Julius-Maximilians-Universität Würzburg
Institut für Informatik
Lehrstuhl für Kommunikationsnetze
Prof. Dr.-Ing. P. Tran-Gia

Architectures for Softwarized Networks and Their Performance Evaluation

Steffen Christian Gebert

Würzburger Beiträge zur
Leistungsbewertung Verteilter Systeme

Bericht 1/17

Würzburger Beiträge zur Leistungsbewertung Verteilter Systeme

Herausgeber

Prof. Dr.-Ing. P. Tran-Gia
Universität Würzburg
Institut für Informatik
Lehrstuhl für Kommunikationsnetze
Am Hubland
D-97074 Würzburg

Tel.: +49-931-31-86630
Fax.: +49-931-31-86632
email: trangia@informatik.uni-wuerzburg.de

Satz

Reproduktionsfähige Vorlage des Autors.
Gesetzt in LaTeX Linux Libertine 10pt.

ISSN 1432-8801

Architectures for Softwarized Networks and Their Performance Evaluation

Dissertation zur Erlangung des
naturwissenschaftlichen Doktorgrades
der Julius–Maximilians–Universität Würzburg

vorgelegt von

Steffen Christian Gebert

aus

Kitzingen

Würzburg 2017

Eingereicht am: 26. Januar 2017
bei der Fakultät für Mathematik und Informatik
1. Gutachter: Prof. Dr.-Ing. Phuoc Tran-Gia
2. Gutachter: Prof. Dr. Paul Müller
Tag der mündlichen Prüfung: 9. Juni 2017

Danksagung

In den sechs Jahren meiner Promotion und insbesondere beim Verfassen dieser Arbeit wurde ich von einer Vielzahl von Kollegen, Freunden und Angehörigen unterstützt und motiviert. Einigen hiervon möchte ich im Folgenden meinen Dank aussprechen.

Meinem Doktorvater Prof. Phuoc Tran-Gia gilt mein Dank für die Gelegenheit, an seinem Lehrstuhl eine Promotion zu verfolgen. Ich bin sehr dankbar dafür, dass er eine Arbeitsumgebung geschaffen hat, in der Teamwork und gegenseitiger Respekt großgeschrieben wird. Die Zusammensetzung des Lehrstuhls und der seitens der Post-Docs stattfindende Wissenstransfer ist meines Achtens einer der größten Erfolgsfaktoren und wichtige Unterstützung für Doktoranden jeden Ausbildungsgrades. Die Teilnahme an Industrie-nahen Forschungsprojekten, die Motivation, Resultate in Publikationen zu verbreiten, sowie die Teilnahme an internationalen Konferenzen tragen weiter zum wissenschaftlichen Erfolg bei. Aber auch der Anspruch, zu jeder Zeit gute Lehre zu leisten - sowohl hinsichtlich Vorlesungen, aber insbesondere auch bezüglich der Betreuung von Studierenden während ihrer Abschlussarbeiten - macht mich stolz, langjähriger Teil dieses Lehrstuhls gewesen zu sein.

Herrn Prof. Paul Müller möchte ich für sein Interesse an meiner Forschung und der Bereitschaft, das Zweitgutachten zu meiner Arbeit zu verfassen, danken. Den Mitgliedern der Prüfungskommission Prof. Rainer Kolla sowie Prof. Hakan Kayal gilt ebenfalls mein Dank.

Das gute Miteinander unter den Kolleginnen und Kollegen am Lehrstuhl für Informatik III zeigt sich in erfolgreicher wissenschaftlicher Arbeit und in zahlreichen privaten Freundschaften. Für die Unterstützung meiner Forschungsarbeiten und auch in verschiedensten Lebenssituationen möchte ich mich ganz herzlich

bei meinem Gruppenleiter Dr. Thomas Zinner bedanken. Sowohl die notwendige Führung zu geben, als auch das Vertrauen zu schenken, eigenständig in seiner Vertretung in Projekten zu agieren, haben zu meiner persönlichen Entwicklung beigetragen.

Über die Jahre, einschließlich die meiner Hiwi-Zeit, hatte ich Gelegenheit, mit einer Vielzahl von netten Kollegen zusammenarbeiten zu können. Für das konstruktive und freundschaftliche Miteinander geht mein Dank hierzu an Kathrin Borchert, Valentin Burger, Lam Dinh-Xuan, Michael Duelli, Stefan Geißler, Nicholas Gray, Matthias Hartmann, Robert Henjes, Matthias Hirth, Tobias Hossfeld, David Hock, Michael Jarschel, Dominik Klein, Stanislav Lange, Frank Lehrieder, Christopher Metter, Anh Nguyen Ngoc, Rastin Pries, Daniel Schlosser, Christian Schwarz, Michael Seufert, Barbara Stähle, Dirk Stähle sowie Florian Wamser. Unserer Sekretärin Frau Alison Wichmann möchte ich Danke sagen für das freundschaftliche Miteinander, sowie die Unterstützung bei allen meinen Problemen mit dem deutschen Bürokratismus und ihre Sorgfalt, meine Defizite diesbezüglich auszubügeln.

Während meines Aufenthalts als Gastwissenschaftler an der Universität Wien hatte ich die Freude, unter Leitung von Prof. Kurt Tutschku mit Florian Metzger, Albert Rafetseder und David Stezenbach zusammenarbeiten. Diesen möchte ich ebenfalls für die schöne gemeinsame Zeit danken.

Zahlreiche der in meiner Dissertation beschriebenen Ergebnisse sind im Rahmen studentischer Abschlussarbeiten oder durch Arbeiten studentischer Hilfskräfte entstanden. Der Umgang mit einer Vielzahl von unterschiedlichen Charakteren hat mich auch um einige Erfahrungen reicher gemacht. Folgenden Studenten gilt mein Dank für ihre Unterstützung und Mitarbeit an meinen Forschungsarbeiten: Shpend Berani, Roland Borsos, Fabian Helmschrott, Stefan Herrnleben, Frederik König, Florian Mayer, Alexander Müssig, Stephan Oberste-Vorth, Benedikt Pfaff, Richard Rauscher, Lorenz Reinhart, Johann Scherer, Johannes Schnappauf, Marco Schreck sowie Alexander Werthmann.

Abseits der Unterstützung in der Universität gebührt ein großer Anteil an meinem erfolgreichen Karriereweg natürlich auch meinem privaten Umfeld. Sehr dankbar bin ich meinen Eltern dafür, dass ich mein Studium ohne finanzielle Sor-

gen bewältigen konnte. Dafür, dass sie meine Begeisterung für Computer und Technik stark gefördert hat, möchte ich v.a. meiner Mutter Erika sehr danken. Auch wenn wir uns einig sind, dass einige der vererbten Charaktereigenschaften nicht immer von Vorteil sind, möchte ich ihr trotzdem hierfür danken.

Schlussendlich gilt ein ganz besonderer Dank meiner Verlobten Julia. Uns beide hat nicht nur der Lehrstuhl zusammen gebracht, wir sind auch weiterhin gerne bei unseren Freunden dort zu Gast. Für ihre Unterstützung und Motivation, sowie auch den leider häufigen Verzicht auf meine Anwesenheit beim Verfassen dieser Arbeit gilt ihr mein unendlicher Dank.

Contents

1 Introduction

Computer networks play a critical role in nowaday's connected world—seen from both viewpoints—the regional scale with provider networks, mobile networks, and enterprise networks, as well as on a global scale with the Internet as interconnection of networks. Novel applications and services use and the underlying network infrastructure and are created literally every day. These applications and services also come with new requirements for the networks and could as well provide a better user experience, if the network-side would provide better support for these applications.

However, the underlying protocols and mechanisms that are used in nowadays communication networks have, however, barely changed since decades. This often prevents the deployment novel services with very specific requirements, e.g., regarding latency, availability, or throughput, or increases costs for using the service. Because interoperability and backward compatibility, which are the requirements of the networks, most innovations have been essentially blocked, very much in contrast to changes on the end devices.

These historic remnants range from the exchange of data among different hosts using the *Internet Protocol* (IP), the addressing schema of hosts, again based on IP, to the exchange of routing information among providers using the *Border Gateway Protocol* (BGP). Novel concepts like *Locator-Identifier Split* (LISP) for addressing or *Path Computation Element Protocol* (PCEP) for routing stagnate in academic research without real-world deployments, as too many stakeholders would have to agree on and coordinate such changes.

