Towards Lightweight, Low-Latency Network Function Virtualisation at the Network Edge

Richard Cziva

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

School of Computing Science

College of Science and Engineering

University of Glasgow

August 2018

Abstract

Communication networks are witnessing a dramatic growth in the number of connected mobile devices, sensors and the Internet of Everything (IoE) equipment, which have been estimated to exceed 50 billion by 2020, generating zettabytes of traffic each year. In addition, networks are stressed to serve the increased capabilities of the mobile devices (e.g., HD cameras) and to fulfil the users' desire for always-on, multimedia-oriented, and low-latency connectivity. To cope with these challenges, service providers are exploiting softwarised, cost-effective, and flexible service provisioning, known as Network Function Virtualisation (NFV). At the same time, future networks are aiming to push services to the edge of the network, to close physical proximity from the users, which has the potential to reduce end-to-end latency, while increasing the flexibility and agility of allocating resources. However, the heavy footprint of today's NFV platforms and their lack of dynamic, latency-optimal orchestration prevents them from being used at the edge of the network.

In this thesis, the opportunities of bringing NFV to the network edge are identified. As a concrete solution, the thesis presents Glasgow Network Functions (GNF), a container-based NFV framework that allocates and dynamically orchestrates lightweight virtual network functions (vNFs) at the edge of the network, providing low-latency network services (e.g., security functions or content caches) to users. The thesis presents a powerful formalisation for the latency-optimal placement of edge vNFs and provides an exact solution using Integer Linear Programming, along with a placement scheduler that relies on Optimal Stopping Theory to efficiently re-calculate the placement following roaming users and temporal changes in latency characteristics.

The results of this work demonstrate that GNF's real-world vNF examples can be created and hosted on a variety of hosting devices, including VMs from public clouds and low-cost edge devices typically found at the customer's premises. The results also show that GNF can carefully manage the placement of vNFs to provide low-latency guarantees, while minimising the number of vNF migrations required by the operators to keep the placement latency-optimal.

Acknowledgements

Behind this thesis there has been many years of discussion, work and collaboration with incredible people. The following is an acknowledgement of them.

First of all, my thanks and appreciation go to my principal advisor Dr. Dimitris P. Pezaros for his continuous feedback, support, funding, motivation and much more. Over the years Dimitris taught me to shoot for the stars and to never underestimate my work.

I would like to also thank Dr. Jeremy Singer for replying to my first unsolicited e-mail sent to him in early 2012. Jeremy took me as an Erasmus student and offered to start a PhD in Glasgow. Also, many thanks to Dr. Christos Anagnostopoulos for his recent support on the theoretical part of my thesis and encouragement in the final year of of my PhD.

Thanks to my friends and officemates Dr. Simon Jouet, Dr. Levente "Zolibá" Csikor and Dr. Kyle White for their friendship, input, collaboration and the occasional *one more* pint.

I would like to express my special thanks to all the people around the globe who believed in me and hired to work for them during my PhD. Thanks to Jerry Sobieski from NORDUnet, Culley Angus, Chris Lorier, Jamie Curtis from REANNZ, and Chin Guok, Inder Monga from ESnet. Without them, my PhD would have been a totally different experience.

I am grateful to my PhD examiners Dr. Colin Perkins (University of Glasgow) and Dr. Nicholas Race (University of Lancaster) for providing feedback on this thesis.

Finally, I would like to thank my family and in particular my mom and dad for their encouragement and unlimited support. Enormous thanks to my fiancée, Anna, for her love, relocation to Glasgow, and the unconditional understanding and support provided during these years.

Élvezd, amíg van még mit, mintha ez a perc lenne az utolsó,

Nem attól haladsz előre, ha össze-vissza futkosol.

Punnany Massif

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Data consumption is growing exponentially in today's communication networks. This irreversible trend is driven by the increase of connected end-users and the wide-spread penetration of new mobile devices (e.g., smartphones, wearables, sensors and more). In addition, data consumption is also accelerated by the increased capabilities of the mobile clients (e.g., higher resolution screens and HD cameras), and the user's desire for high-speed, always-on, multimedia-oriented connectivity. In numbers, it has been estimated that connected devices will exceed 50 billion, generating zettabytes (1 billion terabytes) of traffic yearly by 2020 [1].

At the same time, the Telecommunication Service Provider (TSP) market is becoming competitive with the rise of many over-the-top providers lowering subscription fees for users. Moreover, today's TSPs often experience poor resource utilisation, tight coupling with specific hardware, and lack of flexible control interfaces which fail to support diverse mobile applications and services. As a result, TSPs have started to lose existing and new revenue, while suffering increased capital and operational expenditure that cannot be balanced by increasing subscription costs [2].

In order to cope with the aforementioned challenges, service providers have started to softwarise their network infrastructure. By virtualising traditional network services (e.g., fire-

walls, caches, proxies, intrusion detection systems, WAN accelerators, etc.), providers can save operational and capital expenses, and satisfy user demands for customised and rapidly evolving services. This transformation, referred to as Network Function Virtualisation (NFV), changes how operators architect their network to decouple network functionality from physical locations for faster and flexible service provisioning [2]. NFV has gained significant attention since its first appearance in 2012, resulting in many, albeit still preliminary, deployments at the provider's Data Centres (DCs).

At the same time, a new concept for service delivery has emerged referred to as Multi-Access Edge Computing (MEC) [3] or fog computing [4]. These concepts present an IT service environment with cloud computing capabilities at the edge of the home, enterprise or IoT network, within close proximity to the end users [3]. Being able to utilise the edge of the network will inherently provide low-latency connectivity between services and the end users, while at the same time it would allow offloading the core of the provider's network. However, the heavy footprint of today's NFV frameworks and their lack of dynamic, temporal orchestration prevents them from being used at the network edge.

In this thesis, the design and implementation of Glasgow Network Functions (GNF), a NFV platform that brings NFV to the network edge by using generic, lightweight Linux containers in a distributed, heterogeneous edge infrastructure is presented. To show the potential of the platform, multiple useful use-cases are highlighted demonstrating that by utilising the network edge (e.g., home, enterprise, IoT edge), providers can alleviate unnecessary network utilisation (which can correspond to savings of millions of dollars per year [5]), perform better troubleshooting on their network, and provide location-transparent services [6].

Furthermore, this thesis advocates that *Edge vNFs* must be dynamically placed in synergy with changing network dynamics (e.g., fluctuating latency on links), user demands (e.g., certain vNFs are used in certain times of the day) and mobility (e.g., users move continuously with the mobile devices) to support low vNF-to-user end-to-end latency. Therefore, this thesis formulates a realistic vNF placement problem for the network edge and also provides a time-optimised scheduler (along with mathematical analyses and theorems) that strives to

achieve latency-optimal allocation of vNFs in next-generation edge networks using the fundamentals of Optimal Stopping Theory [7]. With experiments over a simulated nation-wide network topology using real-world latency characteristics, we show that, by re-calculating the optimal placement and migrating vNFs at a carefully selected time, unnecessary vNF migrations can be prevented to save resources for the operators, while the number of latency violations in the network can be kept bound to satisfy the QoS requirements of users.

1.2 Thesis Statement

This work asserts that by creating, moving, and dynamically managing lightweight virtual network functions in close physical proximity to the end users (for example at the edge of the network) will enable Telecommunication Service Providers to deliver low-latency, QoS-guaranteed network services with low impact on the provider's network. This hypothesis has been be tested by the development and evaluation of a container-based network function framework, applying NFV and SDN technologies.

1.3 Contributions

The contributions of this thesis are as follows:

- A comprehensive review of the past and current approaches that aim to introduce programmability at different planes of the network.
- An analysis of the current limitations of today's NFV frameworks and vNF orchestration algorithms, and the resulting hurdles of dynamic resource allocation, management and orchestration when applied on future edge networks.
- A formulation of a latency-optimal edge vNF placement problem and an exact solution using Integer Linear Programming and the Gurobi solver.

- A dynamic placement scheduler that minimises the number of vNF migrations required by latency changes while adhering to service level guarantees using the fundamentals of Optimal Stopping Theory.

- The design and implementation of Glasgow Network Functions (GNF), a comprehensive framework that manages virtual, lightweight container network functions on various hosting devices (from low-cost network edge devices to a public cloud VMs) and transparently assigns vNFs to selected user's traffic.

- An open-source implementation of today's most popular types of network functions as lightweight GNF vNFs that can be specialised to individual user requirements.

- An evaluation of the benefits of GNF vNFs compared to unikernels and traditional VMs, and an evaluation of GNF on public cloud providers (Amazon EC2, Google Compute Engine, Microsoft Azure).

- An evaluation of GNF's dynamic, latency-optimal vNF placement solution over a nation-wide, simulated network topology consisting of edge and cloud servers.

1.4 Organisation of the Thesis

The work presented in this thesis is structured as follows:

- **Chapter 2** discusses the need and evolution of virtualised, software-based network infrastructures and introduces the emerging requirements of today's end users. As a future architecture to support this evolution, the chapter outlines the concepts of Network Function Virtualisation and Software Defined Networking. It then describes the current challenges large-scale telecommunication networks face and the incentive for network providers to leverage the newly available edge architecture alongside their internal infrastructure. The chapter details current platforms for virtualised service delivery and existing orchestration solutions for vNFs.

- **Chapter 3** critically discusses the limitations of previous frameworks and orchestration algorithms, and motivates the need for a lightweight NFV platform. After introducing the most important design considerations, the chapter compares different virtualisation technologies (e.g., virtual machines, unikernels, and Linux containers) and their suitability for the network edge. Furthermore, this chapter identifies the challenges of virtualised service orchestration and presents a latency-optimal placement algorithm for vNFs, alongside a dynamic placement scheduler that enables efficient re-allocation of vNF services in response to evolving temporal network properties.

- **Chapter 4** provides the technical aspects of the implementation for the design discussed in Chapter 3. The Glasgow Network Function (GNF) framework is presented, detailing how it manages the network functions, their placement, and the network traffic in various deployment scenarios. The chapter concludes with the implementation details of selected real-world network functions (e.g., firewalls, deep packet inspection modules, etc.) as GNF vNFs, highlighting the capabilities of the system.

- **Chapter 5** presents a comprehensive evaluation of GNF. First, it shows some characteristics of container vNFs that are important for network services (e.g., throughput, delay, instantiating time, overhead etc). Then, it presents how GNF vNFs perform in public cloud environments (e.g., Amazon EC2, Microsoft Azure, Google Compute Engine). Finally, this chapter presents an evaluation of the dynamic, latency-optimal edge vNF placement orchestration over a nation-wide simulated network topology, using real-world latency characteristics.

- **Chapter 6** summarises the work and contributions and impact of this dissertation. It provides some use-cases of the GNF platform (e.g., IoT DDoS mitigation, on-demand network troubleshooting, vNFs that are following users) and discusses potential research directions as future work in this topic.

Chapter 2

Background and Related Work

2.1 Overview

The Internet was designed almost 50 years ago as a network of computers, used only by a limited number of researchers to transmit electronic messages of files and scientific results. Since then, the Internet community has witnessed a tremendous change in the number of users and devices and also in how users interact with the network. This increase of usage and change of expectations have led network operators to look into better ways of managing and orchestrating their network in order to reduce operational expenses, and to improve the utilisation of existing resources to improve return on investment. However, managing such widespread, distributed infrastructure is challenging.

In order to facilitate efficient management and orchestration, and allow operators to better control their network, both academia and industry have made considerable effort since the early 1990s, when the Internet has really started to take off with more diverse applications and users from the general public. As shown in Figure 2.1, contributions to network programmability start from early ideas originating from the active networking paradigm [31] until recent advances in Network Function Virtualisation and the use of commodity (and even resource-constrained) hardware located at the edge of the network for advanced, programmable network services. While the innovations are still ongoing, the recent advances

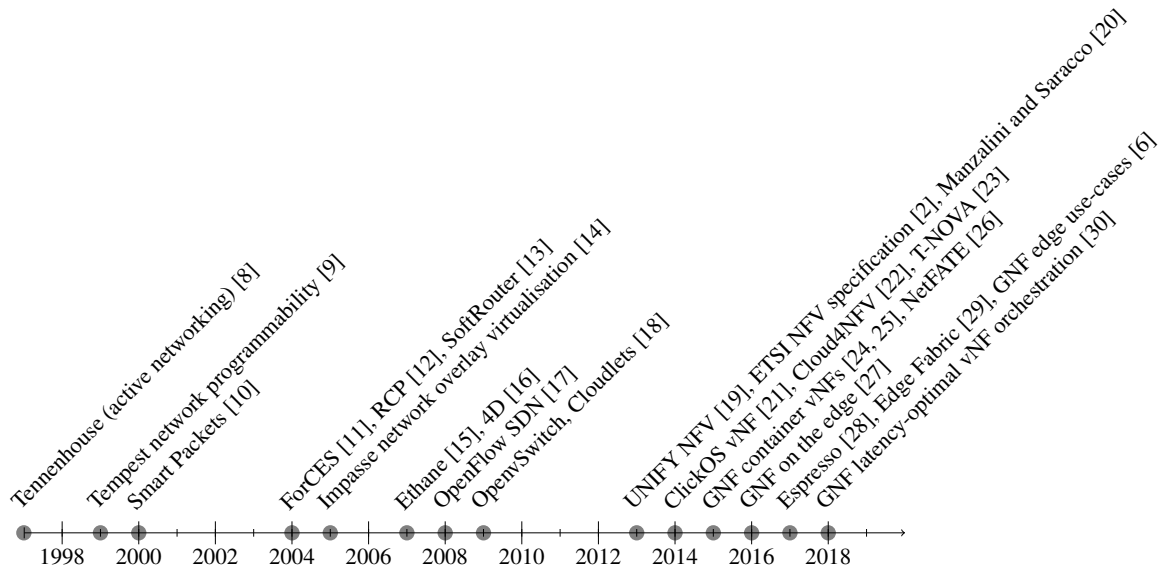


Figure 2.1: Selected key contributions to programmable network configuration and service delivery over the past 20 years leading to the contributions presented in this thesis.

enable a shift from legacy networking where services are bound to specific locations to deliver user and location-specific network services.

In this chapter, the related work and background of this thesis are outlined. First, in Section 2.2, the research leading to network virtualisation with SDN and NFV is presented by drawing a timeline from the 1990s, when the first ideas around network programmability were born. Then, in Section 2.3, the need for virtualised in-network services is described with details on next-generation, edge-based network services. In Section 2.4, particular attention is drawn to the the orchestration challenges of virtualised services by describing and presenting influential research papers and projects from the recent years. Finally, Section 2.5 summarises the key findings of this chapter.

2.2 Towards a Virtualised Network Infrastructure using SDN and NFV

In this section the early work on network programmability is reviewed, drawing a timeline starting from the 1990s when the Internet has just started to take off as a closed research network until today, when all network providers apply high levels of automation over their net-

work. The review presented below highlights how previous efforts (e.g., active networking, ForCES, Ethane) paved the way towards Software Defined Networking (SDN) and Network Function Virtualisation (NFV), today's prominent trends to deliver and manage softwarised network services for the users in a highly automated, "zero-touch" way.

2.2.1 Early Approaches to Network Programmability

In the early- to mid-1990s, the Internet had taken off with many new applications outpacing file transfer and email for scientists [32]. This innovation at the end hosts was possible due to the layered structure of network protocols. While many new applications have been built on top of the protocol stack, improving and changing the behaviour of network services and the network itself was very hard, due to its distributed, multi-vendor nature and the slow protocol standardisation that is coordinated by the Internet Engineering Task Force (IETF) [33, 34].

A notable, clean-slate approach for network programmability has been presented by the active networking program [31] in the early 2000s where a programming interface exposed processing, storage resources along with control on network queues from specific devices [35]. This network API enabled the execution of custom operations that were carried in-band, encapsulated in the data packets (also known as capsules or smart packets) [36]. Envisioned use-cases of active networking included network support for new applications, fine-grained control over packet forwarding and introducing the possibility for network experimentation.

On top of presenting the well-known network APIs and an encapsulation model, active networking research papers have also offered the vision of a unified architecture for middlebox orchestration which essentially meant unifying a wide range of network middleboxes and managing them in a unified way instead of ad-hoc, one-off approaches [37] - a very similar goal of this thesis which resonates well with today's efforts on Network Function Virtualisation introduced later.

While active networking has offered the vision of unified network programmability, it has not taken off for several reasons. One notable challenge has been a disagreement in the research community whether the network should be kept simple (only forwarding packets between

ports) or it should be enabled with processing for smarter functionality. On the other hand, active networking lacked commercial deployments and remained a research project for a long time due to the fact that there was no real need for the proposed programmability back in the early 2000s. Moreover, running executable code inside the network has raised security concerns.

Another prominent early approach was presented for ATM networks by the researchers at University of Cambridge in the work titled as Tempest [9], where the authors have outlined a practical framework for network programmability with dynamic introduction and modification of network services. Tempest enabled the deployment of alternative control architectures in the same network, allowing service providers to effectively become network operators for some well defined partition of the physical network. Moreover, the Tempest architecture even allowed switch-forwarding behaviour to be defined by a controller, foreshadowing the work on data and control plane separation.

2.2.2 The Separation of the Control and Data Planes

In mid 2000s, the demand for higher-performance, more reliable networks increased along with the number of connected users reaching 1 billion in 2005[1]. Subsequently, to provide connectivity for increasing number of users, providers have started to focus on network management functions, widely called Traffic Engineering (TE) - controlling the physical path used to deliver traffic. However, at the time, this was only possible by controlling conventional routing protocols that were very primitive, restricted and more importantly tied into conventional routers and switches. This meant that network management tasks such as creating customised routing, debugging and telemetry were extremely difficult [38].

At the same time, there was a tremendous improvement in the speed of general purpose computers with new processors and high speed memories coming out yearly, leaving the control plane of networking equipment greatly underpowered [32]. These frustrations and the ability to implement new network functions using the resources available at general purpose com-

[1]https://ourworldindata.org/internet/ Accessed on 13 February 2018

puters resulted in two innovations: an *open interface* between the control and data plane of the network and also the notion of a *logically centralised control plane* [32].

As a result of these efforts, the ForCES (Forwarding and Control Element Separation) [11] working group at IETF has proposed a standardised, open interface between control and data planes, while logical centralisation of control has also been suggested at protocols such as the Routing Control Platform [39] in the mid 2000s. While these control and data plane separation projects were important milestones in IP networks to enable innovation in separate planes, the concept of separating a control plane was already known in the telecommunication industry many years before (e.g., SS7 [40] introduced signaling plane separation in 1998).

ForCES had its own shortcoming: equipment vendors had little incentive to adopt it as it exposed internal properties of the network that was not in the interest of the vendors in order to keep competitive advantage. To solve these issues, the Ethane [41] project (and its predecessor SANE [42]) instead of exposing internal properties of the device, only proposed a logically centralised, flow-level solution for access control targeted to enterprise networks. Ethane reduced a switch to only flow tables where flows could be only populated by a high-level controller. As we show in the next section, along with industry support and the right definition of a control protocol, essentially this project became OpenFlow [17], the project that has changed the networking industry.

2.2.3 Innovation at the Control Plane - SDN and OpenFlow

During mid and late 2000s, the success of experimental infrastructures (e.g., PlanetLab [43]) increased the interest towards large scale network experimentation. Many large-scale research projects such as the US funded Global Environment for Network Innovations (GENI) [44] or the European funded Future Internet Research & Experimentation (FIRE) [45] have started to propose ideas on how networks need to be managed on scale. Specific experiments from these programs include FIRE's FEDERICA project that created a scalable, Europe-wide, clean slate infrastructure for Internet experimentation or GENI's MobilityFirst experiment

that was selected for early validation of a massive scalable global name resolution service and a generalised storage aware routing. Amongst all of these, researchers at Stanford focused on the campus network and created OpenFlow [17], today's most popular realisation of the Software Defined Networking (SDN) vision.

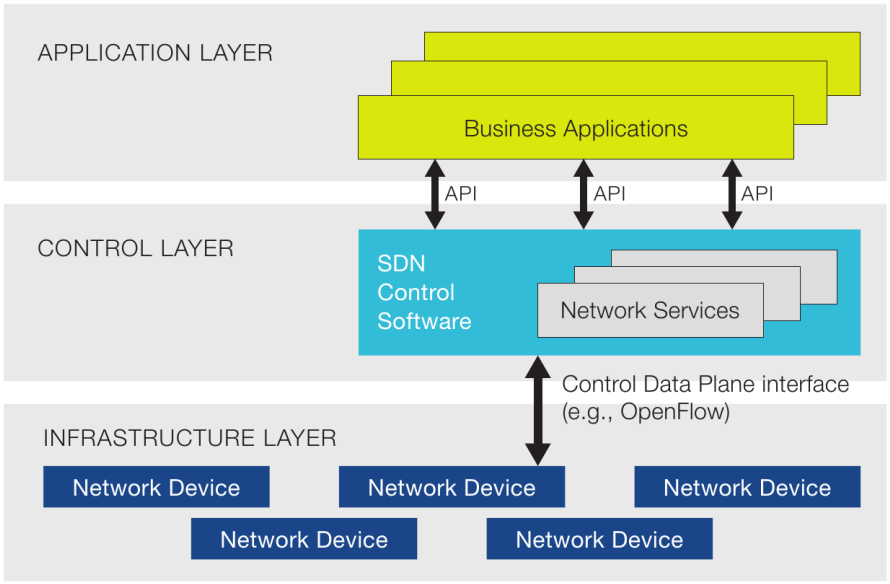


Figure 2.2: Software Defined Networks logical architecture. Source: SDxCentral

SDN brings innovation to the network control plane and changes the way networks are designed and managed today. It has two defining characteristics. First, it separates the network control plane specifying routing policies, route finding algorithms from the data plane's hardware devices that forward the packets, as shown in Figure 2.2. Secondly, SDN consolidates the control plane into a logically centralised SDN software controller (commonly referred as SDN controller) that provides a global view of the network and is able to control multiple data plane elements. This logical centralisation allows implementing and modifying control procedures (e.g., routing, traffic engineering, security functions) in one designated place (treating the network as a big switch), instead of having to modify all network devices in the network separately (often using different control protocols to different devices) to implement such control procedures. Moreover, many SDN controllers are designed for multi-tenancy, providing a so called network operating system that allows multiple high-level SDN applications to run on a virtualised view instead of directly interacting with physical devices.

SDN allows policies, configuration and network resource management to be applied in short timescales, and a single control protocol to implement a range of functions such as, among others, flow-based routing [17], QoS enforcement [46], network virtualisation [47], traffic engineering [48], policy management [49], enabling content caching [50] and even managing Virtual Machine placement in cloud DCs [51].

2.2.3.1 The OpenFlow Protocol

OpenFlow [17] by McKeown et al. realises the SDN vision by defining a data forwarding pipeline along with a simple TCP-based communication protocol to be used between the SDN controller and the data plane devices of the network.

The OpenFlow data plane is based on multiple, but simple match-action tables that are able to hold a fixed amount of flow entries describing a matching pattern (that matches in bits in the packet header) and associated actions (e.g., forwarding, dropping, modifying a header field in a packet). Flow entries also contain a set of counters to track number of bytes and packets matched along with priority information that is used when a packet matches multiple flow entries.

The OpenFlow data plane is managed by an associated OpenFlow communication protocol that allows setting up and removing flow entries, querying flow statistics. While OpenFlow has had many versions in the last couple of years, three of these can be considered the most influential: OpenFlow 1.0 (which is used even today since it was the first protocol published), OpenFlow 1.3 (today's de facto standard in hardware devices), and OpenFlow 1.5 (the most up-to-date version that is gaining popularity now).

2.2.3.2 SDN Deployments, Devices, and Controllers

The introduction of the SDN vision was quickly followed by various deployments from large-scale network and data centre operators. One of the first reported deployments of SDN is accredited to Google, where the technology was used to interconnect private DCs globally in 2013. This deployment, called B4 [52], allows setting up bandwidth guarantees between

any two hosts, even if the hosts are located in two different DCs. In the B4 paper, the authors described how they support multiple routing protocols simultaneously and how they perform centralised traffic engineering using SDN.

Many switch vendors have also joined the SDN revolution. Legacy network equipment vendors such as Cisco and Juniper have added OpenFlow capabilities to some of their devices alongside their own proprietary control protocols. Example legacy devices equipped with OpenFlow support include Cisco's Nexus 9000 series devices[2] or Juniper's MX series edge routers[3]. New SDN hardware vendors, such as Big Switch Networks, Noviflow, Barefoot Networks and Corsa Technologies have started to manufacture clean slate devices designed for SDN, with high-speed programmable pipelines and up-to-date OpenFlow support. Example devices include Corsa's DP2400 device that is a fully OpenFlow 1.5 compliant switch with 48Gb packet buffering and a line-rate throughput of 200Gbps per device[4]. Apart from hardware devices, dozens of software switches have also been designed for SDN (e.g., Open vSwitch [53], mSwitch [54], Pisces [55]).

SDN network operators have the choice to use one of the numerous SDN controllers. Popular SDN controllers include the OpenDaylight [56] or ONOS [57] controllers which are used by many enterprise network providers (e.g., Huawei, AT&T), the Faucet [58] controller that has been deployed by many research and education network providers and universities. The Ryu[5] controller that has been used in dozens of research experiments due to its lightweight design and support for latest versions of the OpenFlow protocol.

2.2.3.3 Limitations of SDN

SDN provides the vision of a centralised, virtualised control plane along with a well defined and simplified data plane. This allows centralising distributed control applications such as unified access control management, network flow and topology management to a network

[2]https://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/guide-c07-731461.html Accessed on: 13 February 2018

[3]https://www.juniper.net/documentation/en_US/release-independent/junos/topics/reference/general/junos-sdn-openflow-supported-platforms.html Accessed on: 13 February 2018

[4]https://www.corsa.com/products/dp2400/ Accessed on: 13 February 2018

[5]https://osrg.github.io/ryu/ Accessed on: 25 January 2018

controller that only deals with control messages. However, SDN was not designed to handle data-intensive operations in its centralised control plane and therefore SDN controllers would introduce significant delays on the network in case each packet was directed to them for inspection or modification (for example for packet inspection purposes).

This limitation of the SDN vision has driven a few orthogonal research directions. First, some researchers have started to experiment with languages that could define an entire data plane from code. Examples of this approach include P4 [59], where a language is presented to define switch's pipelines or BPFabric [60], where the authors proposed a platform independent instruction set to define data-plane functions such as anomaly detection using Berkeley Packet Filters.

An other approach to introduce programmability to data plane functions can be achieved with Network Function Virtualisation that is introduced in the next Section.

2.2.4 Network Function Virtualisation

To extend the programmability proposed at the control plane with SDN, the Network Function Virtualisation (NFV) paradigm aims at replacing hardware-based equipment implementing data plane functionality (e.g., security and performance functions) with software-based virtual network functions (vNFs). It enables implementing and running vNFs on Off-The-Shelf x86 servers by using commodity programming languages, frameworks and virtualisation techniques. NFV therefore offers faster deployment and provisioning of service functions, and addresses the problem of compatibility of vendor-specific hardware while reducing the capital and operational expenditure associated with network service deployment [61].

2.2.4.1 NFV Reference Architecture

The European Telecommunications Standards Institute (ETSI), the standardisation body for NFV, defines a reference architecture of an NFV platform [2]. Since its first release in 2012, this architecture has been the basis for most NFV frameworks. As shown in Figure 2.3, the

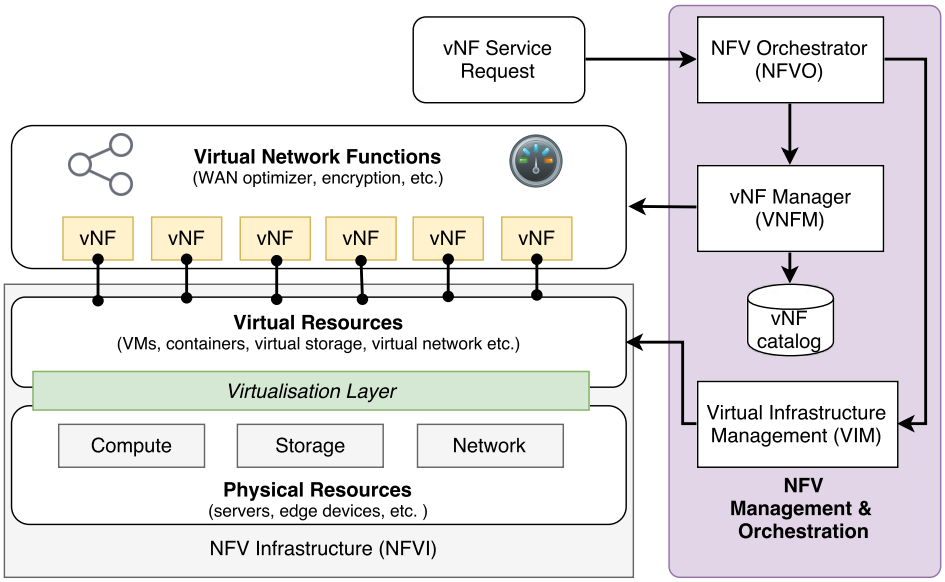


Figure 2.3: ETSI NFV reference architecture

architecture consists of three main blocks described below: Management and Orchestration (MANO), the NFV Infrastructure (NFVI) and the virtual Network Functions.

NFV Infrastructure (NFVI) The NFV Infrastructure (NFVI) component of the architecture consists of the physical resources on which vNFs can be deployed. It abstracts the hardware resources (such as compute, storage and network) through server virtualisation (e.g., x86 virtualisation using Virtual Machines), so they can be logically partitioned and provided for vNFs.

The NFVI component works directly with vNFs, controlled by the virtual infrastructure managers (VIMs) managed by the NFV orchestrator, as shown in Figure 2.3. NFVI is critical to realise the business benefits outlined by NFV, as it enables the use of commodity hardware for multiple network services.

Management and Orchestration (MANO) The Management and Orchestration (MANO) component is responsible for the overall control over an NFV system.

Within that, the vNF Manager (VNFM) is responsible for setting up the software implementing the vNF functionality inside the virtual resource (VM). The VNFM usually connects to a database or catalogue that describes how to set up particular vNFs. As an example, an

intrusion detection vNF can have a set of files describing attack signatures along with scripts to set up a particular intrusion detection software, all stored in a vNF catalogue or repository.

The NFV Orchestrator (NFVO) has high-level control over the entire NFV infrastructure with connections to the VIM and VNFM components of the system. It has two main roles. First, it takes vNF requests from operators and allocates the new vNFs in the infrastructure based on a vNF orchestration strategy. Second, it monitors the health of the system and reacts to certain events impacting performance.

Finally, the Virtual Infrastructure Management (VIM) component manages the physical infrastructure and creates virtual resources when called by the NFVO. It is also responsible for the health checks of the physical infrastructure and reporting available resources.

Virtual Network Functions Virtual Network Functions (abbreviated as VNFs, vNFs or NFs) are the software components that implement the virtual services, encapsulated in a chosen virtualisation technology and deployed on a specific computer server in the infrastructure. vNFs can be moved between servers (restricted only by the vitualisation layer) and be stopped and re-started within few minutes [24, 21]. These vNFs are implemented using virtual compute, storage, and network resources, while being managed by a vNF Manager (VNFM) component of the system, as shown in Figure 2.3.

2.2.4.2 Relationship of NFV and SDN

Even though SDN is not part of the ETSI NFV architecture, the two technologies often exist together, as shown in the Figure 2.4. While many different ways of integration of these two complimentary technologies are possible, in general, NFV provides the programmability to the higher layers of the protocol stack (i.e., application level network services), while SDN allows the control over the lower layers (i.e., routing) of the network. In a practical example, NFV would allow the fast creation of a vNF running a web cache, while SDN would be used to automatically re-route selected users' traffic through this web cache.

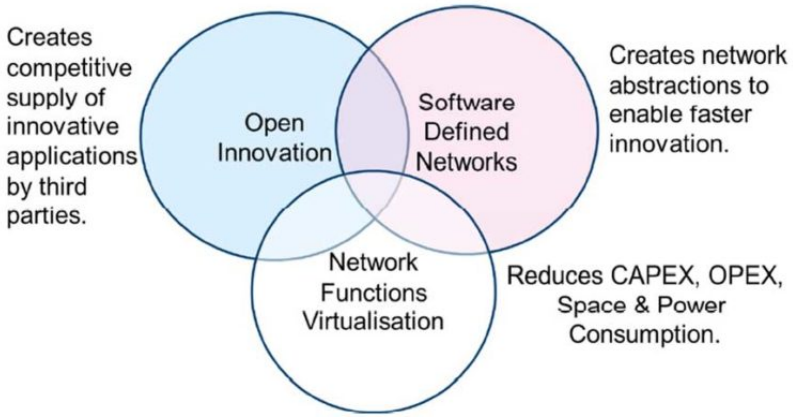


Figure 2.4: Network Functions Virtualisation Relationship with SDN. Source: SDxCentral

2.2.4.3 Relation to the End-to-End Principle

Traditional networks have been built using the end-to-end principle, meaning that the end-points in a network conversation should be responsible for guaranteeing they get the levels of reliability they need from the network while treating in-network services and functions only as performance enhancements [62]. The IP protocol, among others, was built using this principle.

While it is important to note that the end-to-end principle does not advocate a dumb network (as it only states that the endpoints should be responsible to application-to-application communication), many in-network middleboxes break the principle. Traditional NATs, security and performance-enhancing proxies (e.g., content cache) and WAN protocol optimisers are only a few of the possible functions that can look beyond the IP header, keep state in the network and break the end-to-end argument. While some of these are best implemented end-to-end (e.g., encryption), functions such as content caches and NATs are best placed in the network.

Meanwhile, some protocols, such as SIP (Session Initiation Protocol) require in-network support in the form of SIP proxy servers that direct calls based on user's locations. Deploying such functions as in-network vNFs provides flexibility in function placement while adhering to the end-to-end principle.

SDN's OpenFlow doesn't inherently support or break the end-to-end principle. With SDN,

one can implement a network where intelligence is centralised to an SDN controller that directs dumb switches. As an example, SDN's vision of network virtualisation creates featureless pipes over the network that does not break the end-to-end principle itself. However, those overlay networks can already include intelligence in the form of virtualised appliances that can subvert the end-to-end principle.

2.2.5 Example NFV Platforms

Many NFV frameworks have been developed in both academia and industry. These frameworks usually focus on a specific network environment (e.g., the core of the provider's network, DC network) or various aspects of vNF description or management (e.g., service description languages, vNF service chaining). In the following paragraphs, some of the most popular NFV frameworks are introduced to the reader with some of their unique contributions to the field as well as details for vNF placement algorithms used, if applicable.

As shown below, some NFV platforms are built on top of the principles of NFV Management and Organization (MANO), an guideline architecture set by a working group (WG) of the European Telecommunications Standards Institute Industry Specification Group (ETSI ISG NFV).

OpenStack OpenStack [6] is a free and open-source software platform for virtualised network, storage and compute infrastructures. It gained popularity over the last 10 years being deployed at many cloud providers (e.g., RackSpace) and enterprise infrastructures with more than 500 companies contributing to its source code.

OpenStack's strength is its ability to control multiple thousands of hosting servers, running tens of thousands of VMs. Therefore, many NFV platforms have been built on top of OpenStack - as introduced later. The weaknesses of the platform are its orchestration engine and its separation from network controllers that are detailed below.

[6] https://www.openstack.org/ Accessed on: 6 July

OPNFV OPNFV[7] is the most influential project today among all the open-source NFV frameworks. OPNFV works closely with the ETSI, the standardisation body of NFV and it is currently a Linux Foundation project with all major operators (e.g., Cisco, AT&T, Ericsson, Huawei, IBM, Intel, NEC, Nokia and many more) contributing to its source code regularly.

The objective of OPNFV is to establish a carrier-grade, integrated open-source reference platform that can be used to validate multi-vendor, inter-operable NFV solutions. The project builds on top of countless open-source projects, such as OpenStack (virtual compute, storage, network management), Intel DPDK (data-plane development kit) [63], OvS (software switch) [53] and only maintains a necessary thin layer of orchestration between them.

OPNFV, as many other NFV frameworks, uses OpenStack's built-in VM placement algorithm to determine the physical location of vNFs. This is implemented in OpenStack's *nova-scheduler* module involving a host weighting (based on, e.g., available RAM on hosts) and filtering (removing hosts that are not supposed to run a VM) stages before selecting the least loaded host by default. This behaviour can be changed to random selection and developers can also provide hints to the scheduler to enforce VMs to be allocated on specific hosts.

Cloud4NFV Cloud4NFV [22] is one of the early NFV platform that originated in academia. The authors have specifically focused on data modelling that allows the description of vNFs (vNF image, VM instance used, storage used, ports and network link requirements) as well as the virtual infrastructure (cores, available VMs, links, and physical ports). Cloud4NFV also presented an implementation where vNFs could be initiated and mapped to physical resources using their data model.

Cloud4NFV's proof-of-concept is based on OpenStack, therefore by default it uses the VM placement algorithm of OpenStack, as described for OPNFV before.

OpenMANO OpenMANO [64] is an open-source project led by the Spanish network provider Telefonica, aiming at implementing an ETSI NFV MANO framework focusing

[7] http://www.opnfv.com Accessed: 10 January 2018

on performance and portability aspects. They main components of OpenMANO include *openmano*, *openvim* and a graphical user interface.

OpenMANO's *openvim* is a lightweight reference implementation of an NFV virtual infrastructure manager that has been optimised for Enhanced Platform Awareness (EPA) to enable high-performance vNF deployments. Some of the EPA features include CPU, memory, NUMA pinning (assigning specific hardware elements to vNFs) and the support for pass-through and virtualised interfaces (e.g., SR-IOV). On top of managing vNFs by calling the appropriate APIs of an enhanced OpenStack deployment, *openvim* also talk to a SDN control for network service configurations. Their other component, *openmano* is a reference implementation of the management software (offering the creation and deletion of vNF templates and instances as well as network service templates and instances) that interface with *openvim* through its APIs, and offers a northbound REST interface to their GUI.

