaired individuals.

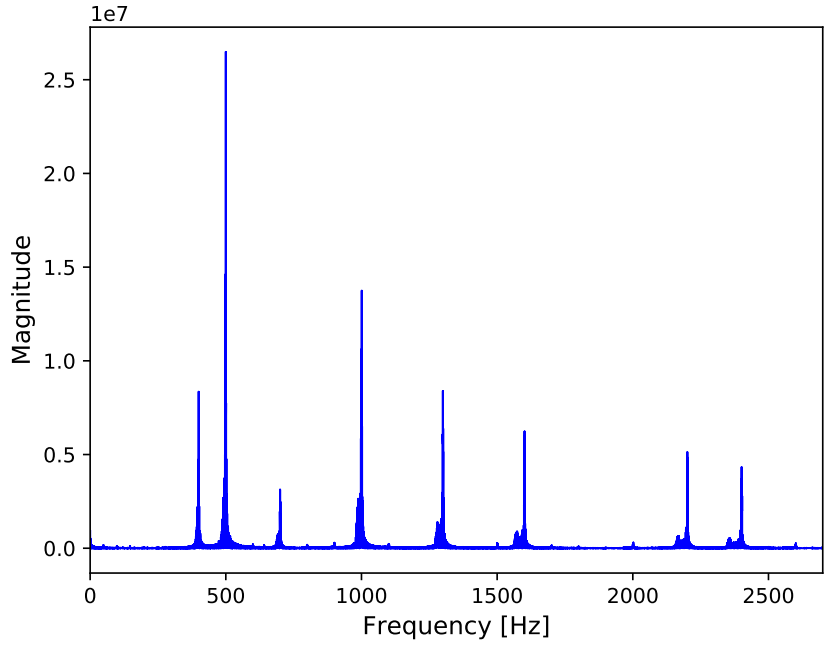
Touch is an important component of learning, particularly to those who heavily rely on touch as a primary communication channelFrom hands-on learning experiences to the use of force feedback devices in virtual learning, there are many instances where touch has demon-

strated its use in learning abstract concepts [B75]. Quorum, an evidence-oriented programming language, specifically has an "auditory" track that enables individuals with VI to program [B76]. The association of network events to tactile feedbacks provides an avenue by which networking principles, which often employ visualizations, may be taught in a multi-sensory way — catering to individuals with different learning styles or disabilities. To this aim, a low-cost starting point for physical layer programmability, built with common materials, is introduced in [P15]. The work lays the foundation for "Vibration-Defined Networking", and suggests potential uses of this (elsewhere explored) technology for physical layer security, to increase network resiliency and for inclusive educational purposes. To assess the practicality of the approach, an architecture for vibration programmability is designed and evaluated, sharing the experience obtained building several hardware testbeds. Different mechanical components to handle vibrations are analyzed. The work exposes some limitations, but also interesting potential research directions.

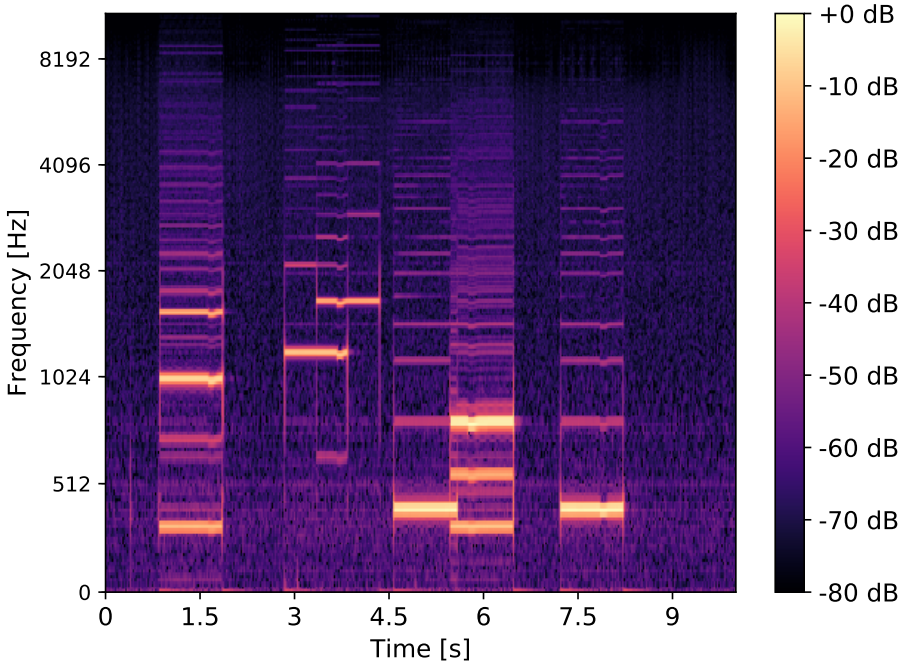
5.11 Remarks

The proposed network management protocol enables programmability of out-of-channel management applications, with the original goal of making Software-Defined Infrastructures more resilient, then extended to other uses case to demonstrate the practicality of the approach. This first implementation used the acoustics spectrum, a fairly unexplored physical layer to be used as non-fate-sharing control and management planes. Some insight on the use of vibrations are given.

Several applications can be recognized by managed nodes running this architecture, recognizing signals from all of them, showing how out-of-channel signals can be used for example for network verification and debugging.



(a) Frequency components represented with Fast Fourier Transform (FFT).



(b) Evolution of signals in time.

Figure 5.5: Sounds generated by multiple applications.

Chapter 6

Flexible service provisioning in Fog scenarios

As anticipated in Section 1.2.3, sometimes the Cloud is not able to offer the responsiveness required by new generation applications. Here is where the Fog comes into play. With Fog Computing, services are brought closer to the user, making it possible to achieve reduced service response times, and also offloading the Cloud infrastructure. This, however, comes at the price of increased complexity in the deployment, management and monitoring mechanisms that support the provisioning of this services, even more so when trying to preserve the XaaS model for service provisioning offered by the Cloud. This is why an orchestrator is required, and one such system, designed and developed from scratch, is presented in the following.