However, few institutions considered taking different paths to bring innovation at least into their very own network, without involvement of external stakeholders and hosts. The need to innovate rapidly led cloud providers like Google and

Facebook, which have to deal with massive amounts of data, into development of custom, software-based solutions according to very own needs within their own networks. By altering the behavior of their network components, these companies are able to use custom routing schemes, increase flexibility, and their link utilization [25], thus lowering the costs without requiring technological changes at end users.

Such influence, however, has not always been possible due to the role of equipment vendors, on which most innovation used to depend on. The functionality of a network device used to be implemented in its firmware. This prevented operators to deploy additional networking functionalities even in their local networks, i.e., routing protocols, traffic engineering and monitoring mechanisms, or custom addressing schemes. Through the advent of mechanisms providing an interface to device behavior, or allowing to replace entire networking devices with software running on commodity hardware, totally new levels of innovation become realistic.

1.1 Classification from Scientific Viewpoint

The scientific background of this work is based on the concept of softwarized networks, where innovation can be brought into network infrastructures by custom-built software instead of hardware shipped and programmed by device vendors. This removes functionality out of device firmware and instead lets it run as software application on a standard server.

Such software-based networks evolved over the past years by the introduction of multiple concepts and technologies, which are investigated and applied in several research projects, e.g., Saser as well as Sendate [26]. In the following, a brief overview over the technologies that transform network infrastructures from hardware-focused to software-driven is given.

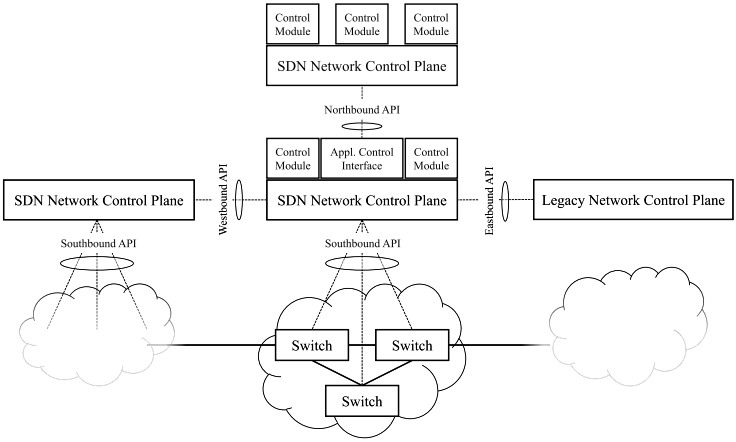


Figure 1.1: Architecture of a Software Defined Network [27].

1.1.1 Software Defined Networking (SDN)

Software Defined Networking (SDN) is a technological concept that lets a centralized software application decide about the network's forwarding behavior. In contrast, in traditional networking approaches, forwarding decisions are taken decentralized by the network devices.

Figure 1.1 depicts said SDN architecture. The centrally located network control plane is connected to all SDN switches via the *Soutbound API*. Through that, the controller is able to insert rules into the switches' *Forwarding Information Base* (FIB), which influences how packets are forwarded. The controller itself offers an *Application Programming Interface* (API) for applications that need to interact with the network, the *Northbound API*.

The benefits of steering the network with all its devices, like switches and routers, through a centralized software include the following [27, 28]:

Agility and flexibility. The controller software runs as an application on standard server hardware, like x86, and a standard operating system, like

Linux. This allows an operator to let the controller software run together with other business applications and without requiring dedicated hardware appliances.
Through the increased development speed of software compared to hardware devices and their respective firmware, changes and innovations are simplified. Finally, the development of this software can follow common software development processes.

Better resource usage. The central view on all network devices allows the controller to optimize the network's performance and resource usage. Knowledge of the complete network topology and link utilizations allows the controller software to better coordinate traffic flows [25] within the network so that a global optimum can be reached.

Vendor-independence. The separation between hardware devices being responsible for packet forwarding and a controller software, which decides about how to forward, requires a communication protocol between these to entities. As this protocol should be standardized instead of proprietary, network devices and controller software using such interface can origin from different sources such as commercial, open source, or custom development. The most popular southbound API protocol (also called *Control Data Plane Interface* (CDPI)) that devices such as switches implement is *OpenFlow* [28, 29].
Similar to other software projects, controller software can offer means for extensibility through plugins. Again, these plugins that add functionality might be offered by vendors or extend the network control plane very specific to the needs of the operator.

Motivated by these advantages, many hardware vendors introduced SDN interfaces into their devices. Further, network vendors as well as software companies offer controller software, either as commercial products or as open source project with commercial add-ons. In addition, large community projects spanning multiple companies were set up during the past years and are shepherded by non-profit

organization like the *Linux Foundation* [30, 31]. The different implementations make the SDN ecosystem very diverse and in many places, it is unclear, where to draw the line between SDN and non-SDN.

1.1.2 Network Functions Virtualization (NFV)

Network Functions Virtualisation (NFV) [32] leverages standard IT hardware in order to replace proprietary hardware appliances. Such appliances include all network functions, such as firewalls, *Deep Packet Inspection* (DPI), or *Customer Premises Equipment* (CPE) in general, as well as entities of the mobile core network. Through the execution as software application on standard server hardware, similar to the SDN controller, different advantages can be expected, many of them overlapping with those of SDN.

These software instances, the *Virtualised Network Functions* (VNFs), run virtualized in cloud environments and can thus be scaled according to the actual need. This avoids over-provisioning of hardware for peak load and reduces energy consumption. In addition, the decomposition of previously bundled functionality into smaller function blocks, similar to the *Microservices* concept [33], allows more fine-grained resource scaling and allocation, as well as higher development speed.

The risk of a vendor lock-in is further decreased, as long as standardized interfaces are used and implemented. The decomposition into functional blocks further allows to combine components of different vendors instead of relying completely on one supplicant, which makes the system hard to replace.

This fine-grained architecture also allows to reduce the overall functionality to the actually needed feature set. Traditional hardware devices usually bundle a lot more features than any operator uses and the vendor decides about what features are included and which are removed.

While NFV does not explicitly rely on SDN, these two techniques merge together very well. The network functions (VNFs) run in software on commodity servers and the traffic flow between those are defined through SDN mechanisms.

In order to streamline NFV activities, a working group within the *European*

Telecommunications Standards Institute (ETSI) defines standards for different aspects of NFV, including its terminology, architecture, and management mechanisms. The reference implementation *Open Platform for NFV* (OPNFV) [34] is currently under early development and coordinated by the Linux Foundation. In this collaborative project, nearly all vendors and many operators in the telecommunication field are participating. In contrast to SDN, the NFV activities seem very well-aligned between different vendors following the ETSI specifications.

1.1.3 Packet Processing in COTS Hardware

By running VNFs on standardized server hardware that is historically not designed for such extremely network-focussed applications, a performance impediment compared to the very specialized *Application-specific Integrated Circuits* (ASICs) in hardware appliances can be expected. The architecture of x86 operating systems and how network data is processed results in a high overhead per packet. These systems were not designed for heavily packet processing oriented applications, but for running applications that receive requests over the network instead and have resource requirements major on compute power.

Modern operating systems already apply optimizations, which overcome the peculiarities of the underlying hardware, such as *Interrupt Moderation*. Besides potentially unwanted effects, the performance of general purpose networking stacks only allows processing in the order of some Gigabits per second.

Therefore, numerous optimization techniques were introduced that allows an application to bypass the operating system's network stack for high speed packet processing. Software frameworks implementing such techniques vary in complexity and the degree of optimization, as well as on the requirements of the underlying software and sometimes even hardware. While implementing a VNF, the development team has to decide, which framework to use based on the characteristics and requirements of the application. Using frameworks such as Intel's *Data Plane Development Kit* (DPDK) [35], the throughput can be increased, as the overhead per packet is reduced. Intel reports "usually less than 80 [CPU] cycles" for receiving and sending a packet [35].

