
Manuel Peuster

# Enhancing Development and Deployment of Softwarised Network Services

## Dissertation

submitted to the

Faculty of Electrical Engineering,
Computer Science, and Mathematics

in partial fulfillment of the requirements for the degree of

Doctor rerum naturalium (Dr. rer. nat.)

Paderborn, October 2019

**Referees:**
Prof. Dr. Holger Karl, Paderborn University, Germany
Prof. Dr. Antonio Capone, Politecnico di Milano, Italy

**Additional committee members:**
Prof. Dr. Christian Plessl, Paderborn University, Germany
Jun.-Prof. Dr.-Ing. Christoph Sommer, Paderborn University, Germany
Prof. Dr. Heike Wehrheim, Paderborn University, Germany

Submission: October 2019
Examination: 16.01.2020

# Abstract

Future communication networks, like the upcoming 5th generation of mobile networks (5G), require a new level of flexibility, agility, and manageability that cannot be achieved by legacy network designs. One reason for this is that legacy networks rely on hardware-based, manually configured packet processing elements, the so called network functions. A solution for this is transforming those hardware elements into software components, which can be quickly developed and deployed, flexibly scaled on-demand, and automatically managed. This transformation is called network softwarisation and changes classical network planning, design, and operation tasks into software development processes, creating new challenges for network operators.

This thesis explores those challenges and introduces concepts and solutions to simplify the development and deployment of softwarised network functions and services. Following a typical network function development cycle, I first investigate the problem of developing stateful network functions that must maintain and share their state during dynamic network reconfigurations, e.g., scaling. To do so, I introduce a distributed state management framework for softwarised network functions and a seamless handover mechanism for traffic control in dynamic network deployments. In the second part, I answer the question of how to quickly prototype and test single network functions as well as complex network services in multi-site network topologies. I introduce a rapid prototyping framework for softwarised networks which cannot only be used for function and service prototyping but also to test orchestration systems in large-scale scenarios. In the third part, I explore the use of performance benchmarking solutions to collect performance characteristics of virtualised network functions and services to support automated resource dimensioning decisions during deployment. Besides the introduction of a fully-automated, end-to-end benchmarking solution, different approaches to model the performance behaviour of the benchmarked functions and services as well as to optimise the benchmarking process as such are investigated.

# Zusammenfassung

Zukünftige Kommunikationsnetze, wie z.B. die fünfte Generation der Mobilfunknetze (5G), erfordern ein hohes Maß an Flexibilität, Agilität und Verwaltbarkeit, welches mit existierenden Netzwerkkonzepten und Netzwerkimplementierungen nicht erreicht werden kann. Ein Grund hierfür ist, dass existierende Netzwerke auf Netzwerkelementen, den sogenannten Netzwerkfunktionen, basieren, welche durch Hardware implementiert sind und meist manuell konfiguriert werden müssen. Ein Lösungsansatz, um diese Einschränkungen zu beseitigen ist diese Netzwerkfunktionen statt in Hardware in Software zu implementieren. Dies erlaubt es, die Netzwerkfunktionen schnell zu entwickeln und bereitzustellen, mehrere Netzwerkfunktionen einfach zu komplexeren Netzwerkdiensten zu verbinden sowie diese nach Bedarf zu skalieren und voll-automatisiert zu konfigurieren. Diese Transformation wird auch als „Network Softwarisation“ bezeichnet und ersetzt die klassischen Abläufe für die Netzplanung, den Netzentwurf und den Netzbetrieb durch Softwareentwicklungsprozesse. Hierdurch ergeben sich viele neue Herausforderungen für die Netzbetreiber.

Diese Dissertation untersucht diese Herausforderungen und präsentiert sowohl Konzepte, als auch konkrete Lösungen, um die Entwicklung und Bereitstellung von softwarebasierten Netzwerkfunktionen und Netzwerkdiensten zu vereinfachen. Dabei folgt der Aufbau dieser Arbeit dem typischen Entwicklungszyklus einer Netzwerkfunktion bzw. eines Netzwerkdienstes. Zunächst untersuche ich die Probleme die beim Entwickeln von statusbehafteten Netzwerkfunktionen, welche ihren Zustand während und nach dynamischen Konfigurationsänderungen beibehalten und teilen müssen, auftreten. Dazu präsentiere ich zum einen ein verteiltes System zum Exportieren, Übertragen und Importieren von Statusinformationen und zum anderen einen Mechanismus zum Umleiten von Datenströmen in dynamischen Netzumgebungen. Im zweiten Teil der Arbeit beantworte ich die Frage, wie man einzelne Netzwerkfunktionen und komplexe Netzwerkdienste mit wenig Aufwand prototypisch umsetzen und in realistischen Szenarien mit mehreren Standorten testen kann. Hierzu präsentiere ich eine Plattform zur Prototypenentwicklung von Netzwerkfunktionen und -diensten. Diese Plattform kann außerdem zum Testen von Orchestrierung Systemen in großen Netzwerkszenarien eingesetzt werden. Im dritten Teil der Arbeit untersuche ich Benchmarking-Ansätze zum Sammeln von Vergleichsinformationen, um die Leistungsfähigkeit von Netzwerkfunktionen und -diensten zu charakterisieren. Mit Hilfe dieser Informationen kann

dann der Entscheidungsprozess zum Ressourcenbedarf bei der Bereitstellung der Funktion oder des Dienstes vollautomatisiert unterstützt werden. Neben einer Ende-zu-Ende-Lösung zum automatisierten Sammeln der genannten Vergleichsinformationen untersuche ich verschiedene Ansätze zum Modellieren und Repräsentieren der erhobenen Daten sowie zur Optimierung des Benchmarking-Prozesses als solchen.

# Acknowledgements

I would like to thank Prof. Dr. Holger Karl for his advice and support over the course of my research. I learned a lot from our motivating discussions and from your valuable feedback while writing this thesis. I also want to thank Prof. Dr. Antonio Capone for agreeing to act as external reviewer for this thesis. It was always fun to work in the Computer Networks group and I am very thankful for the interesting discussions and close collaborations with my colleagues. In particular, I want to thank Sevil, Hadi, and Stefan for the good times we had during our numerous trips to project meetings and conferences. I am grateful that I had the opportunity to work on various international research and innovation projects. Especially the intense work within the SONATA and 5GTANGO projects was a lot of fun and I am thankful that I had the possibility to meet and work with so many bright colleagues.

All this would not have been possible without the constant support of my family and friends. Thank you! I want to, in particular, thank Tanja for always supporting me with her love, her patience, and her understanding for long workdays and short vacations. I would like to especially thank my parents, Angelika and Karl-Heinz, for always believing in me and backing all decisions I made in my life. Finally, I want to thank my sister, Jana, for always—in the best sense of the word—competing with me and for supporting me with your opinions.

# Contents

# 1. Introduction

Modern communication networks are not only built out of cables, fibres, wireless links, or simple packet-forwarding elements. They also contain a variety of different packet processing elements, so-called network functions or middleboxes, which process, forward, manipulate, or drop packets. Those network functions have been usually realised as dedicated, proprietary hardware boxes that provide a predictable performance and are manually set up, configured, and integrated into our networks. Typical examples are firewalls, intrusion detection systems (IDSs), traffic shapers, or caches that are deployed on the network path between the end users and the accessed end services. Those functions are deployed and managed by network operators and are often combined to more complex network services (NSs). A packet entering such an NS traverses the involved functions one after the other before it leaves the NS on its way to, e.g., the end user. This is called chaining and it requires manual rewiring whenever a new function should be added to the service or the service's configuration should be changed.

A key problem of these hardware-based legacy NS is that they are inflexible and cannot be deployed on-demand [Chi+12]. They are, for example, tied to their physical location and provide a fixed amount of resources making them difficult to use for the upcoming generations of networks, which promise more agility, e.g., reduced time-to-market, more flexibility, e.g., on-demand scaling of NS resources, and better manageability, e.g., fully-automated network management [Chi+12; 5GP18].

Besides those technical challenges, network operators are currently overwhelmed by the rise of more and more over-the-top (OTT) providers, such as Google, Facebook, and Amazon, who are serving and, more importantly, monetising most of today's online content. One of the reasons for the success of these OTTs is the high level of agility with which they operate and adapt to new markets. In addition to this, new services, like WhatsApp, Skype, and FaceTime are cannibalising classical communication services, like telephony or short message service (SMS). As a result, network operators are seeking new business opportunities and revenue streams as well as ways to improve their operational flexibility to not become simple "bit-pipe" providers [Bes10]. Moreover, more and more industries depend on excellent network connectivity and often come with very specific—sometimes contradicting—demands, such as ultra-low latency or ultra-high data rate [IEE17]. Legacy, general-purpose

networks, as they are currently run by the network operators, offer limited support for those "vertical use cases", such as remote surgery, smart manufacturing (industry 4.0), connected vehicles, Internet of Things (IoT), and public protection and disaster relief [IEE17]. Hence, network operators need new solutions to offer vertical-specific networking solutions and services. To address those challenges, the networking community started to introduce a new concept, called "network softwarisation", which turns network elements either into programmable entities or transforms them entirely into software components. Those components can then be executed on commodity hardware using different virtualisation technologies. They can also be moved to cloud infrastructures to make them available on-demand, simplify their management, and improve their scalability.

The two most prominent concepts in the field of network softwarisation are software defined networking (SDN) and network function virtualisation (NFV). The goal of SDN is to make network elements, like switches, programmable to improve the flexibility of networks. The instruction sets and programming models used in SDN are, however, limited and the network elements are still realised as hardware elements with a fixed location in the network. A usual SDN scenario relies on a (logically) centralised control entity that has a global view on all managed network elements and can thus optimise decisions and control tasks. The NFV concept, in contrast, aims to entirely implement the network elements as software components which can then be deployed and executed using virtualisation technologies, like lightweight containers, unikernels, or full-fledged virtual machines (VMs). This decouples the functionalities of the network elements from the underlying hardware and turns them into so-called virtual network functions (VNFs). In such a setup, cloud computing concepts can be applied and the VNFs can be quickly instantiated at different locations in the network and their resources can be scaled on-demand. At the same time, the capital and operational expenses are reduced because VNFs can be executed on of-the-shelf commodity servers.

Chapter 2 describes the technical background of both concepts in more detail and provides an overview of the applied standards, involved protocols, and widely used software projects.

## 1.1. Chances and challenges in network softwarisation

One of the most important benefits of network softwarisation is the possibility to quickly develop and deploy new functionalities. Those can either be new features, added to a single VNF or to a complex NS, but also completely new NSs that are rolled out in an operator's network for the first time. In any case, changes on software components can be applied much quicker than on legacy hardware components and should require much less manual effort.

The remaining management effort can be reduced even further by automating most parts of the deployment and operation tasks of software networks. But the concept of network softwarisation enables even more: It allows us to apply development and operation (DevOps) methodologies [BWZ15], widely used in modern software development processes, to the networking domain. DevOps not only promises to bring the network development and the operation tasks closer together and to further reduce time-to-market, it also establishes a direct feedback loop from operational systems to the developers. This allows them to to quickly detect problems, e.g., performance issues, and react accordingly [Kar+16]. In such scenarios, new code is automatically deployed to production within minutes, which allows to release new features several times per day [BWZ15].

The use of softwarisation and DevOps concepts is, however, new to the telecommunication industry and requires organisational, technical, and operational changes if they should be widely adopted [BWZ15; Kim+15]. Besides several benefits, the use of SDN and NFV together with DevOps also introduces several challenges and requires new processes, paradigms, and tools to support VNF and NS developers as well as operators. Those supporting technologies are the main focus of this thesis. More specifically, I focus on the following three challenges.

First, if network functions are deployed as VNFs on top of virtualised infrastructure, their number might be (automatically) adapted to the traffic demand by adding additional replicas of the VNFs or by removing existing ones—a process called "horizontal scaling". Those scaling operations work seamlessly as long as the involved VNFs do not maintain state, i.e., they are stateless network functions. If this is not the case and stateful VNFs are involved, solutions are needed to distribute the state when new VNFs are added and maintain the state when existing VNFs are removed (terminated). Such solutions concern the development phase, e.g., implementing interfaces to import and export VNF-specific state, as well as the operation phase of a VNF, e.g., traffic control and flow rerouting during scaling operations. To address this, I present solutions to support development and operation of stateful VNFs in Part I of this thesis.

Second, developers as well as operators want to test VNFs and NSs before they are deployed to production. One option is to use lab-scale testbed installations that provide some network function virtualisation infrastructure (NFVI) on which VNFs and NSs can be deployed and tested. However, such testbeds come with several downsides. First, the installation and maintenance of such testbeds requires a lot of effort. Second, it is complicated to reset such testbeds to ensure a clean environment for upcoming tests, especially if the testbed is shared with other developers. Third, most NFV scenarios consider distributed infrastructures in which many spatially distributed and interconnected NFVIs, the so-called points of presence (PoPs), are used. This is hard to replicate with

testbed installations that typically provide only a basic NFVI installation representing a single PoP, or a small number of PoPs connected to a fixed topology. To solve this, I present a rapid prototyping solution for NFV functions and services that allows to emulate arbitrary topologies of multiple NFVIs in a lab-scale testbed or even on a developer's laptop. After that, I present an extension to this prototyping platform that adds support to prototype advanced service function chaining (SFC) scenarios. The presented platform cannot only be used for prototyping VNFs and NSs but also to test NFV orchestration solutions in massively distributed environments with hundreds of NFVI PoPs, which is not possible with legacy testbed setups and development environments. To this end, I utilise a software testing concept called "smoke testing" that aims to quickly check if the basic functionalities of a complex system work correctly, without testing all details of the system. I extend this concept and introduce the concept of "emulation-based smoke testing for NFV orchestrators" together with my lightweight prototyping platform in Part II of this thesis.

The third challenge results from the fact that software-based network functions, running on cloud infrastructure and sharing physical resources with other functions, show a completely different performance than legacy functions deployed as dedicated hardware appliances [Mor17]. Where the vendors of hardware functions provide fixed specifications about the achievable performance, e.g., minimum throughput or maximum latency, the performance of VNFs heavily depends on the execution environment as well as their configuration. This makes it hard for operators to deploy software-based functions without violating any quality of service (QoS) agreements. This becomes even more important if the network operation is fully automated and controlled by orchestration solutions. Those orchestrators need to have some knowledge about the relationship between resource assignments, configurations, and achievable performance of the managed VNFs to optimise deployment decisions. One option is to use monitoring solutions to collect performance metrics at runtime, but operators are also interested in those insights before VNFs and NSs are deployed to production. To this end, I use software benchmarking concepts and apply them to the NFV domain in Part III of this thesis. The resulting solution, called "NFV benchmarking", provides the workflows and tools to automatically learn about the performance of VNFs and NSs and produce so-called NFV performance profiles (NFV-PPs). Such an NFV-PP is an offline model of the performance of a VNF or a complete NS and can be distributed and shared as additional metadata together with the VNFs and NSs it describes. It can finally be used by NFV orchestration systems to optimise the deployment and operation of those VNFs and NS.

## 1.2. Publications

The content presented in this thesis was developed in the time between January 2015 and October 2019. During this period, I authored 4 journal papers and 12 peer-reviewed conference papers. The following chapters are directly based on these papers and contain verbatim parts of them. The copies from my own publications are not explicitly marked as such, to ease the flow of reading, yet all sources are mentioned at the beginning of each chapter. Even though I am the main author of all of these papers, I will use "we" for the reminder of this thesis to indicate that the presented results are based on joint work. The following list gives an overview of the used papers categorised by topics. In addition to these papers, I contributed as co-author to 24 additional journal and conference publications which are not listed here. Paper [Peu+17] was awarded the *2017 IEEE NetSoft Best Demo Award*, paper [PKK18b] the *2018 IEEE NetSoft Best Student Paper Award*, and paper [PSK19b] the *2019 IFIP/IEEE CNSM Best Poster Award*.

#### VNF state management and flow control

[PK16a] Manuel Peuster and Holger Karl. 'E-State: Distributed state management in elastic network function deployments'. In: *2016 2nd IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Seoul, South Korea, June 2016, pp. 6–10. DOI: 10.1109/NETSOFT.2016.7502432

[PKK18b] Manuel Peuster, Hannes Küttner, and Holger Karl. 'Let the state follow its flows: An SDN-based flow handover protocol to support state migration'. In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Montreal, QC, Canada, June 2018, pp. 97–104. DOI: 10.1109/NETSOFT.2018.8460007

[PKK19] Manuel Peuster, Hannes Küttner, and Holger Karl. 'A flow handover protocol to support state migration in softwarized networks'. In: *International Journal of Network Management* 29.4 (Apr. 2019), e2067. DOI: 10.1002/nem.2067

#### Rapid prototyping platforms for NFV

[PKV16] Manuel Peuster, Holger Karl, and Steven Van Rossem. 'MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments'. In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Palo Alto, CA, USA, Nov. 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490

[Peu+17] Manuel Peuster et al. 'A flexible multi-pop infrastructure emulator for carrier-grade MANO systems'. In: *2017 3rd IEEE Conference on Network*

*Softwarization and Workshops (NetSoft)*. IEEE. Bologna, Italy, July 2017, pp. 1–3. DOI: 10.1109/NETSOFT.2017.8004250

[Peu+18a] Manuel Peuster et al. 'A Prototyping Platform to Validate and Verify Network Service Header-based Service Chains'. In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Verona, Italy, Italy, Nov. 2018, pp. 1–5. DOI: 10.1109/NFV-SDN.2018.8725614

[PKK18a] Manuel Peuster, Johannes Kampmeyer, and Holger Karl. 'Containernet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains'. In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Montreal, QC, Canada, June 2018, pp. 335–337. DOI: 10.1109/NETSOFT.2018.8459905

[Peu+19d] Manuel Peuster et al. 'Prototyping and Demonstrating 5G Verticals: The Smart Manufacturing Case'. In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. Paris, France, June 2019, pp. 236–238. DOI: 10.1109/NETSOFT.2019.8806685

#### Emulation-based testing of NFV orchestrators

[Peu+18b] Manuel Peuster et al. 'Emulation-based Smoke Testing of NFV Orchestrators in Large Multi-PoP Environments'. In: *2018 European Conference on Networks and Communications (EuCNC)*. IEEE. Ljubljana, Slovenia, Slovenia, June 2018, pp. 1–9. DOI: 10.1109/EuCNC.2018.8442701

[Peu+19a] Manuel Peuster et al. 'Automated testing of NFV orchestrators against carrier-grade multi-PoP scenarios using emulation-based smoke testing'. In: *EURASIP Journal on Wireless Communications and Networking* 2019.1 (June 2019), p. 172. ISSN: 1687-1499. DOI: 10.1186/s13638-019-1493-2

#### Performance benchmarking and testing of NFV functions and services

[PK16b] Manuel Peuster and Holger Karl. 'Understand Your Chains: Towards Performance Profile-Based Network Service Management'. In: *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*. IEEE. The Hague, Netherlands, Oct. 2016, pp. 7–12. DOI: 10.1109/EWSDN.2016.9

[PK17] Manuel Peuster and Holger Karl. 'Profile your chains, not functions: Automated network service profiling in DevOps environments'. In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Berlin, Germany, Nov. 2017, pp. 1–6. DOI: 10.1109/NFV-SDN.2017.8169826

[PK18] Manuel Peuster and Holger Karl. 'Understand Your Chains and Keep Your Deadlines: Introducing Time-constrained Profiling for NFV'. In: *2018 IEEE/IFIP 14th International Conference on Network and Service Management (CNSM)*. IEEE. Rome, Italy, Nov. 2018, pp. 240–246

[Peu+19c] Manuel Peuster et al. 'Joint testing and profiling of microservice-based network services using TTCN-3'. In: *ICT Express* 5.2 (June 2019), pp. 150–153. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2019.02.001

[Peu+19b] Manuel Peuster et al. 'Introducing Automated Verification and Validation for Virtualized Network Functions and Services'. In: *IEEE Communications Magazine* 57.5 (May 2019), pp. 96–102. ISSN: 0163-6804. DOI: 10.1109/MCOM.2019.1800873

[PSK19b] Manuel Peuster, Stefan Schneider, and Holger Karl. 'The Softwarised Network Data Zoo'. In: *2019 IEEE/IFIP 15th International Conference on Network and Service Management (CNSM)*. IEEE. Halifax, Canada, Oct. 2019

Besides those scientific publications, my research also lead to several open-source projects, some of which are actively used by the networking community. I received the *Outstanding Technical Contributor Award* for the OpenSource MANO (OSM) release THREE cycle from the European Communications Standards Institute (ETSI) for contributing the vim-emu project to OSM. Table 1.1 gives an overview of those projects and pointers to their online resources. Some of them have been developed as part of the SONATA-NFV [SON15b] and 5GTANGO [5GT17a] H2020 5GPPP projects, in which I was heavily involved between 2015 and 2019.

Table 1.1.: Open-source projects that have been created as part of my research activities

| Project | Part of | License | Reference |
|---|---|---|---|
| Containernet | | Mininet License | [Peu16] |
| vim-emu | OSM [ETS16c] | Apache 2.0 | [Peu17] |
| tng-pkg | 5GTANGO SDK | Apache 2.0 | [5GT17a] |
| tng-bench | 5GTANGO SDK | Apache 2.0 | [5GT17a] |
| SNDZoo | | CC-BY-SA 4.0 | [PSK19a] |

Finally, my research directly contributed to a standardisation activity within the Internet Engineering Task Force (IETF)'s benchmarking methodology working group (BMWG) [Ros+18]. In this work, we define and specify models and methodologies to describe and automate VNF benchmarking tasks:

[Ros+18] Raphael Vicente Rosa et al. *Methodology for VNF Benchmarking Automation*. Internet-Draft. IETF, July 2018. URL: https://datatracker.ietf.org/doc/draft-rosa-bmwg-vnfbench/ (visited on 08/19/2019)

## 1.3. Structure of the thesis

The structure of this thesis follows the typical development and deployment cycle of a VNF or NS. More specifically, the content of this thesis is divided into three main parts, after presenting generic technical background in Chapter 2. The following list presents these three parts and the chapters contained in each of them as well as references to publications and supervised student thesis on which those chapters are based. Further details about the used material are given in the introduction of each chapter.

### Part I: Development and operation support

#### Chapter 3: Developing stateful VNFs

One challenge VNF developers face during the development phase of stateful VNFs is how to implement functionalities for the migration of the VNF's internal state, especially during dynamic lifecycle operations, such as scaling, replication, or update. In Chapter 3, I introduce a state management solution for elastic VNF scenarios that helps to transfer state from one VNF to another when it is needed. This chapter is based on [PK16a].

#### Chapter 4: Operation support for stateful VNFs

If a set of stateful VNFs is able to properly migrate and share the internal state, it still needs support from the NFV platform to reroute traffic upon, e.g., scaling operations, which must cooperate with the used state management solution. This chapter introduces a flow handover protocol that allows seamless, order-preserving, and loss-free flow handovers between VNFs. It is based on two papers [PKK18b; PKK19], which both rely on concepts and prototypes developed during an outstanding Bachelor thesis done by Hannes Küttner under my supervision [Küt17]. Mr. Küttner implemented a well-designed prototype of the initial handover concept; the initial concept was defined by me. He was also able to identify and solve several technical issues in the protocol design and performed parts of the experiments used in the evaluation.

### Part II: Rapid Prototyping

#### Chapter 5: Rapid prototyping of NFV functions and services

Once a VNF implementation is ready, it needs to be tested and integrated with other VNF implementations to build larger and more complex NSs. To support developers with this task, I present Containernet and vim-emu. Together they build a rapid prototyping platform for VNFs as well as NSs which is able to emulate user-defined multi-PoP NFV scenarios. This chapter is based on [PKV16].

**Chapter 6: Adding NSH-enabled SFC prototyping capabilities**
A key concept in NFV is SFC, which allows to interconnect arbitrary network functions and reroute traffic through different forwarding paths, e.g., based on the traffic type. Chapter 6 discusses an extension of my prototyping platform with an network service header (NSH)-based SFC approach. This chapter is based on [Peu+18a] and uses a reference implementation and experiment data provided by Frédéric Tobias Christ as part of his Bachelor thesis [Chr18].

**Chapter 7: Supporting the evolution of MANO systems using emulation-based smoke testing**
Besides new VNFs and NSs, new orchestration concepts need to be prototyped and tested as well. To this end, I introduce the concept of "emulation-based smoke testing". The presented solution makes use of the emulation platform introduced in Chapter 5 and extends it to automatically test state-of-the-art management and orchestration (MANO) systems in large-scale multi-PoP scenarios. This section is based on two papers [Peu+18b; Peu+19a]. The chapter also presents a scalability analysis of my emulation platform.

## Part III: Performance Benchmarking

**Chapter 8: Automated benchmarking for NFV**
Initial deployments of new (versions of) NSs are challenging because they require expert knowledge for resource dimensioning decisions. This limits the degree of automation of deployment processes or might lead to inefficient resource usage due to over-dimensioning of assigned resources. To change this, I introduce a benchmarking concept for VNFs and NSs which has initially been presented in [PK16b]. As part of this work, I present an end-to-end automation concept for NFV benchmarking experiments based on [PK17], allowing to automatically collect performance data of VNFs and NSs under different configurations.

**Chapter 9: Benchmarking under time constraints**
One challenge for automated NFV benchmarking comes from the fact that the potential configuration space of an NS, which needs to be explored during benchmarking, becomes very large. To this end, I present solutions to only benchmark a subset of configurations and predict the performance values for the missing configurations, as presented in [PK18].

**Chapter 10: Collecting, analysing, and publishing benchmarking data sets**
Using the concepts and tools presented in the previous chapters, I collect a series of data sets from real-world NFV scenarios and make them available for

use by other researchers. To do so, I define the methodology and workflows to collect the data sets and introduce the softwarised network data zoo (SNDZoo) project, which is a large collection of open NFV data sets, as initially presented in [PSK19b]. This chapter also presents an approach to turn the raw data into NFV-PPs that are compatible to commonly used scaling and placement optimisation approaches, based on integer linear programs (ILPs).

**Chapter 11: Final thoughts**
I summarise and draw conclusions on the results presented in this thesis in Chapter 11. Finally, I give an outlook on further research directions and future work in the field of NFV development, deployment, and operation support.

# 2. Background

This chapter introduces the basic concepts and technologies on which the remainder of this thesis is based. In particular, I introduce the two main enablers of network softwarisation: SDN and NFV.

## 2.1. Software defined networking

The general idea behind SDN is to decouple network control tasks from packet-forwarding tasks. This allows to centralise the control functionality of a network into a so-called "SDN controller" while leaving the actual packet-forwarding functionality at the network elements. The SDN controller can then utilise its global view of large parts of the network to optimise forwarding decisions executed by the network elements. Implementation-wise, this is typically realised by replacing legacy switches and routers with so-called "SDN switches" offering programmable data planes. Those switches are then connected to the (logically) centralised SDN controller executed on commercial off-the-shelf (COTS) hardware or in virtualised environments, e.g., on cloud infrastructure. The decoupled design abstracts the underlying network infrastructure and turns it into a single virtualised entity that can be used and shared by different NSs or business applications as shown in Figure 2.1 [Ope12].

Enabling network programmability by offering a single abstract view to the complete network, instead of single network devices, allows applications to optimise the underlying network to their needs. It allows to dynamically steer and route flows through the network when needed, e.g., sending traffic to intermediate network functions, such as an IDS. Thus, new services can be integrated into existing networks on-demand without requiring manual reconfigurations. This is an important feature for the concept of NFV as described in Section 2.2. In this thesis, SDN is used as part of the presented NFV scenarios and solutions. It is in particular used in Chapter 4 in which an SDN-based handover protocol for flow migration in elastic NFV deployments is presented.

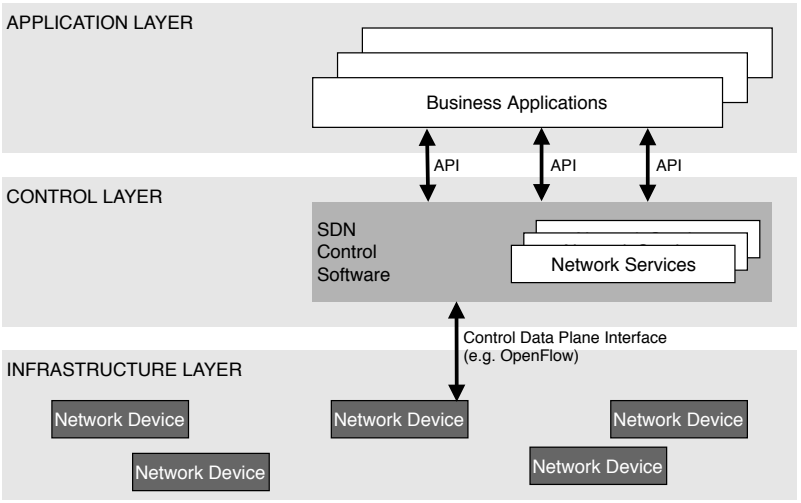



Figure 2.1.: Software defined network architecture as described by the ONF (taken from [Ope12])

### 2.1.1. Interfaces

One of the most important parts of an SDN deployment is the *control data plane interface* between the control layer and the infrastructure layer with its networking devices, such as switches. This unified interface, shown in Figure 2.1, aims to simplify the way how network elements are configured. Instead of heaving different, vendor-specific, and manually applied configuration schemes, a single interface to all devices can be used. The networking community has, however, not agreed to a single protocol that is used on this interface and a variety of options are available. The most prominent protocol is the open-source protocol OpenFlow [McK+08], which promises unified control of heterogenous networks composed of network devices from different vendors. Another open protocol is NetConf [Enn06], which is a more generic management and configuration protocol for a wide range of network devices. But there is also a variety of vendor-specific, proprietary solutions such as Juniper Contrail [Jun12] or Cisco Application Centric Infrastructure (ACI) [Cis19].

### 2.1.2. Open-source SDN controllers

When it comes to SDN controllers, even more options are available, many of which are implemented as open-source projects. Without going into too much detail, I provide a short overview of the most prominent SDN controllers in Table 2.1. All of them offer a different set of features and target different deployment scenarios. For example, OpenDaylight [The13] and ONOS [Lin14b] are built to be used in large production environments such as OpenStack [Ope10b] cloud deployments. POX [The15b] and Ryu [Ryu17], in contrast, focus more on

easy programmability and are better suited to prototype novel control concepts and new SDN applications. In this thesis, most of the presented prototypes rely on the Ryu controller, e.g., the flow handover protocol presented in Chapter 4 and the NFV prototyping platform presented in Chapter 5.

Table 2.1.: List of open-source SDN controllers (non exhaustive)

| Project | Language | License | Reference |
|---|---|---|---|
| NOX | C++ | GPL | [The15b] |
| POX | Python | Apache 2.0 | [The15b] |
| Beacon | Java | GPL | [Eri13] |
| Floodlight | Java | Apache 2.0 | [Pro11] |
| Ryu Controller | Python | Apache 2.0 | [Ryu17] |
| OpenDaylight | Java | EPL 1.0 | [The13] |
| ONOS | Java | Apache 2.0 | [Lin14b] |

## 2.2. Network function virtualisation

The term of NFV was introduced in 2012 in a white paper authored by a series of network operators and officially published by ETSI [Chi+12]. The white paper highlights the increasing problems that operators face within their networks, which are built of a high number of hardware-based appliances composing different services. Those special-purpose appliances require increasing capital investments, are often inflexible when it comes to reconfigurations, and quickly reach their end of life. Finally, the management of networks consisting of a wide variety of different hardware appliances becomes increasingly complex and barely feasible [Chi+12]. This motivates the main idea behind the NFV concept which aims to transform network functions, which have previously been implemented and deployed as hardware appliances, into software components. These components can then be executed on top of general-purpose information technology (IT) servers and can be isolated and managed by virtualisation technologies.

The NFV concept plays well with the SDN ideas presented in the last section. Both technologies can be considered to be highly complementary and are usually used in combination. SDN is mostly used to control the data plane, e.g., make packet-forwarding decisions. NFV, in contrast, is typically used to implement more complex and compute-intensive network functionalities, e.g., deep packet inspection (DPI) functionality. An example for the combined deployment is a complex NS consisting of multiple VNFs, which are deployed using NFV technologies. In this scenario, SDN can be used to steer the traffic through the involved VNFs, a concept called SFC.

NFV potentially offers a couple of benefits compared to legacy networks, as described in [Chi+12]. First, NFV can reduce both capital expenses (CAPEX) and operational expenses (OPEX). CAPEX can be reduced because network functions can run on of-the-shelf IT servers, which are much cheaper than specialised hardware. Further, development costs for software components are often lower, especially when it comes to updating or replacing old generations of network functions. OPEX can be reduced because of simplified maintenance tasks, e.g., bug fixes. In addition, VNFs can be scaled and provisioned on-demand reducing the used resources and thus costs.

Second, NFV encourages innovation of new network functionalities and services because building software-based network functions can be done faster than building their hardware-based counterparts. This reduces the time-to-market by allowing to apply modern software development methods, such as automated testing, continuous integration (CI), and continuous delivery (CD). Also lab-scale testing becomes much easier because new ideas can be easily validated on existing testbeds, without the need to purchase specific hardware.

Third, NFV enables a new level of deployment flexibility, opening the door for automatic, on-demand network optimisation. An example for this is placement and scaling of VNFs and NSs. VNFs can, in contrast to hardware boxes, be deployed, moved, or replicated to any compute infrastructure in an operator's network. As a result, VNFs can be dynamically moved to optimise, e.g., the distance to the end user (latency) or the carbon dioxide footprint (by moving to an energy-efficient data centre). Those optimisations can be automatically controlled by software, a so-called orchestrator, and be performed on-demand, e.g., based on monitored or predicted user demands.

Fourth, NFV has the potential to establish open markets in the information and communication technology (ICT) industry by allowing functions and services of different vendors to run on the same unified infrastructure. This also reduces the operator's risk of vendor lock-ins and might allow them to compose NSs using VNFs from different vendors. It might also strengthen the role of open source and open standards in the ICT domain [Nau+16]. This fourth point, however, heavily depends on the mindsets and preferences of the operators. Technologies like NFV can only act as enablers.

In this thesis, I mostly focus on benefits two and three and introduce concepts and solutions to simplify the development and deployment of NFV functions and services. In the remainder of this section, I introduce more details and concepts about NFV, starting with a description of a typical scenario before focusing on the architecture, function and service description, as well as composition approaches, orchestration solutions, and finally on the underlying infrastructure.

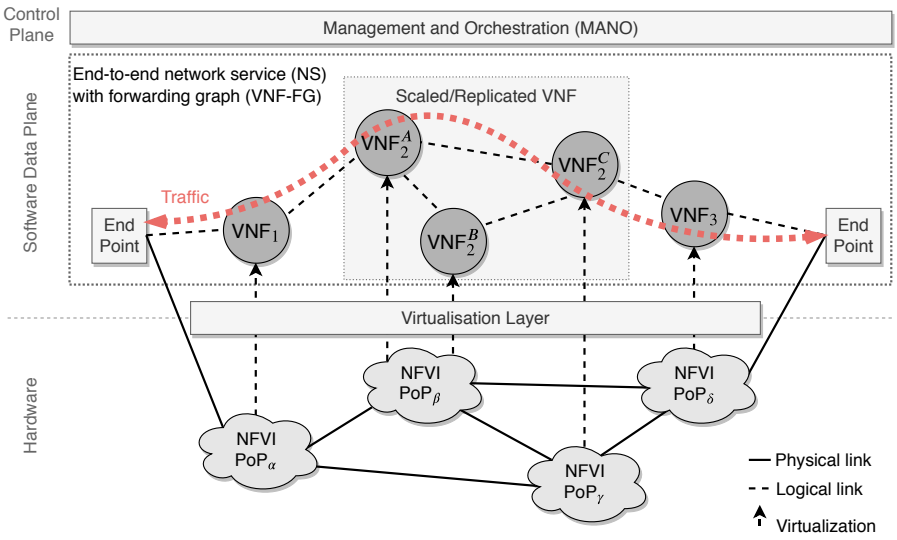



Figure 2.2.: A typical NFV scenario with multiple physical PoPs and a complex NS composed of multiple VNFs managed and controlled by a MANO system (based on [ETS14b]).

### 2.2.1. An NFV scenario

This section briefly describes a typical NFV scenario as I consider it in this thesis, if not stated otherwise. The presented scenario is aligned with the scenarios considered by ETSI [ETS17a], 3rd Generation Partnership Project (3GPP) [3GP15], and by IETF [QN15; HP15]. Aligning this thesis with the relevant real-world NFV scenarios supports the applicability of the developed concepts and solutions.

Figure 2.2 shows an end-to-end NS, which is composed of multiple VNFs. It is placed between the end users and the end services and the traffic traverses the involved VNFs. The VNFs are deployed on top of multiple physical infrastructures. Some VNFs are scaled and/or replicated by running multiple instances of them. The physical infrastructure locations can be larger data centres, small infrastructure in a central office, infrastructure co-located with a mobile base station, or even edge devices offering virtualised resources for the execution of VNFs. Each of them is a so-called NFVI as further detailed in Section 2.2.5. Since all these infrastructure installations are spatially distributed, they are also called PoP or NFVI-PoP. All these NFVIs are under the control of a (logically) centralised orchestration system, the so-called MANO system, which manages the deployment of NSs and is further described in Section 2.2.4.

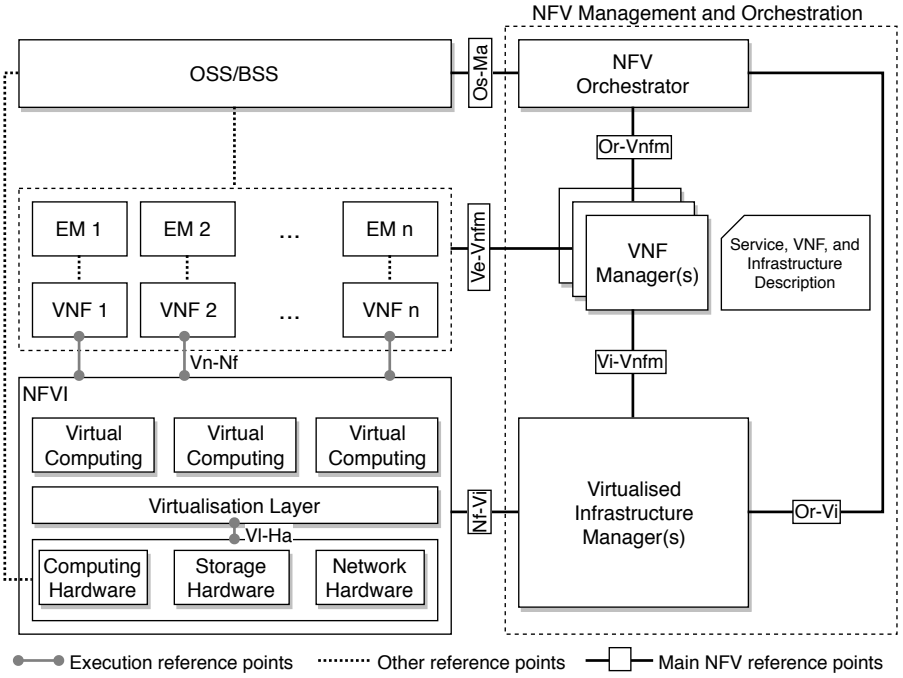



Figure 2.3.: ETSI's NFV reference architectural framework as it is presented in [ETS14b]

### 2.2.2. The NFV reference architecture

ETSI's NFV industry specification group (ISG) leads the standardisation activities in the NFV domain and released the specification document "Network Functions Virtualisation (NFV); Architectural Framework" [ETS14b] in 2014. In this document, a reference architecture for NFV is presented, which is widely agreed and adopted in the ICT industry and other standards developing organisations (SDOs). This architecture defines and unifies the building blocks used in NFV scenarios as well as the reference points between them, without fixing implementation details of the building blocks or interfaces. Figure 2.3 shows this reference architecture and its building blocks which I describe in the following.

1. **Virtual network function (VNF):** VNFs are software implementations of a network function which are packaged and executed inside a virtualisation container, like a VM or Docker container. Multiple VNFs can be interconnected to form complex NSs. A single VNF can also be decomposed into smaller execution entities, e.g., multiple VMs, which are then called virtual deployment units (VDUs) and are out of scope of the reference architecture in Figure 2.3. VNFs are executed on top of the NFVI.
2. **Element manager (EM):** The EM is responsible for the functional man-

agement of one or multiple VNFs. The main purpose of having EMs is to have a translation components which can translate management requests from the NFV MANO components to (proprietary) management interfaces of specific VNF implementations. Depending on the used MANO solution, different realisations of the EM concept can be found, as described in Section 2.2.4.

3. **NFV infrastructure (NFVI):** NFVI is the environment on which VNFs are executed and builds the abstraction layer between hardware resources and virtual resources. An NFVI does not only offer virtual compute resources but also virtual storage and networking resources, e.g., virtual subnets. It provides all means to run complex VNFs and NSs on it. ETSI published a series of specifications that describe the NFVI and its performance requirements [ETS15; ETS18j]. More details about this important building block are given in Section 2.2.5.
4. **NFV orchestrator (NFVO):** The NFVO is one of the three building blocks of the NFV management and orchestration (MANO) part shown in the reference architecture. The NFVO is responsible for the end-to-end management and orchestration of NSs composed of multiple VNFs. This includes automated management tasks, like scaling, placing, and healing of VNFs. ETSI further specifies the MANO components in [ETS16a]. The NFVO is described in more detail in Section 2.2.4.
5. **VNF manager (VNFM):** In contrast to the NFVO, the VNFM component focuses on VNF lifecycle management, e.g., instantiation, configuration, updating, scaling, and termination. ETSI foresees the possibility that multiple VNFMs are deployed, each managing only a subset of the deployed VNFs. This improves scalability and allows to build VNF-specific orchestration solutions. The VNFMs closely collaborate with the EMs.
6. **Virtual infrastructure manager (VIM):** VIMs are the third main building block of the MANO part. They are responsible to manage the virtualised resources provided by the NFVI as further described in Section 2.2.5. Typically, NFVOs and VNFMs are able to connect to multiple VIMs from different vendors using internal abstraction models, often called VIM drivers.
7. **Service, VNF, and infrastructure description:** To be able to deploy an NS or VNF, the MANO systems needs to have a description of those artefacts. There are multiple standardised description approaches for this, as further described in Section 2.2.3.
8. **Operation/business support system (OSS/BSS):** Finally, an NFV environment needs to integrate with existing OSSs and BSSs of an operator.

Besides the building blocks, Figure 2.3 also shows three types of reference points. First, the execution reference points which indicate that one building block is executed by another one, e.g., VNF is executed by NFVI. Second,

the main NFV reference points are shown which describe interfaces between the involved building blocks. ETSI further specifies those reference points and outlines their data models in a series of interfaces and architecture (IFA) documents, such as [ETS18a; ETS18b]. However, those documents are still high-level and do not provide concrete application programming interface (API) specifications.

In all architecture discussions in this thesis, I build upon this reference architecture and map the developed artefacts and concepts against it, if not stated otherwise. The goal is to align my work with the current developments in industry and standardisation, easing adoption.

### 2.2.3. Service description and packaging approaches

The programming model mainly used in the NFV domain is based on so-called descriptors which allow developers to specify how VNFs should be deployed and how NSs are composed [Gar+16]. Those descriptors explicitly do not contain the actual software implementation of the VNFs as such but specify how existing VNF components, e.g., given as VM or container images, have to be deployed, configured, and interconnected. As a result, the standardised and practically used descriptor solutions are based on markup languages, such as XML, JSON, or YAML.

Independently of the concrete description model, there is a common agreement to have different kinds of descriptors to define different artefacts of an NFV deployment. First, there is the so-called virtual network function descriptor (VNFD) which defines a single VNF which may or may not consist of multiple VDUs. The main task of this descriptor is to define which virtual resource to assign to the corresponding VM or container, how to configure the VNF software, which connection the VNF offers, and how to mange the VNF's lifecycle. Depending on the actual standard or implementation, there might be slight variations of this descriptor, e.g., to define cloud-native (container-based) VNFs, also called cloud-native network functionss (CNFs) [5GP18]. Nevertheless, conceptually they all follow the same goals and principles.

Second, there is the so-called network service descriptor (NSD) that specifies how a set of given VNFs is bundled and interconnected (or chained) to compose a large, more complex NS. An NSD typically contains a list of references to the VNFDs of the VNFs to be used as well as additional connection information, like entry points. In addition, most NSDs contain information about the so-called VNF forwarding graphs (VNF-FGs) and/or VNF forwarding paths (VNF-FPs) that describe how packets should traverse the different VNFs when they are sent through the NS. These features are typically used to implement SFC.

Third, some standards and implementations offer further descriptors, e.g., to specify the deployment of so-called network slices [Par+18]. However, these additional descriptors are not relevant for this thesis.

Multiple SDOs work on and publish NFV descriptor specifications. The most relevant ones are the "TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0" [Oas16] developed as open standard based on topology and orchestration specification for cloud applications (TOSCA). As well as the descriptor and data models developed by ETSI. The TOSCA description models are based on specifications initially developed for the cloud-computing domain and the ETSI description models are developed from scratch and are based on ETSI's IFA specifications [ETS18c; ETS18d; ETS18e]. However, recently both description approaches started to converge and ETSI published first specifications that describe how to express the ETSI data models using TOSCA [ETS18f].

Implementation-wise, many NFV-related projects, especially the ones focusing on orchestration, come with their own description formats. Even though many of them claim to be either aligned or compliant to the existing TOSCA and ETSI standards, they usually differ in many details, like the exact naming of fields. Orchestration solutions like 5GTANGO [5GT17a] or OSM [ETS16c] base their descriptor models on the ETSI specifications whereas the open network automation platform (ONAP) [Lin18a] relies on TOSCA. A more detailed overview about these relationships is given in Table 2.2 in Section 2.2.4.

To exchange descriptors and to bundle them with additional artefacts, like VM images, the concept of VNF and NS packages is used. Those packages can be exchanged and shared between different roles, components, platforms, and environments. Standardisation-wise, TOSCA specifies the cloud service archive (CSAR) format [Oas13] which is a generic format to package all kinds of cloud applications, including VNFs and NSs. However, the CSAR format is too unspecific for many NFV use cases which is the reason that ETSI specifies an extended package format for VNFs [ETS18h]. A package format for NSs is currently under development. On top of that, the 5GTANGO project specifies an extended version of ETSI's package format to not only include VNFs and NSs in a single package, but to also be able to bundle them with a set of test scenarios that can be used by their automated verification and validation (V&V) platform [5GT17b; 5GT18a]. The 5GTANGO package specification was mainly developed by me during my work in the 5GTANGO project.

In this thesis, I either use the description format defined by 5GTANGO [5GT18b] or the one defined by OSM [ETS18l], which are both aligned to ETSI's descriptor models. Most of the solutions and tools, developed in this thesis, can be applied and used with both description formats, if not stated otherwise.

### 2.2.4. Management and orchestration

A MANO solution can be considered as the "brain" of every NFV deployment. It is responsible to control, manage, and monitor the entire lifecycle of NSs and VNFs, a process called lifecycle management (LCM). The following sections present the main workflows, typical architectures, and a selection of open-source MANO solutions relevant for this thesis.

#### 2.2.4.1. Workflows

Once an NS and its VNFs are defined and packaged, using the description approaches presented in Section 2.2.3, they are, together with other artefacts like VM images, uploaded to a MANO system to be deployed. This process of uploading NS- and VNF-related artefacts is also called "on-boarding" and usually considered the first step in the LCM of an NS or VNF under control of a MANO system. After on-boarding, the NS can be instantiated on top of one or multiple NFVIs connected to the MANO system and managed by one or multiple VIMs, as shown in Figure 2.3. The decision on which of the connected NFVIs the NS and thus its VNFs should be deployed can either be made manually as part of the instantiation request sent to the MANO system or be fully automated and based on optimisation algorithms [MKK14]. To perform the instantiation, the MANO system uploads the used artefacts to the destination NFVIs and triggers their instantiation, e.g., start of the VMs or containers. During this process, initial configurations, also called "day-0 configurations", like the injection of secure shell (SSH) keys, is performed. Finally, the network is configured to setup the SFC between the involved VNFs, including the creation of one or multiple VNF-FGs and VNF-FPs. After that, the instantiation is done and the NS if fully operational.

The operational NS is continuously monitored and can be reconfigured at any point in time. These reconfiguration actions, also called "day-1 and day-2 configurations", can either be triggered manually or be triggered automatically, e.g., updating a VNF using an automated CI/CD pipeline. To perform these actions, the MANO system relies on its VNFM and EMs, which offer standardised configuration interfaces towards the, potentially proprietary, VNFs. In real-world implementations, these configuration mechanisms are usually realised by programmable plugins that are served together with the NSs and VNFs and are executed within the MANO system. Examples are the function-specific managers (FSMs) and service-specific managers (SSMs) used by the SONATA service platform (SP) [Drä+17] or the Juju Charm-based approach [Can12] used by OSM [ETS16c].

A special case of these reconfiguration tasks is manual or automated scaling of single VNFs or entire NSs. Scaling can either be done by adding or removing

resources to/from VMs or containers, called "vertical scaling", or it can be done by adding or removing complete instances of a VM or container, called "horizontal scaling". Both types of scaling are considered to be key functionalities of NFV solutions because they allow the on-demand adaption of an NS's resources to its load. A special challenge in this process is to automatically pick the right amount of resource to be assigned to NSs and VNFs, especially if not much historical monitoring data is available. This is called "resource dimensioning" and requires insights into the behaviour of VNFs and NSs, which can, for example, be obtained using benchmarking approaches. Such benchmarking approaches are further investigated in this thesis (Part III). Finally, a MANO system is also responsible to terminate NSs and VNFs once they have reached the end of their lifetime and are not needed anymore. During NS and VNF termination, all previously allocated resources are freed to be available for new deployments.

#### 2.2.4.2. Architectures

ETSI's reference architecture, as shown in Figure 2.3, already provides a high-level architecture for MANO systems, splitting them in two main components, the NFVO and VNFM. An additional ETSI specification [ETS16a] refines this and adds additional components, like catalogues and repositories to the MANO system. The former are used to store static information, like descriptors and artefacts that are on-boarded to the MANO system, and the latter store dynamic information, like runtime records, e.g., internet protocol (IP) addresses of instantiated NSs and VNFs. Even though those high-level components are more or less visible in the architectures of all real-world MANO systems, most of them split those components into smaller functional blocks, often following a micro service-based design pattern. As an example, OSM rel. FIVE splits the VNFM into a resource orchestrator (RO) and VNF configuration and abstraction (VCA) component and the NFVO is represented by a module called LCM. Monitoring functionality is implemented in a standalone component, called MON, and is able to talk to all other components of the system. On top of all this, OSM offers a unified northbound interface implemented in a component called NBI. All those small components are interconnected through a message broker offering publish/subscribe communication semantics, an approach initially used by the SONATA SP [SON15a].

More important than the alignment with the architecture specifications published by ETSI is the alignment to interface standards to enable interoperability between MANO components and to allow reuse of artefacts, e.g., NS packages. ETSI specifies those on two levels. First, a high-level specification is done in the so-called IFA documents, e.g., [ETS18a; ETS18b], based on which the ETSI solution (SOL) specifications are created. The latter provide concrete implementation details for APIs and data models. The most prominent ones

are SOL002 (RESTful VNF configuration), SOL004 (VNF package format), and SOL005 (RESTful MANO northbound) [ETS18g; ETS18h; ETS18i], but more specifications are under development and can be found on the ETSI website [ETS19].

#### 2.2.4.3. Open-source MANO solutions

Table 2.2 gives a brief overview over available open-source MANO solutions. Besides orchestration solutions that have been developed within different research projects, like T-NOVA TeNOR [Rie+16], UNIFY ESCAPE [Son+15], or SONATA SP [Drä+17]; some smaller projects, like OpenBaton [Fra15], also exist. Further, there are a couple of projects developed by companies that offer open-source versions of their products with reduced functionality, e.g., Cloudify [Clo18]. Finally, there are two big players that are supported by many operators, SDOs, and vendors: The first is OSM [ETS16c] and the second is ONAP [Lin18a]. Both are backed by their own, growing open-source communities.

Table 2.2.: List of open-source MANO solutions (non exhaustive)

| MANO | Project | License | Model | Reference |
|---|---|---|---|---|
| TeNOR | T-NOVA | Apache 2.0 | ETSI | [Rie+16] |
| Unify | UNIFY | Apache 2.0 | custom | [Son+15] |
| SONATA SP | SONATA-NFV | Apache 2.0 | ETSI | [Drä+17] |
| OpenBaton | Frauenhofer FOKUS | Apache 2.0 | ETSI | [Fra15] |
| Cloudify | Cloudify Inc. | Apache 2.0 | TOSCA | [Clo18] |
| RIFT.ware | RIFT.io | multiple | ETSI | [RIF16] |
| OSM | ETSI OSM | Apache 2.0 | ETSI | [ETS16c] |
| ONAP | Linux Foundation | multiple | TOSCA | [Lin18a] |

In this thesis I use OSM as MANO solution for the presented prototype implementations. OSM which was chosen because it is, at the time of writing, the NFV MANO platform offering the most features and highest degree of production readiness. Parts of my research presented in Chapter 5 even got adopted and are now officially part of the OSM project [ETS16c]. However, concepts presented in this thesis can be equally implemented using other platforms, such as SONATA-NFV or ONAP, by investing sufficient engineering effort.

### 2.2.5. NFV infrastructure

The infrastructure used to deploy and execute VNFs and NSs is, besides the high-level management parts, the most important layer of the NFV stack. At the end, the achieved performance of all VNFs and NSs directly depends on the infrastructure on which they are executed. Further, the used infrastructure does directly influence the OPEX and needs to operate as efficiently as possible. In this section, I give a brief overview of state-of-the-art NFVI concepts and solutions, not only considering the actual infrastructure (NFVI) but also the often tightly coupled VIMs.

Today, the de-facto infrastructure standard in production NFV environments is based on infrastructure as a service (IaaS) cloud concepts and reuses management systems and infrastructure solutions from the cloud community, such as OpenStack [Ope10b]. Based on this, integration projects, like OPNFV [Lin16], have evolved, which create bundles of infrastructure solutions that are specifically tailored for NFV scenarios, e.g., by combining OpenStack with SDN controllers like OpenDaylight [The13] or ONOS [Lin14b]. But also commercial providers and vendors, such as Amazon EC2 [Ama15] and VMware vSphere [VMw13], offer different IaaS solutions specifically tailored for NFV scenarios.

IaaS solutions from the cloud computing community are, however, usually not optimised for fast packet processing, which results in bad performance of VNFs executed on them. One reason for this are the high number of context switches needed when network packets are transferred between the physical interface of a cloud node and the virtual interfaces of the VNFs. To solve this, IaaS platforms are usually extended with additional features and acceleration technologies for fast packet processing. Examples are Intel's data plane development kit (DPDK) [Lin17] and single-root input/output virtualisation (SR-IOV).

#### 2.2.5.1. Container-based NFV

Besides full-fledged VMs, container-based solutions have gained a lot of attention in recent years. Container solutions, such as Docker [Doc13], are by design much more lightweight than VMs and allow much denser deployments on the same physical resources, due to less overheads introduced by, e.g., guest kernels. It also leads to faster startup and configuration times, facilitating use cases like on-demand scaling. However, the isolation offered by most container solutions cannot keep up with normal VMs. Still, the NFV community predicts containers to be the future of NFV, especially if scenarios with compute resources at the edge of the network are considered, e.g., multi-access edge computing (MEC) [ETS14a].

Container-based VNFs, so-called CNFs, can be either deployed on bare-metal container platforms, e.g., a bare-metal Kubernetes [Lin14a] installation, or within existing cloud infrastructure using solutions like the OpenStack Magnum project [Ope17]. Magnum allows to easily deploy and mange Kubernetes [Lin14a] clusters on top of OpenStack. Kubernetes itself is an orchestrator to run and manage huge amounts of containers in cluster setups and it can be used as VIM to manage CNFs. MANO solutions like SONATA SP, OSM, and ONAP are working towards the support of Kubernetes as VIM and its seamless integration with other VIM and NFVI solutions. I present a container-based NFV platform for experimentation and rapid prototyping in Chapter 5 of this thesis.

# Part I.

# Development and operation support

# 3. Developing stateful VNFs

One of the key benefits of NFV is the possibility to automatically scale VNFs and NSs on-demand. As discussed above, one option for this is adding VNF instances to the NS or removing them when needed—horizontal scaling. Horizontal scaling, however, becomes challenging when the scaled VNFs are stateful, since their state must be migrated between instances when new instances are added to or old instances are removed from the deployment. In this chapter, I present a state management framework that supports VNF developers to implement stateful VNFs that can automatically share their internal state in such elastic, on-demand environments. This chapter is based on my paper [PK16a] and contains figures and verbatim copies of the paper's text. After reviewing existing work about VNF state management in Section 3.2, the chapter presents the concepts and system design of "E-State", a state management solution that does not require a central controller to exchange VNF state in Section 3.3. After that, a prototype implementation is presented and compared to three other approaches in Section 3.4 and Section 3.5. Section 3.6 concludes this chapter.

## 3.1. Introduction

One of the main problems in creating elastic VNF deployments, in which VNF instances can be added and removed on-demand, is the fact that many VNFs are stateful. Typical examples for such stateful VNFs are network address translation (NAT) boxes that store mappings between ports and hosts or IDSs that keep track of pattern matchings to detect attacks. Typical VNF application state can be divided into two classes [Raj+13]. The first class contains *global state* accessed independently of the processed traffic. The second class contains *partitioned state* that consists of chunks of state directly related to one or multiple network flows or sessions processed by the VNF. These flow- and session-specific state chunks can be identified by the same information used to identify single flows or sessions [Raj+13]. For IP-based traffic, this is usually done by 5-tuples consisting of source IP, target IP, source port, target port, and transport protocol. In typical VNFs, most parts of the application state are represented by the second class that can easily be distributed across multiple VNF instances [Raj+13; Gem+12].