FORCH (Fog ORCHestrator) was designed to help dynamically allocating and managing resources in order to provide for different needs of the end users. It aims at facilitating Fog XaaS discover available Fog nodes, track offered resources and services, monitor utilization, listen to service requests from user, allocate services to node, mediate between end users and the Fog infrastructure. The FORCH architecture is composed of a number of software modules, each offering its API, enabling the flexible deployment of different services implemented through any of the XaaS models, benefiting both the infrastructure and the end users. The design, development, imple-

mentation and validation of FORCH is still being carried out, with descriptions of the current components and relevant experimental validation published in [P13] and reported in the remainder of this chapter.

6.1 Motivation and challenges

Dynamicity is inherently a fundamental characteristic of the Fog philosophy. The amount of resources available to a Fog network varies in relation to the quantity and the capacity of the Fog nodes that are available at a given moment, and can therefore be allocated to end users to provide the service they request. Moreover, the capabilities and the amount of resources offered by a Fog node are not known a priori, i.e., before the node itself connects to the Fog infrastructure, as Fog nodes are meant to offer their resources to the infrastructure at non-predetermined moments in time. In order to handle such a dynamic and heterogeneous set of Fog Computing resources, a suitable orchestration component is needed [B37, B36]. Such orchestrator should be able to discover how many and which Fog nodes are available, keep track of the resources they offer and their utilization, listen to service requests from the end users and decide which node must be allocated to them, if feasible, then proceed to instructing both the end user and the infrastructure on said allocation, allowing the end user to access to the allocated resources and obtain the requested service.

Borrowing from the XaaS Cloud Computing service model classification introduced in Section 1.2.3, the analysis focuses on the general situation where a Fog Computing service can be categorized in different classes suitable to satisfy different end user needs, with different levels of flexibility. In the Software-as-a-Service (SaaS) case, end users typically require a predetermined application to ingest some data, perform computations or evaluations on it, and return a result. In the Platform-as-a-Service (PaaS) case, instead, end users want to be able to develop their own applications on a given software platform. Finally, in the Infrastructure-as-a-Service (IaaS) case, end users need to

have a portion of computing and network resources allocated to them, and a concerted way of installing their own platform and software on it. It is therefore evident the need for a Fog Computing architecture that is aware of these different service models along with their related peculiarities and challenges in terms of resource allocation, usage and monitoring.

6.2 Fog computing system architecture

FORCH is a modular orchestration system for Fog Computing infrastructures, aware of the different service models, and able to act as a resource management layer placed between the end user and the infrastructure itself. Such architecture has not yet been proposed in the literature. In fact, for example, in [B38] a framework integrating different IoT-enabled systems and the Fog/Cloud infrastructure is proposed, with a focus on important aspects such as security, but said framework only implements the PaaS class of service, without addressing the SaaS and IaaS scenarios. Also, in [B37] a Fog orchestration scheme is proposed specifically for IoT scenarios, whereas in [B36] the main Fog orchestration challenges and possible solutions are discussed. However, these works do not include testbed implementations and do not address different service models for enabling a multi-purpose application scenario, i.e., not necessarily limited to the IoT.

Figure 6.1 shows the architecture of the Fog system, highlighting its logical components. The supervising entity is the titular orchestrator, FORCH. It coordinates the activities in the Fog system, interacting with the users by providing them information on the available services offered by the Fog infrastructure, and receiving their requests for new service allocations. FORCH also interacts with the Fog nodes to manage services deployed on them, as well as with repositories in the Cloud, to gather information on the available platforms and software tools. FORCH has multiple components, each developed as an independent module:

– **User Access** (UA): the point of contact for users to interact

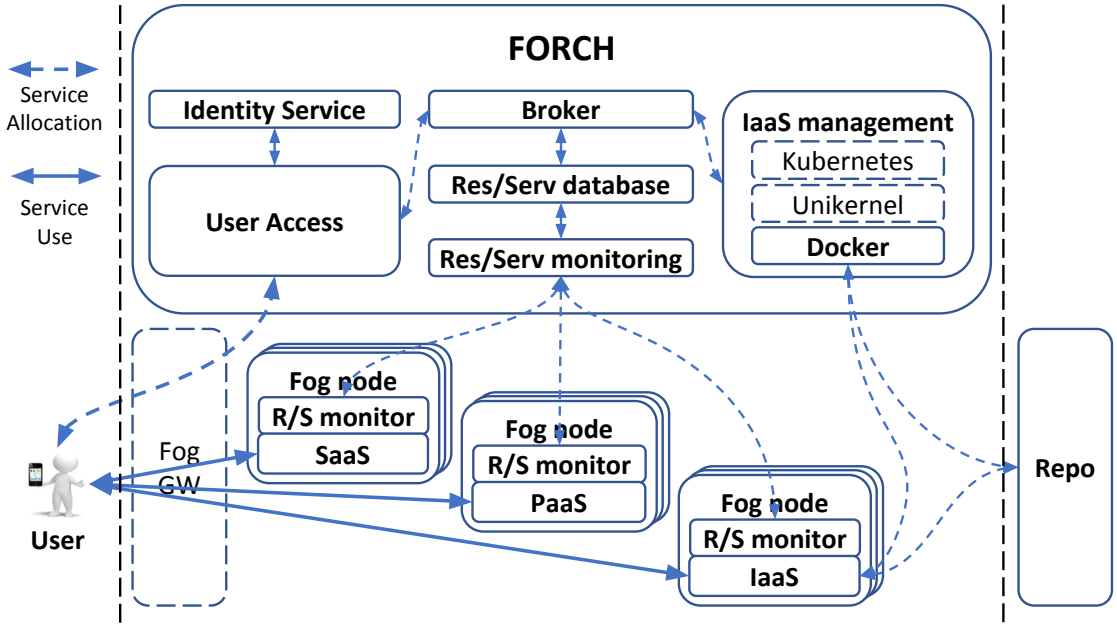


Figure 6.1: Reference architecture.

with the FORCH;

- Broker (BR): serves as mediator between the user and the resource management modules of the FORCH, by routing user requests to the appropriate component of the FORCH, and routing their responses back to the user, facilitating the allocation and management of services in the Fog system;
- **Resource and Service Database** (RD): stores information on the resources of the Fog system and their current state, in terms of availability, residual capacity, and current services deployed on them;
- **Resource and Service Monitoring** (RM): provides the database with information on the Fog nodes by acting as collector for the monitoring data sent over from the agent modules in the Fog nodes;
- **IaaS Management** (IM): manages resources allocation and service deployment on IaaS Fog nodes, taking advantage of multiple lightweight virtualization technologies (e.g., containers, Unikernels) and interacting with different related image repositories

and management/orchestration platforms (e.g., Docker, Eclipse fog05).