	Cloud	SDN	NFV
Main goals	• Cost savings • Increased agility • Cost savings	• Cost savings • Increased agility • Vendor independence	• Cost savings • Increased agility • Vendor independence
Methods	• Virtualization (compute, storage, network) • Everything-as-a-Service approach (XaaS) • APIs for everything	• Centralized control plane • Run control plane in software • APIs for the network	• Replace appliances with software instances • Run network function (CP + DP) in software • common APIs for network functions
Issues	• Applications need to be made for the cloud • Privacy • Focus on applications, not the network	• Scalability vs. degree of control/detail • Little benefits when used "standalone" • Implementation detail	• Data plane performance • Management & Orchestration • Mind shift for dealing with failure

Table 1.1: Comparison of technologies cloud, SDN, and NFV.

1.1.4 Open Issues

Besides all the appealing benefits of more agile, flexible, and efficient network infrastructure, numerous issues hinder the migration to completely softwarized networks. Table 1.1 provides an overview over the technologies of softwarized IT environments, such as SDN, NFV, and *cloud*. We compare the different goals, methods, as well as the open issues.

The open issues of SDN and NFV will be covered in this monograph. The control plane performance of SDNs will be investigated in Section 4.2, while its benefits in combination with extensions for improved network management as well as network security will be described in Section 3.2 and Section 3.4. One of the

core NFV challenges is the packet processing performance, which will be investigated based on measurements in Sections 4.3 and 4.4. Performance prediction for VNFs will be aided by the analytical model provided in Section 5.2. By describing guide lines for lifecycle of VNF software versions, Section 3.2 contributes to the management of NFV-based networks.

1.2 Scientific Contribution

This monograph addresses challenges that arise by applying the previously described techniques for software-based networks to different use cases. First, mechanisms to assess and to improve the performance of software-based networking components are described. Second, possible architectural changes caused, as well as enabled, by SDN and NFV are addressed. Finally, new ways to manage these softwarized networks are introduced.

An overview of the contributions of this work is given in Figure 1.2. It classifies the individual research studies that were carried out during the course of this work based on the methods applied by the author. Conducted experiments, measurements, and proof-of-concept implementations make up the practice-oriented part. The theoretical contributions use simulation and queueing theory, or provide design guide lines for a potential implementation. The color coding indicates the chapter, to which a particular scientific publication or demonstration contributes.

The first area of contribution covers the performance of particular components. On the one hand, this include the controller software that represents the network control plane. The performance of these components is an important indicator, how on what granularity and how rapidly the network can react, i.e., how frequently new flows can be programmed based on new flow arrivals.

To evaluate the performance characteristics of network functions implemented through the means of NFV, two commercially available firewall products were compared regarding their data plane performance. While the hardware device provided lower and especially more consistent latencies, the increase in processing time when using the software implementation might be neglected, based on

the particular use case. Finally, a performance assessment for a prototypical VNF for a mobile network highlights the big potential offered by acceleration frameworks such as Intel DPDK compared to standard Linux networking.

Finally, the last significant contribution related to the performance of network components is an analytical model for packet processing on a physical server using the Linux operating system and its networking implementation. Based on given service times of the VNF and distributions of the inter arrival times of new packets, the effects of tuning parameters can be evaluated in regard to their influence on processing times, induced jitter, and packet loss.

The second area of contribution covers novel networking architectures that are required either for, or enabled by softwarized networks. One use case, S-BYOD, illustrates how SDN allows to implement micro segmentation to safely isolate the traffic of different devices and services within a corporate environment.

This part further covers the placement of entities of a softwarized networks, with a major focus on the placement of SDN controllers within wide area networks. The planning tool that was further developed respects multiple, often competing objectives, like controller load, distance to the nodes, or distances between multiple controllers. As particular thresholds for reaction times and resiliency vary among different operators, this framework does not make any assumption about good or bad placements. Due to the fact that many operators are not even able to define thresholds for several of the metrics, Pareto-optimal placements are returned and available for manual selection.

The final third area, which this monograph covers, are management aspects of softwarized networks. By providing APIs, the network infrastructure now offers better means for monitoring, optimization, and automation. How such APIs allow for implementing novel monitoring approaches is illustrated by the ZOOM algorithm. Network programmability is exploited to instantiate new hardware counters for varying flow aggregates in the switches. By evaluating these counters in a controller software and iteratively programming more specific flow rules, monitoring tasks like elephant detection can be implemented in a completely different and more dynamic way than with traditional networking gear and protocols like NetFlow.

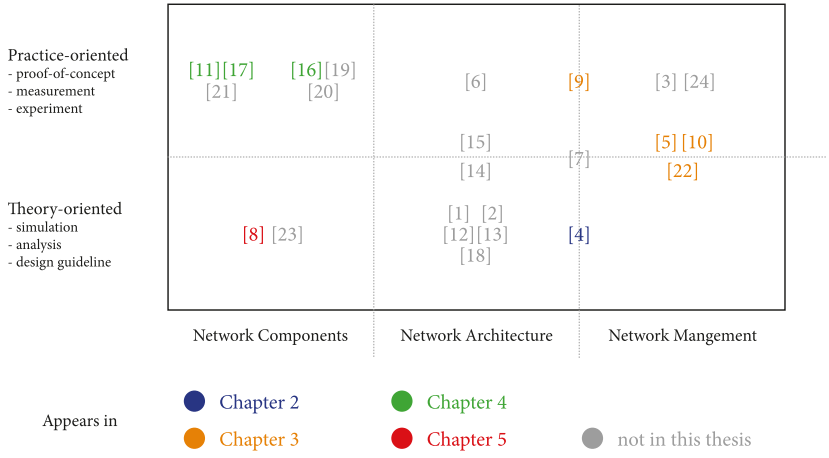


Figure 1.2: Contribution of this work as a classification of the research studies conducted by the author.

The last contribution is in regard to automated deployment processes for networking infrastructure. Based on the successful software engineering concept continuous delivery, guide lines for its application to network infrastructures are developed. Of particular focus is the automated testing process prior to the automated rollout. Again based on a successful software engineering concept, behavior-driven development, functional tests contain acceptance criteria that would stop any automated deployment, as soon as a test fails. Given such confidence, network engineers can finally move faster, have audit trails provided by version control, and get test environments through instantiation in virtual environments.

1.3 Outline of This Thesis

The organization of this monograph is as follows. Each chapter contains an introductory section providing background information as well as related work.

After detailing the conducted studies and describing the scientific contribution, the chapter close with a section summarizing the lessons learned.

The remainder of this thesis is structured as follows. Chapter 2 focuses on improving the planning process of optical multi-layer networks. By modifying the input parameters supplied to an existing network planning tool, it is evaluated, how much *Operating Expense* (OPEX) in terms of energy spent for fiber links and adapters can be saved.

Chapter 3 elaborates on novel management approaches for software-based networks. The first part describes, how continuous delivery and automated acceptance test help a network operator to reduce lead times for changes. In contrast to traditional, big-bang deployments, automated tests help to move faster and deploy in smaller changes within shorter time frames. Further, an example how programmability and APIs help to ease network management is given using a monitoring implementation for elephant detection monitoring.

Chapter 4 investigates the performance of the components within softwarized networks. The benchmarking studies for different VNF implementations reveal different performance characteristics, on the one between physical and software implementation, but also between software implementations using the operating system APIs or those using specialized acceleration frameworks. Further, a benchmarking software to assess the control plane performance of OpenFlow-based SDN is described. This helps operators for capacity planning when moving towards such software-based network.

Chapter 5 introduces an analytical model for computing the key characteristics of a VNF using the Linux networking APIs. Based on incoming packet inter-arrival times and the service time of the network function, the estimated packet loss and added delay can be computed. Additionally, the effects of parameter tuning of *Network Interface Cards* (NICs) can be foreseen.

Finally, Chapter 6 concludes this monograph and summarizes the presented results and achievements.

2 Multi-Layer Optical Networks

Computer networks range from small ones, the *Local Area Networks* (LANs), to large ones, the *Wide Area Networks* (WANs), which finally form the Internet. Nearly all WANs use optical fiber as media, as a single fiber strand provides bandwidth capacities in the order of tens of terabits per second.