A problem appears when an elastic VNF deployment is changed and VNF instances are dynamically added to (scale-out) or removed from (scale-in) the system. In such cases, the flow or session assignments are changed to rebalance the traffic among the available instances, which can lead to lost state information.

One obvious solution to handle this in the scale-out case is assigning only new connections to recently added instances and keep existing connections on old instances that already contain the corresponding state. Even though this solution is easy to implement, it will lead to imbalanced load situations because connections cannot be moved away from overloaded instances, limiting the benefits of horizontal scaling. For the scale-in case, the obvious solution is to keep instances in the system as long as there are ongoing connections assigned to them. But this comes with the downside that scale-in operations, more specifically the termination of VNF instances, may be blocked for an unpredictable amount of time by long-living connections. This will, depending on the pricing model of the used infrastructure, lead to non-optimal resource consumption and thus increased OPEX. These problems create the need of a state management system which is able to automatically share and move application state between VNF instances.

This chapter introduces the "E-State framework". This framework provides an approach to share application state between elastically deployed VNF instances by using logically distributed state memory that is accessed by each VNF instance. With this solution, no central control application is needed and the system becomes more fault-tolerant and scalable. We introduce a proof-of-concept implementation of our approach and compare it to three other approaches: A system without state management, a system with centralised state memory, and a system that uses a generic distributed memory solution not optimised for VNF state management.

## 3.2. Related work

There exist several solutions for VNF state management [OR12; Raj+13; Gem+14]. In the first approach, Olteanu et al. [OR12] propose a mechanism that is inspired by virtual machine migrations. It moves VNF state from one VNF instance to another in three steps. In the first step, the complete state is copied to the target instance so that this instance is ready to process new flows. In the second step, all new flows are forwarded from the source instance to the target instance, which processes them and allocates new state. The third step freezes the processing of the remaining old flows on the source instance and moves their flow-related state to the target instance. At the end, old flows are redirected to the target instance by changing forwarding rules on an SDN switch.

The second solution is called Split/Merge [Raj+13]. It is implemented as a shared library that acts as a memory allocator for VNFs. It exposes an API that allows allocating flow-related and globally-shared state. A central controller decides which state should be moved between the instances. The approach implements a mechanism to merge state by using custom combiner functions defined by the VNF developer.

The third framework is called OpenNF [Gem+14]. It provides coordinated control of VNF state and network forwarding rules. This framework uses a central management application to move state and flows from one instance to another. To integrate a VNF into the system, it has to implement a set of API functions to pull and push state information. When the central management application decides to move a flow from one instance to another, it pulls the state from the source instance and pushes it to the target instance. During this process, arriving packets are buffered at the controller until the state is transferred to the target. Then the buffered packets are forwarded to the target instance. By using these mechanisms, OpenNF is able to perform loss-free and order-preserving flow moves. Two extensions of this approach [GA15; KDS15] add solutions for direct state transfers between VNFs to protect the controller from becoming the bottleneck when incoming packets are buffered. However, the system management remains centralised in both extensions.

All these approaches utilise a centralised control component to decide which parts of state are moved between VNF instances. Our framework is unique in not relying on such a centralised state management controller; rather, it utilises logically distributed state memory to receive state information from other instances when they are needed. Olteanu et al. [OR12] do provide a simple migration mechanism for VNF state. They do not provide solutions to share global state between VNF instances and each state item is always only visible on exactly one instance. Split/Merge [Raj+13] provides custom combiner functions to merge state from different instances, which is also possible with our solution. But Split/Merge's combiner functions are only executed when flows are consolidated on one VNF instance and are not used to provide a global view on the entire state space when needed. OpenNF [Gem+14] uses central applications to control the state management. This requires knowledge about the VNFs running in the system to decide which parts of the state should be moved. In contrast, our system reacts to flows moving in the underlying network and does not depend on state management decisions taken outside of the VNF instances.

Further state management solutions for VNFs are called CoGS [SGZ17] and SliM [Nob+17]. CoGS also provides a global view on the state of multiple VNFs, but uses a centralised coordinator to host this state. SliM aims to reduce overheads introduced by duplicated packet processing during state migration and proposes an interface to only share those packet information between the VNFs that are relevant for state changes. CoGS and SliM focus on specific

subproblems and especially SliM is complementary to our solution. Both papers reference our original work on E-State [PK16a] and were presented one year after our solution.

Solutions like OpenState [Cas+15], on the one hand, focus on enabling stateful functions on programmable switches inside the data plane and thus are complementary to E-State which targets software-based VNFs running on commodity servers. FlowBlaze [Pon+19], on the other hand, focuses on implementing stateful network functions in hardware to target, e.g., SmartNICs. This is out of scope of E-State.

Other more generic solutions to share common information between VNF instances are distributed memory systems, like a REDIS Cluster [Red15] or Apache Cassandra [The15a]. These approaches provide good scaleability but have no notion about the structure of the managed state. Our solution, in contrast, explicitly exploits the state structure and keeps flow related information on the VNF instance that needs it.

## 3.3. A distributed state management framework

We introduce the "E-State framework", a flexible and scaleable state management solution that enables elasticity for stateful VNFs. E-State is built as a software library used to access and share state information. In our design, every VNF instance becomes one node of the distributed state memory and thus the system automatically scales with the number of VNF instances.

### 3.3.1. State management with global view

A VNF can use E-State to store arbitrary chunks of state data, e.g., a serialised data structure representing a runtime object. We call these chunks *state items*. A simple solution to share state items between VNF instances would be to allocate them in a distributed data structure and write all updates directly to this structure. The obvious problem of this approach is the additional delay that is introduced when one VNF instance frequently reads or writes items stored on another instance, similar to, e.g., page trashing in distributed shared memory.

A better solution is to exploit the fact that most state items are directly related to processed flows or sessions. All accesses to these state items are performed by a single VNF instance and accesses to other VNF instances are only needed when the traffic assignment changes. As a result, E-State provides an access pattern offering fast reads and writes to flow-related state items of the local

VNF instance and the possibility to read state items on other instances when needed.

#### 3.3.1.1. General Design

In the E-State framework each VNF instance stores all its state items in its own local state memory that is never written by other VNF instances. This results in small access delays and ensures that each VNF instance has a strictly consistent view of its own state items.

Even though this simple design would ensure that the internal VNF state is visible to our system, it does not yet provide a solution to exchange state items between VNF instances. To overcome this, the system offers a special read operation that allows a VNF to request state items from all VNF instances in the elastic deployment. Using this, a VNF is always able to request *global state* information about the entire elastic system. An example for this is receiving the match counter value from each VNF instance of an elastic IDS system. The only thing the VNF developer has to take into account when global reads are used is that they provide an eventual consistency model instead of the strict consistency model provided by local operations. However, our framework offers the flexibility to implement stricter consistency models so that it can also be used as an experimental platform for different state sharing approaches.

#### 3.3.1.2. Reduce operation

A global read operation receives a list that contains up to $n$ state items where $n$ is the number of VNF instances in an elastic deployment. Such a list does not provide a consolidated view on the system and needs some processing to compute a *global view* of the requested state items. To do so, VNF developers can specify a *reduce* function that maps a list of state items to a single, consolidated state item representation. A typical example is a reduce function that is passed a list of counter values and returns their sum or mean. This concept is comparable to custom merge functions presented in [Raj+13] but provides more flexibility since a global read can be applied whenever a VNF needs it.

Figure 3.1 demonstrates the E-State concept. It shows three instances of a VNF (e.g., three VM instances or containers) all using the E-State library interconnected through a network, e.g., the management network used to control the VNFs within the NFV platform. Each instance allocates different state items in its local state memory depending on the flows they process (*State.A* - *State.E*). Some state items appear on multiple instances since a local version of the contained state is allocated by each VNF instance. The figure shows how these replicated state items can be fetched by other instances with

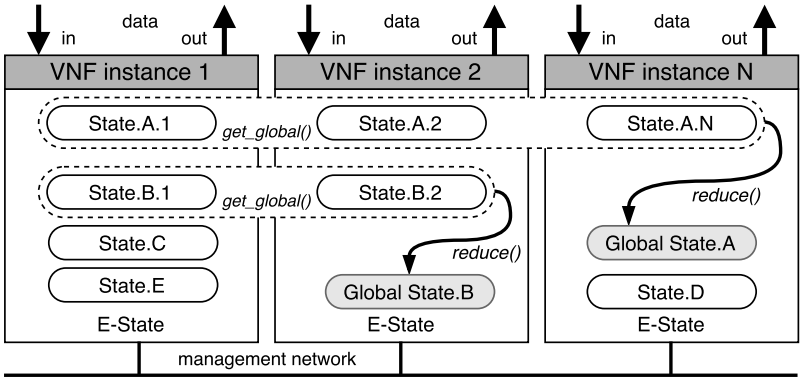



Figure 3.1.: State management with local and global view

the *get_global* operation and how a consolidate state item that reflects the global view on the system is computed on-the-fly by reduce functions.

Such reduce functions cannot only be used to combine state items but also to select a particular item out of the collection of state items stored on different instances. The most common use case for this is finding the state item that was updated most recently. To do so, a reduce function needs a happened-before relationship between state item updates which can, e.g., be based on real-time timestamps with the risk to produce wrong relationships caused by clock drifts. A better solution for this is using a vector clock mechanism that provides happened-before relationships between state items on different instances [Lam78]. Another solution for this is selecting the latest updated replica based in its values. One candidate for this are state items which are known to contain monotonically increasing values, e.g., sequence numbers of the processed packets. A reduce function could simply return the item with the highest sequence number. Nevertheless, this approach depends on the kind of managed state and does obviously not work when packets are reordered. Our system can implement all these solutions and the developer can decide which one is needed.

#### 3.3.1.3. Flow reassignments

The main use case of our state management system are scenarios in which VNF instances are added or removed and the flow assignment is changed. In such cases, redirected flows appear on their target VNF instance, which needs to fetch the corresponding flow-related state items from the source instance. This is done with the global read operation and does not require an explicit fetch or move functionality used by other approaches [Raj+13; Gem+14]. The entity that is responsible to reroute the traffic, e.g., an SDN controller, can optionally support this process by marking packets of moved flows, e.g., by

setting virtual LAN (VLAN) tags. Using this, the target VNF instance can easily distinguish new flows from redirected flows and does only need to perform global reads when a redirected flow is detected. Such a system is presented in Chapter 4.

### 3.3.2. Programming model and APIs

The E-State API is inspired by a key-value store and provides three basic functions: *set*, *get*, and *delete*. Further, it provides a *get_global* function. They are defined as follows:

- `set(key, state_item)`**:** Creates or updates state items stored locally in the shared library.
- `get(key):state_item`**:** Returns the value of a state item stored in the local library. This gives a local view to the system.
- `del(key)`**:** Removes the specified state item from the local state store.
- `get_global(key, *red_func):state_item`**:** Returns the result of the specified reduce function that is applied to all state items stored on all connected VNF instances matching the given key. This function has no side effects on any VNF instance and does not change state items.

The global view is requested with the `get_global` API call; it is passed a reduce function pointer as second parameter. Such a reduce function has to have the following signature:

- `red_func(list<state_item>):state_item`

The function is passed a list of state items and returns a combined representation of them. This allows VNF developers to specify how the mapping from multiple state items to a consolidated global view should be done. It is recommended that the given reduce function is commutative since the order of passed state items is not fixed and may change between calls. Reduce functions should also be idempotent and associative to avoid unexpected results from global read operations.

Our system uses arbitrary strings as keys to identify different state items. We do not fix the used key structure and leave it to the VNF developers to specify their own schemes (e.g., 5-tuples to identify flows). These keys are checked for equality when a specific state item is requested. In addition to this, the `get_global` function allows to use wildcard symbols in its keys based on regular expressions that are matched against existing keys. This allows to request a set of different state items that are then used as input for the reduce function.

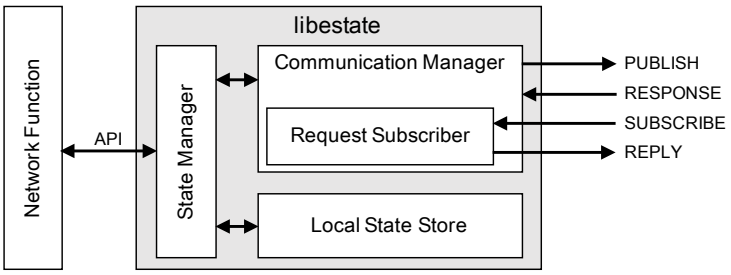



Figure 3.2.: Design of the shared library including a communication manager that interacts with other E-State instances

## 3.4. Prototype implementation

This section describes the prototype implementation of our E-State framework which is available online [Peu15]. The main component of our system is a shared library, called "libestate", that is implemented in C++ and offers a standard C interface against which VNF applications can link. This allows our shared library to be used with almost all programming languages able to link against C interfaces. The interface offers all functions described earlier, including a `get_global` function that expects a pointer to a custom reduce function or the name of a predefined reduce function as one of its arguments.

Figure 3.2 shows the main modules of our library. The *state manager* is responsible for providing the interface to the VNF and to control all internal procedures. It interfaces with the *local state store*, which is a key-value store responsible for holding state items registered by a VNF. These two components alone already enable local state management with the `set` and `get` methods. To allow our library to receive state items form other instances and thus to obtain a global view of the entire state space, we introduce a third module called "communication manager". This module uses the distributed messaging system ZeroMQ [iMa15] as communication backend. It contains a *request subscriber* module which runs in an independent thread and replies to state requests from other instances. The use of ZeroMQ allows our library to not only communicate over transmission control protocol (TCP) connections but also to transparently use direct inter-process communication (IPC) if multiple VNF instances run as different operating system processes on a single host.

E-State uses a publish/subscribe communication pattern together with ZeroMQ's push/pull pattern to do global state requests. To do so, each VNF instance is always subscribed to all other instances of the same elastic deployment. Each instance is then able to publish `get_global` requests and the other instances reply to it. A limitation of this approach is the quadratic number of interconnections needed, which is caused by the broker-less design of ZeroMQ, and can be solved by building clusters with a limited number of VNFs sharing

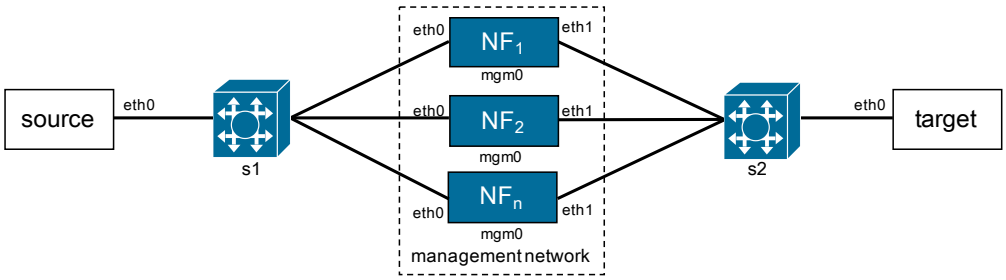



Figure 3.3.: Mininet topology used for prototype evaluation

the same global state or by changing the underlying communication library. We leave those optimisations for future work (Chapter 11.3).

## 3.5. Evaluation

We test our "libestate" prototype in a Mininet [LHM10] environment running on a machine with Intel(R) Core(TM) i5-4690 central processing unit (CPU) at 3.50GHz and 16GB memory. Figure 3.3 shows the topology used for our experiments. It consists of a source host that sends multiple iperf-generated TCP flows to a target host over an elastic cluster of VNF instances that forward and monitor the traffic. These VNF instances are Mininet hosts with two ethernet interfaces configured as ethernet bridges. All VNF hosts are also connected to a management network for communication between libestate instances. The two SDN switches between source and target are controlled by a custom SDN application running on top of a POX [The15b] controller that proactively installs forwarding rules on the two switches to control the traffic distribution and to reroute the flows between available VNF instances.

Each network link in our topology has a maximum bandwidth of 1 Gbit/s and no artificial delay. We use Mininet's CPU sharing limitation feature for each Mininet host to emulate a realistic scenario in which an additional VNF instance corresponds to additional computation resources. Without this, a higher number of VNF hosts in the system would not result in performance improvements because the CPU time available for each single Mininet host would decrease. We limited the hosts as follows: Source and target host are limited to 20 % CPU each and every VNF instance is assigned to 2.5 % of the overall CPU time summing up to 40 % CPU usage when the maximum of 16 VNFs is active. The remanning 20 % are used for other components, like the SDN controller or the centralised state memory used in some of our experiments. We use a custom VNF implementation that runs in each VNF host and performs pattern matching on the processed flows, like an IDS.

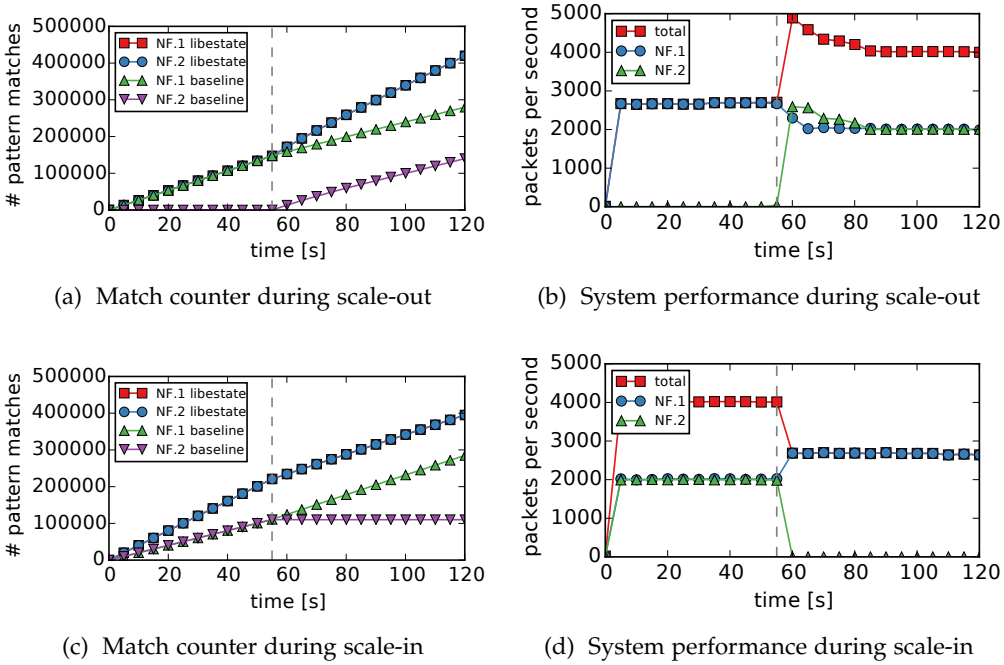
Our first experiment demonstrates how an IDS can benefit from our state management system when it is scaled at runtime. For the scale-out case, the experiment starts with a single VNF instance (*NF.1*) over which all TCP flows are forwarded. After about 55 seconds, the scale-out procedure is initiated and half of the flows are rerouted to a new VNF instance (*NF.2*) by installing additional rules on the two SDN switches. Figure 3.4 (top) shows the scale-out scenario. The vertical dashed line marks the point in time at which scaling starts. The left part of the figure shows the values of the pattern match counter on both instances. The experiment is executed two times. At first, with a baseline system without any state sharing functionality and second, with our libestate system. In the baseline case, the second instance starts its match counters from zero after flows are moved to it, even though the first instance has already detected intrusive packets. This might lead to missed detections and influences the correctness of the overall IDS system. In the libestate case, the state for the moved flows is transferred to the second instance and the operation can continue without information loss. The right part of the figure shows how the overall performance of the elastic VNF deployment increases after the system is scaled out. It shows that *NF.2* can directly proceed with processing after the flows are moved.

Figure 3.4 (bottom) shows the scale-in case in which the experiment starts with two VNF instances and then moves all flows to *NF.1* after about 55 seconds. It shows how the match counter of *NF.2* stops counting in the baseline version and the information is lost.

The second experiment evaluates the scaling behaviour of our state management system with an elastic deployment of 2 to 16 replicated VNF instances. It compares our libestate prototype to three other approaches using the average number of processed packets per second (PPS) to show the overall system performance and the average state item request delay to show the state sharing performance. First, we compare to the baseline IDS implementation not using any state management mechanisms[1]. Second, we compare to a state management system (*centralmem*) that uses a single REDIS instance [Red15] to maintain the state of all VNF instances. And third, we compare to a distributed REDIS cluster (*clustermem*) that runs one REDIS node on each VNF instance and can be used through the same API as the centralised version.

The left plot of Figure 3.5 shows how the performance of the IDS system increases when additional instances are added and how the scaleability of a system with centralised memory is limited. This is, on the one hand, caused by additional delay introduced by turning each state access into a network request and, on the other hand, by the maximum number of requests a centralised solution can serve. It also shows that a distributed memory solution does not

[1]Global values of the baseline experiment are calculated offline by summing up the local values logged on each VNF instance.




(a) Match counter during scale-out

(b) System performance during scale-out

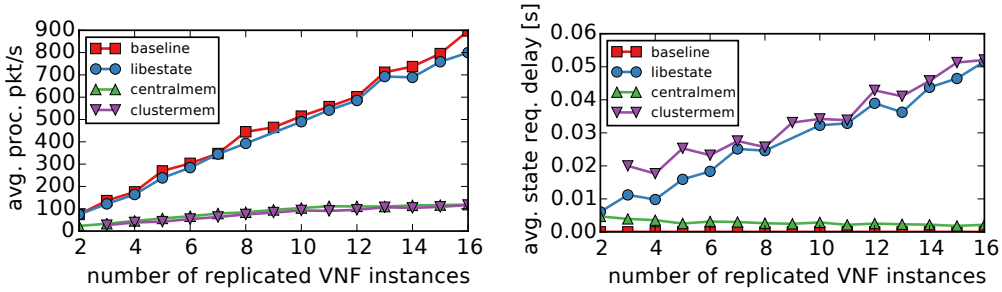
(c) Match counter during scale-in

(d) System performance during scale-in

Figure 3.4.: Match counter value of two IDS instances (left) and overall system performance before and after scale operation (right).

provide benefits because state items are not stored on those VNF instances that access them most often. The performance of our library, in contrast, is near to the baseline performance since most of the state accesses are done locally and the global state is requested less often. It is important to note that the baseline implementation does not share any state information when flows are moved. Our system, in contrast, maintains all state information and provides comparable performance, which is a clear advantage. However, an increasing number of VNF instances results in an increased delay for each global request performed by our library (Figure 3.5, right). This is expected because the system always requests state items from all instances of an elastic VNF. It is interesting that the request delays of the *clustermem* version are higher than the delays of the *centralmem* version. The reason is that state items have to be fetched from multiple cluster instances to obtain the global view instead of requesting all items at once from centralised memory.

The third experiment shows the delays that can be expected for different state item sizes in a system with four VNF instances. It shows that the request delays quickly increase when the size of the transmitted state increases. However, typical flow state items are only tens of KByte [Raj+13] and our experiment shows that item sizes around 60 KByte ($2^{16}$ bytes) can still be requested in less than 150 ms. The delays for these requests are on the same order as the times




(a) PPS (b) State request delay

Figure 3.5.: System performance (packets per second) (left) and state item request delay (right) for different numbers of replicated VNF instances

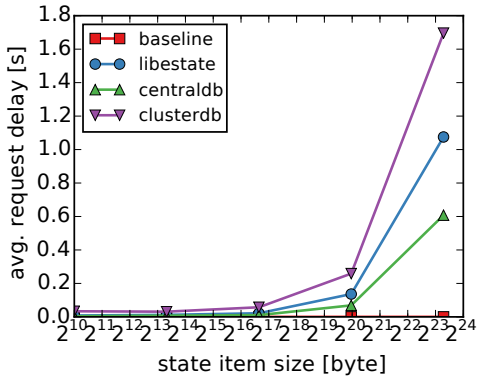



Figure 3.6.: Request delays of different state item sizes

needed by OpenNF's move operations presented in [Gem+14].

## 3.6. Conclusion

This chapter presents E-State, a novel approach to manage application state in elastic VNF deployments. Our solution shows that NFV state management can be done without central control components, which are used by existing approaches [Raj+13; Gem+14]. The presented prototype can easily be extended to provide additional consistency models for state requests to study tradeoffs between consistency requirements and management overhead. The results of our experiments show that our system outperforms approaches that use generic centralised or distributed state memory solutions.

Even if the recent developments in the cloud and NFV domains show that the best way to build network functions for NFV, and especially for cloud-native deployments [5GP18], is to redesign the functions and to remove all stateful

parts, there will still be stateful network functions in legacy deployments. The presented solution helps to run such legacy functions, not designed for cloud-native deployments, inside elastic environments and can thus be considered as an important tool for the transition from legacy to fully softwarised networks.

During the execution of the experiments, presented in this chapter, we have noticed several shortcomings of the Mininet platform when it comes to prototyping NFV scenarios, e.g., the limited isolation of Mininet hosts resulting from the shared file system. This is one of the motivations for creating a rapid prototyping platform for the NFV domain as presented in Part II of this thesis.

# 4. Operation support for stateful VNFs

Enhancing stateful VNFs with state sharing mechanisms allows to migrate state between VNF instances during dynamic operations, like scaling. But to actually implement such dynamic operations, additional platform support for traffic control, e.g., flow rerouting between VNF instances, is needed. To this end, I present SHarP, a flow handover protocol that allows seamless, order-preserving, and loss-free flow handovers between VNFs. This part is based on Hannes Küttner's Bachelor thesis [Küt17], which was supervised by me. In the thesis, Mr. Küttner developed the initial SHarP prototype using the idea of marking and buffering packets to ensure order-preserving and loss-free handovers as I described it in the thesis announcement. The high-quality prototype implementation done by Mr. Küttner allowed us to do an in-depth analysis of the developed concepts, for which Mr. Küttner collected the data under my supervision and helped during its analysis. We published the results of this work at a conference [PKK18b] as well as in a journal paper [PKK19], both written by me. This chapter is based on this journal paper and contains figures and verbatim copies of the text from the paper. After discussing the relevant related work in Section 4.2, we present the design of SHarP in Section 4.3 and evaluate the presented solution in Section 4.4.

## 4.1. Introduction

There exist state management solutions, like Split/Merge [Raj+13] or OpenNF [Gem+14], to tackle the problem of stateful VNFs in elastic NFV deployments. They jointly manage the state migration between VNF instances and the rerouting of traffic between them. The downside of these approaches is that they impose complex modifications of the VNF implementations in order to provide the required interfaces to extract and inject state information into the involved instances. We argue that this is a major obstacle for an interoperable and open NFV landscape. It requires VNF vendors to custom-tailor their VNFs to the NFV platform on which they should be on-boarded if they want to benefit from the state management solutions offered by these platforms. Alternatively, VNF vendors could implement their very own state management and traffic rerouting solution, not tight to any platform, but this results in silo solutions in which VNFs of different vendors might not be compatible to each other, e.g., chaining them is not possible.

To remove this obstacle, we present "SHarP", a very lightweight traffic-steering solution for elastic VNF deployments that leaves the actual choice of the state migration solution to the VNF vendor, while still providing these solutions with additional control triggers, if needed. The resulting system provides a clearer separation of concerns than existing solutions by implementing the traffic rerouting functionality as part of the NFV platform and leaving the state management task to the VNFs. As a result, VNF vendors benefit from SHarP's sophisticated rerouting mechanism offering features, like loss-freeness and oder-preserving, while still being able to run their own state migration systems custom-tailored to the involved VNFs and their internal state. The presented approach combines the benefits of both worlds, which makes SHarP a better fit for practical, real-world deployments.

The key contributions of this chapter are as follows: We introduce our seamless flow handover protocol design that does not require a dedicated control interconnection between the SDN controller and the involved VNFs. Our handover protocol assigns the majority of the packet buffering tasks, required to provide a loss-free and order-preserving flow rerouting mechanism, to the destination VNF instances and thus reduces the load on the centralised SDN controller. More specifically, the controller is only used for buffering during the first phase of our the proposed rerouting mechanism as described in Section 4.3.3. During the other phases, e.g., during the phase in which the state is migrated, all buffering happens at the destination VNF. In addition, we introduce the "handover support layer (HSL)": a helper component that can easily be integrated into existing VNF implementations and requires fewer modifications than existing approaches, like the FreeFlow library used by Split/Merge [Raj+13]. Finally, we provide an extensive evaluation based on a set of testbed experiments. These experiments verify that the controller buffer usage of the proposed approach scales well with the packet rate of the data plane and stays constant irrespective of the time required for state transfers between the VNFs. The results also show that our handover solution has only minor impact, e.g., in terms of introduced delays, to the moved flows.

## 4.2. Related work

Steering and moving flows between dynamically allocated VNFs is already well studied and several approaches which target different use cases like load balancing, service chaining, or scaling exist [JTS08; Qaz+13; Fay+14]. However, none of them provides supporting information and triggers to integrate with additional state management mechanisms and not all of them provide seamless handover mechanisms that do not introduce additional packet loss. As a result, these approaches are not useful for stateful VNFs.

Table 4.1.: Comparison of related NFV state management and flow handover solutions

| | Split/Merge [Raj+13] | Pico Rep. [RWJ13] | OpenNF [Gem+14] | OpenNF Ext [GA15] | OpenNF DiST [KDS15] | OpenNF Tag [Wan+17] | CoGS [SGZ17] | SliM [Nob+17] | SHarP |
|---|---|---|---|---|---|---|---|---|---|
| Flow handover support | ● | ○ | ● | ● | ● | ● | ○ | ○ | ● |
| Loss-free handover | ○ | ○ | ● | ● | ● | ● | ○ | n.a. | ● |
| Order-preserving handover | ○ | ○ | ● | ● | ● | ● | ○ | n.a. | ● |
| Distributed packet buffering | ○ | ○ | ○ | ○ | ● | ● | ○ | ○ | ● |
| Fixed to state migration solution | ● | ● | ● | ● | ● | ● | ● | ● | ○ |
| No changes in VNFs required | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ○ | ● |
| Designed for scaling use case | ● | ○ | ● | ● | ● | ● | ● | ● | ● |

n.a. = not applicable

Other solutions that are designed to migrate state of virtual machine instances exist. But they come with a large overhead because they move much more state information than needed to operate a VNF [LLJ14]. In addition, more specific approaches that focus on joint traffic steering and state migration of VNFs have been proposed. The most prominent ones are Split/Merge [Raj+13], Pico Replication [RWJ13], OpenNF [Gem+14] with its extensions [GA15; KDS15], CoGS [SGZ17], as well as a novel approach called SliM [Nob+17] and a tagging-based solution presented in [Wan+17]. We give a structured overview of these NFV state management and handover solutions in Table 4.1.

In contrast to these approaches, which focus on joint state management and traffic steering, our approach (SHarP) focuses on the latter only. As a result, SHarP offers much more flexibility by leaving the choice of the used state management approach to the VNF vendor instead of fixing it for the complete execution environment; even different state management schemes for different VNFs or groups of VNFs are possible. This simplifies the on-boarding of VNFs to different platforms since the platforms do not introduce any requirements for specific state-management interfaces. An example for a complementary state management solution is E-State [PK16a], presented in the previous chapter; it works seamlessly with SHarP. Other distributed state management solutions, like the recently introduced CoGS [SGZ17], SliM [Nob+17] or Fog-Store [May+17] approaches, are also complementary to SHarP and could benefit from its loss-less flow migration procedures. In contrast to OpenNF, our system distributes most of the buffering process required for loss-less handovers to the destination VNF instances; this heavily reduces the controller load and provides better scalability. The work of [Wan+17] analyses OpenNF

and is of a more theoretical nature. It backs our findings of drastically reduced controller load when most buffering happens at the destination VNF. In contrast to our SHarP prototype, their solution does not provide a flow detection mechanism to support the selection of the right parts of the overall state to be migrated.

## 4.3. Seamless handover protocol (SHarP)

The design of our handover protocol follows two main goals. First, the flow handover mechanism has to explicitly support state migration procedures but should not mandate any specific state migration solution. Second, our solution will offer improved scalability compared to existing approaches. More specifically it aims to reduce the load on the central controller by minimising the number of packets the controller has to buffer during a loss-less and order-preserving handover.

To achieve these design goals, we defined the following set of requirements: The first requirement for a handover mechanism is a *flexible flow selection (R1)* interface that allows to select single flows as well as groups of flows that shall be moved from one VNF to another. These handovers should be performed as fast as possible to *minimise service interruption times (R2)* and they have to ensure that they do *not introduce additional packet loss or packet reordering (R3)*. To be able to handle many flows, the *scalability (R4)* in terms of control load and buffer usage is important. Finally, a handover mechanism has to be designed for *compatibility (R5)* without requiring specific modifications from VNF implementations to accommodate a wide range of different VNFs.

### 4.3.1. Handover scenario

SHarP is designed to work with networks that contain at least two SDN switches: an ingress and an egress switch as shown in Figure 4.1. Our design extends to any number of switches, yet to simplify presentation, we limit ourselves here to two switches; evaluation results do not depend on number of switches. Between the switches, multiple VNF instances are located and their data plane interfaces are connected with one port to the ingress switch and one port to the egress switch as shown in Figure 4.1. In addition to this, the VNFs are connected to a management network that allows them to directly exchange information between each other. Data flows enter the SHarP-enabled VNF deployment from a source ($Host_1$) through the ingress switch, traverse one VNF instance (or a chain of multiple VNF instances), and leave the system through the egress switch towards the destination ($Host_2$). Bi-directional flows in which packets are sent from the destination ($Host_2$) to the source ($Host_1$) are

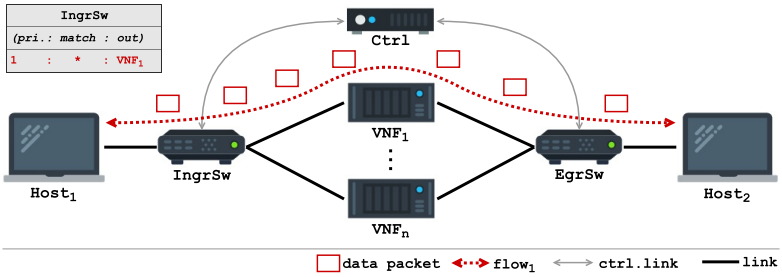



Figure 4.1.: Example network with multiple VNF instances, ingress and egress switch as well as a data flow processed by $VNF_1$ (icons taken from [Küt17])

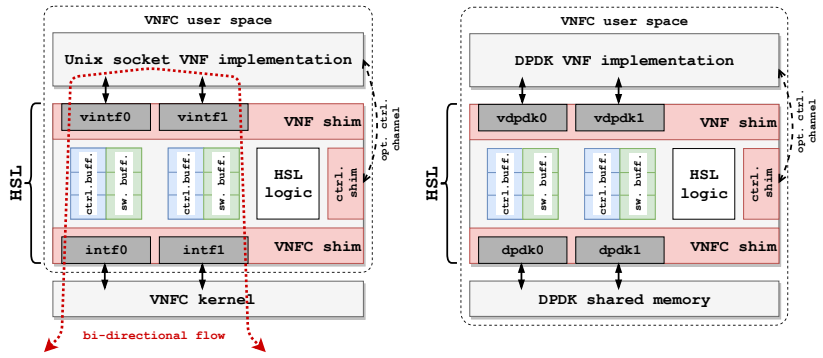
also supported (Figure 4.1). Flows can be moved between VNF instances using the proposed handover mechanism by triggering the handover procedure through the northbound API of the controller. For example, the flow shown in Figure 4.1 will be moved from $\text{VNF}_1$ to $\text{VNF}_n$.

The involved VNFs do not need a direct connection to the controller as this is not commonly the case and thus would impose a needless requirement. Instead, control messages sent by the controller to the VNFs are forwarded by the switches and intercepted by an intermediate software layer that is running inside the VNF's container (or VM). This layer also buffers packets as required to ensure loss-free and order-preserving handovers (described in Section 4.3.2). We assume that the links of the example networks do not introduce any additional packet loss or packet reordering.

### 4.3.2. Transparency towards VNF and state management

One of the main requirements for SHarP is to be as transparent as possible towards VNF implementations that operate in a SHarP-enabled environment (R5). This also means that SHarP must not enforce the use of a particular state management or state sharing framework. Instead, it provides the means to assist state sharing solutions, like E-State [PK16a], with functionalities to pause and buffer incoming flows or to inform the actual state migration solutions when a handover is performed by the network.

This functionality is completely encapsulated in an additional software layer, called HSL, that is located between the actual VNF implementation and the network interfaces of the VNF container (VNFC) as shown in Figure 4.2. This software layer acts as a bridge and is able to forward and intercept packets between the interfaces of the VNFC and the VNF implementation. Depending on the used technology, the HSL appears to be completely transparent to the used VNF implementations. For example, if the implementation is done with Unix sockets, as shown in Figure 4.2a, the VNFC shim of the HSL directly connects to the network interfaces of the VNFC (i.e., the network interfaces




(a) Unix socket-based setup (b) DPDK-based setup

Figure 4.2.: HSL sitting between VNFC and VNF implementation

of the VM in which the VNF runs). Then the HSL creates one virtual network interface for each of the network interfaces of the VNFC and forwards packets between those pairs of interfaces. The actual VNF implementation connects to those virtual interfaces as if they are the normal network interfaces of the VNFC. Using this approach, the HSL can be transparently placed between the network interfaces of the VNFC and the VNF implementation and gets access to all received and sent packets.

The HSL also implements a control logic that intercepts control messages sent by the SHarP controller through an SDN switch over the data plane of the system. Those control messages allow the SHarP controller to trigger events, like preparing the destination VNF for a handover, without requiring a direct connection between controller and VNF. Besides this control logic, first-in, first-out (FIFO) packet buffers are implemented and used to buffer incoming packets when the destination VNF is not yet ready to process them, i.e., the state transfer from the source VNF has not completed. The HSL distinguishes between *ctrl. buffers* and *switch buffers* to separate packets that are coming from the controller from packets that are directly arriving from the data plane switch. This distinction is done using packet flags that mark packets that have previously been processed by the SHarP controller, as described in Section 4.3.3. Figure 4.2a shows a bi-directional example flow that enters the VNFC through its default network interfaces provided by the VNFC kernel. The flow is then intercepted by the HSL and directly forwarded to the virtual interfaces to which the actual VNF implementation is connected. In this example, no packets are buffered.

The HSL optionally offers a control channel to the VNF implementation used to inform the VNF about the status of the handover, e.g., to trigger its state migration mechanism. We leave it to the VNF to prepare and migrate all state belonging to the flows that are handed over. This allows us to transparently

handle multiple VNF implementations without needing information about the internal state structure, a major difference to OpenNF [Gem+14]. This control channel can be realised with different kinds of interprocess communication solutions, such as Unix sockets, pipes, or message queues. It is an optional SHarP-specific modification imposed to the VNF implementation. The amount of further VNF modifications depends on the used state migration solution rather than on SHarP. This reduces the overhead of integrating a VNF into a SHarP environment. For example, a VNF vendor or integrator only needs to install the HSL inside the VNFC and reconfigure the VNF to use the right network interfaces—a process that can be partly automated and requires far less effort than changing code of the VNF implementation.

All parts of the HSL are implemented as modular, plugin-like components (shims) that can easily be replaced to make the HSL agnostic to different data layer interfaces. Besides the standard Unix socket shim shown in Figure 4.2a, more NFV-specific implementations are possible. For example, HSL shims that are based on DPDK [Lin17] as shown in Figure 4.2b. In the DPDK case, the HSL directly accesses the DPDK shared memory in the VNFC's user space and does not need to access any Unix network interface. It is worth noting that Figure 4.2 shows an example setup for a VNF that has two data interfaces (`intf0` and `intf1`), but the general design of the HSL allows arbitrary numbers of interfaces and their corresponding buffers.

### 4.3.3. Handover procedure

SHarP's handover procedure is split into the following three phases: Phase 1: *Initialisation*, Phase 2: *Pause*, and Phase 3: *Resume & Release*. The idea behind this split is that a handover first needs to be prepared, e.g., the flows to be handed over need to be detected and selected (R1). After this, the flow processing of the VNFs needs to be paused so that the actual handover from one VNF to another can be performed without loosing any state changes (R3/R5). Finally, after the flows are migrated to the destination VNF, processing has to be resumed and packets that might have been buffered during the previous phase need to be released (R3). The SHarP controller uses its internal knowledge as well as signalling messages, which are injected into the date plane, to trigger the transitions from one phase to another. This way, it keeps the overall handover time short (R2). More specifically, it moves from Phase 1 to Phase 2 once the VNFs signal that they are prepared and ready for the handover. It moves from Phase 2 to Phase 3 when all traffic is rerouted, the state has been transferred, and the packet processing can continue at the destination VNF.

We now describe each of the three phases in more detail. Figure 4.3 shows the three phases for handing over a single unidirectional flow from $\text{VNF}_1$ (source VNF) to $\text{VNF}_n$ (destination VNF). It also shows the forwarding table entries

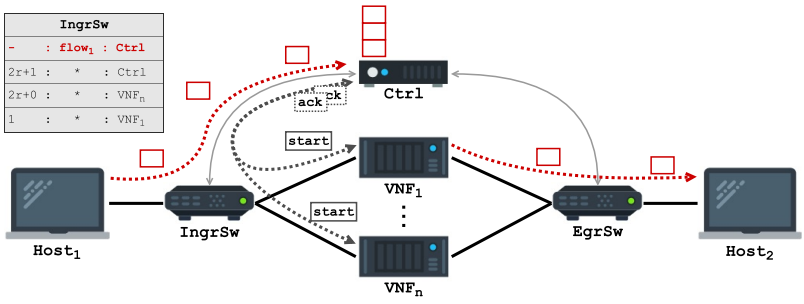
of the ingress switch (IngrSw). We decided to show the handover of a unidirectional flow to keep the figures clean and understandable. SHarP supports the handover of bidirectional flows by performing symmetric handover steps on the ingress and the egress switch.

At the beginning of the first phase, the scenario looks like the one shown in Figure 4.1 in which all flows between $Host_1$ and $Host_2$ are processed by $VNF_1$. A handover is triggered by a request to the northbound API of the SHarP controller (Ctrl). The request contains an OpenFlow-like matching rule for a flow (or a group of flows) to be moved, a priority $r$ for the handover request, as well as the identifier (e.g., medium access control (MAC) address or switch port ID) of the destination VNF to which the flows should be moved. The priority $r$ enables our system to organise the handover procedure among multiple handover requests and allows the user of the system, e.g., an NFV orchestrator, to overwrite existing handover rules. A SHarP handover request does not require any further knowledge about the state of the network, in particular, the requesting external entity, e.g., an NFV orchestrator, does not need to know by which VNFs the flows matching the request are currently processed (R1/R5). In the example given in Figure 4.1, the handover request will move all flows from $VNF_1$ to $VNF_n$ by using a wildcard (*) in its match field.

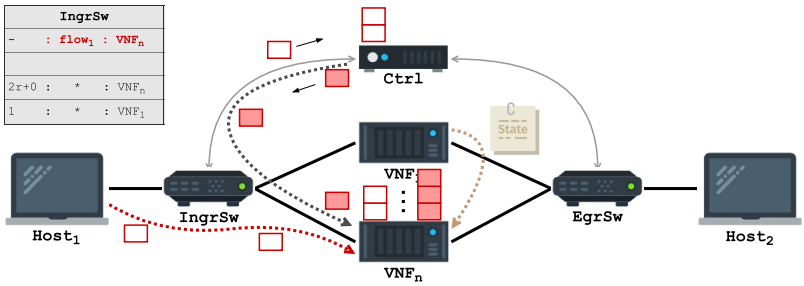
Once the request arrives at Ctrl, it installs a so-called "flow detection table entry" on IngrSw that matches all flows specified by the handover request and forwards their packets to Ctrl. The priority of this entry $p_t$ is set to $p_t = 2r + 1$ so that there is room for another table entry belonging to this handover request with priority $r$. Using this fixed mapping of handover rule priorities $r$ to forwarding table entry priorities $p_t$ on the switch ensures a clear separation of forwarding entries belonging to different handover requests. Next, a second table entry is installed that matches the same flows but forwards their packets to the destination $VNF_n$. This entry has priority $2r + 0$ such that it will only be used once the *detection table entry* is removed.

Figure 4.3a shows how incoming packets from $Host_1$ are matched and forwarded to Ctrl, which buffers them. Packets that are still processed by $VNF_1$ leave the system via EgrSw. In this state, the controller learns about all flows that are affected by the handover and can generate exact match entries for each of these flows to hand them over one by one. To do so, one exact table entry for each flow is installed in IngrSw which forwards all packets of this particular flow to Ctrl. These exact entries implicitly have the highest priority since no wildcard fields are used anymore[1]. The detection phase stays active until a *maximum silence time*, which is set as the idle timeout of the *detection entry*, is reached and the *detection entry* is removed from IngrSw. Flows that have not been detected during this time are treated as new flows by our system. They

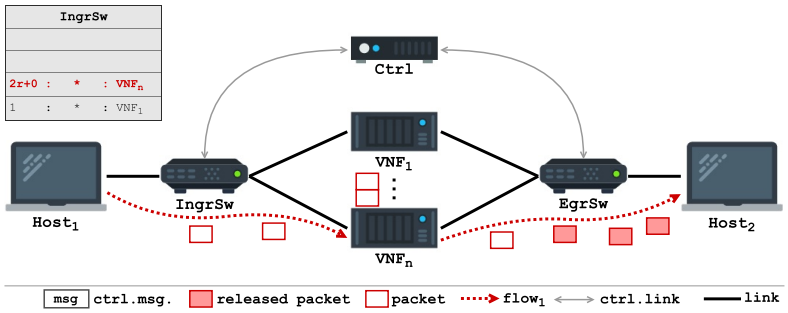
[1]OVS documentation: http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt




(a) Phase 1 (*Initialisation*): Flow detection and handover initialisation. New packets are buffered at the controller.




(b) Phase 2 (*Pause*): Installed temporary forwarding entry to destination $\text{VNF}_n$. Buffering done at destination $\text{VNF}_n$ to allow early controller buffer release and load distribution. Trigger of state transfer solution.




(c) Phase 3 (*Resume & Release*): Final forwarding state reached and state migration finished. Release and replay of buffered packets at destination $\text{VNF}_n$.

Figure 4.3.: Three phases of SHarP's handover procedure for a flow moved from $\text{VNF}_1$ to $\text{VNF}_n$ (icons taken from [Küt17])

are directly forwarded to $VNF_n$ by the table entry with priority $2r + 0$. When the detection phase is over, `Ctrl` sends `START_HO` messages to the involved VNFs using a `PACKET_OUT` event on `IngrSw` to inject them into the data plane. The controller knows the destination VNF from the handover request and the source VNF by utilising the controller's internal knowledge about the previous network configuration. The HSL in the VNF intercepts the control message and can, e.g., trigger the preparation of the state transfer before replying with acknowledgments as shown in Figure 4.3a.

Once `Ctrl` receives the `ACKs` it enters the second phase of the handover procedure that is shown in Figure 4.3b. Immediately after this phase has started, `Ctrl` starts to mark (e.g. by VLAN tag or encapsulation) and release the packets from its buffer and sends them towards the destination $VNF_n$ via `IngrSw`. $VNF_n$ detects the marked packets and puts them in its internal `ctrl_buff` because it knows that they have been buffered at `Ctrl` before. At the same time, `Ctrl` updates the exact forwarding table entry to forward all new packets of the flow arriving at `IngrSw` directly to $VNF_n$. At $VNF_n$, the packets are buffered in the internal `sw_buff` of the VNF to not mix them up with the packets previously buffered at the controller (important for R3).

One problem at this point is that `Ctrl` needs to know when it has received all packets that are not already forwarded to $VNF_n$. But there may be packets that are still in flight between `IngrSw` and `Ctrl`. To solve this, `Ctrl` instructs `IngrSw` to duplicate and flag packets (`BUFFER_FOLLOW_UP`) that are forwarded to $VNF_n$ and to send the flagged copy of them also to `Ctrl`. In this configuration, `Ctrl` can inject a test packet into the data plane at `IngrSw` and will immediately know that it has seen all packets not yet forwarded to $VNF_n$ once it receives the test packet. Thus, `Ctrl` knows that it does not need to buffer any new packets and removes the packet duplication configuration from `IngrSw`.

During the entire second phase shown in Fig 4.3b, no traffic is processed by any of the VNFs and all arriving packets are buffered in the two buffers of the destination $VNF_n$. In this state, the VNFs can trigger their state management solutions, which can transfer the VNF's internal states over the management network between the VNFs. HSL can support these state management solutions by giving them information about the source and destination VNF as well as the exact flow identifier.

The third phase of the handover, shown in Figure 4.3c, is entered once `Ctrl` has released all its buffered packets and the state management mechanism at the VNFs indicates that all state has been moved. The HSL then immediately starts to release the buffered packets towards the VNF implementation of $VNF_n$ to be processed using the state that has been moved from $VNF_1$ to $VNF_n$ in the previous step. It first releases its `ctrl_buff` and afterwards its `sw_buff` to ensure that all packets are processed by $VNF_n$ in the same order as they have entered the SHarP system (see [PKK19]). Finally, `Ctrl` can remove

the additional handover table entries from `IngrSw` and reach a stable system state in which all flows involved in the handover are processed by $VNF_n$. More details, like control packet formats and handover rule removal procedures, are described in [Küt17].

One of the benefits of SHarP is that it does not require a dedicated control channel between `Ctrl` and the involved VNFs. Instead, all control messages are injected/fetched from the switch and sent over the data plane to the VNFs, where the HSL intercepts them (R5). The used control messages are encapsulated inside standard Ethernet frames using `EtherType=0x821c`. This `EtherType` value is chosen because it is usually unused [Joe13] and will thus not interfere with other standardised protocols. In case this `EtherType` is already used in the domain in which SHarP should be deployed, this configuration needs to be changed. The payload of each control message contains a command code to express its functionality, an identifier used to reference the handover to which the control message belongs, and a sequence of type-length-values (TLV) fields. Those TLVs hold additional context information, like the matching rule used for the handover or its priority. If control messages are sent from `Ctrl` to a VNF, the destination VNF's Ethernet address is used as destination address of the control message. If control messages are sent from a VNF to `Ctrl`, the address field is left empty, as the switch detects all control messages matching their `EtherType` and forwards them to `Ctrl` using `PACKET_IN` events.

### 4.3.4. Removing buffer load from the controller

For a seamless handover, packets need to be buffered while the state is synchronised between the VNF instances and no state updates can be performed. Later, the buffered packets can be released to the destination instance to be applied to the state. In OpenNF [Gem+14], packet buffering takes place completely at the controller which may lead to performance issues. The controller can quickly be overloaded if the amount of packets to be buffered is large, i.e., because of a long-lasting state transfers. Our system design, in contrast, reduces the buffer load of the controller by moving the responsibility to buffer incoming packets during a state transfer to the destination VNF instance. The SHarP controller only needs to buffer packets during the period of time in which the handover is initialised (Figure 4.3a) and tries to release this buffer as early as possible (R2). In particular, the buffer is released before the actual state transfer is started, which makes the controller buffer usage of SHarP independent of the state transfer. We show this property in more detail in our evaluation (Section 4.4).

Buffering most of the packets directly at the destination instance has the additional advantage of using the capacity of the destination VNF instance. A

VNF only needs to buffer the packets belonging to flows that are redirected to that instance and not of all handovers in the network, further improving scalability of the entire system (R4). Further details about bidirectional handovers, message flows, and preserved packet order can be found in our journal paper [PKK19].

## 4.4. Evaluation

We analyse SHarP with a set of experiments to validate that our handover protocol behaves as expected, e.g., no packet loss or reordering occurs and the controller buffer usage remains constant even when the state migration time increases. We have also presented a direct comparison to OpenNF, based on a theoretic analysis which was entirely done by Mr. Küttner, in our journal paper [PKK19]. We have used this theoretical analysis, not presented in this thesis, because of outdated codebases and limited documentation of OpenNF, which makes an experimental evaluation in our lab setup not feasible. Summarising these theoretic results, the analysis has shown that the number of packets processed by the controller depend on the packet rate of the data plane in both systems. In SHarP, however, the number of packets processed by the controller is expected to be about five times smaller compared to OpenNF [PKK19]. More importantly, the analysis has shown that in OpenNF the number of packets processed by the controller heavily depends on the state transfer time, and thus the size of the transferred state, whereas it is expected to be constant in the SHarP case [PKK19]. We back this finding and characterise the SHarP system with the experimental evaluation, presented in the following.

In our experiments, we use six metrics to characterise the performance of our system: The handover duration (1), maximum packet delay introduced by handover (2), controller buffer usage (3), VNF buffer usage (4), packet loss (5), and packet reordering (6). Our results present these metrics as a function of data plane data rate and the duration it takes the VNFs to migrate their state. The maximum packet delay is the main indicator for the delay introduced into the service as the handover is executed. The buffer usage at the controller and at the VNF indicate how well SHarP fulfils the claim that only a small amount of data has to be buffered and processed at the controller. Further, we show how SHarP behaves under load when multiple handover requests for many flows arrive within less than a second.

We built a prototype of the SHarP controller based on the Ryu SDN Framework [Ryu17]. The prototype offers an easy-to-use, northbound interface that offers the required functionalities to trigger handover procedures between arbitrary VNF instances. In addition to the controller prototype, we implemented a Python-based HSL prototype that acts as a bridge between the VNFC and the actual VNF implementation using standard Unix sockets as shown

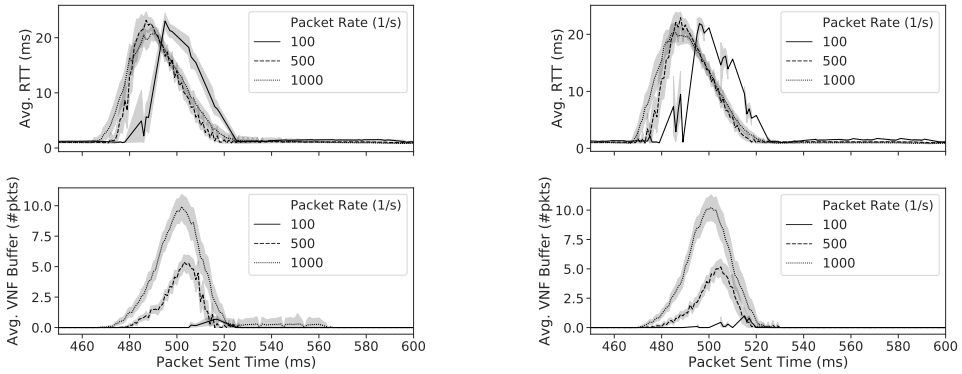
in Figure 4.2a. Those implementations were done by Mr. Küttner during his thesis [Küt17]. The use of Python limits the throughput of the HSL prototype but still allows us to evaluate SHarP in terms of buffer usage and handover performance. The Python-based HSL prototype is also the reason why we limit our experiments to 1000 PPS. A high-performance implementation of the HSL using DPDK [Lin17] is planned as future work (Section 11.3). Both prototypes (SHarP and HSL) are open-source and available on GitHub [KP18].

All experiments are executed on an SDN testbed based on the emulation framework Containernet [Peu16], which will be presented in Chapter 5, running on a server with an Intel(R) Core(TM) i7 CPU 960 @ 3.20 GHz and 24 GB memory. The used network topology is the same as shown in Figure 4.1 consisting of two hosts, two switches, and two VNFs that are able to forward arbitrary traffic between their input and output interfaces. This setup reflects a typical setup of a SHarP system. Our solution does, however, also support more complex networks involving more switches and VNFs. In such a case, a SHarP controller has to coordinate the flow rules on all switches that are directly involved in the handover procedure. These are usually those switches that are placed directly before or after the involved VNF instances, e.g., the software switches in the NFV platform that also take care of the VNF chaining. The SHarP controller does explicitly not need to control all switches of a network.

The involved Containernet links are configured to not introduce any artificial delay or bandwidth limits and we ensured that the link saturation is below 10 % in all experiments so that the links do not become the bottleneck of the setup. Both the hosts and VNFs are represented by Docker (1.12.3) containers connected to the emulated network created by Containernet (2.3.0d1) containing two Open vSwitches (2.5.2). The VNFs are implemented as layer 2 bridges that do not perform any additional packet processing to not bias our handover performance measurements. Our prototype controller is implemented on top of Ryu 4.13. We simulate the state migration process in all experiments, which allows us to evaluate the actual handover protocol independently of a specific state management implementation. Using this approach, we have full control over the state migration process and can test SHarP in different scenarios, e.g., with different state transfer durations.

### 4.4.1. Handover characteristics

The first part of our experimental evaluation analyses handovers performed with our prototype and how they impact the rerouted packets and the involved buffers. During the experiments, a constant user datagram protocol (UDP) traffic flow is generated on $Host_1$ and sent to $Host_2$ over the first VNF. $Host_2$ receives the packets and sends them back to *Host1* creating a bidirectional traffic flow which is then handed over to $VNF_n$ by our SHarP controller.




(a) Handover behaviour for different packet rates using 58 byte packets

(b) Handover behaviour for different packet rates using 1000 byte packets

Figure 4.4.: Packet delay and VNF buffer state during a handover for different packet rates and packet sizes (based on data from [Küt17])

During this procedure, we collect the metrics mentioned before as follows: First, each of the packets is identified by a unique sequence number so that any lost, reordered, or duplicated packet can be easily identified. Second, the round-trip time (RTT) of the packets is measured at $Host_1$ to identify packet delays that are introduced by the execution of a handover. Third, we measure the buffer usage at the VNFs as well as at the SHarP controller during the entire experiment. Finally, the total handover duration, which is defined as the time taken between the initial handover request and the final migration of the flow to the destination VNF, is measured at the controller.

The first set of experiments focuses on a single handover procedure, with state transfer duration set to 0 seconds, to analyse what happens to the packets and VNFs during a flow migration. The experiments have been performed with 58 byte packets (Figure 4.4a) as well as with 1000 byte packets (Figure 4.4b) and each experiment is repeated 100 times. All error bars shown in this chapter indicate 95 % confidence intervals. The upper parts of Figure 4.4 show packet delays over experiment time with the handover happening at about 470 ms. The results show how the delay of the packets quickly increases to about 25 ms before they drop to their old level after about 50 ms. Except for smaller variations, this effect remains the same for different packet rates and packet sizes. The lower parts of Figure 4.4 show the number of packets stored in the destination VNF buffer during the handover. Depending on the packet rate, different numbers of packets need to be buffered. All of them are quickly released, once the handover is done. These results confirm that SHarP handovers show a stable performance under different conditions.