The end user, connected through a Fog Gateway (FG), is only allowed to interact with the UA module through its REST API, which exposes endpoints for functionalities that the user is allowed to access directly. Most functionalities of the FORCH, each of them exposing a REST endpoint on the relevant software module, are for internal use only, although they are triggered by the requests of the user.

Fog nodes, regardless of their capabilities, host a module acting as an agent of the RM collector module of the FORCH. Through this agent module, the RM can acquire information regarding the services the Fog node can offer along with its residual resources, and monitor its activities once a service is deployed on it. As a particular subset of Fog nodes, IaaS nodes are able to interact directly with the software and platform repository of choice in order to download the software or image needed to provide the service that FORCH wants to deploy on them.

Based on the classification of the different services the Fog infrastructure can provide, three types of entities are distinguished among those FORCH can allocate Fog nodes to: **Applications** (APPs) in the SaaS case, **Software Development Platforms** (SDPs) in the PaaS case, and **Fog Virtualization Engines** (FVEs) in the IaaS case. All of them being software entities, they address mutually distinct sets of end user needs with different levels of flexibility. APPs take values as input and return results based on those input values (e.g., a block of a computationally-intensive series of operations), or listen for incoming requests and serve them (e.g., a Web-based application). Diversely, SDPs are meant to accept blocks of code written in a predetermined language and/or using specific development libraries, execute them, and return the output to the end user. (e.g., Remote Java or Python Interpreters). Lastly, FVEs allow a Fog node to host virtualized appliances so that the end user can deploy its own virtualized system and have it perform the task it needs with maximum flexibility.

```
User -> O-UA
        O-UA -> O-RD (GET /appcat)
        O-UA <- O-RD
User <- O-UA
```

Figure 6.2: Sequence diagram for `GET /apps`.

6.3 A use case

The term *service* represents either an APP, a SDP of a FVE, regardless of the technology the infrastructure is able to offer that service to the user. At the end of each successful service allocation, the user is given a symbolic and unique identifier of the node the requested service has been allocated on, along with a service port where the allocated service is listening. The service can be accessed through an predefined interface, specific for the service and known to the user. The interaction between the user and the service happens through the Fog Gateway, which is in charge of the translation of the symbolic node identifier to a valid locator (e.g., an IPv4 address), making the user oblivious of the network details inside the Fog infrastructure. Furthermore, the FG is in charge of coordinating the authentication of the user to the system, as well as of the encryption of the communication. In the remainder of this section, the use case of a user requesting an APP is considered and described.

Before requesting access to an application, the user can retrieve the APP catalog, i.e., the list of applications the Fog infrastructure is able to offer, with a call to the method `GET /apps`, represented in Figure 6.2. The catalog summarizes the registered applications, which does not necessarily mean those APPs are already deployed in the infrastructure at the time of the retrieval of the catalog. Every application is identified by an *app_id* associated to it.

The user can then request access to a specific APP, by means of the method `POST /app/<app_id>`, where the identifier *app_id* corresponding to the desired APP is the one found in the APP catalog. The request triggers the sequence of calls represented in Figure 6.3,

where the acronyms used start either with O- or N-, representing the (Fog) Orchestrator and (Fog) Node respectively.

- the user makes the request to the O-UA;
- the O-UA forwards the request to the O-BR;
- the O-BR gathers information on the current availability of Fog nodes and deployed applications in the Fog infrastructure, by contacting the O-RD, which has been interfacing with the monitoring agent module and the service allocation module of each of the active Fog nodes, gathering information periodically on their status; this allows the O-BR to check whether the requested APP is already implemented on a SaaS Fog node, resulting in two possible scenarios:
  - if the application is provided by at least one SaaS node, the O-BR proceeds by comparing the resource utilization of each of those nodes, and discards those whose resource usage is higher than a predetermined threshold, which can be imposed on CPU, RAM, disk, network utilization, or a combination thereof; if no node remains, the O-BR will try to allocate this APP on a IaaS node; inveserly, if there is at least one available SaaS nodes, the O-BR then picks one on the basis of a heuristic, which in the most simple case gives priority to the least utilized node; finally, it responds to the O-UA with details on the chosen node, so that the user is able to find it; in this case, this concludes the procedure (`OBR_APP_AVLB_S`, 200);
  - in case the application is not yet available on any SaaS node, or if all candidate SaaS nodes are fully utilized already, the O-BR goes through the list of nodes again, looking for IaaS ones; the O-BR compares the resource utilization of each node and picks one if possible, similarly to the previous case; if no IaaS node is able to host this application, the procedure fails, and the O-BR communicates it to the user through the O-UA (`OBR_APP_NAVL`, 503); on the contrary, if a IaaS node has been picked, the O-BR proceeds

in the allocation;

- the O-BR retrieves the list of available FVE images from the O-IM, and checks which of them implements the requested APP; the O-IM has, in turn, retrieved the image list from the repository of the FVE of choice;
- the O-BR triggers the allocation of the requested APP on the chosen IaaS node by means of the selected image, through the O-IM;
- the O-IM forwards the request to the N-IM component of the chosen Fog node; if, for any reason, the Fog node results unreachable, the procedure concludes by a back propagation of an error message, so that the user can retry in a different moment (`OIM_FND_NAVL`, 500);
- if the image is not already present on the IaaS node, the N-IM handles its download (*pull*) from the repo;
- the N-IM launches a container on the IaaS node base on the image, and back propagates the details on the allocated container to the O-BR, which, in turn, will propagate them to the O-UA and finally to the user, successfully concluding the procedure (`OBR_APP_ALLC_I`, 201).