As no equipment exists that nearly exploits this capacity in form of allowing to establish such high-bandwidth connection directly between two devices, multiplexing techniques are used to aggregate the traffic of multiple links. The multiplexing scheme used in the optical domain based on *Frequency Division Multiplexing* (FDM) is *Wavelength-Division Multiplexing* (WDM): Using different wavelengths, i.e, colors of light, multiple connections can be mixed, sent over one single fiber, and split into multiple links at the receiver side, e.g., at the other site of the ocean. The highest link speed that is currently deployed transfers 100 Gigabit per second (Gbps) – and occupies one wavelength. The efficiency of WDM and the thus available high bandwidth results in *Space-Division Multiplexing* (SDM) being applied only conservatively: Even transatlantic fiber cables contain only four pairs of optical fiber [36].

As 100 Gbps links are in most cases again an aggregation of multiple slower links, more multiplexing happens on top of these optical links. Based on *Time-Division Multiplexing* (TDM), multiple virtual connections with a fixed bandwidth can be established over a single link, or statistical multiplexing allows to dynamically allocate bandwidth to the multiplexed connections.

The availability of a multitude of such multiplexing techniques, as well as the practice to apply multiple of these techniques on top of each other in form of logical layers, lead to the term *multi-layer networks*.

On each of these logical layers, *virtual paths* spanning multiple hops between

the different *sites* of the network can be established. This abstraction allows to simplify the network configuration and helps increase to processing speed, as not every hop needs to apply compute-intense forwarding techniques like IP routing. Instead, an optical signal can be directly passed from one optical fiber to another without even converting it into an electrical signal.

Combined with a bandwidth contingent, such virtual paths are referred to as *demands*, either uni- or bidirectional between the two sites of the network. Contrary to the end customer area, where a link's capacity is often shared between many customers and thus not available to everybody at the same time, bandwidth contingents are here usually reserved and guaranteed per demand.

The particular *realization* of a demand, which is computed in the *network planning* phase, can rely on any of the previously mentioned multiplexing techniques. The multi-layer network planning process will be covered in depth in the following sections, where special emphasis will be put on energy efficient planning of multi-layer networks.

2.1 Background and Related Work

This section introduces the techniques behind multi-layer networks and highlights the most relevant research work done in the field of multi-layer network planning with a focus on energy efficiency and resilience. Further, an overview over the challenges of the planning process will be given.

2.1.1 Network and Internet Architecture

The Internet is not a single, homogeneous network, but an interconnection of many networks owned, managed and used by different parties. However, not all of these interconnected networks provide the same functionality or exist for the same purpose. Traditionally, networks can be categorized into the following three types of networks (cf. Figure 2.1).

Access networks. Private users as well as smaller business customers are connected to the *access network* of their *Internet Service Provider* (ISP). Typi-

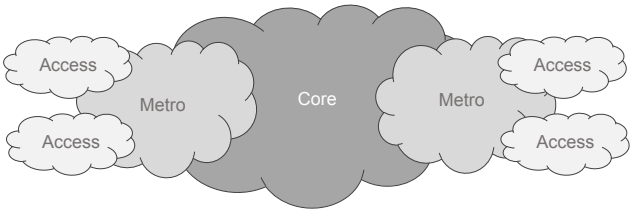


Figure 2.1: Network hierarchy.

cal technologies used in these networks are *Digital Subscriber Line* (DSL), based on the telephone system, and *Data Over Cable Service Interface Specification* (DOCSIS), which uses the cable TV system. Both of them are based on copper cables, while optical techniques such as *Gigabit Passive Optical Networks* (GPON) are more and more being deployed. In the wireless area, the *xG* mobile standards (3G/UMTS, 4G/LTE) are dominating at least in western countries.

Metro networks. The interconnection of multiple access networks as well as networks of enterprise customers that are spanning multiple campuses are examples for *metro networks*. Such networks mostly rely on optical network technologies given their span of a city or region level. The capacities of metro links vary, ranging from few Gbps for connecting different enterprise sites or aggregation of customer premise links to data center interconnects of tens or hundreds of Gigabits.

Core networks. The highest bandwidths occur in *core networks*, the "heart" of each provider network. These networks provide capacities for connecting millions of customers with data centers comprising tens or hundreds of thousands servers, as well as transoceanic connectivity. The previously mentioned fastest link rates of 100 Gbps are mostly deployed in core networks, combined with multiplexing of multiple such links on one fiber. Ranging from a country- to worldwide span, and aggregating the previ-

ously described networks' traffic, core networks are the most complex, critical and expensive type of network. To realize such networks in an affordable way, a combination of several multiplexing technologies have to be made.

The first part of this monograph will focus on the planning process of such optical multi-layer networks, which is an important concept for nowadays core networks. The complexity of choice based on several available multiplexing techniques, as well as the costs of required equipment makes a well designed network essential. The result of this planning process is a list of equipment to buy and a configuration of which demand is realized using which technology using a defined path through the network.

2.1.2 Multi-Layer Networks

The benefit of the multi-layered network concept is that it simplifies the network architecture by hiding the details of lower layers, both in the practical usage as well as in case of failures, which will be covered later in Section 2.1.4.

Figure 2.2 illustrates the multi-layered network principle for a WAN in a simplified way using three layers. How many layers are used in practice, depends on the operators' needs.

The graph G_0 of the *lower layer* is usually the optical fiber that an operator owns or rents. The time scale of changes in this layer are in the order of months or years. Due to the immense costs of burying new optical fiber strands across cities or oceans, such networks are rarely fully meshed. This is illustrated in Figure 2.2, where e.g. no link $e_{1,3}$ between sites 1 and 3 exists. Connecting all sites directly would be simply too expensive. However, this lack of direct links is solved by the higher layers, which will provide such virtual paths spanning over multiple fiber links and hops.

When investigating the shown *middle layer* G_1, it can be observed that there is a link $e_{1,3}$, which does not exist in the layer below. This logical path is created using one of many techniques, which will be explained in the following section.

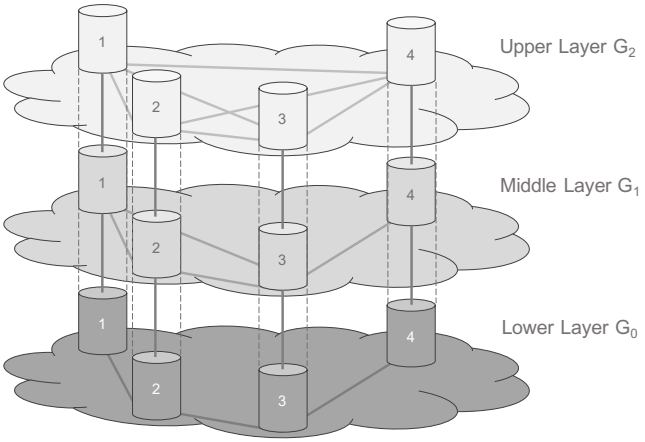


Figure 2.2: Multi-layered network.

Finally, the graph G_2 of the *upper layer* is fully meshed. In the networking context, this means that there is full connectivity between all sites of the network.

Multi-Layer Network Techniques

The previously discussed multiplexing schemes that mix optical or electrical signals are realized using different standardized techniques. In the following, a brief overview over the most important multi-layer network techniques will be given.

OTH describes the transport technology for the *Optical Transport Network* (OTN) [37, 38]. This hierarchy is based on the concepts *Optical Channel* (OCh), *Optical Multiplex Section* (OMS), and *Optical Transmission Section* (OTS). Solely based on optical technology, the OTN directly accesses the optical fiber including *Forward Error Correction* (FEC) mechanisms. As illustrated in Figure 2.3, OTSs span one hop, e.g., between a router and an optical amplifier. On top, one wavelength spanning multiple sections between two *Optical Add-Drop Multiplexers* (OADMs) provides a

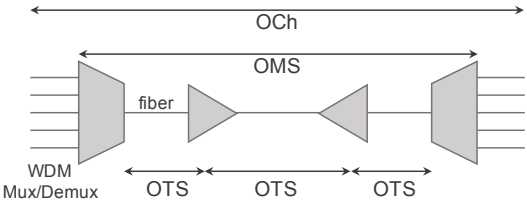


Figure 2.3: Optical Transport Hierarchy (OTN).

wavelength-multiplexed path, the OMS. Finally, the end-to-end optical channel (OCh) provides a transparent circuit, which allows to encapsulate higher layer frames.