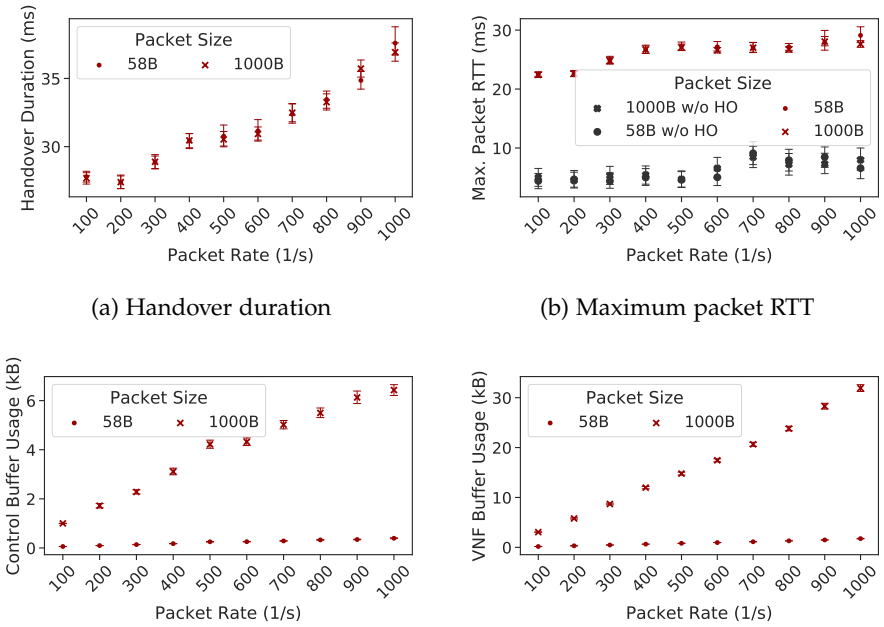
In the second set of experiments, we execute handovers with different packet rates, packet sizes, and state transfer durations. Every configuration is executed

100 times with a fully restarted network and controller setup to eliminate side effects from previous runs. The goal of these experiments is to analyse the general behaviour of SHarP under different conditions. The first set of results given in Figure 4.5 shows the handover performance as a function of the data rate of the moved flow given as PPS. The results shown in Figure 4.5 are based on measurements using a packet sizes of 58 bytes and 1000 bytes. As shown by the figures, the packet size has no impact on the handover duration or packet delays, but obviously accounts for more buffer usage if larger packets are used.

Figure 4.5a shows that the overall handover duration yields a linear increase with the changing packet rate, since more packets need to be processed. The maximum packet delay introduced by the handover procedure is shown in Figure 4.5b. It slightly increases with the packet rate and stays at around 27 ms at higher rates. This maximum delay is limited by the time the controller needs to notify the VNF about the handover and the VNFs to synchronise the state. If there is no state to be exchanged, the packet delay stagnates towards the end since the round-trip time between controller and VNF does not increase. Further, Figure 4.5b also compares the measured results against a baseline scenario in which no handover is performed (see: w/o HO). It shows that handovers increase the maximum packet delay by a constant value of 20 ms to 30 ms. The buffer usage of the controller and the VNFs is shown in Figure 4.5c and Figure 4.5d, respectively. As the packet rate increases, the entire system has to buffer more packets. This results in a linear increase in buffer usage at both the controller and the VNF. However, the controller buffer usage is lower by a factor of about five than the VNF buffer usage, contributing to the scalability of the system since the VNF buffer usage is distributed across the involved VNFs. During all experiments, no packets are lost, reordered, or duplicated, verifying the seamless nature of our handover mechanism.

The handover performance as a function of state transfer duration is shown in Figure 4.6. The increase in the state transfer duration is simulated by artificially introducing a delay after which the VNFs signal the completion of the state transfer. The experiments are executed with a fixed packet rate of 1000 PPS using packet sizes of 58 bytes and 1000 bytes. The state transfer duration is increased by 100 ms every step, ranging from 0 ms to 1000 ms. Figure 4.6a shows that the handover duration increases linearly with the additional time introduced by the state transfer, as expected. The maximum packet delay shown in Figure 4.6b is only offset by a small constant delay from the state transfer duration it experiences. This shows that the packets are indeed released from the buffers as soon as possible and that the packet delay is directly influenced by the state size and transfer duration. This is also highlighted by the comparison to a baseline scenario in which no handover is performed (see: w/o HO), as shown in the Figure 4.6b.

The most important results of our evaluation are given in Figure 4.6c and




(a) Handover duration

(b) Maximum packet RTT

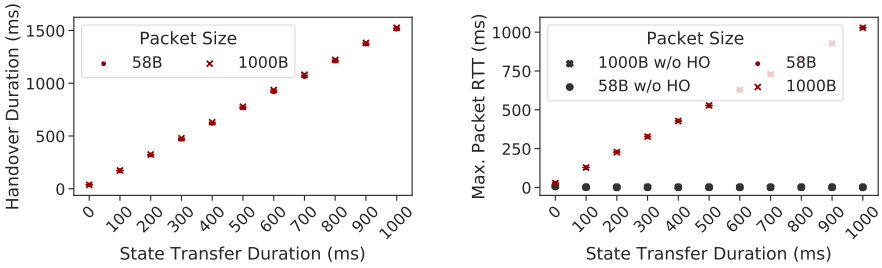
(c) Controller buffer usage

(d) VNF buffer usage

Figure 4.5.: Handover performance of SHarP dependent on UDP PPS with a packet sizes of 58 and 1000 bytes (based on data from [Küt17])

Figure 4.6d. They present the buffer usage at the controller as well as at the VNF and highlight the reduced controller load of SHarP. Even though the total amount of packets buffered in the system increases with the state transfer duration, the number of packets buffered at the controller remains constant. This produces a significantly lower workload for the controller compared to OpenNF, which is achieved by buffering the majority of the packets during the state transfer at the VNF, as the graph in Figure 4.6d attests. Again, no packet loss or reordering is detected during the experiments.

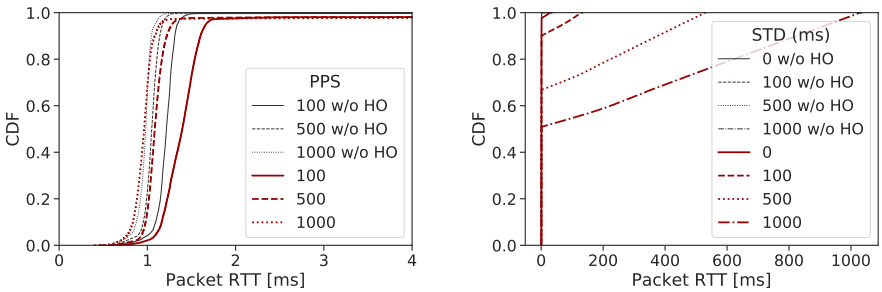
Further, Figure 4.7 shows the packet delay distributions of 58 byte packets sent during an experiment with either no handover (w/o HO) or a single handover. The left part of the figure shows that the packet rate has only minor impact on the delay and only few packets are delayed by the handover. The right part, in contrast, shows the impact of the state transfer duration to the delay experienced by the packets. It clearly shows that longer state transfer durations lead to a high number of delayed packets and thus is critical for the overall performance of elastic VNF deployments. In the right figure, the baseline (w/o HO) measurements are barely visible since they obviously do not experience the increased packet delays introduced by the state transfers.




(a) Handover duration (b) Maximum packet RTT

(c) Controller buffer usage (d) VNF buffer usage

Figure 4.6.: Handover performance of SHarP dependent on the state transfer duration with 1000 UDP PPS and packet sizes of 58 bytes and 1000 bytes (based on data from [Küt17])




(a) Packet RTT for different packet rates (b) Packet RTT for different state transfer durations

Figure 4.7.: Distribution of packet delays during different handover experiments using a packet size of 58 bytes (based on data from [Küt17])

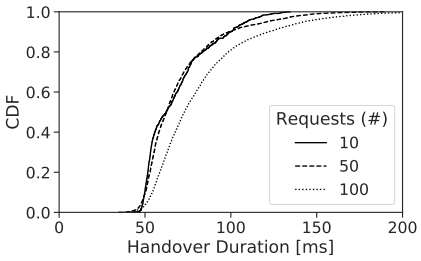
### 4.4.2. Multi-handover performance

The second part of our experimental evaluation focuses on how our SHarP prototype behaves in an environment in which an NFV orchestrator requests many handovers, e.g., because large parts of a service are reconfigured. We again use our previously described experiment setup and send bidirectional UDP traffic between $Host_1$ and $Host_2$. Each experiment is again executed 100 times. Instead of focusing on the handover of a single flow, multiple flows (up to 100) are now generated in parallel, each with a packet rate of 50 packets/s, and moved from $VNF_1$ to $VNF_2$ during the experiments. We generate handover requests using a Poisson arrival process with a rate of 2, 5, or 10 handover requests per second ($\lambda = 2, \lambda = 5, \lambda = 10$) and send them to SHarP's northbound interface. This simulates an environment in which the NFV orchestrator reconfigures the service multiple times per second, showing that SHarP is already designed for future, cloud-native NFV deployments in which reconfigurations may happen on a sub-second basis, which is usually not the case in today's VM-based deployments.

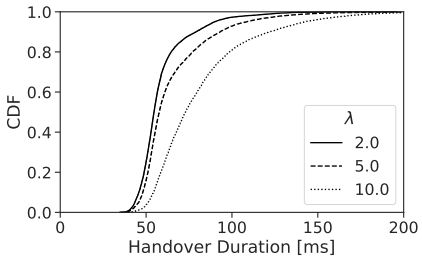
During the experiment, handover durations are measured and the total number of handover requests as well as their arrival rate is changed to evaluate the impact of the overall system load to individual handovers. Figure 4.8 shows the results of these experiments for different numbers of handover requests, request arrival rates, and flows with small (58 byte) and large (1000 byte) packets. Figures 4.8a and 4.8c show the behaviour of the handovers for an increasing number of performed handovers. They show that the handovers become slightly slower when more of them are executed. This effect is a bit stronger when larger packets are used (Figure 4.8c), which can be explained by the generally higher load in the system due to higher buffer usage at the controller. The handover request rate also impacts the handover duration as shown in Figures 4.8b and 4.8d. With a higher rate, the handovers become slightly slower. The size of the packets in the moved flows have almost no impact on this, which is an important property of SHarP because the handover performance does not depend on the nature of the moved traffic. In general, 90 % of the handovers in the experiment are completed in less than 120 ms. This shows that SHarP can deal with multiple handovers per second without substantial performance degradation.

## 4.5. Conclusion

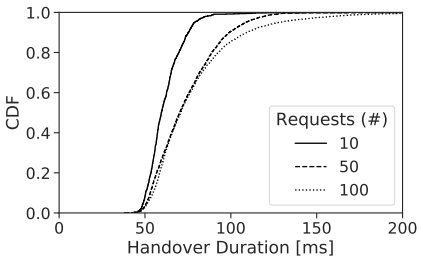
This chapter presents SHarP, a novel flow handover mechanism that provides loss-free and order-preserving flow migration for both unidirectional and bidirectional flows. We show how SHarP preserves the order of packets by using



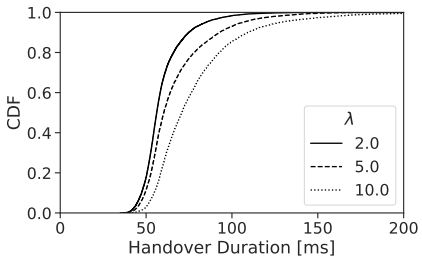

(a) Handover durations for 10 to 100 requests with fixed arrival rate ($\lambda = 10$) moving flows with 58 byte packets




(b) Handover durations for 100 requests and changing arrival rates ($\lambda$) moving flows with 58 byte packets




(c) Handover durations for 10 to 100 requests with fixed arrival rate ($\lambda = 10$) moving flows with 1000 byte packets




(d) Handover durations for 100 requests and changing arrival rates ($\lambda$) moving flows with 1000 byte packets

Figure 4.8.: Distribution of handover durations for multiple handovers using different numbers of handover requests, request arrival rates, and flows with small and large packets (based on data from [Küt17])

multiple FIFO buffers and subsequently releasing the packets. In contrast to existing approaches, SHarP does not come with an integrated state management solution but provides the means to support any state management solution implemented by a given VNF by sending triggers to it whenever flows are migrated. We believe that this is a much more practical separation of concerns since it leaves the choice of the used state management mechanism to the VNF vendors. It also reduces the required effort to integrate exiting VNFs into the system because no changes in the VNF's code are needed and the integration effort is limited to the installation of additional software components in order to enable support for flow handovers.

Our experimental evaluation clearly shows that SHarP reduces the maximum packet delay that constitutes the service interruption time during a handover, as it mostly depends on the initial time required to signal the VNF plus the state transfer duration. The interruption time only increases slightly with an increased packet rate and does not worsen at higher packet rates. Compared to a baseline scenario, the maximum packet delays are increased by 20 ms to 30 ms. The evaluation of the controller buffer at increasing packet rates and state transfer durations shows that with SHarP, the controller's buffer usage, and thus the amount of processed packets, only depends on the round-trip time between controller and VNFs and on the packet rate. It does not depend on the time taken for the state transfer that is usually hard to predict and heavily depends on the VNF implementation. This gives SHarP a major advantage over similar handover approaches. Our results also show that SHarP is ready for future, cloud-native NFV deployments with sub-second reconfiguration times.

# Part II.

# Rapid prototyping

# 5. Rapid prototyping of NFV functions and services

Once a VNF developer has finalised the implementation of a VNF, e.g., by using the concepts and solutions presented in the first part of this thesis, the VNF must be tested and integrated with other VNFs to create the final NS. Those NSs are then deployed across multiple sites, so called multi-PoP environments. This allows to improve service performance by optimising its placement in the network. But prototyping and testing of these complex, distributed scenarios becomes very challenging. The reason is that not only the NS as such has to be tested but also its integration with MANO systems. Existing solutions, like simulators, basic network emulators, or local cloud testbeds, do not support all aspects of these tasks.

To this end, I present a rapid prototyping platform for VNFs, NSs, and SFCs that is able to execute production-ready VNFs, provided as containers, in an emulated multi-PoP environment. These VNFs can be controlled by any third-party MANO system that connects to our platform through standard interfaces. Based on this, a developer can use our platform to prototype and test VNFs as well as complex NSs in a realistic environment running on a single machine, e.g., a developer's laptop.

This chapter is based on my paper [PKV16] and contains figures as well as verbatim copies of the paper's text. All presented concepts have been developed by me. The co-authors of the paper contributed small parts of the prototype implementation, e.g., additional monitoring features, which are not part of this chapter. After presenting related work in Section 5.2, Section 5.3 and Section 5.4 introduce the general concepts behind the presented prototyping platform and evaluate its network emulation features. After that, a concept to simulate resource limits of multiple PoPs, which are emulated by the platform, is presented and tested in Section 5.5. The presented platform is based on two sub-projects, "Containernet" and "vim-emu" which are both available as open-source projects [Peu16; ETS17b] and are actively used by other researchers, e.g., at the time of writing, Containernet has more than 120 stars and is forked more than 70 times on GitHub. As a result, various collaborators submitted bugfixes and code contributions which have all been reviewed and approved by me, e.g., more than 30 pull requests have been created by external users and merged by me.

## 5.1. Introduction

Developing and implementing VNFs and NSs that are deployed across distributed multi-PoP infrastructures is a complex software development process that consists of two main parts. First, the development of the VNFs and the composition of the resulting NS as such. Second, the integration of the NS with a MANO system that manages the service during its lifecycle. The second part involves the implementation and test of management interfaces but also the design, implementation, and validation of service-specific management components, like auto-scaling rules or placement strategies [Kar+16]. This complicates the overall development process. To reduce this complexity, extended tool support is required to reduce time-to-market, to save costs, and to improve the quality of service.

A special problem in this process is the lack of tools to support local prototyping or testing of complete NSs in end-to-end multi-PoP scenarios. In practice, this means that developers need to set up local NFV testbeds on which the developed VNFs and NSs are deployed and tested. Setting up those testbeds and maintaining them is, however, a complex task that requires many resources, especially for multiple PoPs, and a lot of expert knowledge, e.g., to configure a fully functional OpenStack installation. Once the VNFs and NSs have been tested on those testbeds, the developer has to export the VM or container images to ship them—a manual process that does not play well with agile environments and slows down turnaround times. This clearly motivates the need for more advanced prototyping solutions like we present in this chapter. Such prototyping solutions should not only allow testing of the involved VNFs, composed NS, as well as the SFC that interconnects the different components, e.g., by sending generated traffic through it. They also have to validate the NS's interactions with a MANO system, e.g., dynamic reconfiguration or placement strategies. This is not possible with existing approaches which either rely on local cloud testbeds that lack multi-PoP support, simulations that only execute simplified versions of network functions, or network emulation tools that do not offer interfaces to interact with MANO systems.

There are several simple and complex use cases for such a development support tool which motivate our proposed solution. These use cases can be divided into two categories. First, use cases that check the functionality of the NS as such (*NF-UC*); this can already be done with existing network emulation solutions, but requires considerable manual effort, e.g, for first migrating the VNF implementations to the emulation platform before porting them back to the production environments. Second, use cases that check the interoperation between NS and a MANO system (*MANO-UC*), which can today only be done with complex cloud testbeds or with public testbeds, such as [Fed18; Sof17], which are not available to every developer. Examples for both use case categories are listed in the following:

**NF-UC1 (Single VNF):** An NS developer wants to deploy and test single VNFs in a local test environment, e.g., by sending some generated traffic through them. A prototyping platform should be able to execute the VNF's code as it is without requiring any adaption. For example, the VNFs should be executed using the same virtualisation technologies as the production environments. During prototyping, a developer wants to interact with the running VNFs to, e.g., change configurations or monitor their behaviour. We address this use case in Section 5.3 and Section 5.4.

**NF-UC2 (Complex NSs):** A developer wants to test entire complex NSs consisting of several VNFs. Such NSs are composed using different description approaches, as described in Section 2.2.3. A prototyping platform should support those or at least offer the means to add support for new description approaches when they are needed. In general, a local test environment should be able to execute such complex NSs so that end-to-end tests, e.g., sending traffic through the service's chain, can be performed. While doing so, a developer needs to be able to interact with each VNF, like in *NF-UC1*. Section 5.4 explains how our proposed solution can support this use case.

**MANO-UC1 (Service management):** NSs are usually deployed under the control of a MANO system. The behaviour of such a deployment needs to be validated to ensure that the NS and MANO integrate correctly. To do so, a test environment needs to be able to interface with existing MANO systems through standardised interfaces. The important point in such a setup is that the used MANO system does *not* notice any difference between the prototyping environments and production environments to be able to realistically test its management operations. We discuss this in more detail in Section 5.4.4.

**MANO-UC2 (Multi-PoP support):** Another important use case is executing NSs in realistic multi-PoP environments to test both the behaviour of a service when its functions are distributed across multiple PoPs and service management approaches, like placement optimisation algorithms that decide which VNF is placed in which PoP. To this end, a multi-PoP prototyping solution needs to offer the means to describe and emulate arbitrary multi-PoP topologies, including realistic inter-PoP networking conditions. Section 5.4 provides the details on how multi-PoP scenarios can be emulated with our proposed solution and Section 5.5 investigates how resource limitations can be used to model multi-PoP scenarios more accurately. Further related work on this topic has been published by us in [SPK18] but is out of scope of this thesis.

To cover the previously described use cases and overcome the shortcomings of existing development support tools (Section 5.2), we introduce vim-emu (VIM

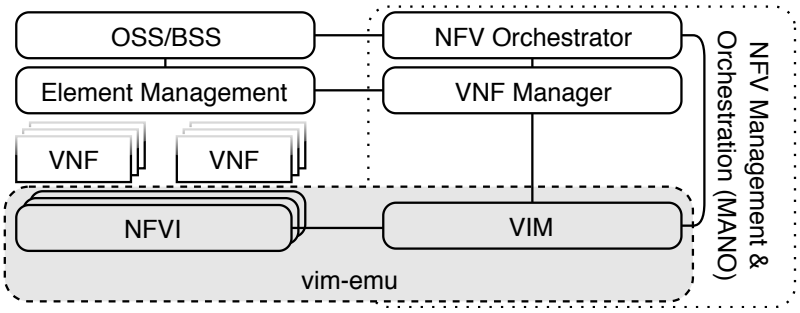



Figure 5.1.: The vim-emu platform in the (simplified) ETSI NFV reference architecture [ETS14b].

Emulator), a novel prototyping platform for NSs. The vim-emu platform was originally published under the name MeDICINE (multi Datacenter servIce ChaIN Emulator) in [PKV16] but renamed later when it became part of the OSM [ETS16c] project. It is able to execute production-ready network functions in realistic multi-PoP environments and allows standard MANO systems to control the deployment, like in a real-world system. Figure 5.1 shows the scope of vim-emu and its mapping to the ETSI NFV reference architecture in which it emulates multiple NFVIs and VIMs.

## 5.2. Related work

NFV development support is still a novel research direction with a limited amount of existing solutions. The small number of existing solutions are often organised as so-called NFV software development kits (SDKs) and mainly focus on manual descriptor creation or generation of base descriptors that can then be changed by a user. They also offer tools to perform syntactical and semantical checks among those descriptors [Van+18; Van+17]. These tools help to identify bugs and errors in the static descriptors, like a missing link in an SFC definition. But they do not offer support for developers when VNFs and their contained software components should be integrated and configured. Other prototyping tools focus on SDN debugging rather than on prototyping of complex NSs [Pel+15]. Some approaches are based on simulations to test and validate management solutions, e.g., placement algorithms, but they only provide very limited realism since the simulated network functions are only proxies and not real implementations used in production [Cal+11; Zha+12; Hen+08]. The very lightweight service platform (VLSP) project [MCG15] offers more realism but the tested VNFs are still limited to simple Java programs and not full-featured, realistic VNF implementations.

Emulation tools, like Mininet, are able to execute arbitrary software, e.g., VNF implementations, in isolated network namespaces [LHM10; Wet+14; Ahr10]. However, moving these VNF implementations from Mininet into a

production environments, i.e., packaging them as containers or VMs, is still a time-consuming and error-prone task. These tools also lack the possibility to emulate distributed PoPs or cloud sites, e.g., they have no functionality to stop and remove hosts at runtime. The ESCAPE platform overcomes some of these limitations by combining a MANO system with different test VIMs (including Mininet and OpenStack) but it does not target development support or prototyping tasks. Its main focus is on orchestration between non-emulated PoPs [Son+15].

Another related solution is called NIEP (NFV infrastructure emulation platform) and pursues similar goals as vim-emu does. It also utilises Mininet to emulate arbitrary network topologies. In those topologies, it runs minimised VMs based on ClickOS [Mar+14], which can be used to implement arbitrary VNF functionalities using the Click modular router [Koh+00] framework. As a result, it offers an interesting solution to quickly prototype and test new VNF designs. However, the use of Click limits the platform to Click-only VNFs, which makes it not suitable to test arbitrary, production-ready VNFs. Further, NIEP does not offer interfaces to integrate with real-world orchestration solutions, which is a shortcoming compared to vim-emu. NIEP was released in 2018 and thus two years after vim-emu.

An alternative to emulation platforms are real cloud testbeds, which might be installed on a single physical machine, but are typically not able to run services in arbitrary network topologies [Ope10a]. And even if they do, they come with considerable management overhead and only a limited number of PoPs that can be used for tests [KRP13].

Table 5.1 compares the relevant features of the existing simulation, emulation, and testbed solutions. It shows that all existing approaches lack some of the features. For example, it shows that only vim-emu supports the combination of production-ready VNFs and multi-PoP scenarios.

After all, Table 5.1 shows that vim-emu is the most sophisticated solution for NFV prototyping in multi-PoP scenarios available at the moment. As a result, vim-emu is reused and extended by other researches, for example, in a project called Fogbed by Coutinho et al. [Cou+18]. Fogbed was released in 2018 and is a fork of vim-emu that modifies it to provide specialised interfaces and APIs to prototype fog computing [Bon+12] scenarios. Further, I contributed to a collaborative work that extends our Containernet platform to support prototyping of P4-based offloading solutions [Bos+14] and is called FOP4 [Mor+19b; Mor+19a].

Table 5.1.: Feature matrix of existing prototyping approaches for NFV

| Feature | Simulations | VLSP [MCG15] | Mininet [LHM10] | ESCAPE [Son+15] | NIEP [Tav+18] | DevStack [Ope10a] | Cloud testbeds [KRP13] | vim-emu [PKV16] |
|---|---|---|---|---|---|---|---|---|
| production ready VNFs | ○ | ○ | ◐ | ● | ◐ | ● | ● | ● |
| arbitrary topologies | ● | ● | ● | ◐ | ● | ○ | ○ | ● |
| multi PoP support | ● | ● | ◐ | ◐ | ◐ | ○ | ◐ | ● |
| PoP resource modelling | ● | ◐ | ◐ | ○ | ○ | ○ | ○ | ● |
| realistic VNF performance | ○ | ○ | ◐ | ● | ● | ◐ | ● | ◐ |
| support for hybrid VNF chains | ◐ | ○ | ○ | ◐ | ○ | ○ | ○ | ● |
| generic chaining support | ◐ | ◐ | ○ | ● | ● | ○ | ◐ | ● |
| NSH-based chaining | ○ | ○ | ○ | ○ | ○ | ● | ◐ | ● |
| MANO system integration | ◐ | ◐ | ○ | ● | ○ | ● | ● | ● |
| run offline/local | ● | ● | ● | ◐ | ● | ◐ | ○ | ● |
| test/prototyping support | ● | ● | ● | ○ | ● | ◐ | ◐ | ● |

● fully supported / ◐ partly supported / ○ not supported

## 5.3. Container-based network emulations

We base the design and implementation of vim-emu on a tool called Containernet [Peu16], which was also developed by us as preliminary work for vim-emu. The main idea behind Containernet is to extend the Mininet network emulation platform so that arbitrary containers (i.e. Docker) can be connected to the emulated network. The benefit of this is that container images with preinstalled and configured software can directly be used within the emulation experiments. For example, the default Apache httpd container can be pulled and used to integrate a simple web server into an emulation experiment—a much simpler process than installing and configuring the server manually in a classical Mininet topology. Compared to Mininet, Containernet already behaves much more like a typical NFV environment in which VNFs are shipped as container or VM images.

In addition, Containernet allows adding and removing containers from the emulated network at runtime, which is not possible in Mininet. This concept allows us to use the environments emulated by Containernet like cloud infrastructure in which we can start and stop compute instances (in the form

```
1  from mininet.net import Containernet
2  # create Containernet network object
3  net = Containernet()
4  # add two Docker containers (Ubuntu and Httpd)
5  d1 = net.addDocker("d1", dimage="ubuntu:trusty")
6  d2 = net.addDocker("d2", dimage="httpd:latest")
7  # add a switch
8  s1 = net.addSwitch("s1")
9  # interconnect containers and switch
10 net.addLink(s1, d1)
11 net.addLink(s1, d2)
```

Listing 5.1: Example of Containernet's Python API

of containers) at any point in time. Another feature of Containernet is that it allows to change resource limitations, e.g., CPU time available for a single container, at runtime and not only once when a container is started, like in legacy container setups. To define the network topologies to be emulated, Containernet offers an extended but fully backward-compatible version of Mininet's Python API. We use this design to lower the entry barrier for researchers that are already familiar with the Mininet APIs. Most of our extensions are hidden behind a single method, called `addDocker(name, dimage)`, which can be used to add a Docker container to a Mininet network. Listing 5.1 gives a brief example of this and shows how two Docker containers are added to the network and are interconnected with a switch.

The two mandatory arguments for the `addDocker` method are a name to identify the container and the identifier of the container image to be used. Besides this, the method offers a large number of optional arguments that allow to transparently pass and forward configuration options to the Docker containers, like environment variables, volumes, etc. [Peu16].

The main challenge to solve when integrating Docker with Mininet is the fact that the default Docker APIs do not offer any built-in solutions to add multiple network interfaces to a Docker container. This means, each Docker container, created with the default Docker API, has always only a single network interface. Having multiple network interfaces in a single container is, however, one of the must-have features to connect containers to arbitrary topologies. To solve this, Containernet uses a trick: Instead of only relying on the single network interface created inside a container by the Docker daemon, it creates a virtual ethernet device pair (veth) [Lin18b] outside the container whenever the `addLink(d1, s1)` method is called. One interface of the pair is then connected to the switch (`s1`), just like in a normal Mininet, and the other interface is moved into the network namespace of the running Docker container (`d1`) using the `ip link set intf1 netns d1.pid` command. Using this, users can add an arbitrary number of network interfaces to a Docker container running

within Containernet. The use of veth pairs is a good fit for the described problem, since they behave like virtual patch cables and simply forward all traffic between two virtual network interfaces. This also fits the idea that a virtual link in a Mininet topology basically emulates a physical network cable. Moving one end of the veth pair into the container, to connect it to the topology, is possible since Mininet, as well as Containernet, are executed with `root` privileges, which are also needed to create and manage the virtual switches. Further it allows to create and remove links at runtime, something which is not possible using Docker's default APIs. Having this, Containernet becomes a great starting point to build a multi-PoP NFV prototyping platform.

The limitation of this design is the fact that some VNFs might not be implementable using lightweight container technology, e.g., because they rely on customised kernel functionalities. In that case, the use of VMs cannot be avoided. To provide a solution for this, we presented an extension of Containernet that allows to add full-featured VMs to the emulated network topologies [PKK18a; Kam17]. However, this is an entirely optional extension to the presented platform which I will not further discuss in this thesis, since more and more VNFs follow a cloud-native design and can thus run in container-based environments [5GP18]. Still, adding additional container technologies to Containernet offers interesting opportunities for future work (Section 11.3).

## 5.4. Emulating multi-PoP NFV scenarios

Even though Containernet allows to specify and emulate arbitrary network topologies with connected containers that execute arbitrary software, e.g., VNF implementations (use case NF-UC1 defined in Section 5.1), it is not able to emulate NFV multi-PoP scenarios or automatically deploy complex NSs (use case NF-UC2, MANO-UC1, and MANO-UC2). To amend this shortcoming, we introduce vim-emu as an additional layer running on top of Containernet. This layer offers the required APIs, endpoints, and abstractions to emulate realistic NFV multi-PoP scenarios. The resulting emulated environment can then be used as prototyping platform for VNFs, NSs, or novel MANO concepts.

To do so, vim-emu provides the following key features: First, it exploits Containernet's topology API to define arbitrarily complex multi-PoP environments with realistic link properties, like delay, data rate limitation, and loss rate. Second, it uses standard containers (i.e. Docker) to execute arbitrary VNFs, allowing a developer to directly deploy the prototyped container images to production after they have been tested locally. Third, it provides cloud-like interface endpoints, e.g., an OpenStack Nova-like [Ope10c] interface, to control each emulated PoP in the platform. This allows developers to connect their local prototyping environment to existing MANO solutions. The following

sections describe the general workflow of vim-emu, give insights about its abstract topology definition as well as PoP control endpoint APIs, and discusses its support for SFC.

### 5.4.1. Workflow

The general concept of vim-emu and the workflow of a developer who uses it to prototype an NFV scenario is shown in Figure 5.2. The depicted workflow focuses on a service-centric prototyping scenario in which a developer prototypes a new NS. Other scenarios in which a developer tests, for example, a novel MANO system against a multi-PoP environment are also possible. The shown scenario assumes that the developer has already installed a MANO system of his choice which is responsible to orchestrate the service on the emulated infrastructure.

The developer's workflow can then be split into the following steps: First, the developer defines an NS, consisting of function (VNFD) and service (NSD) descriptors as well as pre-built container images or Dockerfiles that contain the network functions to be tested. The actual format of this NS and its descriptors depends on the used MANO system that deploys the NS on top of our platform. Then, the developer defines a multi-PoP topology on which the service should be tested (step 1) and starts the vim-emu platform with this topology definition (step 2). After the platform has been started, the developer connects the MANO system of his choice to the emulated PoPs by using a flexible endpoint API (step 3) described in Section 5.4.4. After this step, the MANO system has full control over the emulated multi-PoP infrastructure and the NS can be deployed on vim-emu. To do so, it needs to be on-boarded to the MANO system (step 4) which deploys each VNF as a container in one of the emulated PoPs, connects the container to the emulated network topology, and sets up the service chain. In this stage, the service is deployed and runs inside the emulated multi-PoP environment (step 5).

To easily test and configure this running prototype of the NS, the developer can directly interact with each running container through Containernet's interactive command line interface (CLI), e.g., to view log files, change configurations, or run arbitrary commands. This makes the interaction with the involved VNFs much easier, since neither remote connections nor preinstalled security keys, i.e., for SSH connections, are needed, as it would be the case in a normal cloud testbed. The NS and its VNFs can then be stimulated with generated traffic, e.g., using tools like `iperf` or `tcpreplay`. Furthermore, a developer and the MANO system can access arbitrary monitoring data generated by the containers connected to vim-emu or the VNFs themselves. Once the developer has verified the correct behaviour and configuration of the NS, it can be terminated and moved to a production environment using the same container

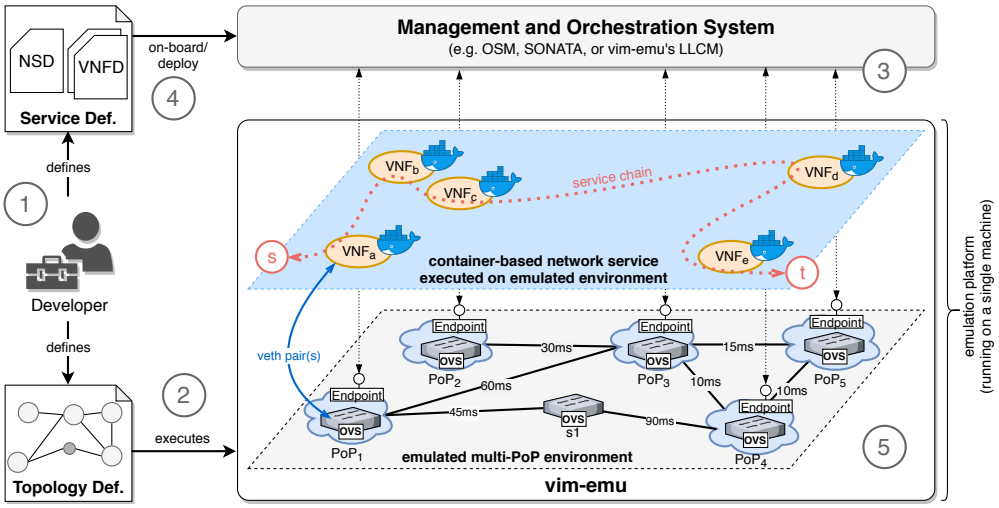



Figure 5.2.: General idea and workflow of the system. The example shows a running emulation environment with five emulated PoPs, five allocated compute instances executing VNFs, and a service chain setup chaining those VNFs through which generated traffic is sent from node *s* to node *t* (logos from [Doc13]).

images and descriptors. If, in contrast, a problem is detected, the developer can quickly change one or more VNFs and re-start the platform to have a fresh environment for the next round of testing.

### 5.4.2. System architecture

The system design of vim-emu is highly customisable. It offers plugin interfaces for most of its components, like API endpoints, container resource limitation models, or topology definitions.

Figure 5.3 shows the main components of vim-emu and how they interact with each other and with the underlying Containernet. The *emulator core* component implements the main emulation environment, e.g., the emulated PoPs. It is the core of the system and interacts with the *topology API* to load topology definitions as described in Section 5.4.3. The flexible *endpoint API* allows vim-emu to be extended with different control interfaces that can be used, e.g., by MANO systems to manage and orchestrate NSs in the emulated environment (Section 5.4.4). The *resource management API* allows to connect resource emulation models that define how much resources, like CPU time and memory, are available in each of the emulated PoPs, as further described in Section 5.5. Finally, we provide an easy-to-use representational state transfer (REST) control API and a CLI [Peu17] that allows developers to interact with vim-emu and manage the lifecycle of single VNFs and complete NSs.

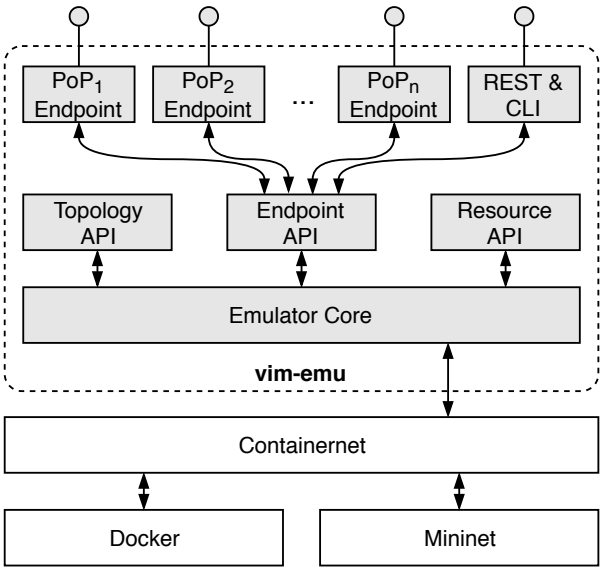



Figure 5.3.: System architecture and components with $N$ active PoP endpoints offering control interfaces to an external MANO system

### 5.4.3. Topology definition

To test NSs in realistic multi-PoP scenarios, vim-emu needs to be able to emulate arbitrary network topologies. To do so, it offers a topology definition API used to define available PoPs, their resources, as well as the network between them and its properties. In contrast to legacy Containernet or Mininet topologies, vim-emu topologies do not describe single network hosts connected to the emulated network. Instead, they define available PoPs, represented by emulated cloud data centres in which compute instances can be started and stopped at any point in time. In the most simplified case, such a PoP emulates just a single node, i.e., a router with attached compute and storage facilities like a Blade server. A more sophisticated node represents a small data centre, which comprises several servers and is internally connected by a single SDN switch. More abstractly, an emulated PoP can also be a complex data centre whose internal connection is simplified into a "big-switch" abstraction (as shown in Figure 5.2). For all these versions, we assume that the MANO system has full control over whether a particular VNF is executed at a particular PoP but does not care about internals of the PoPs.

A vim-emu topology allows to add an arbitrary number of SDN switches between PoPs (Figure 5.2, *s1* and *s2*). These SDN switches as well as any SDN switches within each PoP can be either controlled by standard SDN controllers, by custom controller implementations provided by the user of the platform, or by the MANO system itself. Thus, complex network and forwarding setups with a high number of inter-PoP switches can be emulated.

```
1  # create basic vim-emu emulation
2  net = DCNetwork()
3  # create two PoPs
4  p1 = net.addPoP("My PoP 1")
5  p2 = net.addPoP("My PoP 2")
6  # create an intermediate SDN switch
7  s1 = net.addSwitch("s1")
8  # connect PoPs: p1 <-> s1 <-> p2
9  net.addLink(p1, s1, delay="10ms", bw=10)
10 net.addLink(p2, s1, delay="50ms", loss=2)
11 # attach cloud endpoints to control each PoP
12 api1 = OpenstackApiEndpoint(port=6001)
13 api1.connectPoP(p1)
14 api1.start()
15 api2 = OpenstackApiEndpoint(port=6002)
16 api2.connectPoP(p2)
17 api2.start()
18 # start the emulation
19 net.start()
20 # start and enter the interactive CLI
21 net.CLI()
```

Listing 5.2: Example vim-emu topology with two PoPs connected to OpenStack-like cloud endpoints

We based our topology API on Containernet's Python API. This has the benefit that developers can use scripts to define or algorithmically generate topologies. It also allows to implement generators that can read given topologies, e.g., based on GraphML files, and turn them into vim-emu multi-PoP topologies. We implemented a prototype of such a generator that transforms topologies provided by the Internet Topology Zoo [Kni+11] into executable emulation scripts [Peu+18b].

Listing 5.2 shows an example topology script defining two PoPs that are interconnected by a single switch. It shows how the PoPs are connected and how the link setup is done (lines 1–10). It also shows how link properties, like delay, loss, and data rate can be set. At the end of the script, the emulation is started (line 19) and vim-emu's interactive CLI is opened (line 21) which allows a user to directly interact with the emulated scenario, e.g., list running PoPs, VNFs, and their interconnections.

### 5.4.4. Flexible endpoint API

After an emulation topology is defined, MANO systems need a way to control the emulated PoPs, e.g., to start and stop compute instances inside them. To realise this, we introduce the concept of *flexible API endpoints* (Figure 5.3). Such an API endpoint is an interface to a single PoP which provides typical IaaS

cloud control interface semantics to manage compute instances or networking. Instead of fixing our design to a single interface implementation, we provide an abstract API and allow users of the platform to implement their own endpoints on top of it. This has the benefit that our platform can be integrated with any MANO system as long as an API endpoint that provides the expected interfaces is created. Examples for such endpoints are OpenStack-compatible interfaces [Ope10b], which are among the most commonly used VIM interfaces in the NFV landscape. But our design allows any other open or proprietary interface to which a MANO system can connect as well. We implemented and presented prototypes of OpenStack-compatible interfaces in [Peu+17] and present more details about this implementation in Chapter 7.

Besides interfaces that act as a bridge between MANO systems and emulated PoPs, more intelligent control components can be implemented on top of the provided endpoint API. An example for this is the "SONATA-NFV Dummy-Gatekeeper" [SON16] which is a simplistic orchestration solution that allows to directly deploy SONATA-NFV compliant NS packages [SON16] on vim-emu without the need of an external MANO solution. This component later evolved to the "5GTANGO lightweight lifecycle manager (LLCM)" [5GT18b], which adds additional support for NS packages compatible with 5GTANGO's advanced NFV package format [5GT18a]. Even though those lightweight control components offer a very restrictive set of features compared to real-world MANO solutions, e.g., no day-0 and day-N configuration mechanisms, no monitoring etc., they still turn out to be useful for quick prototyping of VNFs and NSs. An example for this is 5GTANGO's smart manufacturing pilot, which was entirely developed on top of vim-emu before it was migrated to its production environment [Sch+19; Peu+19d].

Technically, API endpoints are instantiated and assigned to PoPs using topology definition scripts as shown in Listing 5.2 lines 12–17. The default approach is adding one endpoint to each PoP so that each emulated PoP provides its own management interface towards the MANO system (Figure5.3). From the perspective of the MANO system, this looks exactly like a real multi-PoP environment offering a heterogenous set of management interfaces towards the available PoPs.

### 5.4.5. Chain management and forwarding paths

To run complex NSs on top of vim-emu, we need to deploy its VNFs and set up the forwarding path between them, like the SFC shown in Figure 5.2. Since all emulated PoPs are based on SDN switches, a vim-emu user can have full control over the forwarding of an NS's network traffic. Setting up a chain where traffic is steered along a defined path is now a matter of setting the correct forwarding entries in the involved SDN switches using an SDN controller. To

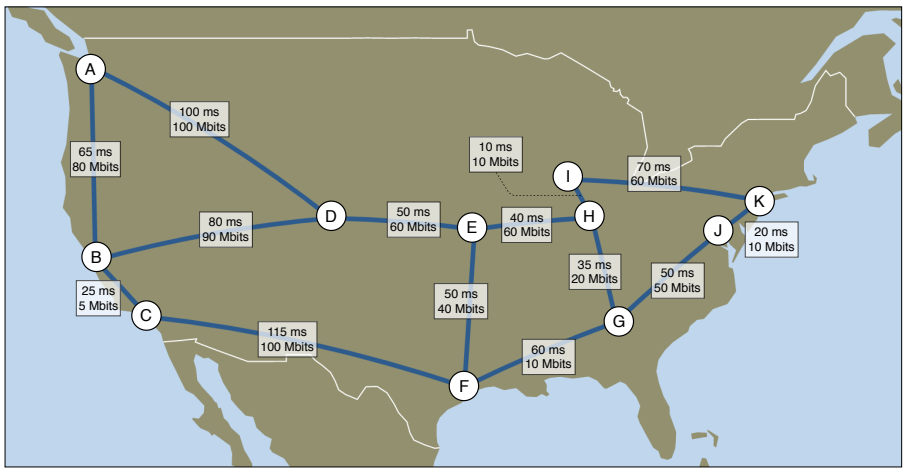


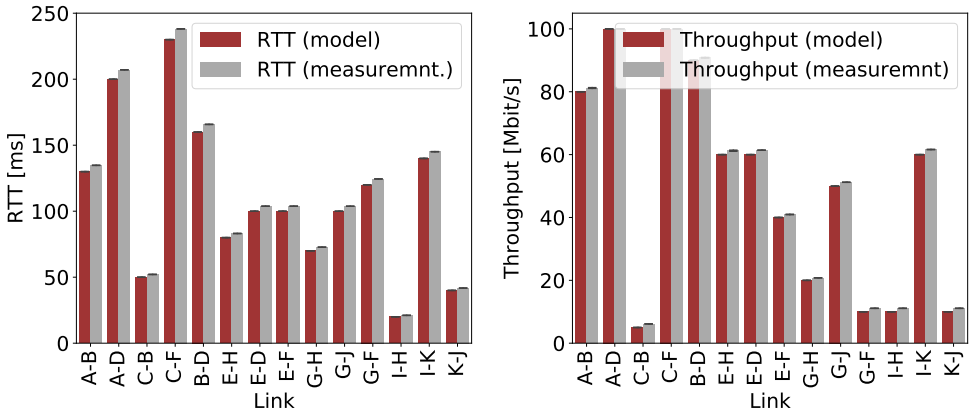

Figure 5.4.: Topology used for multi-PoP evaluation (based on "Abilene" topology [Kni+11])

support users to setup arbitrary forwarding path and thus chain the VNFs, we provide a simplified API that brings basic SFC functionality to vim-emu while hiding the complexity of low-level SDN protocols. This API allows to chain running containers, i.e., VNF instances, by calling a single Python method, i.e., `setChain(vnf1, ... , vnfN)` that gets a list of VNF instance objects as inputs. An internal graph representation of the topology and its attached containers is kept to compute the forwarding path with the fewest hops or the smallest delay, depending on the used vim-emu configuration. This basic SFC implementation uses VLANs to isolate different forwarding graphs and to steer traffic to the correct VNFs. Even though this approach works fine for many prototyping use cases, a more sophisticated SFC solution, relying on NSH [QEP18], has been added to vim-emu as further detailed in Chapter 6.

### 5.4.6. Evaluation

To evaluate the multi-PoP features of vim-emu and in particular the correct emulation of networking properties between the PoPs, we perform a set of experiments. We execute these experiments using a topology called "Abilene", which is part of the Internet topology zoo (ITZ) [Kni+11] and is shown in Figure 5.4. The topology has 11 PoPs (A–K) and 14 links between them. We assign the shown rate limits and delays to each link and let vim-emu run this topology. We then measure the RTT as well as the achieved throughput on each of the links and compare the measured results with the values modelled by the emulated topology. The goal is to confirm that the connections between the emulated PoPs behave as intended when real traffic is transmitted between them.

We execute these experiments on a single physical machine with an Intel(R)




(a) RTT (ICMP)

(b) Throughput (TCP)

Figure 5.5.: Modelled vs. measured RTT and throughput between the PoPs of the emulated "Abeline" topology (Figure 5.4)

Xeon(R) W-2145 CPU at 3.70 GHz CPU, 32 GB of memory, running Linux 4.4.0-142-generic. To perform the measurements, a single VNF, represented by an Ubuntu 16.04 Docker container with `ping` as well as `iperf3` installed, is deployed in each of the PoPs. The measurements are then executed between these test VNFs. The RTT is measured using `ping` with its default options and the achievable TCP throughput is measured with `iperf3` also using the default options. Each measurement is repeated 30 times and the error bars show 95 % confidence intervals. Figure 5.5a presents the RTT results. It can be seen that all measurements are quite constant (error bars are almost not visible) and that measurements are close to the modelled RTTs used as inputs to the emulator. In general, the measured RTTs tend to be slightly higher than the modelled ones, which can be explained by the additional delay that is added by the intermediate Open vSwitch (OVS) switch in each PoP and the network stacks of the measurement VNFs that need to be traversed in addition to the link between the PoPs.

A similar picture can be seen in Figure 5.5b, which reports the TCP throughput achieved on each of the links. Again the measured values are close to the modelled values, which proofs that the network emulation features work as expected. It also shows that vim-emu is capable of emulating realistic network topologies with configurable link properties between the PoPs. These features can, for example, be used to test VNF placement algorithms in realistic scenarios, going beyond simple simulation approaches in which no real VNFs are executed, as we have shown in [SPK18].

## 5.5. Emulating PoP resource limits

Even though cloud systems provide virtually infinite compute resources to their customers, realistic scenarios, especially with small PoPs, look different. Such PoPs offer limited compute, memory, and storage resources which have to be considered by a MANO system when placement and scaling decisions are taken. Those limitations especially apply in fog and MEC scenarios [Bon+12; ETS14a]. To emulate such resource-limited PoPs, vim-emu offers the concept of *flexible resource models*. Such a model can be assigned to a PoP and simulates the resource behaviour of that PoP by restricting the resources assigned to the containers that are deployed in that PoP. As a result, the actual resources, e.g., CPU time or memory, available to a container and its contained VNF implementation are restricted. This looks, from the perspective of the VNF implementation running in the restricted container, like a real-world scenario in which the resources might be limited, e.g., because the available CPU time has to be shared with other VNFs. A resource model can also reject instantiation requests for new containers, if not enough simulated resources are free and available, as we show in the first example in Section 5.5.1.

Technically, resource models can be assigned to and configured for each PoP within the topology definition as shown in Listing 5.3 lines 5–10. These models do bookkeeping of the used resources and are called whenever resources should be allocated or released, e.g., a new container should be started or a running container should be stopped. When a new container should be started, the user specifies the amount of simulated resources that should be allocated to this container. This corresponds to requesting a specific resource configuration (also called flavour) in a real-world IaaS scenario in which a new compute instance should be instantiated in a PoP. Upon this request, the used model computes CPU, memory, and storage limits to be applied to the requested container, i.e., it restricts the resources available to the container in the emulation scenario. For example, a model can automatically reduce the available resources, say CPU time, of all containers running inside an emulated PoP when additional containers should be allocated in this PoP. This simulates an oversubscription scenario in which all containers of a PoP have to share a limited set of resources, as further detailed in the second example given in Section 5.5.1.

A generic API allows developers to easily create their own resource models. For example, a telco operator that deploys services in its own PoPs might have more control about available resources than a web service provider that buys cloud resources from a third party, like Amazon. The telco operator can enforce detailed configurations, like core pinning or even use a fixed dedicated physical node for a specific VNF. These options are often not offered in shared third-party cloud environments. Vim-emu can, however, be used as a prototyping platform in both cases. The telco operator might, e.g., use a resource model in

```
# create two PoPs
p1 = net.addPoP("My PoP 1")
p2 = net.addPoP("My PoP 2")
# initialize and assign resource models to PoPs
r1 = ModelA_FixedLimit(
    max_cu=24, max_mu=80, max_su=90)
r1.assignPoP(p1)
r2 = ModelB_Oversubscription(
    max_cu=80, max_mu=120, max_su=280)
r2.assignPoP(p2)
# ...
net.start()
```

Listing 5.3: Example vim-emu topology with two different resources models, each assigned to a particular PoP

which resources are strictly reserved whereas the web service provider uses a model in which the service's performance is influenced by other services that share the same infrastructure. Models for other operational metrics, like prizing models, can also be implemented, e.g., increase prizes for resources if a PoP is highly utilised.

### 5.5.1. Models

To showcase how a vim-emu resource model looks like, we provide two example CPU limitation models that are implemented in our prototype; memory and storage models that use the same ideas and concepts are available as well. We use models that limit the available CPU time of the containers rather than models that change the number of CPU cores, because it gives us much more flexibility in terms of possible configurations since vim-emu is usually executed on a single physical (or virtual) machine with a small number of available CPU cores, e.g., a developer's laptop.

The goal of the presented models is to limit the overall available CPU capacity of each PoP in a way such that the utilisation of one PoP does *not influence* other PoPs. This is an important feature to realistically emulate multi-PoP scenarios on a single (physical or virtual) machine on which vim-emu is executed. Our example models use the notion of compute units (CUs) to describe the relative amount of CPU time allocated to a single container. For example, a container that requests 4 CUs will get twice as much CPU time as a container requesting 2 CUs. These relative resource requirements can be described independently from absolute available resources. This results in emulated scenarios in which the relative resources available to each container (running a VNF) reflect the resources available to them in real-world deployments, e.g., $\text{VNF}_i$ that gets half the resources of $\text{VNF}_j$ in a real-world deployment, will get also half as

Table 5.2.: Definitions used to build our example CPU resource models

| Symbol | Description |
|---|---|
| $E_{\text{cpu}} \in (0, 1]$ | Percentage of physical CPU time available for the overall emulation. For example, all containers together will not use more than 75 % of the physical CPU if $E_{\text{cpu}} = 0.75$. |
| $N \in \mathbb{N}_{>0}$ | Number of PoPs in the emulated topology. |
| $\text{mc}_p \in \mathbb{N}_{>0}$ | Number of CUs available in PoP $p$. |
| $\text{ac}_p \in \mathbb{N}$ | Number of CUs already allocated by running containers in PoP $p$. |
| $\text{nc}_c \in \mathbb{N}_{>0}$ | Number of CUs requested for a new container $c$. |
| $P_c \in [0, 1]$ | Percentage of physical CPU time computed by the resource model and finally assigned to container $c$. |

much resources as $\text{VNF}_j$ in the emulation. More specifically, we define a CPU model using the symbols described in Table 5.2 and use this to define a CPU limitation function that gets the number of CUs requested for a container ($\text{nc}_c$) and the PoP on which the container should be deployed ($p$) as inputs and maps this to the absolute CPU time that will be assigned to the container: $f : \text{nc}_c \times p \rightarrow P_c$.

Figure 5.6 presents an abstract example of this model. It shows the total available CPU time ($[0, 1]$) of the host machine which executes the emulation. The global parameter $E_{\text{cpu}} = 0.75$, which has to be configured before vim-emu is started, defines that 75 % of the total CPU time can be used for emulating the PoPs, leaving the remaining CPU time to the host operating system and other tasks. In this example, an emulation scenario with two PoPs is shown. During initialisation of the scenario, the "size" of the two PoPs in terms of available resources is defined. PoP1 is configured to provide six CUs ($\text{mc}_1 = 6$) and PoP2 is configured to offer half of the resources of PoP1 ($\text{mc}_2 = 3$). This shows how a user can use the abstract notion of CUs to define the relative size of the PoPs involved in the emulation.

Figure 5.6 also shows how these PoP configurations are used to initially compute the absolute size of a single CU by dividing the CPU time available for the emulation by the total number of configured CUs as shown in Equation 5.1. There is no VNF deployed in this example and as a result none of the available CUs is allocated.

$$\frac{E_{cpu}}{\sum_{i=1}^{N} \text{mc}_i} = \frac{0.75}{6 + 3} = 0.0833 \tag{5.1}$$

We use this abstract model to introduce two example resource models with

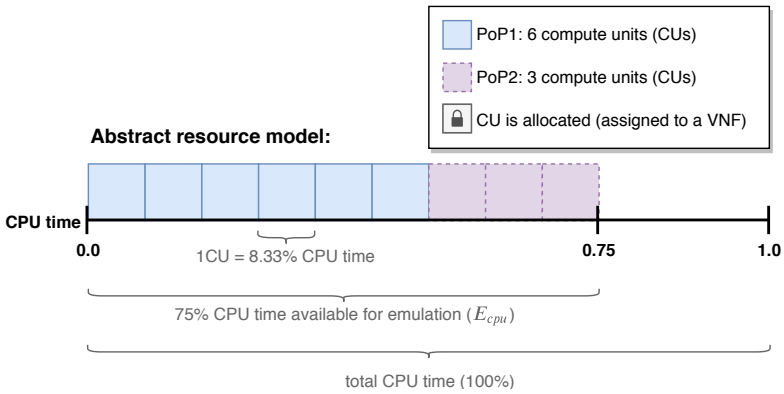



Figure 5.6.: Abstract example of the presented resource models showing two PoPs with different sizes (PoP1=6 CUs, PoP2=3 CUs) and how these abstract CU are mapped to the available CPU time of the host machine on which the emulation is executed

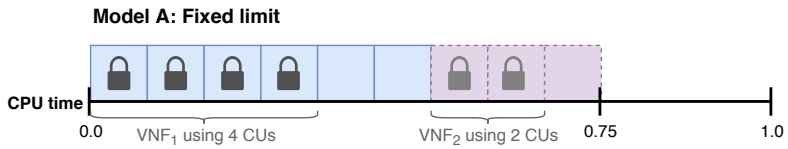
different complexity in the following sections. First, a model that simulates PoPs with a fixed amount of available resources that cannot be oversubscribed is presented. Second, a model is introduced that uses a fixed resource limit per PoP but allows oversubscription when more and more containers are deployed in a PoP. It is important to note that the abstract model shown in Figure 5.6 is an example to demonstrate how the resource model feature of vim-emu can be used. Vim-emu is not limited to this model and the used notations.