As for orchestration, OpenMANO can be used with three different VM providers. If integrated with OpenStack, it will be using the default placement algorithm of OpenStack as described for OPNFV before. In case OpenMANO is integrated with VMware's vCloud Director, it uses VMware's proprietary placement algorithm that optimises for high-availability and balanced load for hosts. OpenMANO can also be integrated with public clouds (e.g., Amazon Web Services), where physical placement of vNFs can not be explicitly controlled.

T-NOVA T-NOVA [23] (Network Functions as a Service over Virtualised Infrastructure) is a recently completed 3-year European-funded project that involved participants from the ETSI NFV standardisation body. The project was driven by enterprises and aimed to offer an integrated solution for the deployment and management of virtualised network functions over converged (network and computer) infrastructures.

On top of a fully fledged ETSI NFV MANO implementation, T-NOVA has successfully presented one of the first marketplaces for network services with accurate billing of services. Their proposed interface has made the click and deploy vision of NFV a reality. As seen in Figure 2.5, operators are able to set many parameters for vNFs, such as monitoring require-

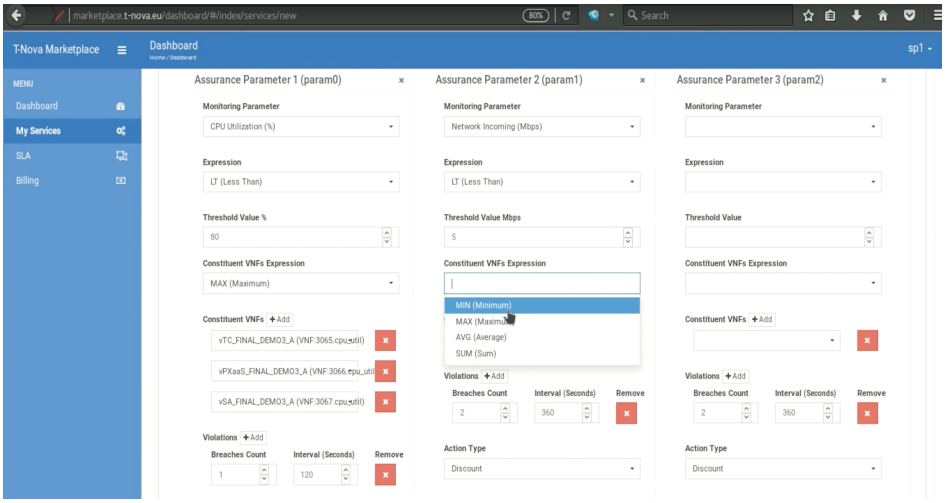


Figure 2.5: T-NOVA marketplace: clicking and deploying vNFs [23]

ments for vNF's CPU usage, among others.

T-NOVA presented a separate software component called TeNOR [65], a vNF orchestrator that performs mapping of virtual services to physical points of presence (PoP) in different network domains. The orchestrator optimises for overall delay, cost of deployment and overall network utilisation at the same time.

ZOOM The ZOOM[8] (Zero-touch Orchestration, Operations and Management) project is an active collaborative project managed by the TM Forum, a global industry association that drives collaboration and problem solving for communication and digital service providers and their ecosystem suppliers.

The ZOOM project, on top of being a ETSI NFV MANO implementation itself, has two distinct characteristics. First, there has been considerable amount of work put into transitioning to NFV by aligning the virtual models of the ZOOM system to industry standard infrastructure management software components. Also, ZOOM has had many small proof-of-concept projects (called catalyst projects) showcasing emerging ideas. Past catalyst projects were led by major network operators and included, among others, the use of predictive analytics for vNF resource management, and the integration of 5G properties to the NFV ecosystem.

ZOOM itself has also been built on top of the OpenStack VM management software, there-

[8]https://www.tmforum.org/collaboration/zoom-project/ Accessed on: 10 January 2018

fore by default it utilises the VM placement algorithm of OpenStack, as described for OPNFV before.

UNIFY The UNIFY [19] project was a recently completed European-funded project showcasing unified programmability over cloud and carrier networks. The idea behind this project was to control both a cloud and an internal carrier network in a unified way, so that vNFs can be deployed and managed with high level of agility.

While the UNIFY project was mainly focused on open interfaces in all layers of the presented unified architecture, the project has also published a few vNF placement algorithms. In one of them, Nemeth et al. [66] presented a fast and efficient algorithm to map constantly arriving service graphs to large networks consisting of tens-to-hundreds of thousands of nodes. Their approach uses a preference value based greedy backtrack algorithm based on graph pattern matching and bounded graph simulation ordering substrate nodes to be selected for a service chain by a composite preference value based on (1) node utilisation, (2) path length measured by latency and (3) average link utilisation on the path leading to the node.

The UNIFY project has also experimented with resource-constrained CPEs and envisioned a similar vision to this thesis: placing services to close proximity of the users (e.g., IPSec terminators, low-latency services) directly on the CPEs. In their work presented in [67], they focused on the network abstractions and APIs required to run a native (i.e., not virtualised) network function on resource-constrained CPE.

NetFATE NetFATE [26] proposes an open framework to enable network function at the edge. Similar to the framework presented in this thesis, they also envision vNFs such as firewalls, routers, WAN optimisers or a VPN termination running at low-cost customer premise equipment. NetFATE uses traditional XEN VMs along with OpenFlow, controlled using the POX SDN operating system.

NetFATE has presented a very simple and preliminary decision algorithm for vNF orchestration in their paper [26]: for each required service, they run a VM on the access CPE node of the traffic requesting it. If a user moves from one site to another and therefore changes

his/her access CPE, all the VMs implementing the requested vNFs are migrated to the new access CPE.

OSM Open Source MANO (OSM)[9] is an operator-led, ETSI-hosted project to develop an Open Source NFV Management and Orchestration (MANO) software stack aligned with ETSI NFV, backed by companies mainly in Europe - such as Telefonica.

OSM promises a VIM-independent, production quality software stack that can oversee multiple geographically distributed NFV deployments that can use different management software. This is to increase interoperability among NFV implementations with a strict and well defined model.

Open-O The Linux Foundation formed the OPEN-Orchestrator Project (OPEN-O)[10] in 2016 to develop an open source software framework and orchestrator to enable agile SDN and NFV operations, back by Asian companies and vendors such as China Mobile and Huawei.

Their focus, similar to OSM introduced before, is to create deeper convergence among SDN and NFV technologies and innovation in the networking industry as well as supporting multi-vendor deployments. However, while OPEN-O is expected to build more around OpenStack, OSM is taking a neutral approach by pushing a common information model.

CORD CORD (Central Office Re-architected as a Datacenter)[11] combines NFV, SDN, and the elasticity of commodity clouds to bring datacenter economics and cloud agility to the Telco Central Office. This effort builds on top of the well-known SDN controller, ONON, led by the Open Networking Foundation (ONF).

CORD lets the operator manage their Central Offices using declarative modeling languages for agile, real-time configuration of new customer services.

[9] https://osm.etsi.org Accessed on: 08 July 2018
[10] http://open-o.org Accessed on: 08 July 2018
[11] http://opencord.org Accessed on: 08 July 2018

Kubernetes Kubernetes [68] is a new production-grade container orchestrator designed to automate deploy, scale and manage containerised applications. It was originally designed by Google and is now maintained by the Cloud Native Computing Foundation. It has been created to work with a range of container tools, including Docker that is used in this Thesis.

As of today, Kubernetes's scheduler (the component that decides where to host containers) takes available resources only from the hosting nodes into account. Since Kubernetes is not integrated with information from the network (e.g., link latency information), it does not place containers to latency-optimal locations as it is presented in this Thesis.

Kubernetes has recently been proposed for NFV by Intel[12]. In this work, Intel has reported that they are doing considerable amount of work to extend the networking capabilities of Kubernetes in order to separate multiple virtual networks and to support Intel SR-IOV, Intel's high performance network IO virtualisation feature. The NFV solution presented in this Thesis predates these efforts.

2.3 Virtualised Network Service Deployment

As outlined in the section before, introducing programmability and virtualisation to both the data and the control plane has got paramount attention from the networking community for over 30 years. As a result of these efforts, technologies such as SDN and NFV are now available for network providers to automate, program and control specific parts of the network from software.

This section first introduces the wide spectrum of hardware-based network appliances that are installed today. Then, it focuses on the benefits of virtualising these services using SDN and NFV and in particular, it describes the advantages of running services in close proximity of the end users (e.g., at the network edge).

[12]https://builders.intel.com/docs/networkbuilders/enabling_new_features_in_kubernetes_for_NFV.pdf Accessed on: 8 July 2018

2.3.1 Traditional In-Network Middlebox Services

Today, network operators rely on a wide spectrum of hardware-based appliances or 'middleboxes' to transform, inspect, filter or otherwise manipulate traffic on top of routing. In recent years, middleboxes have become fundamental parts of operational networks, providing approximately 45% of the network devices to enforce security policies (e.g., firewalls, intrusion detection and deep packet inspection services) or performance guarantees (e.g., WAN optimisers, proxies, load-balancers) throughout the topology [69].

The most commonly deployed middleboxes are [21, 69]:

1. Network Address Translator (NAT) [70]: a NAT middlebox allows the modification of source and destination IP addresses. Typically it is deployed at the edge of a private network to hide multiple internal hosts behind one public IP address to conserve global IP address space.

2. Firewalls (FW) [71]: A firewall is utilised to filter traffic based on pre-defined policies. Policies usually concern packet header fields (e.g., IP source or destination) or inspect the payload.

3. Intrusion Detection Systems (IDS) [72]: Similar to firewalls, IDS systems are usually deployed at the edge of the network to inspect incoming and outgoing traffic. However, in contrast to firewalls, IDS systems are not usually designed to make accept/decline decisions in real-time. Rather, they are deployed to analyse anomalies and report findings to network operators. As an example, an IDS system can report if the number of connections has exceeded the usual number in a particular time window.

4. Load Balancer (LB) [73]: Load balancers are designed to equally split network flows (or packets) between multiple servers with the aims of optimising resource usage, avoiding overloading servers, minimising response time, maximising system throughput and to introduce fault-tolerance for certain services.

5. WAN Optimiser: A WAN optimiser can optimise traffic on wide-area network links

in order to improve performance or to reduce high costs of wide area bandwidth. This can involve modifying some transport layer properties for flows or simply compressing and caching data at the edge of the network.

6. Flow monitor: A flow monitor can be utilised to collect flow information (e.g., flow size, number of packets, protocols used) from all flows that are directed to it (e.g., by mirroring a port from a backbone router). These middleboxes help understanding the traffic and provide information that can be used for network planning and routing optimisation.

Recent studies have shown that the advent of diverse consumer devices that rely on different, network-intensive cloud services as well as the growing need for in-network security will increase the demand for middleboxes even further [74]. Despite their expanding popularity, hardware-based middleboxes have significant drawbacks: (1) they incur significant capital investment due to being provisioned and optimised for peak-demand, (2) are cumbersome to maintain due to the expert knowledge required, and (3) cannot typically be extended to accommodate new functionality as new operational requirements emerge. Moreover (4), their proprietary software on which they run, limits innovation and creates vendor lock-in [2, 69].

In the following section, the virtualisation of these middlebox services is introduced to overcome the aforementioned limitations.

2.3.2 Virtualising Middlebox Services

The list below provides commonly considered and already deployed network functions from various mobile and residential network providers. The list contains a collection of proof of concept (POC) trials as well as information on virtualised services running in production networks.

1. Virtualisation of network switching and routing elements: Examples include virtualised routers, software implementations for a carrier-grade NAT or virtualisation of

the Broadband Remote Access Server (vBRAS) that provides session management for broadband subscribers from software. vBRAS has been a key NFV application for many large mobile network operators, such as ZTE that have recently released a vBRAS solution for is capable of connecting and managing the signalling for 10 million subscribers[13].

2. Virtualised Customer Premises Environment (vCPE): vCPE is a service delivery model where network functions embedded in customer premises equipment are virtualised (and potentially moved to a data centre) to improve agility and reduce cost. vCPE, similar to vBRAS has also been in the focus for network operators (e.g., British Telecom, China Mobile) as well as network equipment vendors (e.g., Juniper or NXP) to facilitate the transition. A recently completed proof of concept trial between China Mobile and NXP have shown a vCPE system that supports isolation between services and provides guaranteed security and reliability between the edge computing gateway system and a remote cloud running the services[14].

3. Application-level optimisation services: Content Delivery Network (CDN) nodes, cache nodes, load balancers, application accelerators are deployed in vNFs for flexibility and resource efficiency. Akamai, a global leader in CDN services, in collaboration with Orange, France's largest residential and mobile network provider have presented a proof of concept virtualised CDN service that dynamically allocates CDNs in real-time and cost efficiently respond to rapidly changing capacity requirements such as large live events or major software updates rolling out globally, among other use cases[15].

4. Network security functions: firewalls, virus scanners, spam protection and intrusion detection systems are also commonly virtualised [75]. A proof of concept demonstration in 2014 by Intel and Brocade has showed various Layer 2-4 DDoS attacks, e.g. NTP reflection attack, being handled in real-time using a VNF router. This PoC

[13]http://www.zte.com.cn/global/about/press-center/news/201711ma/1124ma Accessed on: 14 February

[14]https://globenewswire.com/news-release/2018/01/16/1289624/0/en/NXP-vCPE-Solution-Successfully-Completes-CMCC-Networking-Trial.html Accessed on: 14 February 2018

[15]https://www.prnewswire.com/news-releases/akamai-collaborates-with-orange-on-nfv-initiative-to-dynamically-scale-cdn-capacity-for-large-events-300223273.html Accessed on: 14 February 2018

also characterised the performance impact of implementing Layer 2-4 DDoS attack detection and mitigation schemes in a VNF router[16].

5. Traffic analysis elements: Analysing traffic can be done for security purposes (e.g., content filtering, parental control), for accounting (e.g., bandwidth usage) or measuring Quality of Service. An example analytics software suite has been proposed by Nokia allowing operators to collect and process network statistics in traditional VMs and scale monitoring capacity according to demand[17]. Other vendors, e.g., Ericsson, are also working on software components that allow providers to look into real-time insight of network properties[18].

6. Service assurance, diagnostics functions: many operators deploy software components that periodically check the health of the network (e.g., latency or available bandwidth). A prominent software suite, PerfSonar[19] is used by wide area operators such as ESnet or REANNZ to periodically test bisection bandwidth and latency between PerfSonar software instances deployed worldwide.

7. Tunnelling gateways and Virtual Private Network endpoints: these software elements allow secure connections (e.g., by using IPSec/SSL) to remote endpoints. This is particularly important to be deployed at the customer edge to create secure connections between the customer's multiple networks. Such solutions are current linking together multiple research labs under GEANT, the European Research Network Operator with a software called MD-VPN (Multi-Domain Virtual Private Network)[20].

The list of virtualised network services is continuously growing, as operators are expanding their services with this new service delivery model and vendors are investing to support open

[16]https://www.brocade.com/content/dam/common/documents/content-types/datasheet/brocade-interop-sdn-ddos-mitigation-flyer.pdf Accessed on: 23 February 2018

[17]https://networks.nokia.com/solutions/deepfield-ip-network-analytics-DDoS-protection Accessed on: 26 January 2018

[18]https://www.ericsson.com/ourportfolio/it-and-cloud-products/expert-analytics Accessed on: 26 January 2018

[19]http://www.perfsonar.net Accessed on: 26 January 2018

[20]https://www.geant.org/Services/Connectivity_and_network/Pages/VPN_Services.aspx Accessed on: 26 January 2018

standards and technologies such as SDN and NFV. These proof of concept trials and production deployments have already achieved industrial awareness and confidence in virtualising network services making it a workable and trusted technology. In the following section, the benefits are further analysed.

2.3.3 Benefits of Service Virtualisation for Providers

2.3.3.1 Reduced Equipment Need and Operational Costs

By running network services in software and consolidating multiple of them to a single server, NFV helps operators reduce capital and operational expenditure by exploiting the economies of scale of the IT industry. Recent studies have reported the cost reducing benefits of NFV and presented up to 95% saving of deployment cost and up to 25% in transport costs, among other categories [76].

Approaches to outsource virtualised network functions to public clouds have also been analysed by Sherry et al. [69]. In this work, the authors have shown that is is possible to outsource over 90% of middlebox hardware from the internal network to a distant cloud and therefore making middlebox management someone else's problem. This outsourcing provides savings in capital expenses, the same way as moving computing services to cloud VMs does and only introduces an average latency penalty of 1.1ms and a median bandwidth inflation of 3.8%.

Reducing power consumption in particular is becoming increasingly important to meet regulatory and environmental standards and to reduce the most expensive part of a provider's operational expenditure, the electric power that is reported to account for 10% of the current operational expenditure of network providers, and it is likely to rise in the next years, according to Bell Lab's GWATT tool [77]. By consolidating services to fewer number of physical hosts, power consumption can be reduced drastically [78].

2.3.3.2 Software-Based Practices

NFV introduces the benefits of software practices to networks. Software solutions are inexpensive compared to hardware appliances as they eliminate the cost of the periodical rebuild or upgrade of the security system and the cost of maintaining vendor-specific knowledge. Software solutions can also benefit vendors: they allow them to put more effort in reducing the complexity of managing and re-configuring their products by providing easy to use programming interfaces. Furthermore, updating or upgrading software services is a matter of dynamically retrieving new source code of software components rather than extending or replacing hardware equipment.

The challenges of software-based solutions include the risk of having not properly tested code deployed in production networks. Software solutions with frequent changes can also often cause interoperability issues that need to be considered. To mitigate this problem, Niwa et al. [79] have proposed an universal fault detection solution for NFV to detect software bugs causing memory leaks, packet congestion or session congestion.

2.3.3.3 Efficient and Flexible Resource Provisioning

The rapid and easy deployment of vNFs increases the system's flexibility to react to events such as changing traffic dynamics, dynamic resource migrations (e.g., mobile users moving in the network) or the adding of new security functions. Therefore, it increases the efficiency of the system to handle changes in requirements. The benefits of NFV resource orchestration will be studied in detail in Section 2.4.

vNFs can also offer dynamic up and down scaling on-demand, leveraging the system's ability to handle traffic changes and at the same time maintain an efficient management of resources which is considered a more efficient approach to the fixed, under-utilised resources of hardware middleboxes [80, 81, 82].

2.3.3.4 Modularity and Chaining of vNFs

As vNFs are implemented in software, they allow efficient modularisation of security services, and small component reuse to build more complex and customised security systems. The modularisation encourages developers and vendors to focus on building more efficient, albeit smaller modules instead of large monolithic applications. As a concrete example presenting modularity, one could build a high performance IDP (Intrusion Detection and Prevention) vNF by using a high-performance packet processing library (e.g., Intel DPDK), a software switch (e.g., Open vSwitch [53]) and an open-source IDP software (e.g., Bro [83] or SNORT [84]). Moreover, modularisation allows scaling only the parts that need scaling and the chaining of vNFs to apply complex security policies. A common service chain consists of packet classifiers and firewalls or IDP functions that are only used for specific set of traffic (identified by the packet classifiers).

2.3.4 Moving Services to the Network Edge

The concept of distributing network intelligence provides the foundation upon which next-generation networks are built. In this section, the current trends in telecommunication networks motivating the need for distributed, edge network intelligence are outlined, along with notable research directions. The following sections also highlight the key characteristics of the network edge and present example edge devices.

By 2020, it is estimated that the number of connected devices will grow exponentially to 50 billion, according to telecommunication market leaders Cisco and Ericsson [1]. As shown in Figure 2.6, new devices include smart mobile phones, wearables (e.g., smart watches, fitness trackers), household electronics equipped with internet connectivity (e.g., smart fridge, smart TV), large-scale sensor networks and actuators.

This growth of mobile and smart devices, and the handling of the generated data challenge network operators in two ways. First, operators have to cope with an exponentially increasing amount of data that has to be efficiently transferred for users with different service level

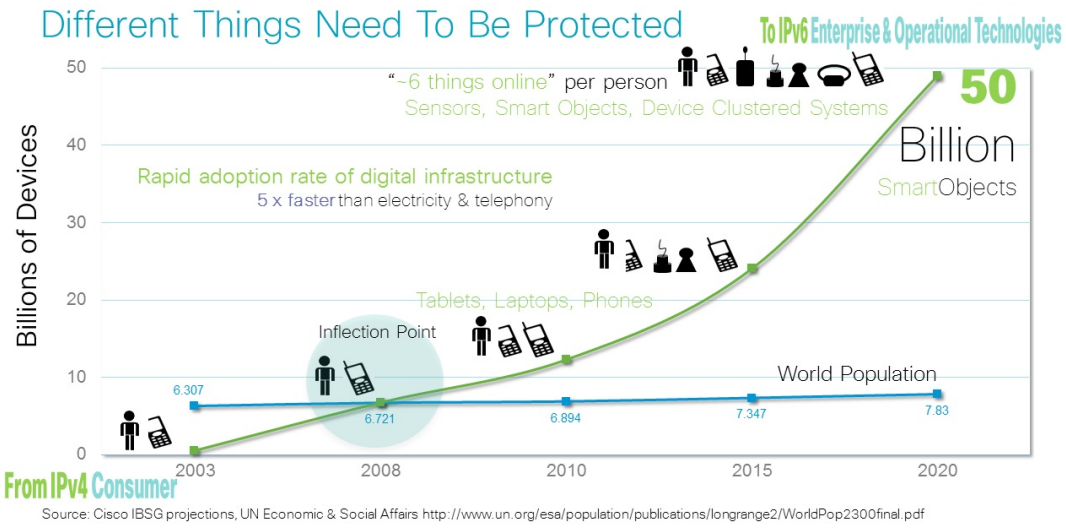


Figure 2.6: IoT Growth (source: Cisco [85])

requirements. Also, at the same time, operators are currently striving to provide new, customised and high-performance services to their mobile users to satisfy increasing expectations from the network such as, for example, always-on connectivity, secure data processing, enabling high-definition video streaming and advanced firewall protection.

2.3.4.1 Characteristics of the Network Edge

The network edge promises to deliver customised, high-performance services to devices (mobile phones, sensors) while reducing unnecessary utilisation of the core of the network [86]. In the following, the key benefits of the network edge are summarised.

Proximity to the Data Being close to the source of the information is important to capture key information for real-time analytics and big data processing at the edge. This also means that congestion can be minimised in other parts of the upstream network. As an example, the authors of GigaSight [87] have shown an architecture that is capable of performing video analytics at the edge of the Internet, therefore reducing the demand for ingress bandwidth.

Low Latency The latency for a mobile service can be considered as aggregation of the following delay components: propagation, transmission, processing and queuing. Specifically, propagation delay is the time required for a packet to travel from the sender user to the receiver vNF, which is a function of distance over speed with which the signal propagates. This distance is typically in the order of tens-of-meters for the cases of dense, small-cell networks (e.g., home WiFi, 5G base station [88]) and are typically no longer more than a kilometre (e.g., company premise equipment) for general network edge cases. This short distance results in a low propagation delay that can be exploited when services are hosted on the edge devices. Predictable low latency can be utilised to react faster to certain events, to significantly improve user experience and to enable new application that can not tolerate delays (e.g., tactile internet, remote driving, factory automation) [89].

Location Awareness In case an edge device is part of a wireless network (e.g., WiFi or cellular), it can leverage low-level signalling information to determine the exact location of each end user. This enables new application to businesses, such as high-accuracy inbound localisation for assisted living [90].

Network Context Information Edge devices can offer context-related information that can differentiate the mobile broadband experience and be monetised. New applications can be developed to connect mobile subscribers with local points of interests, events and businesses [91].

On-Premises Privacy Protection An edge device can be located on-premises, meaning that it can be isolated from the public Internet. This is especially important when proprietary or health-related sensitive data has to be managed using secure, on-premise services and therefore are not sent to distant cloud servers or the public Internet [92].

2.3.4.2 Trends in Edge Computing Research

Edge computing is currently a trending research topic, tackled from various angles. This section gives a high level overview of some of these topics that have one in common: to introduce an intermediate layer of computation (or storage) between the user's device and the cloud.

Fog Computing Despite the increased adoption of cloud computing, there are still issues unsolved due to the inherent problems such as unpredictable latency, lack of mobility support and location-awareness. Fog computing addresses these problems by providing elastic resources and services at the network edge [4]. This capability of distributed intelligence fog in the network is a core architectural component of next generation IoT networks for three main reasons [85, 93]:

1. Data collection: collecting data in a central location has scalability issues when processing has to be done for billions of active IoT devices.

2. Network resource management and bandwidth conservation: As network bandwidth can be scarce (especially in the core of the network), transferring data to the cloud unavoidably leads to large network utilisation that can be mitigated by distributing data processing elements.

3. Closed loop functioning: For some use cases, IoT requires a reduced reaction times. For instance some sensors (e.g., alarms) need to communicate with relevant actuators in short timescales, without unacceptable delays caused by sending and receiving data from the cloud.

Mobile Cloud Computing As a well-known early work proposing the benefits of the edge for smartphone users, the authors in [94] have presented the work and term called cloudlets. Their system brings the cloud to the mobile users by exploiting computing services in their spatial vicinity to leverage cloud computing free of WAN delays, jitter, con-

gestion and failures. The cloudlets architecture has been influential for many research papers (notable works include MAUI [95] or CloudCloud [96]) suggesting the offloading of computation from mobile devices to the cloud or nearby wireless hot-spots equipped with computational resources that is referred to as the network edge today.

Network Functions at the Edge This work is not the first to explore network functions at the network edge. As an example Lombardo et al. [26] have presented the NetFATE (Network Functions At The Edge) architecture that aims putting vNFs at the edge of the network. Their platform is based on free and open source software on both the provider's and customer's equipment, allowing function deployment simplification and management cost reduction. However, their solution does not truly support edge devices without virtualisation support and moreover, they do not present an orchestration algorithm for their system.

Web-Scale Edge Deployments Recently, two papers have been presented from industry that are taking advantages of an edge infrastructure. First, Google has presented Espresso [28], an SDN-based Internet peering edge routing infrastructure. Espresso uses commodity switches and host-based routing/packet processing to implement a novel traffic-engineering capability. Their edge infrastructure consists of commodity devices at peering locations - a.k.a edge metros that connect Google to end users around the world. In their paper, they propose how simple caching and distribution of static content at the edge can deliver substantial improvements to video buffering.

Edge Fabric [29], published by Facebook, has similar traffic engineering goals to Espresso from Google. Edge Fabric proposes a system that provides connectivity for two billion users along with near real-time congestion avoidance at the edge of Facebook's network. To realise the edge, Facebook deployed more than 20 points of presences (PoPs) globally and directed user traffic to the edge locations with careful DNS configurations and BGP advertisements.

Commercial Applications for the Edge Many equipment vendors have presented industrial solutions promoting the edge. Examples include ADVA's Network Edge[21] devices that allow vNF services (covering demarcation and performance assurance, security and synchronisation network functions) to be hosted on enhanced, yet simplified forwarding devices.

As an other example, an ARM-ecosystem partner, Virtosys[22] have recently presented an ARM-based edge device that is capable of running small analytics network functions for local, nearby IoT devices.

2.3.4.3 Example Devices at the Edge

In recent years, customer edge devices have become smarter and their capabilities have increased to run advanced network services, such as Quality of Service (QoS) differentiation, parental control filters, bandwidth reservations, and other multi-service functions.

In Table 2.1, a few popular edge devices are presented alongside their released date, architecture, CPU and memory parameters. The list includes large-scale residential customer premises equipment (CPE) from the UK (Virgin), US (Google Fiber) and France (Orange) as of 2017. On top, a few low-cost commodity home routers have been added to this list along with three IoT gateway devices from HP Enterprise, Dell, and NEXCOM that can be used for data processing and storage.

As it can be observed from Table 2.1, recent CPE devices and home routers are equipped with relatively powerful computing capabilities (e.g., CPUs up to 1.6 GHz) and sizeable RAM (up to 1GB) to run a Linux-based operating system (e.g., OpenWRT or DD-WRT), and some lightweight network functionality as demonstrated in [27], where a commodity TP-Link home router with 560 MHz CPU and 128MB of RAM was used to run multiple vNFs (rate limiting, content filtering, and firewall vNFs).

Apart from the low-cost edge devices such as home routers and residential CPE, some vendors have also introduced IoT gateways with high-end CPUs and up to 64 GB of RAM to

[21] https://www.advaoptical.com/en/newsroom/press-releases/20161107-adva-optical-networking-introduces-one-network-edge Accessed on: 15 February 2018

[22] http://www.virtuosys.com Accessed on: 19 Oct 2017

accommodate services such as intelligent data analytics at the edge of the network. We envision all of these devices (residential CPEs, home routers and IoT gateways) along with other in-network NFV servers to be part of a distributed edge infrastructure where vNF services can run.

Customer Device	Released	Architecture	CPU	Memory
Residential CPE Home Routers				
Virgin SuperHub 3 (Arris TG2492S)	2015	Intel Atom	2x1.4 GHz	2x256 MB
Google Fiber Network Box GFRG110	2012	ARM v5	1.6 GHz	not known
Orange Livebox 4	2016	Cortex A9	1 GHz	1 Gb
Commodity Wireless Routers				
TP-Link Archer C9 home router	2016	ARM v7	2x1 GHz	128 MB
Ubiquiti EdgeRouter Lite 3	2014	Cavium MIPS	2x500 MHz	512 MB
Netgear R7500 Smart Wifi Router	2014	Qualcomm Atheros	2x1.4 GHz	384 MB
IoT Edge Gateways				
Dell Edge Gateway 5000	2016	Intel Atom	1.33 GHz	2GB
NEXCOM CPS 200 Industrial IoT Edge Gateway	2016	Intel Celeron	4x2.0 GHz	4GB
HPE Edgeline EL4000	2016	Intel Xeon	4x3.0 GHz	up to 64GB

Table 2.1: Example edge device specifications [6].

2.4 Orchestrating Virtual Network Functions

One of the main challenges of deploying NFV is to achieve fast, scalable and dynamic placement management of vNFs and vNF chains, as outlined in Section 2.3. Orchestration of vNFs has to take into account server-side properties (e.g., CPU, memory, IO resources available on a particular hosting device), as well as network parameters (e.g., keeping the latency between vNFs and the user low, making sure network links are not overloaded and connectivity can be made for all vNF assignments) that could change dynamically after an initial allocation has been made. This is further complicated in an edge vNF scenario, where the physical movement of users between edge devices require movements of associated vNFs.

In general, vNF orchestration problems are similar to the well-studied Virtual Machine (VM) orchestration problems applied in clouds and Data Centres [97] [98]. However, while VM orchestration algorithms share similarities with vNF orchestration as they take server side resources (e.g., CPU, memory available) into account, they usually do not take network properties into account and therefore can not be directly applied for vNF orchestration.

The next two subsections will detail the most prominent work in NFV resource allocation and dynamic orchestration.

2.4.1 NFV Resource Allocation Strategies

Allocating vNFs is a NP-hard optimisation problem, as it can be seen as a generalisation of the Virtual Network Embedding (VNE) problem that has been proven to be NP complete [99]. For this reason, resource allocation algorithms can be divided into three categories: exact solutions, heuristic solutions and meta-heuristic solutions.

Exact solutions propose optimal techniques to solve a small instance of a problem and provide a baseline to evaluate heuristic-based solution. Heuristics solutions do not necessarily produce an optimal vNF allocation, but they provide a solution for even large problem instances, while keeping execution time low. Meta-heuristic solutions can improve some

heuristics solutions that can get stuck in a local optimum far away from the real optimum. In the following sections, examples will be given to all of these categories.

2.4.1.1 Exact vNF Placement Solutions

Exact (i.e., optimal) solutions typically use linear programming (LP, also called linear optimisation) that achieves the best outcome in a mathematical model whose requirements are represented by linear relationships. Linear programs can be expressed in canonical forms as:

$$\begin{aligned} \max_{x} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{2.1}$$

where x represents the vector of variables (unknown, to be determined), c and b are vectors of known coefficients, A is a known matrix of coefficients, and $(\cdot)^{\mathrm{T}}$ represents the matrix transpose.

In particular, Integer Linear Programming (ILP), a generalisation of linear programming that restricts variables to be integers, is well suited to formulate placement problems, as an integer decision variables can express a particular vNF-host placement naturally, as shown later in Section 3.5. However, while linear (non-integer) programs can be solved efficiently in the worst case, integer programming problems are in many practical situations (those with bounded variables) NP-hard [100]. ILP problems, if feasible, can be solved by software programs called solvers such as the proprietary CPLEX[23] or Gurobi [101] solvers or the open-source GNU Linear Programming Kit (glpk) [102]. These solvers usually rely on algorithms such as branch and bound or branch and price [103].

A notable exact vNF placement solution was presented as VNF-P [104], where Moens et al. used ILP in an NFV-enabled hybrid environment where specific and general purpose hardware coexist. VNF-P, as many in this field, is aimed to minimise the number of physical devices used to save operational and capital expenses for a network provider. This work (due

[23] https://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/. Accessed on: 18 Oct 2017

to the scaling limitations of exact solutions) has been evaluated on a small scale network with various traffic loads, presenting findings that their algorithms finish in 16 seconds or less making it feasible to react quickly to changing traffic demands such as morning and evening peak time traffic fluctuation.

Another exact placement solution was presented in [105] where Bari et al. formulated a vNF orchestration problem that aims to determine the required number and placement of vNFs that optimises network operational costs and utilisation, without violating service level agreements. They present an evaluation implemented using the CPLEX solver and show that a vNF based approach for security and performance policies can bring more than $4x$ reduction in operational costs of a network.

In [106], Jang et al. focus on the placement of vNF service chains and formulate an optimisation problem to efficiently utilise the network resources (e.g., bandwidth and links). Their simulation results show that their linear programming based optimal solution can reduce the network resource usage and therefore accommodate more flows compared to non-optimal solutions.

In [107] Gupta et al. have proposed a linear programming approach to minimise the bandwidth consumed from routing traffic through selected paths by placing vNF service chains to optimal locations in a programmable network / server environment. They also introduced the term "Network-enabled Cloud" (NeC) that is an extension of a cloud environment with programmable packet and optical network nodes. The results presented in their paper show that distributed NFV capability can reduce network resource consumption.

In the TeNOR project [65] (which is part of the T-NOVA NFV platform introduced before), the authors have presented two models to map vNFs to different points of presences (PoP)s of an NFV infrastructure. Their objective function for an ILP model is a weighted sum of three components: (1) the cost of assigning vNFs to a particular PoP, (2) the sum of the overall delay, and (3) the overall resource link usage.

Finally, Baumgartner et al. [108] have proposed an optimisation problem for mobile core networks to place vNFs onto nodes of the physical substrate network and to determine op-

timal traffic routing between them. Their work, as many in the field, minimises the cost of occupied link and node resources. They show simulation results using two nation-wide network topology examples and present results that outperform traditional VNE optimisation approaches.

2.4.1.2 Heuristic vNF Placement Solutions

Heuristic, non-exact solutions have been proposed by many researchers in order to minimise the execution time to find a placement solution. These solutions usually rely on fast algorithms that do not necessarily reach optimal resource allocation and can be stuck in a local optimum, yet provide a relatively good solution with short execution time for even large problem instances [109].

A heuristic solution for vNF placement is presented by Beck and Botero in their work CoordVNF [110]. CoordVNF is a heuristic method that coordinates the composition of vNF service chains and their embedding into the substrate network. Their system aims to minimise bandwidth utilisation in the network, while computing results within a reasonable time compared to an exact approach. Results present that CoordNFV can produce an allocation in 7ms as opposed to an exact approach running for almost 13 hours on the same network.

Another heuristic solution by Bruschi et al. targets energy-efficiency [111]. Their solution presents a game-theoretic approach for vNF resource allocation by seeking the minimisation of individual cost functions, and compares the power consumption and delay achieved with energy-aware and non-energy-aware strategies. Results present at least 10% saving of the power consumption and delay product using the energy-aware strategy.

In an other example, Riggio et al. [112] have proposed a management and orchestration framework for enterprise wireless networks. This ETSI NFV compatible framework proposes a heuristic-based algorithm that optimises vNF placement according to application level constraints (e.g., latency). Their proposed system has been evaluated using simulated network topology aiming for high number of accepted service chain requests and low node and link utilisation.

In [113] Mohammadkhan et al. formulated the problem of network function placement as a Mixed Integer Linear Program (MILP). This formulation not only determines the placement of services and the routing of flows in the network, it also seeks to minimise resource utilisation. They have developed a heuristic to solve this problem incrementally, allowing them to support large problem instances and to solve the problem for incoming flows without impacting existing flows. Their approach demonstrates that partitioning a large problem to smaller pieces can produce a close to optimal result.

Finally, a heuristic algorithm for vNF placement has been presented by Nemeth et al. in [66] for carrier grade networks by using a real-time service graph mapping based on a greedy backtracking. In their work, they focused on minimising the time it takes to compute a placement for an online variation of the vNF placement problem, where very large number of vNF placement requests arrive within few a seconds. In addition, this solution allows fine-tuning parameters to achieve a desired acceptance ratio and quality of orchestration of the algorithm.

2.4.1.3 Meta-heuristic vNF Placement Solutions

Meta-heuristic algorithms can find near-optimal solutions by iteratively improving a solution using a discrete search space. As opposed to heuristic methods, meta-heuristics improve solutions by escaping from local optima in reasonable and predictable running times. Popular meta-heuristics for combinatorial problems (such as the vNF placement problem) include simulated annealing by Kirkpatrick et al., [114], genetic algorithms by Holland et al., [115] or tabu search [116] by Glover.