An analogous procedure is devised for the provisioning of SDPs and FVEs.

6.4 Testbed implementation

The structure of the testbed is logically coherent with the architecture of FORCH depicted in Figure 6.1. A VM with 2 cores and 4 GB of RAM hosts all the FORCH software components, which were developed as separate Python3 programs meant to be run independently and communicating with each other via REST APIs. A number of Fog nodes are deployed on different hardware platforms. Two nodes are emulated by two separate VMs, each with 1 core and 2 GB of RAM. Two additional nodes are implemented by an Intel NUC MiniPC equipped with a 4-core 8th-gen Intel i7 processor and 16 GB

```
User -> O-UA
        O-UA -> O-BR
                O-BR -> O-RD (GET /nodes)
                O-BR <- O-RD
                O-BR -> O-RD (GET /apps)
                O-BR <- O-RD
                O-BR --------> O-IM
                O-BR <-------- O-IM
                O-BR --------> O-IM
                               O-IM -> N-IM
                                       N-IM -> repo
                                       N-IM <- repo
                               O-IM <- N-IM
                O-BR <-------- O-IM
        O-UA <- O-BR
User <- O-UA
```

Figure 6.3: Sequence diagram for `POST /app/<app_id>`.

of RAM, and a RaspberryPi Single Board Computer, model 3B+, equipped with a 4-core ARMv7l processor and 1 GB of RAM, respectively.

The role of monitoring system is played by Zabbix [B77], a software suite providing monitoring of resources and functionality of a generic system, and coming in the form of a set of agent modules and a collector module. The collector and the agent modules run as background daemons, the former acting as the RM module of the FORCH, the latter being replicated in each of the Fog nodes. Once the monitoring modules are appropriately configured, they remain available and reachable at need through their REST APIs. Docker [B59] was chosen as FVE of choice—although it is not the only possibility—due to its ease of configuration, compatibility over different systems and large availability of APIs.

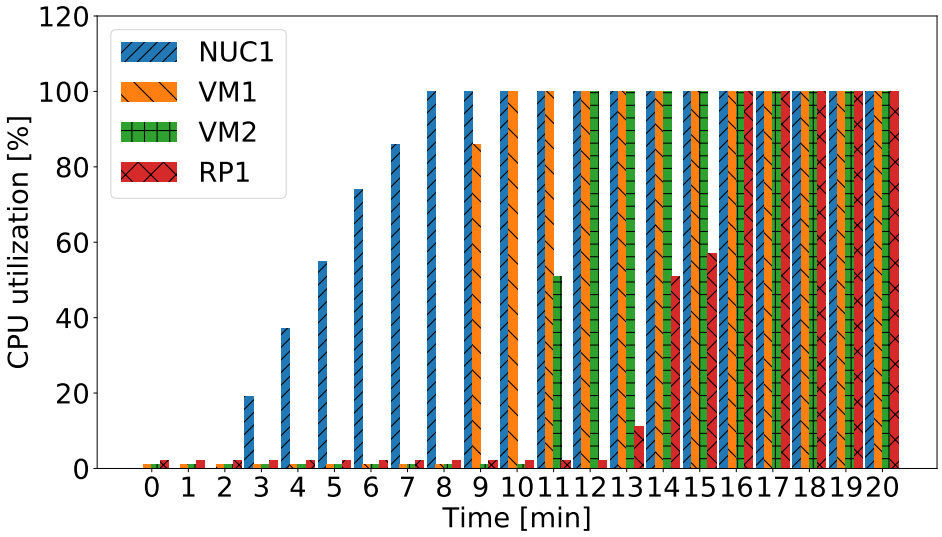


Figure 6.4: CPU utilization of Fog nodes during a sequence of allocation requests. FORCH tries to load all available nodes, always giving precedence to the most suitable available one, and leveraging the flexibility of the XaaS model to deploy services on platforms/engines that support them, this way achieving the highest resource occupation possible.

6.5 Proof-of-concept evaluation

As a PoC evaluation, FORCH is put in a situation where a sequence of requests and allocations saturates all the resources of all the available Fog nodes, forcing the orchestrator to reject subsequent requests.

Figure 6.4 shows the CPU utilization of the four Fog nodes connected to the testbed Fog infrastructure, reporting how the system behaves in the situation where a sequence of homogeneous requests and allocations increasingly saturates the computation resources of the available Fog nodes. For the sake of simplicity, during this experimental validation session the Broker module always selects the Fog nodes in sequence, ordered by their monitoring ID, therefore the resource of each node are expected to saturate before the next node is picked for deploying a service on it. The reported evaluation was conducted by having a "user" (actually a `bash` script) to request an

independent allocation of an application which consumes a tunable amount of resource [B78]. A new request is made every minute. A Fog node is considered fully occupied when its CPU utilization exceeds 90% of its computing power. As shown in the figure, the first node - which happens to be the most powerful one - receives consecutive allocations that make its CPU utilization rise steadily until saturating. At that point, the next node starts receiving allocations until its CPU utilization saturates. Then, a similar process happens for the other two Fog nodes, until all of them have a fully-busy CPU and can no longer accepts new service allocation, at which point the user is denied any further allocation. With specific reference to the figure, instances of the requested service are allocated to the NUC1 node as long as its CPUs are not fully loaded (at $t = 8$ min), then the other nodes are selected for the service requests that follow, until the whole infrastructure is fully utilized (at $t = 17$ min). In case no additional service request is received, the nodes start to free the resources when each service is completed (from $t = 18$ min to $t = 21$ min). This example is intended only as a basic PoC, as FORCH is predisposed to handle heterogeneous service requests, monitoring a variety of usage metrics.