#### 5.5.1.1. Model A: Fixed limit

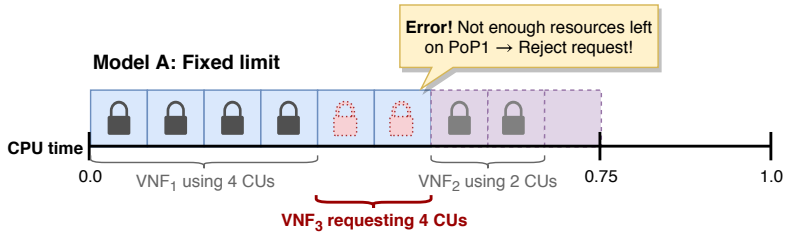
Our first model assigns a fixed amount of CUs to each PoP in the system and does not allow to oversubscribe a PoP. This is modelled by defining a mapping function ($f_p$) for each PoP ($p$) that maps requested CUs ($\text{nc}_c$) to absolute CPU time ($P_c$) as shown in Equation 5.2. If a new VNF container is requested and not enough free CUs are left in the target PoP, the instantiation request is rejected (function $f_p$ returns 0). If enough CUs are available, the absolute CPU time ($P_c$) for the container is computed based on the overall CPU time available for the emulation ($E_{\text{cpu}}$) and the CU limit assigned to the selected PoP ($\text{mc}_p$) as described in Equation 5.1. As a result a PoP can never be over-utilised. In this model, the amount of absolute CPU time per CU stays constant throughout the entire emulation.

$$f_p(\text{nc}_c) = \begin{cases} \frac{E_{cpu}}{\sum_{i=1}^{N} \text{mc}_i} \cdot \text{nc}_c, & \text{if } \text{ac}_p + \text{nc}_c \leq \text{mc}_p \\ 0 \text{ (reject)}, & \text{else} \end{cases} \tag{5.2}$$

**Example:** To better demonstrate the behaviour of this model, we present an example in Figure 5.7. The example is split into two steps. The first step,




(a) Step 1: $VNF_1$ (using 4 CUs) and $VNF_2$ (using 2 CUs) are deployed and running on PoP1 and PoP2




(b) Step 2: Instantiation request of $VNF_3$ (with 4 CUs) sent to PoP1 is rejected, because PoP1 has only two free CUs left

Figure 5.7.: Example scenario using Model A: Fixed limit

shown in Figure 5.7a, depicts a situation in which two VNFs are deployed in the system. More specifically $VNF_1$ is deployed in PoP1 and allocates the four CUs it has requested when it was instantiated. Similarly, $VNF_2$ runs in PoP2 and allocates two CUs. This means there are two and one free CUs available in PoP1 and PoP2, respectively.

In the second step, shown in Figure 5.7b, $VNF_3$ with four CUs is requested and should be started in PoP1. During the processing of the request, vim-emu contacts the resource model and asks for the additional four CUs in PoP1. In this example, the request cannot be fulfilled since the used model does not allow oversubscription and thus rejects the request. As a result, vim-emu is not able to start $VNF_3$ in PoP1.

#### 5.5.1.2. Model B: Cloud-like oversubscription

Our second model does not enforce a fixed CU limit per PoP. Instead, it allows oversubscription, which is a typical concept in IaaS clouds. For example, OpenStack Nova sets its default `cpu_allocation_ratio` property to 16:1 which means that for each physical CPU core of an OpenStack cluster up to 16 virtual CPU (vCPU) cores can be allocated [Ope16]. This results in situations in which compute instances get less resources than initially requested and allocated because cores are shared with other compute instances. As a result, a VNF might slow down if more and more VNFs are started in the same PoP so that the PoP becomes oversubscribed.

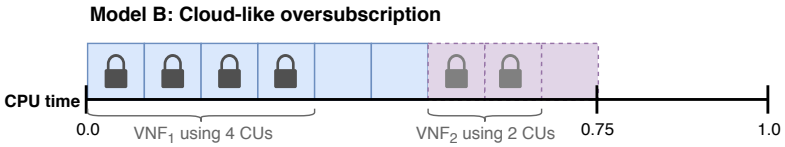
To model this behaviour, we use a so called "oversubscription factor" which we define as the fraction of available and currently used CUs in a PoP as shown in Equation 5.3. This factor reduces the available CPU time of all containers within the same PoP once more CUs are requested than there have been initially assigned to that PoP, i.e., $\text{ac}_p > \text{mc}_p$. We do this by updating the resource limits of all containers of PoP $p$ whenever a new container is added or removed from $p$.

The key concept here is that only the CPU time per CU of the oversubscribed PoP is reduced *but not* the CPU time of other PoPs. As a result, our model creates a realistic environment in which oversubscription in one PoP does not influence the VNF instances running in another PoP. Using this model means, in particular, that each PoP offers a virtually infinite number of CUs and will never reject a request. Instead, the absolute CPU time per CU in the oversubscribed PoP will be continuously reduced as more and more CUs are allocated. This is the main difference to Model A. Mixed forms of both presented models are, however, possible, e.g., the oversubscription factor of Model B could be bounded so that, at some point, requests are rejected like it is done by Model A.

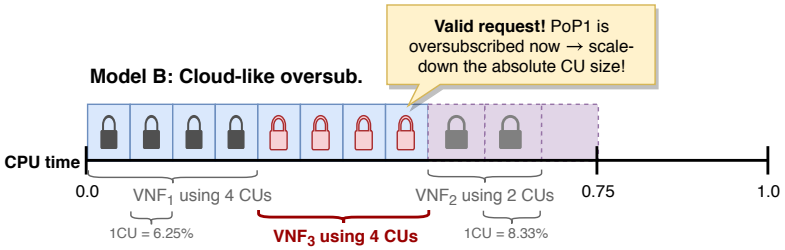
$$f_p(\text{nc}_c) = \frac{E_{cpu}}{\sum_{i=1}^{N} \text{mc}_i} \cdot \underbrace{\frac{\text{mc}_p}{\max\{\text{ac}_p; \text{mc}_p\}}}_{\text{oversubscr. factor}} \cdot \text{nc}_c \tag{5.3}$$

This model provides a playground for realistic multi-PoP scenarios, e.g., an overloaded PoP might motivate a MANO system to reassign its containers to other PoPs with better performance. As a result, the previously oversubscribed PoP is relieved and the performance of containers it is hosting improves.

**Example:** We demonstrate Model B with our previously used example. Figure 5.8a again shows the situation in which $\text{VNF}_1$ and $\text{VNF}_2$ are deployed and running on PoP1 and PoP2. In this situation, the instantiation of $\text{VNF}_3$ that wants to use four CUs is requested. In contrast to the previous example, the request is not rejected as shown in Figure 5.8b. Instead, PoP1 is oversubscribed and the absolute CPU time per CU in PoP1 is scaled-down by the oversubscription factor to make room for the requested CUs. After that, a single CU in PoP1 maps to only 6.25 % CPU time once $\text{VNF}_3$ is deployed. As a result, all eight CUs of $\text{VNF}_1$ and $\text{VNF}_3$ fit into PoP1. The total absolute CPU time available for PoP1 stays, however, constant so that the oversubscription has no effect on PoP2 and its VNFs.




(a) Step 1: $VNF_1$ (using 4 CUs) and $VNF_2$ (using 2 CUs) are deployed and running on PoP1 and PoP2



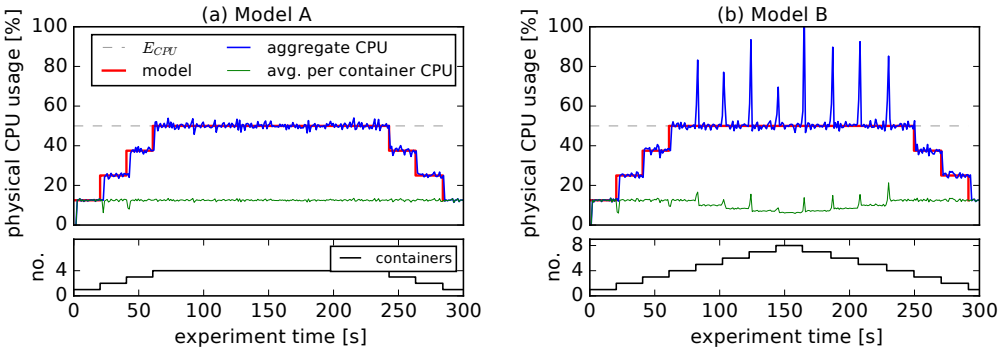

(b) Step 2: Instantiation request of $VNF_3$ (with 4 CUs) sent to PoP1 is granted by oversubscribing PoP1 and reducing the absolute CPU time of the CUs of PoP1

Figure 5.8.: Example scenario using Model B: Cloud-like oversubscription

### 5.5.2. Implementation

Implementation-wise, our system does not use Docker's default CPU share limitation API since it is not sufficient for our use case. The first reason for this is that it only limits the CPU share if two or more containers want to utilise the entire CPU at the same time. It does not limit the CPU time if only one container is utilised and the competing ones are idle. Instead, we use the "CPU bandwidth control" functionalities of Linux's completely fair scheduler (CFS) [TRR10] which allows us to assign a fixed upper limit of CPU time to each container in the system. The second reason for our custom implementation is that the used Docker API does only allow to set resource limitations when a container is started but not to update limits at runtime. We bypass these shortcomings by directly manipulating the `cgroup` system properties. Using this, Containernet sets the CPU `period` and CPU `quota` fields in the CFS properties of a given container according to the percentage of CPU time this container should be able to use. This gives a fine-grained control over the absolute fraction of CPU time a single container is allowed to consume.

In addition to this, Containernet is also able to control the number of CPU cores available to each container as well as to pin a container to a specific set of cores on the host machine. This can, for example, be used to pin all containers of an emulated PoP to a single core to achieve better isolation between containers assigned to different PoPs.




(a) Model A: Fixed Limit (b) Model B: Cloud-like oversubscription

Figure 5.9.: Modeled vs. measured container CPU usage

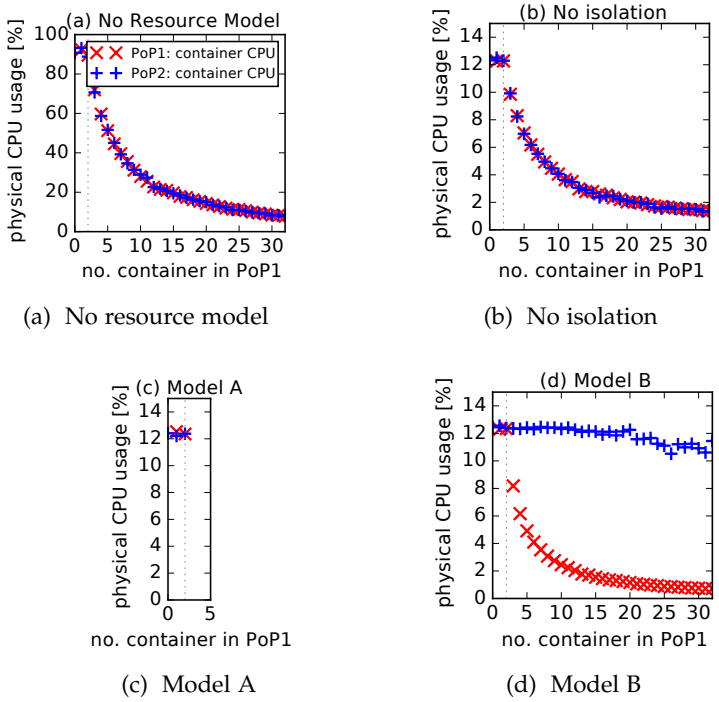
### 5.5.3. Evaluation

We perform a series of experiments to prove the presented concept, validate the behaviour of the introduced resource models, and to showcase the system. The experiments use topologies with one and two PoPs in which Docker containers that run a workload generator (`stress`) are started. The workload generator is configured such that every container always tries to fully utilise the CPU. The overall available CPU time for all containers of the emulation is set to $E_{\text{cpu}} = 0.5$ and the experiments are executed on a single physical machine with Intel(R) Core(TM) i5-4690 CPU @ 3.50GHz and 16GB memory.

The goal of our first experiment is to validate our implementation by checking that the measured CPU usage of containers in a single PoP is aligned to the theoretical CPU usage computed by our models. During the experiment a new container (requesting one CU) is allocated every 20 s during the experiment until a total of eight containers are requested. After additional 20 s, these containers are terminated one by one, again using 20 s between each termination request. The maximum limit of available CUs in the PoP is set to four CUs so that some of the requests are rejected (Model A) or the PoP becomes oversubscribed (Model B).

Figure 5.9 shows the aggregated CPU usage for the entire PoP as well as the average CPU usage for a single container, both measured with Docker's status API that returns detailed CPU time statistics. Additionally, the expected CPU utilisation calculated by our models is plotted. The results show that the measured CPU utilisation for all containers in the PoP is close to the limits computed by the model. The graph at the bottom of the figures shows the number of running containers in the PoP.

Figure 5.9a shows how Model A rejects requests after four containers are running and thus enforces the fixed upper resource limit of the PoP. Model B,




(a) No resource model (b) No isolation

(c) Model A (d) Model B

Figure 5.10.: Cross-PoP resource isolation using different resource models

in contrast, accepts all eight containers since it allows oversubscription, which is shown in Figure 5.9b. It also shows how the available average CPU time, available to all containers, is reduced when the PoP becomes oversubscribed. An interesting observation in Figure 5.9b are the spikes in the measured CPU usage. They happen during the reconfiguration of CPU limits of already running containers which is needed by Model B and happens whenever a container is added or removed from the system. The root cause for this seems to be Linux's `cgroups` implementation. Still, those peaks happen only for short time periods (usually less than 1 s) and thus do not have a big impact to long running experiments. Investigating this issue will be part of future work.

The second experiment shows how the presented resource models provide resource isolation between PoPs. It emulates a topology with two PoPs, each with a limit of two CUs. During the experiment, the average physical CPU time available for a single container is measured for different numbers of `stress` containers running in PoP1. The number of `stress` containers running in PoP2 is fixed to two containers. With this setup, we can observe how the changing number of containers in PoP1 influences the performance of containers in PoP2.

Figure 5.10 shows the results for different resource models. Figure 5.10a shows what happens when no resource limitation model is used. All containers

compete for the entire physical CPU time and the performance of containers in PoP2 is reduced when more containers are added to PoP1. The same happens in Figure 5.10b with the difference that it uses a common resource model for both PoPs. Thus, the containers do not influence each other until the maximum of two containers is running in PoP1. Figure 5.10c shows the behaviour of Model A which does not allow oversubscription and rejects all requests when two containers are already running in a PoP. The behaviour of Model B is shown in Figure 5.10d. The figure validates that the model enables resource isolation between PoPs even when PoP1 is oversubscribed and the CPU time for each of its containers is reduced. Figure 5.10d shows that there is still some minor impact to the performance of PoP2 when containers are added to PoP1, i.e., PoP2's performance is reduced by around 2 % when PoP1 is oversubscribed by a factor of 16×. Our results show that vim-emu can simulate resource limitations in multi-PoP environments while ensuring resource isolation between PoPs. As a result, local prototyping of realistic multi-PoP NFV scenarios becomes possible.

## 5.6. Conclusion

We introduce vim-emu, a prototyping platform for NFV that goes beyond its initial NFV use cases and is an excellent prototyping and test platform for all kinds of distributed services. Our experiments show that vim-emu can emulate realistic network conditions and resource limitations in multi-PoP environments while ensuring resource isolation between PoPs. By using Docker containers to execute VNFs within our emulation platform, vim-emu allows developers to directly move their tested services into production without requiring additional changes, which speeds up development cycles.

We argue that vim-emu is an important step towards a fully integrated development support toolchain for NS development. The entire project has been recently adopted by ETSI OSM which offer free hosting of vim-emu's open-source code [ETS17b] and provide a Jenkins-based CI/CD pipeline as well as code review and bug tracking tools for the project. The following chapters will present several extension to vim-emu, e.g., the integration with different MANO systems and advanced service chaining techniques.

# 6. Adding NSH-enabled SFC prototyping capabilities

Even though vim-emu already supports basic service chaining, e.g., using a VLAN-based approach to separate different VNF-FGs and VNF-FPs, it does not support more advanced service function chaining (SFC) technologies as they are standardised by IETF [QN15]. To rectify this, we build an extension to the vim-emu platform that adds support for NSH-based service chaining. This chapter contains figures and verbatim copies of my paper [Peu+18a], which is based on work I did together with Frédéric Tobias Christ. It introduces the general concept to integrate NSH into vim-emu, which was designed by me, and presents a prototype implemented by Mr. Christ in his Bachelor thesis [Chr18; CP18]. Mr. Christ not only contributed to the prototype implementation but also helped to run the presented experiments, under my supervision, which lead to the presented results; they were analysed by me and are presented as part of a case study.

## 6.1. Introduction

A key concept to deploy complex NSs within NFV environments is SFC as defined by IETF [QN15]. SFC allows to combine multiple VNFs or CNFs into larger services. In this context, single VNFs or CNFs are usually referred to as service functions (SFs). Multiple SFs are traversed by packets according to the SFC configuration, using one or multiple service function paths (SFPs) [HP15]. To simplify forwarding along an SFP, Quinn et al. [QEP18; QG14] introduced NSH as a possible protocol to encapsulate and mark packets according to the SFPs they are assigned to. The main benefits of NSH are the possibility to create topology-independent SFPs and its ability to pass arbitrary metadata between the involved SFs, e.g., classification information.

However, in a recent survey, Medhat et al. [Med+17] identify the lack of NSH support in current switch and SF implementations as one of the key challenges for SFC, which is limiting its applicability and leading to unnecessary complexity in today's NFV solutions. We argue that one reason for the lack of practical implementations are missing prototyping environments in which NSH-enabled network functions (NFs) can be easily developed and tested.

Even if major VIMs, like OpenStack, are currently integrating SFC solutions into their platforms, their availability for prototyping is still very limited. One reason for this is that they focus on production-grade systems rather than on systems that offer development support, e.g., debugging.

To this end, we extend vim-emu with support for NSH-based SFC [Peu+18a]. This extension allows researchers as well as function and service developers to quickly test their NSH-enabled components and to validate and verify their functionality before putting them into production. The resulting NSH-enabled platform can be used in three different ways. First, an SF developer can use our platform to validate and verify that an SF behaves correctly in an NSH-enabled SFC setup. Second, SFC integrators can test complex service chains with many SFs in a controlled environment. Third, MANO solutions can use our platform as an experimentation and test backend to test their service orchestration functionality, e.g., to verify that their chaining logic requests correct forwarding paths.

To simplify the prototyping process, we also design a set of pre-packed auxiliary components such as a generic NSH-enabled SF and a traffic generator for NSH encapsulated traffic. Using these components, we perform a case study that showcases the resulting platform.

## 6.2. Related work

Besides the request for comments (RFC) mentioned in Section 6.1, the SDN and NFV communities investigate a variety of different aspects within SFC, e.g., regarding resilience against failures [BBS16; Med+16] or dynamic readjustment to changed service requests [Liu+17]. So far, these research findings have been mostly evaluated using simulation [BBS16; Liu+17] or heavyweight testbeds [Med+16], which are not available to every developer and are hard to set up. Future evaluations and case studies can benefit from the lightweight prototyping solution proposed in this chapter.

Davoli et al. provide a proof-of-concept implementation of an SFC control plane using NSH [Dav+17] in combination with a set of OVSs used as service function forwarders (SFFs), which is similar to our design. Each of their SFs is, however, also implemented around an OVS instance which is different to our approach in which SFs can be simply realised with lightweight and easy-to-use Docker containers. Further, our approach focuses on enabling quick prototyping, e.g., quickly spin up complex SFC scenarios on a developer's laptop, including emulated multi-PoP scenarios, whereas the proof-of-concept presented in [Dav+17] consists of manually configured VMs connected to a fixed topology.

We build our NSH-based prototyping platform on top of vim-emu, which is highly scalable and can efficiently emulate hundreds of PoPs [Peu+18b]. This makes it a perfect fit for lightweight NSH prototyping and large-scale NSH experiments. Other emulation platforms like Mininet [LHM10], Maxinet [Wet+14], or VLSP [MCG15] are not suitable for prototyping of SFC approaches as they do not emulate NFV infrastructure to which hosts can be added or removed at runtime and be chained dynamically. In contrast to vim-emu, they do not provide standard VIM interfaces, which allow using MANO systems for managing chained network services. While VLSP provides some support for attaching MANO systems, it does not allow to execute real-world SFs.

Pelle et al. [Pel+15] provide a framework for troubleshooting and debugging SDN, but it does not focus on quick prototyping of SFC. Similarly, ESCAPE [Cso+14] is not suitable for quick prototyping of SFC approaches as it focuses on orchestration between non-emulated PoPs. Finally, simulation approaches [Cal+11; Zha+12; Hen+08] do not support real SF implementations and can thus not be considered as prototyping environments. Hence, emulation platforms are more appropriate for prototyping and the validation and verification of realistic SFC setups, functions, and tools.

## 6.3. Requirements

To solve the problem of missing prototyping platforms for NSH-enabled SFs, we first collected the requirements that such a platform should fulfil to properly support developers: (i) The platform has to be able to quickly deploy the prototyped SFs. This includes the execution of SFs written in different programming languages using arbitrary frameworks and libraries. (ii) A developer should be able to configure arbitrary network topologies in which SFs or complex SFCs can be tested. (iii) The created test networks should allow to transport and deliver real network traffic and implement the correct forwarding behaviour of NSH-encapsulated packets. (iv) A prototyping platform should seamlessly integrate with other tools commonly used in the NFV landscape, e.g., MANO systems.

Even though vim-emu already fulfils requirements (i) and (ii), it still lacks support for NSH-based SFC and only provides a very simplistic chaining model using VLAN tags statically assigned to SF ports. Hence, we address requirements (iii) and (iv) by extending vim-emu as shown in Figure 6.1 and adding support for NSH. We describe those extensions in the following sections.

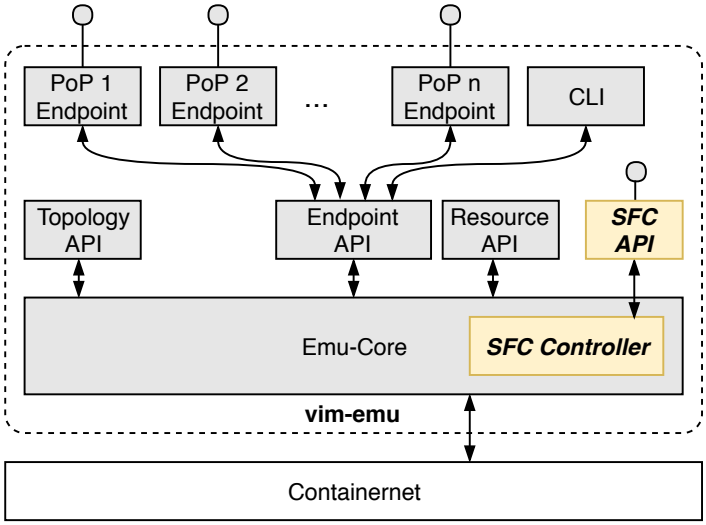



Figure 6.1.: Extended vim-emu architecture with additional SFC controller and API (based on [Chr18])

## 6.4. Adding NSH support to the emulation platform

Starting from the bottom, we first look at the networking layer of vim-emu and identify the required changes to support NSH. Each emulated network topology consists of multiple PoPs and each PoP in a vim-emu emulation is represented by a single virtual SDN switch instance. Thus, every PoP simplifies its internal network using a big-switch abstraction, turning an emulated multi-PoP topology into a much simpler network of virtual switches as shown in Figure 6.2. All these switches can be controlled by a single SDN controller, fitting to the IETF SFC architecture and NSH design, which expects to be deployed in a single control domain [QEP18].

The next question is whether the used switches support NSH encapsulated packets, i.e., match the NSH fields, and can be used as SFFs [HP15]? Fortunately, OVS already comes with experimental support for NSH starting with version 2.9. It supports the installation of NSH-specific rules using either its command-line client (`ovs-ofctl`) or OpenFlow using the extensible match (OXM) feature introduced in OpenFlow 1.2 [McK+08; Ope11]. Since OVS version 2.9 is fully compatible to earlier versions, it can directly be integrated into vim-emu without loosing any other functionalities or features.

Making use of the NSH support of the involved OVS switches has the benefit that each switch in the emulated multi-PoP topology can be directly used as SFF, as shown in Figure 6.2. It, in particular, means that prototyping scenarios running on vim-emu do not have to deal with legacy networking components that do not offer support for NSH. This simplifies the use of the platform, e.g., an SFP can be directly established between any SF pair in the system.

An important advantage of our design is the fact that all emulated PoPs are part of a single layer 2 network. Further, vim-emu allows to flexibly define and change the used topologies and provides realistic inter-PoP connections, e.g., emulated delays. The benefits of this become more visible when comparing our platform to a testbed-based multi-PoP prototyping setup that relies on multiple OpenStack installations at different physical locations [Sof17]. Even though such setups provide all features of OpenStack, e.g., support for VMs, they also come with many downsides. First of all, interconnecting multiple OpenStack installations over the Internet, to build a reliable network to test different SFC scenarios, is a tedious and error prone task, e.g., to setup tunnels between the OpenStack sites. Another problem is that OpenStack does only come with support for SFC within a single OpenStack domain, e.g., its API has no notion of inter-PoP chains as described in Section 6.4.2, making SFC setups between two OpenStack sites challenging. Finally, testbeds consisting of multiple OpenStack sites are usually quite inflexible and it is often not possible to change their topology on-demand; this collides with requirement (ii).

Another reason why vim-emu puts all PoPs and the involved SFFs into a single layer 2 network is that it allows us to simply use Ethernet to transport the NSH encapsulated packets [QEP18], as shown in Figure 6.2. If other transport protocols should be used within our platform the user can extend the SFC controller accordingly.

Considering all this, it becomes clear that our emulation-based design removes a lot of complexity from the SFC setup if multi-PoP scenarios are considered. From the perspective of the prototyped SFs and SFCs it provides, however, a full-featured and standard-compliant SFC architecture [HP15], once NSH is supported by the underlying network emulation and requirement (iii) is satisfied. As next step, the core layer of vim-emu needs to be extended as shown in Figure 6.1. This extension is twofold. First, an "SFC controller" component is added to the emulator core; it translates high-level chaining requests into low-level rules and installs them into the involved switches as described in Section 6.4.1. Second, we extended vim-emu by additional SFC APIs that allow a user to create and configure SFPs between the deployed functions (Section 6.4.2).

### 6.4.1. SFC controller

The SFC controller is added to the emulator core and receives chaining requests from the SFC API. These requests contain the identifiers of the ports, i.e., network interfaces of the SFs, to be chained. The controller then translates those high-level chaining requests and installs the resulting rules in the switches of the emulated topology. This process uses the available knowledge about the topology to calculate the shortest path (using number of hops or emulated

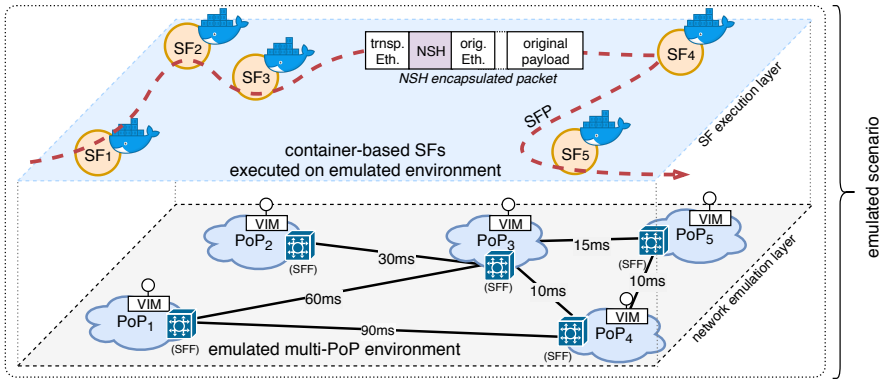



Figure 6.2.: Emulated network scenario with five interconnected PoPs and five Docker-based SFs deployed among them (logos from [Doc13])

link delays) between the two SFs that are about to be chained. It generates the so-called rendered service path (RSP) [HP15], containing a list of all SFs and SFFs that should be traversed by the packets assigned to the path. Our current prototype [CP18], which was developed as part of Mr. Christ's Bachelor thesis under my supervision [Chr18], simplifies the SFC model at this point by only using the first available path between two SF ports. As a result, our prototype does not support multiple paths between the same SFs, which is not needed for the case study presented in Section 6.5 and simplifies the implementation of the prototype controller by not requiring load balancing features as considered by the IETF [HP15]. It is, however, possible to extend the presented prototype with those features and our API already provides the basics for this by supporting the concept of `port pair groups` as discussed in the next section. A possibility to implement load balancing among multiple SFPs, and maybe multiple instances of the same SF, is extending the SFC controller to instruct the involved SFFs to distribute the traffic among the available paths, e.g., based on policies defined in the controller.

In our prototype, we use a custom implementation for this SFC controller based on the Ryu SDN controller [Ryu17]. We pick this approach over existing SFC implementations, provided by SDN controllers like OpenDaylight [The13], firstly because they are heavyweight and would destroy the lightweight nature of the presented platform. And secondly, the SFC implementations of controllers like OpenDaylight [The13] focus on production-grade systems and are tailored to interface with existing cloud deployments, e.g., they deploy their own OVS instances as SFFs which does not fit our needs.

### 6.4.2. SFC API

As vim-emu already comes with APIs that mimic the OpenStack northbound interface to start, stop, and configure SFs inside the emulated PoPs, we align our chaining API prototype to OpenStack as well. More specifically, we align our SFC API with the OpenStack Neutron SFC API model [Ope18] by reusing the concept of having three basic elements that define an SFP—or in OpenStack terminology a `port chain`. A `port chain` is defined as an ordered set of `port pair groups`, each containing at least one `port pair`. Each port of a `port pair` corresponds to a specific network interface of an SF and thus the ordered set of `port pairs` within the `port chain` defines how packets should traverse the different SFs. The reason to additionally wrap `port pairs` with `port pair groups` is to group SFs that provide equivalent functionality and enable load balancing among them [Ope18].

Following this model, we add a new REST interface to vim-emu that offers API endpoints to create, list, update, and delete `port pairs`, `port pair groups`, as well as `port chains`. This API design is closely following the OpenStack SFC approach and simplifies the integration of our prototyping platform with other NFV solutions, e.g., MANO systems, to satisfy requirement (iv).

To support inter-PoP chaining, we provide a single chaining API endpoint for all PoPs of an emulated network instead of providing one endpoint per PoP. Using a single endpoint is, however, different from the original OpenStack approach, which only deals with SFC setups within a single OpenStack domain. We made this design choice to allow developers to easily create complex chaining setups in multi-PoP scenarios and simplify the use of our prototyping system (ii), even though it results in slightly different semantics compared to the original OpenStack interfaces. We have later evolved this implementation, which is used for the prototype presented in this chapter, to a second SFC API implementation that provides one endpoint per PoP. This additional API can be consumed by the native OpenStack client tools and has been tested in combination with OSM. This additional work has been done as part of a Master thesis by Erik Schilling [Sch19] under my supervision.

### 6.4.3. Simplified prototyping using pre-packaged SFC components

Having a prototyping platform for NSH-enabled SFs available is already helpful. But we have noticed that building such NSH-enabled SFs is challenging because there are no usable SF implementations that implement NSH yet. Also the Linux kernel module that plans to add native NSH support to the Linux kernel [Dat17] has not been available when this research has been performed. Even though this kernel module is available at the time of writing, it must still

be considered as experimental and lacks documentation. As a result, we introduce some easy-to-use, pre-packed NSH-enabled SFs to support developers to quickly setup NSH experiments and to experiment with complex SFCs.

More specifically, we add an NSH-enabled forwarder SF as well as a basic NSH traffic generator SF to our platform. The forwarder receives NSH-encapsulated packets from an SFF, logs the information from the NSH such as service path identifier (SPI) and service index (SI) for debugging and analysis purposes, decrements the SI field in the NSH according to the IETF-defined behaviour [QEP18], and returns the packet to the SFFs from which it was received so that it can be forwarded to the next SF.

The traffic generator can be used to generate flows of NSH-encapsulated packets with configurable rate, SPI, and SI values. It provides the means to test if the prototyped SFC scenarios correctly forward and handle NSH encapsulated packets, but is rather limited in terms of performance and configurations options for the generated traffic. The reason for this is that the traffic generator as well as the NSH-enabled forwarder SF are both implemented using the Scapy Python library [Sec15]. The first reason why we pick Scapy is because all SFs in our platform are executed as Docker containers which share their kernel with the host operating system. As a result, we would need to introduce a complex dependency to the host operating system if we want to make use of the experimental NSH kernel module. The problem of Scapy is, however, its limited performance since it is implemented in Python and runs as a normal user space application—an acceptable drawback since the main focus of our platform are functional tests of SFC setups. The second reason for our design decision is that Scapy already implements an NSH packet header module that follows the official standard [QEP18] and allows for quickly prototyping NSH-enabled SF implementations.

We assume that this situation will change in the future and once a fully-working NSH implementation is available in the kernel of the host machine on which our platform is executed, the prototyped SFs should be able to make use of it without further changes of our platform. This will also allow to use existing, more sophisticated traffic generators. An alternative option for generating NSH traffic with higher data rates might be the use of packet traces that contain NSH packets and can be replayed with higher rates using tools like `tcpreplay` [KA00a]. In any case, our Scapy-based traffic generator provides all the features required to perform a case study that showcases the functionality of the presented platform as described in the next section.

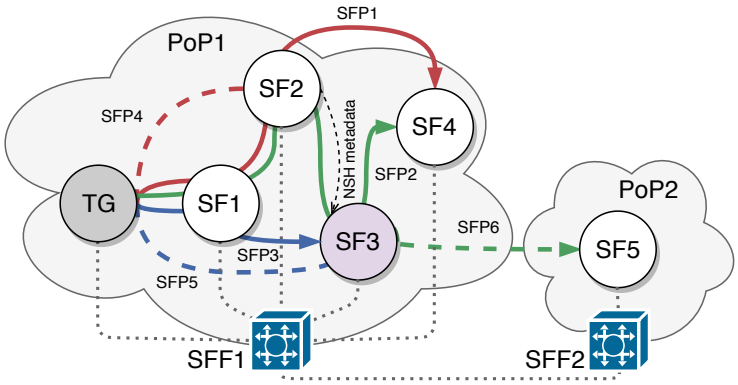



Figure 6.3.: Experiment setup over two PoPs: A traffic generator (TG) is injecting packets into an SFC with five SFs and six different SFPs.

## 6.5. Case study

We use our vim-emu-based prototype to perform a case study which showcases that the presented platform design can be used to prototype complex SFC scenarios across multiple PoPs. This case study does not only demonstrate the use of NSH to realise multiple SFPs between a set of SFs, it also verifies that all packets are correctly forwarded by our platform. We also show how reconfiguration functionalities, i.e., reclassification of packets that rely on NSH metadata, can be tested on our platform.

The presented experiments have been executed by Mr. Christ under my supervision [Chr18]. In the experiments, we use a complex SFC that consists of one traffic generator (TG) and five forwarder SFs that are deployed on two PoPs as shown in Figure 6.3. The traffic generator is located at the beginning of the SFC and mimics an SFC classifier [HP15] that encapsulates incoming traffic with NSHs. We establish a total of six different SFPs among the available SFs as shown by the coloured lines in Figure 6.3: SFP1 (red), SFP2 (green), SFP3 (blue), SFP4 (red dashed), SFP5 (blue dashed), and SFP6 (green dashed). These paths are grouped (by colour) to pairs and the dashed paths represent alternatives to the solid paths. They are used to show how our platform can redirect packets when their SFP identifier is changed. $SF_3$ plays a special role in this setup, since this SF can reclassify packets from SFP2 to SFP6 and thus redirect the traffic from $SF_4$ to $SF_5$. We exploit this feature in our case study to demonstrate reclassification at runtime and the use of NSH metadata in our platform.

During an experiment, the traffic generator generates a pre-defined amount of packets with a given rate for each of the SFPs and sends them to the SFC. Detailed numbers about the generated traffic are shown in Table 6.1.

Table 6.1.: Generated traffic per SFP (based on data from [Chr18])

| color | path id. | rate | packets sent | ∑ packets sent |
|---|---|---|---|---|
| red | SFP1 | 8 pkt/s | 60 | 300 |
| | SFP4 | 8 pkt/s | 240 | |
| blue | SFP3 | 24 pkt/s | 380 | 580 |
| | SFP5 | 24 pkt/s | 200 | |
| green | SFP2 | 16 pkt/s | 440 | 440 |
| | SFP6 | 16 pkt/s | 0 | |

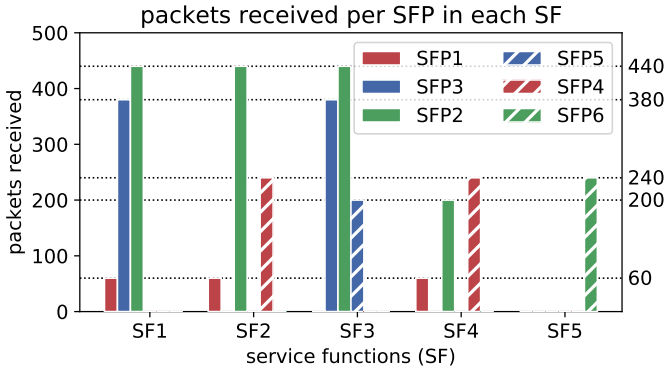



Figure 6.4.: Total packets received per SFP in each SF during the experiment and the expected values shown as horizontal lines (based on data from [Chr18]).

In each SF, received packets are identified by their SPI and the number of packets seen per SFP is logged. This allows us to verify that the right number of packets, belonging to the right SFP, traverses the correct set of SFs. Figure 6.4 shows these counters for each SF and verifies that the expected amount of packets has been seen (indicated by horizontal lines). The goal of this is to show that our platform and the implemented SFC approach does not loose packets and does not forward packets to the wrong SFs. This experiment can be understood as a functional test of our platform. We do not measure the maximum forwarding performance of the SFC because of the used Scapy-based traffic generator that is not able to generate high data rates, as described in Section 6.4.3. The expected forwarding performance of the system depends, however, mainly on the performance of the used OVS switches which can easily process multiple gigabit per second (Gbit/s) on a normal laptop as we verified by running non-NSH traffic through the platform. A user of our platform can, in general, expect that the forwarding performance of the tested SFCs is similar to the performance achieved by network experiments executed on Containernet [Peu16] or Mininet [LHM10].

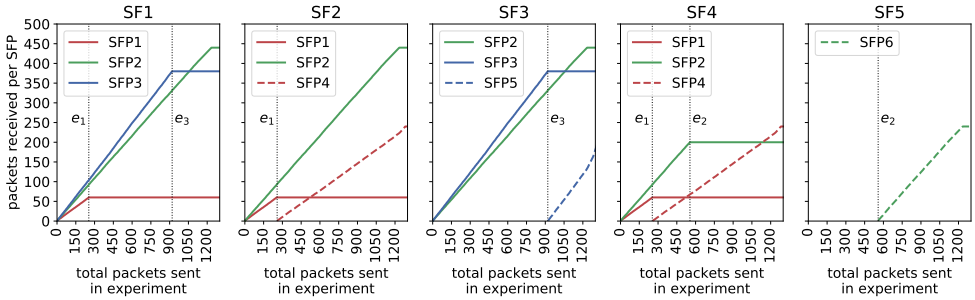



Figure 6.5.: Packets received per SFP over the total number of packets sent to the experimental SFC. One plot per SF and vertical markers for events $e_1$, $e_2$, and $e_3$ (based on data from [Chr18]).

Having verified that the total number of packets seen by each SF is correct, we investigate a more dynamic scenario in which we change the assigned SFPs of the generated traffic and activate the reclassification in SF3 at predefined points in time. During the presented scenario, the following events are triggered in the system to simulate dynamic changes:

- $e_1$: The traffic generator stops the generation of packets for SFP1 and starts to generate packets for SFP4 ⇒ SF1 does not see red packets anymore. SF2 and SF4 still see the red packets.
- $e_2$: SF2 starts to inject metadata into the NSH of packets belonging to SFP2 which causes SF3 to reclassify SFP2 packets to SFP6 packets ⇒ SF4 does not see green packets anymore and SF5 starts to receive green packets.
- $e_3$: Generation of SFP3 packets is stopped and SFP5 traffic is generated instead ⇒ SF1 does not see blue packets anymore and SF3 still sees blue packets.

The goal of this experiment is to test if our platform can be used to prototype more complex and dynamic SFC scenarios. It also shows that the auxiliary SFs shipped with our platform (traffic generator and forwarder SF) correctly handle the NSH traffic and can be used to build prototypes that make use of advanced NSH features, i.e., NSH metadata.

Figure 6.5 shows the events and the counters for the packets seen per SFP over the total number of generated packets for each of the SFs in the experiment. It verifies that the correct number of packets arrives at the correct SFs. It also shows how the three events impact the flow of the packets in the system, e.g., how packets marked with SFP1 and SFP2 disappear in SF1 at event $e_1$ and $e_3$, respectively.

A special case in this experiment is marked by event $e_2$. At this point in time, SF3 starts to reclassify packets marked with SFP2 and changes their identifier to SFP6. As a result, the involved SFFs forward the packets to SF5 instead of

SF4. This also explains why we do not need to generate SFP6 traffic as shown in Table 6.1. To trigger the event in SF3, we exploit the metadata field of NSH: Once SF2 has seen more than 200 packets of SFP2, it starts to set a flag in the NSH metadata field of the processed packets. SF3 then reacts to this flag in the metadata field and starts the reclassification. This example shows how an SFC with dynamic reclassification mechanisms can be prototyped in our platform and how developers can easily play with the advanced features of NSH, e.g., metadata transport between SFs. The case study shows that the presented platform works correctly and can be used to locally prototype complex SFCs using NSH.

## 6.6. Conclusion

NSH can be considered as one of the key enablers for the wide adoption of SFC. However, the availability of platforms, frameworks, kernel modules and tools that support NSH is still limited, which makes experimentation with this technology challenging. The presented prototyping platform changes this and enables researchers and developers to quickly prototype NSH-enabled components or test novel service management systems against an easy-to-deploy, NSH-enabled platform. The presented platform is very lightweight and still allows for experiments with complex SFCs using advanced NSH features such as metadata-triggered reclassification, as shown by the presented case study. Our platform should be understood as starting point which provides the means for custom NSH-related developments. It is designed to be extended by its users, e.g., to add support for further transport encapsulations by extending its SFC controller.

# 7. Supporting the evolution of MANO systems using emulation-based smoke testing

In this chapter, I present an approach and a platform to test MANO systems in large-scale NFV scenarios. The chapter uses figures and verbatim copies of text from my papers [Peu+18b] and [Peu+19a]; it uses vim-emu version 4.0 [ETS17b]. The presented concepts, designs, and experiments have been entirely done by me. However, the publications are joint work together with Michael Marchetti and Gerardo García de Blas, who gave support during the first integration of vim-emu into the OSM project, e.g., to setup the used Jenkins pipelines in the OSM infrastructure. The chapter starts by motivating the need for prototyping and testing MANO systems in Section 7.1. After that, it provides background about the integration of MANO solutions into the NFV ecosystem and about the concept of smoke testing in Section 7.2. Section 7.3 presents the relevant related work. After that, I introduce "emulation-based smoke testing" for MANO solutions in Section 7.4. This approach and the scalability of our prototyping platform is then evaluated and a case study with two versions of OSM is presented in Section 7.5. Finally, the benefits of the presented approach are discussed in Section 7.6 and the chapter concludes in Section 7.7.

## 7.1. Introduction

The presented prototyping platform turned out to be very helpful for VNF, NS, and SFC development, including complex NSs such as 5GTANGO's smart manufacturing pilot [Peu+19d]. However, NFV is still a novel topic and research and innovation does not only happen on the VNF and NS layer. In fact, the MANO part of the NFV architecture is one of the main topics of interest, due to many open research questions and optimisation opportunities, e.g., automated placement [Sou+19; MKK14]. This raises the question how to quickly prototype and test new MANO features? And in particular, how to test MANO solutions in realistic, large-scale multi-PoP environments without

having a large, distributed testbed available? An example for this are placement optimisation algorithms that need to be tested in realistic environments and different topologies, as we show in [SPK18].

This motivates us to reuse and extend our existing prototyping platform and its interfaces to turn it into a prototyping and test environment for MANO systems. To this end, we present an emulation-based test platform, enabling automated tests of MANO systems in large multi-PoP scenarios. The presented solution is inspired by a concept called "smoke testing" [DRP99] which focuses on testing only the main functionalities of a system and skips unimportant details to reduce test times (Section 7.2.2). Our platform does exactly this by re-implementing a subset of the OpenStack APIs, the de-facto standard for VIM interfaces, today. A MANO system can then use these APIs to deploy container-based test NSs on the emulated NFVI PoPs. By adding an ETSI standard-compliant test suite to the setup we can automatically test MANO systems in end-to-end scenarios. We show this in a case study in which we test two major versions of OSM. We also show that our test platform prototype can easily emulate up to 1024 PoPs on a single physical machine and that it can reduce the setup time of a single test PoP by a factor of 232× compared to a DevStack-based PoP installation. During this work, we discovered some interesting insights and bugs that would not have been found with existing, lab-scale NFVI testbeds offering only a handful of PoPs.

## 7.2. Background

Before presenting our solutions, we first analyse components and interfaces required to test MANO systems and give deeper insights into the smoke testing concept.

### 7.2.1. Management and orchestration in NFV

MANO systems are complex software systems and represent the main control entity in NFV-enabled network infrastructures. Besides basic LCM tasks, MANO systems are also responsible for performing more complex orchestration tasks, like scaling, self-healing, or failover management [Par+18] (see Section 2.2.4). Those tasks are often automated and part of a closed control loop, which uses monitoring data as inputs to trigger orchestration decisions based on pre-defined policies or service-/function-specific management algorithms [Kou+18]. To provide all these functionalities, MANO systems usually interface with a high number of external components, as shown in Figure 7.1. The figure shows a simplified version of ETSI's NFV architectural

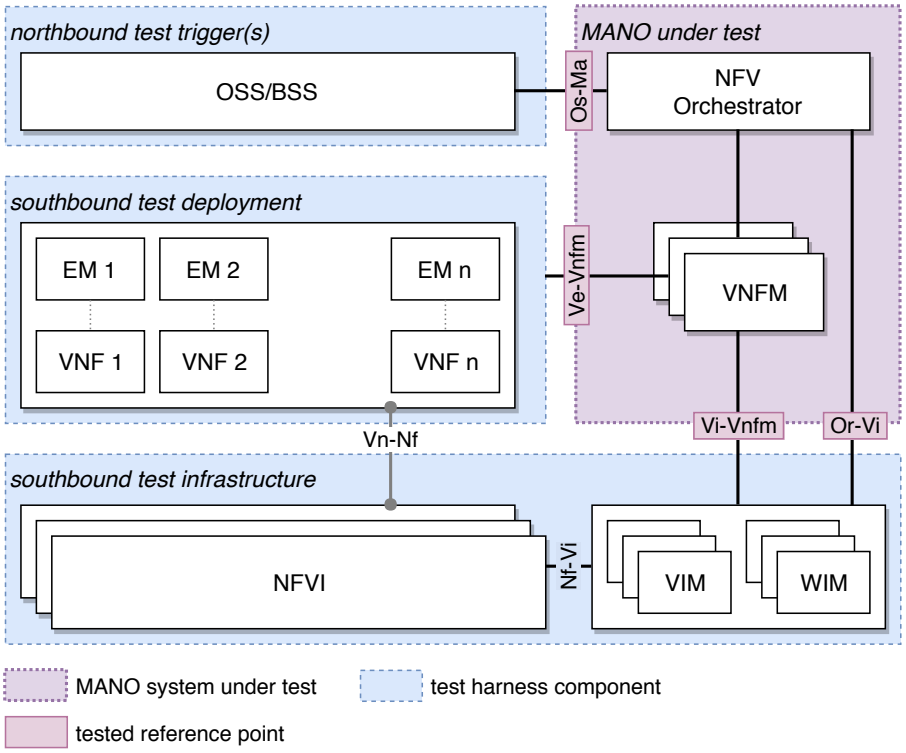



Figure 7.1.: A simplified version of ETSI's NFV architectural framework [ETS14b] showing the main components of an NFV environment, including the MANO system which we want to test. The figure highlights which of the NFV components need to be mocked to build a test harness for ETSI-aligned MANO systems.

framework [ETS14b] and highlights the MANO system and its interfaces to external components.

In general, the interfaces of a MANO system can be categorised into northbound and southbound interfaces. The northbound interfaces are those interfaces used by service providers, platform providers, or OSSs/BSSs to trigger LCM actions, like NS instantiation. They are consolidated within the *Os-Ma* reference point in the ETSI architecture (Figure 7.1). The southbound interfaces of a MANO system are considered to be those interfaces that connect to the underlying NFVI and the corresponding management components, like VIMs and wide area network infrastructure managers (WIMs). Those interfaces are part of the *Vi-Vnfm* and *Or-Vi* reference points. In addition, the interfaces that connect to the instantiated VNFs and NSs, e.g., for configuration and monitoring tasks, are also considered part of the southbound interfaces of a MANO system. They are represented by the *Ve-Vnfm* reference point.

When looking at this complex environment, it becomes clear that testing MANO systems in isolation, e.g., using unit tests, is not sufficient to ensure that they behave as desired. More specifically, testing solutions are needed

that efficiently test the interoperability of a given MANO system in different environments, e.g., a large number of connected VIMs in multi-PoP scenarios. Our proposed solution offers exactly this by providing a lightweight "test harness" for MANO systems. Figure 7.1 shows which of the components in the ETSI architecture need to be implemented by this test harness to build a full test environment around a MANO system. The first component contains the *test triggers* that connect to the northbound interface of a MANO system and trigger OSS/BSS actions. The second and most important component of the test harness is the *test infrastructure*, which is connected to the MANO's southbound interface and can be used by the MANO system to test NFV deployments without requiring one or more full-featured NFVI installations. Those deployments are the third component of the test harness, called "test deployments", for example, lightweight NFV services or service mockups.

### 7.2.2. Smoke testing

The term "smoke testing" was originally introduced by the electrical engineering community and describes a basic test to see if the tested device catches fire (smokes) after it is plugged into a power source. Later, the term "smoke testing" was taken up by the software testing community and used to describe rapid tests that verify that the most basic but critical functions of a system work as they should [McC96]. They are also called "build verification tests" and should be executed whenever a new built of a system (or of a subcomponent of that system) becomes available. They can be considered as a preliminary testing stage that is used to qualify builds for further, more complex tests, like regression or integration tests. The important thing here is that smoke tests do not substitute regression or integration tests; they are still needed to test every detail of a system. The main goal of smoke tests is to ensure that the basic functionality of a software product works, e.g., the program can be started and the default usage path does something meaningful. Using this approach, broken builds with major bugs are rejected early before more time and resource-intensive tests are deployed and executed [DRP99].

We have noticed that those smoke testing concepts perfectly match the problem of testing complex NFV MANO systems where testing suffers from the high resource demands of end-to-end tests due to the needed NFVIs. Our main idea is to use a more lightweight NFV environment, including a very lightweight NFVI, that allows to test the basic functionalities of a MANO system, e.g., NS on-boarding and initial instantiation. This can be done before testing the MANO system and all its features in a full-fledged NFV environment, which might not even be available to each individual developer of a MANO system. Section 7.4 presents our smoke testing approach for NFV in more detail.

## 7.3. Related work

Automated testing of NFV deployments is still a novel research direction with a limited amount of solutions. Most of them focus on testing NFVIs and their corresponding data planes or the corresponding VIMs, e.g., the test tool collection of OPNFV with projects like Yardstic, Functest or Nfvperf [Lin16]. They neither consider testing of VNFs, complex NSs nor MANO solutions, which makes those solutions complementary to our work. Some recent work focuses on end-to-end testing in 5G networks [Cat+16] or the verification and validation of NSs and VNFs [Zha+17], including our own work [Peu+19b]. Even though [Zha+17] and [Peu+19b] consider the case of applying integration tests in the NFV domain to test interoperability between different VNFs, none of them explicitly considers the need of testing the core part of NFV deployments: The MANO system. In the software engineering community, smoke testing has already been established since several years, providing the ability to quickly integrate new versions of different software components [DRP99], which is what our solution introduces for NFV MANO systems.

Another related research direction are automated performance tests of either VNFs, NSs, or NFVIs, which I present in Part III of this thesis. A handful of solutions have been proposed for performance testing of VNFs with the goal to characterise their performance under different configurations or in different environments [Cao+15; RBR17]. Some solutions focus more on end-to-end performance tests for complete NSs, like [PK17], arguing that testing the performance of a single VNF in isolation does not yield representative results. All of these solutions require an end-to-end deployment of the tested VNFs and NSs during their tests, but none of them focuses on testing the performance of the MANO system as such.

A straight-forward solution to setup those NFVIs for testing is to use testbed installations. Testbeds can either be installed locally, e.g., lab-scale installations, or third-party testbeds can be used remotely. In our early work [KRP13], we have proposed a locally installed multi-cloud testbed based on a handful of physical machines, each representing a single cloud site, i.e., a small OpenStack installation. Those machines are then interconnected and traffic-shaping solutions are added to emulate realistic delays between the sites. The problem with local installations, like [KRP13], are their limited resources that prevent large-scale test cases, e.g., with many PoPs. Remote testbeds, like [Chu+03; Sof17; Fed18], may offer the required NFV infrastructure and interfaces, but their main focus is the development, experimentation, and evaluation of NSs rather than being an infrastructure for automated test pipelines. Most of their infrastructure deployments and management functionalities are fixed, e.g., the used SDN controllers, VIMs, and MANO solutions, offering limited space for custom-tailored MANO tests. In addition, they are shared between many

users, which means they may not always be available to quickly execute automated tests on them. In general, these testbed solutions are complementary to our presented approach and should be used for final, manually-deployed integration tests rather than for automated smoke testing.

Another option for automated smoke tests is using locally available network emulation approaches, like Mininet [LHM10], CORE [Ahr+08], or VLSP [MCG15]. Unfortunately, these solutions focus on prototyping and evaluation of new protocols or network management paradigms rather than on interactions with production-ready MANO solutions. None of these solutions offers de-facto standard VIM northbound interfaces for easy MANO system integration, like our solution does with its OpenStack-like interfaces. Even if VLSP focuses on MANO-like experiments in the NFV domain, it lacks the ability to execute real-world VNF software, which is possible in our platform that uses lightweight container solutions to run VNFs in an emulated environment.

## 7.4. Emulation-based smoke testing

We introduce the approach of "emulation-based smoke testing" and discuss its benefits and limitations in Section 7.4.1. After that, we present our prototype in Section 7.4.2.

### 7.4.1. Approach

An emulated test infrastructure provides some major benefits when compared to a real NFV multi-PoP deployment, e.g., based on OpenStack. First, the state of emulated VIMs and NFVIs can be made volatile, which ensures that tests are always executed in a clean environment. For example, there are no zombie VMs left in a PoP resulting from a former test execution in which the used environment and infrastructure was not cleaned properly. Such a cleanup is complicated in real-world VIM systems which are designed to maintain their state, i.e., every action or configuration is not only applied to the specific subsystem, e.g., a VM is terminated, it is also stored in a persistent database. As a result, the system can get easily out of sync if things fail, e.g., the VM crashes but is not correctly removed from the database. As a result, the reinstallation or reinitialisation of the entire VIM system might become necessary to ensure a clean state for a new test. Second, the setup of an emulation platform can be expected to be much quicker and the needed resources are far less than for a full-featured VIM installation and the configuration of the attached compute, storage, and networking infrastructure. More importantly, an emulation platform can even be executed on a single machine (physical or VM), making it a much better fit for existing test pipelines,

e.g., based on Jenkins [Jen11]. It also allows parallelisation by using multiple VMs, each containing its own emulated NFVI deployment. Third, emulated infrastructure can be easily scaled to hundreds (or even thousands) of PoPs whereas a fully automated setup of hundreds of interconnected OpenStack installations is very challenging and may be even infeasible to realise in short timeframes, as we show in Section 7.5.1.

Figure 7.2 shows the proposed testing setup in which a test controller, e.g., Jenkins [Jen11] or a simple shell script, automatically sets up an environment that emulates a pre-defined multi-PoP topology (1). This setup can either be done on a physical machine or a VM, the so-called "test executor". Once this is done, the test controller configures the MANO system to be tested and connects it to the VIM interfaces of the emulated PoPs. In the figure, we use OSM as an example for a MANO system under test (SUT); we emphasise again that other MANOs can be used. After that, the test controller triggers the test cases against the MANO's northbound interface (2), e.g., deploying a test NS. To do so, either custom test suites or pre-defined standard-compliant test suites, e.g., our test suite for ETSI NFV's SOL005 [ETS18i] MANO northbound interface specification introduced in Section 7.4.2.2, may be used. Those tests should trigger the main functionalities of a MANO system, starting from VNF and NS on-boarding, followed by browsing the elements of a MANO's catalog, to the instantiation and scaling of a VNF or an NS. By doing so, the MANO system is tested end-to-end. Once a test NS is instantiated, the test controller checks if the resulting deployments and configurations on the emulated infrastructure, done by the MANO system during the tests, are correct (3). For example, it checks if the number of VNFs deployed on the PoPs is correct and if the intended configuration values have been applied to them. Once all tests are done, the test controller destroys the emulated infrastructure by stopping the emulation environment and freeing the test executor machine. It can then start a new emulation instance, e.g., with a different multi-PoP topology, for further tests.