SDH/SONET. Based on TDM, *Synchronous Digital Hierarchy* (SDH) (in Europe) and *Synchronous Optical Networking* (SONET) (in the US) provides virtual circuits. Incoming data from higher layers is encapsulated, multiplexed using TDM and sent either directly over optical fiber or encapsulated in OCh.

Carrier Ethernet. Following the success of Ethernet [39] in LAN environments, Carrier Ethernet was created as a cost-effective transport while meeting carrier requirements. This includes the integration with other carrier techniques like SDH/SONET, which allows to establish point-to-point Ethernet connections. *Carrier Ethernet* (CE) applies statistical multiplexing.

MPLS/IP. The most expensive layer is the *Multi-Protocol Label Switching* (MPLS)/*Internet Protocol* (IP) layer, where statistical multiplexing is done based on labels attached to packets. Incoming packets are inspected for their destination IP address and marked with an MPLS label at the *Label Edge Router* (LER). This expensive operation, in terms of routing table lookup, results in benefits inside the network, where *Label Switch Router* (LSR) can perform less expensive label switching. This process requires inspection of every packet and large routing/forwarding tables made of

expensive memory. Still, the cost per bit as well as the power consumption is way higher compared to the optical techniques [40], yet, it offers more flexibility.

Aim of introducing the most common multi-layer techniques was to give an impression of the parameter choice that exists when setting up a network infrastructure. Based on optical fiber, higher layers realize virtual paths between distant sites that appear only one hop away. The choice of technology starts with translucent optical wavelength switching, which can be realized very efficiently at line rate, but limits flexibility, as the incoming wavelength needs to match the outgoing one. In contrast, MPLS/IP routing requires large amounts of electrical power to take decisions about the path every single packet should take through the network. Its benefit, however, is that it offers the biggest degree of flexibility by applying statistical multiplexing and making use of the foundation offered by the underlying layers.

2.1.3 Planning of Multi-Layer Networks

The topic of multi-layer network design is about defining the equipment required to set up a network consisting of multiple layers, i.e. different optical multiplex layers and MPLS/IP on top. Based on the traffic demands in the network, a multitude of possible hardware and routing decisions have to taken into account for computing the optimal configuration. Further, means to cope with network failures have to be established. This makes the network planning process such a complex task. In the following, particular objectives that increase the parameter space for realizing a multi-layer network will be discussed in detail.

Input Parameters. The network planning process can be based the following input data, which can be specified by the network operator:

a) an optical *fiber topology* G_0

b) a *component list* of devices including their possible extensions (e.g. how many line cards or optical transceivers can be plugged in)

c) a *Capex model* for all these devices and their accessories

d) an *Opex model* including costs for energy, housing, maintenance etc.

e) an end-to-end *demand matrix* that dictates for what capacities the network should be dimensioned

f) information about the required availability of the network resp. what kind of failures have to be protected.

Objective Function. Not only are the input parameters multidimensional, but also the objective function that determines, what equipment and path choices are good or bad. Further, some of the goals are competing. Exemplary contributors to this function include:

a) Capex costs, i.e., the one-time expenditures to buy and set up all equipment.

b) Opex costs, i.e., the ongoing costs for energy, employee's wages, license fees and so on.

c) Path lengths should be minimized so that transferred data takes as little detour as possible. Long paths not only occupy capacity on multiple links and nodes, but also increase its failure probability.

d) Routed/protected demands vs. *Service Level Agreements* (SLAs), i.e., while it should be the goal of the network operator to fulfill all demands, as well as to establish means for keeping connectivity also in case of multiple failures, it might be more economical to pay a customer SLA violation fees compared to preparing for every unlikely failure case.

Including all of these aspects into the calculation is hardly possible, as, e.g., only few (publicly available) Capex and especially Opex models exist. Still, this huge parameter space makes multi-layer planning a complex optimization problem. This leads to approximations and application of different optimization techniques, including *Mixed Integer Linear Programs* (MILPs) [41] and heuristics [42].

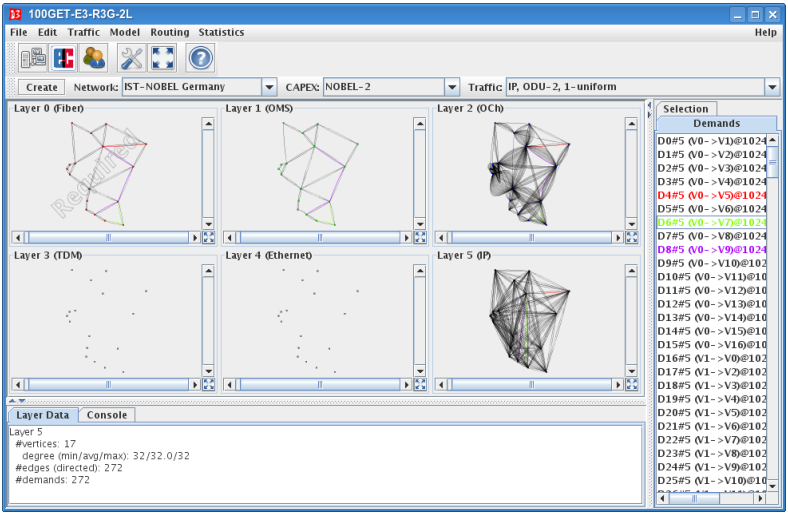


Figure 2.4: Screenshot of the MuLaNEO network planning software, taken from [44].

An example of such a network planning software is MuLaNEO (Multi-Layer Network Engineering and Optimization, [43]). This open-source network planning software originating from the 100GET research project allows to plan a multi-layer network based on a Capex model and an end-to-end traffic demand matrix. MuLaNEO's user interface is shown in Figure 2.4, which depicts the realization on the different layers.

2.1.4 Resilience Aspects of Multi-Layer Networks

One complex aspect that needs to be incorporated in the network planning process is resilience.

The state that a network infrastructure correctly provides connectivity to its users can be harmed by numerous factors. This includes human errors by, e.g., network engineers that deploy a faulty configuration leading to outages, elec-

tric defects of network equipment, or physical damage, when cables are cut by diggers.

Resilience Objectives

In relation to the resilience of (wide area) networks, the following three objectives play an important role:

Availability accounts for the time share, during which a network or network component is working correctly and provides its service. This time share is usually measured in percentage over a time span per year and often specified in the SLAs of service providers. Typical availability goals of critical infrastructure range from 99.99 to 99.999% per year, resulting in a maximum unavailability of 53 to 5 minutes per year. Given this notation, the terms "four nines" / "five nines" availability is also common.

Lost Traffic allows to better describe the operational state of a network and better account for the connectivity between particular endpoints. Traffic is *lost*, if a traffic demand between two points in the network suffers connectivity issues caused by any kind of failure. The bandwidth amount of this traffic demand then counts as lost traffic either as an absolute bandwidth measure or relative as share of the total network capacity.

Robustness represents the networks ability to prevent the occurrence of lost traffic during certain kinds of failures, e.g., single or double node or link failure scenarios. Such robustness against failures also in the case of infrastructure failures can be achieved by protection mechanisms.

Protection Mechanisms

In order to allow the network to overcome failures of links or nodes within a very short amount of time, the ability to provide backup paths that act as protection against failures on the primary path, is also incorporated in the network planning process. Such pre-established paths enable mechanisms for fast failover

mechanisms like *IP Fast Reroute* (IPFRR) [45] that allow to meet carrier-grade requirements, i.e., failover times in the order of tens of milliseconds.

In the following, a brief overview over the most important protection and recovery mechanisms will be given, while more information can be found in [46]. Although only protection mechanisms against link failures will be described in the following, the same also holds for node failures, i.e., if routers or *Reconfigurable Optical Add-Drop Multiplexers* (ROADMs) are broken or become unavailable as a result of environmental influences like loss of power, earth quakes etc.