There has only been a few attempts to use a meta-heuristic approach for vNF placement. One of them has been presented by Mijumbi et al. in [117] by using a tabu search based on local (neighbourhood) search methods used for mathematical optimisation. Their simulation results show high acceptance ratio, low average flow execution time, and low embedding cost under varying network conditions, but showed only a small gain by using tabu search compared to traditional, less complex greedy algorithms.

In the TeNOR orchestrator [65] presented for the T-NOVA NFV framework, the authors have introduced a Reinforcement Learning (RL) approach to service mapping. The strength of their RL solution is that it is much faster than an ILP and it is easy to compute with iterative execution of simple equations. Also, the RL presented can learn the system dynamics and hence is able to maximise not just an immediate reward, but rather the expected total reward in the long run.

2.4.2 Dynamic vNF Orchestration and Re-Allocation Strategies

Network parameters (e.g., cost of certain links, queue length in switches, latency on certain links, location of users) are constantly changing in any network, sometimes even in an unpredictable fashion [118, 119]. This is amplified in a mobile edge scenario presented in my thesis, where user movements are more frequent between edge devices (mobile cells or wireless routers), causing frequent changes in user-to-vNF latency impacting QoS.

This raises an important question for all vNF resource allocation strategy: *how* and *when* to re-compute the placement to efficiently react the new network conditions. A common goal for such dynamic orchestration schemes is to minimise the number of vNF migrations caused by re-calculation (since they incur significant network overhead while transferring vNF state between hosts [120]), while keeping QoS (e.g., latency from vNFs to users) satisfied.

There have only been a couple of systems dealing with such dynamic vNF orchestration after an initial placement has been made. As an example, Kim et al [121] have presented VNF-EQ, a dynamic placement algorithm of virtual network functions for energy efficiency (benefiting operators) and QoS guarantees (required by users). They have used a genetic algorithm for their dynamic orchestration and showed that their problem is a variation of the multi-constrained path selection problem known to be NP-complete. Comparing this thesis with VNF-EQ, this work looks at optimising end-to-end latency for users, while keeping the number of vNF migrations as low as possible as opposed to optimising energy consumption.

In the work of VNF-OP [122] on top of presenting an exact placement of vNFs, from Bari et al. have also investigated dynamic properties of vNF allocation to adjust locations when traf-

fic volumes change. In their dynamic, heuristic-based orchestration approach, they adapt the vNF locations to changing traffic conditions, resulting in fluctuations in energy costs, their optimised metric. In the contrary, the vNF orchestration presented in this thesis optimises for minimised latency from users to their vNFs, instead of CAPEX / OPEX savings.

2.5 Summary

This chapter has set the scene for the thesis by going through many efforts on introducing programmability to telecommunication networks over the past 20 years. As described on Section 2.2, the latest developments of NFV and SDN allow today's network providers to manage their network services in an efficient, cost-effective, and automated way. While SDN provides a centralised control plane for the network, NFV gives the ability to virtualise traditional middlebox services in software.

Then, Section 2.3 outlined the benefits of virtualising traditional middlebox services such as firewalls, caches and intrusion detection modules. The section has also highlighted opportunities that lie at the network edge - such as location awareness and low-latency service delivery that is becoming an important offering for service providers.

Section 2.4 provided a complete overview on vNF orchestration algorithms, discussing optimal, heuristic and meta-heuristic solutions. As shown, most of the vNF placement solutions optimise for server resources and do not react to changing network dynamics or provide latency-optimal orchestration.

Based on the previous sections, two main directions for this thesis can be identified:

- While the benefits of the network edge are clear, today's NFV frameworks have not been designed to incorporate a distributed edge infrastructure due to their heavy vNFs that can not be dynamically started and stopped on these devices. Therefore, there is a need for a lightweight NFV framework that enables even low-cost edge devices to be part of a distributed NFV framework.

- Extending the previous works on vNF orchestration, a dynamic, latency-optimal vNF placement solution is required to manage the aforementioned lightweight vNFs according to dynamic network properties and the provider's QoS offerings to leverage the potentials of such NFV framework.

In the next chapter, the thesis further analyses these limitations and collects the main design requirements and considerations for a lightweight, latency-optimal NFV framework.

Chapter 3

Designing a Lightweight, Latency-Optimal NFV Framework

3.1 Overview

As outlined in the previous chapter, traditional NFV platforms have been limited to standard virtualisation technologies (e.g., XEN, KVM, VMware, Hyper-V, etc) by running vNFs on fully fledged VMs using complete operating systems such as Windows, Linux and FreeBSD installed on them. However, this limits these systems' deployability on many devices (e.g., network edge or a public cloud VMs) and also poses constraints on the flexible management of network functions, since traditional VM-based vNFs can not be migrated, started or stopped quickly [6, 123].

Also, as it has been highlighted in Section 2.4, the previously examined vNF orchestration solutions do not take changing network dynamics or user movements into consideration and usually only provide a placement solution that optimises for efficient resource utilisation of the hosting servers. However, this does not support low-latency applications of NFV, where the goal is to minimise the latency between the user and vNFs [124].

To address these limitations, this chapter presents selected design considerations extending a generic ETSI NFV reference architecture [2]. The chapter first presents the motivations and

a high-level system requirements for the envisioned lightweight NFV platform in Section 3.2 and collects the main design considerations introduced by lightweight NFV in Section 3.3. Then, in Section 3.4, it describes and compares various lightweight alternatives to traditional VMs and justifies why containers have been selected for the designed NFV platform. Finally, Section 3.5 presents the fundamentals of dynamic, latency-optimal vNF placement orchestration that is required to efficiently place and re-allocate vNFs due to movements of the users or changes in temporal network dynamics (e.g., congestion, link failures).

3.2 System Requirements

This section provides the motivation for a new NFV framework by detailing the need for lightweight vNFs and latency-optimal orchestration on our target infrastructure described. Then, this section gives an outline on how would the system be operated in production and presents the core design elements of it.

3.2.1 The Need for Lightweight Network Functions

Network Function Virtualisation has been coined by large telecommunications network operators at the beginning of 2010s to virtualise in-network services in the core of the network by using commodity, yet powerful x86 servers and network cards instead of buying proprietary hardware appliances for network services like firewalls, caches and traffic analysers. This initial effort was focused around co-located servers hosting large VMs that were created to process large amounts of traffic in the core of the network, as described in Section 2.2.5.

In order to cope with the emerging requirements introduced by increasing number of mobile and IoT devices in Section 2.3.4, providers have started to virtualise the network edge and move vNFs from the core to the edge of the network, hosted on less-powerful, but physically much closer devices.

While the benefits of the network edge are clear, most traditional NFV platforms (e.g., Cloud4NFV [22], UNIFY [19], T-NOVA [23], OPNFV, NetFATE [26] introduced in Section 2.2.5) have been built on top of powerful commodity servers, exploiting VMs (using technologies such as XEN or KVM) for vNFs, therefore (1) adding a considerable overhead on top of services, (2) restricting fast and elastic service management and (3) often not being lightweight enough to run on low-cost edge devices due to their extensive memory, disk space usage or hardware requirements for virtualisation.

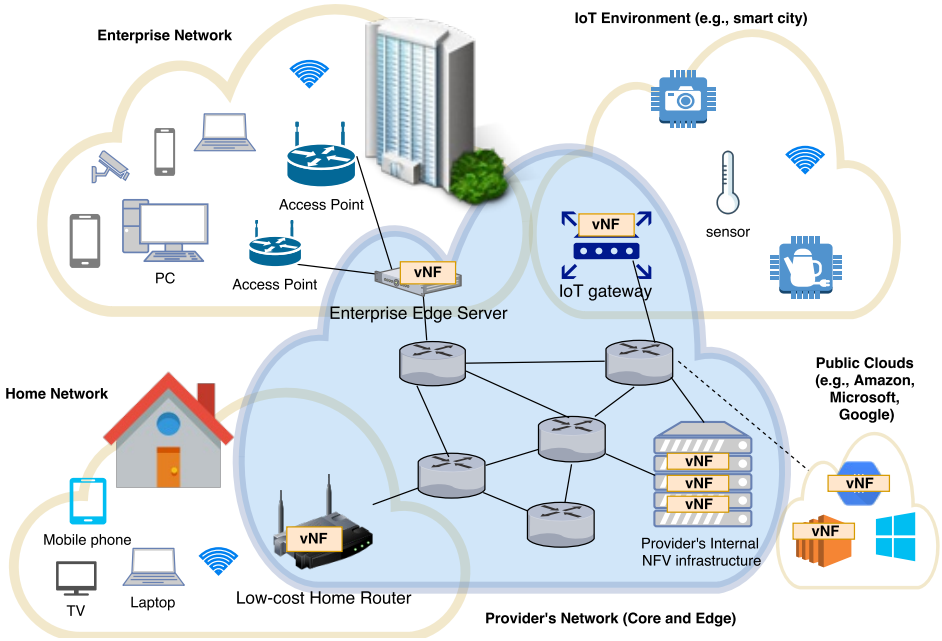


Figure 3.1: Target infrastructure for our NFV platform [6].

To solve the aforementioned challenges, the designed system will explore the options for lightweight NFV to enable small, lightweight vNFs to be run on a distributed infrastructure consisting of wide spectrum of underlying devices, such as a variety of edge routers, central NFV servers as well as public cloud VMs, as shown in Figure 3.1.

3.2.2 The Need for Dynamic, Latency-Optimal vNF Placement

Placement of vNFs (allocating vNFs to physical servers) has been one of the most studied topics in NFV during the recent years. Solutions from academia have been presented for optimal ([104, 105, 106, 107, 65, 108]), heuristic ([112, 113, 66]) and meta-heuristic place-

ments ([117, 65]), while industry systems have commonly utilised a heuristic placement scheduler (e.g., OpenStack's simple weight-and-filter scheduler). These placement solutions have either of the following limitations.

First, as outlined in Section 2.4, most placement solutions are designed to reduce the number of physical servers utilised for vNFs which can be translated to savings in both capital and operational expenditure (since less servers have to be purchased, managed and therefore powered on). While this objective is important inside the core of the network, the designed system explores a latency-optimal placement solution that allocates vNFs as close as possible to its users in order to support low-latency services.

Moreover, the orchestration algorithms presented so far do not take dynamic changes of network properties into account, such as latency fluctuations, user movements and different traffic patterns. However, these properties influence the user frequently, when vNFs are deployed on the edge, since for example users are likely to move between edge devices that increases latency between the user and their associated vNFs left behind in their previous edge node.

In contrast to previous vNF placement solutions, this design investigates a dynamic, end-to-end latency-based optimisation problem for lightweight edge vNFs. Such orchestration algorithms can rely on real-time topology and latency information along with QoS requirements defined by the network operator to figure out the optimal time to re-calculate the placement of vNFs that minimises the number of vNF migrations, yet sticks to the performance guarantees provided by the network operator.

3.2.3 Operational Overview

Operationally, this platform would provide a number of benefits for service providers. A core contribution lies in the ability of operators to create, deploy and manage in-network, latency-optimal services on-demand, in a matter of seconds instead of days or months. With numerous middleboxes currently deployed at various network operators, this platform offers service deployment using vNFs across a heterogeneous infrastructure, as shown in Fig-

ure 3.1. As pictured, the targeted infrastructure consists of IoT gateways that connect sensors to the Internet as well as enterprise edge servers connecting an entire organisation to the provider's network. Also, the designed platform should support home environments with low-cost home routers at the edge of their network to bring programmable network services as close as possible to the end users.

The proposed system is to be managed from either a user interface, or by directly calling APIs provided by the platform. In case an operator needs to assign a web cache to a user, the administrator should be able to log in to the web interface, find the user and click on a web cache to be assigned to the user. After this, the system would automatically figure out the best, latency-optimal location to host this vNF at and initiate the request in the background. An operator could also perform a similar vNF assignment using the provided APIs directly.

While the system is primarily focused on network and service providers, end users could also select from various vNFs from the envisioned platform. As an example, an end user might want to assign a web cache to their device and s/he could do this using a self service portal. After selecting a service, it will be deployed to a physical location selected by the vNF orchestration.

Each underlying component of this system is designed to be built from code. That is, changing a parameter in one of the vNFs is essentially changing configuration files and recompiling a new version of that vNF, instead of manually editing properties on a graphical interface of the software. This allows operators to automate the creation of new vNFs to the system.

Developing and maintaining a test environment for such a system is another key operational requirement. A test environment could replicate some properties of the production network and allow new services to be tested without impacting production traffic. With the help of SDN and various tunnelling techniques, the system components (e.g., edge devices or vNFs) can be tested with real, mirrored traffic inside development environments. Also, by using prototyping environments such as ESCAPE [125] (that utilises the well-known Mininet [126] software for SDN prototyping along with Click [127] software vNFs) and packet traces (e.g.,

CAIDA dataset[1]) or traffic generators [128] one could even test vNFs on a regular notebook.

Since this platform would enable user specific services, providers could also charge for these services individually to increase revenue. Therefore, the system will provide usage information to external software that could calculate billing per customer if necessary. The actual pricing of services is out of scope of this thesis, although some pricing models for NFV have been discussed by Gu et al. in [129].

It is important to note that there are clear trade-offs between complexity, security, deployability, and performance when designing and operating a system like this. As an example, while performance can be enhanced using acceleration technologies such as Intel DPDK and SR-IOV, they introduce complexity and restrict deployability to devices that support such technologies. Similarly, additional security considerations (e.g., traffic encryption, various layers of authentication of the APIs) could hinder deployability and introduce complexity. It is believed that such considerations have to be made by the providers, adjusting to their existing infrastructure, devices, users and their services.

3.2.4 High-Level Design Requirements

To achieve an architecture that fits its purpose, the related research has been reviewed, and current vNF platforms have been studied. From these findings, the following high-level design requirements have been distilled:

1. The platform should use vNFs that can run on multiple underlying devices, even on low-cost hardware that can be found on the network edge or residential customer premises.

2. The platform should allow the assignment of vNFs to individual user's traffic in a matter of seconds, supporting seamless migration of services.

[1] https://www.caida.org/ Accessed on: 16 January 2018

3. The platform should support public cloud infrastructures as hosting platforms for vNFs for additional capacity in case required (e.g., sporting events).

4. The platform should support signed vNF images for trust purposes to eliminate the risk of entrusted code to be executed on remote devices.

5. The platform should monitor the resource utilisation of vNFs and provide health information of the underlying infrastructure.

6. The platform should support flexible, latency-optimal placement of vNFs based on temporal network properties and location of end users which is crucial when services are deployed at the edge of network.

7. Transparent traffic routing should be used to keep on-going network connections alive and to be able to swap vNFs without affecting users' traffic.

8. The platform should be open-source to enable experimentation and innovation.

9. An easy to use, graphical management interface needs to be provided for operators to assign and monitor vNFs.

These design requirements direct the work in two ways. First, they influence the underlying technology required to implement many system components. Also, some requirements outline an optimisation problem for latency-optimal vNF placement.

3.3 Design Considerations

This section identifies a few design considerations for a lightweight NFV platform to allow running vNFs on network edge devices and supports quick migration of vNFs across the infrastructure in a safe and high-performance manner. These considerations extend many general considerations introduced in the general ETSI NFV standard [2] and the IETF draft on "An Analysis of Lightweight Virtualization Technologies for NFV" [123].

3.3.1 Providing Continuity, Portability and Elasticity

vNF service continuity can be interrupted due to several reasons: hardware failures, software problems of the vNF, unavailable physical resources, or vNF migrations. This is especially important when vNFs are hosted on commodity, customer premise equipment which has limited control from the network provider. Therefore, some considerations have to be made to minimise the downtime of network services. One way to provide continuity is by introducing fault tolerance to the system. As an example, important vNFs can run on multiple hosting devices in the network and be used in a master-slave way, where a slave vNF is only activated in case of failure of the master.

As vNFs are not completely decoupled from their underlying infrastructure, they can still rely on specific guest operating systems, network cards and their special capabilities alongside many acceleration techniques (e.g., Intel SR-IOV, DPDK) provided by the hardware, among other properties. Therefore, when designing lightweight vNFs for a heterogeneous infrastructure, operators need to make sure that the vNFs are portable, and therefore can be installed on multiple devices, ideally on all servers, edge devices and even in cloud VMs.

Elasticity should also be treated as a first class parameter for a lightweight NFV system. It essentially means to be able to adapt to workload changes by provisioning and de-provisioning resources in an automatic manner. As an example of elasticity, free or idle vNFs can be turned off in case, for example, the particular user is not connected to the network. At the same time, when the load is high, a vNF should be able to migrate from a low-cost edge node to a more powerful server. This envisioned elasticity can be achieved by using lightweight vNF encapsulations (e.g., using containers introduced in Section 3.4) that support many underlying devices, fast stop and start-up times, as well as by doing continuous monitoring of system resources and temporal user demand to dynamically adjust vNF placement using orchestration algorithms [130].

3.3.2 Performance Considerations

Performance requirements vary with each vNF type, vNF configuration and deployment scenario. As an example, in-path vNFs deployed to inspect live traffic have strict requirements to keep the introduced delay as low as possible and therefore require fast CPUs and memory, while, e.g., vNFs deployed offline saving packet traces to files require faster data storage [131]. Therefore, backing [123], the following should be considered when examining lightweight vNF performance:

- vNF lifecycle management: The time it takes to instantiate a vNF is one of the most important factors when talking about lightweight vNF platforms. This time consists of the time required for creating and starting a vNF (e.g., a VM or container) as well as the time it takes to start up the software inside the vNF. This time is sometimes called vNF provisioning time and it is usually measured in minutes in today's vNF platforms, while lightweight vNFs are reducing this time to seconds [24].

- Runtime performance: Runtime performance of a vNF depends on the software as well as the amount of resources allocated to the vNF. Example metrics representing vNF performance include achievable throughput, introduced delay, or application level metrics, such as the number of concurrent connections that can be handled by a software router [132].

- Performance of communications: vNFs often need to communicate with other vNFs or with a manager software. For example, chained or multiplexed anomaly detection vNFs are designed to distribute information among vNFs [133]. Also, stateless vNFs presented by Kablan et al. rely on high-speed communication between vNFs and a remote database as introduced in [134]. Moreover, many monitoring vNFs are designed to communicate with a manager software to periodically notify operators of packet statistics [24]. Therefore, it is important to provide an efficient, secure and high-speed inter-vNF and vNF-to-manager networking solution.

3.3.3 Security and Privacy Considerations

Since there is no physical separation between vNFs, and in general lightweight virtualisation methods provide weaker isolation [135], security and privacy considerations are crucial. Security of NFV platforms can be analysed on multiple fronts. To mention just a few, one could analyse the data security inside the network function, the way network functions are deployed and delivered to hosting devices, as well as how traffic is managed between vNFs.

Securing vNFs internally is making sure that vNFs do not leak or allow access to internal data (e.g., flow states). This is usually the job of the vNF developer, yet some support for this can be given from the NFV infrastructure. As an example presented as S-NFV [136], operators can enhance the isolation between vNFs using hardware-extensions such as Intel SGX. Configuration of Security Enhanced Linux parameters can also be used to enhance isolation between vNFs sharing the same platform [137].

Considering a large-scale, geographically distributed deployment of lightweight vNFs as it is envisioned in this thesis, vNF images and data need to be distributed over multiple, sometimes untrusted networks (e.g., public Internet). Therefore, trust is a central concern to secure the integrity and the producer of all the data a system operates on. As a possible mitigation, some image distribution solutions support cryptographically signing images[2].

Securing how traffic is managed means controlling traffic from the source to the destination using secure channels. In a highly dynamic scenario, where vNFs are designed to be migrated in short timescales, it is important to always direct the right traffic through the right vNFs, which can be guaranteed using a centralised control software that synergistically manages vNFs as well as the network [138, 139].

3.3.4 Management of vNFs at Scale

According to ETSI NFV [2], management of vNFs is primarily concerned about lifecycle management and determining the physical location for vNFs. On top of this, the manage-

[2]https://docs.docker.com/engine/security/trust/content_trust/ Accessed on: 7 March 2018

ment software needs to account for failures of the infrastructure, security and policy considerations, performance management and continuity, as discussed above. While this is a reasonable task for typical NFV deployments consisting of a couple of racks and around a few hundred vNFs[3], managing thousands of small, lightweight vNFs in a distributed, heterogeneous infrastructure that consists of hundreds of edge devices is challenging in the following ways [123]:

- Centralised control interfaces: each NFV framework has to provide a secure control interface where the state of the system can be managed by the operator. The location and status (lifecycle, associated users, service owner, software version, etc.) of all vNFs need to be managed using secure connections from many vNFs and hosting devices, requiring carrier-grade control software that can handle large number of simultaneous connections.

- Visualisation at scale: Visualising the network topology as well as the deployed services is important in order to monitor the health of the system. Therefore, a user interface has to be able to visualise and cluster hundreds of edge devices and vNFs. This can be achieved with latest UI technologies using powerful Javascript libraries or latest hardware-accelerated visualisation techniques (such as WebGL).

- Extensibility of the management software: A NFV framework would not exist without the accompanying software feeding information to it (e.g., a CRM software can feed user's vNF subscription information to an NFV platform) or getting information out of it (e.g., a billing system to extract service usage details from a NFV platform). To integrate such systems, a management software needs well documented and platform-independent APIs with platform and language-independent data schemes.

- Proactive, dynamic management: This is particularly important, as one of the promises of NFV is to be able to react to changes in network dynamics within seconds. To do this, the management software should perform dynamic re-orchestration of vNFs

[3]https://www.prnewswire.com/news-releases/verizon-launches-industry-leading-large-openstack-nfv-deployment-300256567.html Accessed on 8 February 2017

when network properties change (e.g., users move or links get overloaded), however, this can be a computationally complex task with large number of users, hosting devices and vNFs, as shown later in this thesis.

3.3.5 Integrating Public Cloud Environments

Public clouds have been proposed to be part of NFV infrastructures in multiple studies. As an example, Sherry et al. presented the outsourcing of vNFs to the cloud [69], while Lan et al. analysed the security issues of NFV in the cloud with Embark [140]. As their main advantage, cloud environments offer network operators the ability to rent computing resources (essentially VMs) on a pay as you go basis, therefore reducing the initial commitment for an in-house NFV infrastructure.

This thesis also investigates public clouds as parts of the NFV platform designed. This allows the provider to outsource vNFs in case there are no resources available at the edge of the network or at the internal NFV infrastructure of the provider [25]. This provides flexibility to handle increased demand for network services, such as at concerts and sports events [141].

While public clouds present a cost-efficient way of hosting vNFs, they introduce a few considerations that are described below.

Limited Network Programmability vNFs require traffic to be steered through them. In traditional networks, this is done either by manually configuring the forwarding tables of the switches and routers along the path, or automatically using an SDN controller. In public clouds, even when the network is configurable, the programmability and flexibility of the virtual network is extremely limited. Typically, the network is abstracted into a single virtual switch connecting all the VM instances together and to the Internet, and simple destination-based routing entries can be inserted. Without the ability to control the forwarding policies with protocols such as OpenFlow [17], traffic management in the cloud is hindered.

Variable Performance The shared nature of the networks in today's multi-tenant cloud data centres implies that network performance can vary and therefore cloud providers usually do not offer predictable network performance to tenants, as analysed by Ballani et al. [142] and Mogul et al. [143]. This variable performance of cloud offerings can be translated to unpredictable vNF performance in case the appropriate measures (e.g., selecting the right instance type) are not taken when vNFs are migrated to the cloud.

Protocol Support In order to simplify Network Address Translation (NAT) and shape the traffic to improve QoS, most public cloud providers only allow traditional transport protocols (e.g., TCP, UDP, ICMP) to be used between VMs. Layer 2.5 protocols such as MPLS and Layer 3 protocols such as IPsec, IP-in-IP or GRE are commonly not supported, preventing traditional traffic encryption and encapsulation mechanisms to be used by default [25].

Support for vNFs Today's popular vNFs platforms (such as the high-performance ClickOS or Brocade Vyatta [144]) are often built on top of specific software components or require network configuration that is rarely supported by most public clouds. ClickOS for instance uses a highly modified XEN hypervisor to improve network performance that prevents its deployment on public clouds. Vyatta requires cloud-specific network configuration and therefore cannot be ubiquitously deployed over public cloud infrastructures.

3.4 Container Network Functions

As outlined in Section 3.2.1, NFV frameworks have traditionally applied VMs to host vNFs. However, recently, new virtualisation technologies have been proposed for NFV, including containers, unikernels (specialised VMs) and minimised distribution of general-purpose VMs [21, 123]. These lightweight technologies can typically avoid the hardware requirements of virtualisation and overheads associated with hypervisors and VMs. In this section,

the major differences between these approaches are compared and the selection of containers for our NFV framework is justified.

3.4.1 Comparison with Virtual Machines and Unikernels

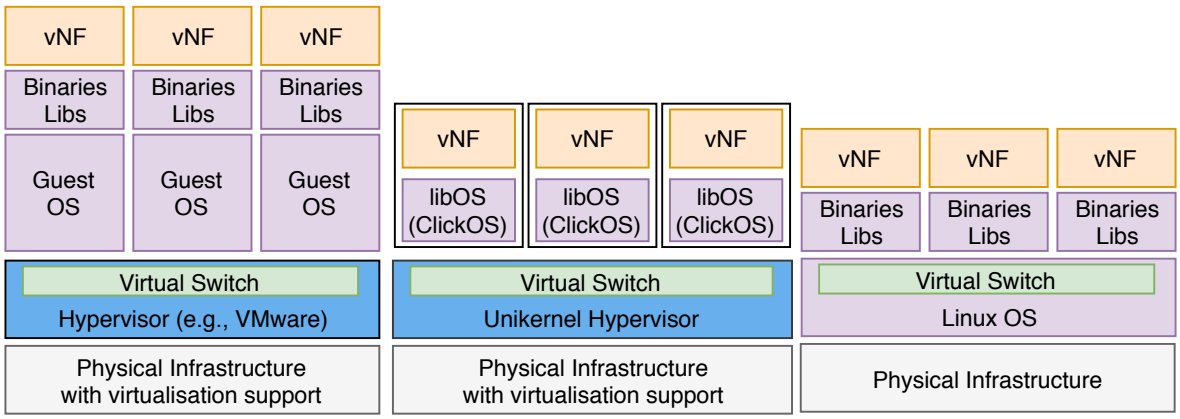


Figure 3.2: VMs vs unikernels (specialised VMs) vs containers.

Virtual Machines Virtual Machines substitute entire machines. They encapsulate an Operating System, along with a separate kernel, libraries, binaries and configuration for vNFs and run on top of software called a hypervisor that manages resource shared between multiple VMs on the same machine, as shown in Figure 3.2. VMs provide high levels of isolation and flexibility for running anything inside them. Modern hypervisors and VMs require hardware support for virtualisation, primarily from the host CPUs. As described in Section 2.2.5, most NFV frameworks today utilise traditional VMs and hypervisors.

Minimised distribution of general-purpose VMs can also be considered for NFV. These are VMs that have been optimised for quick start up time and small disk usage by removing unnecessary components that vNFs do not require (such as a graphical user interface, unused drivers and CPU support, multi-user support, etc.). While these are significantly smaller than traditional VMs (Tiny Core OS takes 11MB instead of 2GB required for an Ubuntu installation), they still require hardware virtualisation support and a hypervisor.

Unikernels Unikernels (or specialised VMs) are purpose-built systems maturing from research projects. Unikernels are usually compiled, single address space machine images that run directly on a special hypervisor without using a complete Operating Systems. Some projects have proposed using unikernels for NFV. A notable unikernel approach for NFV have been presented by ClickOS [21], where the authors show a secure and optimised way of packet processing using the Click [127] software environment and a version of the XEN hypervisor. While this approach provides high performance, depending on a purpose-built hypervisor limits deployability and, moreover, unikernels usually restrict vNFs to be implemented on a specific software environment (e.g., Click in terms of ClickOS).

Containers Figure 3.2 compares containers with VMs and unikernels in a graphical way. As their main advantage, traditional VMs (used by, e.g., the Cloud4NFV [22] or the OPNFV platforms) allow vNF users to specify their Operating System for each individual network function, while vNFs inside unikernels (e.g., the ClickOS [21] system) and containers need to utilise a specific platform. In case of unikernels, vNFs are compiled to small binaries that are executed on purpose-built hypervisors, while containers allow any software to be run on a general purpose Linux kernel using a shared kernel.

Therefore, containers are a good compromise between unikernels and commodity VMs as they allow generic software to be used in the vNFs similar to VMs, while providing lightweight encapsulation similar to Unikernels. At the same time, containers incur significantly lower overhead than traditional VMs and can be deployed on any Linux environment without virtualisation support (available from commodity routers to high-end servers) with similar performance to the host machine [135]. Similar to unikernels, containers also allow much higher network function-to-host density and smaller footprint at the cost of reduced isolation. Using containers, commodity devices and public cloud VMs are able to host up to hundreds of vNFs, as shown in [51, 123].

Containers, on top of being lightweight alternatives to VMs and unikernels, also advocate a micro-services architecture by design. This means that containers are created per service and therefore provide independently deployable, smaller, modular services that encapsu-

late a unique process and communicate through well-defined, lightweight mechanisms such as REST APIs with JSON data transfer (as opposed to collection of services residing in VMs) [145]. Microservices have been advocated for NFV in various works, such as in [146], where Jahromi et. al. presented a CDN architecture built on top of vNF micro-services.

3.4.2 Disadvantages of Containers

While containers provide many benefits, especially when used on low-cost edge devices (as described in the previous section), there are technology-related challenges to be considered when choosing containers for vNFs.

3.4.2.1 Security and Isolation

Containers typically offer weaker isolation between co-located instances than traditional VMs or unikernels. While this has many benefits on the performance (i.e., avoiding packet copying) and agility of the vNFs, it can potentially result in interference between vNFs if deployed without proper resource guarantees, as analysed by ETSI [147]. However, deploying with OS-level security measures (such as using SELinux with access control security policies support, and using AppArmor to set per-program restricted access profiles) containers can be mature enough for production environments. Additionally, hardware-level improvements can be made (e.g., Intel SGX), introduced below in Section 3.4.3. Recently, security improvements have also focused on minimal host OS distributions for reducing the attack surface while executing host management tools in isolated management containers [147].

3.4.2.2 Limited Operating System and CPU Architecture

Containers need to share the same operating system and kernel on a given hosting device. Different kernel versions provide different capabilities, while operating systems provide various tools to install software. These differences make it harder (and sometimes impossible) to create vNFs that run on various platforms. As a result, developers need to know the oper-

ating systems versions in advance to create containers that run on all them. The importance of this is amplified with vNF migrations, where vNFs need to be able to move from one device to an other to keep their locations optimal to temporal network characteristics.

An other limitation lies in the lack of virtualisation. Since all software inside the containers need to run directly on the hosting platform's CPU, vNFs, containers compiled on a particular architecture (e.g., ARM v6) can not be directly executed on an x86 CPU. To overcome this issue, a container with all of its dependencies have to be compiled for all possible CPU architectures, causing additional complications for vNF developers.

3.4.2.3 Limitations of the Micro-Services Architecture

Container micro-services are beneficial for some vNF chains that allow destructing to small, lightweight services naturally - e.g., packet firewall chains. However, some applications are monolithic, need to be hosted on a specific location and therefore would only benefit containerisation as a packaging solution.

Additional disadvantages of the micro-services architecture include the increased management complexity, additional marshalling and un-marshalling between vNFs, increased complexity of testing and monitoring of production systems.

3.4.3 Hardware Requirements for Containers

Containers have been inherently designed to be able utilise any type of hosting device, whether it be a fully fledged x86 server or a low-cost ARM single-board computer. This means that containers do not require CPU or I/O support such as Intel's Virtualisation Technology (VT-X) for Intel CPUs or AMD Virtualisation for AMD CPUs, in general. This is important for our case, as network edge devices can have as low specifications as 128 MB of memory and 2x1 GHz of CPU, as shown in Table 2.1. However, it is important to mention that containers are complied to a specific architecture and ABI, using a particular system

call interface which can restrict the migration of built containers between various underlying hardware.

With the recent increase in popularity of containers, some hardware vendors (e.g., Intel, ARM) have started to provide architecture extensions to improve performance and security of such technologies. A good example for such hardware-support is the Intel Software Guard Extensions (SGX) that allows vNF developers to protect selected code and data from disclosure and modification on a shared platform, as evaluated in [148]. Intel SGX promises this isolation with the use of enclaves, which are essentially protected areas of execution in memory that application developers can use to put code into.

An other hardware extension to containers has been proposed by Intel's Clear Containers project in 2015, where the goal was to address the security concerns of containers by using Intel's VT-X. By utilising hardware-enforced isolation with VT extensions, containers can provide the same security as a virtual machine. The findings from the Clear Containers project have been applied at a new container system called Kata[4] which is currently under the governance at the OpenStack Foundation and is likely to be supported in OpenStack soon.

3.4.4 Operating System Requirements for Containers

Containers are essentially isolated processes with separate process and network namespaces (i.e., routing tables). This isolation between virtual networks and process spaces is achieved by different components of the Linux kernel. The most important kernel components required by containers are the following:

1. cgroups (abbreviated from control groups): limits, accounts for and isolates resource usage (CPU, memory, disk I/O, network, etc.) for a collection of processes.

2. namespaces: isolate and virtualise system resources (process IDs, hostnames, user IDs, network access, filesystem, etc.) of a collection of processes.

[4]http://katacontainers.com Accessed on: 5 April 2018

3. capabilities: provide permission checks for processes.

4. netfilter: packet filtering framework that essentially provides the *iptables* firewall, allowing the management of multiple container's traffic on the same host.

5. Apparmor: a security module that allows system administrators to restrict processes to network or file access.

6. Netlink: provides network communication between containers.

7. SElinux (Security-Enhanced Linux): a module that provides access control over files enforced from the kernel.

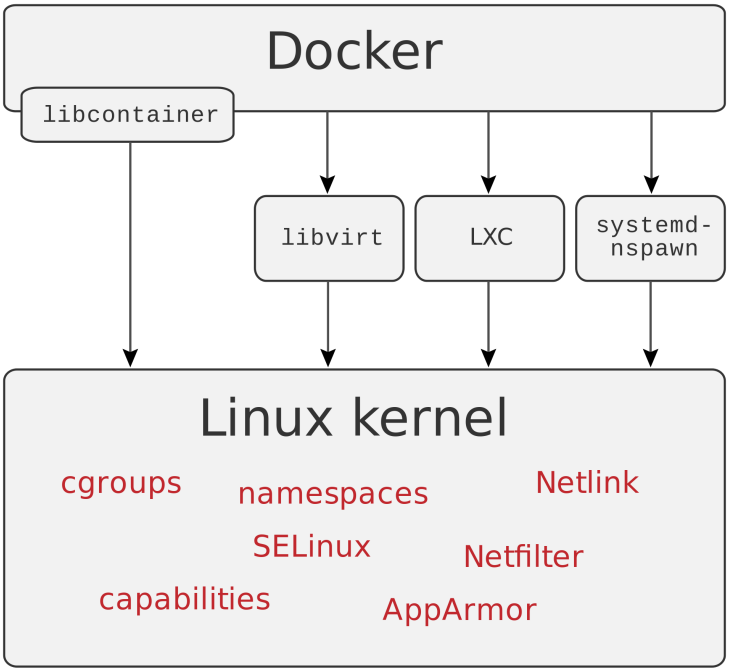


Figure 3.3: Kernel modules required for container technologies (e.g., Docker, LXC). Source: https://delftswa.github.io/chapters/docker/

While containers do not require hardware support as other virtualisation technologies (as detailed in Section 3.4.3), they need the aforementioned components to be built into the host Operating System's kernel. As shown in Figure 3.3, different containerisation technologies use these modules in different ways. For instance, the Docker container system creates containers through a *libcontainer* interface that has to be compiled and available for Docker to

work. However, the LXC system uses lower-level packages from the Linux kernel directly, such as cgroups and network namespaces (among others). Such small differences can restrict a few containerisation platforms to some Linux distributions. For example, on some versions of OpenWRT (the most popular Linux distribution for home routers), Docker's libcontainer module has not been ported yet, meaning that Docker containers can't be started as of today. In this case, one could still use one of the other container alternatives described in the following section.

3.4.5 Docker and Other Container Alternatives

While in general containers are built on top of the same Linux kernel-supported namespacing solutions as discussed in the previous section, there are many systems that provide a comprehensive toolset on top of just creating isolated processes.

Probably the most popular one today is Docker [5], that transformed the industry after its initial release 4 years ago. Docker constructs containers using filesystem layers that allow sharing the same files between different containers. This minimises disk usage and makes images download faster from image repositories. Images can also be signed with cryptographical algorithms for security purposes. Moreover, Docker also has a descriptor file (called Dockerfile) that can be used to build and extend Docker containers in a well defined way.

Another well-known example for a container system is the Linux Containers [149] (LXC) project that has been in the Linux kernel for much longer, but it lacks many tools provided by Docker. As an example, LXC does not have a built-in, standardised container build system as Docker has and also, LXC doesn't allow sharing layers of the filesystem between containers, therefore consuming more space on the devices. Also, LXC does not provide image repositories, making fast container deployments and image distribution harder.

Some projects have focused on even lighter container alternatives. As an example, the Furnace[6] project presents a lightweight, pure-Python container implementation which is essen-

[5] https://www.docker.com Accessed on: 1 February 2018
[6] https://github.com/balabit/furnace Accessed on: 16 January 2018

tially only a wrapper around the Linux namespace functionality through libc functions such as *unshare()*, *nsenter()* and *mount()*. Furnace can be used in devices where fully fledged container technologies such as Docker are not available, but some level of process isolation and management is required.