As expected, there are also a couple of limitations when using an emulation-based infrastructure for testing. First, not all features of the original OpenStack APIs will be supported by the emulated infrastructure. This behaviour is intentional and helps to achieve the goal of a very lightweight substitution of a full-featured NFVI. In our prototype implementation, presented in the next sections, we focused on the API endpoints required to let typical MANO solutions, like OSM, believe that they talk to a real OpenStack installation, namely the OpenStack Keystone, Nova, Glance, and Neutron endpoints. Each of these OpenStack endpoints is required for NFV scenarios. More specifically, the Keystone endpoint is required to register the VIM to the MANO solution, i.e., perform the login procedures and generate API tokes. Further, it provides the MANO system with pointers to the other API endpoints, e.g., Nova. The Nova endpoint is the main endpoint to manage compute resources. It is used

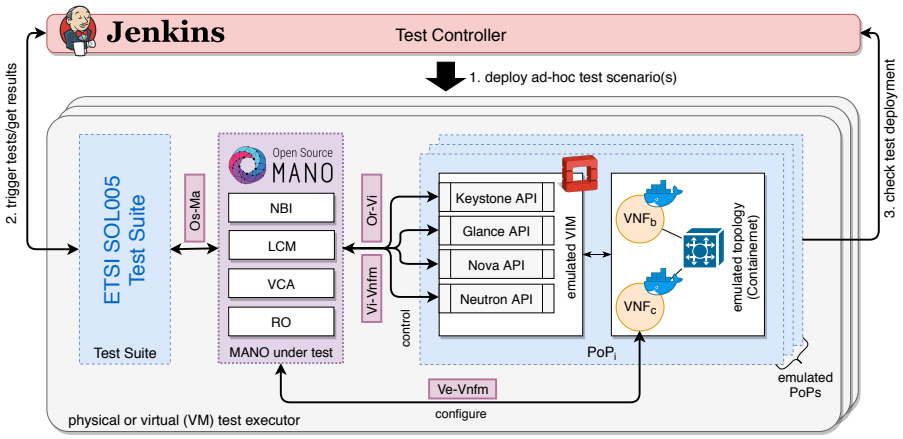



Figure 7.2.: An automated testing setup for a MANO system, using OSM as an example. The test controller automatically sets up the emulated infrastructure (multiple PoPs) in a test executor machine and tests the MANO system against this fresh infrastructure using a test suite, e.g., aligned to ETSI SOL005 (logos from [Doc13; Ope10b; ETS16c; Jen11]).

by the MANO system to deploy, configure, and terminate VNFs. The Glance endpoint is required to let a MANO system do basic image management, e.g., on-board disk images or list the available disk images. Without this endpoint, a MANO system would fail because it cannot validate if a required disk image is available in the emulated system. Finally, the Neutron endpoint is required to do the network setup between the VNFs including advanced network configurations for SFC. Other OpenStack endpoints, like the Cinder block storage endpoint, are not used by current MANO solutions and are thus not relevant for and not implemented by our test platform.

Second, our emulated infrastructure will not be able to deploy VNFs as full-blown VMs; instead it is limited to lightweight Docker containers. This means that it is not possible to use existing, VM-based VNF implementations as part of the tests. For example, its not possible to run a proprietary firewall VNF only available as Kenrel-based Virtual Machine (KVM) disk image inside our test environment. It also means that our platform does not support acceleration technologies often used by VNF VMs, e.g., SR-IOV. Nevertheless, these limitations are fine since our platform aims to test the basic MANO functionalities rather than concrete features of specific VNFs.

Third, the total available resources of the emulated infrastructure are limited. Even though the platform allows to emulate hundreds of PoPs as shown in Section 7.5.1, the resources of the host machine are still limited. To improve this and to be able to scale to even larger test setups, a distributed version of our platform could be implemented as future work (see Section 11.3).

These limitations must be kept in mind when using our emulation-based

smoke testing approach in a testing pipeline. In general, emulation-based smoke tests should not be considered as a full replacement of a final integration test against a real multi-PoP environment but as a much faster, intermediate testing stage that can easily be executed for each new commit to the MANO system's code base.

### 7.4.2. Prototype

We built a prototype of the described testing platform to validate our design and to check the feasibility of the proposed testing approaches. The core of our prototype is based on vim-emu [PKV16] and is described in Section 7.4.2.1. After extending the emulation platform to support large-scale MANO test scenarios with many emulated PoPs, we added a test suite for MANO systems to it (Section 7.4.2.2). Finally, we integrated the entire system with existing testing solutions to be able to automatically run them within CI pipelines as shown in Section 7.4.2.3.

#### 7.4.2.1. Multi-PoP emulation platform

The emulation platform consists of three main layers as shown in Figure 7.3. First, the network emulation layer, shown at the bottom of the figure, is based on Containernet [Peu16] (as described in Section 5.3) and allows to execute network functions inside Docker containers that are connected to arbitrary, user-defined network topologies [PKV16].

The *VIM emulation* layer of our platform, shown in the middle of Figure 7.3, creates an abstraction for the network emulation and allows a user to define arbitrary topologies with emulated NFVI PoPs. Each of these emulated NFVI PoPs represents a single VIM endpoint and allows to deploy, terminate, and configure VNFs executed inside the emulated PoP, as described in Section 5.4. This allows the emulation platform to emulate realistic, distributed NFVI deployments, e.g., by adding artificial delays to the links between the PoPs. We utilise this to allow the emulator to automatically load topologies from the ITZ library [Kni+11], as we show in Section 7.5.2. Once the container-based VNFs are deployed and running in the system, traffic can be steered through multiple VNFs by using the emulator's chaining functionalities as presented in Chapter 6.

The top layer of our emulation platform provides a set of APIs that mimic the original OpenStack APIs endpoints for each of the emulated PoPs and translate OpenStack requests, e.g., `openstack compute start`, into requests that are executed by the emulation platform, e.g., start a container-based VNF in one of the emulated PoPs. We mimic the OpenStack APIs because

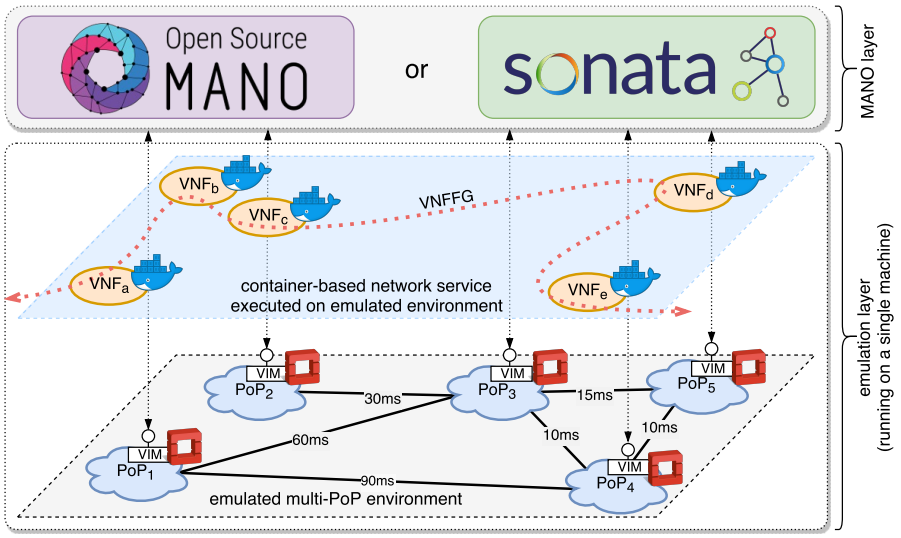



Figure 7.3.: A multi-PoP topology with five emulated OpenStack-like NFVIs running on a single physical machine (bottom) and five Docker-based VNFs running on the emulated infrastructure (middle), all controlled by a real-world MANO system (top) (logos from [Doc13; Ope10b; ETS16c; SON15b]).

OpenStack is currently the de-facto standard VIM and is supported by most MANO systems.

Figure 7.3 shows a usage scenario in which our emulation platform (bottom layer) emulates five interconnected PoPs, each offering its own OpenStack-like northbound APIs. This emulated infrastructure can be controlled by any real-world MANO system that is able to use OpenStack, e.g., OSM [ETS16c] or SONATA [SON15b] (top layer). The MANO system is used to instantiate a complex, distributed NSs, consisting of five VNFs, on top of the emulated infrastructure (middle layer). With this setup, the emulated infrastructure and the instantiated NSs look like a real-world multi-PoP NFVI deployment from the perspective of the MANO system consisting of multiple data centres deployed at different geographical locations.

#### 7.4.2.2. Standard-compliant MANO test suite

Once the emulated NFVI is up and running and the MANO system that is supposed to be tested is installed, running, and configured, everything is ready for test execution. The only missing piece in such a setup are the actual test cases as well as mechanisms to invoke the tested MANO system during the tests. One option to do this is to implement test cases that are custom-tailored to the MANO SUT. This approach makes a lot of sense if very specific aspects of a single MANO solution should be tested, e.g., a proprietary management interface. However, the goal of NFV is to establish an open environment

with well documented and standardised interfaces. An example for this is the *Os-Ma-Nfvo* reference point defined by ETSI [ETS14b] and its interface specification ETSI NFV-SOL005 [ETS18i].

Motivated by this, we started to design a standardised test suite for ETSI's *Os-Ma-Nfvo* reference point, implemented it as part of our prototype, and released it under an Apache 2.0 license [Peu18a]. To make this test suite as reusable as possible, we use a two-layered design. The top layer, which is based on Python's `unittest` library, implements the abstract test logic according to the written interface specifications of ETSI SOL005. Those tests then call the bottom layer of our test suite which contains plug-able connection adapters, abstracting MANO-specific connection details that are not part of the interface specification, e.g., authentication mechanisms. Our prototype comes with an example MANO adapter that supports OSM starting from rel. FOUR and uses OSM's client libraries to access OSM's northbound interface.

Table 7.1 presents an overview over the implemented tests. It shows different operations of the tested interfaces grouped by the resources they manipulate. The table also shows the availability of each interface endpoint in ETSI SOL005 and its implementation status in OSM rel. FOUR. Some endpoints, e.g., the endpoints to manipulate the VIMs that are connected to a MANO system, are only available in OSM's interface but not in ETSI's specification. Those differences to the written specification usually originate from additional requirements of practical implementations. Other endpoints, e.g., NSs healing, are defined by ETSI but are not yet available in OSM. To keep the table short, it does not show the *NSs performance and fault management* interfaces defined by ETSI, since they are not yet available in OSM. Finally, the table presents mean runtimes of each test, as we further describe in Section 7.5.2.3.

#### 7.4.2.3. CI pipeline integration

One of the key points in modern software testing is automation. Today, most software projects, including MANO system projects, use CI approaches to automatically execute tests whenever a developer commits new code to the code base. Those tests are organised in so-called test pipelines that start with static code style checks, continue with detailed unit tests, and end with basic integration tests between the project's components. Once all these tests passed, the resulting software artefacts have to be tested in more complex environments to check their compatibility with external components, e.g., different VIM solutions, to find integration issues.

The main problem of those complex tests is the required test infrastructure, e.g., to setup multiple OpenStack-based VIMs and to maintain them. Another problem with those tests is their scaleability: Even if some lab-scale OpenStack installations are available, they can only be used to execute a limited number

Table 7.1.: Mapping between interfaces specified/implemented by ETSI SOL005 and OSM rel. FOUR and their coverage in the presented test suite. The table also shows the mean runtime of each test.

| | ETSI | OSM | Tests Suite | Mean Runtime |
|---|---|---|---|---|
| Resource: **VIMs** | | | | |
| Create | ○ | ● | ● | $1.31s \pm 0.19s$ |
| List | ○ | ● | ● | $2.58s \pm 0.20s$ |
| Resource: **Individual VIM** | | | | |
| Show | ○ | ● | ● | $1.26s \pm 0.23s$ |
| Update | ○ | ● | ○ | |
| Delete | ○ | ● | ● | $1.16s \pm 0.21s$ |
| Resource: **NSDs** | | | | |
| Create | ● | ● | ● | $0.52s \pm 0.05s$ |
| List | ● | ● | ● | $0.55s \pm 0.03s$ |
| Resource: **Individual NSD** | | | | |
| Show | ● | ● | ● | $0.54s \pm 0.06s$ |
| Update | ● | ● | ○ | |
| Delete | ● | ● | ● | $0.53s \pm 0.07s$ |
| Resource: **VNFDs** | | | | |
| Create | ● | ● | ● | $0.24s \pm 0.04s$ |
| List | ● | ● | ● | $0.40s \pm 0.05s$ |
| Resource: **Individual VNFD** | | | | |
| Show | ● | ● | ● | $0.24s \pm 0.04s$ |
| Update | ● | ● | ○ | |
| Delete | ● | ● | ● | $0.23s \pm 0.04s$ |
| Resource: **NS instances** | | | | |
| Create | ● | ● | ● | $9.35s \pm 0.32s$ |
| List | ● | ● | ● | $9.56s \pm 0.27s$ |
| Resource: **Indiv. NS instance** | | | | |
| Show | ● | ● | ● | $9.44s \pm 0.27s$ |
| Update | ● | ○ | ○ | |
| Scale | ● | ● | ○ | |
| Create Alarm | ○ | ● | ○ | |
| Export Metric | ○ | ● | ○ | |
| Heal | ● | ○ | ○ | |
| Terminate | ● | ● | ● | $9.44s \pm 0.27s$ |
| Resource: **VNF instances** | | | | |
| List | ○ | ● | ● | $9.81s \pm 0.30s$ |
| Resource: **Indiv. VNF instance** | | | | |
| Show | ○ | ● | ● | $9.67s \pm 0.26s$ |

of test cases at a time. They can easily become a bottleneck as the number of developers and thus the number of contributions increases. A common solution for this is to reduce the frequency of complex tests by not executing them for each new commit, but only once a day. At this point, our emulation-based smoke testing solution can help and improve the test workflow because it can be used as an intermediate test stage between frequent basic tests and complex integration tests in real environments.

More specifically, our emulation-based solution provides some characteristics which make it a perfect fit for a frequent execution in CI pipelines. First, the entire platform can be started and configured with a single command. Second, our platform always starts in a clean state. There is no need to manually cleanup the environment after a test has been executed. Third, the emulator can be packaged and executed within a container (nested Docker deployment) or a VM which make distribution, initial setup, and integration with existing test environments easy. It also allows highly parallelised test setups because multiple VMs, each running one emulation platform instance, can be deployed on an existing test infrastructure and used completely independently from each other. This feature should be particularly helpful for multi-branch test pipelines. Finally, the resource footprint of the emulation platform is very small and it can be (re-)started within seconds (or minutes if hundreds of PoPs should be emulated) as we show in the following sections.

## 7.5. Results

The evaluation of the proposed smoke testing approaches and our platform prototype can be split into two parts. First, we evaluate the scaleability of our emulation platform in Section 7.5.1 using the same approach as in [Peu+18b] but using scenarios ten times larger to push the platform to its limits. Second, we conduct a case study using OSM as a state-of-the-art MANO solution and test it against our platform using real-world topologies in Section 7.5.2. In this case study, we not only test two major releases of OSM, namely OSM rel. THREE and OSM rel. FOUR, and compare them, but also analyse the runtimes of our ETSI-compliant test suite executed against OSM rel. FOUR.

### 7.5.1. Emulation platform scalability

To get a first idea about the setup time savings that can be expected from emulated PoPs, we compare the setup times of our emulation platform configured to emulate a single OpenStack-like PoP with the setup times of a single-node OpenStack DevStack [Ope10a] installation, which can be considered as the most simple way to install a fully-featured OpenStack in a PoP. We execute

both setup procedures 10 times on a single physical machine with Intel(R) Core(TM) i5-4690 CPU @ 3.50 GHz as well as 16 GB memory and find a mean setup time for a single emulated PoP of 2.48 s compared to a mean setup time of 576.42 s for a fresh DevStack installation, which is more than 232 times slower. This comparison makes sense since we want to ensure that we always test against a clean environment and thus a fresh installation of DevStack would always be required.

To quantify the scaling abilities of our emulation platform, we run a set of experiments to study its behaviour when topologies with many PoPs are emulated or when hundreds of NS instances are deployed on the emulated infrastructure. This experiment and all following experiments are executed on a single physical machine with Intel(R) Xeon(TM) E5-1660 v3 CPU with 8 cores @ 3.0 GHz as well as 32 GB memory and are repeated 10 times. All error bars in this chapter show 95 % confidence intervals. In the first experiment, we analyse the startup and configuration time of the emulation platform for different synthetic topologies with different numbers of PoPs. Figure 7.4 shows the setup time breakdown for up to 1024 PoPs using four topologies. It shows how much time is used by which of the four phases of the emulation setup procedure: *Initialisation*, *PoP setup*, *link setup*, and *emulation start*. The *linear* topology connects all PoPs into a long chain and the *star* topology connects all PoPs to a single central PoP. The two randomised (*rnd*) topologies get the number of PoPs $|V|$ and a factor $k$ as inputs and interconnect the PoPs with $|E| = \lfloor k|V| \rfloor$ links where $|E|$ is the number of created links. This is done by considering the set of all possible links between all involved PoPs (i.e. a full mesh) and selecting a subset of $|E|$ links to be used. The selection is done by random sampling using Python's `random.sample()` function and results in disconnected topologies, in which not every PoP can reach every other PoP, if $k < 1.0$. This is intended since we are interested in measuring the platform's resource footprint for various topology setups, including those disconnected topologies, and not in testing the interconnection between the PoPs.

The results show that in all topologies 128 PoPs can be set up in between 91.8 s and 197.7 s, which is a huge improvement when compared to 128 DevStack installations. Even the maximum tested number of 1024 PoPs can, on average, be created in 3,704.0 s using the *rdm(k=0.5)* topology. The results of the randomised topologies indicate that the number of links which have to be established in the topology has a non-negligible impact on the overall setup time. Further, the plots indicate a non-linear relationship between number of PoPs and total setup times. We have identified the OVS daemon (`ovs-vswitchd`) with its single-threaded design as the root cause of this. The OVS daemon becomes the bottleneck in large deployments as it has to manage one OVS instance per PoP in the deployment.

We also analyse the memory consumption for these four topologies and directly compare their total setup times (Figure 7.5). The figure shows that the

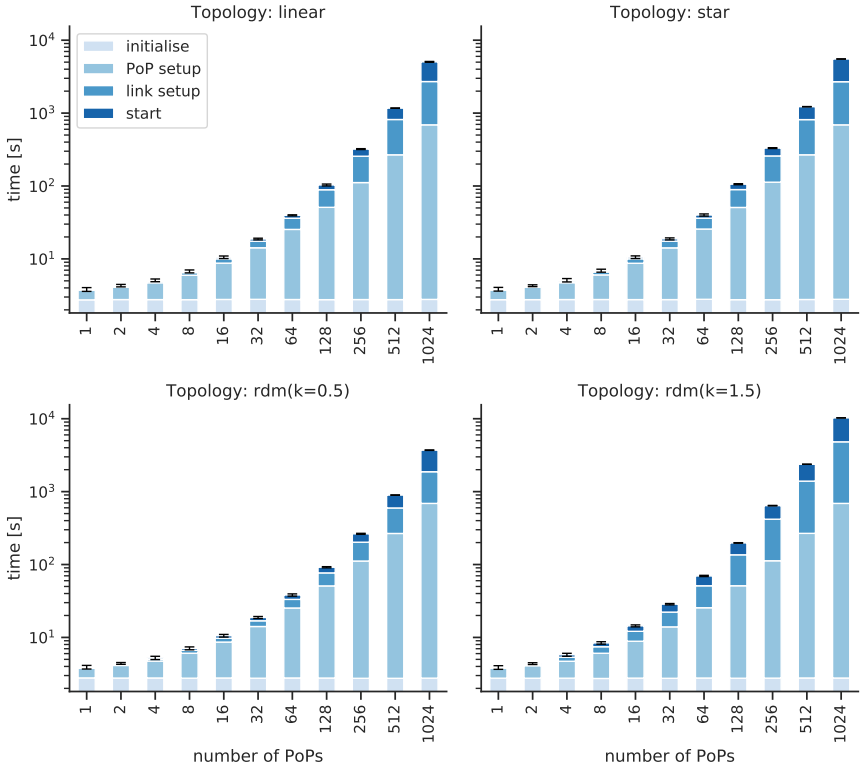



Figure 7.4.: Breakdown of the emulator setup times into four phases using four different topologies

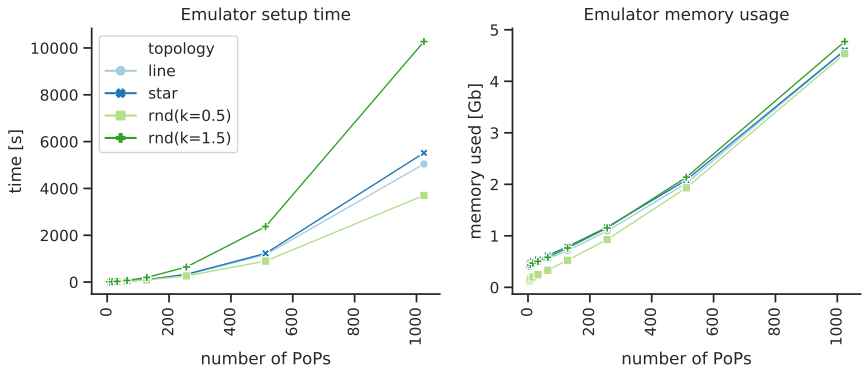



Figure 7.5.: Emulator setup times and memory usage

total memory used by the tested environment increases proportionally to the number of PoPs in the topology. In general, not more than 5 Gb of memory is used, even with large topologies, which shows that our emulation platform can easily be executed on existing test nodes or locally on a developer's laptop.

Finally, we study the time required to deploy a large number of VNFs on top of the emulated infrastructure. We again use our *liner*, *star*, *rdm(k=0.5)*, and *rdm(k=1.5)* topologies with either 8 or 128 PoPs and deploy up to 256 VNFs on those PoPs (VNFs are randomly placed). The used VNFs are based on the default Docker `ubuntu:trusty` image and do not run any additional software, since we are only interested in the bare instantiation times. Figure 7.6 shows that the instantiation times scale proportionally with the number of VNFs and are also influenced by the number of links in a topology, i.e., more links slow down the system. Please note that most error bars are hidden behind the markers of the plots. It can be seen that with our platform hundreds of VNFs can be quickly deployed on a single machine, enabling fast tests of large deployment scenarios.

### 7.5.2. Case study: OSM rel. THREE vs. OSM rel. FOUR

In our case study, we have decided to compare OSM rel. THREE and OSM rel. FOUR [ETS16c] because OSM rel. FOUR is the first release following OSM's new micro service architecture with a central message bus and implements ETSI's SOL005 interfaces. OSM rel. THREE, in contrast, has a more monolithic design with three or four large components using fixed APIs for communication. Besides the improved flexibility, the design of OSM rel. FOUR also promises better scalability and performance, which we validate with our experiments.

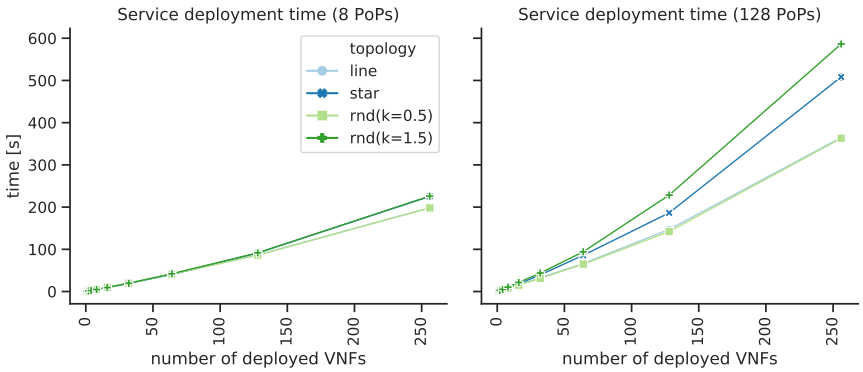



Figure 7.6.: NS instantiation times on a small and a large topology using NSs with up to 256 VNFs.

The setup for the study is the same as described in Figure 7.2 but we use a scripted test controller that automatically performs a series of experiments and collects additional data. Besides the general functionality of the VIM attachment procedure, we investigate the behaviour of OSM when it has to interact with large multi-PoP deployments and a high number of instantiated NSs. To be more realistic, we use a set of real-world topologies with different sizes that are taken from the ITZ library [Kni+11]. In our case study, each node of a given topology is turned into a single PoP emulating an OpenStack VIM, resulting in topologies with 4 to 158 PoPs. The delays between the PoPs are calculated based on the geolocations provided by the ITZ dataset. These are test cases which are not covered by existing NFV testbed installations that usually only use a single PoP installation.

#### 7.5.2.1. OSM in large multi-PoP environments

In the first set of experiments, we analyse the *VIM attach* procedure, which is used to connect OSM to a single PoP using the `osm vim-create` command. Figure 7.7 shows the total setup time breakdown to start the emulated infrastructure and to attach all emulated VIMs to OSM. The numbers behind the topology names indicate the number of nodes and links in the topology. The results show that the time required to attach the VIMs to OSM uses most of the test environment's setup time, but the system can still be deployed and configured in between 200 s and 330 s, even if the largest topology with more than 150 PoPs is used. The figure also shows the request times for all `osm vim-create` requests. It indicates that the attachment procedure becomes slightly slower when larger topologies are used. Comparing the results between the two OSM releases, OSM rel. FOUR shows improved setup times and reduced request times to attach the VIMs. It can also be seen that the setup

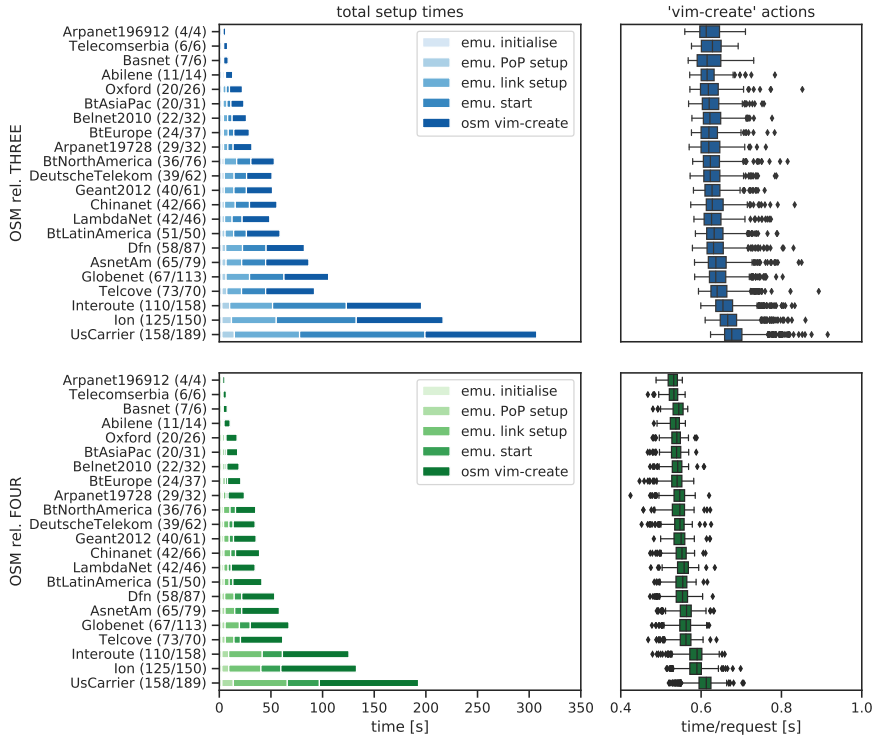



Figure 7.7.: OSM and emulator setup times with real-world topologies

times of the emulation platform are smaller in the OSM rel. FOUR case. The reason of this is the significantly smaller resource footprint of OSM rel. FOUR, which is executed on the same physical machine as the emulation platform.

#### 7.5.2.2. OSM service instantiation and termination

In the second set of experiments, we investigate OSM's NSs management behaviour. More specifically, we test the NSs instantiation (`osm ns-create`), NSs termination (`osm ns-delete`), and NSs show (`osm ns-show`) operations. To do so, we use a test NS consisting of two linked VNFs. We request OSM to sequentially create 64 instances of this NS, which corresponds to 128 deployed VNFs. Later, these NSs are terminated one after each other. In each instantiation request, the NS is randomly placed on the available PoPs of the three used topologies (Figure 7.8). The given instantiation and termination times represent the time until the requested containers (the VNFs of the NS) are started or stopped, not only the raw API response times.

The results show that an NS instantiation takes between 7 s and 12 s in most of the cases if OSM rel. FOUR is used. OSM rel. THREE, in contrast, shows

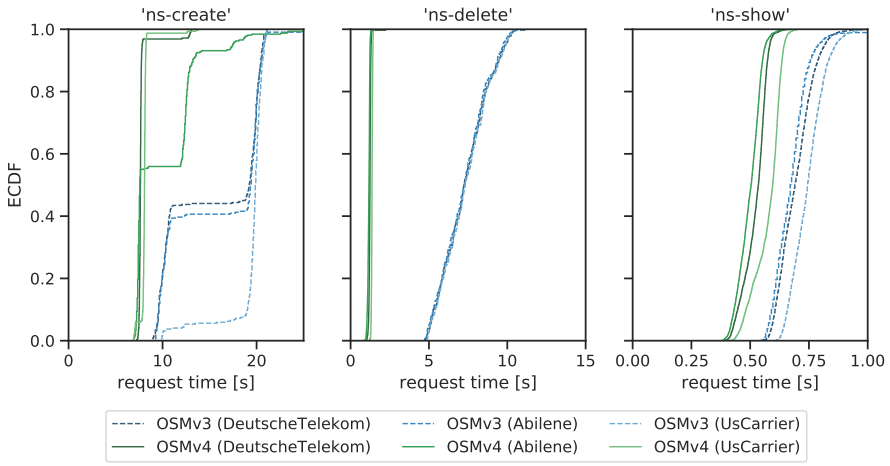



Figure 7.8.: OSM service management interfaces request time analysis

instantiation times between 10 s and 20 s. The results also show that the instantiation times in OSM rel. FOUR are more stable. The increased instantiation times shown by OSM rel. FOUR, when the small Abilene (11 PoPs) topology is used, are caused by the fact that more NSs are instantiated per emulated PoP. The analysis of NSs termination operations, presented in the middle of Figure 7.8, clearly shows that NS termination is much faster and shows smaller variance in OSM rel. FOUR compared to OSM rel. THREE. NS termination times also have only very small dependencies on the used topologies. Further, the request times to fetch details of a running NSs instance have been improved in OSM rel. FOUR as shown in the right part of the figure. In general, `osm ns-show` requests are much faster than the other operations, since nothing in the actual NS deployment is changed during a request.

This test validates the design choices made in OSM rel. FOUR and shows that they improve the overall system performance. Those large-scale test cases would not have been feasible without our presented testing platform and clearly show its usefulness for the NFV community and how it can support future 5G developments.

#### 7.5.2.3. ETSI-compliant test suite

Using our ETSI SOL005-compliant test suite, presented in Section 7.4.2.2, we record the request times for more endpoints of OSM's northbound interface. Table 7.1 shows the mean request times and 95 % confidence intervals among 10 runs of the test suite against OSM rel. FOUR using an emulated PoP as a single connected VIM. The results show that the request times are all very stable and the use of our emulation platform allows to execute a complete test run in about 67.03 s, which is a result of the fast instantiation times of VNFs

and NSs. Similar test runs in cloud testbeds will take substantially longer. We do not present results for OSM rel. THREE, because of its missing ETSI SOL005 support and the resulting incompatibility with the test suite.

## 7.6. Discussion

The results of our case study show the evolution of OSM and how its performance has improved in rel. FOUR. Especially the reduced resource requirements contribute to a better performance when used with many PoPs.

During our case study, we have found and reported some interesting issues, for example, a bug in OSM rel. THREE that prevents a user to instantiate an NS on the 101st or higher-numbered PoP. The reason for this is a hard-coded query limit that causes the OSM client to only fetch the first 100 PoPs that are attached to the system. This results in a "PoP not found" exception when an NS should be instantiated on, e.g., PoP 101. Based on our feedback, this issue is fixed in OSM rel. FOUR. We have also noticed that for every `osm vim-show <pop x>` command the entire VIM list is fetched by the OSM client, instead of only fetching the information of the requested PoP. This increases request delays when OSM is used with many attached PoPs. An observation that we have also reported to the OSM community and has lead to further improvements of the code base.

It is important to note that such issues would not have been discovered by today's NFV test deployments which usually do not use more than a handful of PoPs. But the 5G and NFV community envisions very large multi-PoP scenarios for future use cases, like IoT. As a result, MANO systems need to be tested against such large multi-PoP networks. To do this, our platform provides a flexible and easy to apply test solution that allows to verify and improve the quality of MANO systems for use cases of future networks.

The presented approach and platform is able to test other MANO solutions. On candidate is ONAP [Lin18a] which is the second "big" player in the open-source MANO landscape. However, the majority of the available codebase of ONAP is, at the time of writing[1], not in a state to perform those experiments, e.g., because of API limitations. Other reasons are the lack of automated installation procedures and the very high resource requirements of ONAP. But we are confident that this will change in the next two or three release cycles.

Another improvement we have recently integrated into the presented platform is support for VNF and NS configuration mechanisms, like Juju Charms [Can12]. This allows VNF and NS developers to use our platform to perform complex integration tests between their developed products and the MANO systems

[1] June 2018, ONAP version 2.0 (Beijing)

while having the benefits of a lightweight test platform that can be deployed locally.

## 7.7. Conclusions

Using emulation-based smoke testing as part of the automated test and integration pipeline, used by MANO software projects, contributes to the quality and production readiness of these complex software systems. The presented approach enables automated testing of MANO systems in large-scale 5G scenarios with hundreds or thousands of PoPs. This is not possible with today's lab-scale NFV testbed installations.

Our case study shows how our presented approaches are used to find bugs and to reveal the performance improvements between two major releases of OSM, one of the most prominent open-source MANO solutions, today.

# Part III.

# Performance benchmarking

# 8. Automated benchmarking for NFV

In this chapter, I present the concept of automated benchmarking in agile NFV environments to gain insights about the performance of VNFs and NSs prior to their production deployment. The chapter is based on my papers [PK16b] and [PK17] and contains figures and verbatim copies of the text from these papers. The chapter presents a benchmarking platform that is published as open-source project [Peu18c]. After motivating the need for benchmarking concepts in NFV, I discuss their integration into the NFV DevOps cycle in Section 8.1.1 and present the resulting challenges and research questions in Section 8.1.2, followed by an overview of related work in Section 8.2. The architecture and concepts of the presented benchmarking platform are described in Section 8.3. The resulting prototype is then used to perform a case study, presented in Section 8.4, which answers the questions if VNFs should only be benchmarked in isolation or within the NS they are part of. Finally, Section 8.5 concludes.

## 8.1. Introduction

The softwarisation of network functions comes with many benefits but also introduces new challenges. One of them is the enforcement of service level agreements (SLAs) in these dynamic software-based environments. This question is aggravated by VNFs being deployed as part of complex NSs, containing multiple VNFs potentially distributed across multiple PoPs. Since these chains of VNFs (or SFCs) are often deployed between the end users and third-party services, enforcing the QoS of the entire SFC is crucial to meet user expectations. MANO systems play a key role in such scenarios where they are responsible to decide how many resources are allocated for each VNF to meet the aforementioned goals—a process called "resource dimensioning". This process does not only happen during the initial deployment of a VNF but also as part of automated LCM operations such as scaling or healing; this means that resource dimensioning itself needs to be highly automated.

Existing approaches for these problems rely on live-monitoring solutions. In such systems, performance data is continuously collected and the configurations of the deployed VNFs are adapted to meet the SLAs. However, this has the downside that it is not possible to make statements about the expected performance and resource requirements of a VNF prior to its deployment. It

also makes the consequences of LCM operations difficult to foresee since the MANO system has no concrete knowledge about the VNF's behaviour under changed configurations, e.g., increased resource allocations. This becomes especially important in very agile DevOps environments in which new versions of VNFs and NSs are directly deployed into production. In such scenarios, up-to-date monitoring data of the new artefacts is not available and can lead to wrong orchestration decisions and degraded performance, as we further detail in Section 8.1.1. Consequently, the NFV and research community started looking for benchmarking solutions which can give insights about the VNF and NS performance that can be expected for a given configuration [Mor17; RRS15; Cao+15]. However, those initial solutions are either tied to specific platforms on which the VNFs are benchmarked, or they require manual steps to setup and run benchmarking experiments. This complicates the integration of the benchmarking procedure into automated DevOps workflows.

Another important aspect for benchmarking in an NFV context is the need to consider complex SFCs and not only single, isolated VNFs. This is because the end-to-end performance of the entire SFC is the metric of interest, especially if the SFC is deployed on the path between end users and a customer service, e.g., between user and a content delivery network (CDN). One option to get the end-to-end SFC performance is to combine benchmarking results of single VNFs using a performance model of the SFC, e.g., using model-based performance prediction approaches known from the software engineering community [BKR07]. The problem of this approach is that SFCs can consist of many VNFs that could possibly be chained in different ways, like different order of functions, different structures (branches), and different forwarding paths, which makes it hard to find correct models. More importantly, the required details about the SFC structure might not even be available, e.g., for proprietary (black-box) SFCs. This means that model-based approaches highly depend on this expert knowledge of VNF and SFC developers who have to manually provide the correct models—a clear contradiction to the idea of automation as we further detail in Section 8.2. An alternative option is to develop benchmarking solutions that are able to benchmark the actual implementations of the SFCs end-to-end, using an experiment-based approach, as we do in this chapter.

More specifically, we analyse and discuss the missing components to automatically collect, process, and use benchmarking data to improve NFV deployments and automate resource dimensioning decisions. We, in particular, focus on DevOps concepts and their interactions between service development and operation. After discussing the need for benchmarking solutions in NFV DevOps scenarios, we identify the main challenges and research questions in this domain. We then present a novel NFV benchmarking solution that addresses the shortcomings of existing approaches: Platform dependency, lack of automation, and missing support for SFC benchmarking. The presented

solution is designed to be completely platform-agnostic using a descriptor-based experiment generation approach. We use the presented solution to show that naive approaches which benchmark single VNFs in isolation and combine their results do not work well and can result in inaccurate predictions for the resulting SFC performance.

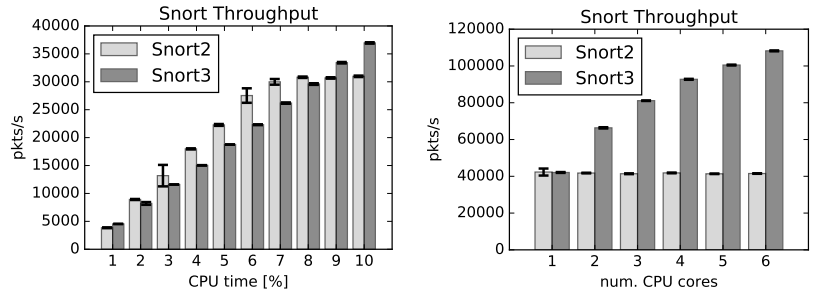
### 8.1.1. Benchmarking as part of the NFV DevOps cycle

The main idea behind the DevOps concept is that the team (or individuals) that build a software artefact are also responsible to operate it [BWZ15]. It bridges the gap between development of software artefacts and the operation of the resulting services [Kim+15]. Updated artefacts, e.g., a new VNF version or a redesigned SFC, are directly deployed into production after they have been quickly tested by an automated CI pipeline. In addition, feedback collected during operation is used for development, e.g., to optimise the performance of the created artefacts.

As a result, manual tests must be removed from the development cycle, in order to apply DevOps in the NFV domain. This becomes challenging for NFV where services are always expected to meet certain SLAs. On one hand, it becomes hard for service developers to validate that changes do not have any negative impact on the resulting performance before they put their artefacts to production. On the other hand, MANO systems will be continuously faced with the management of new artefact versions, which means that resource dimensioning algorithms, e.g., scaling algorithms, have to be continuously adapted to the behaviour of the managed artefacts. This can be tricky because historical monitoring information, available from old artefact versions, might not provide correct assumptions about the new version. For example, assume a developer fixes a performance bug in an IDS VNF that reduces its resource requirements. A MANO system will not know about this and it will allocate too much resources to the new IDS instance.

We verify these assumptions with a set of experiments in which we test the performance of commonly used VNFs under different resource configurations [PK16b]. Figure 8.1 shows the results of those experiments, which are executed on a machine with Intel(R) Core(TM) i7-960 CPU@3.20 GHz, 4 physical cores, hyper threading, and 24 GB memory. Each experiment is repeated 25 times and the error bars indicate 95 % confidence intervals.

Figure 8.1 shows the performance of two different Snort IDS [Cis16] versions for different CPU configurations. Figure 8.1a shows their behaviour under limited CPU time allocations ($\leq 10\,\%$) on a single core under which both Snort versions behave almost identical. In contrast to this, Snort 3 outperforms the old Snort 2 version and shows a completely different scaling behaviour when the number of available CPU cores is increased (Figure 8.1b). The obvious reason




(a) Limited CPU time (b) Number of CPU cores

Figure 8.1.: Comparison of two major versions of the Snort IDS system under different CPU configurations

for this is the fact that Snort 3 introduces multithreading. A MANO system cannot know about such changes and either needs to rely on expert knowledge or on benchmarking data, similar to Figure 8.1, to support its orchestration decisions. In our work, we focus on collecting such benchmarking data prior to the production deployment of the benchmarked components. An alternative to this is to modify VNF and NS configurations of the production system itself and learn from the monitored changes. The clear downside of this is that such modifications could cause performance degradations of the production system and it would barely be possible to test border cases, e.g., configurations with very limited resources, in a production system.

Another important point that motivates the need for pre-deployment benchmarking is based on the assumption that low-level metrics, like throughput, are often not sufficient to perform good resource allocation decisions. Especially for QoS optimisations, application-level metrics, like frames/s of a video stream, are more interesting. However, due to encryption and privacy issues, it is not always possible to collect such metrics from operational services, e.g., no DPI mechanisms are available. In an offline benchmarking solution, in contrast, developers are able to collect many more performance metrics since the entire test setup is under their control. It is, for example, possible to add additional measurement VNFs, called "probes", to the benchmarked SFC; these probes can tap to intermediate connections within an SFC to collect statistics or generate specific traffic patterns to tigger specific actions of an SFC, e.g., malicious traffic to test the rules of an IDS VNF.

To collect this benchmarking data, a mechanism is needed that automatically gathers performance information of single VNFs and/or full SFCs prior to their production deployment. This can be done by iteratively deploying the benchmarked VNFs or SFCs, which are also called SUT in this context, under different configurations, e.g., resource allocations or software configurations, stimulating them and collecting metrics that describe their resulting perfor-

mance. This mechanism, which we call "offline benchmarking", has to be tightly integrated into the NFV DevOps cycle and must be fully automated so that it can be executed, e.g., whenever the code of a VNF changes or during the on-boarding procedure of a VNF or NS to an NFV platform.

### 8.1.2. Challenges and research questions

Introducing benchmarking into the NFV ecosystem and enabling it to be part of the DevOps cycle raises several research questions that we outline in this section. These questions will then be discussed in the remainder of this thesis.

Q1: *How to automatically benchmark VNFs prior to their deployment to obtain insights into their performance behaviour under different configurations?*
VNFs are complex, multi-layer software systems and predicting their resulting performance for a given configuration is hard. To overcome this, benchmarking mechanisms that actually execute VNFs to gather insights about their resulting performance can be used. Such benchmarking approaches need to be fully automated to seamlessly integrate them into the envisioned DevOps-based NFV workflows. This challenge is addressed in the reminder of this chapter.

Q2: *How to design an automated benchmarking solution that provides a high degree of flexibility?*
VNFs can be benchmarked using black-box, grey-box, and/or white-box approaches. Benchmarking processes have to consider different metrics, starting from low-level system metrics up to application-level metrics, and need to be executed on different NFV platforms. This requires a highly flexible design of the benchmarking solution. This challenge is addressed in the reminder of this chapter.

Q3: *Is it sufficient to benchmark VNFs in isolation and infer the performance of an SFC composed of those VNFs? Or should SFCs always be benchmarked end-to-end?*
Benchmarking single VNFs will already increase the knowledge about the runtime behaviour of an SFC composed of these VNFs. But such VNF-specific results might not be able to capture all runtime dynamics of complex SFCs. To identify bottlenecks and relationships between resources assigned to different parts of the SFC, it might be required to benchmark the entire chain end-to-end, as we show in Section 8.4.1.

Q4: *How to deal with large configuration spaces? Is it possible to to benchmark only a subset of VNF and SFC configurations and predict the missing results to speed up the benchmarking process?*
VNFs and SFCs can have many configuration parameters resulting in large configuration spaces. An automated benchmarking process might

not be able to efficiently explore those configuration spaces or the resulting benchmarking process might simply take too long, e.g., if it is applied during the on-boarding phase. Benchmarking only a subset of configurations and interpolating the missing results might be a solution for this. This is addressed in Chapter 9.

Q5: *How to represent and share benchmarking results so that they can easily be picked up by others?*
The results of benchmarking processes, also called NFV-PPs, need to be efficiently represented and stored so that they can be easily reused and shared. Besides storing the raw results, performance profiles might be represented by their statistical properties or using machine-learning models. Initial solutions for this are presented in Chapter 10.

Q6: *How can next-generation MANO systems utilise the resulting performance profiles?*
Existing MANO systems base most of their resource dimensioning, scaling, and placement decisions on monitoring data collected at runtime. Having detailed performance profiles available as input to these systems, before the service is deployed, or before it has to be scaled up or down, will help to optimise their decisions. To do so, several extensions like standardised interfaces and updated control loops are required. We have presented initial work on this topic in a collaborative paper [Drä+18]. Further optimisation solutions that assume to have such profiles available are presented by my colleagues in [DKM18; DSK18]. To complement this work, Chapter 10 presents a solution to automatically turn raw NFV-PPs into piecewise constant or linear functions that can directly be used as inputs for the mentioned optimisation solutions.

## 8.2. Related work

In the software engineering community, model-based performance prediction solutions for composed software exist [BKR07; BKR09]. They rely on abstract component models that can be used to simulate the resulting performance of complex, composed systems. They aim to evaluate architectural design decisions [BKR07], e.g., differences between micro service-based architectures. Such a model-based solution is, in principle, an interesting approach to evaluate the performance behaviour of a complex SFC which is nothing more than a composed service. In the DevOps scenarios, discussed in this thesis, the components (the VNFs) are, however, often considered to be black boxes that are implemented by a third party. This means that implementation details of these components might not be available making it impossible to build such simulation models. Further, the need of manual modelling of the components does not allow for full end-to-end automation of the benchmarking process,

e.g., executed during the on-boarding process. As a result, model-based approaches from the software engineering community can only be considered as an additional tool for the design (pre-development) phase of VNFs and SFCs, which is out of scope of this thesis. They cannot be considered as solution to automatically collect performance data of already implemented, third-party, or black-box VNFs and SFC that are about to be deployed.

A lot of work about benchmarking of virtualised applications has already be done by the cloud computing community, proposing a couple of solutions to benchmark single cloud applications [Woo+08; TZK16; TZK17] and some solutions to benchmark composed applications [Tak+13; Gia+15]. Especially [TZK16; TZK17] is comparable to the benchmarking approach presented in this thesis [PK16b]. Similar to our approach, the authors use resource limiting features of a hypervisor to test workloads under different configurations. Their goal is, however, to quantify the sensitivity of a VM to shared cloud resources for a given workload rather than to derive full performance profiles of a given application. They do this by closely monitoring the hypervisor metrics during application execution, which is different to our approach that focuses mainly on the input/output and application-level metrics of VNFs and SFCs, e.g., achieved throughput or observed delays. Their solution focuses on scenarios with a single VM and cannot be used to benchmark SFCs. The other solutions [Woo+08; Tak+13; Gia+15] can also not directly be applied to NFV scenarios due to different application modelling approaches. For example, none of them allows to directly deploy a VNF or SFC specified with ETSI-aligned descriptors as our solution does. Further, they lack support for SFC scenarios which limits their usefulness for NFV use cases.

Benchmarking in NFV use cases is already considered by standardisation bodies, like IETF [Mor17] and ETSI [ETS16b], but the availability of real-world solutions to perform such benchmarks is still limited. Most existing solutions have emerged in parallel with the solutions presented in this thesis [RRS15; Cao+15; BS15; Kha+18; RBR17]. The first approach is called "VNF benchmarking as a service (VBaaS)" [RRS15] and proposes a framework to benchmark NFV infrastructure as well as single VNFs, but lacks support to benchmark complex SFCs. Another approach is called "NFV-VITAL" [Cao+15] and introduces a VNF characterisation framework based on an orchestrator component that allows a user to automatically benchmark SFCs. This approach is close to our solution but it is limited to services described by HEAT templates [Ope10d], which offer only limited chaining support. In contrast, the work presented in [BS15] provides a theoretical model to estimate VNF performance. This model does not consider SFCs and requires detailed knowledge about elementary operations performed inside VNFs, which is not necessarily available, e.g., for proprietary VNFs. None of the presented solutions focuses on the impact of SFC reconfiguration, e.g., reordering, as it is done in our case study. The authors of [Kha+18] present a platform called "NFV Inspector" that

mainly focusses on providing a systematic approach for VNF classifications. It is complementary to our work. A highly automated DevOps environment is not explicitly considered by any of these solutions. NFV-VITAL [Cao+15] provides some degree of automation but with limited flexibility compared to our experiment description and configuration approach.

Finally, a solution called "Gym" was initially presented in [RBR17] and provides a solution for end-to-end automation of VNF benchmarking experiments. Gym, however, focuses on single and composed VNFs and does not have built-in support to profile SFCs. Gym was developed in parallel with our solution and uses, in its latest version, our NFV prototyping platform vim-emu, presented in Part II of this thesis, as its execution platform. At some point, we joined forces with the authors of Gym and started to work on a joint IETF draft on VNF benchmarking automation [Ros+18] for which Gym as well as the solutions presented in this thesis act as official reference implementations.

## 8.3. Automated performance benchmarking of NFV functions and services

The first two questions (Q1/Q2) presented in Section 8.1.2 highlight the need of a flexible, fully-automated NFV benchmarking framework, which we address in this section. To design this framework, we identified the following requirements. (R1) Automation: Allow fully automated (i.e. scriptable) benchmarking experiments without any human interaction after the experiment's initial description. (R2) Flexibility: Support many NFV platforms so that benchmarking experiments can be executed in different environments. This includes support for different control and monitoring interfaces as well as different description languages. (R3) Abstraction: Once a developer has described a benchmarking experiment it should be possible to execute this experiment on different target NFV platforms. (R4) Integration: The benchmarking tool as such has to be integrable into different workflows, e.g., to be executed inside a CI/CD pipeline or to become part of an NFV platform's on-boarding procedure.

Based on those requirements, a benchmarking framework can be split into three areas. First, a description approach is needed that defines the benchmarking procedures, i.e., defines how a benchmarking experiment should be conducted. Second, an NFV platform with its infrastructure and MANO facilities is used to deploy, configure, and execute the SUT, which can be either a single VNF or a complex SFC. Third, a benchmarking controller is needed that coordinates the benchmarking process and controls the test system by interfacing with the NFV platform. The following sections describe the design and implementation

of our benchmarking framework, called "tng-bench", which fulfils the above requirements and covers all three areas.

### 8.3.1. Benchmarking platform design and workflow

We designed the core of our benchmarking framework, containing the benchmarking controller, as a highly modularised system that allows to replace many of the components that interface with external systems so that it can be extended and used with any NFV platform. The key idea of this design is to utilise existing VNF and NS description mechanisms used by MANO systems to control SUT deployments and benchmark a SUT with different parameterisations and configurations. This is unlike many existing approaches that manipulate the execution platforms directly. Our approach has the clear benefit that our system becomes agnostic to the target NFV platform and does not require interface changes in these platforms. Further, it ensures that our system can quickly be adapted to new MANO solutions by implementing an additional module to generate NFV descriptors for the new platform. This approach is only limited by the expressiveness of the VNF and SFC model used by the target platform, i.e., if a SUT can be expressed with the descriptors of a target platform, we can provide a plugin for it.

Figure 8.2 shows our general system design as well as its benchmarking workflow. The figure also contains annotations with technology options for external components. In the first step (*1. Define*), a user creates a so-called "performance experiment descriptor (PED)", a YAML-based descriptor file that contains all necessary information to perform a benchmarking experiment. In particular, a PED references the SUT that should be benchmarked, e.g., a VNF/NS package or descriptor, and it includes descriptions of all service configurations that should be tested, e.g., different resource assignments for the tested VNFs (see Section 8.3.2 for more details). The PED is used to trigger our benchmarking framework by using its CLI interface. Alternative interfaces, like REST-based interfaces, are possible as well. These interfaces also allow the integration of our benchmarker into existing CI/CD workflows (R4). The benchmarker reads the PED and forwards the request to its *descriptor engine*. This module takes the descriptors of the SUT referenced by the PED file and embeds (or extends) them with additional measurement VNFs, called "measurement probes (MPs)". Our system offers default MPs that contain standard networking test tools, like `iperf` or `hping`. A tester can replace these by any custom measurement VNF that may contain domain-specific or proprietary traffic generators. For example, an message queuing telemetry transport (MQTT) traffic generator can be used to benchmark IoT scenarios. After the embedding step, one copy of the new SUT description, for each configuration specified in the PED, is generated (*2. Generate*). This results in a set of SUT packages $C = \{c_i \mid i \in \mathbb{N} \wedge 0 < i \leq n\}$ for $n$ different configurations

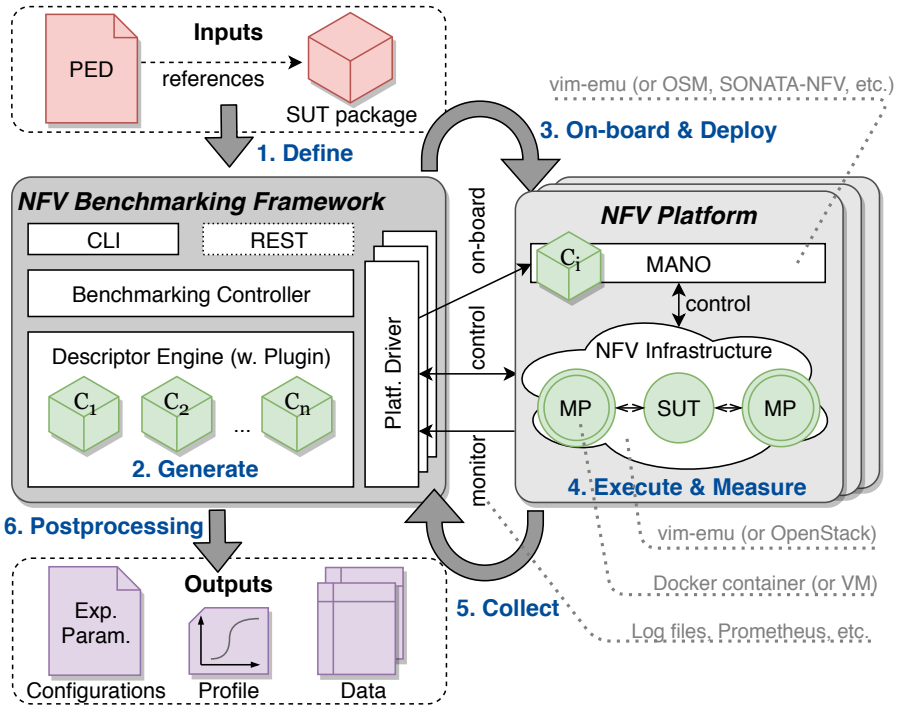



Figure 8.2.: System architecture of our benchmarking framework interacting with several NFV platforms. The figure also shows the general workflow and generated artefacts and is annotated with external technologies that can be used.

that should be tested. Each of them includes resource configurations, like number of CPU cores assigned to a VNF, MP configurations, like the packet sizes used by a traffic generator, as well as SUT-specific configurations. The *descriptor engine* itself offers a plugin interface for service description generators so that our benchmarker becomes service-descriptor agnostic and can be extended to further description formats, e.g., OSM [ETS16c] (R2).

In the third step (*3. On-board & Deploy*), the system iterates over the generated configurations. In each step, one of the generated configurations ($c_i$) is on-boarded and deployed on a free execution platform using the *platform driver* modules. These drivers act as a client to the execution platform and form an abstraction layer between specific MANO northbound interfaces and our internal control mechanisms (R3).

Once a service instance is up and running, MPs are triggered, e.g., the contained traffic generators are activated, and start to stimulate the SUT. The system now enters the measurement phase in which the actual performance achieved by the SUT is recorded (*4. Execute & Measure*). The measurement phase either ends after a predefined time limit or once a MP has finished its work, e.g., a full traffic trace has been replayed. After this, the SUT instance is destroyed and removed from the platform before the next SUT configuration

is deployed ($c_{i+1}$). We call the deployment, execution, and test of a single SUT configuration a *benchmarking round*. The overall benchmarking process finishes once all $n$ benchmarking rounds (one for each of the $n$ configurations) have been executed.

During a benchmarking round, performance data is collected in two ways (*5. Collect*). First, SUT-internal performance metrics are monitored, including log files inside the MP and SUT (the later is only possible in white-box benchmarking scenarios with direct access to the internals of the SUT). Second, platform metrics, like packet counters of network interfaces, are collected through the platform's monitoring APIs. The latter are platform-specific and may not be available on each execution platform. In both cases, two types of data can be collected. First, experiment data, which is collected at the end of each benchmarking round and summarises this round, e.g., total number of packets processed by the SUT in the benchmarking round. Second, time series data, which is continuously collected during the benchmarking round, e.g., samples of CPU utilisation collected once per second. Both types of data are stored and are available to the user afterwards.

As a last step (*6. Post-processing*), all measured data collected from various sources are aggregated and stored in unified, table-based formats. For experiment data, each row in the table represents exactly one of the tested SUT configurations. For time-series data, multiple tables, one for each collected metric, indexed by time stamps are created. These tables are then passed to a post-processing module that automatically triggers user-defined analysis scripts that can, for example, perform statistical analysis on the collected data sets (R1).

### 8.3.2. Describing benchmarking experiments

One of our key contributions is the so called performance experiment descriptor (PED) which is an easy-to-understand, human-readable description format used to define benchmarking experiments end-to-end. This not only simplifies the creation and definition of new experiments, it also helps to repeat existing experiments and provides a standardised way to exchange and share experiments. Listing 8.1 shows a (shortened) example of such a YAML-based descriptor showcasing the key features of our description approach.