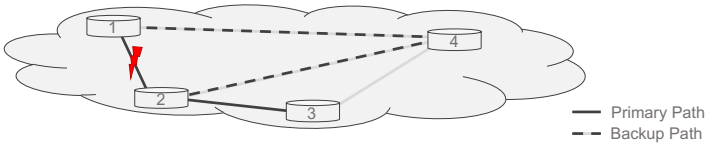
Recovery Extents. Figure 2.5 illustrates two approaches to protect the path $p_{1,3}$ between nodes 1 and 3 regarding the failure scenario that link $e_{1,2}$ between nodes 1 and 2 fails. The two approaches are local and global recovery and straight black lines denote the primary path, while the dashed lines denote the backup path.

a) In the case of *local recovery*, shown in Figure 2.5a, the backup path $\{e_{1,4}, e_{4,2}\}$ is realized via a path disjoint to the link failing in the observed failure scenario. After overcoming the defect link, the rest of the primary path is used.

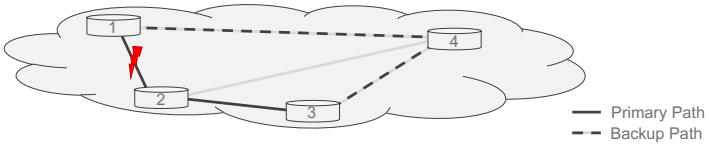
b) An alternative approach, namely to use a path that is disjoint to the complete primary path is *global recovery* and illustrated in Figure 2.5b. In case of this protection scheme, the link $e_{2,3}$ is not included in the backup path, although not failing.

Protection Schemes. Protection schemes denote, how the recovery extents are used to protect the primary path, i.e., if exclusively or shared and resulting in what degree of resource wastage.

a) Using *1+1 Dedicated Protection*, all traffic is simultaneously transferred using both paths, similar to *Redundant Array of Independent Disks* (RAID) in the storage area. At the receiver side, the data received the fastest or with the best quality is used. While this results in very short failover times, the complete duplication of all traffic results in a waste of resources.



(a) Local recovery.



(b) Global recovery.

Figure 2.5: Recovery Extents.

b) More efficient protection – in terms of resource utilization – can be realized using *1:1 Dedicated Protection with Extra Traffic*, where every backup path again protects exactly one primary path, but still can be used in failure-free times. Once a failure occurs, the traffic of the protected primary path is redirected using the backup path, optionally with a higher priority than other traffic on that link.

c) *M:N Protection* with $M \leq N$ specifies a set of M paths to protect the N primary paths, while the N primary and M backup paths do not share links, i.e., are part of different *Shared Risk Groups* (SRGs). This protection scheme exploits the fact that simultaneous failures on multiple paths are very unlikely and thus reserving full capacity for every path on the backup links is a waste of resources.

What all of these protection schemes have in common is that primary and backup paths must not fail at the same time, i.e., not be part of the same SRG [47]. While it sounds intuitively trivial that both primary and backup path must not be affected by a single failure, ensuring this in multi-layer networks is more complex. Lower layers might hide the “reality” so that

while two paths seem disjoint, they might be sharing the same fiber on the lowest layer and thus would both be affected by a cable cut.

2.1.5 Energy Efficiency of Core Networks

As previously seen, the consolidation of multiple backup links leads to more efficient resource usage. The resulting savings not only include Capex savings by reducing the equipment to buy, but also Opex in form of fewer network equipment being active and consuming energy. The contributors to the energy consumption of WANs include cooling systems, power consumption of network devices including their line cards and optical transceivers, as well as optical refreshers that need to be placed every 50-80 km on a fiber link.

Before Section 2.2 introduces a strategy for planning energy efficient networks, an overview over related work will be given in the following.

Access networks are currently dominating the power consumption of the Internet [48]. With increasing access speeds, the core network's capacity and thus also the power consumed in the core network is expected to increase continuously. In 2017, the power consumption of the Internet core is expected to exceed the power consumption of access networks [49]. This results in higher operational expenditures for large ISPs, if the network stays unchanged in terms of its energy-awareness.

A survey on green networking [50] categorizes research work into four categories: *Adaptive Link Rate*, *Interface Proxying*, *Energy Aware Infrastructure* and *Energy Aware Applications*. Our work touches the first category Adaptive Link Rate, with sleep modes and rate switching of network interfaces as well as the category of Energy Aware Infrastructure.

As long as the current network utilization allows, rerouting of some demands along alternative paths allows to temporarily disable some links. That already a few optimizations per day (three in the example) allow 10% energy savings while making use of stand-by modes is shown in [51]. In contrast, dynamic adaptation of the rate speed is more complex, not only from technical side, but also from network planning effort. Into a similar direction goes [52], in which the authors state

that 25% energy can be saved, if primary paths are aggregated over fewer links and links exclusively used for backup paths are switched into stand-by mode.

Energy efficiency in IP-over-WDM networks is observed in [53] and three mechanisms are suggested: *Fixed Upper Fixed Lower* (FUFL), *Dynamic Upper Fixed Lower* (DUFL) and *Dynamic Upper Dynamic Lower* (DUDL), which determine the freedom of dynamically rerouting traffic either on lower (the WDM) or upper (the IP) layer. After traffic aggregation through rerouting, unused line cards can be switched off. The authors mention that dynamics, esp. in the lower layer where a coordinated reconfiguration of optical cross-connects has to be done, are technically more challenging than rerouting to parallel paths, as suggested with FUFL. Nevertheless, the authors report energy savings in a simulation based on day/night traffic demands also for the simple FUFL scenario, while bigger savings are reported for the DUFL approach, where rerouting takes place on IP layer. This MILP-based network planning approach requires abstractions, and thus differs from this work, in which no such abstractions are made, as the design process and software for the network is not touched, but stays in the way it was before.

2.2 Energy Efficient Network Planning

This part describes an approach for energy efficient network planning based on a legacy planning software, which does not care about energy consumption by itself. We evaluate the possible influence of the network planning process towards reducing energy consumption of optical multi-layer core networks. In particular, we propose to remove redundant links in the network, and to route corresponding network traffic via other links. Based on the reduced network topology, we compute the required network equipment for realistic traffic demands using a network planning tool. Due to the lack of an accurate model for operational expenditures and energy consumption we choose the link length as cost function. We show the applicability of our idea and demonstrate the energy saving potential using realistic network topologies.

As described in Section 2.1.3, planning of multi-layer core networks can become very complex. This is due to the sheer amount of possible realizations for

routing specific traffic demands using the different available layers. Hence, the planning process is crucial for operators, since on the one hand a smooth and resilient operation of the network has to be assured, and on the other hand the costs for the network equipment have to be minimized.

While the Capex for network equipment can be estimated quite accurately, calculating Opex is complex. Besides the rent for the buildings where equipment is located or for the dark fiber, Opex also include the wages for staff managing the equipment, and finally, costs for the power consumed by the equipment. Due to its complexity, network planning software is highly specialized and often solely focuses on the Capex calculation [42].

By reducing the number and overall length of fiber links, the network's energy consumption and thus Opex can be reduced. Additionally, reducing the number of the active fiber connections not only avoids energy costs, but also lowers the rent for a fiber infrastructure, if not owned by the operator. By including these factors already in the planning process, the network's utilization can be increased through aggregation of the traffic to fewer links and avoiding unnecessary equipment not only being bought, but also consuming energy [54].

Therefore, the approach described in the following is to reduce energy consumption by reducing the network topologies. From a given fiber topology, edges are removed before starting the network planning process. Such a removed edge is later not used in the network and no energy is consumed for line/port cards in routers, multiplexers, as well as amplifiers that are usually placed every 50 km. In order to guarantee resilience in case of an outage, it is assured that each site is still connected via two links at least. Due to the lack of publicly available Opex models for network equipment, the link lengths are chosen as cost function. Based on this cost function, the effect of the proposed mechanisms on the Opex are evaluated and compared with the original topology.

In the following section, the problem formulation as well as the metrics applied to network topologies are given in Section 2.2.1, followed by Section 2.2.2 explaining the used MILPs. Results of applying the mechanisms to realistic network topologies are shown in Section 2.3.

2.2.1 Modeling

This section first elaborates on properties of network topologies and metrics that influence the performance and reliability of a network. Afterwards, the four observed types of minimal subgraphs of a network topology graph are presented.