6.6 Remarks

The Fog orchestrator presented in this chapter is still under development, in fact [P13] only sets the foundation for the development of FORCH by introducing the idea and including a preliminary demonstration of its functionalities. More use cases are being investigated and will soon be the object of another publication. This work is expected to yield a modular system that is open to successive integrations and improvements, towards the development of a complete system for service deployment and orchestration in dynamic Fog scenarios.

Chapter 7

Conclusion

This Thesis has presented the outcomes of research efforts that investigate a variety of aspects and features of software-based infrastructure. Some of them consolidate recent novel results, while others propose new solutions and lay the groundwork for valuable advancements in the state of the art. Each chapter has described different solutions, yet remaining consistent with the main topic of the Thesis, and drawn relevant specific conclusions. All four of the main themes delineated in Section 1.3 have been covered, by validating and enhancing existing solutions and proposing new ones, as well as advancing the state of the art with substantial contributions, which have already been mentioned throughout the treatise. The works discussed in the previous chapters have represented the main focus of my research activities, and my contribution to each of them represented the majority of the effort to produce them. Concurrently, I took part in a number of other research efforts, contributing in various capacities. All of these publications are showcased in Figure 1.5, distributed by relevance to each of the main themes of the Thesis. To sum up once more and conclude, the following is a summary of the works I was involved in during the course of my PhD studies, each introduced by a brief comment on its main topic and associated to the resulting publication(s):

– an intent-based NBI for dynamic SFC over SDN and non-SDN domains alike is validated and applied to interesting real-world scenarios in [P1, P6, P7];

- the implementation of a SFC-aware control plane using OpenFlow is presented in [P2] and applied to a composite scenario in [P10];
- a behavior-driven approach to intent-based infrastructure management is presented in [P4];
- an intent-based approach to service specification on an ETSI NFV environment is proposed in [P8];
- the implementation of SFC through Segment Routing supporting 5G network slicing is evaluated in [P9];
- the performances of SDN and SFC integration in the OpenStack platform are evaluated in [P5, P3];
- an innovative protocol for network management using out-of-channel signalling is presented in [P11, P15];
- the design and implementation of a unified and standalone monitoring module for SDN/NFV infrastructures is presented in [P12];
- an architecture for distributed SFC with guarantees is presented in [P14];
- a service orchestrator for Fog scenarios is introduced in [P13].

Acronyms

CLI Command Line Interface.

DC Data Center.

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.

MANO Management and Orchestration.

MMP Music Management Protocol.

NAT Network Address Translation/Translator.

NBI North-Bound Interface.

NFV Network Function Virtualization.

NFVO Network Function Virtualization Orchestrator.

NSH Network Service Header.

QoS Quality of Service.

SDN Software-Defined Network(ing).

SF Service Function.

SFC Service Function Chain(ing).

TC Traffic Controller/Shaper.

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] 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 Computer Communication Review* 42.4 (2012), pp. 13–24.

[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] S. Zhang, X. Xu, Y. Wu, and L. Lu. "5G: Towards energy-efficient, low-latency and high-reliable communications networks". In: *2014 IEEE International Conference on Communication Systems*. Nov. 2014, pp. 197–201.

[B4] I. Parvez, A. Rahmati, I. Guvenc, A. I. Sarwat, and H. Dai. "A Survey on Low Latency Towards 5G: RAN, Core Network and Caching Solutions". In: *IEEE Communications Surveys Tutorials* (2018), pp. 1–1.

[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] *Network Functions Virtualisation (NFV); Management and Orchestration*. The European Telecommunications Standards Institute (ETSI). Dec. 2014. URL: http://www.etsi.org/technologies-clusters/technologies/nfv.

[B7] D. Lenrow. *Intent: Don't tell me what to do! (tell me what you want)*. 2015. URL: https://www.sdxcentral.com/articles/contributed/network-intent-summit-perspective-david-lenrow/2015/02/.

[B8] C. Janz, N. Davis, D. Hood, M. Lemay, D. Lenrow, L. Fengkai, F. Schneider, J. Strassner, and A. Veitch. "Intent NBI – definition and principles". In: *Open Networking Foundation* 2 (2015).

[B9] A. Lerner. *Intent-based networking. Gartner Blog Network*. 2017. URL: https://blogs.gartner.com/andrew-lerner/2017/02/07/intent-based-networking/.

[B10] A. Clemm, L. Ciavaglia, L. Granville, and J. Tantsura. "Intent-Based Networking-Concepts and Overview". In: *Internet Engineering Task Force, Internet-Draft* (2019).

[B11] C. Li, Y. Cheng, J. Strassner, O. Havel, W. Liu, P. Martinez-Julia, J. Nobre, and D. Lopez. "Intent Classification". In: *Internet Engineering Task Force, Internet-Draft* (2019).

[B12] T. Luo, H. P. Tan, and T. Q. S. Quek. "Sensor OpenFlow: Enabling Software-Defined Wireless Sensor Networks". In: *IEEE Communications Letters* 16.11 (2012), pp. 1896–1899.

[B13] D. Zeng, T. Miyazaki, S. Guo, T. Tsukahara, J. Kitamichi, and T. Hayashi. "Evolution of Software-Defined Sensor Networks". In: *Mobile Ad-hoc and Sensor Networks (MSN), 2013 IEEE Ninth International Conference on*. 2013, pp. 410–413.

[B14] L. Galluccio, S. Milardo, G. Morabito, and S. Palazzo. "SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks". In: *2015 IEEE Conference on Computer Communications (INFOCOM)*. 2015, pp. 513–521.

[B15] C. Buratti, A. Stajkic, G. Gardasevic, S. Milardo, M. D. Abrignani, S. Mijovic, G. Morabito, and R. Verdone. "Testing Protocols for the Internet of Things on the EuWIn Platform". In: *IEEE Internet of Things Journal* 3.1 (2016), pp. 124–133.