In the header (line 1–5), the descriptor contains general information and version fields. The PED also contains an uniform resource locator (URL) to the SUT definition which can be provided in different formats, e.g., a 5GTANGO service package (line 7). This reference is used by the *descriptor engine* to access the definitions of the SUT that should be benchmarked.

```
1  descriptor_version: 0.2
2  vendor: "de.upb"
3  name: "ped1_ids_proxy_service_example"
4  version: "0.1"
5  author: "Manuel Peuster, Paderborn University"
6  # path to the SUT we want to profile
7  service_package: "file://services/ns-2vnf-ids-proxy.1.0.tgo"
8  # definition of benchmarking experiments
9  service_experiments:
10   - name: "service_throughput_traces"
11     # basic experiment configurations
12     repetitions: 25
13     time_limit: 120
14     time_warmup: 30
15     # NS to be used (vendor.name.version reference)
16     target:
17       vendor: "de.upb"
18       name: "ns-2vnf-ids-proxy"
19       version: "0.1"
20     # definition of measurement probes
21     measurement_probe:
22       - name: "mp.input"
23         connection_point: "ns:input"
24         container: "mpeuster/tng-bench-mp"
25         address: "20.0.0.1/24"
26       - name: "mp.output"
27         # (...)
28     # experiment parameters to be tested
29     experiment_parameters:
30       - node: "de.upb.ids-suricata.0.1"
31         cmd_start: ["./start.sh small_ruleset",
32                     "./start.sh large_ruleset"]
33         cpu_bw: {"min": 0.05, "max": 1.0, "step": 0.05]
34         cpu_core_set: ["0", "0, 1"]
35         mem_max: [128, 256, 512, 1024]
36         # (...)
37         io_bw: null
38       - node: "de.upb.proxy-squid.0.1"
39         # (...)
40       - node: "mp.input"
41         cmd_start: "tcpreplay --preload-pcap smallFlows.pcap"
42         # (...)
43   - name: "service_throughput_iperf"
44     extends: "service_throughput_traces"
45     # (...)
```

Listing 8.1: Example PED (shortened) showing the main features of our experiment description approach

Each PED specifies one or multiple experiments (line 9). Each experiment has a unique name (line 10), number of repetitions (line 11), a time limit for a single benchmarking round (line 13), and a warmup time that gives the SUT some time to bootstrap and configure before it is actually tested, e.g., traffic is

sent to it (line 14). An experiment also contains a `target` field containing an identifier of the service descriptor to be used (line 16). This is needed because SUT packages might contain multiple VNFs and NSs. They are referenced using a triple of `vendor`, `name`, and `version` field, as it is typically done in the ETSI, 5GTANGO, or OSM data models.

In the next section of the PED, the measurement probes used in the experiment are defined. Besides a unique name (line 22), these definitions must include a reference to a connection point of the SUT so that the descriptor engine knows how to combine and interconnect measurement probes with the benchmarked VNF or NS (line 23). The definitions also specify the container or VM image that should be used to deploy the measurement probe (line 24) and offer optional fields for network configurations (line 25).

Finally, a set of experiment parameters, for each of the VNFs contained in the SUT as well as for all measurement probes, must be specified. Those parameters include, for example, resource configurations, like the number of vCPU cores (line 34) or start parameters for the involved VNFs and probes (line 31). Platform-dependent resource configurations, like the available CPU time of a container (also called CPU bandwidth in the container domain), are also possible and are ignored if the experiments are executed on a platform that does not support such configurations (line 33).

To simplify the specification of complex parameter studies and significantly reduce the effort required to define new experiments, we add two features that are inspired by the configuration language of simulation tools, like OMNeT++ [OMN05]. First, we support macros for automated parameter expansion. Those macros can be either specified as loops (line 33) or as lists of values (line 35) that should be tested for a specific parameter. Second, we support inheritance for experiment descriptions (line 44). If a second experiment extends a first experiment, it inherits all configurations specified by the first experiment and can overwrite only the parameters that should be different. With these two features, our description approach allows to efficiently specify complex parameter studies that can then be automatically executed by the presented benchmarking framework.

Based on the PED and the SUT specification, our benchmarking framework will generate one SUT configuration for each combination of parameters that should be tested. This is done by computing the Cartesian product [War90] of all specified experiment parameters and their assigned values. This means that the example experiment in Listing 8.1 results in $\mathit{repetitions} \cdot |\mathit{cmd\_start}| \cdot |\mathit{cpu\_bw}| \cdot |\mathit{cpu\_core\_set}| \cdot |\mathit{mem\_max}| = 25 \cdot 2 \cdot 20 \cdot 2 \cdot 4 = 7680$ different configurations, and thus benchmarking rounds, to be executed. This already shows why automation is key in this area, because manually executing hundreds or even thousands of different experiments is infeasible.

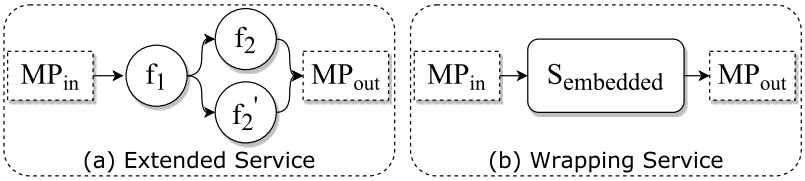



Figure 8.3.: SUT descriptor generation examples. Extended SUT descriptor (a) and embedded SUT descriptor $S_{\text{embedded}}$ (b).

The generation of these configurations highly depends on the VNFD and NSD models used by the target execution platforms. These are often similar but in most cases not exactly the same. For example, the descriptors of SONATA [SON15b], 5GTANGO[5GT17a], and OSM [ETS16c] are all based on the ETSI description model [ETS18c], but they differ in implementation details, like field names. This is why we support the use of different plugins to generate the SUT configurations, allowing us to implement specific generators for any execution platform.

One of the main functionalities of these generators is to extend the VNFD or NSD of the SUT with additional measurement probes (that can be considered VNFs themselves). This can be achieved with two different approaches as shown in Figure 8.3. The first approach extends the service graph of the SUT itself by appending the additional probe VNFs to the connection points specified in the PED (a). The second approach, in contrast, does not modify the NSD of the SUT but embeds it into another NSD that contains the measurement probes (b). As shown in the figure, the second approach has a much cleaner design and simplifies the generator implementation. However, it requires that the execution platform's description models support hierarchical service structures, which is an advanced feature often not supported by today's MANO systems (e.g., 5GTANGO or OSM). This is why option (a) is still needed and used in our prototype.

### 8.3.3. Packaging benchmarking results

Our benchmarking framework collects different metrics, including time series metrics, during each benchmarking round and stores them in table-based data formats for further processing. The collection can either be done by using the outputs of the test tools executed inside the measurement probes or with platform-specific monitoring systems accessible trough the platform drivers.

The collected results can finally be analysed and normalised, for example, lookup tables can be created that are then bundled with the SUT and used by MANO systems, e.g., for NS-specific scaling decisions. These analysis tasks highly depend on the use cases and the kind of collected data, as Chapter 10 shows. To this end, our system allows to plug-in arbitrary post-processing

scripts that are automatically executed at the end of a benchmarking experiment. Those scripts can be used to turn the raw data into an NFV-PP format that can be understood and used by other systems. To bundle the final benchmarking results with the SUT, 5GTANGO's advanced NFV packaging format can be used [5GT18a]; it allows to reference digitally-signed benchmarking results. This concept does not only allow to exchange benchmarking results between different stakeholders in the NFV ecosystem, e.g., between operators of public catalogues and service providers. It also supports new business models enabled by a concept called "verification and validation (V&V) platform" introduced by the 5GTANGO project and presented in one of my papers [Peu+19b]. This concept is out of scope of this thesis.

## 8.4. Case study: Chain-based benchmarking

We perform a series of experiments to test our benchmarking solution and to answer the question whether approaches that combine benchmarking results, obtained from isolated VNFs benchmarks, to model the end-to-end performance of an SFC are sufficient or if end-to-end benchmarking solutions are needed?

### 8.4.1. Scenarios and approach

For this study, we use linear SFCs consisting of up to three different VNFs all acting as forwarding elements. The first used SFC ($S_{\text{OVS}}$) contains three simple switching VNFs ($f_{\text{OVS}}$) each realised by an OVS instance running as layer 2 learning switch in userspace datapath mode within a Docker container. We consider this SFC as our baseline scenario in which we expect a direct linear relationship between resources assigned to the SFC and achieved performance, since all packets are just forwarded between the VNFs, without further processing, until the end of the SFC is reached.

Further, we use three more complex VNFs, each configured to act as layer 4 forwarding element, that are chained in different orders to create three additional SFCs ($S_1$, $S_2$, $S_3$). The used VNFs are Nginx ($f_N$) [NGI04] configured as TCP load balancer, the TCP relay Socat ($f_S$) [Soc01], and Squid proxy ($f_P$) [Squ96] with disabled caching functionality to forward every packet. These SFCs are used to investigate the impact of more complex VNFs, which work on higher networking layers, on the end-to-end SFC performance. The goal is to demonstrate that, even with a simple linear chain, it becomes complicated to accurately model the performance of the SFCs based on single-function VNF benchmarking results. To do so, we measure the performance for each of the isolated VNFs as well as for multiple setups of the full SFCs. All tested VNFs

Table 8.1.: Benchmarking scenarios considered in the case study

| | |
|---|---|
| isolated VNF $f_{\text{OVS}}$ | $\text{MP}_U \longleftrightarrow f_{\text{OVS}} \longleftrightarrow \text{MP}_W$ |
| isolated VNF $f_N$ | $\text{MP}_U \longleftrightarrow f_N \longleftrightarrow \text{MP}_W$ |
| isolated VNF $f_S$ | $\text{MP}_U \longleftrightarrow f_S \longleftrightarrow \text{MP}_W$ |
| isolated VNF $f_P$ | $\text{MP}_U \longleftrightarrow f_P \longleftrightarrow \text{MP}_W$ |
| SFC $S_{\text{OVS}}$ | $\text{MP}_U \longleftrightarrow f_{\text{OVS}} \longleftrightarrow f_{\text{OVS}} \longleftrightarrow f_{\text{OVS}} \longleftrightarrow \text{MP}_W$ |
| SFC $S_1$ | $\text{MP}_U \longleftrightarrow f_N \longleftrightarrow f_S \longleftrightarrow f_P \longleftrightarrow \text{MP}_W$ |
| SFC $S_2$ | $\text{MP}_U \longleftrightarrow f_S \longleftrightarrow f_P \longleftrightarrow f_N \longleftrightarrow \text{MP}_W$ |
| SFC $S_3$ | $\text{MP}_U \longleftrightarrow f_P \longleftrightarrow f_N \longleftrightarrow f_S \longleftrightarrow \text{MP}_W$ |

and SFCs are deployed between two probes, a web-service ($\text{MP}_W$) and end users ($\text{MP}_U$), which are used to perform the measurements. Table 8.1 shows a full list of all considered scenarios and involved VNFs.

All scenarios are deployed with the presented benchmarking framework using vim-emu [PKV16] as execution platform. As described in Part II of this thesis, vim-emu allows to quickly deploy complex SFCs consisting of container-based VNFs on a single physical machine. All used VNFs are packaged as Docker containers. In addition, we execute experiments on multiple physical machines, using Maxinet [Wet+14], to verify that vim-emu running on a single physical machine can be used for benchmarking as we proposed in [PK16b]. In both cases, each VNF container is allocated to a dedicated CPU core for isolation. To check the performance of our VNFs and SFCs under different resource configurations, we allocate different fractions of CPU time to each individual VNF container to emulate a large set of different resource configurations. It is important to note that the resulting performance numbers, generated by these experiments, should not be taken as absolute values but they allow us to compare our scenarios. The experiments are executed on a single machine with Intel(R) Core(TM) i7-960 CPU @ 3.20 GHz, 8 cores, hyper threading, and 24 GB memory. For the distributed (Maxinet) setup, multiple of these machines, interconnected by 10G Ethernet interfaces, are used. In the Maxinet setup, a dedicated machine is used for each deployed VNF container. All error bars in this chapter indicate 95 % confidence intervals based on 10 repetitions.

We focus on two metrics to measure VNF and SFC performance: Overall throughput and response time.

### 8.4.2. Throughput: Isolated function vs. service chain

In the first set of experiments, we measure the total throughput between web service ($\text{MP}_W$) and end users ($\text{MP}_U$) by downloading 1.0 MByte files with random content to simulate a virtualised content delivery network (vCDN)

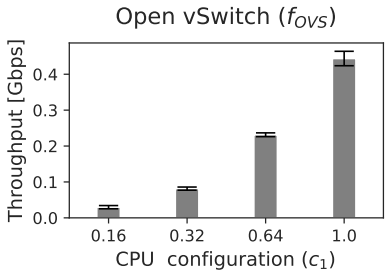



Figure 8.4.: Throughput of the OVS VNF under different CPU time configurations executed in a single-machine (vim-emu) setup.

service that is served through our VNFs and SFCs. In this setup, we use Apache2 [The93] running in $MP_W$ and Apachebench [The93] installed in $MP_U$ to run the downloads.

The first experiment focuses on our baseline scenarios in which we measure the throughput of a single OVS VNF under different configurations as well as the throughput achieved by an SFC of three chained OVS VNFs. Figure 8.4 shows the results of the isolated VNF measurements. It clearly shows the linear relationship between CPU configuration and the resulting performance. The small variance of the results show that the vim-emu platform is well suited for benchmarking experiments.

Using the results reported in Figure 8.4, which provide us with performance numbers for an individual VNF, we naturally model the expected throughput of a linear SFC ($\tilde{T}_{\text{SFC}(C)}$), composed of three chained OVS VNFs, as the minimum among the throughputs measured in the single-VNF experiment ($T_{\text{OVS1}}, T_{\text{OVS2}}, T_{\text{OVS3}}$) for a given configuration, as shown in Equation 8.1. As input to this model, a tuple $C$ containing a CPU configuration for each of the three VNFs is used: $C = (c_1, c_2, c_3)$. The assumption behind this model is that the VNF with the smallest throughput for the given configuration determines and limits the overall performance of the composed SFC.

$$\tilde{T}_{\text{SFC}(C)} = \min\{T_{\text{OVS1}(c_1)}, T_{\text{OVS2}(c_2)}, T_{\text{OVS3}(c_3)}\} \tag{8.1}$$

We then compare the predicted SFC throughput ($\tilde{T}_{\text{SFC}(C)}$) to measurement results we obtain from an experiment in which we deployed the complete SFC with three OVS VNFs and measured its end-to-end throughput. Figure 8.5 shows this comparison for different CPU time configurations. It can be seen that the measured end-to-end performance of the SFC is close to the performance predicted by the model in most of the cases. For some configurations, however, the modelled performance is up to 7.61 % less than the actually measured performance. This result shows that for simplistic SFCs, composed of VNF that do simple packet forwarding, naive performance models can

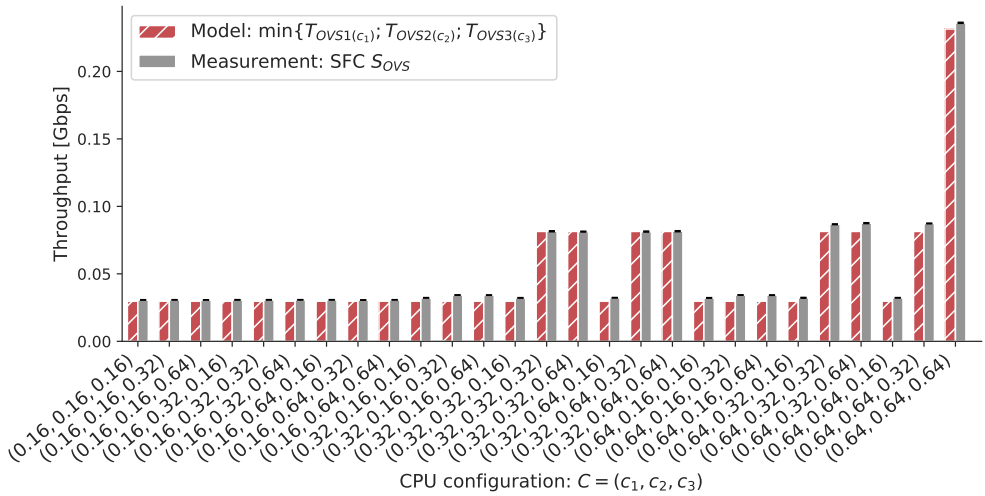



Figure 8.5.: Throughput of an SFC composed of three OVS VNFs under different CPU time configurations compared to the expected throughput modelled on basis of the results from our single-VNF measurements.

produce reasonable results. But many real-world SFCs are composed of VNFs that perform more complicated tasks such that a model-based approach might be infeasible as we confirm in the following.

The next experiment relies on the same setup as the previous experiment but uses the layer 4 forwarding VNFs ($f_N$, $f_S$, $f_P$) and SFCs ($S_1$, $S_2$, $S_3$) described in Table 8.1. In addition, we not only execute this experiment on a single vim-emu machine but also run it as a distributed scenario on four physical machines using Maxinet. The goal of this is to confirm that our results are not biased by the use of a single-machine running vim-emu as experimentation platform.

Figure 8.6 shows the results for each of the three used VNFs and different CPU configurations, measured in the single-machine (vim-emu) and multi-machine (Maxinet) setup. Again, the results indicate a linear relationship between CPU time and throughput and show that $f_N$ performs best. Moreover, it highlights that the absolute performance values of the single-machine setup are much better than in the multi-machine setup. The reason for this is that our benchmarking platform directly interconnects the VNF containers in the single-machine setup (virtual Ethernet pairs between containers). This means that there are no intermediate software switches that might consume additional resources and no tunnels between physical machines that need to be traversed. It can also be observed that the variance of the results in the single-machine case is smaller, which is also caused by the absence of intermediate switches and tunnels between the VNFs. It can be seen that vim-emu-based single-machine setups are better suited to solely focus on VNF performance and to

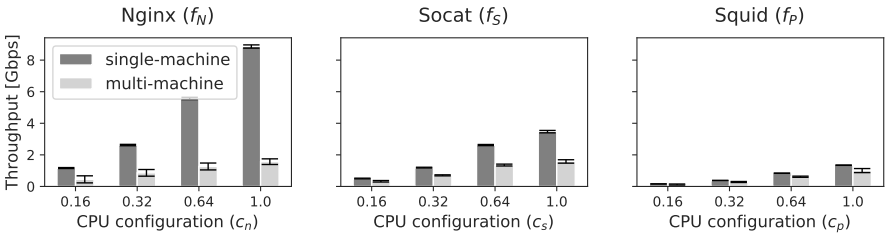



Figure 8.6.: Throughput of the three VNFs under different CPU time configurations executed in a single-machine (vim-emu) and multi-machine (Maxinet) setup.

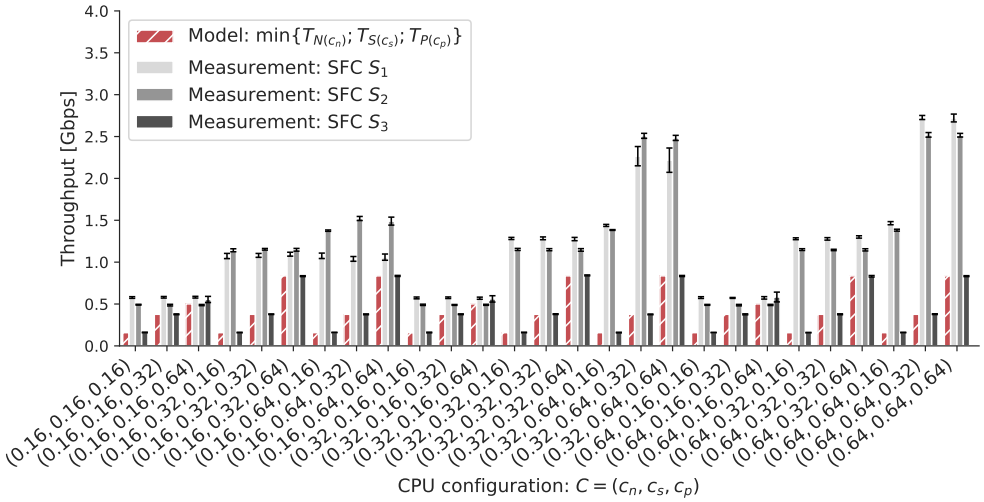
get more stable results.

Similar to the OVS experiment, we use the results from the isolated VNF measurements, reported in Figure 8.6, to model the expected throughput of a linear SFC composed of all three VNFs. As before, the model uses the minimum of the throughput measured in the single-VNF experiments ($T_N, T_S, T_P$) for a given configuration $C = (c_n, c_s, c_p)$, as shown in Equation 8.2.

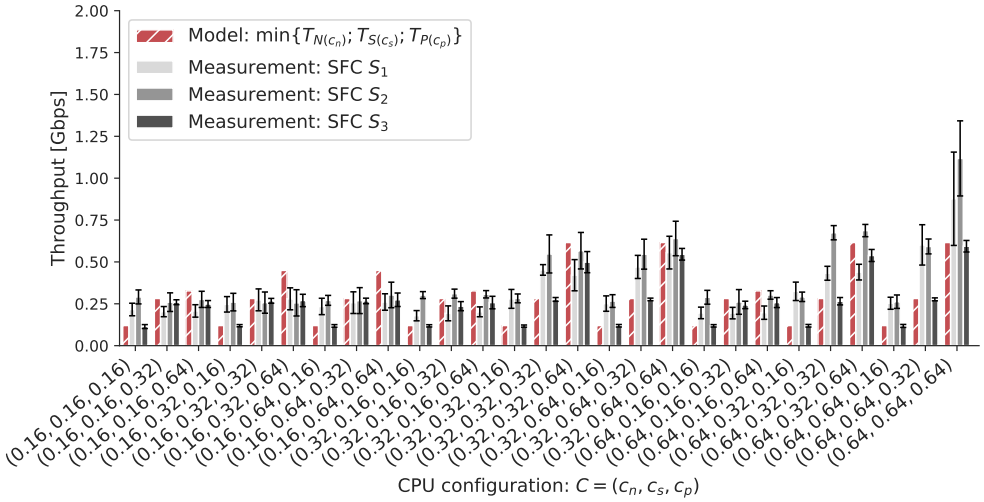
$$\tilde{T}_{\text{SFC}(C)} = \min\{T_{N(c_n)}, T_{S(c_s)}, T_{P(c_p)}\} \tag{8.2}$$

Figure 8.7 compares the predicted SFC throughput ($\tilde{T}_{\text{SFC}(C)}$) to the measured results we obtain from the end-to-end benchmarking experiments. All these experiments are executed on our single-machine setup (Figure 8.7a) and on our multi-machine setup (Figure 8.7b). The overall performance of the single-machine setup is higher (up to 2.6 Gbps) compared to the multi-machine case (1.2 Gbps). The later also shows more variance in the results, which is caused by the additional software switches and tunnels needed to deploy the benchmarked SFC across multiple physical machines.

Both setups show that even though the modelled results are often near to the experimental results there are still many configurations in which the real performance of the SFC is much better than predicted by the model—a non-intuitive result. In fact, the end-to-end measurements show an up to seven times higher throughput than initially predicted in some of the configurations. This is different from the results obtained from the OVS experiment, which has been much closer to the modelled results. A reason for this could be that the involved VNFs act on layer 4 and above, which means that between every pair of VNFs a dedicated TCP connection is used that terminates at the next VNF. As a result, each VNF does its packet processing independently from the other VNFs and forwards the packets to the remaining VNFs of the chain whenever it is ready to do so. In the OVS case, in contrast, flows are passing through the SFC end-to-end without being terminated at the individual VNFs, i.e., the packets are simply forward without further processing.




(a) single-machine setup




(b) multi-machine setup (using Maxinet)

Figure 8.7.: Throughput of three SFC configurations under different CPU time configurations compared to the expected throughput modelled on basis of the results from our single-VNF measurements. Experiments are executed in our (a) single-machine setup and (b) multi-machine setup.

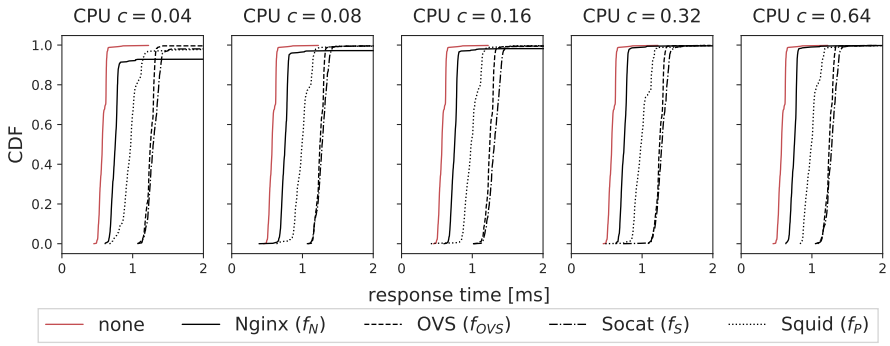



Figure 8.8.: Empirical response time CDFs for each VNF and a setup without VNF between $MP_U$ and $MP_W$.

The presented results also show that SFC $S_3$ performs worse than the other two SFCs even though the exact same VNFs are used in all three cases. It shows that the ordering of the functions in an SFC can have an effect on the SFC's end-to-end performance. This is something that can clearly not be captured if VNFs are benchmarked in isolation. We confirm this finding with a second set of experiments that focus on response time as a second performance metric.

### 8.4.3. Response time: Isolated function vs. service chain

In the second set of experiments, we investigate the impact of different VNF and SFC configurations on the response time. This is done by performing 500 response time measurements (hypertext transfer protocol (HTTP) HEAD requests using Httping [Htt05] installed in $MP_U$) in each benchmarking round. These experiments are only performed in the single-machine setup, because a multi-machine Maxinet setup would have introduced a lot of bias to the response times caused by intermediate switches and network tunnels that need to be traversed. Figure 8.8 shows the results for our single-VNF scenarios as well as for a scenario in which no VNF is deployed between $MP_U$ and $MP_W$. It shows that each VNF implementation has slightly different response times and that the allocated CPU time has only a small effect on these response times.

We again create a model to predict the behaviour of our SFC scenarios based on single-VNF measurements. This model approximates the response time of an SFC by the sum of the response times measured in the individual VNF experiments. This assumption makes sense since we know that all our SFC scenarios use linear chains in which packets have to traverse every VNF. Building the sum of the response times from the single-VNF measurements can be understood as summing up independent random variables and is done by computing the discrete convolution using their probability density

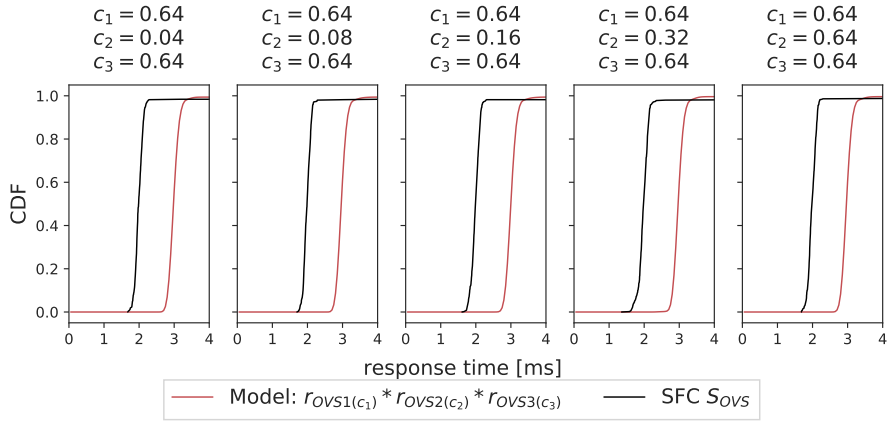



Figure 8.9.: Measured empirical response time CDFs of the OVS SFC compared to the modelled response times derived from single-VNF measurements.

functions (PDFs). Let $r_x(t)$ with $x \in \{f_1(c_1), f_2(c_2), f_3(c_3)\}$ be the PDF derived from the results of the given single-VNF experiment $x$ with a given CPU configuration. For example, $r_{N(0.32)}(t)$ is the density of the response times of the Nginx VNF configured with 0.32 % CPU time. The approximated response time PDF of an SFC with three functions is then defined as shown in Equation 8.3.

$$\begin{aligned} \tilde{r}_{\mathrm{SFC}(C)}(t) &= (r_{f_1(c_1)} * r_{f_2(c_2)} * r_{f_3(c_3)})(t) \\ &= \sum_{i=0}^{t} \left( \sum_{j=0}^{i} r_{f_1(c_1)}(j) r_{f_2(c_2)}(i-j) \right) r_{f_3(c_3)}(t-i) \end{aligned} \tag{8.3}$$

This model can be used for all our linear SFC scenarios due to the associative and commutative nature of the convolution. Figure 8.9 compares this model to the measured results of our OVS SFC experiments for changing CPU times of the second VNF in the chain. It demonstrates that the SFC does not behave like expected and shows response times that are faster as the modelled response times.

The same happens for our second set of SFCs ($S_1$, $S_2$, $S_3$) as shown by Figure 8.10. The figure compares the response time under different CPU time configurations of VNF $f_P$ with the modelled response times. Changing the CPU configurations of the other VNFs gives similar results. Again, the measured response times are faster than the modelled response times. It can also be seen that the order of the VNFs in our linear SFCs matters if the layer 4 forwarding VNFs are used. It confirms the results of the throughput experiments and illustrates that modelling SFC performance based on single-VNF measurements, without detailed knowledge about the used VNFs and SFC

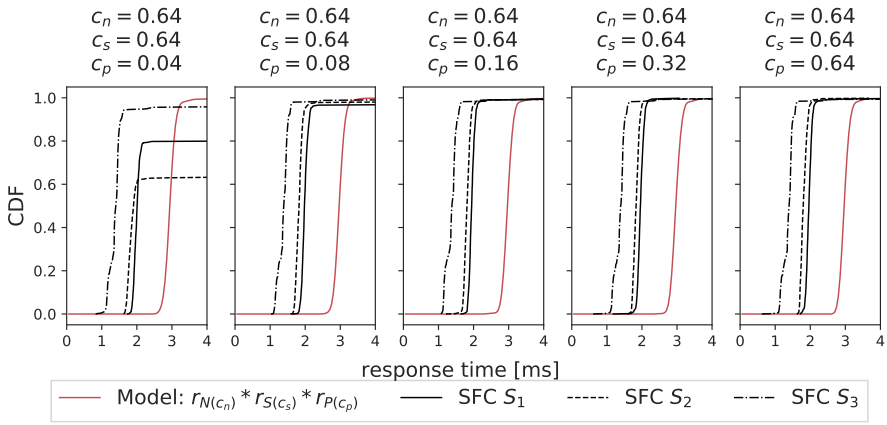



Figure 8.10.: Empirical response time CDFs measured for three SFC setups ($S_1$, $S_2$, $S_3$) compared to the modelled response times derived from single-VNF measurements.

structure, does not lead to accurate results. It shows that finding good performance models for SFCs is often complicated, might require detailed insights into the implementation of the used VNFs as well as structure of the SFC, and can barely be automated. As a consequence, SFC benchmarking solutions that support end-to-end SFC measurements, as presented in this chapter, turn out to be a much better solution for fully automated environments in which black-box benchmarking is required.

## 8.5. Conclusion

The presented benchmarking framework allows to fully automate end-to-end SFC benchmarking experiments using a simple, yet flexible, description approach. It goes beyond existing NFV benchmarking solutions and is a step toward fully automated NFV DevOps toolchains. The presented results show that the order of the VNFs in an SFC directly affects the SFC's end-to-end performance. This behaviour cannot easily be simulated with natural performance models combining single-VNF benchmarking results. Our study also demonstrates that end-to-end SFC benchmarking is a better solution for automated benchmarking setups than benchmarking VNFs in isolation since it removes manual modelling steps and allows black-box benchmarking of entire SFCs. We foresee such benchmarking procedures not only in the SFC development process but also as part of automated verification and validation systems [Peu+19b] and as part of on-boarding procedures giving MANO systems initial information about the behaviour of SFCs prior their deployment.

The prototype of the presented framework is open-source and acts as one of

the reference implementations of the IETF draft about VNF benchmarking automation methodologies [Ros+18]. This draft also aims to standardise a description approach for VNF benchmarking based on the PED approach presented in this chapter.

# 9. Benchmarking under time constraints

This chapter focuses on the challenge of large configuration spaces that need to be explored during automated benchmarking processes. Such large configuration spaces become especially challenging if a benchmarking process should be finalised in a fixed amount of time in which not all possible configurations can be tested. Besides a formal problem formulation of NFV benchmarking under time constraints given in Section 9.2, we present related work in Section 9.3. The key contributions of this chapter are twofold: First, the design, workflow, and used algorithms for a time-constrained benchmarking (T-CB) system are presented and an open-source prototype platform [Peu18b] is introduced in Section 9.4. Second, the prototype is used to evaluate the general T-CB concept in Section 9.5, before concluding the topic in Section 9.6. This chapter is based on my paper [PK18] and contains figures and verbatim text from this paper.

## 9.1. Introduction

Benchmarking processes collect information about the runtime behaviour of the tested SFC (the SUT) by deploying it under different resource configurations and testing its resulting performance. The results of these benchmarking processes, the so-called NFV-PPs, are then used by MANO systems to optimise their resource dimensioning decisions. It is essential that the obtained NFV-PPs provide enough information to accurately describe the behaviour of the deployed SFC to meet performance goals and to avoid unexpected performance degradations caused by, for example, automatically deployed service updates.

A challenge for benchmarking solutions is, on the one hand, based on the fact that the configuration space sizes of even simple SFCs, which have to be explored during the benchmarking process, tend to become very large if the number of configuration parameters or number of involved VNFs increases. For example, consider an SFC with five VNFs, in which each VNF can have 1–16 CPU cores, {128, 256, 512, ..., 16384} MB memory and non-uniform memory access (NUMA) enabled or disabled; this leads to $16 \cdot 8 \cdot 2 = 256$ possible configurations per VNF and $256^5$ possible configurations for the

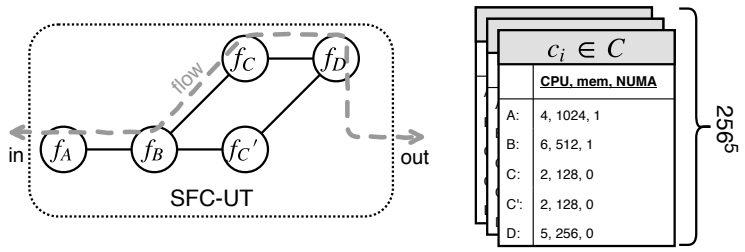



Figure 9.1.: Example scenario of an SFC with five VNFs and their configuration parameters

entire SFC, as shown in Figure 9.1. It can be seen that the configuration space of such an SFC is huge and it becomes infeasible to test each of these configurations during the benchmarking process—keep in mind that each test entails re-deploying or re-configuring the SUT. On the other hand, the benchmarking processes as such are expected to be performed as part of the NFV DevOps cycle and thus have to be completed in a reasonably short time, i.e., a couple of hours or even minutes [Kar+16]. Because of this, novel benchmarking solutions are needed that do not require to exhaustively test the complete configuration space of an SFC. These benchmarking solutions aim to already produce usable NFV-PPs at the beginning of the benchmarking process and potentially improve their quality, in terms of result accuracy, over time. In this context, result accuracy means the error between the expected performance values described in the NFV-PP and the performance values achieved in reality.

To fill this gap, we introduce the concept of "time-constrained benchmarking (T-CB)"; the goal is to produce SFC benchmarking results within a given time limit. After formulating the problem, we present the design, workflow, and used algorithms for a T-CB-enabled benchmarking system and evaluate it.

## 9.2. Problem formulation

Given the huge configuration space of an SFC and the fact that the re-deployment or re-configuration of an SFC takes a considerable amount of time [NL17], executing benchmarking measurements on the complete configuration space is infeasible. It gets even worse if the benchmarking process should be performed in a given time frame, i.e., with a given time constraint $l$. As a result, only a subset of the complete configuration space can be tested and benchmarking measurements for untested configurations need to be predicted using the available results.

More formally, we define the set of all possible configurations $V$ of a VNF as the Cartesian product of a series of sets $V = F_1 \times F_2 \times ... \times F_m$, where each set represents a single, discrete configuration parameter (also called a feature) $F_i$

and its possible values, e.g., number of CPU cores: $F_{\text{cores}} = \{1, 2, ..., 16\}$. Using discrete configuration parameters works fine since all relevant real-world configurations are also based on discrete values. In a complex SFC, multiple VNFs are combined and each of them can be configured independently of other VNFs with a configuration $c \in V$. For simplicity of the model, we assume that every involved VNF supports the same configuration space and thus all available configuration features of $V$. Based on this, the overall configuration space $C$ of a complex SFC with $n$ VNFs can be defined as $C = V_1 \times V_2 \times ... \times V_n$. Its cardinality as function of available configuration features and number of VNFs is given by $|C| = \left(\prod_{i=1}^{m} |F_i|\right)^n$.

For each SFC configuration that should be tested, the service is deployed and configured (taking setup time $t_{\text{up}}$), its performance is measured for a fixed amount of time ($t_{\text{measure}}$), e.g., traffic traces are sent to the SFC for $t_{\text{measure}}$ seconds, and it is terminated to free the resources (taking time $t_{\text{down}}$). After this, the next SFC benchmarking round is started and the next configuration is tested [PK17]. The total time $t$ needed to benchmark a single SFC configuration is hence $t = t_{\text{up}} + t_{\text{measure}} + t_{\text{down}}$ where $t$ is usually dominated by $t_{\text{up}}$ and $t_{\text{down}}$ [NL17]. We consider $t$ to be constant for each configuration to be tested.

As a result, the number of configurations $k$ that can be tested in a given time limit $l$ is limited by $k \leq \left\lfloor \frac{l}{t} \right\rfloor$. More specifically, we define the subset of configurations to be tested within the given time limit as $\tilde{C} \subset C$ and $|\tilde{C}| = k$. This subset is defined by a *selection function* $S_k : C \to \{0, 1\}$. For each configuration $c \in \tilde{C}$, we obtain benchmarking results ($R$), denoted by a function $\tilde{P} : \tilde{C} \to \mathbb{R}$ (benchmarking results could be tuples of real numbers or other values as well; this does not matter for the remaining discussion). We lift these measured benchmarking results, obtained on $\tilde{C}$, to predicted results for the entire configuration space by defining a benchmarking predictor $P : C \to \mathbb{R}$ that uses the measured results for $\tilde{C}$ as training data. We denote these predicted benchmarking results as $\hat{R}$.

This model poses two questions. First, how to best select the subset of $k$ configurations to be tested? And second, using the performance measurements made on these $k$ configurations, how to best pick the function that lifts $P$ so that the resulting NFV-PPs accurately predict the performance of the SFC compared to real measurements, i.e., minimising the prediction error? We will give answers to these two questions in the remainder of this chapter.

To measure the quality of the predicted results $\hat{R}$ compared to the measured results $R$, we use the *normalised root-mean-squared deviation (NRMSD)* as our main evaluation metric. The NRMSD is based on the mean-squared error (MSE), calculated over the entire configuration space with $n$ predictions, and is normalised using the range $(\max(R) - \min(R))$ which is available from

the measured data $R$ (Equation 9.1). We pick a normalised metric to ease comparison between NFV profiles with different scales.

$$\text{NRMSD} = \frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{R}_i - R_i)^2}}{\max(R) - \min(R)} \tag{9.1}$$

## 9.3. Related work

A couple of solutions for performance benchmarking of virtualised applications has already been proposed by the cloud-computing community. Most of them focus on solutions to benchmark single-VM applications [Woo+08; TZK16] but some solutions also support complex, composed applications [Tak+13]. In addition to this, the NFV community has also started to search for benchmarking solutions that focus specifically on NFV use cases [ETS18k; Ros+18]. These solutions either focus on benchmarking single VNFs or on evaluating NFV infrastructure deployments [RRS15; BS15; RBR17]. Others do consider benchmarking of complex SFCs [Cao+15] to characterise the performance of end-to-end services, which cannot be derived from isolated VNF benchmarks [PK17; NSS18], as also shown in Chapter 8. Some of the SFC benchmarking solutions even support automated testing of different VNF sequences in an SFC [NSS18]. All of these solutions, however, face the challenge of huge SFC configuration spaces that have to be explored, leading to impractical runtimes of the benchmarking process. None of the existing approaches can automatically select a subset of configurations to be benchmarked to systematically reduce the time needed to characterise an SFC. Even if it is possible to simply stop these existing NFV benchmarking solutions after the time limit is reached, irrespective of their current system state, it would cause very poor or even incomplete benchmarking results. This is because those solutions might only have explored a very small or unimportant part of the configuration space at the point in time when they are stopped. Because of this, smarter solutions are needed that start to sample the configuration space at large, in the first steps, and then successively improve the result quality until the time limit is reached. This is where our T-CB approach can be used as a complementary extension to the existing approaches as we do not tie our T-CB design to a specific benchmarking solution as shown in Section 9.4.1.

One solution that does smart selections in a cloud application context is called PANIC [Gia+15]. In their paper, the authors compare three selection approaches: Uniform grid, random sampling, as well as their greedy adaptive sampling algorithm (PGAS), each combined with different prediction (or interpolation) approaches, like linear regression. Their results indicate that testing a small subset of the overall configuration space already yields sufficient

results to generate reasonably good benchmarking results. Their solution focuses on a two-dimensional configuration space and their evaluation uses cloud benchmarks based on big data frameworks like Hadoop. Even if their PGAS algorithm can be used for NFV scenarios, as we show in Section 9.5 where we compare PGAS to our algorithms, the PANIC system as such was not designed for NFV scenarios and does not offer support for complex SFCs, e.g., it does not support configurations spaces with many features as it is required for SFCs.

An extension to PANIC is a decision tree-based selection approach, which was recently proposed in [GTK17]. This approach uses decision trees to iteratively partition the configuration space based on the accuracy of linear regression models applied to each of these partitions. The final partitioning and the selected configuration points, obtained during the partitioning phase, are then used to train a new decision tree-based model to finally represent the cloud application's performance behaviour. According to [GTK17], this approach tends to show a reduced accuracy when only small numbers of configurations are tested. Further, the approach tends to show better results when executed on configuration spaces with few, for example, two dimensions, which is rarely the case if SFCs, often consisting of more than two VNFs, should be benchmarked. As a result, their solution is not directly applicable to our scenarios. The use of decision trees, however, seems to be a promising approach and we have further investigated it in a Bachelor thesis written by Heidi Neuhäuser [Neu19] under my supervision. Her results show that a decision tree-based solution can improve the selection process in some cases, but they are not substantially better in general. This is why we think further research on this topic is needed, as described in Chapter 11.

Another interesting approach is called "NetBOA" and proposes the use of black-box optimisation techniques (i.e. Bayesian optimisation) to build an adaptive and "data-driven" traffic generator to measure network performance [Zer+19]. Even though the focus of the authors is not on time-constrained benchmarking of SFCs, the problem solved by NetBOA is similar to ours, e.g., the challenge of large configuration spaces. As a result, NetBOA can be considered to be complementary to our work and motivates further work, e.g., by adapting the NetBOA algorithm to be used as selector component for the system presented in this chapter.

Even though some of the related solutions try to reduce the size of configuration spaces that have to be explored during benchmarking, none of them has a notion of time-constrained benchmarking nor do they integrate with NFV benchmarking platforms. This underlines the novelty of our T-CB concepts for the NFV domain.

## 9.4. Designing a T-CB system

Based on our work presented in Chapter 8, we designed an SFC benchmarking system that explicitly supports time-constrained benchmarking. The system gets all possible configuration parameters and a fixed time limit as inputs, runs automated performance measurements on the SUT until the time limit is reached, and derives an NFV performance profile based on the available measurements.

### 9.4.1. Building blocks and workflow

Using the problem formulation (Section 9.2) we identify and analyse the required building blocks and workflows of a T-CB system and developed a prototype as shown in Figure 9.2. Its components are placed around one or multiple benchmarking platforms, e.g., tng-bench (Chapter 8), that execute the SUT and measure its performance under different resource configurations. Before that, the configurations to be tested are selected by the selection component $S_k$ (step 1) and serve, together with the service description, as inputs to the benchmarking platform(s) (step 2). Note that we do not tie our T-CB system to a specific SFC benchmarking platform and designed it to use arbitrary, existing solutions. To combine other benchmarking platforms with T-CB, they need to have an interface where the configurations to be tested during the benchmarking runs can be specified as well as an interface to output the measured performance results—requirements that are fulfilled by tng-bench [PK17], Gym [RBR17], and Probius [NSS18] (see Chapter 8). The measured results are forwarded to the prediction component $P$, which uses them as training data and generates predicted performance values for all possible configurations of the SUT (step 3). Finally, the predicted results $\hat{R}$ can be forwarded to a MANO system to optimise its resource assignment decisions and for automated lifecycle actions, like scaling, healing, or reconfiguration (step 4). Alternatively, the obtained benchmarking results $\hat{R}$ can be used to analyse the behaviour of the SUT, e.g., by an SFC developer, to debug performance issues.

The two main building blocks that distinguish this system from existing benchmarking solutions [PK17; RBR17; NSS18] are the selection component $S_k$ and the prediction component $P$. The selection component gets all possible configuration parameters of the SUT ($C$) and a maximum number of $k$ configurations to be selected as inputs before selecting the first configuration to be measured by the benchmarking platform. Once this is done, the selection algorithm selects the next configuration to be measured. This round-based design allows to not only use static selection algorithms, like random sampling, but also

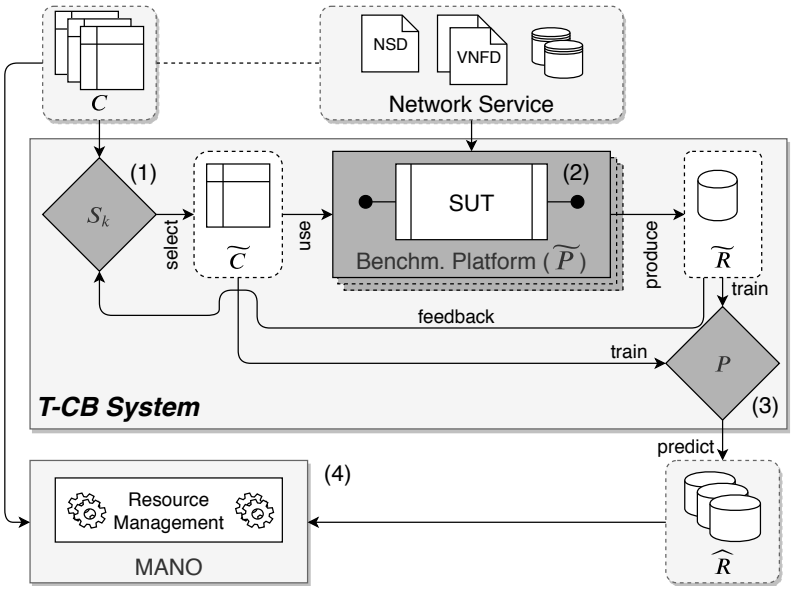



Figure 9.2.: Main building blocks and workflow of our T-CB system, build around existing benchmarking platforms, feeding the resource management component of a MANO system.

dynamic selection algorithms that use feedback of already performed measurements as additional input, potentially improving the selection quality. We present an example of a feedback-based selection algorithm in the next section. If multiple parallel benchmarking platforms are available in a T-CB system, a centralised $S_k$ component selects the configurations to be benchmarked, which are then distributed among the available benchmarking platforms (see Section 9.4.2). It is important that all parallel benchmarking platforms are built on top the exact same hardware and use the same software, in such a scenario, to ensure comparability between the obtained results. The resulting measurements are collected and processed by a centralised prediction component as described in Section 9.4.3.

### 9.4.2. Selection component

We implement three different selection algorithms for our T-CB system. First, a simple random selection algorithm, called uniform random selection (URS), that selects configurations uniformly at random and does not rely on the feedback of previous measurements.

Second, we re-implement the PGAS cloud benchmarking algorithm proposed by Giannakopoulos *et al.* [Gia+15] as a first example for a feedback-based algorithm. PGAS initially picks a fixed number of samples at the borders of

the configuration space (border points) before picking further samples based on the maximum distances between the measured results of the previous samples.

Third, we develop our own feedback-based algorithm, called weighted random VNF selection (WRVS), which comes in two flavours: WRVS1 and WRVS2. WRVS's general idea is to favour the configurations that belong to the VNFs of the SFC which seem to have a higher impact on the overall SFC performance. To detect those VNFs, the algorithm is split into two phases. First, a bootstrapping phase is used to select $n$ (WRVS1) or $2n$ (WRVS2) initial configuration points for an SFC with $n$ VNFs. Each of these configuration points minimises or maximises the configuration parameters of one of the $n$ VNFs and sets the configuration parameters of all other VNFs to their median values. More specifically, WRVS1 picks configurations that minimise parameters and WRVS2 picks configurations that minimise and maximise the parameters. For example, in an SFC with three VNFs in which only the number of vCPU cores (1...16) can be configured, the first two configuration points, using WRVS2, would be `(vnf1=1, vnf2=8, vnf3=8)` and `(vnf1=16, vnf2=8, vnf3=8)`. Thus, *vnf1* is once tested with minimum number of vCPUs and once with the maximum number of vCPUs with the goal to gain knowledge about the impact of the vCPU configuration parameter of *vnf1*. The definition how to minimise and maximise parameters is given by the developer of the VNFs or by the benchmarking experiment developer as annotation to the configuration space used as input to our system. Using this initial selection scheme, the algorithm collects information about the impact of the individual VNFs to the overall SFC performance. This impact is defined as the change to the overall SFC performance $\Delta_i$ when the configuration of VNF $i$ is altered. These values are used as weights in the next phase of the algorithm.

The second phase of the algorithm starts after $n$ or $2n$ configurations have been tested and randomly picks configurations from the overall configuration space until the limit of $k$ configurations is reached. This random selection process uses the weights from the first phase to favour the selection of configuration points which alter the configurations of those VNFs that have higher weights assigned. That is, configuration points from VNFs with a higher $\Delta_i$ are more likely to be selected for further benchmarking rounds. As a result, the feedback from the benchmarking process guides our selection algorithm to improve the overall benchmarking result by focusing on those parts of the configuration space that seem to have higher impact to the overall SFC performance.

### 9.4.3. Prediction component

This component can either be based on simple regression techniques or on more complex machine learning solutions. In our prototype, we utilise the

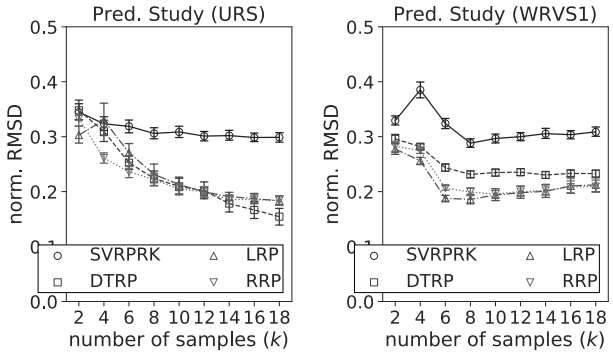
*scikit-learn* machine learning library [Bui+13] to implement the prediction component. Besides a simple *polynomial regression predictor*, we also use *support vector regression* predictors with different kernels, *decision tree regression*, *lasso regression*, *elastic net regression*, *ridge regression*, and *stochastic gradient descent regressions* predictors, resulting in a total of 11 prediction algorithms available in our prototype.

## 9.5. Evaluation

We use our T-CB system prototype [Peu18b] to evaluate our design and to study the impact of different selection algorithms, prediction algorithms, service topologies, and number of samples on the overall result quality, i.e., prediction error. To do so, we execute a set of benchmarking experiments in which all possible configurations of the SUT are tested. This exhaustive benchmarking step gives us baseline results serving as ground truth for later comparison. Then we use T-CB to execute benchmarking experiments that only test a fixed number of configurations $k$ and compare their results to the initial experiments using the NRMSD metric. In all our experiments, we use the maximum throughput as VNF and SFC performance metric captured during the benchmarking measurements. However, the presented system is not limited to throughput and can be used for arbitrary benchmarking metrics.

The presented experiments utilise our previous work about automated SFC benchmarking [PK17] and are based on a real-world benchmarking process using an SFC with three VNFs. This allows us to evaluate our system in a real-world scenario including realistic performance numbers that are based on measurements. As described in Chapter 8, the used SFCs are linear chains that contain three forwarding VNFs (Nginx [NGI04] configured as TCP load balancer, the TCP relay Socat [Soco01] and Squid Proxy [Squ96]) in different orders, resulting in three possible SFC topologies. Each VNF has a single configuration parameter (CPU time) and the maximum achieved throughput during a HTTP download is measured. The measurements to test each possible configuration of this chain took about 39 hours, with 60 seconds measurement time per configuration. Our raw measurements are available online and can be reused by other researchers [Peu18b]. Each of the following experiments has been repeated 30 times and the error bars indicate 95 % confidence intervals.

The first experiment analyses the behaviour of different prediction algorithms implemented in our T-CB system using URS and WRVS1 selectors. Figure 9.3 shows a comparison of four prediction algorithms, namely support vector regression (SVRPRK), decision tree regression (DTRP), lasso regression (LRP), and ridge regression (RRP). Results for the other implemented prediction algorithms, which produce similar results, are available in the project repository [Peu18b]. The figures show the NRMSD for different numbers of samples



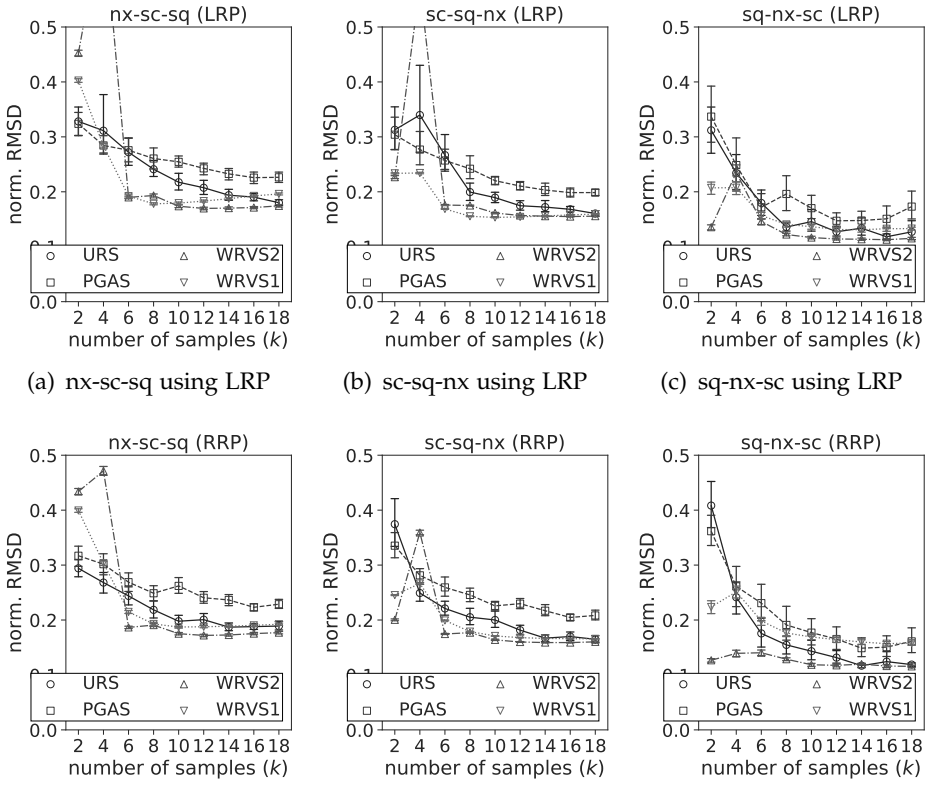

(a) URS (b) WRVS1

Figure 9.3.: Comparison of prediction algorithms for different numbers of measured samples using URS and WRVS1 selectors

$k$ selected from the overall configuration space of the benchmarked SFCs. The results show that already a small number of samples allows to reasonably well predict the behaviour of the overall configuration space of an SFC. They also show that the LRP and RRP predictors perform best for the used SFCs, which is why we selected them to be used in the following experiments.

Next, we compare the behaviour of our selection approaches using data from three different SFCs, namely nx-sc-sq, sc-sq-nx, and sq-nx-sc (Chapter 8), together with the LRP and RRP prediction algorithms as shown in Figure 9.4. The results show that the WRVS algorithm works better than the random selector (URS) and the PGAS approach in most of the cases, once $k \geq 6$ samples are selected. However, for $k = 4$ the models are often overfitted when WRVS is used. This could be a result of the static selection of the initial points during the bootstrapping phase of WRVS. The results show that the specific SFC in combination with the selected prediction approach directly impacts the selection quality, i.e., WRVS1 works much better with the RRP predictor if the sq-nx-sc SFC is benchmarked.