Network Topology Characteristics

We count the following metrics and properties as important, when characteristics of network topology graphs are evaluated:

Opex Costs. The number of active links and their total length result in energy and thus Opex costs. Here in this model, we set the Opex costs for a link proportional to the edge length in the network graph as an approximation. If an edge is removed from the topology graph, the fiber link is not used and thus no energy consumed (neither on the way through amplifiers, nor at the endpoints through interface cards).

k-connectedness. In order to ensure fault-resilience in case of link or node failures, disjoint backup paths have to be available. In this work, a value of $k = 2$ is used, which means that two separate paths between any node pair are required, which later enables the design software to establish two disjoint paths.

Network Diameter. The network diameter denotes the longest of all shortest paths between any two node pairs in the network.

Network Configuration Characteristics

A network planning or network design software computes a solution for a configuration given the input of a topology, a component list and end-to-end traffic demands. The following metrics are of special interest for the following evaluation.

Routed Demands. For every source/destination pair $(u, v) \in V \times V$ there exists a traffic demand D_{uv} with a certain bandwidth requirement. These

demands are planned by the design software and later established either as MPLS paths or on lower layers, e.g. translucent optical paths. If no such path can be established due to capacity shortage, the demand is seen as *not routed*.

Protected Demands (Protection). Besides a primary path, also a backup path has to be planned and later established for every demand. While two-connectedness of a network graph is required for disjoint primary and backup paths, two-connectedness is not a sufficient criteria, as the capacity of the equipment can be exceeded. The share of protected demands out of all demands results in the degree of *protection*.

Accepted Solution. This true/false criteria sums up the degree of routed and protected demands. A planned solution (the result of the network planning process) is accepted only if a degree of 100% protection is reached, which implies that all demands are routed. A solution is not an accepted solution, if the degree of protection is lower than 100%. In practice, an operator might choose to not reach full protection, which however is an economical aspect and thus not further covered.

Suggested Algorithms

The goal of reducing the energy consumption under the premise that the planning software is able to find an accepted solution is approached from two sides:

ADD. Starting with a reduced topology graph, edges are added, until an accepted solution is found. The majority of this study will focus on different subgraphs, which provide a good starting point for such planning.

DEL. Starting with the original topology (G_{ORIG}), as many edges as possible are removed, as long as the solution remains acceptable.

Algorithm 1: The *DEL* Algorithm.

Input: Full fiber topology $G = (V, E)$
Set $E' =$ orderByLengthDecr(E);
Set $E_i = \emptyset$;
for $e \in E'$ **do**
  $C =$ callPlanningSoftware($(V, E - E_i)$);
  if *protection(C) == 100%* **then**
    $E_i = E_i \cup \{e\}$;
  end
end
return $(V, E - E_i)$

Removing Edges (DEL)

The *DEL* algorithm is given in Algorithm 1 and starts with the full fiber topology. The topology given to the design software as input is modified in each iteration and the longest active and not yet examined edge of the topology is removed prior to starting the planning. From a topology (V, E), E' denotes the list of edges ordered descending by their length. In every step of the algorithm, the first (longest) edge $e \in E'$ is moved to the set of inactive edges E_i. If the planning software returns an accepted solution based on the reduced topology $(V, E - E_i)$, then e is kept within E_i. In case that not all demands can be protected, e is removed from E_i again (and will be active in the final topology). The process ends after all edges were examined once ($E' = \emptyset$).

By checking the two-connectedness of the topology, the DEL algorithm can be accelerated, as planning runs that certainly cannot end up with full protection can be avoided, if the removal of an edge e leads to a one-connected topology.

Adding Edges (ADD)

The *ADD* algorithm starts from the exact opposite direction and is listed in Algorithm 2. The planning software is invoked with a reduced topology graph (V, E') with $E' \subset E$. Suggested algorithms for creating these subgraphs are described in the next sections. In case that the planning software is not able to compute a

Algorithm 2: The *ADD* Algorithm.

Input: Reduced fiber topology $G' = (V, E')$ with $E' \subset E$

```
// plan the network with the input topology
Set C' = callPlanningSoftware((V, E'));
while protection(C') < 100% AND E' ≠ E do
    for e ∈ (E − E') do
        // add one edge to the topology
        E'' = E' ∪ {e};
        C'' = callPlanningSoftware((V, E''));
        if protection(C'') > protection(C') then
            // use this as new best solution C' = C'';
            E' = E'';
        end
    end
end
return (V, E')
```

configuration C that is not seen as an accepted solution, the following algorithm is executed: For every edge $e \in (E - E')$, an extended subgraph $(V, E' \cup \{e\})$ is given as input parameter to the planning software, which is then executed $|E - E'|$ times. The edge e that increased the degree of protection the most is kept for the further planning and added to E'. This is repeated as long as the result of the planning software is not yet giving full protection, thus not an accepted solution.

A trivial and quicker alternative instead of planning the network for every potential edge e would be to pick the shortest edge, thus the configuration that leads to the smallest increase in Opex. However, tests have shown that this quickly leads to most or all links being set to active and therefor was not progressed any further.

The following reduced topologies are subgraphs of the original topology G_{ORIG} in Figure 2.6. All proposed subgraphs have the aim to fulfill one or more of the characteristics of well-suited topologies and thus enabling the planning software to create a solution fulfilling the characteristics described previously.

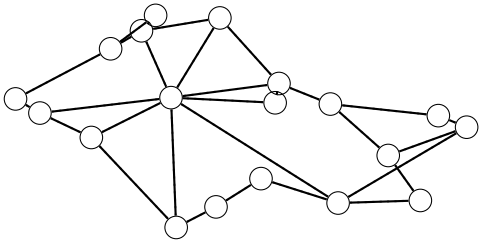


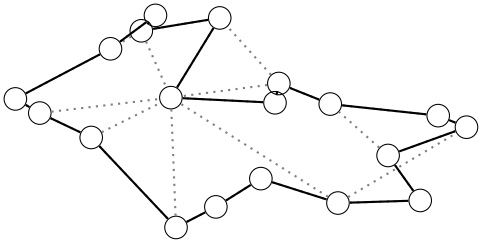
Figure 2.6: Original topology (G_{ORIG}).

Minimum Spanning Subgraph, two-connected (G_{MSS}).
The first subgraph, the minimum spanning subgraph, is a two-connected graph connecting all nodes with each other with the minimal total edge length. The result is shown in Figure 2.7a, where the dotted links of the original topology are deactivated, while the solid black ones stay active. The number of active links in a subgraph G_{MSS} of a graph $G = (V, E)$ is $|V|$.

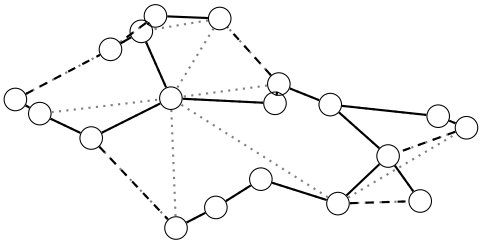
The resulting subgraph has tendencies to form one or more rings. While a ring topology is common for networks, the downside of this approach is that the paths between two nodes can grow very large, especially in case of a single link failure.

Minimum Spanning Tree, two-connected (G_{MST^+}). To counter the limitations of the G_{MSS} and to decrease the primary path lengths, all nodes are first connected using a *Minimum Spanning Tree* (MST). Afterwards, two-connectedness is ensured in the same way as with the G_{MSS}, namely by adding the edges resulting in the shortest total length. Figure 2.7b illustrates this with solid black lines showing the MST and the dashed lines being the edges added for two-connectedness. The dotted out edges denote edges of G_{ORIG} that are deactivated in the example for this subgraph.

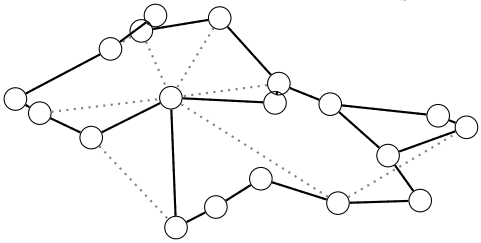
The availability of more links in the topology in comparison to G_{MSS} leads to shorter paths, as more direct paths are available in most topology



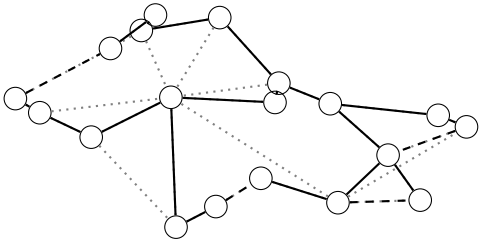
(a) Minimum Spanning Subgraph, two-connected (G_{MSS}).