To sum up different alternatives, Docker has proved itself as a good fit for the NFV system designed: it is available on most Linux systems, provides a well defined build process for containers, uses a layered filesystem to minimise disk usage and offers many software tools for secure vNF image management and network configuration between containers.

3.5 Latency-Optimal vNF Orchestration

This section introduces the latency-optimal vNF orchestration designed for our lightweight NFV framework. This orchestration is divided into two pieces. First, Section 3.5.1 presents a latency-optimal vNF placement problem formulated and solved as an optimisation problem. Then, Section 3.5.2 presents a dynamic extension to this initial placement problem in order to follow changes in network properties (e.g., link latency fluctuations or user movements).

3.5.1 Optimal Placement of vNFs at the Network Edge

Following up the motivation described in Section 3.2.2, this section introduces the *Edge vNF placement* problem as an Integer Linear Program (ILP) to calculate the latency-optimal allocation of vNFs in next-generation edge networks.

3.5.1.1 Rationale

With recent advances in virtualisation and NFV, vNFs can be hosted on any physical server, for instance an edge device close to the user, the internal infrastructure of the provider or in a distant cloud [2] [3]. In order to provide the best possible vNF-to-user end-to-end (E2E)

latency, operators are aiming to place vNFs in close proximity to their users, by first utilising edge devices that are close to the user and falling back to hosting vNFs in the devices further from the user when edge devices are out of capacity, as outlined in Section 2.3.4. The model also accommodates different bandwidth requirements as control plane vNFs typically consume less bandwidth than data plane vNFs.

Network parameters	**Description**
$\mathbb{G} = (\mathbb{H}, \mathbb{E}, \mathbb{U})$	Graph of the physical network.
$\mathbb{H} = \{h_1, h_2, h_j, \ldots, h_H\}$	Compute hosts (e.g., edge devices, cloud VMs) within the network.
$\mathbb{E} = \{e_1, e_2, e_m, \ldots, e_E\}$	All physical links in the network.
$\mathbb{U} = \{u_1, u_2, u_o, \ldots, u_U\}$	All users associated with network functions.
$\mathbb{P} = \{p_1, p_2, p_k, \ldots, p_P\}$	All paths in the network.
W_j	Hardware capacity $\{cpu, memory, io\}$ of the hosts $h_j \in H$.
C_m	Capacity of the link $e_m \in E$.
A_m	Latency on the link $e_m \in E$.
Z_k	Last host in path $p_k \in P$.
vNF parameters	**Description**
$\mathbb{N} = \{n_1^1, n_2^2, n_i^o, \ldots, n_N^U\}$	Network functions to allocate, where the vNF $n_i^o \in \mathbb{N}$ is associated to user $u_o \in \mathbb{U}$.
R_i	vNF's host requirements $\{cpu, memory, io\}$ of vNF $n_i \in \mathbb{N}$.
θ_i	The maximum latency vNF $n_i \in N$ tolerates from its user.
Derived parameters	**Description**
b_{ijk}	Bandwidth required between the user and the vNF n_i in case it is hosted at h_j using the path p_k. Derived from the physical topology and the vNF requests.
l_{ijk}	Latency between the user of the vNF n_i in case it is hosted at h_j and uses the path p_k. Derived from the physical topology and the vNF requests.
Variables	**Description**
X_{ijk}	Binary decision variable denoting if n_i is hosted at h_j using the path p_k or not.

Table 3.1: Table of parameters for our system model.

3.5.1.2 System Model

Table 3.1 shows all the parameters used for the formulation. The physical network is represented as an undirected graph $\mathbb{G} = (\mathbb{H}, \mathbb{E}, \mathbb{U})$, where $\mathbb{H}$, $\mathbb{E}$ and $\mathbb{U}$ denote the set of hosts, the links between them, and the users in the topology, respectively. vNFs can be placed to any host in this graph, meaning that all physical hosts have $cpu, memory, io$ capabilities to host vNFs. Also, all links have a physical limit on their bandwidth (e.g., 1 Gbit/s or 100 Mbit/s) that is taken into account for the placement.

As there are different types of vNFs (e.g., lightweight firewalls or resource-intensive Deep Packet Inspection modules), $cpu, memory, io$ requirements for each vNF n_i have been introduced. On top of the compute requirements, all vNFs have delay requirements according to the service provider's QoS offerings that are denoted with θ_i. Similarly, all vNFs specify bandwidth constraints required along the path from their user (since some vNFs are used heavily by the users, some are not). These bandwidth requirements are materialised using the derived parameter b_{ijk} that denotes the bandwidth required along the path p_k in case vNF i associated to user u_o (n_i^o) is hosted at h_j. Another important derived parameter of the model, l_{ijk}, denotes the E2E latency from a user to a vNF in case n_i^o vNF is hosted at a certain host (h_j) while its traffic is routed using a certain path p_k. l_{ijk} is calculated by summing all A_m individual latency values of each link along the path p_k from user u_o (user of vNF n_i^o) to the vNF hosted at host h_j.

Finally, let X_{ijk} be our main binary decision variable for the model, that is:

$$X_{ijk} = \begin{cases} 1 & \text{if one allocates } n_i^o \text{ to } h_j \text{ using path } p_k \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

3.5.1.3 Problem Formulation for Optimal Placement

The *Edge vNF placement problem* is defined as follows:

Problem 1. *Given the set of users $\mathbb{U}$, the set of vNF hosts $\mathbb{H}$, the set of individual vNFs $\mathbb{N}$,*

and a latency matrix l, one has to find an appropriate allocation of all vNFs that minimises the total expected end-to-end latency from all users to their respective vNFs:

$$\min . \sum_{p_k \in \mathbb{P}} \sum_{n_i^o \in \mathbb{N}} \sum_{h_j \in \mathbb{H}} X_{ijk} l_{ijk} \tag{3.2}$$

Eq. 3.2 looks for the values of X_{ijk}, while it is subject to the following constraints:

$$\sum\nolimits_{n_i^o \in \mathbb{N}} \sum\nolimits_{p_k \in \mathbb{P}} X_{ijk} R_i < W_j, \forall h_j \in \mathbb{H} \tag{3.3}$$

Constraint (3.3) ensures that the hardware limitations of hosts are adhered to (since CPU, memory, and IO resources are finite, one can only host a certain number of vNFs at particular hosts). This constraint sums up all the resource usage requirements of the vNFs R_i (selected by the X_{ijk} decision variable) and ensures that the sum for a host h_j is below W_j, the total capacity of the host.

$$\sum\nolimits_{h_j \in \mathbb{H}} \sum\nolimits_{p_k \in \mathbb{P}} X_{ijk} l_{ijk} < \theta_i, \forall n_i^o \in \mathbb{N} \tag{3.4}$$

Constraint (3.4) ensures that the maximum vNF-to-user E2E latency tolerance (θ_i) is never exceeded at placement. This means that a vNFs will always be located to locations where their latency from their user is less than this threshold. The calculation of all possible latency values between all possible vNF allocation is stored in the l_{ijk} derived parameter that holds the latency between the user of the vNF n_i in case it is hosted at h_j and uses the path p_k.

$$\sum\nolimits_{h_j \in \mathbb{H}} X_{ijk} = 1, \forall n_i^o \in \mathbb{N}, \forall p_k \in \mathbb{P} \tag{3.5}$$

Constraint (3.5) states that each vNF must be allocated to exactly one of the hosts (for instance, to a nearby edge device or a cloud VM). This is ensured by checking if a particular host's allocation counter sums up to 1.

$$\sum_{h_j \in \mathbb{H}} X_{ijk} b_{ijk} < C_m, \forall e_m \in p_k, \forall p_k \in \mathbb{P} \tag{3.6}$$

Constraint (3.6) takes bandwidth requirements of vNFs (b_{ijk}) and link capacities ($linkcap(m)$) along a path, and ensures that none of the physical links along the path get overloaded.

$$X_{ijk} = 0, h_j \neq Z_k, \forall p_k \in \mathbb{P}, \forall h_j \in \mathbb{H} \tag{3.7}$$

Finally, constraint (3.7) ensures that only valid user-to-vNF paths can be selected, that connect the user with the potential host of the vNF. This means that the path to a vNF should always end up on the host where the vNF is, basically linking path selection to vNF placement.

3.5.2 Dynamic Placement Scheduling

This section presents a way to dynamically schedule the re-computation of the optimal placement formulated in Section 3.5.1 to fit a real-world scenario, where the E2E user-to-vNF latency changes over time due to events such as congestion on any of the links in the path or user mobility resulting in an increased latency between the user and the vNF [150].

3.5.2.1 Rationale

After a vNF placement is retrieved by implementing and running the ILP model presented in Section 3.5.1, the binary decision variables $X_{ijk} = x_{ijk}$ are instantiated to the values 0 or 1, such that the objective latency function $\mathcal{F}(\{x_{ijk}, l_{ijk}\})$ is minimised w.r.t. latency values l_{ijk}. Let us denote this instantiation $\mathcal{I}_0$. Now, a dynamic environment has to be considered where the latency matrix l_{ijk} between vNF i located at host j using path k varies with time due to, e.g., temporal load variation at the vNF hosting platform, changes in the position (i.e., physical distance) of the user from the vNF, changes in temporal network utilisation, etc. This implies that l^t_{ijk} varies with time t, and hence the overall latency $\mathcal{F}_t =$

$\sum_{i,j,k} x_{ijk} l^t_{ijk}$ varies with time given an initial vNF placement $\mathcal{I}_0$. This time dependency evidently results to changes in the objective function $\mathcal{F}$ which indicates that, at some time instance t, the placement $\mathcal{I}_0$ may not minimise $\mathcal{F}_t$. In this case, one should re-evaluate the minimisation problem (Section 3.5.1) with up-to-date latency values, hence obtaining a *new* vNF placement $\mathcal{I}_t$ at every time instance.

Based on a new vNF placement, vNFs might be needed to *migrate* from one host to another in order for the new placement $\mathcal{I}_t$ to minimise the objective function (overall minimum latency) $\mathcal{F}_t$. This migration cost due to different placement configurations is non-negligible. Specifically, given placements $\mathcal{I}_t$ and $\mathcal{I}_\tau$ at time instances t and τ with $\tau > t$, respectively, that both minimise the expected latency $\mathcal{F}_t$ and $\mathcal{F}_\tau$, respectively, the migration cost $\mathcal{M}_{t\to\tau}$ is defined as:

$$\mathcal{M}_{t\to\tau} = \sum_{ijk} I(x^t_{ijk}, x^\tau_{ijk}), \tag{3.8}$$

with $I(x, x') = 0$ if $x = x'$, and 1 otherwise. We encounter the problem to determine *when* a new optimal vNF placement needs to be evaluated to avoid possibly redundant re-computation and additional cost for vNF migration from one host to another. The trade-off here is that, at time instance $\tau > t$, the system can *tolerate* some *deviation* from the optimal (minimum) latency computed at time instance t to avoid a new optimal placement at τ along with a possible migration cost $\mathcal{M}_{t\to\tau}$ with expected migration cost $\mathbb{E}[\mathcal{M}_{t\to\tau}] = nhP(x^t_{ijk} = x^\tau_{ijk})$, where n is the number of users and h is the number of hosts. To materialise this tolerance, it is assumed that a vNF i can tolerate some latency increase, up to its θ_i. As a real-world example, a strictly real-time packet processing vNF can have a maximum of 10 ms latency tolerance [151]. That is, vNF *latency tolerance violation* is defined at time t as the indicator:

$$L^t_{ijk} = \begin{cases} 1 & \text{if} \quad l^t_{ijk} > \theta_i \\ 0 & \text{otherwise} \end{cases} \tag{3.9}$$

Since each vNF is only using one path and is allocated to one server, a single vNF's tolerance

can be defined as:

$$L_t^i = \sum_j \sum_k L_{ijk}^t, \tag{3.10}$$

while, at time t, the violation tolerance for all vNFs is then:

$$L_t = \sum_i L_t^i \tag{3.11}$$

The system is monitoring the evolution of the overall violation tolerance L_t with time in order to determine when to re-evaluate the optimisation placement problem in order to minimise the cost of migration. In the remainder, principles of Optimal Stopping Theory are applied to formalise this dynamic decision making problem.

3.5.2.2 Problem Formulation for Dynamic Placement Scheduling

Let us accumulate the user's violations in terms of latency tolerance from time instance $t = 0$, where the optimal placement of all the vNFs is obtained by minimising $\mathcal{F}$, up to the current time instance t, i.e., the following cumulative sum of overall violations up to t is obtained:

$$Y_t = \sum_{k=0}^{t} L_k. \tag{3.12}$$

Given the initial optimal placement $\mathcal{I}_0$, latency tolerance θ_i per vNF i, and expected migration cost $\mathbb{E}[\mathcal{M}_0]$ independent of time, we require that the maximum latency tolerance of the system users not to exceed a tolerance threshold $\Theta > 0$. That is, at any time instance t, if $Y_t \leq \Theta$ a re-evaluation of the vNFs placement is not enforced, hence avoid incurring any possible vNF migration cost $\mathcal{M}_{0\to t}$. If $Y_t > \Theta$, then the system should re-compute a new optimal placement and incur an expected migration cost $\mathbb{E}[\mathcal{M}_{0\to t}]$. The parameter Θ can be carefully set and adjusted by operators to reflect a specific QoS level and control the rate of latency violations allowed. The model can also be extended to implement various QoS levels by setting up different values of Θ for diverse sets of users (e.g., subscribers paying for a premium vNF service have a lower Θ than other flat-charged customers).

The challenge is to find the (optimal stopping) time instance t^* for deriving an optimal placement for the vNFs, such that Y_{t^*} be as close to the system's maximum tolerance Θ as possible. This Θ can then reflect the Quality of Service offerings of the network provider. If Y_t exceeds this threshold, then the system incurs a given expected vNF migration cost $\mathbb{E}[\mathcal{M}_0]$. Specifically, the goal is to maximise the cumulative sum of tolerances up to time t without exceeding Θ. To represent this condition of minimising the distance of the cumulative sum of violations from the maximum tolerance Θ, the following reward function is defined:

$$f(Y_t) = \begin{cases} Y_t & \text{if} \quad Y_t \leq \Theta, \\ \lambda \mathbb{E}[\mathcal{M}_0] & \text{if} \quad Y_t > \Theta, \end{cases} \tag{3.13}$$

where factor $\lambda \in [0, 1]$ weighs the importance of the migration cost to the reward function. Formally, the time-optimised problem is defined as finding the *optimal stopping time* t^* that maximises the expected reward in (3.13):

Problem 2. *Find the optimal stopping time t^* where the supremum in (3.14) is attained:*

$$\sup_{t \geq 0} \mathbb{E}[f(Y_t)]. \tag{3.14}$$

The solution of Problem 2 is classified as an optimal stopping problem where we seek a criterion (a.k.a. optimal stopping rule) when to stop monitoring the evolution of the cumulative sum of overall violations in (3.12), and start off a new re-evaluation of the optimisation for deriving optimal vNF placements. Obviously, one can re-evaluate the vNF placement optimisation algorithm at every time instance, but this comes at the expense of frequent vNF migrations. On the other hand, the system could delay the re-evaluation of the optimal placement at the expense of a higher number of latency violations. The idea is to tolerate as much latency violations as possible w.r.t. Θ but not to exceed this. The optimal stopping rule we are trying to find will provide an optimal dynamic decision-making rule for maximising the expected reward (since Y_t is a random variable) from any other decision-making rule. Before

proceeding with proving the uniqueness and the optimality of the proposed optimal stopping rule, some essential preliminaries of the theory of optimal stopping are outlined below.

3.5.2.3 Solution Fundamentals

The theory of optimal stopping [152], [7] is concerned with the problem of choosing a time instance to take a certain action in order to maximise an expected payoff. A stopping rule problem is associated with:

- a sequence of random variables $Y_1, Y_2, \ldots$, whose joint distribution is assumed to be known;
- and a sequence of reward functions $(f_t(y_1, \ldots, y_t))_{1 \leq t}$ which depend only on the observed values $y_1, \ldots, y_t$ of the corresponding random variables.

An optimal stopping rule problem is described as follows: The system is looking at the sequence of the random variable $(Y_t)_{1 \leq t}$ and, at each time instance t, the algorithm chooses either to *stop* observing and apply a decision (which, in this case, is the re-evaluation of our optimal vNF placement and trigger for the appropriate vNF migrations) or *continue* (with the existing vNF placement without re-evaluating its optimality). If the algorithm decides to stop observing at time instance t, the reward $f_t \equiv f(y_t)$ in induced. The goal is to choose a stopping rule or stopping time to maximise the expected reward.

Definition 1. *An optimal stopping rule problem is to find the optimal stopping time t^* that maximises the expected reward: $\mathbb{E}[f_{t^*}] = \sup_{0 \leq t \leq \mathcal{T}} \mathbb{E}[f_t]$. Note, $\mathcal{T}$ might be ∞.*

The information available up to time t, is a sequence $\mathbb{F}_t$ of values of the random variables $Y_1, \ldots, Y_t$ (a.k.a. filtration).

Definition 2. *The 1-stage look-ahead (1-sla) stopping rule refers to the stopping criterion*

$$t^* = \inf\{t \geq 0 : f_t \geq \mathbb{E}[f_{t+1} | \mathbb{F}_t]\} \tag{3.15}$$

In other words, t^* calls for stopping at the first time instance t for which the reward f_t for stopping at t is (at most) as high as the expected reward of continuing to the next time instance $t+1$ and then stopping.

Definition 3. *Let A_t denote the event $\{f_t \geq \mathbb{E}[f_{t+1}|\mathbb{F}_t]\}$). The stopping rule problem is monotone if $A_0 \subset A_1 \subset A_2 \subset \ldots$ almost surely (a.s.)*

A monotone stopping rule problem can then be expressed as follows: A_t is the set on which the 1-sla rule calls for stopping at time instance t. The condition $A_t \subset A_{t+1}$ means that, if the 1-sla rule calls for stopping at time t, then it will also call for stopping at time $t+1$ irrespective of the value of Y_{t+1}. Similarly, $A_t \subset A_{t+1} \subset A_{t+2} \subset \ldots$ means that, if the 1-sla rule calls for stopping at time t, then it will call for stopping at all future times irrespective of the values of future observations.

Theorem 1. *The 1-sla rule is optimal for monotone stopping rule problems.*

We refer the interested reader to [152] for proof. □

In the remainder, a 1-sla stopping rule is proposed which, based on Theorem 1, is optimal for our Problem 2.

3.5.2.4 Optimally-Scheduled vNF Placement

At the optimal stopping time t^*, the optimal placement of the vNFs is re-evaluated, i.e., deriving the *new* optimal placement I_{t^*} and incur a migration expected cost $\mathbb{E}[\mathcal{M}_0]$ given that $\lambda > 0$.

Theorem 2. *Given an initial optimal vNF placement $\mathcal{I}_0$ at time $t = 0$, the optimal placement $\mathcal{I}_t$ is re-evaluated at time instance t such that:*

$$\inf_{\tau \leq 0}\{\tau : \sum_{\ell=0}^{\Theta - Y_\tau} \ell P(L=\ell) \leq (Y_\tau - \lambda \mathbb{E}[\mathcal{M}_0])(1 - F_L(\Theta - Y_\tau))\} \tag{3.16}$$

where $F_L(\ell) = \sum_{l=0}^{\ell} P(L=l)$ and $P(L=\ell)$ is the cumulative distribution and mass function of L in (Eq. 3.11), respectively.

Proof. The target is to find an optimal 1-sla stopping rule, that is to find the criterion in (3.15) to stop observing the random variable Y_t and then re-evaluate the vNF placement. Based on the filtration $\mathbb{F}_t$ up to time instance t, we focus on the conditional expectation $\mathbb{E}[f(Y_{t+1})|Y_t \leq \Theta]$, that is:

$$\begin{aligned}
&\mathbb{E}[f(Y_{t+1})|Y_t \leq \Theta] = \\
&\mathbb{E}[Y_{t+1}|Y_t \leq \Theta, Y_{t+1} \leq \Theta]P(Y_{t+1} \leq \Theta) + \\
&\mathbb{E}[\lambda\mathbb{E}[\mathcal{M}_0]|Y_t \leq \Theta, Y_{t+1} > \Theta]P(Y_{t+1} > \Theta) = \\
&\mathbb{E}[Y_t + L|L \leq \Theta - Y_t]P(L \leq \Theta - Y_t) + \\
&\mathbb{E}[\lambda\mathbb{E}[\mathcal{M}_0]|L > \Theta - Y_t]P(L > \Theta - Y_t) = \\
&\sum_{\ell=0}^{\Theta-Y_t}(Y_t + \ell)P(L = \ell) + \lambda\mathbb{E}[\mathcal{M}_0](1 - \sum_{\ell=0}^{\Theta-Y_t}P(L = \ell)) = \\
&\sum_{\ell=0}^{\Theta-Y_t}\ell P(L = \ell) + (Y_t - \lambda\mathbb{E}[\mathcal{M}_0])F_L(\Theta - Y_t) + \lambda\mathbb{E}[\mathcal{M}_0].
\end{aligned}$$

For deriving the 1-sla, one has to stop at the first time instance t where $\mathbb{E}[f(Y_{t+1})|Y_t \leq \Theta] \leq Y_t$, that is, at that t:

$$\sum_{\ell=0}^{\Theta-Y_t}\ell P(L = \ell) + (Y_t - \lambda\mathbb{E}[\mathcal{M}_0])F_L(\Theta - Y_t) + \lambda\mathbb{E}[\mathcal{M}_0] \leq Y_t,$$

which completes the proof. □

Theorem 2 states that starting at $t = 0$, the algorithm stops at the first time instance $t \geq 0$ where the condition in (3.16) is satisfied. Based on this criterion, the expected reward in (3.13) is minimised. The proposed 1-sla stopping rule in Theorem 2 is optimal based on Theorem 1, given that Problem 2 is 'monotone'. This means that we stop at the first time t such that $f_t(Y_t) \geq E[f_{t+1}(Y_{t+1})|\mathbb{F}_t]$, with the event $\{Y_t \leq \Theta\} \in \mathbb{F}_t$. That is, any additional observation at time $t + 1$ would not contribute to the reward maximisation. We then proceed with the following:

Theorem 3. *The 1-sla rule in (3.16) is optimal for our vNF placement Problem 2.*

Proof. Based on Theorem 1, our 1-sla rule is optimal when the difference $E[f_{t+1}(Y_{t+1})|\mathbb{F}_t] - f_t(Y_t)$ is monotonically non-increasing with Y_t. Given the filtration $\mathbb{F}_t$, this difference is non-increasing when $Y_t \in [0, \Theta]$, therefore the 1-sla rule is unique and optimal for Problem 2. □

After any re-evaluation of the optimal vNF placement with optimal solution $\mathcal{I}_t$, the cumulative sum of the latency violations Y_k for $k > t$ is observed. Then, the 1-sla optimal criterion in (3.16) is re-evaluated at every time instance k for possible triggering of the rule. The computational complexity of this evaluation depends on the number of vNFs in the entire system. The stopping criterion evaluation should be of trivial complexity, avoiding time-consuming decision-making on whether to activate the re-evaluation or not. From (3.16), the stopping criterion depends on the calculation of a summation from 0 to $\Theta - Y_k$, at time instance k. Evidently, one can recursively evaluate this sum at time k by simply using the sum up to time $k-1$ plus a loop of length $\mathbb{E}[|Y_k - Y_{k+1} + 1|]$. Hence, the time complexity of evaluating the sum at k is $O(N)$, where N is the number of vNFs in the entire system.

3.6 Summary

The design concepts and core requirements of a lightweight NFV framework capable of running and dynamically managing virtual network functions on various underlying infrastructures (e.g., low-cost edge devices or public cloud VMs) were presented in Section 3.2, while Section 3.3 discussed the most important considerations of such framework. Section 3.4 outlined why containers were selected to encapsulate vNFs as opposed to traditional VM-based technologies or specialised VMs, providing much lower hardware requirements, and allowing fast and efficient lifecycle management for the services.

To allow low-latency service deployment and dynamic management, Section 3.5 presented a dynamic orchestration solution for vNFs that can be placed anywhere in the infrastructure.

This latency-optimal orchestration consists of a vNF placement optimisation problem (shown in Section 3.5.1) as well as a dynamic extension (shown in Section 3.5.2) that allows the system to re-compute the placement in order to follow changes in network properties (e.g., link latency fluctuations or user movements).

Overall, the design principles presented in this chapter will enable network providers to deploy containerised virtual network functions in edge and cloud infrastructures, and manage the placement of services in a dynamic way to always provide the best latency to users, while minimising the number of migrations required. Following this chapter, the implementation details of the Glasgow Network Functions framework are presented.

Chapter 4

The Glasgow Network Functions (GNF) Framework

4.1 Overview

Following the design considerations and characteristics outlined in the previous chapter, we have implemented the Glasgow Network Functions (GNF, formerly GLANF) framework[1] [6, 24, 25, 27, 124]. GNF is a lightweight NFV platform that has been designed to provide low-latency services to users by managing container-based vNFs on diverse underlying devices, such as network edge nodes, cloud VMs or fully fledged NFV servers residing in providers' core networks. GNF has the following main characteristics:

- **Container-based:** vNFs are encapsulated in lightweight Linux containers to provide fast lifecycle management, platform-independence, high throughput, low delay, and low resource utilisation.
- **Minimal footprint:** GNF vNFs run at very low-cost (e.g., taking only a few MBs of memory), allowing its deployment on commodity and low-end devices that do not support hardware-accelerated virtualisation.

[1] https://netlab.dcs.gla.ac.uk/projects/glasgow-network-functions (Accessed on: 05/03/2017)

- **Support for vNF roaming:** With their small footprint and encapsulated functions, GNF vNFs seamlessly follow users between edge devices, providing a consistent and location-transparent service.
- **Latency-optimal vNF allocation:** GNF allocates lightweight vNFs to hosting devices in close proximity of the end users, therefore providing the lowest possible latency between vNFs and the users.
- **End-to-end, SDN-based transparent traffic steering:** Providers can attach and remove vNFs from users transparently, without adversely impacting the flow of traffic.

Figure 4.1 provides a high level overview of GNF. The overall architecture is organised in four planes, following the recommendations of the ETSI NFV standard [147]: the infrastructure plane (consisting of the edge devices and the central NFV infrastructure where vNFs can be hosted), the virtual infrastructure management plane (VIM), the orchestration plane, and a high-level service plane.

In the following sections, the implementation details of GNF are provided. First, we show the requirements from the network and server infrastructure in Section 4.2. Then, we detail the software running at the service, orchestration and VIM planes in Section 4.3, 4.4 and 4.5, respectively. Finally, Section 4.6 shows lightweight implementations of today's most popular type of vNFs in GNF.

4.2 Infrastructure Setup

As shown in Figure 4.1, the infrastructure plane of GNF consists of various hosting platforms such as home routers, enterprise and IoT gateway servers, traditional NFV servers and even public cloud VMs. The requirements for these are detailed in Section 4.2.1. The requirements for the underlying network interconnecting these devices are detailed in Section 4.2.2.

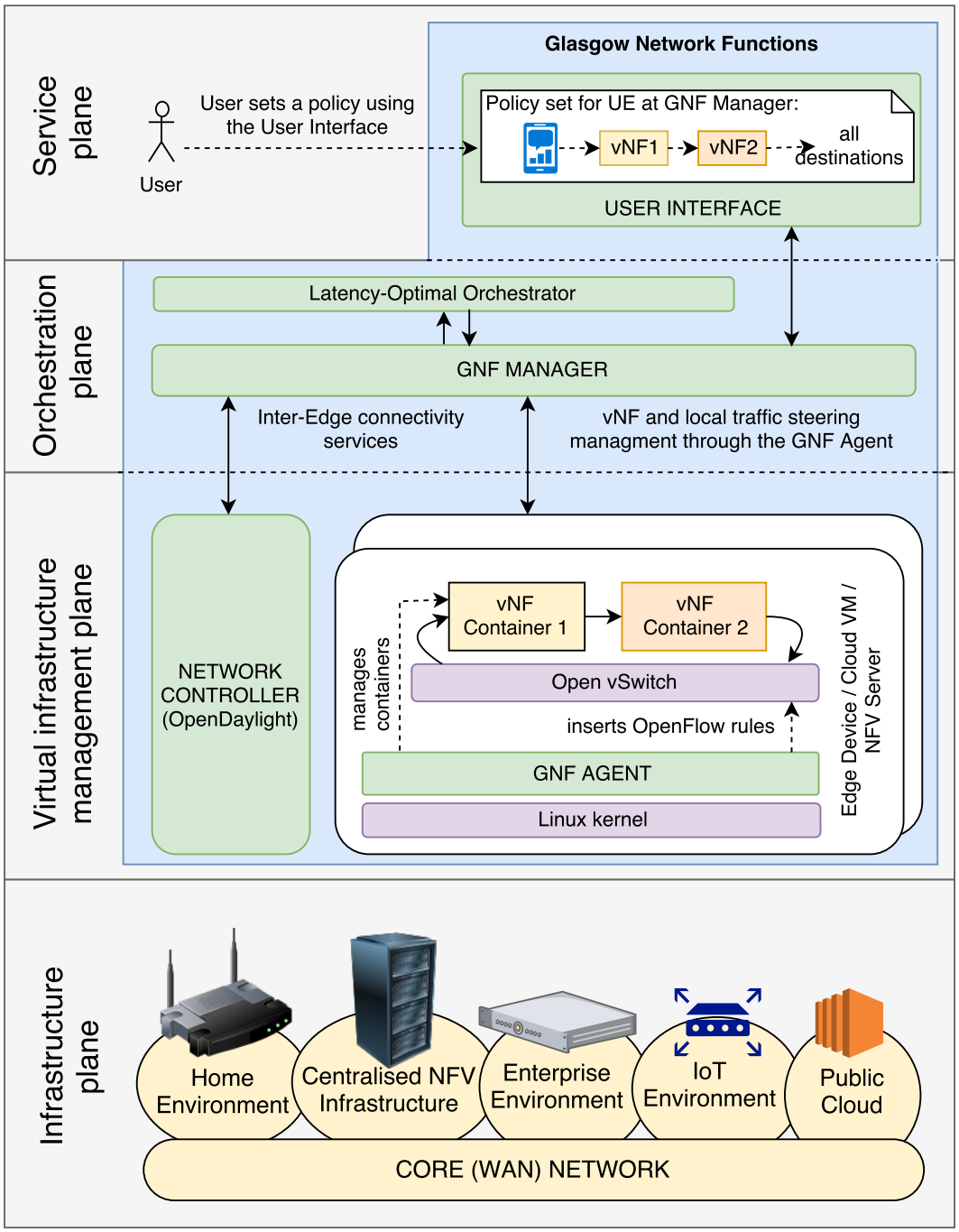


Figure 4.1: The GNF architecture - an overview [6].

4.2.1 vNF Host Devices

GNF is designed to utilise diverse underlying devices to host vNFs. Below we introduce some of these devices (shown in Figure 4.1) and provide the Operating System versions that have been selected for them.

NFV Servers, Enterprise and IoT Gateways NFV infrastructures usually consist of multiple racks of fully-fledged x86 servers with plenty of RAM and multiple CPU cores. Popular servers used today for NFV are for instance Dell's R series machines (such as, Dell R805) that are explicitly designed for virtualisation with carefully selected processor technology, memory capacity and I/O performance. Enterprise gateways can be envisioned using similar, high-end machines, but only a few of them at each site. Recently proposed IoT gateways from Dell, NEXCOM and HP Enterprise have wide range of hardware specifications from 1.33 GHZ Intel Atom CPU and 2 GB memory to Intel Xeon 4x3.0GHz CPU with up to 64 GB or memory, as shown in Section 2.3.4.

The requirement for these devices is an installation of a Linux operating system, preferably Ubuntu 16.04 LTS with a Linux kernel version higher than 3.10 to support Docker containers and a version of Python 2.7.

Home Routers Home routers can be made part of the GNF platform by running a Linux-like operating system with a recent kernel that is capable of hosting containers. While most providers now deliver relatively powerful equipment to their customers (e.g., Orange, France's most popular network provider, ships home routers with 1GB of memory and a 1GHz Cortex A9 CPU as shown in Section 2.3.4), these devices run a very restricted operating system that can't be used for GNF out of the box.

To make home routers part of the GNF infrastructure, devices need to be flashed with a Linux operating system, such as OpenWRT 15.05.1 (Chaos Chalmer) that is capable of running

Linux Containers (LXC)[2]. Recently, other Linux-based, extensible operating systems for home routers have been released, such as the VyOS [3] open-source network operating system or the recent EdgeOS [153] operating system specialised for the network edge.

Public Cloud VMs Public cloud VMs can be used as cost-efficient extensions to the providers' fixed-capacity NFV infrastructure (e.g., in case extraordinary demand for vNFs need to be handled, cloud VMs can be used to host vNFs). Since public clouds provide VMs instead of bare-metal servers, GNF is designed to use these VMs as underlying virtual hosting devices.

For GNF, Ubuntu 14.04 LTS instances have been requested from all three supported cloud providers: Amazon EC2, Google Compute Engine, and Microsoft Azure. In general, small, general-purpose VMs with 1GB of RAM and 2 CPU cores can be used, however, the forwarding performance achievable depends on instance types as well as the provider, which will be evaluated in Section 5.3.

4.2.2 SDN Network Elements

GNF envisions a fully SDN-compliant, simplified OpenFlow network to be present between all vNF hosting devices and the user. Below, the requirements from the network infrastructure are highlighted.

OpenFlow SDN Devices A fully SDN-compliant network requires OpenFlow switches to be deployed throughout the network to allow centralised control over the network and to manipulate forwarding rules on all network devices (allowing hop-by-hop SDN traffic management). Specifically, GNF uses the OpenFlow 1.3 protocol to control all network devices without using any vendor extension. Section 4.5.2.1 will describe the OpenFlow rules installed in these devices.

[2]unfortunately Docker is not yet supported in OpenWRT, although LXC can be used as a substitute
[3]https://vyos.io Accessed on: 9 February 2017

GNF has no requirements for the speed of network links, the bandwidth available on all links is fed to the orchestration algorithms presented in Section 3.5.1.

Management Network for SDN SDN management networks are typically designed for sparse and latency-sensitive traffic where maintaining high throughput is not as critical as for the data-carrying production network. The management network should therefore be designed to provide reliable and consistent performance regardless of the load over the production traffic. According to related literature, management networks can be one of three types [75]:

1. In-band: the simplest way is to use the same 'in-band' network for the management and the production traffic of the tenants. In this scenario, there is no isolation between production and management traffic, therefore the management traffic is subjected to congestion or low bandwidth caused by production traffic [154].

2. Logical out-of-band (OOB): in this approach, the management network is logically separated using, for instance, VLANs or dedicated flow rules in the switches. Extending the in-band solution, this approach allows QoS enforcement to prioritise management traffic over tenant's traffic (by, for instance, assigning different queues in switches). However, as isolation can only happen in certain points of the network (at routers capable of QoS enforcement), logical OOB does not guarantee fully-fledged isolation of management and user traffic.

3. Physical OOB: a physically separated network can be set up solely for management purposes. While this incurs significant investment in new switches and network interfaces for hosts, this solution is preferred for critical environments. In fact, a physically separated management network is being deployed at many production cloud DCs [155] and it is the recommended solution for the OpenStack[4] open-source cloud software used for NFV frameworks presented in Section 2.2.5.

[4]https://www.openstack.org

For simplicity, the control network of the GNF prototype uses the same in-band network as the vNFs' traffic, without any isolation.

Connecting to the Cloud As described previously, public cloud VMs can also be part of the provider's GNF infrastructure. However, connecting an external cloud network to a provider's GNF installation requires special arrangements to achieve transparent, centralised traffic management between users and their vNFs in the cloud.

Connecting a provider's network to a cloud network can be done using various technologies that are usually provided by the cloud providers. One possible solution (that's offered by the Google Cloud Platform[5]) is to initiate a secure Virtual Private Network (VPN) connection from the internal virtual network to the provider's network. This allows accessing the services running in the cloud with internal IP addresses and therefore allows encapsulation tunnels (e.g., using GRE of VXLAN) to be created between switches in the internal infrastructure and VMs in the cloud.

Another solution is using 802.1q VLANs and dedicated connections from the internal network to the cloud provider's network. This is supported by AWS using their AWS Direct Connect service, and it eliminates VPN hardware that frequently can't support data transfer rates above 4 Gpbs, according to Amazon[6].

[5]https://cloud.google.com/vpn/docs/concepts/overview
[6]https://aws.amazon.com/directconnect/ Accessed on: 16 February 2018

4.3 GNF Service Plane

The GNF service plane provides high-level administrator access to GNF. It allows providers to either directly call the GNF Manager API or use the User Interface to manage vNF chains.

4.3.1 User Interface

The GNF web-based *User Interface (UI)* gives a graphical representation of all the managed hosting devices as well as all connected User Equipment (UE). The UI has been developed to showcase an easy, one-click assignment of network functions to active end users in the network that is not possible with the UI's of today's NFV platforms.

The UI operates by calling the REST (Representational State Transfer) API of the Manager, and gives the ability to specify vNFs and assign them to selected UE traffic. The UI is built using Node.js, an open-source JavaScript run-time environment using the Express.js[7] web application framework. The pages use HTML5, CSS3 and the Bootstrap[8] open-source front-end library.