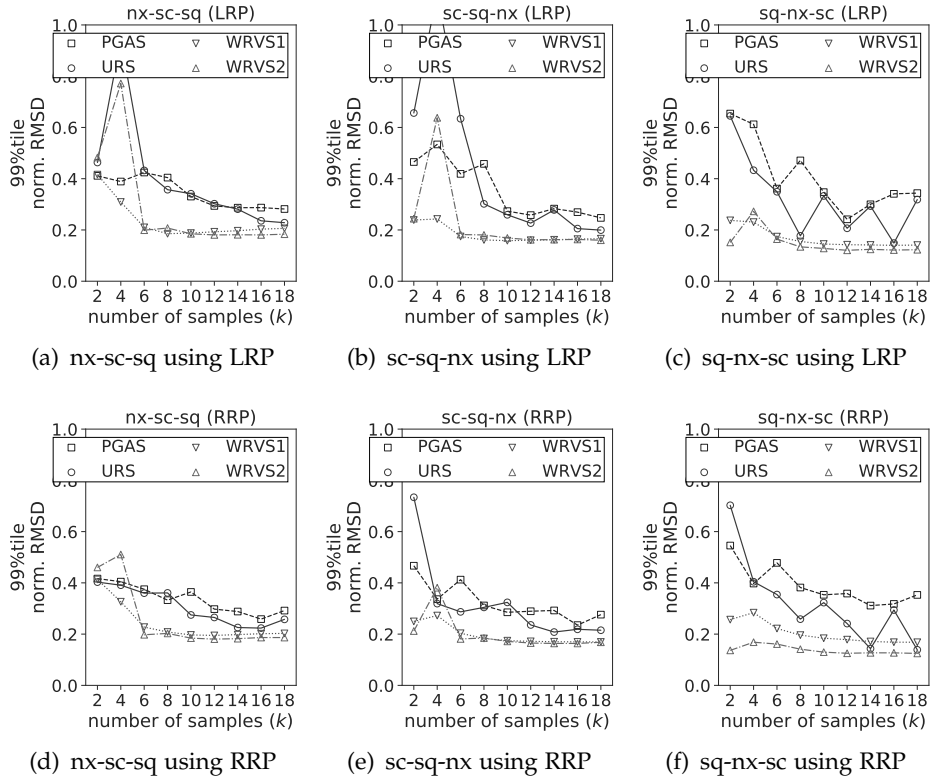
Nevertheless, those results report the mean over 30 experiment repetitions, which has the potential to hide the shortcomings of the randomised URS selector, which can, by chance, have a very good or very bad result in a single experiment run. In a real-world system, however, the benchmarking and selection process cannot be repeated multiple times because of the given time constrains. As a result, the selection approach should produce good results, e.g., a low NRMSD, in the first run. To study the behaviour of our selection approaches under those circumstances, we present the 99th percentiles of the NRMSD in Figure 9.5. The results clearly show the benefits of the proposed WRVS selection approach. In all scenarios, WRVS outperforms the two other approaches when $k \geq 6$. The results show that the URS selector shows a very




(a) nx-sc-sq using LRP (b) sc-sq-nx using LRP (c) sq-nx-sc using LRP

(d) nx-sc-sq using RRP (e) sc-sq-nx using RRP (f) sq-nx-sc using RRP

Figure 9.4.: Selector performance comparison using three real-world SFCs and two prediction approaches (LRP and RRP)




(a) nx-sc-sq using LRP (b) sc-sq-nx using LRP (c) sq-nx-sc using LRP

(d) nx-sc-sq using RRP (e) sc-sq-nx using RRP (f) sq-nx-sc using RRP

Figure 9.5.: Selector performance comparison using three real-world SFCs and two prediction approaches (LRP and RRP) showing the 99th percentile over the 30 experiment repetitions

unstable behaviour as the number of selected samples increases, a result of the random selection process (Figure 9.5c). Our WRVS approach, in contrast, shows a stable behaviour as the number of samples increases. The results also show that the PGAS approach does not work well for such SFC scenarios and is outperformed by WRVS in all scenarios. In some scenarios URS performs even better than PGAS, e.g., Figure 9.5f. If very small sample sizes (e.g., $k < 6$) are used, WRVS sometimes shows an unstable and overfitting behaviour, which seems to be caused by its bootstrapping phase, an issue to be investigated in future work.

Besides the experiments presented in this chapter, which use real-world performance measurements, we have also performed a set of experiments using synthetic performance models and have presented them in [PK18]. Those models use randomly selected functions (e.g., linear, polynomial, or exponential) to model the performance of the involved VNFs. The resulting experiments, however, did not provide any additional insights compared to the results reported here. Taking into account the results of the previous chapter, which clearly show that modelling the end-to-end performance of SFCs with simplis-

tic models does not work well, it becomes clear why such a synthetic approach is of limited use to evaluate our time-constrained benchmarking approach. These synthetic results can, however, be useful to test the basic functionality of the presented prototype and can, e.g., be used for unit testing.

## 9.6. Conclusion

The size of real-world SFC configuration spaces makes the application of existing NFV benchmarking solutions infeasible for agile DevOps environments. Even though existing NFV benchmarking solutions could be simply stopped after a given amount of time, the produced benchmarking results would only reflect a small subset of the configuration space and would loose important information about the SUT's performance behaviour. This chapter presents a solution for this by introducing our T-CB concepts for NFV.

To study these concepts, we present our open-source T-CB system [Peu18b] and use it to analyse different selection and prediction algorithms. Our results show that a T-CB system can generate reasonably accurate NFV-PPs by benchmarking only small subsets of the overall configuration space. We show that the subset of configuration points that are benchmarked has a big impact to the quality, in terms of prediction error, of the resulting NFV-PPs. Our presented selection algorithm outperforms related approaches from the cloud computing community and produces better and more stable results than approaches that randomly pick configurations for benchmarking. As a result, benchmarking processes can be performed under time constraints without loosing too much result accuracy. Still, there are other promising ideas to build such selection approaches, for example, based on decision trees as shown in [GTK17; Neu19], which have the potential to further improve the presented T-CB solution in the future.

# 10. Collecting, analysing, and publishing benchmarking data sets

This chapter focusses on using the previously presented benchmarking approaches to collect real-world data sets from different NFV scenarios and makes them available for other researchers, e.g., to use them to train machine learning (ML) models. This goes beyond the small data sets collected for the evaluations of the previous chapters and verifies the practicability of the presented solutions. This chapter is based on my paper [PSK19b], from which it contains figures and verbatim text copies. The softwarised network data zoo (SNDZoo) project, presented in this chapter, archives the collected data sets and is available online [PSK19a]. After discussing related work in Section 10.2, we first introduce the methodology and workflow used to collect the data sets in Section 10.3. Second, we present and analyse our data sets, containing millions of performance measurements collected from different real-world VNFs from the security, web, and 5G vertical (IoT) areas, in Section 10.4. In this section, we also show how to automatically generate models, i.e. NFV-PPs, that can be used by MANO optimisation algorithms, e.g., placement or scaling algorithms. We conclude in Section 10.5.

## 10.1. Introduction

To automate network management and to realise so-called zero touch network and service management (ZSM) [Mia+17], more and more ML and artificial intelligence (AI)-based network management solutions arise and claim to be able to manage and optimise different aspects of our networks [Wan+18]. Many of them focus on automated resource dimensioning for NFV scenarios, which try to optimise the amount of resources assigned to each involved VNF. To do so, they predict the upcoming network load as well as the performance a VNF achieves using a given amount of resources [Mij+17; Sun+18].

But ML-based solutions are always data-driven and do not only depend on programming code, which is a huge difference to legacy management and automation approaches. This means that ML approaches can only work efficiently if enough data is available to train and test the involved models, before they are put into production. Even if data is available in some custom

environments, e.g., in the form of volatile monitoring metrics, we are still missing a publicly available collection of open data sets that can be used to evaluate and compare different ML solutions and algorithms with each other. Such open data sets are a common tool in other communities, like image recognition [Den+09] or natural language processing [Maa+11], and accelerated the adoption of ML solutions in those domains.

We argue that the software networking community also needs such open data sets to simplify and streamline ML research and to improve the reproducibility of new ideas. To this end, we introduce the SNDZoo—an open repository to collect, host, and share software networking data sets. The SNDZoo is, to the best of our knowledge, the first effort to build such a central repository for softwarised network data. It specifically focuses on NFV and SDN performance data sets, collected through performance measurements of real-world network setups. These data sets go beyond existing collections of open networking data sets, such as topology data sets or traffic traces [Orl+10; Kni+11].

## 10.2. Related work

As in many other domains, ML recently found its way into the ICT sector and networking domain [Wan+18]. The use of ML is especially appealing for network management and optimisation use cases in softwarised networks. It can, for example, be used for NFV/SDN scenarios, which shall be highly automated to allow zero-touch network operations following DevOps methods [Kar+16; Mia+17]. Existing work focuses on, e.g., learning and predicting of service metrics, such as response time and frame rate [SPF17; SS18], scaling and resource dimensioning [Mij+17] as well as placement decisions [Sun+18]. But all of this work relies on custom data sets which might not even be publicly available. The available public data sets, such as [SPF17], are however not available in a common repository and thus are often hard to find. Our work improves this situation by offering a common repository to host and share data sets focusing on performance measurements of softwarised networks as well as the involved platforms and components. This also complements and goes beyond existing collections of open network data sets mainly focusing on network topology graphs, like [Kni+11; LK14; Orl+10].

To collect the presented data sets, automated solutions to benchmark NFV and SDN scenarios, including our own work [PK16b; PK17] presented in Chapter 8, can be used [RBR17; Cao+15; NSS18; Kha+18]. They are complementary to the work presented in this chapter.

## 10.3. Methodology & workflow

Collecting data sets from softwarised network scenarios is more than performing a handful of manual measurements on a testbed running in a lab. In fact, manual measurements should be avoided where possible to be able to (i) quickly and objectively reproduce the measurements, (ii) run a given measurement in a new environment, e.g., outside the lab, and (iii) make use of the fact that software-based networking scenarios can be automatically deployed, allowing to completely remove all manual steps from the process. On top of that, software-based networks usually offer a much higher degree of configuration freedom, e.g., in terms of virtual resources assigned to a VNF, resulting in many different configurations for which measurements can be and need to be performed.

We use our NFV benchmarking framework presented in Chapter 8 to implement such a fully-automated data collection methodology. A data collection setup consists of two main components. First, the setup needs an NFV benchmarking framework that is responsible to control, manage, and automate the measurement and collection process. Second, it needs one or multiple NFV platforms, consisting of MANO layer and NFVI, as presented in Figure 8.2 (Chapter 8). They are used to deploy and execute the experiment setups, including the deployment and execution of the SUT. This concept is independent of the technical realisation of the different components. It is, for example, possible to not only use our own benchmarking framework, but also other benchmarking solutions such as Gym [RBR17] or NFV Inspector [Kha+18], combined with different NFV platforms, such as vim-emu [PKV16; Peu17], OSM [ETS16c], or SONATA SP [SON15b], to collect data from different environments.

Using our own benchmarking framework (Chapter 8), we collected two kinds of metrics. First, the experiment metrics that are collected from the probes as well as the SUT at the end of an experiment, e.g., total number of processed packets. Those metrics are captured by collecting log files from the involved containers before termination. Second, we collect time series metrics using the Prometheus time series database [Lin12] controlled by our benchmarking framework. Prometheus periodically fetches all metrics, including the resource usage of the involved containers (using cAdvisor [Goo14]) as well as SUT-specific metrics as we detail in the following section.

## 10.4. Collecting, analysing, and publishing the first data sets

We collect eight initial data sets, focusing on the performance of real-world VNFs under different configurations to kick-off the SNDZoo project and to

test the presented methodology, workflow, and tools. All initial data sets are automatically collected and can be reproduced using our benchmarking tool. We want to highlight that the scope of the SNDZoo is not limited to performance data sets of single VNFs.

### 10.4.1. Experiment setup

To collect the data sets, we pick eight VNFs from three different categories as shown in Table 10.1: Security (IDS systems), web (load balancers, proxies), and IoT (MQTT brokers). This way, we not only have VNFs that transparently forward the traffic while passively analysing it (IDS systems), but also active VNFs that can modify the traffic (proxies). We also have scenarios (MQTT broker) that can be considered as examples for 5G vertical use cases, such as IoT, smart manufacturing, or industry 4.0 [Sch+19].

In the first category (`SEC01 - SEC03`), we benchmark three IDS VNFs, namely Suricata 4.0 [OIS09], Snort 2.9 [Cis16], and Snort 3.0 [Cis16]. Each IDS is configured as transparent layer 2 bridge and passively monitors the incoming traffic. To stimulate the VNFs, we use two publicly available traffic traces with small and big flows that are continuously replayed at maximum speed [KA00b]. During the benchmarking experiment, the VNFs are configured with different IDS rule sets, taken from [ET 19], and with different resource assignments, i.e., CPU time (10 % to 100 %) and memory (256 MB and 1024 MB). As a result, 80 different configurations are tested[1] and each configuration is repeated 20 times, resulting in 1,600 experiment runs, each testing a single configuration.

The second category (`WEB01 - WEB03`) represents web scenarios in which we test an Nginx 1.10.3 [NGI04] and a HAProxy 1.6.3 [HAP01] load balancer VNF as well as a Squid 3.5.12 [Squ96] proxy. The VNFs are placed between a source probe (user requests generated by Apache Bench [The93]) and a target probe (web server running Apache 2.0 [The93]). As shown in Table 10.1, 80 different configurations are executed, including small and large requests and different resource assignments, i.e., CPU time (10 % to 100 %) and memory (64 MB, 128 MB, 256 MB, and 512 MB).

In the third category (`IOT01 - IOT02`), the MQTT brokers Mosquitto 1.6.2 [Ecl09] and Emqx 3.1.0 [EMQ16] are tested using Malaria [Mal13], an MQTT load generator. The broker is placed between two probes running Malaria instances. One acting as publisher and the other is acting as subscriber allowing us to measure the end-to-end delay of MQTT messages. Besides different resource assignments for the broker VNF, Malaria is executed with different configurations, e.g., message sizes between 10 and 1000 bytes as well as two

[1] There are only 40 configurations in the case of Snort 3.0 because of the smaller rule set available for this version of Snort.

different MQTT QoS levels (1 and 2). Please note that the full details of all used configurations, versions, and workloads are published along with the data sets [PSK19a].

Table 10.1.: Overview of the eight VNF benchmarking data sets initially published in the SNDZoo [PSK19a]

| Name | Class | VNF | Probe | Tested Configurations | Repetitions | Experiment Metrics | Time Series Metrics | Total Exp. Runtime | Total data points |
|---|---|---|---|---|---|---|---|---|---|
| SEC01 | IDS Systems | Suricata | Traces | 80 | 20 | 280 | 157 | 71.1 h | 15.5M |
| SEC02 | | Snort 2.9 | Traces | 80 | 20 | 280 | 169 | 72.5 h | 16.7M |
| SEC03 | | Snort 3.0 | Traces | 40 | 20 | 281 | 593 | 35.8 h | 28.7M |
| WEB01 | Load balancers | Nginx | AB | 80 | 20 | 268 | 43 | 70.4 h | 4.6M |
| WEB02 | | HAProxy | AB | 80 | 20 | 268 | 43 | 70.2 h | 4.6M |
| WEB03 | Proxys | Squid | AB | 80 | 20 | 268 | 43 | 70.5 h | 4.6M |
| IOT01 | MQTT Broker | Mosquitto | Malaria | 80 | 20 | 275 | 90 | 71.6 h | 9.1M |
| IOT02 | | Emqx | Malaria | 80 | 20 | 275 | 109 | 115.8 h | 10.9M |

### 10.4.2. Data collection

To collect the presented data sets, we use the setup presented in Section 10.3 using vim-emu [PKV16] as NFV platform, which is executed on a machine with Intel(R) Xeon(R) W-2145 CPU at 3.70 GHz CPU, 32 GB of memory, running Linux kernel 4.4.0-142-generic[2]. In our experiments, the tested VNFs as well as the probes used to stimulate them are deployed as Docker containers. Each of these containers is always pinned to a single physical CPU core for better isolation between VNFs and probes [PK17].

We collect a large number of different experiment metrics for each tested configuration as well as a large number of time series metrics during experiment execution. More specifically, we collect 268 to 281 experiment metrics after each experiment and between 43 and 593 time series metrics during each experiment. This results in up to 474,400 collected time series records in data set `SEC03`[3], as shown in Table 10.1. Each of these records contains about 60 data points resulting from a collection frequency of 0.5 Hz and an experiment runtime of 120 s per configuration. Those numbers highly depend on the involved VNFs and the number of metrics they expose. The used Snort 3.0 VNF, for example, exposes more than 400 metrics that can be collected, e.g., packet counters for different protocol types.

The use of our automated NFV benchmarking framework allows to reproduce all presented experiments. To do so, nothing more is required than two Linux machines on which tng-bench and vim-emu are installed (see [Peu18c] for more detailed instructions). All involved VNFs can be downloaded from the SNDZoo repositories as pre-defined Docker containers [PSK19a] and used to re-run the experiments. We also publish the used experiment descriptors along with the data sets. The absolute performance numbers in those data sets obviously depend on the underlying hardware and will differ for new measurements performed in different environments. Generic aspects, however, will still be visible in the resulting data sets [PK16b], e.g., trends that can be identified between different VNF configurations.

### 10.4.3. Resulting data sets

The presented data sets contain between 4.6 and 28.7 million data points and are, for example, usable as training and testing data sets for different kinds of prediction or optimisation algorithms in the NFV domain. The collection process of each data set took between 35.8 h and 115.8 h, as shown in Table 10.1.

[2]The full hardware and software specifications of the used testbed are available as part of the SNDZoo repositories.

[3]Considering configurations, repetitions, and collected metrics results in: 40·20·593=474,400 time series records.

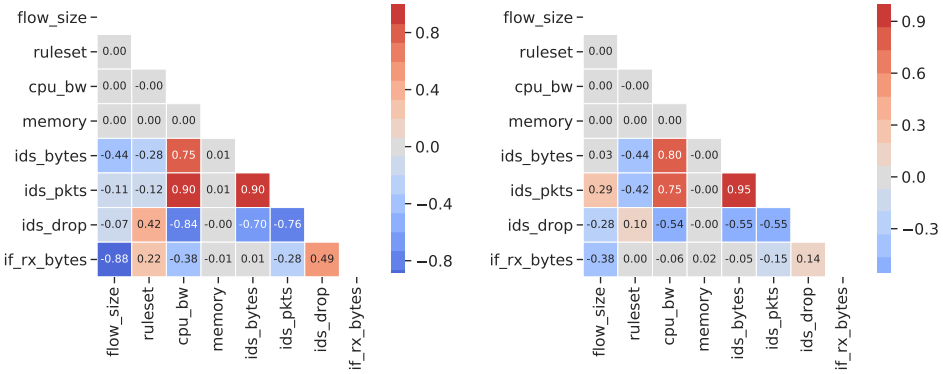
Even though the SEC03 data set has the shortest runtime, it contains the most data points. The reason for this is that the tested VNF, Snort 3.0, exposes more VNF-specific metrics than any other tested VNF. The measurements to collect IOT02 took more time than the others because the used VNF (Emqx) takes much longer to start up and to be ready to process traffic.

To get some first insights into the relationships between different parameters and metrics in the presented data sets, we investigate the correlation among them. Figure 10.1 presents matrices showing Pearson correlation coefficients for a selected subset of eight parameters and metrics from each data set. We skip the matrices for data sets SEC03 and WEB03 in the figure to keep it short and because they do not yield additional insights.

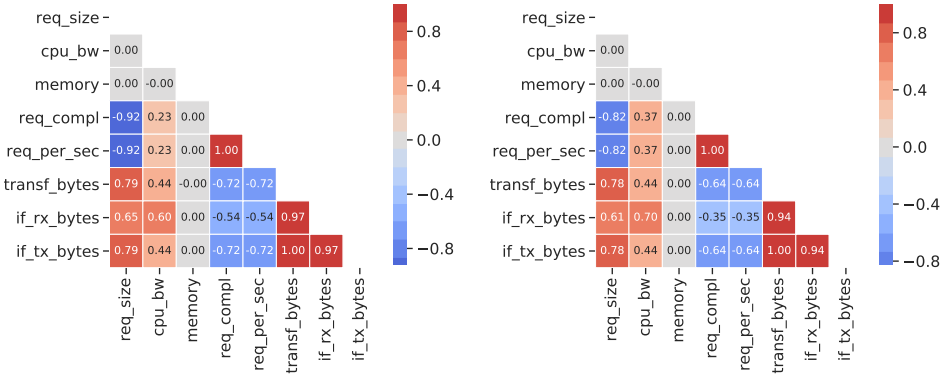
Figure 10.1a and Figure 10.1b present the results of the two security data sets captured from the Suricata and Snort 2.9 IDSs. They show that the chosen `rulset` has a direct impact on the number of packets processed and dropped (`ids_pkts` and `ids_drop`) by the IDS systems. It further shows that CPU resources (`cpu_bw`) for both VNFs are much more important than the amount of assigned memory. The CPU configuration shows the highest correlation to the main performance metrics (`ids_*`). Memory (`memory`), in contrast, has almost no effect on the resulting performance. The Snort 2.9 VNF is more sensitive to the size of the processed flows (`flow_size`) than the Suricata VNF. Similarly, the Snort 2.9 VNF is much less sensitive to the selected ruleset (see `ruleset` and `ids_drop`) compared to the Suricata VNF. This is an interesting insight for service developers that plan to use either of these VNFs as part of their service. A developer could, e.g., select the Snort 2.9 VNF because a growing ruleset will not result in a substantially growing number of missed packets. Such insights would be hard to obtain without the use of our presented benchmarking solutions.

The correlation matrices of the Nginx VNF (Figure 10.1c) and HAProxy VNF (Figure 10.1d) look both very similar, meaning that both VNFs will show comparable behaviour. The size of the requests (`req_size`) has a big impact to the resulting throughput (`transf_bytes`), where larger requests usually perform better. Again, CPU (`cpu_bw`) shows a strong positive correlation to the resulting performance, whereas memory does not have an impact at all. If more requests can be performed per second (`req_per_sec`), obviously more requests are successfully completed (`req_compl`), resulting in a strong correlation. The packet counters of the network interfaces (`if_rx_bytes` and `if_tx_bytes`) are directly correlated with the amount of data processed by the VNF implementations (`transf_bytes`), which shows that the VNF implementations do not drop many packets during processing.

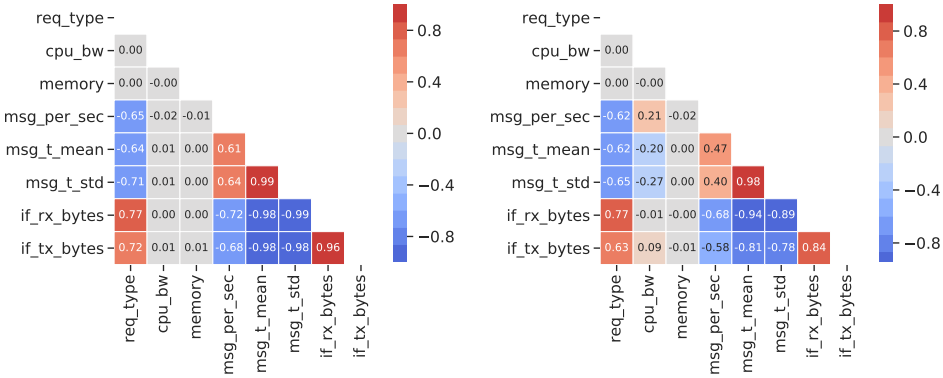
Finally, the IoT data sets show a direct relationship between the selected message type (`req_tpe`), e.g., message size or QoS level, and the resulting performance of the tested MQTT brokers (Figure 10.1e and Figure 10.1f). The




(a) SEC01 (Suricata [OIS09]) (b) SEC02 (Snort 2.9 [Cis16])




(c) WEB01 (Nginx [NGI04]) (d) WEB02 (HAProxy [HAP01])




(e) IOT01 (Mosquitto [Ecl09]) (f) IOT02 (Emqx [EMQ16])

Figure 10.1.: Matrices showing the Pearson correlation coefficients from a subset of metrics for different data sets

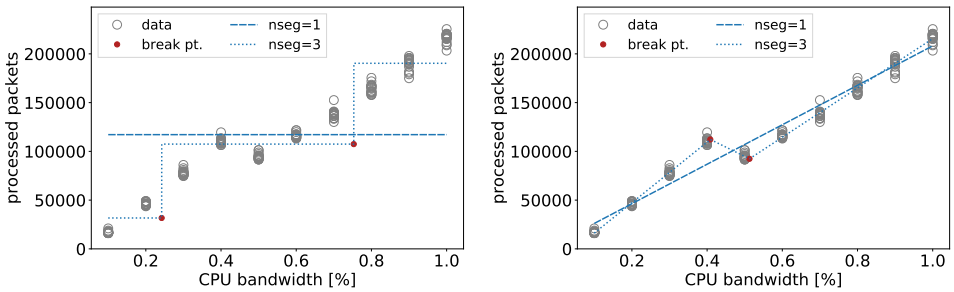
higher the QoS level, the lower is the number of messages that can be processed per second (`msg_per_sec`). The results show that the performance of the Mosquitto VNF is only marginally impacted by the selected CPU configuration. This indicates that we are not able to fully saturate the VNF during our experiments and underlines the efficient and lightweight implementation of Mosquitto. Emqx, in contrast, shows a stronger correlation between CPU configuration and resulting performance. One reason for this could be its Erlang-based implementation [EMQ16], which requires the Erlang runtime. Memory has again almost no impact to the performance of the two VNFs.

These results give insights into the nature of our data sets and show how they could be used to detect interrelationships between different parameters and metrics, which cannot easily be discovered without benchmarking. All this is done without knowing about the internals of the VNF implementations. These results can support developer decisions as well as MANO optimisation approaches.

### 10.4.4. Using the data sets

A typical use case for benchmarking data sets is their use within MANO systems to optimise resource dimensioning decisions, e.g., during initial deployment or scaling operations. However, using the raw data to automatically lookup the relationships between VNF configurations and the resulting performance by a MANO system, which has no further insights into the data, is complicated. A better solution for this is to represent the benchmarking results, i.e., NFV-PPs, by regression models. The benefit of this is that those models can be easily used by MANO systems and can provide (depending on the used model) a complete mapping between configuration space and performance values, even though not all of these configurations have been tested during benchmarking (see Chapter 9). Picking the right models, which are compatible to existing MANO optimisation solutions and show low regression errors, is not a simple task as we show in the following.

When looking at existing work in the field of MANO optimisation algorithms, like [MKK14; HB16; DKM18], it becomes clear that many of the presented optimisation algorithms rely on (mixed) integer linear programs (ILPs) using hardcoded constant or linear mappings between VNF configurations and performance metrics. To improve accuracy, some of the existing work extends this approach and use so-called "piecewise constant and linear functions" [DKM18; RBB16]. The benefit of these piecewise models is that they can better represent complex performance profiles while still keeping the models linear and as a result usable for ILPs. If those hardcoded representations should now be replaced by NFV-PPs that are automatically generated from our data sets,




(a) Piecewise constant (b) Piecewise linear

Figure 10.2.: Example that represents a single metric (processed packets) from the SEC01 data set using (piecewise) constant and linear functions with different numbers of segments (nseg) that can be easily reused by optimisation solutions, e.g., ILPs

we have to generate piecewise constant or linear regression models to be compatible with those optimisation approaches.

Figure 10.2 presents an example of these piecewise regression models, showing an NFV-PP based on our SEC01 data set representing the relationship between CPU time and processed packets using piecewise constant and linear functions. To fit this example, we use a Python library called PWLF [JV19] that relies on least squares regression [Golo4]. PWLF combines this with a global optimisation approach, using differential evolution [SP97], to automatically fit piecewise-linear models with a given number of segments to a data set and find the locations of the break points.

Figure 10.2a shows how the measured data can be represented by piecewise constant functions with either a single segment or with three segments (resulting in two break points). It becomes clear that in the constant case, more segments will usually result in more accurate models. Figure 10.2b shows two linear models with one and three segments. While the model that uses only a single segment fits the data already well (coefficient of determination $R^2 = 0.961$), the model with three segments fits even better ($R^2 = 0.995$). Looking at this examples, the question arises how many segments a model should use? An answer to this question is, however, always a trade-off between model accuracy as well as model complexity (number of segments). Using a constant number of segments for all models is also not a good idea, since the best number of segments heavily depends on the underlying data set. As a result, the decision must be taken on-the-fly during model generation. We present a fully-automated solution for this in the remainder of this section.

As a first step, to automatically optimise the number of used segments, we define a cost metric that takes the model quality as well as model complexity into account. We call this metric piecewise model cost (PMC). PMC depends on the number of segments $s$ used in the model, a penalty factor $\omega$, as well as

the normalised root-mean-squared deviation (NRMSD), which is a normalised error metric calculated over the number of samples $n$, as already introduced in Chapter 9. Based on this, we define the PMC as follows:

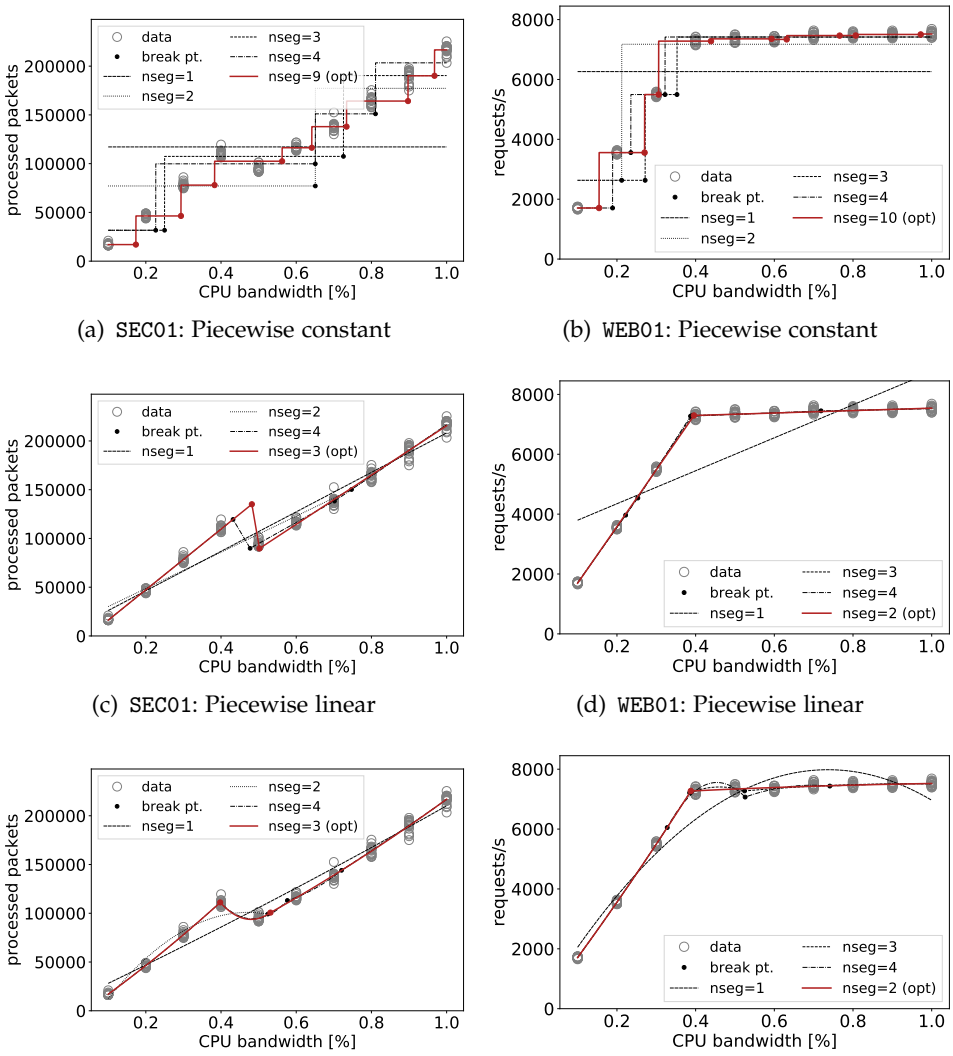
$$\mathrm{PMC}_\omega = \underbrace{\left(\frac{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2}}{\max(y) - \min(y)}\right)}_{\mathrm{NRMSD}} + s\omega \tag{10.1}$$

The idea behind this definition is to combine an error metric, the NRMSD, with a cost function that increases with the number of used segments ($s\omega$). By changing the penalty factor $\omega$, the tradeoff between model quality and model complexity can be fine-tuned, i.e., $\omega$ is used to control the impact of the number of used segments $s$ to the overall costs. By increasing $\omega$, the resulting models become less complex, meaning they use fewer segments. The assumption behind this is that, in general, the model quality tends to improve with the number of used segments, even though the improvements become smaller and smaller the more segments are used. To compensate for this and to limit the model's complexity, this tuneable cost function is added.

We then minimise the PMC metric to optimise the number of segments that should be used for the generated models. Figure 10.3 shows this process for two models that are based on the `SEC01` and `WEB01` data sets. Each subfigure shows the original data points (CPU time vs. processed packets or requests per second) and a set of models with different numbers of segments (nseg $\in [1, 2, 3, 4]$) as well as one model using the optimal number of segments (opt) according to the PMC. The figures also show the selected break point locations that are again computed using the PWLF Python library [JV19]. In all experiments, $\omega$ is set to 0.005, which turns out to work well and result in reasonable simple models with less than 10 segments.

The first set of results uses piecewise constant models that tend to require a higher number of segments to fit the used data sets. For example, the model shown in Figure 10.3a uses up to nine segments, which fits much better than the models with less segments. This can be seen in Figure 10.4a which reports the regression performance as MSE and coefficient of determination ($R^2$) of the models shown in Figure 10.3. The second result, shown in Figure 10.3b, selects ten segments for its optimal model which makes sense when comparing this to the reported error metrics in Figure 10.4b that show fluctuating MSE values until a certain number of segments is used.

In the second set of results, piecewise linear models are used. The results in Figure 10.3c and Figure 10.3d show that our proposed optimisation approach picks models with three and two segments that fit the data very well, indicating that it works as expected. This impression is clearly confirmed when looking at




(a) `SEC01`: Piecewise constant

(b) `WEB01`: Piecewise constant

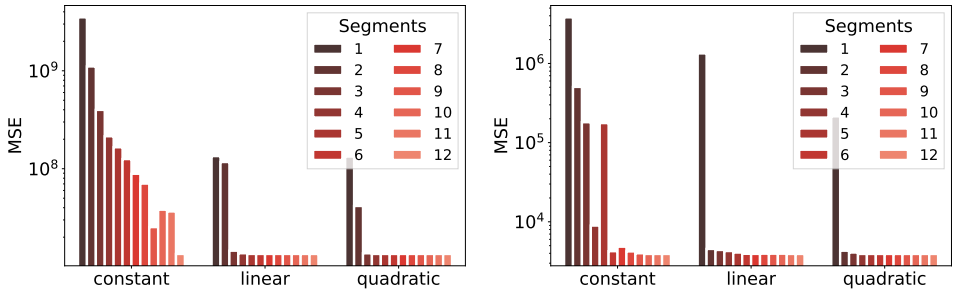
(c) `SEC01`: Piecewise linear

(d) `WEB01`: Piecewise linear

(e) `SEC01`: Piecewise quadratic

(f) `WEB01`: Piecewise quadratic

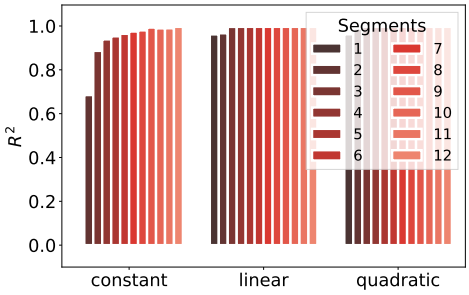
Figure 10.3.: Automatically optimised piecewise constant, linear, or quadratic fits to represent NFV-PPs



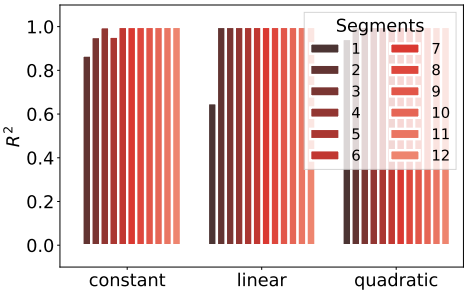

(a) `SEC01`: Mean squared error




(b) `WEB01`: Mean squared error




(c) `SEC01`: Coefficient of determination




(d) `WEB01`: Coefficient of determination

Figure 10.4.: Regression performance of models presented in Figure 10.3 for different model parameters and up to 12 segments used for the piecewise fits

the resulting regression performance reported in Figure 10.4a and Figure 10.4b. It can be seen that the MSE heavily drops at the selected number of segments and that it only marginally decreases if more segments are used.

A third set of results, depicted in Figure 10.3e and Figure 10.3f, shows the behaviour of our approach if quadratic models are used. These quadratic models are, however, of limited use for many MANO optimisation approaches and we analyse them just to show that our system is not limited to constant or linear models. Again, a relatively small number of segments (three and two) is selected which fits to the reported regression performance in Figure 10.4.

Using this automatic generation of piecewise constant, linear, and quadratic functions allows to directly use the NFV-PPs, resulting from the collected data sets, within existing MANO solutions. Especially the first two model types (constant and linear) are directly compatible and usable with existing MANO optimisation approaches [DKM18]. This demonstrates the usefulness of the concepts presented in Part III of this thesis for use during deployment and operation of network services. The results also show that we can automatically optimise the number of used segments and thus keep the complexity of the generated models low while not losing too much model accuracy.

### 10.4.5. Publishing the data sets

We version all experiment configurations that are used to collect the presented data sets using Git [Git05] so that we can reproduce old experiments even if they are updated or improved over time. This also allows us to use GitHub [Git19] as the hosting platform to publish and share the contents of the SNDZoo. However, keeping large data sets, containing multiple files with gigabytes of data, within a Git repository is bad practice and results in poor performance. To solve this, we make use of a recently introduced project called data version control (DVC) [DVC17]. DVC focuses specifically on versioning ML data sets and allows to store and version large files on external storage solutions such as Amazon S3 [Ama15], while referencing them from a Git repository. Users can then access and download a specific data set by simply running two commands (`git clone` and `dvc pull`) on their machine.

Each data set hosted in the SNDZoo is located in its own Git/DVC repository with a connected Amazon S3 bucket holding the data files (a total of 7.3 GB). Those repositories not only contain the configurations and resulting data sets but also meta data like the hardware and software specifications of the machines on which the measurements have been performed. Further, they contain license information as well as all raw measurements, time series, and logs produced by our benchmarking framework.

Besides the data set repositories, SNDZoo also provides repositories containing the sources and descriptions of used VNFs and services. All published data sets are indexed on and linked from the SNDZoo website [PSK19a]. They are published under creative commons CC-BY-SA 4.0 license.

## 10.5. Conclusion

We introduce the SNDZoo project as the first open collection of NFV performance data sets. The SNDZoo project aims to support the adoption of ML/AI in the software networking community by enabling researchers to work with common data sets simplifying comparisons and reproduction of results. We publish eight initial data sets, containing performance measurements collected from a set of real-world security, web, and IoT VNFs. We plan to continue our efforts and add new data sets over time. Our broader vision for the SNDZoo is to have a large collection of different data sets for a wide variety of use cases, scenarios, and deployments. Not only focusing on NFV and SDN performance measurements but also on measurements of MANO system and management performance.

Using these data sets, we show an automated model generation approach which can be combined with the automated data collection approach, presented in Chapter 8, to fully automate the end-to-end workflow between benchmarking at the development phase and MANO optimisations during the deployment and operation phase—an important contribution to close the DevOps loop for NFV services.

# 11. Final thoughts

This chapter summarises the key contributions of my thesis and draws final conclusions. Afterwards, future research directions are presented and briefly discussed.

## 11.1. Summary

In this thesis, I have investigated the gaps between development and deployment of softwarised network services and I have proposed a series of concepts, solutions, and supporting tools to close these gaps. The overall goal of my thesis has been to simplify the development and deployment of softwarised network services to accelerate the adoption of DevOps concepts in the ICT domain. This has been addressed within three different areas of work.

First, E-State, a solution to handle the state of VNFs within elastic deployments, has been presented in Chapter 3. E-State allows to migrate VNF-specific state between VNF instances. This migration is performed upon request when, e.g., a VNF is horizontally scaled. In contrast to existing solutions, E-State does not require a centralised management component and is able to perform state migrations directly between the VNF instances. It also allows each VNF instance to request the global state among all existing instances, which is a key feature to support seamless scaling of stateful applications, like IDSs. The presented results confirm that E-State outperforms approaches that rely on generic state sharing solutions and provides an efficient solution to run stateful VNFs in elastic deployments. E-State can be considered as supporting technology that simplifies both the development and the deployment of stateful network functions.

In contrast to existing approaches, and to not impose any strict requirements to the underlying NFV platform, E-State has not been designed to reroute traffic between VNF instances to balance flows within elastic deployments. Instead, I have introduced a second supporting technology, called SHarP, which closes this gap and can be combined with E-State, if needed. SHarP has been presented in Chapter 4. Even though it is designed to complement E-State, it can also be used as a standalone solution to enable seamless traffic rerouting in elastic VNF deployments. The presented results show that SHarP allows

to efficiently reroute flows between VNFs, while ensuring that no packets are dropped or re-ordered.

During the work on E-State and SHarP, I have noticed the general lack of prototyping solutions within the NFV community, motivating the second area of work presented in this thesis. With Containernet and vim-emu, I have presented a novel emulation-based prototyping solution to close this gap. The resulting platform enables NFV developers, for the first time, to quickly prototype complex NSs, built of a series of chained VNFs, in a lightweight environment, e.g., on a developer's laptop. Chapter 5 has presented the general concepts and abstractions used to locally emulate NFV infrastructure. The presented results demonstrate how the resource limitation functionalities of Linux `cgroups` can be reused to emulate environments that behave similar to resource-limited infrastructure, e.g., NFVI PoPs deployed at the network edge.

Chapter 6 has investigated extensions to this prototyping platform and aligned it with recent IETF standardisation activities by introducing support for NSH-based traffic steering, i.e., SFC. The presented case study shows how our prototyping platform can be used to implement complex SFC scenarios and test them in a local environment. The platform allows the assessment of such novel chaining technologies without requiring expensive testbeds—one of the key benefits of my solution.

The presented prototyping approach is not limited to VNFs and NSs but can also be used to prototype and test MANO systems as it has been shown in Chapter 7. The chapter has introduced a novel approach called "emulation-based smoke testing"; it extends the smoke testing concept by using emulation-based test environments to test MANO systems in realistic, large-scale scenarios. The results show how the presented solution can efficiently emulate arbitrary topologies with hundreds of NFVI PoPs and how it integrates with state-of-the-art MANO solutions. It can be considered a supporting technology for the development phase of VNFs, NSs, and MANO systems.

After VNFs have been implemented, NSs have been composed, and their prototypes have been tested within my prototyping platform, they are deployed to production. At this stage, resource allocation decisions are required to adequately dimension the deployed VNFs and NS to meet the agreed quality goals. This has motivated the third area of my work, in which I have used benchmarking to understand the behaviour of given (possibly proprietary) VNFs and/or NSs prior to their production deployment. Chapter 8 has presented a solution to fully automate this benchmarking procedure and to integrate it into existing NFV workflows and architectures. The results clearly showed that VNFs must not only be benchmarked in isolation but also as part of the NS to which they belong—something not considered in existing approaches.

The problem of performing automated benchmarking procedures under given time constraints has been investigated in Chapter 9. The main challenge in such scenarios are the large configuration spaces that need to be explored during the benchmarking process. The presented results show that it is possible to optimise such scenarios by benchmarking only a subset of configurations and using prediction approaches to extrapolate the resulting data.

Finally, I have used the developed benchmarking solutions to collect data sets from real-world VNFs in Chapter 10. After analysing those data sets and showing how they could be used to generate inputs for existing MANO optimisation algorithms, I have published them for use by the community. This has resulted in the first collection of open data sets to be used by researchers working on NFV optimisation solutions, such as placement and scaling algorithms. The presented benchmarking concepts, approaches, and tools can be considered as supporting technologies for the intersection between development, deployment, and operation of VNFs as well as NS and contribute to close the DevOps loop for NFV scenarios.

## 11.2. Conclusions

During the time of my research, network softwarisation has evolved from initial ideas and early prototypes to widely agreed concepts, protocols, standards, as well as technology stacks. Based on these developments, first field trails have been performed and more and more production deployments, often using open-source technology stacks, can be discovered. Nevertheless, developing, prototyping, and deploying new VNFs and NSs remains a challenging task. This thesis has addressed a broad scope of those challenges and has presented concepts and solutions to support developers at different stages of the development and deployment process.

State management is still a hot topic in the community. Even if there is a clear trend to design newly developed NFs as stateless components, following cloud-native or even server-less design principles, most of the NFs on the market are still based on legacy implementations which require some sort of state management solutions. With E-State and SHarP, I have presented two flexible solutions for this.

Existing prototyping solutions for the software networking domain were limited to SDN use cases. My work has changed this and has initiated two open-source projects, Containernet and vim-emu, which are now widely used by the NFV community[1]. I have also utilised these tools to build novel prototyping and testing concepts for the NFV domain. For example, Containernet has been, at the time of its initial release, the first container-based NFV platform that

[1]Containernet has, e.g., more than 120 stars and more than 70 forks on GitHub [Peu16].

supports chaining over multiple network interfaces. It has later evolved to the first prototyping platform with NSH support. Vim-emu itself is still the leading platform for testing MANO solutions in large-scale scenarios without requiring large testbeds. The user base of both tools drastically increased after vim-emu has been adopted by the OSM project, leading to hundreds of repository clones and several upstream contributions by the community. These projects have also been picked up and used by other researchers, e.g., to prototype P4 offloading, fog computing, or IoT scenarios [Mor+19b; Cou+18; Ngu+19]. All these aspects are important for the sustainability of my work and will ensure that those two projects will continue.

The domain of NFV benchmarking has gained more and more traction in the last years, especially as concepts like ZSM arose. My work has contributed in several ways to this research field. Most importantly, I have presented a benchmarking framework that allows for end-to-end automation of arbitrary NFV benchmarking scenarios and is now acting as one of the two reference implementations in our ongoing standardisation effort within IETF BMWG [Ros+18]. The presented approach to perform benchmarking under time constraints works well, but has its limits, i.e., benchmarking runs which only perform a handful of measurements will always produce results with limited accuracy. Finally, I have shown how my benchmarking framework can be used in practice by publishing open data sets, which are the first of their kind in the NFV community. Even though the amount of data sets is still limited, my thesis has introduced all required workflows and tools to collect, process, and publish additional data sets, paving the way to contribute new data sets collected by other researchers.

## 11.3. Future research

There are several promising research directions which could be explored to extend the presented work or further optimise the presented solutions.

**E-State: Reduced communication overhead** In the current prototype of E-State, every VNF subscribes to every other VNF using the broker-less ZeroMQ protocol. This works fine for many practical cases in which not more than a couple of dozen instances are deployed. Nevertheless, it means a quadratic overhead in terms of subscriptions that need to be established, certainly limiting the scalability of the prototype. There are two options to improve this. The first option is to replace ZeroMQ by a messaging approach that uses a centralised broker, e.g., Apache Kafka [The11]. This will reduce the subscription overheads since each VNF instance will only subscribe exactly once to a common topic on that broker. The downside of this approach is

the required broker, which needs to be deployed in addition to the VNFs, introducing additional management overhead and a single point-of-failure. Another option is to cluster the VNFs with the assumption that it is enough to retrieve the global view on the state only within a cluster of VNFs with a limited size. For example, a multi-instance IDS deployment could be clustered in a way that some IDS instances are responsible to only monitor a certain type of traffic, say all HTTP traffic, whereas another cluster of IDS instances only deals with voice over Internet protocol (VoIP) traffic. In this scenario, it should be sufficient that the HTTP instances only receive information from other HTTP instances, when, e.g., a global view on the matching counters of the HTTP-specific rules is required. Alternatively, geographical and/or topological clustering might be an option, assuming that state is mainly shared between flows coming from the same geographical location. Using such a clustering approach, several research questions arise, e.g., how to best cluster the VNFs or which cluster size to use?

**E-State: Additional consistency models** E-State implements an eventual consistency model that works well as shown in Chapter 3. There might be, however, specific VNFs that require stricter consistency models, e.g., to ensure that each VNF always accesses the latest state. Those stricter consistency models could be implemented on top of the presented prototype and be compared among each other. Further, it would be interesting to integrate this with a couple of real-world VNFs to see the impact of the selected consistency model to the performance of the VNFs (e.g., IDS detection rate).

**SHarP: DPDK-based HSL implementation** The presented prototype of SHarP uses a Python-based HSL implementation. Implementing this intermediate software layer, including its buffers and packet marking functionalities, with Python has been a good choice to quickly build the SHarP prototype and evaluate the general concept. For a production deployment, however, this prototype does not provide the required performance, as mentioned in Chapter 4. The right solution for this is to reimplement the HSL using some zero-copy high-performance packet processing technology, such as Intel's DPDK [Lin17]. Such an implementation would follow the exact same design as the existing prototype but would require substantial more engineering efforts.

**Containernet: Integration of other container technologies** The core idea behind Containernet is to integrate container technologies into Mininet-based network topologies to emulate realistic NFV scenarios. In its current version, Containernet uses Docker as underlying container technology, because it has been the most advanced container solution at the time the project has been started and it is still the most commonly used container platform, today.

Nevertheless, during the time of my research many more container technologies, such as linux containers (LXC) or Kata containers [Can08; Kat17], have emerged. It would be interesting to further extend Containernet and integrate such technologies. Especially the Kata container approach, which offers VM-like isolation, looks promising to replace some work I have done in collaboration with Johannes Kampmeyer to integrate full-featured VMs into Containernet [PKK18a]. This work has never been integrated into the master branch of Containernet as its dependencies and configuration requirements have been too complex, which would have limited the usability of the platform for normal users.

**Distributed vim-emu** Vim-emu already scales well and allows to emulate hundreds of PoPs on a single machine, as shown in Chapter 7. An option to push this further would be to distribute vim-emu across multiple machines. To build such a distributed version of vim-emu, Maxinet [Wet+14] can be used. Since Maxinet is already compatible with Containernet, it can run and control multiple Containernet instances on different machines and interconnect them using generic routing encapsulation (GRE) tunnels. There is no technical reason that prevents the integration of vim-emu and Maxinet. It will, however, require a major rewrite of the additional abstraction layers of vim-emu to implement a distributed version of it.

**Benchmarking: Update benchmarking results in production** The presented benchmarking approaches allow to gain insights into the behaviour of VNFs and NSs prior their production deployment. Once those VNFs and NSs are deployed and running in production, they are monitored by the MANO system which means additional data about real system load and resource usage is collected. An interesting research question in this scenario is whether and how it would be possible to combine the online monitoring data with benchmarking results to further improve the insights the system can gain.

**Benchmarking: Further selection algorithms and model enhancements for time-constrained scenarios** The concept of time-constrained benchmarking, presented in Chapter 9, compares several selection approaches and shows that even though the presented algorithm works well in general, it sometimes overfits or performs only slightly better than the other approaches. Other selection algorithms, like the adaptive decision tree approach which has been developed in a Bachelor thesis by Heidi Neuhäuser under my supervision [Neu19] are an interesting field for further research.

**Benchmarking: Standardised models** The key idea behind the automated benchmarking approach which has been presented in Chapter 8 is having a simple-to-use description approach to quickly define and model benchmarking experiments. The presented PED description approach achieves exactly this; however, it is still a custom solution that is only used by the benchmarking framework presented in this thesis. To change this and to get our description approach used by third-party benchmarking solutions, my collaborators and I are working towards a standardised description model as part of our activities within IETF BMWG [Ros+18]. The goal of this work and the work performed in the future is to propose and standardise a YANG-based description model for NFV benchmarking experiments that is based on the models presented in this thesis. This will simplify the application of the presented benchmarking concepts even further and can turn automated benchmarking solutions into a vital part of software-based network ecosystems and NFV DevOps processes.

# List of Acronyms

**3GPP** 3rd Generation Partnership Project
**AI** artificial intelligence
**API** application programming interface
**BMWG** benchmarking methodology working group
**BSS** business support system
**CAPEX** capital expense
**CDN** content delivery network
**CDU** cloud-native deployment unit
**CD** continuous delivery
**CFS** completely fair scheduler
**CI/CD** continuous integration/continuous delivery
**CI** continuous integration
**CLI** command line interface
**CNF** cloud-native network functions
**COTS** commercial off-the-shelf
**CPU** central processing unit
**CSAR** cloud service archive
**CU** compute unit
**DevOps** development and operation
**DHCP** dynamic host configuration protocol
**DPDK** data plane development kit
**DPI** deep packet inspection
**DTRP** decision tree regression
**DVC** data version control
**EM** element manager
**ETSI** European Communications Standards Institute
**FIFO** first-in, first-out
**FSM** function-specific manager
**Gbit/s** gigabit per second
**GRE** generic routing encapsulation
**GUI** graphical user interface
**HSL** handover support layer
**HTTP** hypertext transfer protocol
**IaaS** infrastructure as a service
**ICT** information and communication technology
**IDS** intrusion detection system

**IETF** Internet Engineering Task Force
**IFA** interfaces and architecture
**ILP** integer linear program
**IoT** Internet of Things
**IPC** inter-process communication
**IP** internet protocol
**ITZ** Internet topology zoo
**IT** information technology
**JSON** JavaScript object notation
**KVM** Kenrel-based Virtual Machine
**LCM** lifecycle management
**LLCM** lightweight lifecycle manager
**LRP** lasso regression
**MAC** medium access control
**MANO** management and orchestration
**MEC** multi-access edge computing
**MeDICINE** multi Datacenter servIce ChaIN Emulator
**MILP** mixed integer linear program
**ML** machine learning
**MP** measurement probe
**MQTT** message queuing telemetry transport
**MSE** mean-squared error
**NAT** network address translation
**NBI** northbound API
**NFVI** network function virtualisation infrastructure
**NFVO** NFV orchestrator
**NFV-PP** NFV performance profile
**NFV** network function virtualisation
**NF** network function
**NRMSD** normalised root-mean-squared deviation
**NSD** network service descriptor
**NSH** network service header
**NS** network service
**NUMA** non-uniform memory access
**ONAP** open network automation platform
**OPEX** operational expense
**OSM** OpenSource MANO
**OSS** operation support system
**OS** operating system
**OTT** over-the-top
**OVS** Open vSwitch
**OXM** extensible match
**PDF** probability density function
**PED** performance experiment descriptor

**PGAS** greedy adaptive sampling algorithm
**PMC** piecewise model cost
**PoP** point of presence
**PPS** packets per second
**QoE** quality of experience
**QoS** quality of service
**REST** representational state transfer
**RFC** request for comments
**RO** resource orchestrator
**RRP** ridge regression
**RSP** rendered service path
**RTT** round-trip time
**SDK** software development kit
**SDN** software defined networking
**SDO** standards developing organisation
**SFC** service function chaining
**SFF** service function forwarder
**SFP** service function path
**SF** service function
**SI** service index
**SLA** service level agreement
**SMS** short message service
**SNDZoo** softwarised network data zoo
**SOL** ETSI solution
**SPI** service path identifier
**SP** service platform
**SR-IOV** single-root input/output virtualisation
**SSH** secure shell
**SSM** service-specific manager
**SUT** system under test
**SVRPRK** support vector regression
**T-CB** time-constrained benchmarking
**TCP** transmission control protocol
**TG** traffic generator
**TLV** type-length-values
**TOSCA** topology and orchestration specification for cloud applications
**UDP** user datagram protocol
**URL** uniform resource locator
**URS** uniform random selection
**VCA** VNF configuration and abstraction
**vCDN** virtualised content delivery network
**vCPU** virtual CPU
**VDU** virtual deployment unit
**vim-emu** VIM Emulator

**VIM** virtual infrastructure manager
**VLAN** virtual LAN
**VLSP** very lightweight service platform
**VM** virtual machine
**VNFC** VNF container
**VNFD** virtual network function descriptor
**VNF-FG** VNF forwarding graph
**VNF-FP** VNF forwarding path
**VNFM** VNF manager
**VNF** virtual network function
**VoIP** voice over Internet protocol
**V&V** verification and validation
**WIM** wide area network infrastructure manager
**WRVS** weighted random VNF selection
**XML** extensible markup language
**YAML** YAML ain't markup language
**YANG** yet another next generation
**ZSM** zero touch network and service management

# List of Figures

# List of Tables

# List of Listings

# Bibliography


[3GP15] 3GPP. *TR32.842: Telecommunication management; Study on network management of virtualized networks*. Jan. 2015. URL: https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=2248 (visited on 02/28/2019) (cit. on p. 15).

[5GP18] 5G-PPP Software Network Working Group. *From Webscale to Telco, the Cloud Native Journey*. July 2018. URL: https://5g-ppp.eu/wp-content/uploads/2018/07/5GPPP-Software-Network-WG-White-Paper-July-2018.pdf (visited on 01/31/2019) (cit. on pp. 1, 18, 38, 70).

[5GT17a] 5GTANGO project consortium. *5GTANGO Development and Validation Platform for Global Industry-specific Network Services and Apps*. 2017. URL: https://5gtango.eu (visited on 12/06/2018) (cit. on pp. 7, 19, 138).

[5GT17b] 5GTANGO project consortium. *D2.2 Architecture Design*. Ed. by Manuel Peuster and Stefan Schneider. Nov. 2017. URL: https://www.5gtango.eu/documents/D22_v1.pdf (visited on 01/26/2018) (cit. on p. 19).

[5GT18a] 5GTANGO project consortium. *5GTANGO Package Specification*. Ed. by Manuel Peuster. Jan. 2018. URL: https://github.com/sonata-nfv/tng-schema/wiki/PkgSpec_LATEST (visited on 02/21/2019) (cit. on pp. 19, 75, 139).

[5GT18b] 5GTANGO project consortium. *D4.1 First open-source release of the SDK toolset*. Ed. by Wouter Tavernier. Apr. 2018. URL: http://5gtango.eu/documents/D41_v1.pdf (visited on 02/21/2019) (cit. on pp. 19, 75).

[Ahr+08] Jeff Ahrenholz et al. 'CORE: A real-time network emulator'. In: *2008 IEEE Military Communications Conference (MILCOM)*. San Diego, CA, USA: IEEE, Nov. 2008, pp. 1–7. DOI: 10.1109/MILCOM.2008.4753614 (cit. on p. 106).

[Ahr10] Jeff Ahrenholz. 'Comparison of CORE network emulation platforms'. In: *2010 IEEE Military Communications Conference (MILCOM)*. IEEE, Oct. 2010, pp. 166–171. DOI: 10.1109/MILCOM.2010.5680218 (cit. on p. 66).

[Ama15] Amazon. *Amazon Elastic Compute Cloud (Amazon EC2)*. 2015. URL: http://aws.amazon.com/de/ec2/ (visited on 03/01/2019) (cit. on pp. 23, 177).