[B16] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. "B4: Experience with a Globally-deployed Software Defined WAN". In: *SIGCOMM Comput. Commun. Rev.* 43.4 (2013), pp. 3–14.

[B17] R. Jain and R. Khondoker. "Security Analysis of SDN WAN Applications–B4 and IWAN". In: *SDN and NFV Security*. Ed. by R. Khondoker. Vol. 30. Lecture Notes in Networks and Systems. Cham, Switzerland: Springer, 2018, pp. 111–127.

[B18] *What is Software-Defined WAN (or SD-WAN or SDWAN)?* URL: `https://www.sdxcentral.com/sd-wan/definitions/software-defined-sdn-wan/`.

[B19] 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)*. 2016, pp. 7–12.

[B20] 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.

[B21] 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.

[B22] D. Zhou, Z. Yan, Y. Fu, and Z. Yao. "A Survey on Network Data Collection". In: *Journal of Network and Computer Applications* 116 (2018), pp. 9–23.

[B23] P. Tsai, C. Tsai, C. Hsu, and C. Yang. "Network Monitoring in Software-Defined Networking: A Review". In: *IEEE Systems Journal* 12.4 (Dec. 2018), pp. 3958–3969.

[B24] B. Yi, X. Wang, K. Li, S. k. Das, and M. Huang. "A Comprehensive Survey of Network Function Virtualization". In: *Computer Networks* 133 (2018), pp. 212–262.

[B25] M. S. Bonfim, K. L. Dias, and S. F. L. Fernandes. "Integrated NFV/SDN Architectures: A Systematic Literature Review". In: *ACM Comput. Surv.* 51.6 (Feb. 2019), 114:1–114:39.

[B26] D. Clark. "The Design Philosophy of the DARPA Internet Protocols". In: *Proc. ACM SIGCOMM*. Stanford, California, USA, 1988, pp. 106–114.

[B27] K. Fall. "A Delay-tolerant Network Architecture for Challenged Internets". In: *Proc. ACM SIGCOMM*. Karlsruhe, Germany, 2003, pp. 27–34.

[B28] I. Pepelnjak. *Fate Sharing in IP Networks*. URL: `https://blog.ipspace.net/2014/08/fate-sharing-in-ip-networks.html`.

[B29] Y. Zhu, X. Zhou, Z. Zhang, L. Zhou, A. Vahdat, B. Y. Zhao, and H. Zheng. "Cutting the Cord: A Robust Wireless Facilities Network for Data Centers". In: *Proc. ACM MobiCom*. 2014, pp. 581–592.

[B30] M. Hanspach and M. Goetz. "On Covert Acoustical Mesh Networks in Air". In: *Journal of Communications* 8.11 (Nov. 2013), pp. 758–767.

[B31] R. Nandakumar, K. K. Chintalapudi, V. Padmanabhan, and R. Venkatesan. "Dhwani: Secure Peer-to-peer Acoustic NFC". In: *Proc. ACM SIGCOMM*. 2013, pp. 63–74.

[B32] R. Hasan, N. Saxena, T. Haleviz, S. Zawoad, and D. Rinehart. "Sensing-enabled Channels for Hard-to-detect Command and Control of Mobile Devices". In: *Proc. ACM ASIA CCS*. 2013, pp. 469–480.

[B33] A. Madhavapeddy, R. Sharp, D. Scott, and A. Tse. "Audio networking: the forgotten wireless technology". In: *IEEE Pervasive Computing* 4.3 (July 2005), pp. 55–60.

[B34] M. Sharif-Yazd, M. Khosravi, and M. K. Moghimi. "A Survey on Underwater Acoustic Sensor Networks: Perspectives on Protocol Design for Signaling, MAC and Routing". In: *Journal of Computer and Communications* 5.5 (Mar. 2017), pp. 12–23.

[B35] M. Hogan and F. Esposito. "Music-Defined Networking". In: *Proc. ACM HotNets*. 2018, pp. 155–161.

[B36] Y. Jiang, Z. Huang, and D. H. K. Tsang. "Challenges and Solutions in Fog Computing Orchestration". In: 32.3 (May 2018), pp. 122–129.

[B37] Z. Wen, R. Yang, P. Garraghan, T. Lin, J. Xu, and M. Rovatsos. "Fog Orchestration for Internet of Things Services". In: 21.2 (Mar. 2017), pp. 16–24.

[B38] S. Tuli, R. Mahmud, S. Tuli, and R. Buyya. "FogBus: A Blockchain-based Lightweight Framework for Edge and Fog Computing". In: *Journal of Systems and Software* 154 (2019), pp. 22–36.

[B39] R. V. Rosa, M. A. S. Santos, and C. E. Rothenberg. "MD2-NFV: The case for multi-domain distributed network functions virtualization". In: *2015 International Conference and Workshops on Networked Systems (NetSys)*. 2015, pp. 1–5.

[B40] K. Phemius, M. Bouet, and J. Leguay. "DISCO: Distributed multi-domain SDN controllers". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. 2014, pp. 1–4.

[B41] B. Sonkoly, J. Czentye, R. Szabo, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, and F. Risso. "Multi-Domain Service Orchestration Over Networks and Clouds: A Unified Approach". In: *2015 ACM SIGCOMM Conference*. London, United Kingdom, 2015, pp. 377–378.

[B42] *Intent NBI - Definition and Principles*. The Open Networking Foundation (ONF). 2016. URL: `https://www.opennetworking.org/sdn-resources/technical-library`.

[B43] *ONOS: Open Network Operating System*. URL: https://www.opennetworking.org/onos/.

[B44] *OpenStack: Open Source Cloud Computing Software*. URL: https://www.openstack.org/.

[B45] *Mininet: An Instant Virtual Network on your Laptop*. URL: http://mininet.org.

[B46] M. D. Abrignani, C. Buratti, D. Dardari, N. El Rachkidy, A. Guitton, F. Martelli, A. Stajkic, and R. Verdone. "The EuWIn testbed for 802.15. 4/Zigbee networks: From the simulation to the real world". In: *ISWCS 2013; The Tenth International Symposium on Wireless Communication Systems*. VDE. 2013, pp. 1–5.