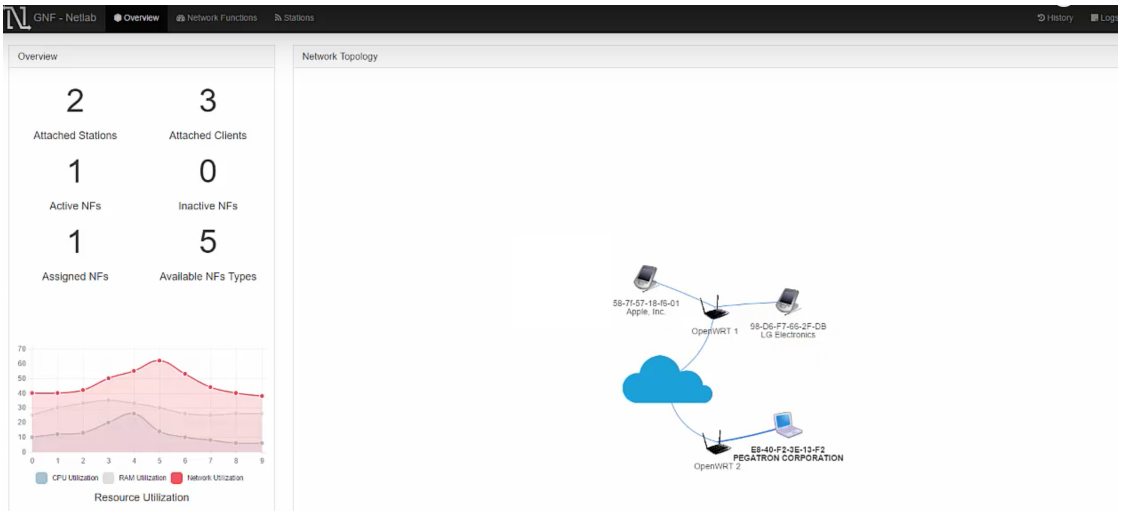


Figure 4.2: The GNF UI's network view: managed devices and users are visible for operators

[7]https://expressjs.com Accessed on: 9 February 2018
[8]https://getbootstrap.com Accessed on: 9 February 2018

Figure 4.2 shows the network view of the UI. In this page, the operator can see all the connected UE and all the managed vNF hosting devices. In this example, only two edge devices are managed, while three clients are connected (one laptop and two mobile phones). This page is built using the D3.JS[9] powerful visualisation library with a live websocket-based connection between the user interface and the backend, which allows displaying end user movements in real-time, without refreshing the page.

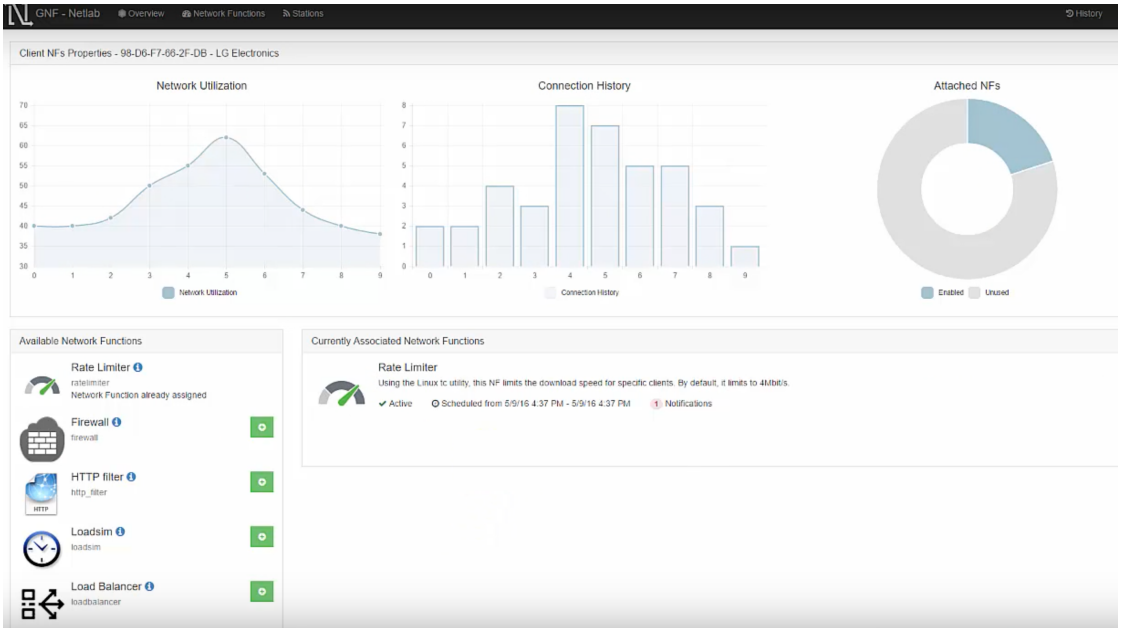


Figure 4.3: The GNF UI's device overview: operators can click on a vNF to be assigned to a user

Figure 4.3 shows the view that is available to the operator after clicking on a user device. In this page, an operator can select a vNF and assign it to a user device with a simple click. Once the green + button has been clicked, the orchestration engine will find a physical device to host the vNFs on. Moreover, as shown in the top left part of Figure 4.3, the temporal network utilisation and a connection history is collected for each UE. This allows better understanding the device's traffic patterns to enhance vNFs and the orchestration logic.

Apart from creating vNFs and assigning them to clients, the UI also displays notifications sent from the vNFs. As an example, a notification can tell the provider if an intrusion has been detected by any of the intrusion detection vNFs. The notifications received from the

[9]http://d3.js Accessed on: 19 February 2018

vNFs are linked to the user devices that help catching users performing malicious activity. Notifications can be acknowledged, deleted, and muted for a specific vNF by the operator.

4.3.2 Public API

GNF, along with the UI, also provides a simple programming interface to operators. The API is built using REST principles and uses the standard HTTP protocol for communication. For state transfer, JSON (JavaScript Object Notation), a lightweight data-interchange format has been chosen as it can be created and parsed by most programming languages efficiently.

Without detailing all public API endpoints of GNF, an important one is shown here. The */api/clients/*{*client.mac*}*/functions* endpoint allows the assignment of network functions, where {*client.mac*} is the MAC address of the client (e.g., a mobile device) to whom we would like to assign a vNF. This endpoint can be called with a HTTP POST request with the data { *policy:"parental_control", active: true* }, in which case the system assigns and activates a *parental_control* vNF to the user. The same endpoint can be called with an HTTP GET method to retrieve all associated vNFs of the user.

Other public GNF endpoints allow the removal of vNFs from a user, managing notifications (listing all of them for a user, acknowledging, deleting, muting) and managing hosting devices (e.g., removing or registering them to be part of the platform) and retrieving platform statistics such as associated vNFs and a client's network traffic. The usage statistics can be fed to external systems in case a service provider is about to charge users for vNFs.

4.4 GNF Orchestration Plane

The orchestration plane, corresponding to the NFV Orchestrator (NFVO) component of the ETSI MANO architecture [2], is implemented in two components. First, the *GNF Manager* (Section 4.4.1) that has network-wide knowledge of all vNF locations, the current topology and usage statistics from all managed devices. A separate component, the *Latency-Optimal*

Orchestrator (Section 4.4.2) has been implemented to co-operate with the Manager, providing vNF placement decisions when requested.

4.4.1 Manager

The Manager keeps a live HTTP connection with the Agents running on the vNF hosting devices to retrieve health statistics and notifications about the vNFs and the hosts. In our proof-of-concept implementation, Agents send connection events, WiFi signal statistics (signal strength, packet counters - if applicable), and CPU and memory utilisation of the device gathered from the Linux kernel. Also, the Manager stores notifications sent from the vNFs that are relayed through the Agent and provides this information to the User Interface. Notifications are arbitrary text messages from the vNFs.

The Manager, on top of exchanging information with the Agents, also talks to the SDN Network Controller that reports network-wide topology and latency information which is then given to the orchestrator introduced below. Also, it invokes the APIs of the network controller to set up connectivity between a user and the associated vNF's hosting platform by setting up flow entries. The network controller is described in Section 4.5.2 in detail.

Finally, the Manager exposes internal APIs for low-level vNF management: starting a vNF on a specific host and stopping a vNF on a host. These endpoints are used only by the orchestrator introduced below and therefore these are not supposed to be called from the UI or by the operators.

4.4.2 Latency-Optimal Orchestrator

The Latency-Optimal Orchestrator is an integral part of the GNF platform, providing the vNF placement decisions for the system. The orchestration logic presented in Section 3.5 has been implemented in this component, interacting with the Manager's internal APIs. The input required for the orchestrator from the Manager is the following:

1. Topology with hosts and their capabilities (CPU, memory, IO) - represented as a graph using the NetworkX[10] library. This information is readily available in the SDN Network Controller detailed in Section 4.5.2.

2. Latency and bandwidth limit on all the links in the topology - represented as labels on the edges of the network graph. Currently, this information is estimated using physical distances in the network topology and supplied manually to the orchestrator. In the future, SDN controllers with link latency estimation can be explored [156].

3. Active users and their vNF assignments along with vNF resource requirements (CPU, memory, IO) - represented as Python dictionaries. This information is stored in the GNF Manager.

To implement the latency-optimal "Edge vNF placement" problem shown in Section 3.5.1, the state-of-the-art mathematical programming Gurobi[11] solver has been used with its Python APIs. The solver allows setting up the binary decisions matrix, along with helper data-structures and the constraints using real-time information from the Manager.

Once the solver is invoked to minimise the objective function (Eq. 3.2), the binary decision variable matrix is populated with values that represent a new placement. This placement can be compared against the current location of vNFs and if required, vNF migrations can be applied. The following vNF migration methods can be considered:

1. Live migration of vNFs: vNFs can be migrated with a full state transfer by initiating a live migration, in case the underlying containerisation platform supports live migration (container live migration is more complex than live migration of VMs and is therefore not supported in many container platforms[12]).

2. Start new and stop old vNFs (used by GNF): containers can decouple their state from their implementation and can be designed to be stateless [134], in which case a vNF

[10] https://networkx.github.io/documentation/networkx-1.10/overview.html Accessed on 9 February 2018
[11] https://www.gurobi.com Accessed on 9 February 2018
[12] https://www.slideshare.net/Docker/live-migrating-a-container-pros-cons-and-gotchas Accessed on: 19 February 2018

migration is (1) starting a new instance of the same vNF on the new hosting device; (2) redirecting all traffic to the new vNF; and (3) stopping the old vNF.

For debugging purposes, the orchestrator also provides a separate user interface using the NetworkX module. This view can be seen in Figure 4.4, where red circles represent hosts, black lines represent network links between them, green circles are the users (there are only two users shown in this Figure), and blue circles are the vNFs. In the example shown, vNFs NF0 and NF1 belong to User0, while vNFs NF6, NF7 and NF8 belong to User1. Assignments of vNFs to hosts are shown with blue dashed lines, and as it can be seen, the vNFs are ended up close to where the users are connected: NF0 and NF1 are allocated to the host where User0 is connected, while NF6 and NF7 are allocated to where User1 is connected. NF8 has been allocated to a device a bit further away (yet this still results in the best latency for the user), due to running out of resources on the closest host after allocating NF6 and NF7.

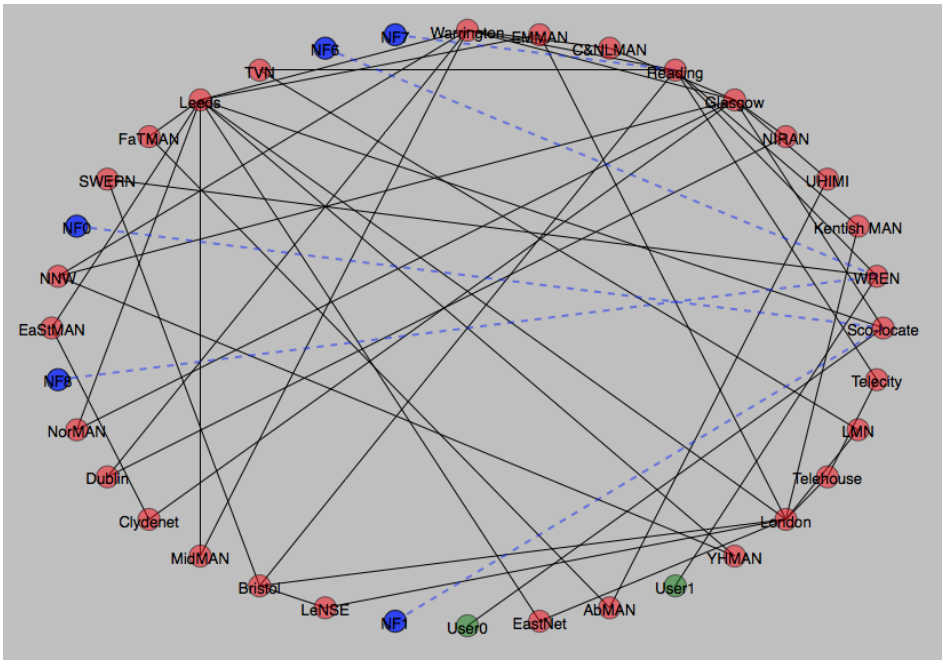


Figure 4.4: Orchestration debugger shows topology of hosts, vNFs, users and vNF assignments after orchestration.

4.5 GNF Virtual Infrastructure Management Plane

This plane of the architecture handles the network connectivity between vNF hosts and manages (starts, stops, migrates, removes, upgrades) vNFs running on these devices. The two main components performing these actions are the Agent and the Network Controller detailed here alongside with details on the traffic classification used in GNF.

Compared to other solutions presented in Section 2.3, GNF's virtual infrastructure management plane has control not only over the hosts, but also over the entire network infrastructure (enabling a unique, transparent traffic handling to be implemented), as presented below.

4.5.1 Agent

The *Agent* is a small daemon implemented in Python that runs on all hosts, let it be an edge device or IoT gateway, a cloud VM or a server inside the provider's network. The Agent is responsible for the creation and deletion of the vNFs (by calling e.g., Docker's API[13]), and for reporting periodically the state of the device to the Manager. The Agent corresponds to the VNF Manager (VNFM) and Virtual Infrastructure Manager (VIM) components of the ETSI MANO architecture [2].

When a new vNF is requested by a user from the User Interface, the Manager, after running the orchestration software, notifies the most suitable Agent which then retrieves the vNF's image from a central repository in case the image is not locally available. Docker images are layered, therefore common files are shared between images, making the disk consumption low and the downloading of vNF images from the repository fast.

If the Agent is hosted on an edge device that handles user's connections directly, it also listens and reports all client connections to the Manager by subscribing for instance to HostAPD (Host Access Point Daemon) events. In case a client leaves the edge (e.g., user roams to an

[13]https://docs.docker.com/develop/sdk/ Accessed on: 19 February 2018

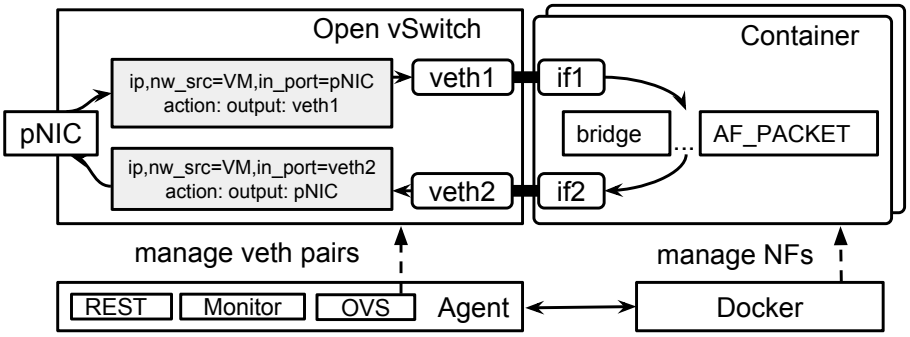


Figure 4.5: Agent's network configuration for a single container.

other edge device or disconnects from the network), the Agent gets notified to either stop, re-locate or leave the vNF running at the same device. When a new client is connected to the network, the Manager is notified to decide whether vNFs need to be started for this client or not.

On top of the lifecycle management of vNFs, the Agent is responsible for setting up the containers' virtual interfaces inside the hosting devices. In GNF, all container vNFs are connected to a local software switch (Open vSwitch 2.0.2) by two virtual Ethernet pairs, where one interface pair is used to receive traffic at the vNF (*if1*) and the other is used to send traffic from the vNF (*if2*). The connection between hosting devices is done by the Network Controller described in Section 4.5.2.

Figure 4.5 provides an overview of the responsibilities of the GNF Agent. As shown, Open vSwitch with a couple of OpenFlow flow entries is used to connect the physical interface of the device to the two virtual Ethernet pairs connecting the containers. Inside the container, the Figure shows a Linux bridge that can link the to interfaces and make the container act as a wire. The Agent itself exposes a REST API to the management network, monitors the health of the machine by calling system libraries (e.g., /sys/proc files), and communicates with Docker's API and Open vSwitch to instantiate and configure the network functions.

4.5.2 SDN Network Controller

The *Network Controller* is an SDN controller that manages transparent traffic redirection between edge devices and the central NFV infrastructure that spans across the provider's network. This component is built on top of OpenDaylight [56], an open-source, carrier-grade SDN platform that gained acceptance from service providers worldwide. A unique, transparent hop-by-hop redirection (which does not break existing connections) has been implemented through the provider's network in GNF's OpenDaylight bundle, where flow entries match on input ports and forward packets on output ports, however, tunnelling techniques can also be used to manage traffic between distributed Open vSwitch instances in networks where operators do not have access to the devices (e.g., in public clouds, where traffic can not be controlled hop by hop), as detailed later.

This SDN controller also maintains the temporal network topology and the location of all hosting devices. This is being done by using OpenFlow's link and host discovery packets. This topology information (along with latency information from the links) is an integral part of the service orchestration presented in Section 4.4.2.

4.5.2.1 Transparent Traffic Steering

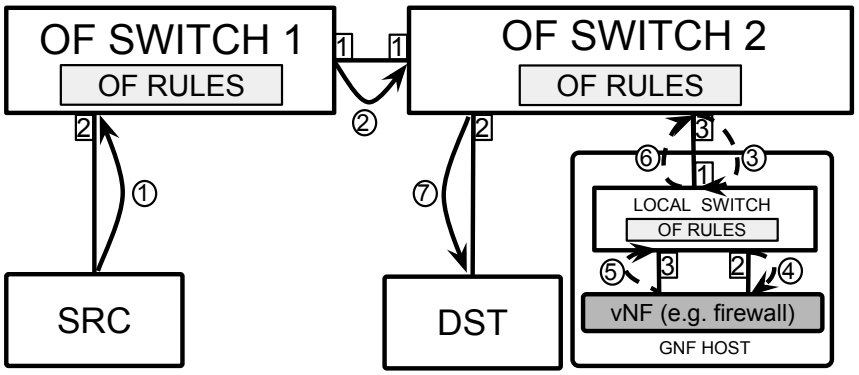


Figure 4.6: Imposing a vNF to selected traffic. Table 4.1 shows the associated OpenFlow rules installed.

Figure 4.6 shows the default traffic path using a shortest path (solid arrows), going from

source SRC to the destination DST through two OpenFlow-enabled switches. On the use of a network function, traffic must be redirected to the NFV host and to a particular vNF (dashed arrows in Figure 4.6). The hop by hop path of the traffic is shown with numbers in circles next to the arrows.

To achieve the path shown between source and destination, the OpenFlow flow rules shown in Table 4.1 are inserted to the appropriate switches, where port numbers can be seen next to the links in Figure 4.6. The first rule in Table 4.1 redirects traffic towards the NFV host, while the last two rules in the table deal with local redirection of the traffic, internal to the NFV host.

Table 4.1: OpenFlow rules to forward packets from SRC to DST through its vNF using the architecture at Fig. 4.6.

Switch	Match	Action
1	input_port: 2, eth_src: SRC	output_port: 1
2	input_port: 1, eth_src: SRC	output_port: 3
local	input_port: 1, eth_src: SRC	output_port: 2
local	input_port: 3, eth_src: SRC	output_port: 1

Since we assume that default L2 or L3 routing is still in place, there is no need for extra rules to route traffic at the egress of the NFV host to the destination (from OF SWITCH 2 to DST). For clarity, only the forward path from source to destination is shown here. The reverse path from the destination back to the source is a simple inversion of the ports in Table 4.1.

Using SDN, arbitrary steering policies can be configured using custom routing entries, dependent on the nature of the network function [24]. Some examples are given below.

Exhaustive Steering All traffic of the user goes through the vNF, allowing an inspection and alteration of the entire traffic at Layer 2 or above. Common network functions requiring exhaustive steering include Intrusion Detection and Prevention (IDPS), firewalls, and Virtual Private Network (VPN) services.

Service-based Steering A subset of traffic is routed through the vNF, while the rest of the traffic follows the default route. This approach reduces the traffic load in the traversed vNFs, allowing better scalability and denser network function collocation. Network functions such as DNS load balancer and transparent web-proxy vNFs can use this type of steering as they operate on Layer 4 with a specific service port (e.g. 53 for DNS and 80 for HTTP).

Replica Steering A replica of the traffic (or a subset of the traffic) is forwarded to the vNF in this scenario, using OpenFlow's capability to output packets on multiple ports. Doing so, this steering prevents performance degradation on the data path such as additional latency. Once a packet has traversed the service, it is discarded. Example vNFs that can use such steering policy are services that only inspect but never modify data, such as, e.g., monitoring, intrusion detection, and traffic characterisation vNFs.

4.5.2.2 Traffic Classification

Traffic classification is required to match and forward packets based on the previously described steering policies. To support fine-grained, yet standard and high-performance classification in the data plane, GNF relies on OpenFlow 1.3 classifiers that allow packets to be matched on properties such as input port and up to 7 protocols, including Ethernet (L2), MPLS (L3), ARP (L3), IP (L3), SCTP (L4), ICMP (L4), TCP (L4) and UDP (L4) headers, as shown in Figure 4.7. However, some protocols or fields are optional, such as MPLS, ARP or Tunnel ID in L1.

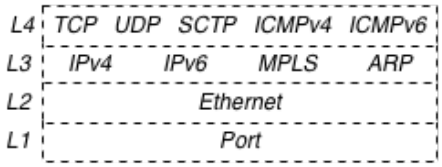


Figure 4.7: OpenFlow 1.3 Classifier Stack (source: http://flowgrammable.org)

Traffic classification can also exploit flow priorities and timeouts provided by OpenFlow, meaning that operators can set overlapping classifiers with different priorities and lifetimes.

A good use of flow priorities is when a flow is set up to filter only a subset of the traffic without modifying the already existing flow entries. GNF uses flow priorities with timeouts with replica steering introduced above: the provider can set to receive a copy of all traffic from a particular user by inserting a high priority flow rule with a 10 second timeout, getting only a 10 second sample of the traffic directed to a particular vNF (e.g., network monitoring vNF).

4.5.2.3 Traffic Tunnelling

GNF can be deployed in case a provider does not have access to OpenFlow devices in the path and therefore can't install flow rules to redirect traffic hop by hop as shown in Section 4.5.2.1. As a concrete example, this is the case when GNF vNFs are deployed in public clouds [25], where the entire network is virtualised and only IP addresses are provided to host VMs after connecting to the cloud as described in Section 4.2.2.

To address the limitations of traffic forwarding while achieving transparent traffic steering in public cloud networks, GNF relies on tunnelling protocols such as Virtual eXtensible LAN (VXLAN) [157] or Generic Routing Encapsulation (GRE) [158]. By using tunnelling protocols, arbitrary Layer 2 (Ethernet) traffic can be encapsulated within an UDP or IP frame and forwarded to any other host using traditional routing mechanisms without modification of the tunnelled traffic. Other encapsulations, such as IP-in-IP can be used, however as it encapsulates IP packets, it would prevent the vNFs from processing non-IP traffic such as L2 broadcast messages or IPv6. Another option can be using TCP tunnels between all VMs (e.g. an SSH tunnel), however, using a reliable transport layer would deteriorate the performance of encapsulated real-time application traffic (such as VoIP, media streams). Therefore, by encapsulating Ethernet frames in IP (GRE) or UDP (VXLAN) packets, flexible and transparent traffic steering can be achieved without configuring all devices along the path.

Once the forwarding limitation has been addressed through tunnelling, the problem of protocol support arises. As VXLAN encapsulates Ethernet frames within UDP datagrams while GRE uses IP packets, it would be sensible to use GRE for a lower encapsulation overhead,

	EC2	Azure	GCE
GRE	✓	✗	✗
VXLAN	✓	✓	✓

Table 4.2: Support for GRE and VXLAN tunnelling protocols in public cloud providers [25].

leading to higher MTU and therefore higher goodput. However, as shown in Table 4.2 and discussed in the Section 3.3.5, providers such as Azure and Google GCE drop any traffic that is not TCP, UDP or ICMP, preventing GRE to be used by tenants. With the universal support of UDP, VXLAN tunnels can be used in today's public cloud providers to configure tunnels at the source and destinations hosts to steer traffic through a VM acting as the vNF host.

VXLAN tunnels are supported by OpenFlow 1.3 actions, therefore OpenFlow switches can set up and remove tunnels without impacting the traffic. In Table 4.3, we show the OpenFlow flow entries installed at the software switches to traverse all traffic from the edge device through a vNF hosted on a VM in the cloud, using the exhaustive steering policy introduced before. In comparison to the flows set up in a deployment where traffic is managed hop by hop as shown in Section 4.5.2.1, when using tunnels, the next hop of the packets is set with *set_tunnel* and *set_field* actions (hop 1, 3, 4, 6 in Table 4.3). The flows in Table 4.3 also ensure that the replies from the Internet are forwarded through the same vNF, assuming that the edge nodes are always on the path between the user and the Internet.

4.6 Network Function Implementations

While there are many possibilities for network operators to implement network functions, this section details a number of Docker-based exemplar network functions for GNF, available through our Docker[14] and GitHub[15] repositories. These vNFs have been selected to reflect the most popular vNFs in provider's networks, according to Sherry et al. [69].

This section first shows the generic setup of GNF vNFs and then provides various example vNFs that have been implemented for GNF.

[14]https://hub.docker.com/u/glanf/
[15]https://github.com/glanf/

Hop	Host	Match	Action
1	EDGE	in_port=USER eth_src: SRC	set_tunnel:100 set_field:VM->tun_dst output:tunnel
2	VM	in_port=eth0 tun_id=100	output:veth1 (vNF1's input)
Processing of outgoing packets in vNF1			
3	VM	in_port=veth2 (vNF1's output)	set_tunnel:101 set_field:EDGE->tun_dst output:tunnel
4	EDGE	in_port=CORE eth_dst=SRC	set_tunnel:102 set_field:VM->tun_dst output:tunnel
5	VM	in_port=eth0 tun_id=102	output:veth1 (vNF1's input)
Processing of incoming packets in vNF1			
6	VM	in_port=veth2 (NF1's output)	set_tunnel:103 set_field:EDGE->tun_dst output:tunnel
7	EDGE	eth_dst: SRC tun_id=103	output:USER

Table 4.3: OpenFlow rules required to forward packets from the edge node through vNF1 hosted on VM using tunnels [25].

4.6.1 Generic vNF Setup

A vNF in the GNF platform consists of the following files that allow the vNF to be easily changed and distributed to multiple platforms.

- Dockerfile - this is a file that describes the inheritance between vNF images and contains the commands to build the vNF from source as well as commands to run when the vNF is started. This file also defines the ports available to the world in case the vNF provides a service for other vNFs.
- (optional) applications: these can be arbitrary scripts or programs invoked after the vNF has started.
- (optional) files: any file can be placed inside the container (such as signature databases for intrusion detection vNFs).

As a particularly basic vNF, the *wire* forwards packets from its input to its output interface. The Dockerfile for the wire vNF is shown in Listing 4.1.

```
1 FROM glanf/base
2 MAINTAINER Richard Cziva
3 ENTRYPOINT ifinit && \
4  brinit && \
5  /bin/bash
```

Listing 4.1: Wire vNF's Dockerfile

The "wire" vNF inherits from a GNF base image, which is a basic installation of Ubuntu 16.04 LTS with networking packages such as *tcpdump*, *openvswitch-common* and *bridge-utils* installed. After specifying image inheritance and stating the maintainer of the package, this Dockerfile specifies that once the vNF is started, it has to call the *ifinit* first. This script essentially blocks until two interfaces (*if1* and *if2*, as shown in Figure 4.5) are injected to the container by the Agent described in Section 4.5.1). After the interfaces appear and the *ifinit*

script finished waiting, this container executes the *brinit* script (also located in the GNF base image) that creates a Linux bridge and adds the two interfaces to it.

Below, the contents of the *brinit* script are shown (Listing 4.2). This bash script sets up a Linux bridge and puts the two interfaces into the newly created bridge *br0*. Finally, it brings the bridge up, so it can start forwarding traffic.

```
1 #!/bin/bash
2   brctl addbr br0
3   brctl addif br0 if1
4   brctl addif br0 if2
5   ifconfig br0 up
```

Listing 4.2: Brinit script used to set up a bridge

As Docker vNF images can be stored in public repositories or retrieved using untrusted networks, content trust can be a central concern, as identified in Section 3.2. To mitigate this problem, Docker allows the enforcement of client-side signing and verification of image tags which technically means that image publishers can sign their images, while providers can ensure that the images they are deploying are signed.

4.6.2 Firewall

A firewall is the most popular network security module today, deployed to monitor network traffic and filter packets based on a predefined ruleset. In fact, firewalls are the most popular vNFs deployed after routers and switches according to Sherry et al. [69].

For this vNF, the Linux standard *iptables* utility has been used. Below, the entire code defining a firewall is shown in Listing 4.3. As seen, it is only extending the wire vNF with two *iptables* commands inserted to the FORWARD chain of *iptables*. In particular, this example accepts only packets that are destined to port 80 (HTTP traffic) and discards everything else.

```
1 FROM glanf/base
2 MAINTAINER Richard Cziva
3 ENTRYPOINT ifinit && \
4   brinit && \
5   iptables -A FORWARD -p tcp --dport 80 -j ACCEPT && \
6   iptables -A FORWARD -j DROP && \
7   /bin/bash
```

Listing 4.3: GNF firewall vNF's Dockerfile

While this firewall is specific to allow only port 80 and drop everything else, it can be made generic with the use of environmental variables (or using scripts for more complicated logic) and be personalised to a user. Environment variables can be passed to containers once started, with default values specified in the Dockerfile. Listing 4.4 shows a small modification to Listing 4.3, making the port configurable for every vNF.

```
1 FROM glanf/base
2 MAINTAINER Richard Cziva
3 ENV PORT 80
4 ENTRYPOINT ifinit && \
5   brinit && \
6   iptables -A FORWARD -p tcp --dport $PORT -j ACCEPT && \
7   iptables -A FORWARD -j DROP && \
8   /bin/bash
```

Listing 4.4: Dockerfile of GNF's personalisable firewall vNF

4.6.3 HTTP Content Filter

Content filtering involves preventing access to certain items, which may be harmful if opened or accessed. While usually they are applied for security purposes, in many cases they implement company policies related to information system usage (e.g., blocking social networking sites at the workplace). The most common items to be filtered in the network are executables, e-mails or websites [159].

An application-level traffic filter has been implemented using Python's Scapy[16] traffic manipulation library and Netfilter[17] queues as an example. The implementation of this vNF differs slightly from the previously presented wire and firewall vNFs, as it requires a separate Python program to be run for packet processing. This is done by directing traffic to a specific Netfilter queue using *iptables* and invoking a Python program that reads from this queue and makes either an accept, modify, or drop decision on each packet. Below, in Listing 4.5, the redirection to the queue and the invocation of the packet filter script is shown.

```
1 ENTRYPOINT ifinit && \
2     brinit && \
3     iptables -A FORWARD -j NFQUEUE -p tcp --destination-port 80 --queue-num 1 && \
4     python ./data/http_filter.py
```

Listing 4.5: HTTP filter redirects traffic to a nfqueue using iptables and invokes a Python script

The source code of the actual HTTP filter implementation is shown in Listing 4.6. As it can be observed, this Python program reads from the queue, parses all packet payload and looks for the string 'hack' in the packet. In case the packet examined contains this string, the packet is dropped and a notification is sent to the Manager with a message "HTTP Filtered content". Otherwise, the packet is accepted to be forwarded.

```
1  from netfilterqueue import NetfilterQueue
2  from scapy.all import *
3  import requests
4
5  url = 'http://172.17.42.1:8081/notification'
6
7  def callback(pkt):
8      parsed = pkt.get_payload()
9      print parsed.encode('hex')
10     if 'hack' in parsed:
11         pkt.drop()
```

[16] http://www.secdev.org/projects/scapy/ Accessed on: 19 February 2018
[17] https://www.netfilter.org/projects/libnetfilter_queue/ Accessed on: 18 February 2018

```
12          print "packet dropped"
13          requests.post(url, data="HTTP Filtered content")
14      else:
15          pkt.accept()
16
17  nfqueue = NetfilterQueue()
18  nfqueue.bind(1, callback)
19  try:
20      nfqueue.run()
21  except KeyboardInterrupt:
22      print 'Keyboard interrupt, stopping.'
```

Listing 4.6: HTTP filter vNF's Python implementation

4.6.4 Rate Limiter

Rate limiting (also called as bandwidth enforcement) can be used to control congestion on a network, or to give certain types of traffic lower priority and reserve bandwidth for higher priority traffic (e.g., VoIP). This is especially useful when applied at the edge of network, close to the source of the information [160].

For this vNF, the *tc* (traffic control) module of the Linux kernel was selected. In particular, traffic shaping has been implemented to limit the bandwidth to a threshold value. Apart from shaping, *tc* can also provide scheduling, policing and dropping of packets. Listing 4.7 shows the full implementation of this vNF . As presented, the required bandwidth can be set with the *BITRATE* environment variable, which is set to 1mbps by default. The commands between line 7 and 9 (for *if1*) and 10 and 12 (for *if2*) set up virtual queues (*qdisc*) that use a Hierarchical Token Bucket (HTB) rate limiter to throttle bandwidth to the desired bitrate specified in lines 9 and 12 with *rate* and *ceil* commands.

```
1  FROM glanf/base
2  MAINTAINER Richard Cziva
```

```
3
4  ENV BITRATE 1mbps
5
6  ENTRYPOINT ifinit && brinit && \
7      tc qdisc add dev if2 root handle 1: htb default 0xfffe && \
8      tc class add dev if2 classid 1:0xffff parent 1: htb rate 1000000000 && \
9      tc class add dev if2 classid 1:0xfffe parent 1:0xffff htb rate $BITRATE ceil $BITRATE && \
10     tc qdisc add dev if1 root handle 1: htb default 0xfffe && \
11     tc class add dev if1 classid 1:0xffff parent 1: htb rate 1000000000 && \
12     tc class add dev if1 classid 1:0xfffe parent 1:0xffff htb rate $BITRATE ceil $BITRATE && \
13      /bin/bash
```

Listing 4.7: Rate limiter vNF

4.6.5 DNS Load Balancer

Load balancing is used today to provide availability, scaling, and resilience for geographically distributed applications. Technically, this can be performed in different layers of the network (e.g., application or transport layer load balancing) [161]. As an example, a DNS load balancer can override domain name resolutions to various IP addresses or perform DNS services entirely.

A basic, stateless DNS load balancer has been implemented for GNF using the Scapy library. This load balancer is essentially a Python script that intercepts DNS queries to a certain domain (the domain to be load balanced) and creates a DNS query reply containing an IP address for the domain selected. This IP address is selected from a given pool of addresses using a random selection which can be extended to use other load balancing techniques (e.g., round robin assignment, static assignment etc), as shown in Listing 4.8.

```
1  from scapy.all import *
2
3  domain = 'facebook.com'
4  webservers = ['10.0.0.1', '10.0.0.2']
5
```

```
6  # implement your load balancing here
7  def getwebserver():
8    return random.choice(webservers)
9
10 def callback(payload):
11   data = payload.get_data()
12   pkt = IP(data)
13   if not pkt.haslayer(DNSQR):
14     payload.set_verdict(nfqueue.NF_ACCEPT)
15   else:
16     if domain in pkt[DNS].qd.qname:
17       webserver = getwebserver()
18       spoofed_pkt = IP(dst=pkt[IP].src, src=pkt[IP].dst)/\
19                 UDP(dport=pkt[UDP].sport, sport=pkt[UDP].dport)/\
20                 DNS(id=pkt[DNS].id, qr=1, aa=1, qd=pkt[DNS].qd,\
21                 an=DNSRR(rrname=pkt[DNS].qd.qname, ttl=10, rdata=webserver))
22       payload.set_verdict_modified(nfqueue.NF_ACCEPT, str(spoofed_pkt), len(spoofed_pkt))
23
24 def main():
25   q = nfqueue.queue()
26   q.open()
27   q.bind(socket.AF_INET)
28   q.set_callback(callback)
29   q.create_queue(0)
30   q.try_run()
31
32 os.system('iptables -A FORWARD -p udp --dport 53 -j NFQUEUE')
33 main()
```

Listing 4.8: DNS load balancer (snippet)

While this is a simple example, there is a lot of potential to further develop similar load balancer vNFs to exploit context information when they are deployed at the edge of the network. For instance, these vNFs can exploit accurate physical location of the devices and route traffic accordingly, allowing much finer geographical partitioning than current techniques that

rely on city or country-level load balancing [162].

4.6.6 Network Monitoring

This vNF is a proof-of-concept network monitoring tool, based on the Linux *netstat* program that reports various network-related information to the operator, such as network connections or interface statistics[18]. Statistics currently given are the following:

- number of collisions (*collisions*)
- number of multicast packets (*multicast*)
- received total bytes (*rx_bytes*)
- received compressed packets (*rx_compressed*)
- received packets dropped (*rx_dropped*)
- received number of packets (*rx_packets*)
- sent total bytes (*tx_bytes*)
- sent compressed packets (*tx_compressed*)
- sent packets dropped (*tx_dropped*)
- sent number of packets (*tx_packets*)

These counters are initiated for each container and they are available for each virtual interface in the container (i.e., both ingress and egress). In order to calculate rates, operators need to read these statistics periodically and compare the differences in counters. This can be done inside the network function or using a separate software where these statistics are processed. In this latter case, one can envision many monitoring network functions sending port counters to a stream data analysis platform (e.g., Apache Spark [163]) and set alerts to specific events such as dropping or increasing rates of traffic detection.