[BBS16] Michael Till Beck, Juan Felipe Botero, and Kai Samelin. 'Resilient Allocation of Service Function Chains'. In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE, Nov. 2016, pp. 128–133. DOI: 10.1109/NFV-SDN.2016.7919487 (cit. on p. 90).

[Bes10] Sandford Bessler. 'Telco Service Delivery Platforms in the Last Decade - A R&D Perspective'. In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*. Berlin, Heidelberg: Springer, 2010, pp. 367–374. ISBN: 978-3-642-16558-0 (cit. on p. 1).

[BKR07] Steffen Becker, Heiko Koziolek, and Ralf Reussner. 'Model-Based Performance Prediction with the Palladio Component Model'. In: *Proceedings of the 6th International Workshop on Software and Performance*. WOSP '07. Buenes Aires, Argentina: ACM, 2007, pp. 54–65. ISBN: 1-59593-297-6. DOI: 10.1145/1216993.1217006 (cit. on pp. 126, 130).

[BKR09] Steffen Becker, Heiko Koziolek, and Ralf Reussner. 'The Palladio component model for model-driven performance prediction'. In: *Journal of Systems and Software* 82.1 (2009), pp. 3–22. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2008.03.066 (cit. on p. 130).

[Bon+12] Flavio Bonomi et al. 'Fog Computing and Its Role in the Internet of Things'. In: *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*. MCC '12. New York, NY, USA: ACM, Aug. 2012, pp. 13–16 (cit. on pp. 67, 78).

[Bos+14] Pat Bosshart et al. 'P4: Programming protocol-independent packet processors'. In: *ACM SIGCOMM Computer Communication Review* 44.3 (July 2014), pp. 87–95. DOI: 10.1145/2656877.2656890 (cit. on p. 67).

[BS15] Mario Baldi and Amedeo Sapio. 'A network function modeling approach for performance estimation'. In: *2015 IEEE 1st International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*. Turin, Italy: IEEE, Sept. 2015, pp. 527–533. DOI: 10.1109/RTSI.2015.7325152 (cit. on pp. 131, 152).

[Bui+13] Lars Buitinck et al. 'API design for machine learning software: experiences from the scikit-learn project'. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. Springer, Sept. 2013, pp. 108–122 (cit. on p. 157).

[BWZ15] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A software architect's perspective*. Addison-Wesley Professional, 2015 (cit. on pp. 3, 127).

[Cal+11] Rodrigo N Calheiros et al. 'CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms'. In: *Software: Practice and Experience* 41.1 (Aug. 2011), pp. 23–50. DOI: doi.org/10.1002/spe.995 (cit. on pp. 66, 91).

[Can08] Canonical Ltd. *LXC: Linux containers*. 2008. URL: https://linuxcontainers.org/ (visited on 03/06/2019) (cit. on p. 184).

[Can12] Canonical Ltd. *Juju Charms*. 2012. URL: https://www.jujucharms.com (visited on 02/13/2019) (cit. on pp. 20, 120).

[Cao+15] Lianjie Cao et al. 'NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions'. In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. San Francisco, CA, USA: IEEE, Nov. 2015, pp. 93–99. DOI: 10.1109/NFV-SDN.2015.7387412 (cit. on pp. 105, 126, 131, 132, 152, 164).

[Cas+15] C Cascone et al. 'OpenState: platform-agnostic behavioral (stateful) forwarding via minimal OpenFlow extensions'. In: *ACM Sigcom symposium on SDN research*. ACM. July 2015 (cit. on p. 30).

[Cat+16] Andrea F Cattoni et al. 'An end-to-end testing ecosystem for 5G'. In: *2016 European Conference on Networks and Communications (EuCNC)*. IEEE. June 2016, pp. 307–312. DOI: 10.1109/EuCNC.2016.7561053 (cit. on p. 105).

[Chi+12] Margaret Chiosi et al. *Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action*. Oct. 2012. URL: https://portal.etsi.org/NFV/NFV_White_Paper.pdf (visited on 12/02/2018) (cit. on pp. 1, 13, 14).

[Chr18] Frédéric Tobias Christ. 'Docker-based Emulation of Service Function Chains Using Network Service Header'. Bachelor's Thesis. Paderborn University, 2018 (cit. on pp. 9, 89, 92, 94, 97–99).

[Chu+03] Brent Chun et al. 'PlanetLab: An Overlay Testbed for Broad-coverage Services'. In: *SIGCOMM Comput. Commun. Rev.* 33.3 (July 2003), pp. 3–12. ISSN: 0146-4833. DOI: 10.1145/956993.956995 (cit. on p. 105).

[Cis16] Cisco Systems. *Snort IDS/IPS*. 2016. URL: http://www.snort.org (visited on 05/02/2019) (cit. on pp. 127, 166, 171).

[Cis19] Cisco Systems. *Cisco SDN Solution Overview*. 2019. URL: https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html (visited on 01/31/2019) (cit. on p. 12).

[Clo18] Cloudify Inc. *Cloudify*. 2018. URL: https://cloudify.co/ (visited on 02/22/2019) (cit. on p. 22).

[Cou+18] Antonio Coutinho et al. 'Fogbed: A Rapid-Prototyping Emulation Environment for Fog Computing'. In: *2018 IEEE International Conference on Communications (ICC)*. IEEE. Kansas City, MO, USA, May 2018, pp. 1–7. DOI: 10.1109/ICC.2018.8423003 (cit. on pp. 67, 182).

[CP18] Frédéric Tobias Christ and Manuel Peuster. *OSM vim-emu NSH prototyping platform*. 2018. URL: https://github.com/sonata-nfv/son-emu/tree/experimental/nsh-sfc (visited on 08/28/2019) (cit. on pp. 89, 94).

[Cso+14] Attila Csoma et al. 'ESCAPE: Extensible service chain prototyping environment using mininet, click, netconf and pox'. In: *SIGCOMM Computer Communication Review* 44.4 (Oct. 2014), pp. 125–126. DOI: 10.1145/2740070.2631448 (cit. on p. 91).

[Dat17] Linux Kernel Driver DataBase. *Network Service Header (NSH) protocol*. 2017. URL: https://cateee.net/lkddb/web-lkddb/NET_NSH.html (visited on 08/28/2019) (cit. on p. 95).

[Dav+17] Gianluca Davoli et al. 'Implementation of Service Function Chaining Control Plane through OpenFlow'. In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Berlin, Germany, Nov. 2017, 1–4. DOI: 10.1109/NFV-SDN.2017.8169852 (cit. on p. 90).

[Den+09] Jia Deng et al. 'Imagenet: A large-scale hierarchical image database'. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. Miami, FL, USA, June 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848 (cit. on p. 164).

[DKM18] Sevil Dräxler, Holger Karl, and Zoltán Ádám Mann. 'JASPER: Joint Optimization of Scaling, Placement, and Routing of Virtual Network Services'. In: *IEEE Transactions on Network and Service Management* 15.3 (Sept. 2018), pp. 946–960. ISSN: 1932-4537. DOI: 10.1109/TNSM.2018.2846572 (cit. on pp. 130, 172, 176).

[Doc13] Docker, Inc. *Docker: Enterprise Application Container Platofrm*. 2013. URL: https://www.docker.com (visited on 03/01/2019) (cit. on pp. 23, 72, 94, 108, 110).

[Drä+17] Sevil Dräxler et al. 'SONATA: Service programming and orchestration for virtualized software networks'. In: *2017 IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE. Paris, France, May 2017, pp. 973–978. DOI: 10.1109/ICCW.2017.7962785 (cit. on pp. 20, 22).

[Drä+18] Sevil Dräxler et al. 'Generating Resource and Performance Models for Service Function Chains: The Video Streaming Case'. In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Montreal, QC, Canada, June 2018, pp. 318–322. DOI: 10.1109/NETSOFT.2018.8460029 (cit. on p. 130).

[DRP99] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999 (cit. on pp. 102, 104, 105).

[DSK18] Sevil Dräxler, Stefan Schneider, and Holger Karl. 'Scaling and Placing Bidirectional Services with Stateful Virtual and Physical Network Functions'. In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Montreal, QC, Canada, June 2018, pp. 123–131. DOI: 10.1109/NETSOFT.2018.8459915 (cit. on p. 130).

[DVC17] DVC project. *DVC: Open-source Version Control System for Machine Learning Projects*. 2017. URL: https://dvc.org/ (visited on 05/02/2019) (cit. on p. 177).

[Eclo9] Eclipse Foundation. *Eclipse Mosquitto: An open source MQTT broker*. 2009. URL: https://mosquitto.org/ (visited on 05/02/2019) (cit. on pp. 166, 171).

[EMQ16] EMQ Technologies Co., Ltd. *EMQ: Scalable and Realtime MQTT Messaging for IoT in 5G Era*. 2016. URL: https://www.emqx.io/ (visited on 05/02/2019) (cit. on pp. 166, 171, 172).

[Enno6] Rob Enns. *NETCONF Configuration Protocol*. RFC 4741. IETF, 2006. URL: https://datatracker.ietf.org/doc/rfc4741/ (visited on 03/01/2019) (cit. on p. 12).

[Eri13] David Erickson. 'The Beacon OpenFlow Controller'. In: *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking (HotSDN)*. ACM. Hong Kong, China, Aug. 2013. DOI: 10.1145/2491185.2491189 (cit. on p. 13).

[ET 19] ET Labs. *Emerging Threats Open Ruleset*. 2019. URL: https://rules.emergingthreats.net/ (visited on 05/05/2019) (cit. on p. 166).

[ETS14a] ETSI. *Mobile-Edge Computing*. Sept. 2014. URL: https://portal.etsi.org/Portals/0/TBpages/MEC/Docs/Mobile-edge_Computing_-_Introductory_Technical_White_Paper_V1%2018-09-14.pdf (visited on 02/15/2019) (cit. on pp. 23, 78).

[ETS14b] ETSI. *NFV002v1.2.1: Network Functions Virtualisation (NFV): Architectural Framework*. Dec. 2014. URL: https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf (visited on 02/18/2019) (cit. on pp. 15, 16, 66, 103, 111).

[ETS15] ETSI. *INF001v1.1.1: Network Functions Virtualisation (NFV); Infrastructure Overview*. Aug. 2015. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-INF%20001v1.1.1%20-%20GS%20-%20Infrastructure%20Overview.pdf (visited on 02/28/2019) (cit. on p. 17).

[ETS16a] ETSI. *IFA009v1.1.1: Network Functions Virtualisation (NFV); Management and Orchestration; Report on Architectural Options*. July 2016. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-IFA%20009v1.1.1%20-%20GS%20-%20MANO%20architectural%20options%20report.pdf (visited on 02/28/2019) (cit. on pp. 17, 21).

[ETS16b] ETSI. *TST001v1.1.1: Network Functions Virtualisation (NFV); Predeployment Testing; Report on Validation of NFV Environments and Services*. Apr. 2016. URL: https://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/001/01.01.01_60/gs_NFV-TST001v010101p.pdf (visited on 02/28/2019) (cit. on p. 131).

[ETS16c] ETSI OSM. *Open Source MANO: Open Source NFV Management and Orchestration (MANO) software stack aligned with ETSI NFV*. 2016. URL: https://osm.etsi.org (visited on 12/06/2018) (cit. on pp. 7, 19, 20, 22, 66, 108, 110, 116, 134, 138, 165).

[ETS17a] ETSI. *NFV001v1.2.1: Network Functions Virtualisation (NFV); Use Cases*. 2017. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV%20001v1.2.1%20-%20GR%20-%20NFV%20Use%20Cases%20revision.pdf (visited on 02/28/2019) (cit. on p. 15).

[ETS17b] ETSI OSM. *OSM vim-emu code repository*. 2017. URL: https://osm.etsi.org/gitweb/?p=osm/vim-emu.git; (visited on 05/21/2019) (cit. on pp. 63, 87, 101).

[ETS18a] ETSI. *IFA007v3.1.1: Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Or-Vnfm reference point - Interface and Information Model Specification*. Aug. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-IFA%20007v3.1.1%20-%20GS%20-%20Or-Vnfm%

20ref%20point%20Spec.pdf (visited on 02/28/2019) (cit. on pp. 18, 21).

[ETS18b] ETSI. *IFA008v3.1.1: Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Ve-Vnfm reference point - Interface and Information Model Specification*. Aug. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-IFA%20008v3.1.1%20-%20GS%20-%20Ve-Vnfm%20ref%20point%20Spec.pdf (visited on 02/28/2019) (cit. on pp. 18, 21).

[ETS18c] ETSI. *IFA011v3.1.1: Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; VNF Descriptor and Packaging Specification*. Aug. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-IFA%20011v3.1.1%20-%20GS%20-%20VNF%20Packaging%20Spec.pdf (visited on 02/28/2019) (cit. on pp. 19, 138).

[ETS18d] ETSI. *IFA014v3.1.1: Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Network Service Templates Specification*. Aug. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-IFA%20014v3.1.1%20-%20GS%20-%20Network%20Service%20Templates%20Spec.pdf (visited on 02/28/2019) (cit. on p. 19).

[ETS18e] ETSI. *IFA015v3.1.1: Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Report on NFV Information Model*. Aug. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-IFA%20015v3.1.1%20-%20GR%20-%20Info%20Model%20Report.pdf (visited on 03/05/2019) (cit. on p. 19).

[ETS18f] ETSI. *SOL001v2.5.1: Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; NFV descriptors based on TOSCA specification*. Dec. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-SOL%20001v2.5.1%20-%20GS%20-%20TOSCA-based%20NFV%20descriptors%20spec.pdf (visited on 02/28/2019) (cit. on p. 19).

[ETS18g] ETSI. *SOL002v2.5.1: Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Ve-Vnfm Reference Point*. Dec. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-SOL%20002v2.5.1%20-%20GS%20-%20Ve-Vnfm%20RESTful%20protocols%20spec.pdf (visited on 02/28/2019) (cit. on p. 22).

[ETS18h] ETSI. *SOL004v2.5.1: Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; VNF Package specification*. Sept. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publicatio

ns_pdf/Specs-Reports/NFV-SOL%20004v2.5.1%20-%20GS%20-%20VNF%20Package%20Stage%203%20spec.pdf (visited on 02/28/2019) (cit. on pp. 19, 22).

[ETS18i] ETSI. *SOL005v2.5.1: Network Functions Virtualisation (NFV) Release 2; Protocols and Data Models; RESTful protocols specification for the Os-Ma-nfvo Reference Point*. Sept. 2018. URL: https://docbox.etsi.org/ISG/NFV/Open/Publications_pdf/Specs-Reports/NFV-SOL%20005v2.5.1%20-%20GS%20-%20Os-Ma-nfvo%20APIs.pdf (visited on 02/28/2019) (cit. on pp. 22, 107, 111).

[ETS18j] ETSI. *TST008v3.1.1: Network Functions Virtualisation (NFV) Release 3; Testing; NFVI Compute and Network Metrics Specification*. Aug. 2018. URL: https://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/008/03.01.01_60/gs_nfv-tst008v030101p.pdf (visited on 02/28/2019) (cit. on p. 17).

[ETS18k] ETSI. *TST009v3.1.1: Network Functions Virtualisation (NFV) Release 3; Testing; Specification of Networking Benchmarks and Measurement Methods for NFVI*. Oct. 2018. URL: https://www.etsi.org/deliver/etsi_gs/NFV-TST/001_099/009/03.01.01_60/gs_NFV-TST009v030101p.pdf (visited on 02/28/2019) (cit. on p. 152).

[ETS18l] ETSI OSM. *Open Source MANO Information Model*. 2018. URL: https://osm.etsi.org/wikipub/index.php/OSM_Information_Model (visited on 03/05/2019) (cit. on p. 19).

[ETS19] ETSI. *ETSI - Welcome to the World of Standards*. 2019. URL: https://www.etsi.org/ (visited on 02/28/2019) (cit. on p. 22).

[Fay+14] Seyed Kaveh Fayazbakhsh et al. 'Enforcing Network-wide Policies in the Presence of Dynamic Middlebox Actions Using Flowtags'. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 533–546. ISBN: 978-1-931971-09-6 (cit. on p. 42).

[Fed18] Fed4Fire+ project consortium. *Fed4Fire: The largest federation of testbeds in europe*. 2018. URL: https://www.fed4fire.eu (visited on 08/22/2019) (cit. on pp. 64, 105).

[Fra15] Frauenhofer Fokus. *OpenBaton*. 2015. URL: http://openbaton.github.io (visited on 02/22/2019) (cit. on p. 22).

[GA15] Aaron Gember-Jacobson and Aditya Akella. 'Improving the Safety, Scalability, and Efficiency of Network Function State Transfers'. In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. HotMiddlebox '15. London, United Kingdom: ACM, 2015, pp. 43–48. ISBN: 978-1-4503-3540-9. DOI: 10.1145/2785989.2785997 (cit. on pp. 29, 43).

[Gar+16] Jokin Garay et al. 'Service description in the NFV revolution: Trends, challenges and a way forward'. In: *IEEE Communications Magazine* 54.3 (Mar. 2016), pp. 68–74. ISSN: 0163-6804. DOI: 10.1109/MCOM.2016.7432174 (cit. on p. 18).

[Gem+12] Aaron Gember-Jacobson et al. 'Toward software-defined middlebox networking'. In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. ACM. New York, NY, USA, Oct. 2012, pp. 7–12. DOI: 10.1145/2390231.2390233 (cit. on p. 27).

[Gem+14] Aaron Gember-Jacobson et al. 'OpenNF: Enabling Innovation in Network Function Control'. In: *SIGCOMM Comput. Commun. Rev.* 44.4 (Oct. 2014), pp. 163–174. ISSN: 0146-4833. DOI: 10.1145/2740070.2626313 (cit. on pp. 28, 29, 32, 38, 41, 43, 47, 51).

[Gia+15] Ioannis Giannakopoulos et al. 'PANIC: modeling application performance over virtualized resources'. In: *2015 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. Tempe, AZ, USA, Mar. 2015, pp. 213–218. DOI: 10.1109/IC2E.2015.27 (cit. on pp. 131, 152, 155).

[Git05] Git SCM Project. *Git: Distributed is the new centralized*. 2005. URL: https://git-scm.com/ (visited on 06/17/2019) (cit. on p. 177).

[Git19] GitHub, Inc. *GitHub*. 2019. URL: https://github.com (visited on 06/17/2019) (cit. on p. 177).

[Golo4] Nikolai Golovchenko. *Least-squares fit of a continuous piecewise linear function*. Aug. 2004. URL: https://www.golovchenko.org/docs/ContinuousPiecewiseLinearFit.pdf (visited on 07/25/2019) (cit. on p. 173).

[Goo14] Google Inc. *Google cAdvisor*. 2014. URL: https://github.com/google/cadvisor (visited on 05/02/2019) (cit. on p. 165).

[GTK17] Ioannis Giannakopoulos, Dimitrios Tsoumakos, and Nectarios Koziris. 'A decision tree based approach towards adaptive modeling of big data applications'. In: *2017 IEEE International Conference on Big Data (Big Data)*. IEEE. Boston, MA, USA, Dec. 2017, pp. 163–172. DOI: 10.1109/BigData.2017.8257924 (cit. on pp. 153, 161).

[HAP01] HAProxy project. *HAProxy: The Reliable, High Performance TCP / HTTP Load Balancer*. 2001. URL: http://www.haproxy.org/ (visited on 05/02/2019) (cit. on pp. 166, 171).

[HB16] Juliver Gil Herrera and Juan Felipe Botero. 'Resource Allocation in NFV: A Comprehensive Survey'. In: *IEEE Transactions on Network and Service Management* 13.3 (Sept. 2016), pp. 518–532. ISSN: 1932-4537. DOI: 10.1109/TNSM.2016.2598420 (cit. on p. 172).

[Hen+08] Thomas R Henderson et al. 'Network simulations with the ns-3 simulator'. In: *SIGCOMM demonstration* 14 (2008), p. 527 (cit. on pp. 66, 91).

[HP15] Joel Halpern and Carlos Pignataro. *Service Function Chaining (SFC) Architecture*. RFC 7665. IETF, 2015. URL: https://datatracker.ietf.org/doc/rfc7665/ (visited on 08/19/2019) (cit. on pp. 15, 89, 92–94, 97).

[Htt05] Httping project. *Httping: Measure the lateceny and throughput of a webserver*. 2005. URL: https://linux.die.net/man/1/httping (visited on 05/02/2019) (cit. on p. 145).

[IEE17] IEEE 5G Initiative. *5G and Beyond Technology Roadmap*. Oct. 2017. URL: https://futurenetworks.ieee.org/images/files/pdf/ieee-5g-roadmap-white-paper.pdf (visited on 08/19/2019) (cit. on pp. 1, 2).

[iMa15] iMatix. *ZeroMQ Distributed Messaging*. 2015. URL: http://zeromq.org (visited on 05/21/2019) (cit. on p. 34).

[Jen11] Jenkins Project. *Jenkins: Build great things at scale*. 2011. URL: https://jenkins.io (visited on 08/22/2019) (cit. on pp. 107, 108).

[Joe13] Donald Eastlake amd Joe Abley. *IANA Considerations and IETF Protocol and Documentation Usage for IEEE 802 Parameters*. RFC 7042. IETF, 2013. URL: https://datatracker.ietf.org/doc/rfc7042/ (visited on 08/19/2019) (cit. on p. 51).

[JTS08] Dilip A Joseph, Arsalan Tavakoli, and Ion Stoica. 'A Policy-aware Switching Layer for Data Centers'. In: *ACM SIGCOMM Computer Communication Review* 38.4 (2008), pp. 51–62 (cit. on p. 42).

[Jun12] Juniper Networks. *Contrail: SDN-enabled management and control software for simplified service delivery*. 2012. URL: https://www.juniper.net/us/en/products-services/sdn/contrail/ (visited on 01/31/2019) (cit. on p. 12).

[JV19] Charles F. Jekel and Gerhard Venter. *pwlf: A Python Library for Fitting 1D Continuous Piecewise Linear Functions*. 2019. URL: https://github.com/cjekel/piecewise_linear_fit_py (visited on 07/25/2019) (cit. on pp. 173, 174).

[KA00a] Fred Klassen and AppNeta. *Tcpreplay - Pcap editing and replaying utilities*. 2000. URL: https://tcpreplay.appneta.com/ (visited on 08/05/2019) (cit. on p. 96).

[KA00b] Fred Klassen and AppNeta. *Tcpreplay: Sample Captures*. 2000. URL: http://tcpreplay.appneta.com/wiki/captures.html (visited on 05/05/2019) (cit. on p. 166).

[Kam17] Johannes Kampmeyer. 'A Hybrid Prototyping and Profiling Platform for NFV and Cloud Services'. Master's Thesis. Paderborn University, 2017 (cit. on p. 70).

[Kar+16] Holger Karl et al. 'DevOps for network function virtualisation: an architectural approach'. In: *Transactions on Emerging Telecommunications Technologies* 27.9 (July 2016), pp. 1206–1215. DOI: doi.org/10.1002/ett.3084 (cit. on pp. 3, 64, 150, 164).

[Kat17] Kata Containers project. *Kata Containers*. 2017. URL: https://katacontainers.io/ (visited on 07/25/2019) (cit. on p. 184).

[KDS15] Babu Kothandaraman, Manxing Du, and Pontus Sköldström. 'Centrally Controlled Distributed VNF State Management'. In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM. New York, NY, USA, Aug. 2015, pp. 37–42. DOI: 10.1145/2785989.2785996 (cit. on pp. 29, 43).

[Kha+18] Michel Gokan Khan et al. 'NFV-Inspector: A Systematic Approach to Profile and Analyze Virtual Network Functions'. In: *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*. IEEE. Tokyo, Japan, Oct. 2018, pp. 1–7. DOI: 10.1109/CloudNet.2018.8549333 (cit. on pp. 131, 164, 165).

[Kim+15] Juhoon Kim et al. 'Service provider DevOps for large scale modern network services'. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. Ottawa, ON, Canada, May 2015, pp. 1391–1397. DOI: 10.1109/INM.2015.7140502 (cit. on pp. 3, 127).

[Kni+11] Simon Knight et al. 'The Internet Topology Zoo'. In: *IEEE Journal on Selected Areas in Communications* 29.9 (Sept. 2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/JSAC.2011.111002 (cit. on pp. 74, 76, 109, 117, 164).

[Koh+00] Eddie Kohler et al. 'The Click modular router'. In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (Aug. 2000), pp. 263–297. DOI: 10.1145/354871.354874 (cit. on p. 67).

[Kou+18] Hadi Razzashi Kouchaksaraei et al. 'Programmable and Flexible Management and Orchestration of Virtualized Network Functions'. In: *2018 European Conference on Networks and Communications (EuCNC)*. Ljubljana, Slovenia, Slovenia: IEEE, June 2018, pp. 1–9. DOI: 10.1109/EuCNC.2018.8442528 (cit. on p. 102).

[KP18] Hannes Küttner and Manuel Peuster. *SHarP prototype repository*. 2018. URL: https://github.com/CN-UPB/sharp (visited on 05/21/2019) (cit. on p. 53).

[KRP13] Matthias Keller, Christoph Robbert, and Manuel Peuster. 'An Evaluation Testbed for Adaptive, Topology-aware Deployment of Elastic Applications'. In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. Hong Kong, China: ACM, 2013, pp. 469–470. ISBN: 978-1-4503-2056-6 (cit. on pp. 67, 68, 105).

[Küt17] Hannes Küttner. 'Seamless SDN-based handover for virtualized network functions'. Bachelor's Thesis. Paderborn University, 2017 (cit. on pp. 8, 41, 45, 49, 51, 53, 54, 56, 57, 59).

[Lam78] Leslie Lamport. 'Time, clocks, and the ordering of events in a distributed system'. In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. DOI: 10.1145/359545.359563 (cit. on p. 32).

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. 'A network in a laptop: rapid prototyping for software-defined networks'. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM. New York, NY, USA, Oct. 2010. DOI: 10.1145/1868447.1868466 (cit. on pp. 35, 66, 68, 91, 98, 106).

[Lin12] Linux Foundation. *Prometheus Time Series Database*. 2012. URL: https://prometheus.io/ (visited on 05/02/2019) (cit. on p. 165).

[Lin14a] Linux Foundation. *Kubernetes: Production-Grade Container Orchestration*. 2014. URL: https://kubernetes.io/ (visited on 03/01/2019) (cit. on p. 24).

[Lin14b] Linux Foundation. *Open Network Operating System (ONOS)*. 2014. URL: https://onosproject.org/ (visited on 02/15/2019) (cit. on pp. 12, 13, 23).

[Lin16] Linux Foundation. *OPNFV Project*. 2016. URL: https://www.opnfv.org (visited on 08/22/2019) (cit. on pp. 23, 105).

[Lin17] Linux Foundation. *Data Plane Development Kit (DPDK)*. 2017. URL: http://dpdk.org (visited on 03/01/2019) (cit. on pp. 23, 47, 53, 183).

[Lin18a] Linux Foundation. *ONAP: Open Network Automation Platform*. 2018. URL: https://www.onap.org (visited on 12/06/2018) (cit. on pp. 19, 22, 120).

[Lin18b] Linux man-pages project. *Linux Programmer's Manual: veth(4) - Virtual Ethernet Device*. 2018. URL: http://man7.org/linux/man-pages/man4/veth.4.html (visited on 03/08/2019) (cit. on p. 69).

[Liu+17] Junjie Liu et al. 'On Dynamic Service Function Chain Deployment and Readjustment'. In: *IEEE Transactions on Network and Service Management* 14.3 (June 2017), pp. 543–553. DOI: 10.1109/TNSM.2017.2711610 (cit. on p. 90).

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. June 2014. URL: http://snap.stanford.edu/data (visited on 05/02/2019) (cit. on p. 164).

[LLJ14] Jiaqiang Liu, Yong Li, and Depeng Jin. 'SDN-based Live VM Migration Across Datacenters'. In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. Chicago, Illinois, USA: ACM, 2014, pp. 583–584. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2631431 (cit. on p. 43).

[Maa+11] Andrew L. Maas et al. 'Learning Word Vectors for Sentiment Analysis'. In: *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Portland, Oregon, USA: Association for Computational Linguistics, June 2011, pp. 142–150 (cit. on p. 164).

[Mal13] Malaria Project. *Malaria: Attacking MQTT systems with Mosquittos*. 2013. URL: https://github.com/etactica/mqtt-malaria (visited on 05/02/2019) (cit. on p. 166).

[Mar+14] Joao Martins et al. 'ClickOS and the Art of Network Function Virtualization'. In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 459–473. ISBN: 978-1-931971-09-6 (cit. on p. 67).

[May+17] Ruben Mayer et al. 'FogStore: toward a distributed data store for fog computing'. In: *2017 IEEE Fog World Congress (FWC)*. IEEE. Santa Clara, CA, USA, May 2017, pp. 1–6. DOI: 10.1109/FWC.2017.8368524 (cit. on p. 43).

[McC96] Steve McConnell. 'Daily build and smoke test'. In: *IEEE Software* 13.4 (July 1996), pp. 144–143. DOI: 10.1109/MS.1996.10017 (cit. on p. 104).

[MCG15] Lefteris Mamatas, Stuart Clayman, and Alex Galis. 'A service-aware virtualized software-defined infrastructure'. In: *IEEE Communications Magazine* 53.4 (Apr. 2015), pp. 166–174. DOI: 10.1109/MCOM.2015.7081091 (cit. on pp. 66, 68, 91, 106).

[McK+08] Nick McKeown et al. 'OpenFlow: Enabling Innovation in Campus Networks'. In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Apr. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746 (cit. on pp. 12, 92).

[Med+16] Ahmed M Medhat et al. 'Resilient Orchestration of Service Functions Chains in a NFV Environment'. In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Palo Alto, CA, USA, Nov. 2016. DOI: 10.1109/NFV-SDN.2016.7919468 (cit. on p. 90).

[Med+17] Ahmed M Medhat et al. 'Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges'. In: *IEEE Communications Magazine* 55.2 (Oct. 2017), pp. 216–223. DOI: 10.1109/MCOM.2016.1600219RP (cit. on p. 89).

[Mia+17] Jie Miao et al. *Zero-touch Network and Service Management*. Dec. 2017. URL: https://portal.etsi.org/TBSiteMap/ZSM/OperatorWhitePaper (visited on 05/02/2019) (cit. on pp. 163, 164).

[Mij+17] Rashid Mijumbi et al. 'Topology-Aware Prediction of Virtual Network Function Resource Requirements'. In: *IEEE Transactions on Network and Service Management* 14.1 (Mar. 2017), pp. 106–120. ISSN: 1932-4537. DOI: 10.1109/TNSM.2017.2666781 (cit. on pp. 163, 164).

[MKK14] Sevil Mehraghdam, Matthias Keller, and Holger Karl. 'Specifying and Placing Chains of Virtual Network Functions'. In: *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. IEEE. Luxembourg, Luxembourg, Dec. 2014, pp. 7–13. DOI: 10.1109/CloudNet.2014.6968961 (cit. on pp. 20, 101, 172).

[Mor+19a] Daniele Moro et al. 'Demonstrating FOP4: A Flexible Platform to Prototype NFV Offloading Scenarios'. In: *2019 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. Dallas, Texas, USA, Nov. 2019 (cit. on p. 67).

[Mor+19b] Daniele Moro et al. 'FOP4: Function Offloading Prototyping in Heterogeneous and Programmable Network Scenarios'. In: *2019 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. Dallas, Texas, USA, 2019 (cit. on pp. 67, 182).

[Mor17] Al Morton. *Considerations for Benchmarking Virtual Network Functions and Their Infrastructure*. RFC 8172. IETF, July 2017. URL: https://datatracker.ietf.org/doc/rfc8172/ (visited on 08/19/2019) (cit. on pp. 4, 126, 131).

[Nau+16] Bram Naudts et al. 'Deploying SDN and NFV at the speed of innovation: toward a new bond between standards development organizations, industry fora, and open-source software projects'. In: *IEEE Communications Magazine* 54.3 (Mar. 2016), pp. 46–53. ISSN: 0163-6804. DOI: 10.1109/MCOM.2016.7432171 (cit. on p. 14).

[Neu19] Heidi Neuhäuser. 'Using Machine Learning to Optimize Time-Constrained Network Service Profiling'. Bachelor's Thesis. Paderborn University, 2019 (cit. on pp. 153, 161, 184).

[NGI04] NGINX Inc. *NGINX: High Performance Load Balancer, Web Server, Reverse Proxy*. 2004. URL: https://www.nginx.com/ (visited on 05/02/2019) (cit. on pp. 139, 157, 166, 171).

[Ngu+19] Tri Gia Nguyen et al. 'SeArch: A Collaborative and Intelligent NIDS Architecture for SDN-Based Cloud IoT Networks'. In: *IEEE Access* 7 (2019), pp. 107678–107694. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2019.2932438 (cit. on p. 182).

[NL17] Thuy Linh Nguyen and Adrien Lebre. 'Virtual Machine Boot Time Model'. In: *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. St. Petersburg, Russia, Apr. 2017, pp. 430–437. DOI: 10.1109/PDP.2017.58 (cit. on pp. 150, 151).

[Nob+17] Leonhard Nobach et al. 'Statelet-Based Efficient and Seamless NFV State Transfer'. In: *IEEE Transactions on Network and Service Management* PP.99 (Oct. 2017), pp. 1–1. ISSN: 1932-4537. DOI: 10.1109/TNSM.2017.2760107 (cit. on pp. 29, 43).

[NSS18] Jaehyun Nam, Junsik Seo, and Seungwon Shin. 'Probius: Automated Approach for VNF and Service Chain Analysis in Software-Defined NFV'. In: *Proceedings of the Symposium on SDN Research*. SOSR '18. Los Angeles, CA, USA: ACM, Mar. 2018, 14:1–14:13. ISBN: 978-1-4503-5664-0. DOI: 10.1145/3185467.3185495 (cit. on pp. 152, 154, 164).

[Oas13] Oasis TOSCA. *TOSCA Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html (visited on 03/05/2019) (cit. on p. 19).

[Oas16] Oasis TOSCA. *TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0*. Mar. 2016. URL: http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/csd03/tosca-nfv-v1.0-csd03.html (visited on 03/05/2019) (cit. on p. 19).

[OIS09] Open Information Security Foundation (OISF). *Suricata: Open Source IDS / IPS / NSM engine*. 2009. URL: https://suricata-ids.org/ (visited on 05/02/2019) (cit. on pp. 166, 171).

[OMN05] OMNeT++ Project. *OMNeT++ Discrete Event Simulator*. 2005. URL: https://omnetpp.org (visited on 12/07/2018) (cit. on p. 137).

[Ope10a] OpenStack Foundation. *DevStack*. 2010. URL: https://docs.openstack.org/devstack/latest/ (visited on 05/16/2019) (cit. on pp. 67, 68, 113).

[Ope10b] OpenStack Foundation. *OpenStack*. 2010. URL: https://www.openstack.org/ (visited on 02/15/2019) (cit. on pp. 12, 23, 75, 108, 110).

[Ope10c] OpenStack Foundation. *OpenStack Compute (Nova)*. 2010. URL: https://docs.openstack.org/nova/ (visited on 03/19/2019) (cit. on p. 70).

[Ope10d] OpenStack Foundation. *OpenStack Orchestration (Heat)*. 2010. URL: https://docs.openstack.org/heat/ (visited on 03/19/2019) (cit. on p. 131).

[Ope11] Open Networking Foundation (ONF). *OpenFlow Switch Specification Version 1.2*. Dec. 2011. URL: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.2.pdf (visited on 04/10/2019) (cit. on p. 92).

[Ope12] Open Networking Foundation (ONF). *Software-Defined Networking: The New Norm for Networks*. Apr. 2012. URL: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf (visited on 02/13/2019) (cit. on pp. 11, 12).

[Ope16] OpenStack Foundation. *OpenStack Nova Scaling Guide*. 2016. URL: https://docs.openstack.org/operations-guide/ops-capacity-planning-scaling.html (visited on 08/22/2019) (cit. on p. 82).

[Ope17] OpenStack Foundation. *OpenStack Container Infrastructure Management service (Magnum)*. 2017. URL: https://wiki.openstack.org/wiki/Magnum (visited on 02/15/2019) (cit. on p. 24).

[Ope18] OpenStack Foundation. *OpenStack Neutron SFC*. 2018. URL: https://docs.openstack.org/newton/networking-guide/config-sfc.html (visited on 08/22/2019) (cit. on p. 95).

[OR12] Vladimir Andrei Olteanu and Costin Raiciu. 'Efficiently migrating stateful middleboxes'. In: *ACM SIGCOMM Computer Communication Review*. Vol. 42. 4. ACM. Oct. 2012, pp. 93–94. DOI: 10.1145/2377677.2377697 (cit. on pp. 28, 29).

[Orl+10] Sebastian Orlowski et al. 'SNDlib 1.0—Survivable Network Design Library'. In: *Networks* 55.3 (Oct. 2010), pp. 276–286. DOI: 10.1002/net.20371 (cit. on p. 164).

[Par+18] Carlos Parada et al. '5GTAGNO: A Beyond-Mano Service Platform'. In: *2018 European Conference on Networks and Communications (EuCNC)*. IEEE. Ljubljana, Slovenia, Slovenia, June 2018, pp. 26–30. DOI: 10.1109/EuCNC.2018.8443232 (cit. on pp. 19, 102).

[Pel+15] István Pelle et al. 'One Tool to Rule Them All: A Modular Troubleshooting Framework for SDN (and Other) Networks'. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. Santa Clara, California: ACM, June 2015, 24:1–24:7. ISBN: 978-1-4503-3451-8. DOI: 10.1145/2774993.2775014 (cit. on pp. 66, 91).

[Peu+17] Manuel Peuster et al. 'A flexible multi-pop infrastructure emulator for carrier-grade MANO systems'. In: *2017 3rd IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Bologna, Italy, July 2017, pp. 1–3. DOI: 10.1109/NETSOFT.2017.8004250 (cit. on pp. 5, 75).

[Peu+18a] Manuel Peuster et al. 'A Prototyping Platform to Validate and Verify Network Service Header-based Service Chains'. In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Verona, Italy, Italy, Nov. 2018, pp. 1–5. DOI: 10.1109/NFV-SDN.2018.8725614 (cit. on pp. 6, 9, 89, 90).

[Peu+18b] Manuel Peuster et al. 'Emulation-based Smoke Testing of NFV Orchestrators in Large Multi-PoP Environments'. In: *2018 European Conference on Networks and Communications (EuCNC)*. IEEE. Ljubljana, Slovenia, Slovenia, June 2018, pp. 1–9. DOI: 10.1109/EuCNC.2018.8442701 (cit. on pp. 6, 9, 74, 91, 101, 113).

[Peu+19a] Manuel Peuster et al. 'Automated testing of NFV orchestrators against carrier-grade multi-PoP scenarios using emulation-based smoke testing'. In: *EURASIP Journal on Wireless Communications and Networking* 2019.1 (June 2019), p. 172. ISSN: 1687-1499. DOI: 10.1186/s13638-019-1493-2 (cit. on pp. 6, 9, 101).

[Peu+19b] Manuel Peuster et al. 'Introducing Automated Verification and Validation for Virtualized Network Functions and Services'. In: *IEEE Communications Magazine* 57.5 (May 2019), pp. 96–102. ISSN: 0163-6804. DOI: 10.1109/MCOM.2019.1800873 (cit. on pp. 7, 105, 139, 147).

[Peu+19c] Manuel Peuster et al. 'Joint testing and profiling of microservice-based network services using TTCN-3'. In: *ICT Express* 5.2 (June 2019), pp. 150–153. ISSN: 2405-9595. DOI: https://doi.org/10.1016/j.icte.2019.02.001 (cit. on p. 7).

[Peu+19d] Manuel Peuster et al. 'Prototyping and Demonstrating 5G Verticals: The Smart Manufacturing Case'. In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. Paris, France, June 2019, pp. 236–238. DOI: 10.1109/NETSOFT.2019.8806685 (cit. on pp. 6, 75, 101).

[Peu15] Manuel Peuster. *E-State prototype repository*. 2015. URL: https://github.com/mpeuster/estate (visited on 05/21/2019) (cit. on p. 34).

[Peu16] Manuel Peuster. *Containernet a Mininet Fork adding Container Support to Network Emulations*. 2016. URL: https://containernet.github.io (visited on 03/06/2019) (cit. on pp. 7, 53, 63, 68, 69, 98, 109, 181).

[Peu17] Manuel Peuster. *OSM vim-emu documentation*. 2017. URL: https://osm.etsi.org/wikipub/index.php/VIM_emulator (visited on 05/21/2019) (cit. on pp. 7, 72, 165).

[Peu18a] Manuel Peuster. *ETSI NFV SOL005 Test Suite*. 2018. URL: https://github.com/mpeuster/etsi-nfv-sol005-test-suite (visited on 08/22/2019) (cit. on p. 111).

[Peu18b] Manuel Peuster. *nfv-t-cp: NFV Time-Constrained Profiling Framework*. 2018. URL: https://github.com/CN-UPB/nfv-t-cp (visited on 08/22/2019) (cit. on pp. 149, 157, 161).

[Peu18c] Manuel Peuster. *tng-bench: Automated Benchmarking of NFV Scenarios*. 2018. URL: https://github.com/sonata-nfv/tng-sdk-benchmark (visited on 12/06/2018) (cit. on pp. 125, 169).

[PK16a] Manuel Peuster and Holger Karl. 'E-State: Distributed state management in elastic network function deployments'. In: *2016 2nd IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Seoul, South Korea, June 2016, pp. 6–10. DOI: 10.1109/NETSOFT.2016.7502432 (cit. on pp. 5, 8, 27, 30, 43, 45).

[PK16b] Manuel Peuster and Holger Karl. 'Understand Your Chains: Towards Performance Profile-Based Network Service Management'. In: *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*. IEEE. The Hague, Netherlands, Oct. 2016, pp. 7–12. DOI: 10.1109/EWSDN.2016.9 (cit. on pp. 6, 9, 125, 127, 131, 140, 164, 169).

[PK17] Manuel Peuster and Holger Karl. 'Profile your chains, not functions: Automated network service profiling in DevOps environments'. In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Berlin, Germany, Nov. 2017, pp. 1–6. DOI: 10.1109/NFV-SDN.2017.8169826 (cit. on pp. 6, 9, 105, 125, 151, 152, 154, 157, 164, 169).

[PK18] Manuel Peuster and Holger Karl. 'Understand Your Chains and Keep Your Deadlines: Introducing Time-constrained Profiling for NFV'. In: *2018 IEEE/IFIP 14th International Conference on Network and Service Management (CNSM)*. IEEE. Rome, Italy, Nov. 2018, pp. 240–246 (cit. on pp. 6, 9, 149, 160).

[PKK18a] Manuel Peuster, Johannes Kampmeyer, and Holger Karl. 'Containernet 2.0: A Rapid Prototyping Platform for Hybrid Service Function Chains'. In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Montreal, QC, Canada, June 2018, pp. 335–337. DOI: 10.1109/NETSOFT.2018.8459905 (cit. on pp. 6, 70, 184).

[PKK18b] Manuel Peuster, Hannes Küttner, and Holger Karl. 'Let the state follow its flows: An SDN-based flow handover protocol to support state migration'. In: *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE. Montreal, QC, Canada, June 2018, pp. 97–104. DOI: 10.1109/NETSOFT.2018.8460007 (cit. on pp. 5, 8, 41).

[PKK19] Manuel Peuster, Hannes Küttner, and Holger Karl. 'A flow handover protocol to support state migration in softwarized networks'. In: *International Journal of Network Management* 29.4 (Apr. 2019), e2067. DOI: 10.1002/nem.2067 (cit. on pp. 5, 8, 41, 50, 52).

[PKV16] Manuel Peuster, Holger Karl, and Steven Van Rossem. 'MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments'. In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Palo Alto, CA, USA, Nov. 2016, pp. 148–153. DOI: 10.1109/NFV-SDN.2016.7919490 (cit. on pp. 5, 8, 63, 66, 68, 109, 140, 165, 169).

[Pon+19] Salvatore Pontarelli et al. 'Flowblaze: stateful packet processing in hardware'. In: *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association. Boston, MA, USA, Feb. 2019, pp. 531–547 (cit. on p. 30).

[Pro11] Project Floodlight. *Project Floodlight*. 2011. URL: http://www.projectfloodlight.org (visited on 02/15/2019) (cit. on p. 13).

[PSK19a] Manuel Peuster, Stefan Schneider, and Holger Karl. *Softwarised Network Data Zoo*. 2019. URL: https://sndzoo.github.io (visited on 05/05/2019) (cit. on pp. 7, 163, 167–169, 177).

[PSK19b] Manuel Peuster, Stefan Schneider, and Holger Karl. 'The Softwarised Network Data Zoo'. In: *2019 IEEE/IFIP 15th International Conference on Network and Service Management (CNSM)*. IEEE. Halifax, Canada, Oct. 2019 (cit. on pp. 5, 7, 10, 163).

[Qaz+13] Zafar Ayyub Qazi et al. 'SIMPLE-fying Middlebox Policy Enforcement Using SDN'. In: *SIGCOMM Comput. Commun. Rev.* 43.4 (Oct. 2013), pp. 27–38. ISSN: 0146-4833. DOI: 10.1145/2534169.2486022 (cit. on p. 42).

[QEP18] Paul Quinn, Uri Elzur, and Carlos Pignataro. *Network Service Header (NSH)*. RFC 8300. IETF, 2018. URL: https://datatracker.ietf.org/doc/rfc8300/ (visited on 08/19/2019) (cit. on pp. 76, 89, 92, 93, 96).

[QG14] Paul Quinn and Jim Guichard. 'Service Function Chaining: Creating a Service Plane via Network Service Headers'. In: *IEEE Computer* 47.11 (Nov. 2014), pp. 38–44. DOI: 10.1109/MC.2014.328 (cit. on p. 89).

[QN15] Paul Quinn and Thomas Nadeau. *Problem Statement for Service Function Chaining*. RFC 7498. IETF, 2015. URL: https://datatracker.ietf.org/doc/rfc7498/ (visited on 08/19/2019) (cit. on pp. 15, 89).

[Raj+13] Shriram Rajagopalan et al. 'Split/Merge: System Support for Elastic Execution in Virtual Middleboxes'. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. NSDI'13. Lombard, IL: USENIX Association, Apr. 2013, pp. 227–240 (cit. on pp. 27–29, 31, 32, 37, 38, 41–43).

[RBB16] Varun S Reddy, Andreas Baumgartner, and Thomas Bauschert. 'Robust embedding of VNF/service chains with delay bounds'. In: *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Palo Alto, CA, USA, Nov. 2016, pp. 93–99. DOI: 10.1109/NFV-SDN.2016.7919482 (cit. on p. 172).

[RBR17] Raphael Vicente Rosa, Claudio Bertoldo, and Christian Esteve Rothenberg. 'Take Your VNF to the Gym: A Testing Framework for Automated NFV Performance Benchmarking'. In: *IEEE Communications Magazine* 55.9 (Sept. 2017), pp. 110–117. ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1700127 (cit. on pp. 105, 131, 132, 152, 154, 164, 165).

[Red15] Redislabs. *Redis in-memory store*. 2015. URL: http://redis.io (visited on 05/21/2019) (cit. on pp. 30, 36).

[Rie+16] Jordi Ferrer Riera et al. 'TeNOR: Steps towards an orchestration platform for multi-PoP NFV deployment'. In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE. Seoul, South Korea, June 2016, pp. 243–250. DOI: 10.1109/NETSOFT.2016.7502419 (cit. on p. 22).

[RIF16] RIFT.io. *RIFT.ware*. 2016. URL: https://riftio.com/ (visited on 02/22/2019) (cit. on p. 22).

[Ros+18] Raphael Vicente Rosa et al. *Methodology for VNF Benchmarking Automation*. Internet-Draft. IETF, July 2018. URL: https://datatracker.ietf.org/doc/draft-rosa-bmwg-vnfbench/ (visited on 08/19/2019) (cit. on pp. 7, 132, 148, 152, 182, 185).

[RRS15] Raphael Vicente Rosa, Christian Esteve Rothenberg, and Robert Szabo. 'VBaaS: VNF benchmark-as-a-service'. In: *2015 Fourth European Workshop on Software Defined Networks*. IEEE. Bilbao, Spain, Oct. 2015, pp. 79–84. DOI: 10.1109/EWSDN.2015.65 (cit. on pp. 126, 131, 152).

[RWJ13] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. 'Pico Replication: A High Availability Framework for Middleboxes'. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, Oct. 2013, 1:1–1:15. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523635 (cit. on p. 43).

[Ryu17] Ryu SDN Framework Community. *Ryu Controller*. 2017. URL: https://osrg.github.io/ryu/ (visited on 02/15/2019) (cit. on pp. 12, 13, 52, 94).

[Sch+19] Stefan Schneider et al. 'Putting 5G into Production: Realizing a Smart Manufacturing Vertical Scenario'. In: *2019 IEEE European Conference on Networks and Communications (EuCNC)*. June 2019 (cit. on pp. 75, 166).

[Sch19] Erik Schilling. 'Telco in a Box: Emulating Full-Stack NFV Deployments'. Master's Thesis. Paderborn University, 2019 (cit. on p. 95).

[Sec15] Secdev.org. *Scapy Project*. http://www.secdev.org/projects/scapy/. 2015 (cit. on p. 96).

[SGZ17] Xiaozhe Shao, Lixin Gao, and Hao Zhang. 'CoGS: Enabling distributed network functions with global states'. In: *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE. Bologna, Italy, July 2017, pp. 1–9. DOI: 10.1109/NETSOFT.2017.8004112 (cit. on pp. 29, 43).

[Soco1] Socat project. *Socat - Multipurpose relay*. 2001. URL: http://www.dest-unreach.org/socat/ (visited on 05/02/2019) (cit. on pp. 139, 157).

[Sof17] SoftFIRE project consortium. *SoftFIRE Approach to Experiment Management: Why and How*. 2017. URL: https://www.softfire.eu/wp-content/uploads/SoftFIRE-White-Paper-2-SoftFIRE-Approach-to-Experiment-Management-Why-and-How.pdf (visited on 08/22/2019) (cit. on pp. 64, 93, 105).

[Son+15] Balázs Sonkoly et al. 'Multi-Domain Service Orchestration Over Networks and Clouds: A Unified Approach'. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, Aug. 2015, pp. 377–378. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2790041 (cit. on pp. 22, 67, 68).

[SON15a] SONATA project consortium. *D2.2 Architecture Design*. 2015. URL: http://sonata-nfv.eu/sites/default/files/sonata/public/content-files/deliverables/SONATA_D2.2_Architecture_and_Design_1.pdf (visited on 03/21/2019) (cit. on p. 21).

[SON15b] SONATA project consortium. *SONATA-NFV*. 2015. URL: http://sonata-nfv.eu (visited on 02/22/2019) (cit. on pp. 7, 110, 138, 165).

[SON16] SONATA project consortium. *D3.1 Basic SDK Prototype*. 2016. URL: http://sonata-nfv.eu/sites/default/files/sonata/public/content-files/deliverables/SONATA%20D3.1%20Basic%20SDK%20Prototype.pdf (visited on 03/21/2019) (cit. on p. 75).

[Sou+19] Nathan F. Saraiva de Sousa et al. 'Network Service Orchestration: A survey'. In: *Computer Communications* 142–143 (2019), pp. 69–94. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2019.04.008 (cit. on p. 101).

[SP97] Rainer Storn and Kenneth Price. 'Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces'. In: *Journal of Global Optimization* 11.4 (Dec. 1997), pp. 341–359. ISSN: 1573-2916. DOI: 10.1023/A:1008202821328 (cit. on p. 173).

[SPF17] Rolf Stadler, Rafael Pasquini, and Viktoria Fodor. 'Learning from Network Device Statistics'. In: *Journal of Network and Systems Management* 25.4 (Oct. 2017), pp. 672–698. ISSN: 1573-7705. DOI: 10.1007/s10922-017-9426-z (cit. on p. 164).

[SPK18] Stafan Schneider, Manuel Peuster, and Holger Karl. 'A Generic Emulation Framework for Reusing and Evaluating VNF Placement Algorithms'. In: *2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Verona, Italy, Italy, Nov. 2018. DOI: 10.1109/NFV-SDN.2018.8725795 (cit. on pp. 65, 77, 102).

[Squ96] Squid Project. *Squid: Optimising Web Delivery*. 1996. URL: http://www.squid-cache.org (visited on 05/02/2019) (cit. on pp. 139, 157, 166).

[SS18] Forough Shahab Samani and Rolf Stadler. 'Predicting Distributions of Service Metrics using Neural Networks'. In: *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE. Rome, Italy, Nov. 2018, pp. 45–53 (cit. on p. 164).

[Sun+18] Jian Sun et al. 'A Q-Learning-Based Approach for Deploying Dynamic Service Function Chains'. In: *Symmetry* 10.11 (Nov. 2018). ISSN: 2073-8994. DOI: 10.3390/sym10110646 (cit. on pp. 163, 164).

[Tak+13] Byung Chul Tak et al. 'Pseudoapp: performance prediction for application migration to cloud'. In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*. IEEE. May 2013, pp. 303–310 (cit. on pp. 131, 152).

[Tav+18] Thales Nicolai Tavares et al. 'NIEP: NFV Infrastructure Emulation Platform'. In: *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*. IEEE. Krakow, Poland, May 2018, pp. 173–180. DOI: 10.1109/AINA.2018.00037 (cit. on p. 68).

[The11] The Apache Software Foundation. *Apache Kafka: A distributed streaming platform*. 2011. URL: https://kafka.apache.org (visited on 07/25/2019) (cit. on p. 182).

[The13] The OpenDaylight Foundation. *OpenDaylight*. 2013. URL: https://www.opendaylight.org (visited on 02/15/2019) (cit. on pp. 12, 13, 23, 94).

[The15a] The Apache Software Foundation. *Cassandra Distributed Database*. 2015. URL: http://cassandra.apache.org (visited on 05/21/2019) (cit. on p. 30).

[The15b] The NOX/POX Project. *NOX/POX OpenFlow Controller*. 2015. URL: http://www.noxrepo.org (visited on 02/15/2019) (cit. on pp. 12, 13, 35).

[The93] The Apache Software Foundation. *Apache HTTP Server*. 1993. URL: https://httpd.apache.org/ (visited on 05/02/2019) (cit. on pp. 141, 166).

[TRR10] Paul Turner, Bharata B Rao, and Nikhil Rao. 'CPU bandwidth control for CFS'. In: *Linux Symposium*. Vol. 10. Google, 2010, pp. 245–254 (cit. on p. 84).

[TZK16] Javid Taheri, Albert Y Zomaya, and Andreas Kassler. 'vmBBThrPred: A Black-Box Throughput Predictor for Virtual Machines in Cloud Environments'. In: *European Conference on Service-Oriented and Cloud Computing*. Springer. Aug. 2016, pp. 18–33. DOI: 10.1007/978-3-319-44482-6_2 (cit. on pp. 131, 152).

[TZK17] Javid Taheri, Albert Y. Zomaya, and Andreas Kassler. 'vmBBProfiler: a black-box profiling approach to quantify sensitivity of virtual machines to shared cloud resources'. In: *Computing* 99.12 (Dec. 2017), pp. 1149–1177. ISSN: 1436-5057. DOI: 10.1007/s00607-017-0552-y (cit. on p. 131).

[Van+17] Steven Van Rossem et al. 'A network service development kit supporting the end-to-end lifecycle of NFV-based telecom services'. In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. Berlin, Germany, Nov. 2017, pp. 1–2. DOI: 10.1109/NFV-SDN.2017.8169859 (cit. on p. 66).

[Van+18] Steven Van Rossem et al. 'Introducing Development Features for Virtualized Network Services'. In: *IEEE Communications Magazine* PP.99 (Feb. 2018), pp. 2–10. ISSN: 0163-6804. DOI: 10.1109/MCOM.2018.1600104 (cit. on p. 66).

[VMw13] VMware, Inc. *vSphere: Server Virtualization Software*. 2013. URL: https://www.vmware.com/products/vsphere.html (visited on 03/01/2019) (cit. on p. 23).

[Wan+17] Wenxin Wang et al. 'Consistent State Updates for Virtualized Network Function Migration'. In: *IEEE Transactions on Services Computing* (Oct. 2017). DOI: 10.1109/TSC.2017.2765636 (cit. on p. 43).

[Wan+18] Mowei Wang et al. 'Machine Learning for Networking: Workflow, Advances and Opportunities'. In: *IEEE Network* 32.2 (Mar. 2018), pp. 92–99. ISSN: 0890-8044. DOI: 10.1109/MNET.2017.1700200 (cit. on pp. 163, 164).

[War90] Seth Warner. *Modern Algebra*. Courier Dover Publications, 1990 (cit. on p. 137).

[Wet+14] Philip Wette et al. 'Maxinet: Distributed emulation of software-defined networks'. In: *2014 IFIP Networking Conference*. IEEE. Trondheim, Norway, June 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857078 (cit. on pp. 66, 91, 140, 184).

[Woo+08] Timothy Wood et al. 'Profiling and modeling resource usage of virtualized applications'. In: *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*. Springer-. Dec. 2008, pp. 366–387 (cit. on pp. 131, 152).

[Zer+19] Johannes Zerwas et al. 'NetBOA: Self-Driving Network Benchmarking'. In: *Proceedings of the 2019 Workshop on Network Meets AI & ML*. NetAI'19. Beijing, China: ACM, Aug. 2019, pp. 8–14. ISBN: 978-1-4503-6872-8. DOI: 10.1145/3341216.3342207. URL: http://doi.acm.org/10.1145/3341216.3342207 (cit. on p. 153).

[Zha+12] Wei Zhao et al. 'Modeling and simulation of cloud computing: A review'. In: *Cloud Computing Congress (APCloudCC), 2012 IEEE Asia Pacific*. IEEE. Shenzhen, China, Nov. 2012, pp. 20–24. DOI: 10.1109/APCloudCC.2012.6486505 (cit. on pp. 66, 91).

[Zha+17] Mengxuan Zhao et al. 'Verification and validation framework for 5G network services and apps'. In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. Berlin, Germany, Nov. 2017, pp. 321–326. DOI: 10.1109/NFV-SDN.2017.8169878 (cit. on p. 105).