[B47] F. Callegati, W. Cerroni, C. Contoli, and F. Foresta. "Performance of Intent-based Virtualized Network Infrastructure Management". In: *Proceedings of IEEE ICC 2017, Paris, France*. 2017.

[B48] *Fed4Fire*. URL: https://www.fed4fire.eu/.

[B49] Y. Boucadair et al. *Service Function Chaining Service, Subscriber and Host Identification Use Cases and Metadata*. draft-sarikaya-sfc-hostid-serviceheader-04.txt. IETF Secretariat, 2017.

[B50] M. Boucadair. *Service Function Chaining (SFC) Control Plane Components*. Internet-Draft draft-ietf-sfc-control-plane-08. Work in Progress. Internet Engineering Task Force, 2016. URL: https://datatracker.ietf.org/doc/html/draft-ietf-sfc-control-plane-08.

[B51] J. M. Halpern and C. Pignataro. *Service Function Chaining (SFC) Architecture*. RFC 7665. 2015. URL: https://rfc-editor.org/rfc/rfc7665.txt.

[B52] P. Quinn, U. Elzur, and C. Pignataro. *Network Service Header (NSH)*. RFC 8300. 2018. URL: https://rfc-editor.org/rfc/rfc8300.txt.

[B53] T. Nadeau and P. Quinn. *Problem Statement for Service Function Chaining*. RFC 7498. 2015. URL: https://rfc-editor.org/rfc/rfc7498.txt.

[B54] *Network Service Header Linux kernel module implementation*. URL: https://github.com/upa/nshkmod.

[B55] *Ryu SDN Framework*. 2020. URL: https://ryu-sdn.org/.

[B56] M.-T. Thai, Y.-D. Lin, P.-C. Lin, and Y.-C. Lai. "Towards Load-Balanced Service Chaining by Hash-based Traffic Steering on Softswitches". In: *Journal of Network and Computer Applications* 109 (2018), pp. 1–10. URL: http://www.sciencedirect.com/science/article/pii/S1084804518300699.

[B57] *sFlow*. URL: https://sflow.org/.

[B58] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-defined networking: A comprehensive survey". In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76.

[B59] *Docker*. URL: https://www.docker.com/.

[B60] L. Chen, J. Xia, B. Yi, and K. Chen. "PowerMan: An Out-of-Band Management Network for Datacenters Using Power Line Communication". In: *Proc. USENIX NSDI*. 2018, pp. 561–578.

[B61] T. Hermann, A. Hunt, and J. G. Neuhoff. *The Sonification Handbook*. Logos Publishing House, Berlin, Germany, 2011.

[B62] *Scapy: Packet crafting for Python2 and Python3*. URL: https://scapy.net.

[B63] *Open vSwitch*. URL: https://www.openvswitch.org.

[B64] *Arduino*. URL: https://www.arduino.cc/.

[B65] *Raspberry Pi*. URL: https://www.raspberrypi.org/.

[B66] F. Esposito. "A Policy-based Architecture for Virtual Network Embedding". PhD thesis. Computer Science Department, Boston University, 2013.

[B67] A. Clemm. *Network Management Fundamentals*. Cisco Press, 2006.

[B68] J. Day. *Patterns in Network Architecture: A Return to Fundamentals*. Prentice Hall, 2008.

[B69] D. Levin, Y. Lee, L. Valenta, Z. Li, V. Lai, C. Lumezanu, N. Spring, and B. Bhattacharjee. "Alibi Routing". In: *Proc. ACM SIGCOMM*. 2015, pp. 611–624.

[B70] S. Mirzaei, S. Bahargam, R. Skowyra, A. J. Kfoury, and A. Bestavros. "Using Alloy to Formally Model and Reason About an OpenFlow Network Switch". In: *CoRR* abs/1604.00060 (2016). URL: `http://arxiv.org/abs/1604.00060`.

[B71] S. Ghorbani and B. Godfrey. "Towards Correct Network Virtualization". In: *Proc. ACM HotSDN*. 2014, pp. 109–114.

[B72] Liu X. et al. "MOZART: Temporal Coordination of Measurement". In: *Proc. ACM SOSR*. 2016, 13:1–13:12.

[B73] J. D. Case, M. Fedor, M. L. Schoffstall, and J. R. Davin. *Simple Network Management Protocol (SNMP)*. STD. IETF, May 1990.

[B74] *Linux Network Namespaces*. URL: `http://man7.org/linux/man-pages/man8/ip-netns.8.html`.

[B75] J. L. Gorlewicz, J. L. Tennison, H. P. Palani, and N. A. Giudice. "The Graphical Access Challenge for People with Visual Impairments: Positions and Pathways Forward". In: *Interactive Multimedia [Working Title]* (2019), pp. 1–17.

[B76] *The Quorum Programming Language*. 2017. URL: `https://quorumlanguage.com/` (visited on 06/26/2019).

[B77] *Zabbix - Enterprise-class Open-source Distributed Monitoring Solution*. URL: `https://www.zabbix.com/`.

[B78] *Stress man page*. URL: `https://linux.die.net/man/1/stress`.

[B79] A. Mahmud and R. Rahmani. "Exploitation of OpenFlow in wireless sensor networks". In: *Computer Science and Network Technology (ICCSNT), 2011 International Conference on*. Vol. 1. 2011.

[B80] T. Miyazaki, S. Yamaguchi, K. Kobayashi, J. Kitamichi, S. Guo, T. Tsukahara, and T. Hayashi. "A software defined wireless sensor network". In: *Computing, Networking and Communications (ICNC), 2014 International Conference on*. 2014.

[B81] *Network Functions Virtualisation (NFV); Architectural Framework*. The European Telecommunications Standards Institute (ETSI). 2013. URL: `http://www.etsi.org/technologies-clusters/technologies/nfv`.