[18]https://www.kernel.org/doc/Documentation/ABI/testing/sysfs-class-net- statistics Accessed on: 21 February 2018

```
1  import requests
2  import time, threading
3  import subprocess
4  import os
5  from subprocess import Popen
6
7  delta = float(os.environ.get('DELTA'))
8
9  if delta is None:
10   print 'Env has not been set properly'
11   sys.exit()
12
13 url = 'http://172.17.42.1:8081/notification'
14
15 def stat():
16       threading.Timer(delta, stat).start()
17       p = Popen(["netstat", "-i"], stdout=subprocess.PIPE, stderr=subprocess.PIPE)
18       out, err = p.communicate()
19       x = out.split('\n')
20       inbytes = x[4].split()[3]
21       outbytes = x[5].split()[3]
22       notif = "InBytes = " + inbytes + ", OutBytes" + outbytes
23       requests.post(url, data=notif)
24
25 stat()
```

Listing 4.9: Network Monitoring vNF

As shown in Listing 4.9, the network monitoring proof of concept vNF implemented for GNF reads all statistics by calling the *netstat* command as a sub-process every minute (configurable with the DELTA environment variable) and sends these statistics to the Manager.

4.6.7 Intrusion Detection using SNORT

SNORT [84] is a lightweight, open-source Intrusion Prevention and Detection System (IDPS) that performs real-time traffic analysis and packet logging for relatively lightly utilised networks. It is being used in thousands of networks worldwide by using the world's most accurate commercial network attack signature database from Cisco Talos alongside with additional rules collected by the open-source community.

In this vNF, SNORT has been configured as an in-line IDPS using *AF_PACKET* with an alert monitor that logs alerts to a file located inside the container. After logging a potential alert (e.g., port scanning, DDoS attack), another script (called as *logrunner.py*) which is continuously reading this file sends notifications to the Manager that allows operators to track intrusion incidents on the GNF UI.

While containers are usually designed to run a single process, this vNF needs to run two programs simultaneously: SNORT itself and the logrunner script. This is done by using a wrapper software called *supervisord* [19] which is given to the container as an entry point to start the sub-processes inside the container as shown in Listing 4.10. *Supervisord* is also used to monitor the processes in the container to restart any process in case of a failure.

```
1 [supervisord]
2 nodaemon=true
3 [program:snort]
4 command=/usr/sbin/snort −c /etc/snort/snort.conf −i if1:if2
5 [program:logrunner]
6 command=/usr/bin/python /data/logrunner.py /var/log/snort/alert
```

Listing 4.10: Supervisord configuration for SNORT vNF processes

[19] http://supervisord.org Accessed on 18 January 2018

4.7 Summary

This chapter introduced the implementation of the GNF container-based NFV framework. First, it outlined the overarching architecture in Section 4.1. Then, the requirements for the infrastructure were presented in Section 4.2 showing Operating System selections for various vNF hosts as well as details on the network setup, such as OpenFlow version and devices required for GNF.

Sections 4.3, 4.4 and 4.5 detailed the three separated planes of GNF: service, orchestration and VIM, respectively. In all of three of them, the software components implemented have been outlined and the communication between the various software components were detailed. Particular details were given on how GNF manages transparent network traffic steering through vNFs in an OpenFlow network or in the cloud and how the orchestrator component implements GNF's latency-optimal vNF orchestration.

Finally, Section 4.6 presented numerous example GNF network representing the most popular types of vNFs in today's networks according to surveys [69]. These vNFs demonstrate the simplicity of creating new network functions relying on actively maintained and understood open software. As it was presented, less than 60 lines of code were required to write vNFs such as a HTTP filter, a load balancer or a network monitor, and less than 30 lines of code were required to create re-distributable container images with Docker for each of these example vNFs.

The next chapter evaluates GNF from various angles: it shows the benefits of container vNFs, examines the performance of public cloud VMs, and presents use-cases for edge NFV. Also, it evaluates the performance of the latency-optimal vNF allocation scheme presented in Section 4.4.

Chapter 5

Evaluation

5.1 Overview

The evaluation shown in this chapter has been selected to show the most important characteristics of the GNF platform, combining simulation results with real measurements taken from public clouds and commodity NFV servers.

First, Section 5.2 shows the performance benefits of lightweight, container-based virtualisation including experimental results on memory consumption, chaining properties, delay introduced by containerisation, and the time it takes to start or stop a container. Some experiments compare container network functions with unikernels and VMs to justify the benefits of containers over these virtualisation technologies introduced in Section 3.4.

Since GNF can outsource vNFs to public clouds as described in Section 3.3.5, selecting the right public cloud provider as well as the right instance type is important to achieve a desired performance with the lowest possible cost. To show these differences, GNF has been evaluated on three major public cloud providers (Google Compute Engine, Microsoft Azure, Amazon EC2) in Section 5.3.

Finally, in Section 5.4, the benefits of the proposed dynamic, latency-optimal vNF placement orchestration are analysed. First, the latency benefits of running vNFs at the edge as opposed to a distant cloud are shown. Then, different dynamic placement schedulers are compared

with regards to the number of latency violations encountered by the users and the number of migrations taken by the provider.

5.2 Performance of Container vNFs

This section highlights the most important characteristics of container vNFs that were measured on a commodity NFV server. The selected measurements highlight performance characteristics most relevant to the requirements of a lightweight NFV system, including achievable throughput, delay introduced, chaining properties as well as start and stop time of vNFs.

The contributions of these measurements are to compare container vNFs with other vNFs (e.g., VMs and ClickOS) and to present measurements that are most relevant to NFV systems, such as delay introduced, chaining properties and encapsulation overhead.

5.2.1 Evaluation Environment

The following measurements were taken using a single mid-range machine with a commodity Intel i7 CPU and 16GB of DDR3 memory. Ubuntu 14.04 LTS with Linux kernel 3.13 was installed on the host alongside with Docker 1.6.2. The pipework script[1] was used to set up virtual Ethernet pairs for all containers.

5.2.2 Delay of Containers

Keeping the delay introduced by vNFs low is important in order to implement transparent services and therefore it is a key benchmark for NFV technologies.

In Figure 5.1, we compare the delay introduced by different virtualisation platforms through showing idle round-trip-time (RTT) measurements taken with the ping utility. As shown, XEN VMs can run in either dom0 or domU, where dom0 is an abbreviation for 'domain zero', a privileged mode VM that manages all physical hardware and provides virtualised

[1] https://github.com/jpetazzo/pipework Accessed on 22 February 2018

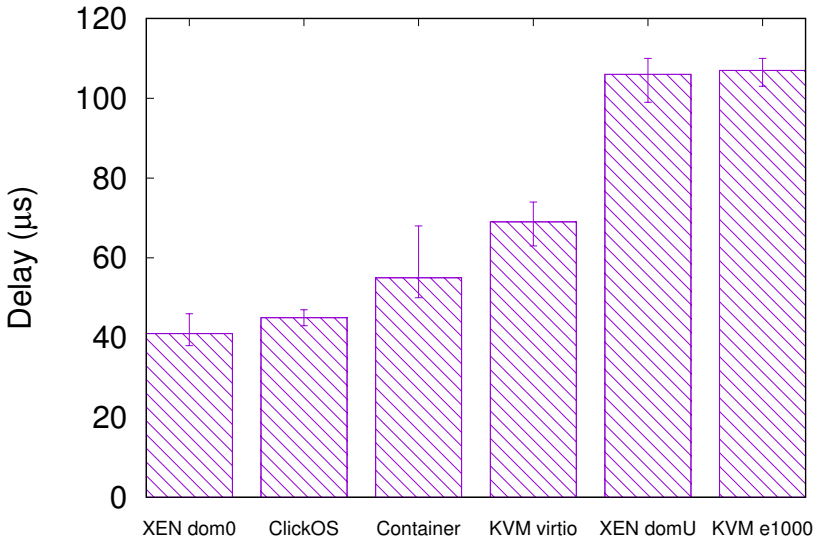


Figure 5.1: Idle ping delays [6].

resources (e.g., network access) to unprivileged, guest domains (abbreviated as domU). For KVM, Figure 5.1 evaluates two network drivers: the general and widely used virtio network virtualisation driver and the Intel specific e1000 driver.

While ClickOS reaches slightly lower delay than containers, Click is built on top of a modified, specialised VM, that optimises packet forwarding performance. On the other hand, container-based functions use unmodified containers on a standard Linux kernel, hence allowing deployment on devices that do not support hardware virtualisation (e.g., all the CPE devices and home routers). As it is also visible, other VM-based technologies such as KVM or XEN domU VMs result in a much higher delay, which is mainly attributed to the packet copy required from the hypervisor to the VMs [21].

5.2.3 Instantiation Time of Containers

In order to provide high flexibility for vNF placement and to support quick vNF migration, the time it takes to create, start, and stop vNFs is crucial for lightweight NFV platforms, as discussed in Section 3.3. The importance of instantiating time is amplified in a network edge scenario, where moving clients require frequent vNF spin-ups and tear-downs.

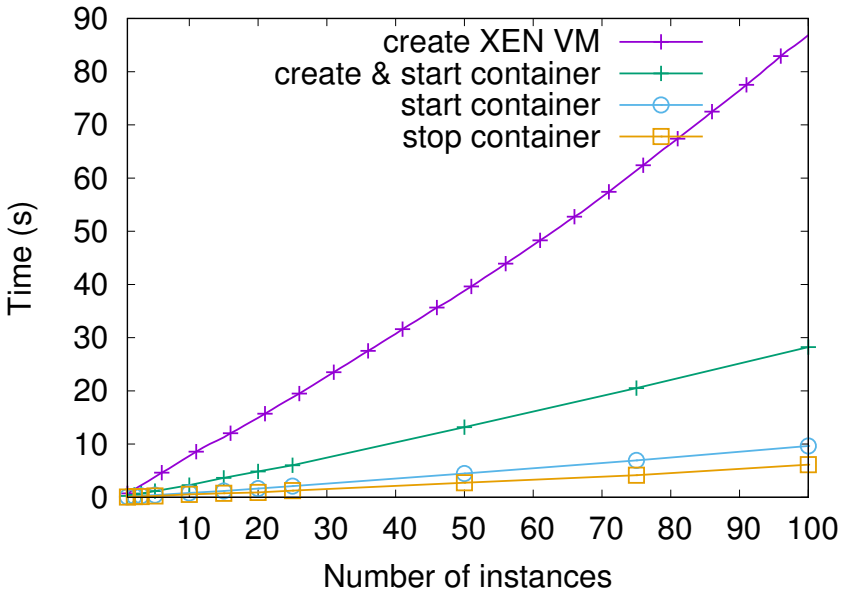


Figure 5.2: Create, start and stop time [6].

Figure 5.2 shows the time required to create, start, and stop containers versus creating (initialising from the VM's configuration file) traditional XEN VMs. As shown, 50 container vNFs can be created and started in 10 seconds, while it takes more than 40 seconds just to initialise the same amount of XEN VMs that have not even booted up. Clearly, traditional VM technologies are not suitable for highly multiplexed, highly mobile vNFs, since frequently moving clients would require new vNFs to be set up and ready to forward traffic in a matter of seconds.

5.2.4 Memory Requirements of Containers

As described in Section 3.2.4, GNF vNFs are designed for devices with relatively low memory on board (e.g., an example commodity home router has 512 MBs of total memory). Therefore, it is important to compare memory requirements of containers with other virtualisation technologies. Below, GNF containers are compared only against ClickOS unikernels, since traditional VMs would require a memory in the order of 100s of MBs per VM, depending on the installed OS and the statically assigned memory [164]. This high memory requirement of VMs eliminates them from being used in low-memory devices, however, similar to containers, ClickOS unikernels share memory between vNFs [21], resulting in a much

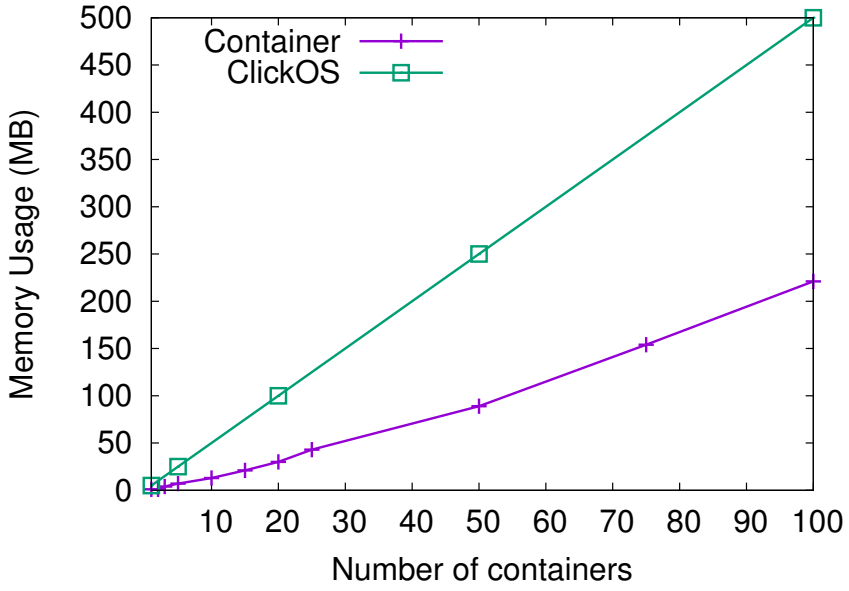


Figure 5.3: Idle memory consumption of container vNFs [6].

lower memory requirements compared to traditional VMs.

As it is highlighted in Figure 5.3, the idle memory requirement for one container is about 2.21MB, which linearly scales to only 221MB with 100 idle containers. As it is also shown, ClickOS has a relatively low memory requirement using a idle container, but it still requires more than twice the amount of memory per vNF [21] as it replicates low-level components such as packet parsing for each vNF requiring statically assigned memory, while containers share these functionality at the kernel, outside the container's memory space, thus being a potential bottleneck.

5.2.5 Throughput of Container Chains

We have used *iperf* to measure maximum throughput between the source and destination hosts connected via chained *wire* vNFs. The wire functionality is a standard Linux bridge forwarding traffic from the ingress to the egress ports of the NF as described earlier in Section 4.6.1. It is therefore the simplest form of vNF and can be used to evaluate the minimum performance impact of a virtualised service. The following experiments use a single MSS-sized TCP stream and the default networking stack configuration of Linux kernel 3.13 (TCP Cubic; initial congestion window of 10 segments; minimum retransmission timeout of

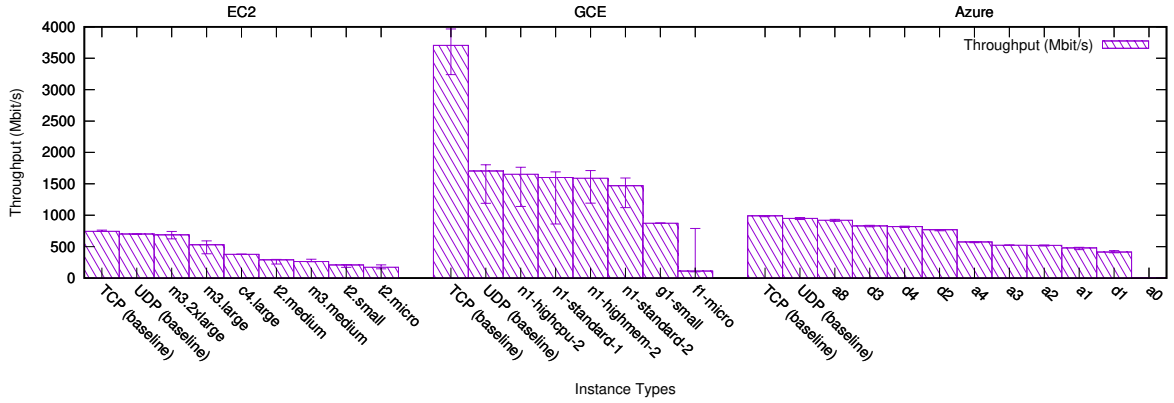


Figure 5.4: Throughput and delay of chained container vNFs [24].

200ms; window scaling, timestamps, selective acknowledgement and server-side ECN).

Figure 5.4 shows the packet processing throughput with vNF chains hosted on a single host and is therefore not limited to the speed of the topology (e.g., 1 Gbps physical switches or network cards). In this figure, we show that the container-based approach to vNFs significantly outperforms unikernels, with a single wire container vNF outperforming a ClickOS wire by more than 4Gb/s, as shown in Figure 5.4. It is also evident that containers scale better as the vNF chain grows: with 9 containers chained together, packets are processed at 13.8Gb/s compared to 3.6Gb/s using ClickOS, while over Gigabit speeds can be still maintained for 100 chained containers. Note that these results evaluate simple wire vNFs as a baseline and can be explained by the fact that containers do not copy the packets from kernel to user space as ClickOS does [24].

5.2.6 Encapsulation Overhead on Throughput of GNF vNFs

In this experiment, the performance penalty of containerisation is examined. Specifically, the throughput through four GNF vNFs is compared with their the same functions deployed without using containerisation (called native deployment).

Figure 5.5 presents the normalised throughput at the 90^{th} percentile of the following example vNFs:

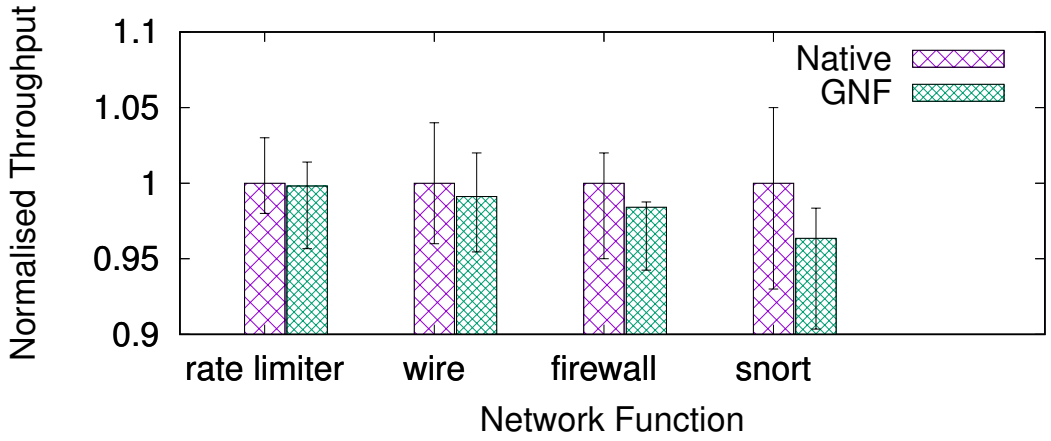


Figure 5.5: Normalised throughput of four vNFs comparing native performance with GNF containerised performance at the 90th percentile [25].

1. rate limiter: using the *tc* utility, this vNF caps the traffic to a specified bandwidth limit, as detailed in Section 4.6.4.

2. wire (bridge): using a simple Linux bridge, this vNF forwards all packets without modification, as detailed in Section 4.6.1.

3. firewall: this vNF relies on the well-known *iptables* to filter packets, as detailed in Section 4.6.2.

4. snort: the classic SNORT intrusion detection software, as detailed in Section 4.6.7.

The results have been normalised to show the impact of containerisation on performance. The actual throughput achievable of these vNFs is specific to vNF configuration (e.g., configuration of SNORT features for the SNORT vNF) and the underlying hosting device (i.e., vNFs will perform worse running on a device with lower specifications).

As shown in Figure 5.5, containerisation adds only a minimal penalty to the throughput. The performance difference between the wire (software switch) and the rate limiter vNFs is less than 1% and in the worst case for the SNORT vNF it is maximum 3.4% which can be attributed to SNORTs heavy resource usage impacting process scheduling on the host device. In the current implementation, a virtual Ethernet pair was used to link OvS and the vNF, however multiple recent benchmarks highlight that OvS internal ports provide better

performance, potentially closing the small gap between native and containerised deployment performances [165].

5.3 GNF in Public Clouds

On top of running vNFs at the edge of the network, GNF can manage vNFs hosted on public cloud VMs. This allows providers to outsource vNFs in case there are no resources available at the edge of the network or at the internal NFV infrastructure of the provider. This gives flexibility to handle increased demand for vNFs, such as at sports events [25, 141].

However, as described in Section 3.3.5, public clouds hide details on their network infrastructure and do not provide performance guarantees. Therefore, it is important for network providers to understand the performance achievable for the vNFs when they are outsourced to the cloud. The evaluation presented below evaluates GNF vNFs' throughput achievable with various providers (Amazon Elastic Compute Cloud (EC2), Google Compute Engine (GCE) and Microsoft Azure) is shown. and instance types as well as the delay introduced by redirecting traffic inside a cloud network.

5.3.1 Evaluation Environment

All measurements were taken in parallel, within a short time period to avoid differences in changing network characteristics. The VMs were started in the Western European region of all the providers. The measurements have been repeated 10 times at every measurement round. Each measurement round involves the request of new instances from the cloud provider to get a new and randomly allocated set of participating VMs. When VMs got co-located (giving a pairwise throughput of 4-5 Gbit/s), our measurement script requested new VMs to exclude this scenario. To generate traffic, high-performance instances (c4.large in EC2, D2 in Azure, n1-standard-2 in GCE) were used as source and destination VMs to eliminate CPU bottlenecks. Regardless of the provider, each VM used the default image of

a Ubuntu Server 14.04 LTS provided with an unmodified version of Open vSwitch 2.0.2. To measure throughput both for UDP and TCP, Iperf3[2] was used.

It is important to note that even with repeated measurements taken in the same geographical region, the results below only present a snapshot of what clouds offer. The variable performance provided by clouds has been examined before by Guohui Wang and Eugene Ng [166] and Isoup and others [167], where both studies presented different performance depending on applications, instance types used, time of the day, and other factors.

5.3.2 Throughput of Various VM Types

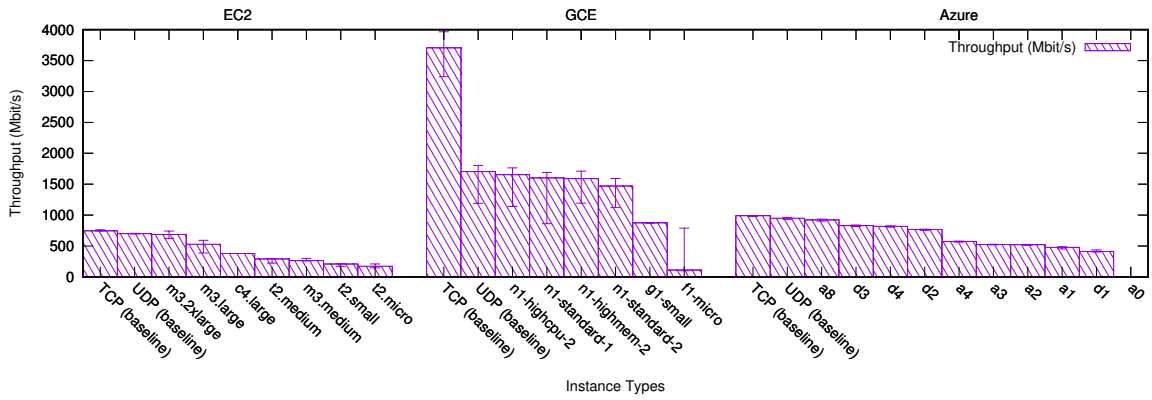


Figure 5.6: RTT and end-to-end throughput evaluation on steered traffic through varying EC2, GCE and Azure VM instance types [25].

In this Section, the end-to-end throughput through a GNF vNF is evaluated with the vNF hosted on different cloud VMs. Using VXLAN tunnels, the impact of the different hosting instance types on the network performance is shown. For this experiment, each each VM runs a GNF wire vNF (as shown in Section 4.6.1), forwarding transparently the traffic from a source VM to a destination VM and vice-versa.

Figure 5.6 compares the UDP and TCP performance of the direct traffic from a source VM to a destination VM (baseline), to the maximum throughput achievable when traffic is forwarded through a wire vNF runs on a third VM between the source and the destination VMs.

[2]http://software.es.net/iperf/ Accessed on 22 February 2018

As the VMs' performance is directly related to the instance type selected by the operator, the differences between different instances are evaluated, ranging from a price per hour of $2.450 (Azure A8) to $0.012 (GCE f1-micro). The instance name shown in the x-axis of the Figure 5.6 represents the instance type of the VM hosting the vNF, for instance, t2.micro represents the throughput from a c4.large instance (source) to a c4.large instance (destination) through a wire vNF hosted on a t2.micro instance.

As shown in the Figure, baseline UDP performance is always lower than baseline TCP performance for all providers. This is important, as VXLAN operates over UDP and therefore any traffic that is encapsulated in VXLAN is capped at UDP's performance. The reason of this difference can be twofold. First, cloud providers apply different QoS policies between UDP and the rest of the traffic, as TCP is the dominant protocol with 99.91% of the traffic [168], and other protocols (e.g., UDP) without congestion control can lead to flow unfairness, congestion and bufferbloat [169]. Second, the different between UDP and TCP performance can be explained with the performance of the Linux kernel [170]. Therefore, it is important to consider that various tunnelling protocols will have impact on the performance achievable in the cloud.

In general, using small and micro instances (t2.micro, f1-micro, A0) as NFV hosts, the end-to-end throughput degrades drastically. This can be attributed to the CPU shares allocated to the VMs, especially impacting smaller instances as described by Guohui Wang and Eugene Ng [166]. By using instances with more and higher frequency vCPU cores (e.g. m3.2xlarge on EC2, n1-highcpu-2 on GCE or A8 on Azure), the achievable throughput is close to the UDP baseline (maximum reachable) at all providers. It is also important to mention that the relationship between the size (price) of the instances and the achievable throughput is not always linear. For example, in our experiments over Azure, using a compute-optimised, more expensive D1 instance provided lower throughput than a cheaper A1 instance. Therefore, providers need to be careful when selecting VMs for the vNFs to keep cost down.

Stability of the throughput can also be observed from Figure 5.6, as the error bars show the first and third quartile of the throughput. While GCE delivers the highest performance,

the throughput is highly variable with 46% lower throughput than the median for the first quartile, and 6% higher for the third quartile (n1-standard-1). EC2 and Azure deliver lower but consistently stable throughput over different measurement rounds and instance types with a maximum of 27% (m3.large) and 3% (D1) differences from the median respectively.

5.3.3 Delay Through Various VM types

As the traffic is steered through a GNF vNF hosted on a VM with an unknown physical location and distance from other VMs and the gateway that connects the provider's internal network to the cloud, it is also necessary to evaluate the impact on the average delay that such traffic steering involves internal to the cloud (the provider's connection to the cloud detailed in Section 4.2.2 is excluded from these measurements). To do so, the Round Trip Time (RTT) has been measured through various clouds hosting GNF vNFs. The measurements were conducted using the ping utility, sending 300 ICMP requests at each measurement round, with each measurement round involving the same steps as defined in the previous section. A large number of measurement rounds have been conducted to avoid any particular virtual machine placement. Figure 5.7 presents a scatter plot of the mean throughput and mean RTT for the various instance types we evaluated. It is important to note that while ICMP measurements provide a good indication for average delay, operation traffic, depending on protocols used, can experience different delays. One possible cause of this delay can be introduced by QoS policies, as it is shown for TCP and UDP traffic later. An other possible uncertainty for ICMP traffic is that network devices can forward ICMP packets slower (using "slow path") compared to normal traffic as analysed by Govindan and Paxson [171]. This can introduce delays depending on the number and type of routers between the examined VMs which requires further measurement to identify (not covered in this thesis).

As shown in Figure 5.7, the RTT through EC2's c4.large instance is approx 0.3 ms, while using Azure's A2 instance, the RTT is 2.3 ms, almost 8 times higher. It can be clearly seen that Amazon's network provides the lowest latency for all instances, followed by Google's 1ms average RTT. On Azure, a high RTT has been measured, even for the large, network

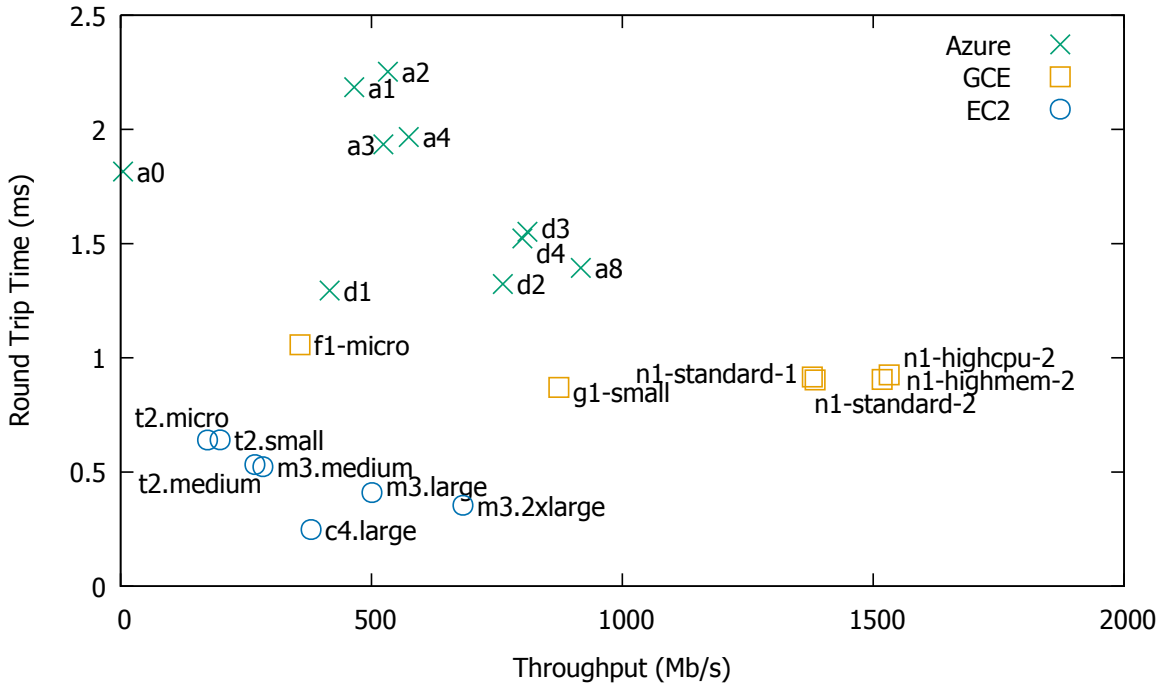


Figure 5.7: Mean throughput and mean RTT of various instance types over Azure, GCE and EC2 [25].

optimised instances (e.g., A8). We can also observe that the delay is usually higher for the cheaper (small and micro) instances for all providers. It is also interesting to observe that the RTT for Azure instances depends highly on their type, while EC2 and especially Google provide a more stable RTT the across various instance types.

On top of network delay properties, Figure 5.7 highlights three distinct clusters of instances that match the three providers. According to this scatter plot, Azure delivers high RTT and average throughput, EC2 provides low RTT and average throughput, while GCE gives high throughput and mid-range RTT for GNF. This difference in latency could simply imply that the network in some providers is slower due to a slower switching fabric, larger buffers or higher over-subscription. However, it could also hinder details on the cloud's placement algorithm, with Amazon EC2 bringing the VMs closer together than GCE or Azure and consequently reducing latency [25]. These subtle differences impact GNF's performance, when deployed in the cloud.

5.4 Dynamic, Latency-Optimal vNF Placement

GNF's latency-optimal, dynamic vNF placement orchestration outlined in Section 4.4 has been evaluated with extensive simulations. To show the properties of such a system, three experiments were designed. After introducing the simulation environment in Section 5.4.1, the latency benefits of running vNFs at the network edge instead of running vNFs at distant clouds are discussed in Section 5.4.2. The second experiment presents how continuously-changing latency causes the system to deviate from a once optimal placement and how it results in latency violations experienced by end users, presented in Section 5.4.3. Finally, in Section 5.4.4, the dynamics and adaptive behaviour of the proposed, optimal stopping-based placement scheduler are shown by comparing it to three alternative schedulers, one based on the periodic re-computation of the placement (1), one based on the re-computation of placement based on an estimated probability for latency violations (2), and one based on the re-calculation of the placement every time instance (3).

5.4.1 Evaluation Environment

This part of GNF has been evaluated using software simulations running on a desktop computer (32GB of RAM, Intel i5). The next paragraphs outline the most important characteristics of the simulated environment.

Network Topology To simulate a realistic network topology of a large footprint network provider, the Jisc nation-wide NREN backbone network[3] was used, as reported by the Topology-Zoo[4] website. To introduce edge resources, compute capacity (i.e., a physical server) has been assumed at all of the points of presence of the Jisc network topology, each added server being capable of running a limited number of vNFs. This deployment scenario of adding compute servers next to already existing network devices of a provider is

[3]http://jisc.co.uk Accessed on: 5 April 2018
[4]http://topology-zoo.org Accessed on: 5 April 2018

the suggested deployment by ETSI's Multi-Access Edge Computing (MEC), since it allows low-cost deployment and interoperability with already deployed network devices [3]. Furthermore, three clouds has been added to the Jisc topology (attached to the London, Bristol and Glasgow POPs) to simulate a distant NFV infrastructure that has unlimited capacity for vNFs as opposed to the limited capacity set for the edge.

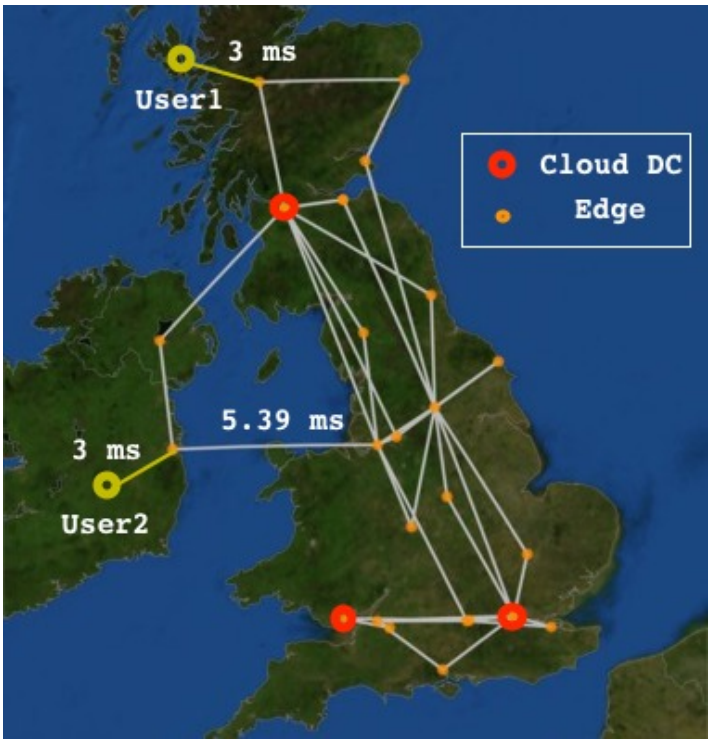


Figure 5.8: Jisc Backbone topology used for our experiments. In this figure, two example users connected at two locations with 3 ms latency to the closest edge are shown [124].

Latency Tolerance of vNFs Latency (round-trip) tolerance of each vNF is an important parameter of the placement algorithm. As different vNFs have different latency requirements, vNFs were split into three categories based on their latency tolerance, and assign a θ_i tolerance to each vNF n_i, representing the maximum latency between the vNF and the end-user beyond which the vNF becomes unusable. As shown in Table 5.1, some real-time functions, such as, inline packet processing (access points, virtual routers and switches, deep packet inspection) cannot afford more than 10 ms from their user. Some "near real-time" vNFs, however, can accommodate up to 30 ms delay. Examples of the latter include control plane functions, e.g., analytics solutions and location-based services. As a third category,

non real-time functions (e.g., OSS, management functions) were introduced that have high delay tolerance of 100 ms. In our experiments, an equally distributed mix of real-time, near real-time and non real-time vNFs were used, however, our claims below can be generalised for any type of vNFs.

Table 5.1: Latency tolerance of different vNF types [30].

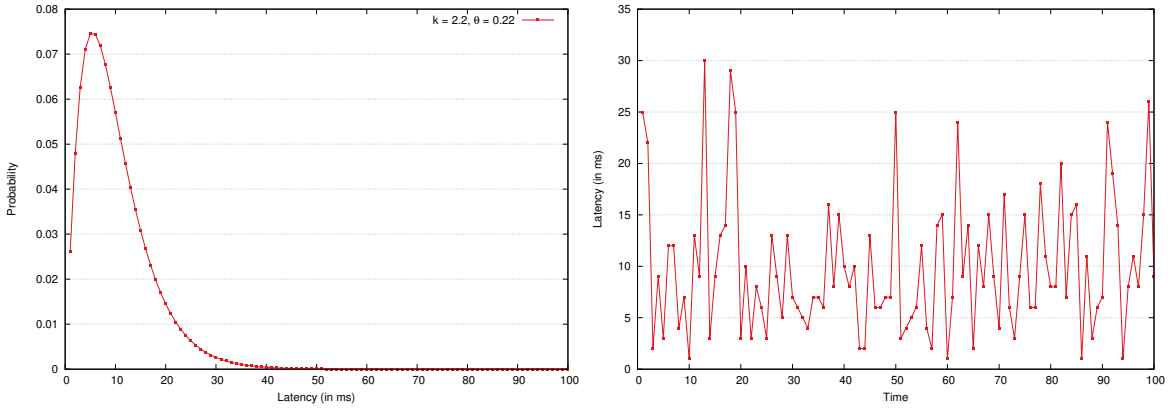
Type of vNF	**Maximum delay**
Real-time (e.g., packet processing functions)	10 ms
Near real-time (e.g., control plane functions)	30 ms
Non real-time (e.g., management functions)	100 ms

Latency Modelling between Locations Latency has been modelled based on real-world round-trip latency measurements collected from New Zealand's research and education wide-area network provider, REANNZ[5] using Ruru [172] a toolkit developed and deployed to collect end-to-end latency measurements from real world network providers. Based on the collected empirical data, R's liftLRD package[6] was used to find that the latency observed between locations in wide area has marginal long memory, giving a Hurst exponent of 0.6 on average. As a result, latency between locations (e.g., points of presences) has been modelled using Gamma distributions (k = 2.2, θ = 0.22), as shown in Figure 5.9a. Gamma distribution has been fitted due to its asymmetric, long-tail property that represents well the increased latency caused by congestion and routing issues in wide area networks [173, 174]. In order to obtain latency values for each link, we sampled this distribution to create a representative time series as shown Figure 5.9b.