[B82] P. Quinn and U. Elzur. *Network Service Header*. Internet-Draft draft-ietf-sfc-nsh-12. Work in Progress. Internet Engineering Task Force, 2017. 37 pp. URL: `https://datatracker.ietf.org/doc/html/draft-ietf-sfc-nsh-12`.

[B83] F. Esposito, J. Wang, C. Contoli, G. Davoli, W. Cerroni, and F. Callegati. "A Behavior-Driven Approach to Intent Specification for Software-Defined Infrastructure Management". In: *Proc. IEEE NFV-SDN*. 2018.

[B84] P. Mell and T. Grance. *The NIST Definition of Cloud Computing*. SP 800-145. The National Institute of Standards and Technology, Sept. 2011. URL: `https://doi.org/10.6028/NIST.SP.800-145`.

[B85] A. Maleki, M. Hossain, J.-P. Georges, E. Rondeau, and T. Divoux. "An SDN Perspective to Mitigate the Energy Consumption of Core Networks–GÉANT2". In: *International SEEDS Conference*. 2017, pp. 233–244.

[B86] O. N. Foundation. "Software-Defined Networking: The New Norm for Networks". In: (Apr. 2012).

Publications

[P1] W. Cerroni, C. Buratti, S. Cerboni, **G. Davoli**, C. Contoli, F. Foresta, F. Callegati, and R. Verdone. "Intent-based management and orchestration of heterogeneous OpenFlow/IoT SDN domains". In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. July 2017.

[P2] **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.

[P3] D. Borsatti, **G. Davoli**, W. Cerroni, C. Contoli, and F. Callegati. "Performance of Service Function Chaining on the OpenStack Cloud Platform". In: *2018 14th International Conference on Network and Service Management (CNSM)*. 2018.

[P4] F. Esposito, J. Wang, C. Contoli, **G. Davoli**, W. Cerroni, and F. Callegati. "A Behavior-Driven Approach to Intent Specification for Software-Defined Infrastructure Management". In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2018.

[P5] F. Foresta, W. Cerroni, L. Foschini, **G. Davoli**, C. Contoli, A. Corradi, and F. Callegati. "Improving OpenStack Networking: Advantages and Performance of Native SDN Integration". In: *2018 IEEE International Conference on Communications (ICC)*. 2018.

[P6] M. Gharbaoui, C. Contoli, **G. Davoli**, G. Cuffaro, B. Martini, F. Paganelli, W. Cerroni, P. Cappanera, and P. Castoldi. "Demonstration of Latency-Aware and Self-Adaptive Service Chaining in 5G/SDN/NFV infrastructures". In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2018.

[P7] M. Gharbaoui, C. Contoli, **G. Davoli**, G. Cuffaro, B. Martini, F. Paganelli, W. Cerroni, P. Cappanera, and P. Castoldi. "Experimenting latency-aware and reliable service chaining in Next Generation Internet testbed facility". In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2018.

[P8] D. Borsatti, W. Cerroni, **G. Davoli**, and F. Callegati. "Intent-based Service Function Chaining on ETSI NFV Platforms". In: *2019 10th International Conference on Networks of the Future (NoF)*. 2019.

[P9] D. Borsatti, **G. Davoli**, W. Cerroni, and F. Callegati. "Service Function Chaining Leveraging Segment Routing for 5G Network Slicing". In: *2019 15th International Conference on Network and Service Management (CNSM)*. 2019.

[P10] **G. Davoli**, W. Cerroni, S. Tomovic, C. Buratti, C. Contoli, and F. Callegati. "Intent-based service management for heterogeneous software-defined infrastructure domains". In: *International Journal of Network Management* 29.1 (2019). e2051 nem.2051.

[P11] **G. Davoli**, F. Esposito, and W. Cerroni. "A Network Management Protocol for Sonification of Software-Defined Infrastructures". In: *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2019.

[P12] P. Borylo, **G. Davoli**, M. Rzepka, A. Lason, and W. Cerroni. "Unified and standalone monitoring module for NFV/SDN infrastructures". In: *Journal of Network and Computer Applications* (2020).

[P13] **G. Davoli**, D. Borsatti, D. Tarchi, and W. Cerroni. "FORCH: An Orchestrator for Fog Computing service deployment". In: *2020 IFIP Networking Conference*. 2020.

[P14] F. Esposito, M. Mushtaq, M. Berno, **G. Davoli**, D. Borsatti, W. Cerroni, and M. Rossi. "Necklace: An Architecture for Distributed and Robust Service Function Chains with Guarantees". In: *IEEE Transactions on Network and Service Management* (2020).

[P15] J. Pasquesi, F. Esposito, **G. Davoli**, and J. Gorlewicz. "Exploring Vibration-Defined Networking". In: *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. 2020.

Acknowledgments

My deepest gratitude goes to Prof. Cerroni, whom I had the privilege to be able to call just Walter since before starting my PhD journey – and what a remarkable journey it has been. Over the course of these past three years I have learnt a lot, I have changed a lot, I have grown up a lot. Walter has been able to provide guidance for all of that. First of all, he has always respected me and never hesitated going far beyond his duties as supervisor to help me sort out a variety of matters, including personal ones. As a supervisor, he has proven to be an excellent mentor – a leader, never a boss – teaching me fundamental technical notions and methods as well as appreciation for the philosophy behind them. Last but equally important, he is the living proof that a righteous, decent, compassionate person can still reach a prestigious position without compromising his values, and by being loved by everyone they have worked with. If everyone had a Walter in their lives, the world would be a better place. I am truly proud of having been his PhD student. I am truly proud of him, and always will be.

I wish to thank my family, who has always been by my side and never failed in making me feel loved. Mamma Terry, babbo Gianni, nonno Marcello, zia Lalla: grazie mille, vi voglio tantissimo bene.

I would like to thank each and every person that has done something, in any capacity, to support me with care, wisdom, respect and trust over these years – but *I'm a lucky man to count on both hands the ones I love*, so I cannot possibly hope to mention them all here. You know who you are. Thank you. I love you all.

Casalecchio di Reno, Bologna - 15 February 2021