5.4.2 Optimal Static Placement

As the optimisation problem was formulated using ILP, it has been implemented using the Gurobi[7] solver as described in Section 4.4. The solver calculates the optimal placement (and

[5]http://reannz.co.nz
[6]https://cran.r-project.org/web/packages/liftLRD/liftLRD.pdf Accessed on: 4 April 2018
[7]https://www.gurobi.com



(a) Probability distribution function of latency (b) Latency time-series

Figure 5.9: Statistical latency characteristics of a physical link.

cumulative user-to-vNF E2E latency as an objective function) for the problem detailed in Section 3.5.1. To show the benefits of running vNFs at the network edge, two deployment scenarios have been compared:

1. Cloud-only deployment: vNFs are only allocated to a set of cloud VMs (three locations in our case).

2. Two-tier edge deployment: in addition to the clouds, all points of presence of the backbone network have equal, but finite amount of computing capabilities to host vNFs. When edge devices run out of resources (as most vNFs get allocated on the edge to minimise user-to-vNF latency), vNFs are allocated to the clouds that are further away, hence incurring higher latency from the users.

In this experiment, end users were assigned to edge locations in a round robin fashion and set a fixed user-to-edge latency of 3 ms, a fair estimation for an average last hop latency based on the studies in [175]. In fact, this last hop latency is the minimum for each user to access a vNF and it only stands until the vNF is hosted at the user's edge device. For this experiment, we have assumed 3 vNFs per user and assigned computing capabilities to all edge nodes with a total capacity of 1000 vNFs, ca. 40 vNFs per edge node, a realistic estimation based on the evaluation of containers shown in the previous Section 5.2.

In Figure 5.10, we show the average latency from users to their vNFs. The cloud-only

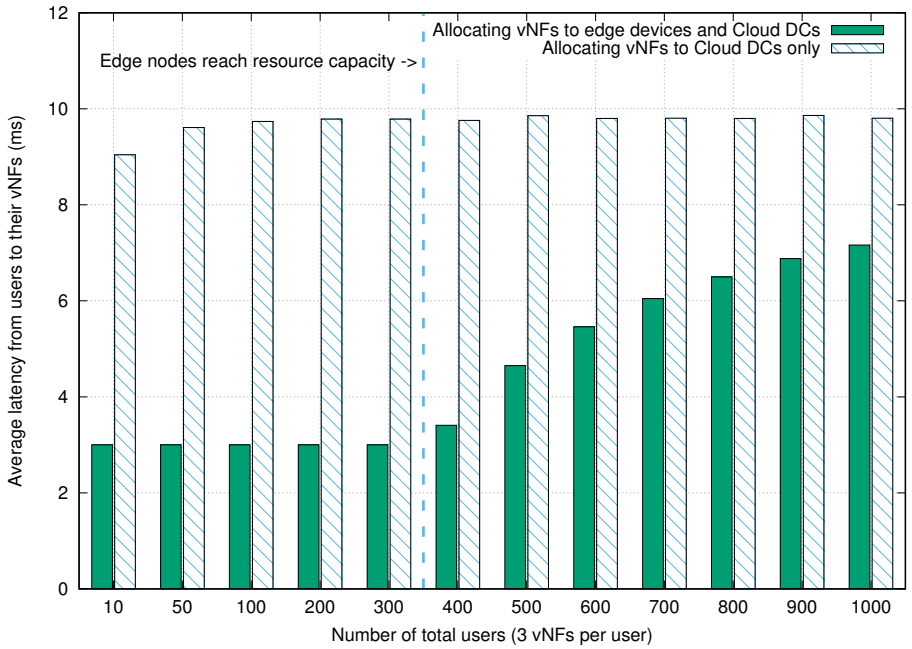


Figure 5.10: Comparing the average vNF-to-user E2E latency between edge and cloud deployments [124].

deployment gives an average latency of 10 ms between users and their vNFs, while running vNFs at the edge results in a 3 ms average latency until the edge nodes reach the limits of their capacity and vNFs. When edge nodes run out of resources (as shown at around 350 simulated users in our setup), the average vNF-to-user E2E latency starts converging to that of the cloud-only scenario since, from this point onward, vNFs are being allocated to the cloud servers, resulting in longer communication paths. Therefore, placing vNFs at the network edge can result in significantly lower (up to 70% decrease in our setup up to 300 users with 3 vNFs each) vNF-to-user E2E latency.

5.4.2.1 Scalability of the ILP

The ILP problem presented in this thesis can take a lot of time to solve, even on up-to-date hardware. The complexity depends on the following parameters:

1. complexity of the network: here, the number of links increase the number of possible path to be examined when selecting one between a user and the host that has their vNF.

It is important to mention that even one link can add thousands of new possible paths in the network.

2. number of users: with each user in the network, the complexity increases.

3. number of vNFs: with each vNF added to the user, the complexity increases.

The above presented network with only 2 users and 5 vNFs results in a model with 8410 binary variables. This grows to 15138 variables with 3 users and 8 variables. While the solving time is 0.1 second for both of these cases on a recent Apple MacBook Pro with 3.1 GHz Intel Core i7 and 16GB of memory, the exponential increase in the number of variables generates over 20 million variables for 20 users and 100 vNFs using JANET's topology. Solving this matrix with 20 million variables on a consumer laptop takes around an hour, not including the time it takes to set-up a model (which is bound to the single-core performance of Python, resulting in around 35 minutes of setup time). Solving a scenario with 1000 users and 3 vNFs per users (total number of 3000 vNFs) using JANET's topology takes around 15 hours of computational time with the model presented in Section 3.5.1.

The scale of the presented ILP model can be extended in different ways to match the requirements of a nation-wide network operator with multiple thousands of vNFs. First, simplifying the network topology (by for example aggregating links for the ILP model or having fewer number of edge hosts) can greatly decrease the number of path that have to be examined between users and potential hosts for vNFs, allowing much fewer number of variables to be created. However, aggregating links would mean that some heuristics decisions need to be made for the actual path allocation and therefore the vNF allocation won't be optimal from the global network's perspective.

Decreasing of the number of decision variables can also be done by restricting possible vNF-to-host allocations that are geographically further away from the user that the physically minimum latency. This means that for example a vNF that has to be allocated in 5 ms from a user in London won't get evaluated by the solver for a host in Glasgow, given that it is minimum 13 ms between London and Glasgow to to propagation delays on optical fiber.

Such restriction of network routes would allow handling 10000 users with 500 edge nodes given that only 3 routes are available between users and the edge nodes (this is realistic given that a close edge node should be only accessible using a handful of physical paths, for instance being a metro node to home customers) using only 15 million decision variables, taking about an hour to set-up (adding variables and constraints to the model) and solve.

Another approach could be the relaxation of the model. This would essentially mean removing the integrality constraint of each variable, resulting in a linear program instead of an integer linear program. This relaxation technique transforms our NP-hard optimisation problem (integer programming) into a related problem that is solvable in polynomial time (linear programming). While the solution from relaxed linear program does not give an exact vNF allocation, it can be used to gain information about the solution to the original integer program.

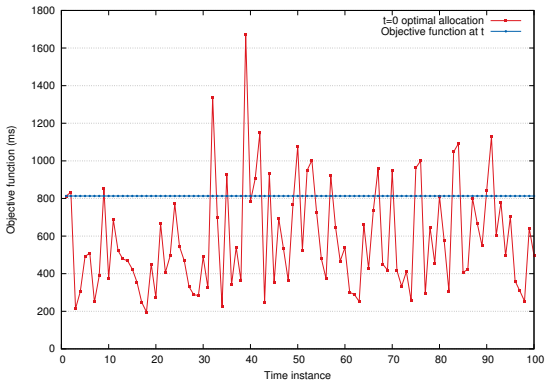
Alternatively, to increase performance, the solver can be executed on better machines (or a cluster of machines) with larger memory providing faster processing than consumer notebooks used in this thesis. Also, while Gurobi was used in this thesis, other ILP solvers can also be examined (e.g., CPLEX, GLPK) to reach the scale of larger operators [176].

5.4.3 Analysing Latency Violations

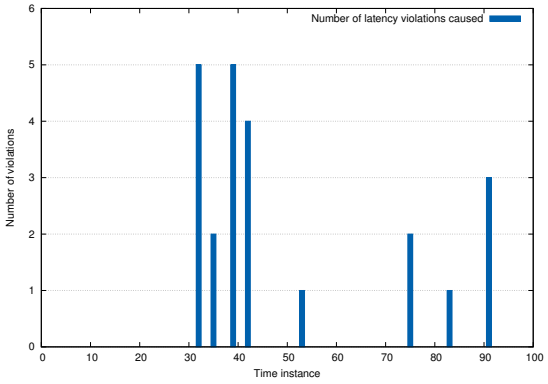
Since the latency matrix l changes over time (due to changes in link latency and changes in users' physical location in the network), the system deviates from a previously optimal placement over time. This deviation from optimal is shown in Figure 5.11a, where the difference between an old value of the objective function (Eq. 3.2) and the temporal value of this objective function calculated at every time instance t (a time instance can be e.g., 5 minutes on a production deployment) is highlighted. As shown, successive objective values can deviate between -25% an 200% from each other, depending on the initial placement and the network topology.

These deviations from a previously optimal allocation can result in latency violations which note that a vNF n_i experiences higher vNF-to-user E2E latency than the θ_i threshold used

for the previous (optimal) placement, as it is shown in Figure 5.11b. As it can be seen by looking at Figure 5.11a and Figure 5.11b at the same time, while deviations from an optimal placement happen at any time, not all deviations are responsible for latency violations (since a particular deviation can be distributed between all users and vNFs, not violating any particular vNF-to-user E2E latency requirement). It is important to note that the number of violations counted at different time instances is the key for calculating the optimal time for migration, since the cumulative distribution and mass functions of the latency violations is used in Eq. 3.14 as $P(L = \ell)$ to predict the number of latency violations the system will experience at the next time instance. These properties are unique for the network topology, vNF locations and the physical locations of the users. We envision learning $P(L = \ell)$ for a period of time (e.g., 300 time instances, representing an entire day in case 5 minutes is selected for a time instance by the operator) before kicking off the placement scheduler. While in operation, we continuously update $P(L = \ell)$ based on the experienced violations.



(a) Deviation from optimal allocation per time instance.



(b) Number of latency violations per time instance.

Figure 5.11: Deviation from the optimal placement and the number of violations per time instance.

5.4.4 Placement Scheduling Using Optimal Stopping Time

5.4.4.1 Time Instance

The experiments below use the term ‘time instance‘. Time instance reflects the time duration between two checks of the optimal stopping condition. This time can be selected by

the operator and it should reflect the actual network properties, such as network topology, variability of the traffic and expected behaviour of the users. In backbone networks with relatively stable traffic a longer time instance should be selected compared to an access network with lot of user movements (connects and disconnects), where traffic changes frequently. The selected time instance should also align with the anticipated trends in a network, such as daily fluctuations. It is also possible to adjust the time instances while the system is running: a short time instance can be selected while users commute from home to work (and back) and a longer one can be selected while they are stationary at their work.

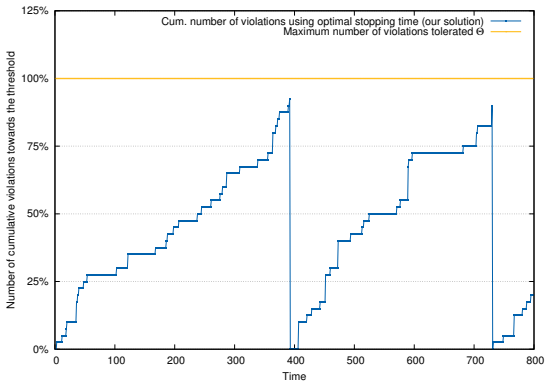
For our simulated nation-wide operator in this thesis, a 5 minute time instance has been selected, since it is short enough to reflect latency changes in the network, but long enough to complete a round of optimisation and migrations in case it is triggered.

For large-scale, nation-wide deployments with thousands of users and hundreds of edge nodes, a vNF allocation is anticipated to run at least an hour (an optimised scenario that can finish in an hour is described in Section 5.4.2.1), therefore the time instance should be set for at least an hour.

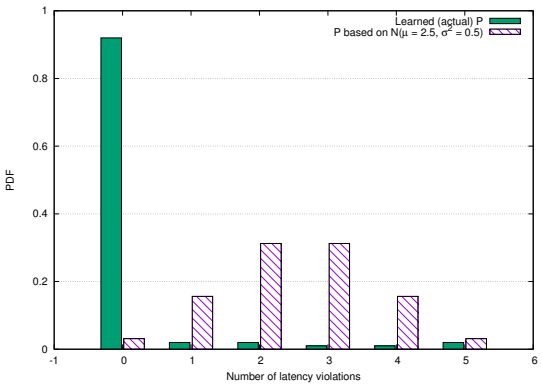
5.4.4.2 Basic Behaviour

For this experiment, we run the system for 800 time instances, representing 2 days and 18 hours of operations that is enough to show present multiple triggers for placement re-optimisation. We present the behaviour of our placement scheduling solution described in Section 3.5.2. As shown in Figure 5.12a, the system experiences accumulated latency violations towards the latency tolerance threshold Θ set by the operator for the system (or Θ can be set for a subset of users to differentiate QoS received by different groups of users). After some time, just before reaching this threshold, the system triggers the re-calculation of the placement and performs vNF migrations in order to reach the new latency-optimal location while zeroing the accumulation of latency. As it is also shown in Figure 5.12a, the scheduling algorithm is fully adaptive to the number of latency violations caused by latency deviations. As an example, the second migration happens much later from the optimal point

than the first one.



(a) Our scheduler optimises for a new placement avoiding reaching the violation threshold.



(b) Distribution of the number of latency violations per time.

Figure 5.12: Our proposed scheduler triggers the optimisation just before reaching a latency violation threshold [30].

It is also important to note that Eq 3.16 relies on $F_L(\ell) = \sum_{l=0}^{\ell} P(L = l)$ and $P(L = \ell)$ that are the cumulative distribution and mass functions of L in (3.11), respectively. In order to learn $P(L = \ell)$, the system was left running for 300 time instances (one day, in order to learn daily latency dynamics) before scheduling the first re-allocation of the placement. This is required, since the optimal stopping time is calculated based on the previously-learned distribution of latency violations per time instance $P(L = \ell)$. It is important to note that the properties of $P(L = \ell)$ determine when to trigger a migration and it can be in various stages of the system.

5.4.4.3 Comparison With Other Placement Schedulers

In Figure 5.13 and Figure 5.14, the trade-off between the number of migrations and the number of latency violations are analysed. The following four placement scheduling strategies were evaluated:

- scheduling every time instance
- scheduling at optimal stopping time with variable, learned probability of latency violations (GNF scheduler)

- scheduling at optimal stopping time with the probability of latency violations fixed
- scheduling at periodic time intervals

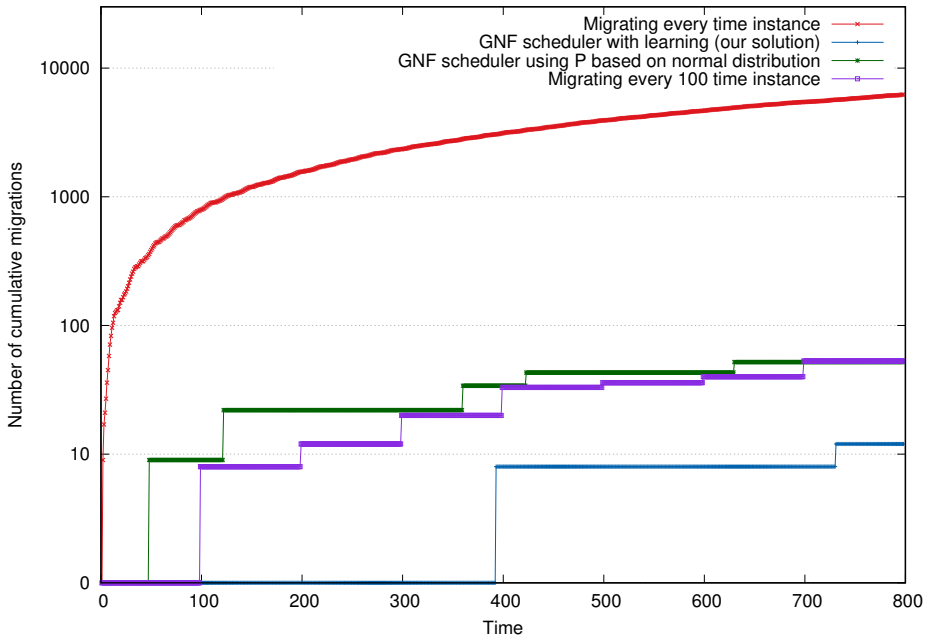


Figure 5.13: Comparison of the number of migrations with various migration scheduling strategies [30].

Scheduling at every time instance Scheduling every time instance means that the allocation of vNFs is re-computed and re-arranged at every time instance (e.g., every 5 minutes), based on the temporal latency parameters of the network. As shown in Figure 5.13, migrating vNFs at every time instance results in a 2-3 orders of magnitude higher number of migrations, since every time instance a new placement is calculated and therefore vNF migrations are conducted (vNFs are moved to new devices or new network routes are selected between each user and their vNFs). As expected, this results in zero latency violations at every time instance since vNFs are always at their optimal location, as shown in Figure 5.14.

Scheduling at optimal stopping time using learned $P(L = \ell)$ This approach, which has been adopted in this work, minimises the number of migrations while it guarantees that the system never reaches the maximum number of allowed violations Θ. This solution

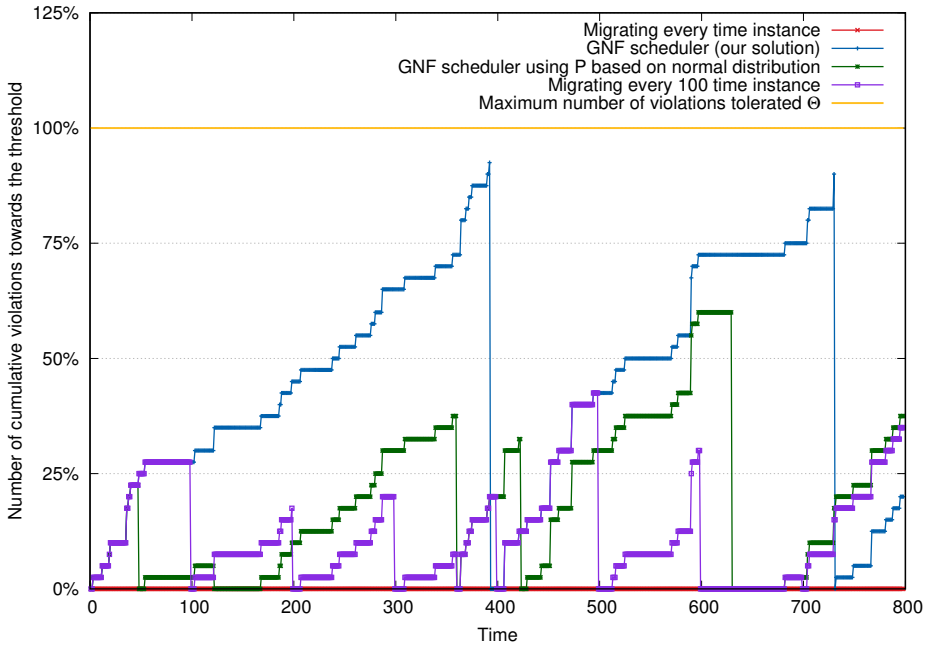


Figure 5.14: Comparing of the number of cumulative violations with various scheduling strategies [30].

depends on the learned distribution of latency violations at every time instance which is gathered by leaving the system to run for 300 time instances (learning phase) before scheduling any re-optimisation of the placement. As shown in Figure 5.13, our strategy re-computes the placement two times and performs only 12 vNF migrations while, as shown in Figure 5.14, it remained under the limit, reaching 87% of the latency violations threshold of the system. From the Figures it can also observed that our scheduler can eliminate at least 76.9% and up to 94.8% of the vNF migrations compared to other schedulers.

Scheduling a new placement using $P(L = \ell)$ following the Normal distribution The distribution of experienced latency violations is a key parameter in our model, as shown in Eq. 3.16. As presented before, in order to get the best result, a long learning phase is advised for the system, where the $P(L = \ell)$ distribution of the latency violations is learned. In this experiment, however, instead of leaving the system to learn $P(L = \ell)$ based on observations, it is assumed that the latency violations in the system follow a Normal distribution $\mathcal{N}(2.5, 0.5)$. The difference between the learned and Normal distributions is shown in Figure 5.12b. While assuming a Normal distribution for $P(L = \ell)$ eliminates

the need for learning, it initiates a new placement much sooner than our scheduler using the learned distribution of latency violations as shown in Figures 5.13 and 5.14. In our case, re-calculation of the placement is scheduled sooner as the probabilities of getting a high number of latency violations at a time instance are higher compared to the learned distribution. While this results in a sub-optimal solution, an operator could start with such approximated distribution for $P(L = \ell)$ and adapt the distribution of latency violations over time to the actual experienced latency violations.

Scheduling a new placement at fixed periodic intervals This strategy initiates vNF migrations at set time intervals, independent of the number of latency violations experienced before. This is the most basic scheduling strategy a network operator could implement by, e.g., initiating a network configuration batch-job re-allocating vNFs at the same time every night. Looking at the number of migrations in Figure 5.13, this strategy results in a moderate number of migrations compared to other strategies. Also, as shown in Figure 5.14, if the selected time interval is short enough (e.g., every 100 time instances), the cumulative number of latency violations never reaches the threshold. However, it is important that this strategy does not depend on $P(L = \ell)$, the latency violations experienced and therefore does not adhere to the latency violation threshold Θ, meaning that a long interval selected (e.g., every 500 time instances in this case) would result in violating the Θ threshold. In fact, GNF tackles the problem of systematically selecting the right interval to re-compute the placement without manual and constant tuning required from the operator.

5.5 Summary

This chapter has presented the most important characteristics of the GNF platform with a mixed set of use-cases, testbed and cloud measurements as well as simulation results.

First, Section 5.2 presented the benefits of container vNFs comparing then against VMs and unikernels. As shown, 50 container vNFs were created and started in just over 10 seconds, while it takes more than 40 seconds just to create the same amount of XEN VMs that have

not even booted up. As for memory requirements, it has been shown that containers only consume around 2.21MB per container, about half the amount required by ClickOS and significantly less than VMs that require hundreds of MBs. Throughput of containers has also been examined and compared with ClickOS. As shown, GNF containers provide better throughput and scaling properties compared to ClickOS, however, this only holds with kernel-based vNFs, where packets are not copied (e.g., firewall, rate limiter vNFs). Finally, the encapsulation overhead of containers has been analysed, showing only a slight penalty (maximum 3.4% degradation) on throughput. The results show that container-based vNFs can be started and stopped in a matter of seconds and provide high performance forwarding while only incurring a minimal performance penalty compared to non-virtualised, native vNFs.

Section 5.3 presented the evaluation of GNF running vNFs in various public clouds, using VMs as hosting devices. Here, the differences in performance between providers and their instance types are shown, comparing throughput, delay, and the hourly price of these underlying instances. This section showed that GNF operators need to carefully select between cloud providers and their VM types, since for example the delay introduced by some VMs varies by the factor of 8.

Finally, Section 5.4 has presented the benefits of running vNFs at the edge of the providers' network using a nation-wide simulated topology. Results have shown that the proposed optimal placement solution for GNF can allocate vNFs to the network edge, resulting in significantly lower (up to 70% decrease in our conditions) vNF-to-user E2E latency. Furthermore, this section has evaluated the dynamic properties of the vNF orchestration and showed that each scheduler shown has a distinct behaviour. While scheduling the re-optimisation of the placement every time instance results in zero latency violations, it requires many vNF migrations, performed at every time instance. On the contrary, migrating periodically results in fewer migrations but it requires a constant tuning from an operator to select the right time for re-optimisation of the placement. This solution, based on optimal stopping theory selects the right time for placement just by looking at the previously experiences latency violations

noted in $P(L = \ell)$. In order to show the importance of $P(L = \ell)$, a scheduler that estimates the latency distribution without learning has also been presented. The results show that our dynamic vNF placement scheduler reduces the number of migrations by 94.8% and 76.9% compared to a scheduler that runs every time instance and one that would periodically trigger vNF migrations to a new optimal placement, respectively.

Chapter 6

Conclusion and Future Work

6.1 Overview

This final chapter of this thesis outlines the contributions of this research in Section 6.2. It revisits the thesis statement in Section 6.3, outlines use-cases for GNF in Section 6.4 and identifies possible future extensions to this work in Section 6.5. Finally, the thesis is concluded with some remarks in Section 6.6.

6.2 Contributions

This work has applied the latest technologies and principles of network programmability, including NFV and SDN, along with dynamic optimisation algorithms to create GNF, a NFV framework that brings latency-optimal vNFs to the network edge. GNF tackles some of the aims of future networks that require low-latency and context-aware service offering, personalised service delivery, and flexible and efficient resource management. The main contributions of GNF are summarised in the following paragraphs.

First, GNF is the first container-based NFV framework that provides transparent assignment of lightweight network services to users and enables running vNFs on a large number of

underlying hosting devices [24]. GNF has been used to demonstrate the benefits of running vNFs at edge of the network [27, 6], as well as public clouds [25]. The contributions range from open-source implementations of popular, real-world container vNFs in containers (shown in Section 4.6) to implementation of NFV software including lightweight agents, vNF managers, state of the art web user interfaces (shown in Section 4.5 and 4.3) and SDN controller modules that achieve various transparent forwarding policies using OpenFlow (described in Section 4.5.2.1), backed up with extensive evaluation presented in Section 5.2 and Section 5.3.

Second, the GNF's latency-optimal placement solution [30, 124] is the first vNF orchestrator specifically tailored to minimise end-to-end latency between vNFs and their users, doing it in a dynamic way by carefully analysing the trade-off between latency violations allowed and the number of vNF migrations required by the operator. These contributions of the vNF orchestration include mathematical models and solutions building on ILP [124] and Optimal Stopping Theory [30] (Section 3.5), and extensive simulation results using real-world latency characteristics over a nationwide network topology, presented in Section 5.4.

This thesis is one of the early works that advocates a new, visionary deployment scenario for NFV by uniting it with today's edge-based architectures, such as MEC and fog computing [6]. While NFV has been traditionally presented in the core of the network, this thesis outlines the benefits and challenges of running similar, yet lightweight vNFs in close physical proximity of the users. Moving intelligence from core to the edge is a popular trend today in computer networking research and the industry, where GNF is an early architecture backed up with proof of concept demonstrations and use-cases, open-source software components and scientific analysis of various components [27, 6].

6.3 Thesis Statement Revisited

In this section, the thesis statement is repeated from Section 1.2, while the remainder of this section indicates how it has been addressed. The thesis statement is restated as follows:

> This work asserts that by creating, moving, and dynamically managing lightweight virtual network functions in close physical proximity to the end users (for example at the edge of the network) will enable Telecommunication Service Providers to deliver low-latency, QoS-guaranteed network services with low impact on the provider's network. This hypothesis has been be tested by the development and evaluation of a container-based network function framework, applying NFV and SDN technologies.

This thesis has started by describing the need for network virtualisation and introducing important underlying principles such as SDN and NFV. Then, it motivated the aims of future networks providing low-latency, context-aware, and personalised network services for users, and showed the limitations of current research and industry solutions.

To be able to host services in close proximity of the end users (for example at the network edge), the concept of lightweight, containerised network functions has been evaluated. As shown, container-based network functions can be created easily and hosted on various, even low-cost devices without virtualisation support (e.g., home routers) or on public cloud VMs, and allow to be spawn up and torn down in a matter of seconds instead of minutes. After understanding some of the security and isolation concerns of the technology, containers are a good fit for the proposed highly dynamic, centrally managed infrastructure. Multiple, real-world vNF examples have also been presented in this thesis.

Finding the balance between latency violations prevented by the operator (representing QoS violations for users), and the cost of re-computing placement and performing vNF migrations (causing interruptions in vNFs) is crucial when the temporal properties (e.g., user's locations and latency fluctuations) of the network change frequently. This trade-off has been analysed with a solution provided using fundamentals of Integer Linear Programming for optimal placement and Optimal Stopping Theory for dynamic, latency-optimal vNF orchestration.

This thesis has shown that it is possible to create, run and manage vNFs at the network edge to provide low-latency services by using lightweight, container vNFs, and a carefully designed dynamic orchestration algorithm.

6.4 GNF Use-Cases

In this section, we present three visionary use cases that highlight the benefits of GNF egde vNFs when deployed by operators. These use cases show specific scenarios of GNF network functions, including IoT DDoS mitigation, on-demand troubleshooting for providers as well as the capability of roaming network functions [6].

6.4.1 IoT DDoS Mitigation

The Internet of Things (IoT) is a proposed development where everyday objects run software equipped with network connectivity that allows them to send and receive data. Recently, there has been an increase of such devices in billions of households and in many "smart cities" installations. Example devices of this architecture include security cameras, light bulbs, smart TVs, weather sensors [177].

Recently, a historically large number of distributed denial-of-service (DDoS) attacks have been witnessed, exploiting vulnerabilities of insecure routers, IP cameras, digital video recorders and other unprotected devices [75]. An example malware family, dubbed "Mirai", spread to vulnerable devices by continuously scanning the Internet for IoT systems protected by factory default or hard-coded usernames and passwords. In one of the recent attacks, an Akamai-hosted website peaked at an unprecedented 665 Gbps (and 143m pps), and resulted in the website taken offline due to the financial implications of the extensive network utilisation[1]. Another similar, unseen IoT DDoS attack has targeted a specific European bank in January 2018, followed by other attacks in the financial sector world-wide[2].

This thesis advocates that such distributed attacks from IoT devices can be efficiently mitigated by providers with a distributed NFV platform like GNF that utilises the network edge as the first point of controlled entry to the provider's network [178]. As a GNF vNF can be deployed on generic home or IoT gateways (also called as "capillary gateways"), malicious traffic can be blocked in a matter of seconds by creating a new iptables-based firewall GNF

[1] https://blogs.akamai.com/2016/10/620-gbps-attack-post-mortem.html Accessed on: 5 March 2017
[2] https://www.recordedfuture.com/mirai-botnet-iot/ Accessed on: 9 April 2018

vNF (outlined in Section 4.6.2) and setting up DROP rules on the selected traffic. Blocking traffic at the customer edge is not only easy even after the attack is launched, but it also avoids unnecessary core network utilisation that costs millions of dollars to serve, according to Akamai. Moreover, edge vNFs could reduce the complexity of securing new applications and devices in the future by automating proactive security configuration in-the-network.

6.4.2 On-Demand Measurement and Troubleshooting

Current telecommunication networks face the difficult task of maintaining an increasingly complex network and at the same time introducing new technologies and services while keeping expenditure low. According to recent studies, configuration of network access control is one of the most complex and error-prone network management tasks that are hard to identify (unless a user complains) and require highly skilled engineers to fix [179]. As these misconfiguration become the main source of network unreachability and vulnerability, providers seek ways to perform customer-centric and automated network troubleshooting.

Through using a platform like GNF, operators can install small vNFs at different points in the network (e.g., customer edge or vNF servers at the core) that perform basic troubleshooting actions using simple tools like *ping*, *traceroute* or *tcpdump*, that are lightweight and are readily available in a Linux distribution (for instance using the network monitoring vNF described in Section 4.6.6). While performing similar actions today takes long manual setup effort and engineer involvement, GNF can allow collecting troubleshooting data (alarms, routing tables, configuration files, etc.) from multiple points in the network in an automated or on-demand fashion. This could reduce operational expenses and result in a faster problem identification and mitigation.

6.4.3 Roaming Network Functions

5G cellular systems are expected to deploy and overlap different types of cells, such as small or spot cells that utilise high frequency (5 GHz or above) to support high capacity

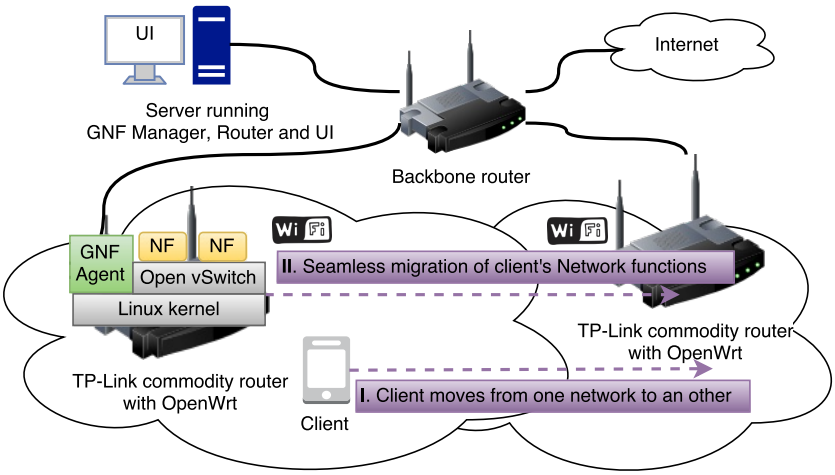


Figure 6.1: vNF migration between access points [27].

transmission with limited spectrum sharing. While these small cells offer high performance, they increase roaming between cells. Hence, to efficiently support users with customised network services, this thesis advocates that network services should also migrate between cells, following the UE [27], as outlined in Figure 6.1.

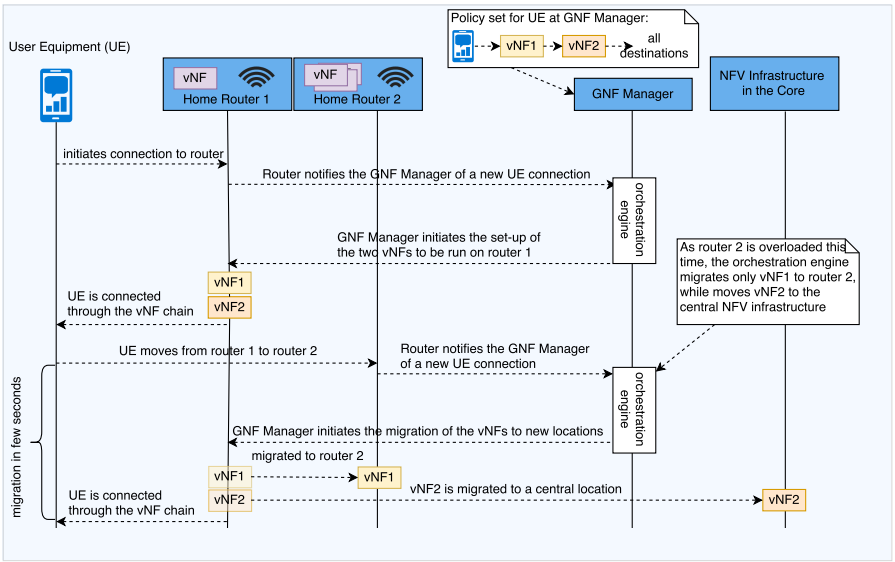


Figure 6.2: vNF migration timeline [6].

As shown in Figure 6.2, GNF allows vNF chains to be associated with a particular UE. In the presented scenario, a simple chain with 2 vNFs is assigned to any traffic leaving the UE. As shown, once the UE is connected, these services can be co-located to the nearest edge device,

called Home Router 1. In case of a movement between cells, the GNF Manager recomputes the allocation of vNFs and initiates migration to achieve optimal placement again. In our example case, one vNF is placed at the edge device closer to the user, while one vNF has been migrated to a central NFV infrastructure. This migration scenario has been demonstrated in [27] with rate limiters, firewalls, and parental filter vNFs and two TP-Link routers.

6.5 Future Work

This thesis highlighted the potential of running and orchestrating lightweight vNFs on various underlying devices in order to support emerging, latency-sensitive network services. The following points provide three possible future extensions.

6.5.1 State Management of vNFs

In GNF, vNFs are treated as stateless, meaning that our examples vNFs lose their internal state after migration. This means that vNFs that inherently rely on state collected over time (e.g., number of SYN packets seen in a specific TCP flow in order to detect SYN flood attacks) will be reset once migrated. There has been considerable research on this topic and in high level there are two directions taken by the community today:

- vNF state can be decoupled and managed separately: this direction is interesting, as it decouples the data from network functions the same way as current cloud compute services do. It means that the state of vNFs are managed in separate storage services that have high-speed connectivity with the vNFs themselves. This has been explored by Kablan et al. in their paper "Stateless Network Functions" [134] where they suggest an in-memory storage connected to the vNFs to store state.

- vNF state should be migrated with a well defined API: this is what Gember et al. proposed with OpenNF [120]. However, while this allows an elegant way to migrate

states, it requires modification to existing vNFs and does not handle vNF failures.

As of today, none of these directions have been explored within GNF, although both would be interesting to examine in the future and to see which would be more applicable when vNFs are running on the network edge. Decoupled state can also be managed in a distributed fashion on the hosting devices that run a particular vNF most often (e.g., we could envision someone has a firewall migrating frequently between their home router and their office router daily and therefore have all the state distributed between these two devices).

6.5.2 Operator-Scale Deployment of Edge vNFs

As outlined in Section 6.4, many use-cases can be envisioned for edge NFV using GNF. If a network operator wishes to deploy GNF over a large-scale testbed, they could do so right now. Such deployment will undoubtedly provide scope for further technical enhancements and future directions in multiple aspects of the system. For instance, an operator-scale deployment with thousands of vNFs could drive scaling the orchestration logic, exploring e.g., a sub-optimal placement as opposed to the optimal placement shown in Section 3.5.1.

An other interesting future direction is defining different service levels for users. While in GNF all users contribute to a Θ latency violation threshold, providers could differentiate group of premium users with a lower Θ than others which is an important goal of future's 5G network operators [180].

6.5.3 Supporting vNF Chains

GNF can currently assign multiple vNFs to each user (a user connected to a firewall and a cache separately), however, it does not support the assignment of vNF chains (a user connected to a cache that is connected to a firewall) for simplification of the problem. Service Function Chaining (SFC) [181] defines an ordered set of abstract network functions along with ordering constraints that must be applied to packets or flows.

While supporting SFC in GNF would not require substantial changes in the vNFs or the traffic management applied, the latency-optimal orchestration presented in this thesis would have to be altered. Specifically, the optimal placement solution presented in Section 3.5.1 will have to be extended to accommodate chains, which would increase the mathematical complexity of the problem, as more possible placement solutions would have to be evaluated. The complexity of the problem can be further increased with unordered vNF chains as well as with multiple vNF chains that share common vNFs to eliminate duplication of functionality deployed in the network [110, 66].

6.6 Concluding Remarks

The computer networking industry is currently undergoing a paradigm change. With continuous improvements in open-source network softwarisation and virtualisation, most network providers strive to realise the vision of a zero-touch network with clear layers of abstraction and orchestration, from the optical up to the application layer as well as from the edge of their network to the core. As a result of this transformation, it is envisioned that in 10 years, network services will be entirely decoupled from physical hardware, and services will be orchestrated dynamically based on, among others, user requirements, physical location of users, high-level intents, QoS guarantees, and temporal traffic patterns.

This thesis was set out to prove the hypothesis that, through the application of softwarisation and virtualisation, operators can provide lightweight, dynamically orchestrated services to end users. Particular attention was drawn to the edge of the network that is able to support the low-latency requirements of some future network services. By designing, developing, and evaluating GNF, a container-based NFV framework, this work has shown one way of creating, running and dynamically managing vNFs on different underlying devices (including low-cost edge devices and cloud VMs), a small step in the long journey we are taking.

Publications

The work reported in this thesis has led to the following publications:

1. Richard Cziva, Christos Anagnostopoulos, and Dimitrios P Pezaros. "Dynamic, Latency-Optimal vNF Placement at the Network Edge". In: *IEEE Conference on Computer Communications (INFOCOM 2018)*. Honolulu, USA, Apr. 2018, pp. 1–9

2. Richard Cziva, Christopher Lorier, and Dimitrios P Pezaros. "Ruru: High-speed, Flow-level Latency Measurement and Visualization of Live Internet Traffic". In: *Proceedings of the SIGCOMM Posters and Demos*. ACM. 2017, pp. 46–47 - **Second place on ACM SIGCOMM Student Research Competition**

3. Richard Cziva and Dimitrios P. Pezaros. "Container Network Functions: Bringing NFV to the Network Edge". In: *IEEE Communications Magazine* 55.6 (2017), pp. 24–31. ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1601039

4. Richard Cziva and Dimitrios P. Pezaros. "On the Latency Benefits of Edge NFV". in: *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. May 2017, pp. 105–106. DOI: 10.1109/ANCS.2017.23

5. Dimitrios P Pezaros, Richard Cziva, and Simon Jouet. "SDN for Cloud Data Centres". In: *Big-Data and Software Defined Networks*. IET Book Series on Big Data. IET, Mar. 2017. DOI: 10.1049/PBPC015E_ch4

6. Abeer Ali, Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. "SDNFV-based DDoS detection and remediation in multi-tenant, virtualised infrastructures". In: *Guide to Security in SDN and NFV - Challenges, Opportunities, and Applications*. GSSNOA '16. Springer, 2017

7. Richard Cziva, Jerry Sobieski, and Yatish Kumar. "High-Performance Virtualized SDN Switches for Experimental Network Testbeds". In: *Proceedings of the 3rd Workshop on Innovating the Network for Data-Intensive Science*. ACM. 2016, pp. 1–8

8. Richard Cziva, Simon Jouët, David Stapleton, Fung Po Tso, and Dimitrios P Pezaros. "SDN-based virtual machine management for cloud data centers". In: *IEEE Transactions on Network and Service Management* 13.2 (2016), pp. 212–225

9. Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. "Roaming Edge vNFs using Glasgow Network Functions". In: *Proceedings of the ACM SIGCOMM 2016 Conference*. ACM. 2016, pp. 601–602

10. Lin Cui, Richard Cziva, Fung Po Tso, and Dimitrios P Pezaros. "Synergistic policy and virtual machine consolidation in cloud data centers". In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 2016, pp. 1–9

11. Simon Jouet, Richard Cziva, and Dimitrios P Pezaros. "Arbitrary packet matching in openflow". In: *High Performance Switching and Routing (HPSR), 2015 IEEE 16th International Conference on*. IEEE. 2015, pp. 1–6 - **Recipient of the Best Paper Award**

12. Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. "GNFC: Towards network function cloudification". In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE. 2015, pp. 142–148

13. Richard Cziva, Simon Jouet, Kyle JS White, and Dimitrios P Pezaros. "Container-based network function virtualization for software-defined networks". In: *Computers and Communication (ISCC), 2015 IEEE Symposium on*. IEEE. 2015, pp. 415–420

14. Richard Cziva, David Stapleton, Fung Po Tso, and Dimitrios P. Pezaros. "SDN-based Virtual Machine management for Cloud Data Centers". In: *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. Oct. 2014, pp. 388–394 - **Recipient of the Best paper award**

Bibliography

[1] L Ericsson. "More than 50 billion connected devices". In: *White Paper* (2011).

[2] M. Chiosi et al. *Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action*. ETSI White paper. 2012.

[3] Yun Chao Hu et al. *Mobile Edge Computing - a Key Technology Towards 5G*. ETSI White Paper 11. 2015.

[4] Shanhe Yi, Cheng Li, and Qun Li. "A survey of fog computing: concepts, applications and issues". In: *Proceedings of the 2015 Workshop on Mobile Big Data*. ACM. 2015, pp. 37–42.

[5] Gaurav Somani et al. "DDoS attacks in cloud computing: Issues, taxonomy, and future directions". In: *Computer Communications* 107 (2017), pp. 30–48.

[6] Richard Cziva and Dimitrios P. Pezaros. "Container Network Functions: Bringing NFV to the Network Edge". In: *IEEE Communications Magazine* 55.6 (2017), pp. 24–31. ISSN: 0163-6804. DOI: 10.1109/MCOM.2017.1601039.

[7] Goran Peskir and Albert N Shiryaev. *Optimal stopping and free-boundary problems*. Springer Science & Business Media, 2006.

[8] D. L. Tennenhouse et al. "A survey of active network research". In: *IEEE Communications Magazine* 35.1 (Jan. 1997), pp. 80–86. ISSN: 0163-6804. DOI: 10.1109/35.568214.

[9] Jacobus E Van der Merwe et al. "The Tempest-a practical framework for network programmability". In: *IEEE network* 12.3 (1998), pp. 20–28.

[10] Beverly Schwartz et al. “Smart Packets: Applying Active Networks to Network Management”. In: *ACM Trans. Comput. Syst.* 18.1 (Feb. 2000), pp. 67–88. ISSN: 0734-2071. DOI: 10.1145/332799.332893. URL: http://doi.acm.org/10.1145/332799.332893.

[11] L. Yang et al. *Forwarding and Control Element Separation (ForCES) Framework*. RFC 3746 (Informational). Internet Engineering Task Force, Apr. 2004. URL: http://www.ietf.org/rfc/rfc3746.txt.

[12] Matthew Caesar et al. “Design and Implementation of a Routing Control Platform”. In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI’05. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–28. URL: http://dl.acm.org/citation.cfm?id=1251203.1251205.

[13] T. V. Lakshman et al. “The SoftRouter Architecture”. In: *In ACM HOTNETS*. 2004.

[14] Thomas Anderson et al. “Overcoming the Internet impasse through virtualization”. In: *Computer* 38.4 (2005), pp. 34–41.

[15] Martin Casado et al. “Ethane: Taking Control of the Enterprise”. In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’07. Kyoto, Japan: ACM, 2007, pp. 1–12. ISBN: 978-1-59593-713-1. DOI: 10.1145/1282380.1282382. URL: http://doi.acm.org/10.1145/1282380.1282382.

[16] Hong Yan et al. “Tesseract: A 4D Network Control Plane”. In: *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*. NSDI’07. Cambridge, MA: USENIX Association, 2007, pp. 27–27. URL: http://dl.acm.org/citation.cfm?id=1973430.1973457.

[17] Nick McKeown et al. “OpenFlow: Enabling Innovation in Campus Networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[18] Tim Verbelen et al. “Cloudlets: Bringing the cloud to the mobile user”. In: *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM. 2012, pp. 29–36.

[19] András Császár et al. “Unifying cloud and carrier network: Eu fp7 project unify”. In: *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*. IEEE. 2013, pp. 452–457.

[20] Antonio Manzalini and Roberto Saracco. “Software Networks at the Edge: a shift of paradigm”. In: *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE. 2013, pp. 1–6.

[21] 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*. USENIX Association. 2014, pp. 459–473.

[22] J. Soares et al. “Cloud4NFV: A platform for Virtual Network Functions”. In: *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. Oct. 2014, pp. 288–293.

[23] George Xilouris et al. “T-nova: Network functions as-a-service over virtualised infrastructures”. In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE. 2015, pp. 13–14.

[24] Richard Cziva et al. “Container-based network function virtualization for software-defined networks”. In: *Computers and Communication (ISCC), 2015 IEEE Symposium on*. IEEE. 2015, pp. 415–420.

[25] Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. “GNFC: Towards network function cloudification”. In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE. 2015, pp. 142–148.

[26] Alfio Lombardo et al. “An open framework to enable NetFATE (network functions at the edge)”. In: *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE. 2015, pp. 1–6.

[27] Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. "Roaming Edge vNFs using Glasgow Network Functions". In: *Proceedings of the ACM SIGCOMM 2016 Conference*. ACM. 2016, pp. 601–602.

[28] Kok-Kiong Yap et al. "Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM. 2017, pp. 432–445.

[29] Brandon Schlinker et al. "Engineering Egress with Edge Fabric: Steering Oceans of Content to the World". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 418–431. ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098853. URL: http://doi.acm.org/10.1145/3098822.3098853.

[30] Richard Cziva, Christos Anagnostopoulos, and Dimitrios P Pezaros. "Dynamic, Latency-Optimal vNF Placement at the Network Edge". In: *IEEE Conference on Computer Communications (INFOCOM 2018)*. Honolulu, USA, Apr. 2018, pp. 1–9.

[31] David L. Tennenhouse and David J. Wetherall. "Towards an Active Network Architecture". In: *Proceedings of the 2002 DARPA Active Networks Conference and Exposition*. DANCE '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 2–. ISBN: 0-7695-1564-9. URL: http://dl.acm.org/citation.cfm?id=874028.874135.

[32] Nick Feamster, Jennifer Rexford, and Ellen Zegura. "The road to SDN: an intellectual history of programmable networks". In: *ACM SIGCOMM Computer Communication Review* 44.2 (2014), pp. 87–98.

[33] D. Papadimitriou et al. *Generalized MPLS (GMPLS) Protocol Extensions for Multi-Layer and Multi-Region Networks (MLN/MRN)*. RFC 6001. Internet Engineering Task Force, Oct. 2010, pp. 1–24. DOI: {10.17487/RFC6001}. URL: %7Bhttp://www.rfc-editor.org/rfc/rfc6001.txt%7D.

[34] D. Awduche et al. *RSVP-TE: Extensions to RSVP for LSP Tunnels*. RFC 3209. Internet Engineering Task Force, Dec. 2001, pp. 1–61. DOI: {10.17487/RFC3209}. URL: %7Bhttp://www.rfc-editor.org/rfc/rfc3209.txt%7D.

[35] Ken Calvert. "Reflections on network architecture: an active networking perspective". In: *ACM SIGCOMM Computer Communication Review* 36.2 (2006), pp. 27–30.

[36] David J Wetherall, John V Guttag, and David L Tennenhouse. "ANTS: A toolkit for building and dynamically deploying network protocols". In: *Open Architectures and Network Programming, 1998 IEEE*. IEEE. 1998, pp. 117–129.

[37] David L Tennenhouse et al. "A survey of active network research". In: *IEEE communications Magazine* 35.1 (1997), pp. 80–86.

[38] J. Van der Merwe et al. "Dynamic Connectivity Management with an Intelligent Route Service Control Point". In: *Proceedings of the 2006 SIGCOMM Workshop on Internet Network Management*. INM '06. Pisa, Italy: ACM, 2006, pp. 29–34. ISBN: 1-59593-570-3. DOI: 10.1145/1162638.1162643. URL: http://doi.acm.org/10.1145/1162638.1162643.

[39] Matthew Caesar et al. "Design and implementation of a routing control platform". In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association. 2005, pp. 15–28.

[40] Travis Russell, Steve Chapman, and Bernard Onken. *Signaling system 7*. McGraw-Hill, Inc., 1998.

[41] Martin Casado et al. "Ethane: Taking control of the enterprise". In: *ACM SIGCOMM Computer Communication Review*. Vol. 37. 4. ACM. 2007, pp. 1–12.

[42] Martin Casado et al. "SANE: A Protection Architecture for Enterprise Networks." In: *USENIX Security Symposium*. Vol. 49. 2006, pp. 137–151.

[43] Brent Chun et al. "Planetlab: an overlay testbed for broad-coverage services". In: *ACM SIGCOMM Computer Communication Review* 33.3 (2003), pp. 3–12.

[44] Mark Berman et al. “GENI: A federated testbed for innovative network experiments”. In: *Computer Networks* 61 (2014), pp. 5–23.

[45] Anastasius Gavras et al. “Future internet research and experimentation: the FIRE initiative”. In: *ACM SIGCOMM Computer Communication Review* 37.3 (2007), pp. 89–92.

[46] Md Faizul Bari et al. “PolicyCop: An autonomic QoS policy enforcement framework for software defined networks”. In: *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*. IEEE. 2013, pp. 1–7.

[47] Roberto Doriguzzi Corin et al. “Vertigo: Network virtualization and beyond”. In: *Software defined networking (EWSDN), 2012 European Workshop on*. IEEE. 2012, pp. 24–29.

[48] Mohammad Al-Fares et al. “Hedera: Dynamic Flow Scheduling for Data Center Networks”. In: *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*. USENIX Association, 2010.

[49] Zafar Ayyub Qazi et al. “SIMPLE-fying middlebox policy enforcement using SDN”. In: *ACM SIGCOMM computer communication review*. Vol. 43. 4. ACM. 2013, pp. 27–38.

[50] Matthew Broadbent et al. “OpenCache: Leveraging SDN to demonstrate a customisable and configurable cache”. In: *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*. IEEE. 2014, pp. 151–152.

[51] Richard Cziva et al. “SDN-based virtual machine management for cloud data centers”. In: *IEEE Transactions on Network and Service Management* 13.2 (2016), pp. 212–225.

[52] Sushant Jain et al. “B4: Experience with a Globally-deployed Software Defined Wan”. In: *SIGCOMM Compututer Communications Review* 43.4 (Aug. 2013), pp. 3–14. ISSN: 0146-4833. DOI: 10.1145/2534169.2486019. URL: http://doi.acm.org/10.1145/2534169.2486019.

[53] Ben Pfaff et al. “The Design and Implementation of Open vSwitch”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, 2015, p. 117.

[54] Michio Honda et al. “mSwitch: a highly-scalable, modular software switch”. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. ACM. 2015, p. 1.

[55] Muhammad Shahbaz et al. “Pisces: A programmable, protocol-independent software switch”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM. 2016, pp. 525–538.

[56] Jan Medved et al. “Opendaylight: Towards a model-driven sdn controller architecture”. In: *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*. IEEE. 2014, pp. 1–6.

[57] Pankaj Berde et al. “ONOS: towards an open, distributed SDN OS”. In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 1–6.

[58] Josh Bailey and Stephen Stuart. “Faucet: deploying SDN in the enterprise”. In: *Queue* 14.5 (2016), p. 30.

[59] Pat Bosshart et al. “P4: Programming protocol-independent packet processors”. In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[60] Simon Jouet and Dimitrios P Pezaros. “BPFabric: Data Plane Programmability for Software Defined Networks”. In: *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. IEEE Press. 2017, pp. 38–48.

[61] Rashid Mijumbi et al. “Network function virtualization: State-of-the-art and research challenges”. In: *IEEE Communications Surveys & Tutorials* 18.1 (2015), pp. 236–262.

[62] Jerome H Saltzer, David P Reed, and David D Clark. "End-to-end arguments in system design". In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 277–288.

[63] Intel. *Intel DPDK: Data Plane Development Kit*. http://www.dpdk.org.

[64] DR Lopez. "OpenMANO: The dataplane ready open source NFV MANO stack". In: *IETF Meeting Proceedings, Dallas, Texas, USA*. 2015.

[65] J. F. Riera et al. "TeNOR: Steps towards an orchestration platform for multi-PoP NFV deployment". In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. June 2016, pp. 243–250. DOI: 10.1109/NETSOFT.2016.7502419.

[66] Balázs Németh et al. "Customizable real-time service graph mapping algorithm in carrier grade networks". In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE. 2015, pp. 28–30.

[67] Roberto Bonafiglia et al. "Enabling NFV services on resource-constrained CPEs". In: *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*. IEEE. 2016, pp. 83–88.

[68] Eric A Brewer. "Kubernetes and the path to cloud native". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM. 2015, pp. 167–167.

[69] Justine Sherry et al. "Making middleboxes someone else's problem: network processing as a cloud service". In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 13–24.

[70] Kjeld Egevang and Paul Francis. *The IP network address translator (NAT)*. Tech. rep. 1994.

[71] D Brent Chapman, Elizabeth D Zwicky, and Deborah Russell. *Building internet firewalls*. O'Reilly & Associates, Inc., 1995.

[72] Craig H Rowland. *Intrusion detection system*. US Patent 6,405,318. 2002.

[73] Michel K Bowman-Amuah. *Load balancer in environment services patterns*. US Patent 6,578,068. 2003.

[74] ABI Research. *World Enterprise Network and Data Security Markets*. 2010. URL: https://www.abiresearch.com/market-research/product/1006059-world-enterprise-network-and-data-security/.

[75] Abeer Ali et al. "SDNFV-based DDoS detection and remediation in multi-tenant, virtualised infrastructures". In: *Guide to Security in SDN and NFV - Challenges, Opportunities, and Applications*. GSSNOA '16. Springer, 2017.

[76] Enrique Hernandez-Valencia, Steven Izzo, and Beth Polonsky. "How will NFV/SDN transform service provider opex?" In: *IEEE Network* 29.3 (2015), pp. 60–67.

[77] Alcetel Lucent Bell Labs. *G.W.A.T.T. Bell Labs application able to measure the impact of technologies like SDN & NFV on network energy consumption. White paper*. http://gwatt.net/intro/1. 2015.

[78] Rashid Mijumbi. "On the energy efficiency prospects of network function virtualization". In: *arXiv preprint arXiv:1512.00215* (2015).

[79] Tomonobu Niwa et al. "Universal fault detection for NFV using SOM-based clustering". In: *Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific*. IEEE. 2015, pp. 315–320.

[80] Xiaoke Wang et al. "Online vnf scaling in datacenters". In: *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE. 2016, pp. 140–147.

[81] Sunny Dutta, Tarik Taleb, and Adlen Ksentini. "QoE-aware elasticity support in cloud-native 5G systems". In: *Communications (ICC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–6.

[82] Fangxin Wang et al. "Bandwidth guaranteed virtual network function placement and scaling in datacenter networks". In: *Computing and Communications Conference (IPCCC), 2015 IEEE 34th International Performance*. IEEE. 2015, pp. 1–8.

[83] Vern Paxson, Scott Campbell, Jason Lee, et al. *Bro intrusion detection system*. Tech. rep. Lawrence Berkeley National Laboratory, 2006.

[84] Martin Roesch et al. “Snort: Lightweight intrusion detection for networks.” In: *Lisa*. Vol. 99. 1. 1999, pp. 229–238.

[85] J Frahim et al. *Securing the internet of things: a proposed framework*. 2016.

[86] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. “Fog computing: A taxonomy, survey and future directions”. In: *Internet of Everything*. Springer, 2018, pp. 103–130.

[87] Mahadev Satyanarayanan et al. “Edge analytics in the internet of things”. In: *IEEE Pervasive Computing* 14.2 (2015), pp. 24–31.

[88] Xiaohu Ge et al. “5G ultra-dense cellular networks”. In: *IEEE Wireless Communications* 23.1 (2016), pp. 72–79.

[89] Federico Boccardi et al. “Five disruptive technology directions for 5G”. In: *IEEE Communications Magazine* 52.2 (2014), pp. 74–80.

[90] Klaus Witrisal et al. “High-accuracy localization for assisted living: 5G systems will turn multipath channels from foe to friend”. In: *IEEE Signal Processing Magazine* 33.2 (2016), pp. 59–70.

[91] Guanling Chen, David Kotz, et al. *A survey of context-aware mobile computing research*. Tech. rep. Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College, 2000.

[92] Weisong Shi et al. “Edge computing: Vision and challenges”. In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646.

[93] 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*. ACM. 2012, pp. 13–16.

[94] Mahadev Satyanarayanan et al. “The case for vm-based cloudlets in mobile computing”. In: *IEEE pervasive Computing* 8.4 (2009).

[95] Eduardo Cuervo et al. “MAUI: making smartphones last longer with code offload”. In: *Proceedings of the 8th international conference on Mobile systems, applications, and services*. ACM. 2010, pp. 49–62.

[96] Byung-Gon Chun et al. “Clonecloud: elastic execution between mobile device and cloud”. In: *Proceedings of the sixth conference on Computer systems*. ACM. 2011, pp. 301–314.

[97] Aibo Song et al. “Multi-objective virtual machine selection for migrating in virtualized data centers”. In: *Pervasive Computing and the Networked World*. Springer, 2013, pp. 426–438.

[98] J.W. Jiang et al. “Joint VM placement and routing for data center traffic engineering”. In: *INFOCOM, 2012 Proceedings IEEE*. Mar. 2012, pp. 2876–2880. DOI: 10.1109/INFCOM.2012.6195719.

[99] Edoardo Amaldi et al. “On the computational complexity of the virtual network embedding problem”. In: *Electronic Notes in Discrete Mathematics* 52 (2016), pp. 213–220.

[100] David G Luenberger, Yinyu Ye, et al. *Linear and nonlinear programming*. Vol. 2. Springer, 1984.

[101] Gurobi Optimization. “Inc.,“Gurobi optimizer reference manual,” 2014”. In: *URL: http://www.gurobi.com* (2014).

[102] Andrew Makhorin et al. *GLPK (GNU linear programming kit)*. 2008.

[103] Eugene L Lawler and David E Wood. “Branch-and-bound methods: A survey”. In: *Operations research* 14.4 (1966), pp. 699–719.

[104] Hendrik Moens and Filip De Turck. “VNF-P: A model for efficient placement of virtualized network functions”. In: *Network and Service Management (CNSM), 2014 10th International Conference on*. IEEE. 2014, pp. 418–423.

[105] Md Faizul Bari et al. “On orchestrating virtual network functions”. In: *Network and Service Management (CNSM), 2015 11th International Conference on*. IEEE. 2015, pp. 50–56.

[106] Insun Jang et al. “Optimal network resource utilization in service function chaining”. In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*. IEEE. 2016, pp. 11–14.

[107] Abhishek Gupta et al. “On service chaining using virtual network functions in network-enabled cloud systems”. In: *Advanced Networks and Telecommuncations Systems (ANTS), 2015 IEEE International Conference on*. IEEE. 2015, pp. 1–3.

[108] Andreas Baumgartner, Varun S Reddy, and Thomas Bauschert. “Mobile core network virtualization: A model for combined virtual core network function placement and topology optimization”. In: *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE. 2015, pp. 1–9.

[109] Juliver Gil Herrera and Juan Felipe Botero. “Resource allocation in NFV: A comprehensive survey”. In: *IEEE Transactions on Network and Service Management* 13.3 (2016), pp. 518–532.

[110] Michael Till Beck and Juan Felipe Botero. “Coordinated allocation of service function chains”. In: *Global Communications Conference (GLOBECOM), 2015 IEEE*. IEEE. 2015, pp. 1–6.

[111] Roberto Bruschi, Alessandro Carrega, and Franco Davoli. “A game for energy-aware allocation of virtualized network functions”. In: *Journal of Electrical and Computer Engineering* 2016 (2016), p. 2.

[112] Roberto Riggio, Tinku Rasheed, and Rajesh Narayanan. “Virtual network functions orchestration in enterprise WLANs”. In: *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE. 2015, pp. 1220–1225.

[113] Ali Mohammadkhan et al. “Virtual function placement and traffic steering in flexible and dynamic software defined networks”. In: *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on*. IEEE. 2015, pp. 1–6.

[114] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. "Optimization by simulated annealing". In: *science* 220.4598 (1983), pp. 671–680.

[115] John Henry Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[116] Fred Glover. "Future paths for integer programming and links to artificial intelligence". In: *Computers & operations research* 13.5 (1986), pp. 533–549.

[117] Rashid Mijumbi et al. "Design and evaluation of algorithms for mapping and scheduling of virtual network functions". In: *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE. 2015, pp. 1–9.

[118] Wee Lum Tan, Fung Lam, and Wing Cheong Lau. "An empirical study on 3G network capacity and performance". In: *26th IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2007, pp. 1514–1522.

[119] Torbjörn Ekman. "Prediction of mobile radio channels: modeling and design". PhD thesis. Institutionen för materialvetenskap, 2002.

[120] Aaron Gember-Jacobson et al. "OpenNF: Enabling innovation in network function control". In: *ACM SIGCOMM Computer Communication Review* 44.4 (2015), pp. 163–174.

[121] Sanghyeok Kim et al. "VNF-EQ: dynamic placement of virtual network functions for energy efficiency and QoS guarantee in NFV". In: *Cluster Computing* 20.3 (2017), pp. 2107–2117.

[122] Md Faizul Bari et al. "nf. io: A file system abstraction for nfv orchestration". In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE. 2015, pp. 135–141.

[123] Anoop Ghanwani et al. *An Analysis of Lightweight Virtualization Technologies for NFV*. Internet-Draft draft-natarajan-nfvrg-containers-for-nfv-03 (Accessed on: 06/03/2017). Internet Engineering Task Force, July 2016. 16 pp.

[124] Richard Cziva and Dimitrios P. Pezaros. "On the Latency Benefits of Edge NFV". In: *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. May 2017, pp. 105–106. DOI: 10.1109/ANCS.2017.23.

[125] Attila Csoma et al. "ESCAPE: Extensible service chain prototyping environment using mininet, click, netconf and pox". In: *ACM SIGCOMM Computer Communication Review*. Vol. 44. 4. ACM. 2014, pp. 125–126.

[126] Rogério Leão Santos De Oliveira et al. "Using mininet for emulation and prototyping software-defined networks". In: *Communications and Computing (COLCOM), 2014 IEEE Colombian Conference on*. IEEE. 2014, pp. 1–6.

[127] Eddie Kohler et al. "The Click modular router". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 263–297.

[128] Kashi Venkatesh Vishwanath and Amin Vahdat. "Realistic and responsive network traffic generation". In: *ACM SIGCOMM Computer Communication Review*. Vol. 36. 4. ACM. 2006, pp. 111–122.

[129] Sijia Gu et al. "An efficient auction mechanism for service chains in the NFV market". In: *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*. IEEE. 2016, pp. 1–9.

[130] Robert Szabo et al. "Elastic network functions: opportunities and challenges". In: *IEEE network* 29.3 (2015), pp. 15–21.

[131] Lianjie Cao et al. "NFV-VITAL: A framework for characterizing the performance of virtual network functions". In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE. 2015, pp. 93–99.

[132] Levente Csikor et al. "NFPA: Network function performance analyzer". In: *Network Function Virtualization and Software Defined Network (NFV-SDN)*. IEEE. 2015, pp. 15–17.

[133] Kapil Sood, Jeffrey B Shaw, and John R Fastabend. *Technologies for secure inter-virtual network function communication*. US Patent 9,407,612. Feb. 2016.

[134] Murad Kablan et al. “Stateless network functions”. In: *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*. ACM. 2015, pp. 49–54.

[135] Stephen Soltesz et al. “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors”. In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 275–287.

[136] Ming-Wei Shih et al. “S-nfv: Securing nfv states by using sgx”. In: *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization*. ACM. 2016, pp. 45–48.

[137] Shankar Lal, Tarik Taleb, and Ashutosh Dutta. “NFV: Security threats and best practices”. In: *IEEE Communications Magazine* 55.8 (2017), pp. 211–217.

[138] Sandra Scott-Hayward, Gemma O’Callaghan, and Sakir Sezer. “SDN security: A survey”. In: *Future Networks and Services (SDN4FNS), 2013 IEEE SDN For*. IEEE. 2013, pp. 1–7.

[139] Lin Cui et al. “Synergistic policy and virtual machine consolidation in cloud data centers”. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 2016, pp. 1–9.

[140] Chang Lan et al. “Embark: securely outsourcing middleboxes to the cloud”. In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 2016, pp. 255–273.

[141] Matteo Pozza et al. “Network-In-a-Box: A Survey about On-Demand Flexible Networks”. In: *IEEE Communications Surveys & Tutorials* (2018).

[142] Hitesh Ballani et al. “Towards predictable datacenter networks”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 242–253.

[143] Jeffrey C Mogul and Lucian Popa. “What we talk about when we talk about cloud network performance”. In: *ACM SIGCOMM Computer Communication Review* 42.5 (2012), pp. 44–48.

[144] Bob Gillian. *VYATTA: Linux IP Routers, Dec 2007*. 2005.

[145] Johannes Thönes. "Microservices". In: *IEEE Software* 32.1 (2015), pp. 116–116.

[146] Narjes Tahghigh Jahromi et al. "An NFV and Microservice Based Architecture for On-the-fly Component Provisioning in Content Delivery Networks". In: *Proceedings of IEEE CCNC 2018* (2018).

[147] ETSI. *DGS/NFV-EVE004. Network Functions Virtualisation (NFV); Virtualisation Technologies; Report on the application of Different Virtualisation Technologies in the NFV Framework*. 2016.

[148] Sergei Arnautov et al. "SCONE: Secure Linux Containers with Intel SGX." In: *OSDI*. USENIX. 2016, pp. 689–703.

[149] Matt Helsley. "LXC: Linux container tools". In: *IBM devloperWorks Technical Library* 11 (2009).

[150] B. Martini et al. "Latency-aware composition of Virtual Functions in 5G". In: *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*. Apr. 2015, pp. 1–6. DOI: 10.1109/NETSOFT.2015.7116188.

[151] F. Ben Jemaa, G. Pujolle, and M. Pariente. "QoS-Aware VNF Placement Optimization in Edge-Central Carrier Cloud Architecture". In: *GLOBECOM*. Dec. 2016, pp. 1–7. DOI: 10.1109/GLOCOM.2016.7842188.

[152] Arthur Q Frank and Stephen M Samuels. "On an optimal stopping problem of Gusein-Zade". In: *Stochastic Processes and their Applications* 10.3 (1980), pp. 299–311.

[153] Jie Cao et al. "EdgeOS_H: A Home Operating System for Internet of Everything". In: *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE. 2017, pp. 1756–1764.

[154] "PowerMan: An Out-of-Band Management Network for Datacenters using Power Line Communication". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018. URL: https://www.usenix.org/node/210927.

[155] Arjun Singh et al. “Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network”. In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 183–197.

[156] Kevin Phemius and Mathieu Bouet. “Monitoring latency with openflow”. In: *Network and Service Management (CNSM), 2013 9th International Conference on*. IEEE. 2013, pp. 122–125.

[157] M Mahalingam et al. “Virtual extensible local area network (VXLAN): A framework for overlaying virtualized layer 2 networks over layer 3 networks”. In: *Internet Req. Comments* (2014).

[158] D Farinacci et al. “Generic Routing Encapsulation over IPv4 networks”. In: (1994).

[159] Pui Y Lee, Siu C Hui, and Alvis Cheuk M Fong. “Neural networks for web content filtering”. In: *IEEE Intelligent Systems* 17.5 (2002), pp. 48–57.

[160] Felix Ming Fai Wong et al. “Improving user QoE for residential broadband: Adaptive traffic management at the network edge”. In: *Quality of Service (IWQoS), 2015 IEEE 23rd International Symposium on*. IEEE. 2015, pp. 105–114.

[161] Klaithem Al Nuaimi et al. “A survey of load balancing in cloud computing: Challenges and algorithms”. In: *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*. IEEE. 2012, pp. 137–142.

[162] C Farrell et al. *DNS encoding of geographical location*. Tech. rep. 1994.

[163] Matei Zaharia et al. “Apache spark: a unified engine for big data processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.

[164] Carl A Waldspurger. “Memory resource management in VMware ESX server”. In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 181–194.

[165] Sergio Livi et al. “Container-Based Service Chaining: A Performance Perspective”. In: *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*. IEEE. 2016, pp. 176–181.

[166] Guohui Wang and T.S.E. Ng. "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center". In: *INFOCOM 2010*. IEEE. Mar. 2010, pp. 1–9. DOI: 10.1109/INFCOM.2010.5461931.

[167] Alexandru Iosup, Nezih Yigitbasi, and Dick Epema. "On the performance variability of production cloud services". In: *Cluster, Cloud and Grid Computing (CCGrid), 2011 11th IEEE/ACM International Symposium on*. IEEE. 2011, pp. 104–113.

[168] Mohammad Alizadeh et al. "Data center tcp (dctcp)". In: *ACM SIGCOMM computer communication review*. Vol. 40. 4. ACM. 2010, pp. 63–74.

[169] Kathleen Nichols Jim Gettys. "Bufferbloat: Dark Buffers in the Internet". In: *Queue* 9.11 (Nov. 2011), 40:40–40:54.

[170] Gaetano Carlucci, Luca De Cicco, and Saverio Mascolo. "HTTP over UDP: an Experimental Investigation of QUIC". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM. 2015, pp. 609–614.

[171] Ramesh Govindan and Vern Paxson. "Estimating router ICMP generation delays". In: *Passive & Active Measurement (PAM)*. 2002.

[172] Richard Cziva, Christopher Lorier, and Dimitrios P Pezaros. "Ruru: High-speed, Flow-level Latency Measurement and Visualization of Live Internet Traffic". In: *Proceedings of the SIGCOMM Posters and Demos*. ACM. 2017, pp. 46–47.

[173] Robert L Carter and Mark E Crovella. *Dynamic server selection using bandwidth probing in wide-area networks*. Tech. rep. Boston University Computer Science Department, 1996.

[174] Chuanxiong Guo et al. "Pingmesh: A large-scale system for data center network latency measurement and analysis". In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. ACM. 2015, pp. 139–152.

[175] Changhua Pei et al. "WiFi can be the weakest link of round trip network latency in the wild". In: *INFOCOM 2016*. IEEE. 2016, pp. 1–9.

[176] Bernhard Meindl and Matthias Templ. “Analysis of commercial and free and open source solvers for linear optimization problems”. In: *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS* 20 (2012).

[177] Jayavardhana Gubbi et al. “Internet of Things (IoT): A vision, architectural elements, and future directions”. In: *Future generation computer systems* 29.7 (2013), pp. 1645–1660.

[178] Shigang Chen and Qingguo Song. “Perimeter-based defense against high bandwidth DDoS attacks”. In: *IEEE Transactions on Parallel and Distributed Systems* 16.6 (2005), pp. 526–537.

[179] Richard Alimi, Ye Wang, and Y Richard Yang. “Shadow configuration as a network management primitive”. In: *ACM SIGCOMM Computer Communication Review* 38.4 (2008), pp. 111–122.

[180] Volkan Yazıcı, Ulas C Kozat, and M Oguz Sunay. “A new control plane for 5G network architecture with a case study on unified handoff, mobility, and routing management”. In: *IEEE Communications Magazine* 52.11 (2014), pp. 76–85.

[181] Joel Halpern and Carlos Pignataro. *Service function chaining (sfc) architecture*. Tech. rep. 2015.

[182] Dimitrios P Pezaros, Richard Cziva, and Simon Jouet. “SDN for Cloud Data Centres”. In: *Big-Data and Software Defined Networks*. IET Book Series on Big Data. IET, Mar. 2017. DOI: 10.1049/PBPC015E_ch4.

[183] Richard Cziva, Jerry Sobieski, and Yatish Kumar. “High-Performance Virtualized SDN Switches for Experimental Network Testbeds”. In: *Proceedings of the 3rd Workshop on Innovating the Network for Data-Intensive Science*. ACM. 2016, pp. 1–8.

[184] Simon Jouet, Richard Cziva, and Dimitrios P Pezaros. "Arbitrary packet matching in openflow". In: *High Performance Switching and Routing (HPSR), 2015 IEEE 16th International Conference on*. IEEE. 2015, pp. 1–6.

[185] Richard Cziva et al. "SDN-based Virtual Machine management for Cloud Data Centers". In: *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. Oct. 2014, pp. 388–394.