(b) Minimum Spanning Tree, two-connected (G_{MST+}).



(c) Minimum Diameter Subgraph, two-connected (G_{MDS}).



(d) Minimum Diameter Tree, two-connected (G_{MDT+}).

Figure 2.7: Reduced network topologies.

types. However, due to the additional edges added, the total length of active edges, and thus the Opex costs, are increased.

Minimum Diameter Subgraph, two-connected (G_{MDS}). As long paths traversing multiple hops can result in diminished network performance, the objective of the G_{MDS} is to lower the network diameter, the longest shortest path between any two nodes in a network.

The G_{MDS} is illustrated in Figure 2.7c. Like the G_{MSS}, it has a tendency to form large rings caused by the two-connectedness criteria. However, not the shortest edges are picked, but instead the edges leading to minimal network diameter, in order to achieve a trade-off between low Opex costs and diminished network performance.

Minimum Diameter Tree, two-connected (G_{MDT^+}). As last suggestion for energy efficient subgraphs, we introduce the two-connected Minimum Diameter Tree. Similar to the G_{MST^+}, the initially calculated minimum diameter tree minimizes the distance between nodes for primary paths, while still forming a tree, thus keeping the Opex costs low. Routing of additional backup paths is enabled through additional links being added to make the topology two-connected with the measure of further shrinking the paths between nodes.

So the results suggest the G_{MDT^+} as a combination of the G_{MST^+} and the G_{MSS}. Combining a prioritization for primary paths over backup paths, saving energy due to a reduced number of active links (respectively shorter links), as well as avoiding degradation of network performance by avoiding increased network diameters as well as possible.

Figure 2.7d illustrates the G_{MDT^+} with the minimum diameter tree in solid black and the additional edges added for two-connectedness in dashed black.

Starting from these pre-computed subgraphs, the *ADD* algorithm adds additional edges and starts the planning software, until full protection can be reached. How this iterative process looks like will be illustrated in the following.

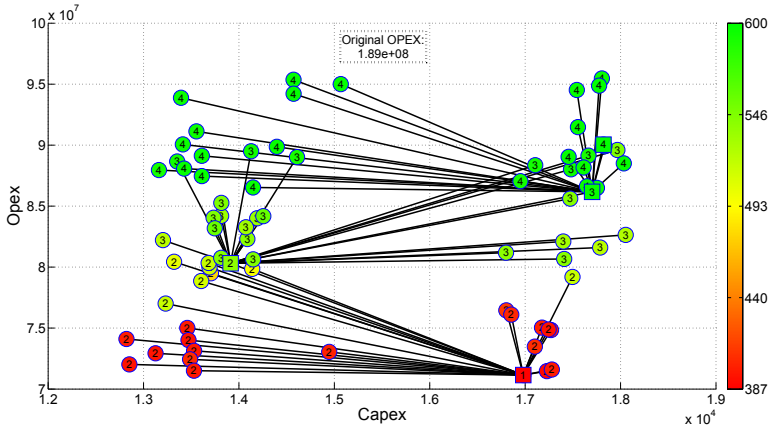


Figure 2.8: Iterative steps of the ADD algorithm for the NTT topology starting from the G_{MDS} subgraph.

Iterative Steps of the ADD Algorithm

Figure 2.8 shows the evolution of an exemplary run of the *ADD* algorithm applied to the G_{MDS} subgraph of the NTT network topology. The algorithm starts in step [1] on the lower right side, where the red color indicates a low number of protected demands. All network configurations computed during a run of the planning software in step i are denoted by (i) (with $i-1$ edges added), while the markers [i] denote the solution that is selected as best one.

After adding one edge, the solution [2] leads to an improved solution, yet still lacks full protection denoted by the yellow-ish color. After adding two more links to the topology, which are added in steps (3) and (4), solution [4] provides is selected by the algorithm.

2.2.2 Mathematical Definitions

Network Flow Model

A *Network Flow Model* [55] can be used to model traffic flow and capacity in communication networks using constraints and an objective function, forming a MILP. In order to find the optimum solution for such an objective function, the solver Cplex [56] is used to compute the energy efficient subgraphs of the given topologies.

Such traffic flows can be seen as a flow with a certain capacity from a source to a destination, potentially passing intermediate nodes. Like with a flow of water in a pipe, a network link carrying reserved bandwidth demands can only be "filled" until its maximum capacity.

In the following, the variables and constraints used by all of the introduced subgraphs are explained. These are the most basic constraints which ensure common requirements, like two-connectedness for disjoint backup paths.

Input Data: *a*) Graph $G = (V, E)$ consisting out of a set of vertices V and a set of edges E.

b) k-connectedness ($k = 2$). Two-connectedness is used to require two disjoint paths between any two nodes.

c) $weight_{uv}$: weight of the edge from u to v. The weight is denoted by the geographical distance between u and v.

Variables: *a*) x_{uv}: 1, if the edge from u to v is used, 0 otherwise.

b) f_{uv}^{st}: 1 if the flow between s and t is using the edge uv, 0 otherwise. Note that we are using integral flows.

Constraints: We pick an arbitrary set $S \subseteq V$ of cardinality k and impose for any node pair $s \in S$ and $t \in V$ the following constraints. These constraints ensure that there are at least k node-disjoint paths between s and t. Note that this suffices to ensure k node-disjoints paths between any node pair $s, t \in V$.

a) k**-connectedness.** An integral flow f^{st} between any two vertices s and t symbolizes the traffic demand from s to t. This constraint ensures that at least k units of flow are sent to vertex t - or in other words at least k paths reach the destination node. Setting $k = 2$ ensures the possibility of reserving disjoint backup paths by ensuring the two-connectedness of the resulting subgraph

$$\sum_{vt \in E} f_{vt}^{st} - \sum_{tw \in E} f_{tw}^{st} \geq k. \tag{2.1}$$

b) **Flow conservation.** Every flow that flows into a vertex v also has to completely flow out of v except for the $v \in \{s, t\}$

$$\sum_{uv \in E} f_{uv}^{st} - \sum_{vw \in E} f_{vw}^{st} = 0 \quad \forall v \in V - \{s, t\}\,. \tag{2.2}$$

c) **Vertex integrity.** Every vertex can be used only once by the flow f^{st}. This ensures that node-disjoint primary and backup paths are computed

$$\sum_{uv \in E} f_{uv}^{st} \leq 1 \quad \forall v \in V - \{s, t\}\,. \tag{2.3}$$

d) **Flows and Edges.** If there is a flow using an edge uv, then this edge has to be used

$$0 \leq f_{uv}^{st} \leq x_{uv} \quad \forall uv \in E. \tag{2.4}$$

Minimum Spanning Subgraph, two-connected (G_{MSS})

The constraints describe until now are already sufficient to let the MILP construct the first and most intuitive subgraph. Starting from the shortest spanning graph, the MILP adds additional edges to ensures all nodes are two-connected. In order to minimize the overall length of active links (edge weight in the MILP), the following objective function is defined

$$\min \sum_{uv \in E} weight_{uv} \cdot x_{uv}. \tag{2.5}$$

Minimum Spanning Tree, two-connected (G_{MST+})

As described in Section 2.2.1, first a minimum spanning tree is constructed, e.g., by using Kruskal's algorithm. Compared to the MILP of G_{MSS}, the variable x_{uv} is set to 1 for all edges uv that are contained in the minimal spanning tree to enforce them being available.

Minimum Diameter Subgraph, two-connected (G_{MDS})

For this problem, we use the same constraints and variables as the G_{MST+}. For any two distinct nodes s, t we additionally introduce two unit flows f'^{st}, f''^{st} modeling the primary and the backup path, respectively. Both flows satisfy all flow constraints described previously for $k = 1$.

The flow f^{st} models the sum of the two flows f'^{st} and f''^{st}

$$f'^{st}_{uv} + f''_{uv} = f^{st}_{uv} \quad \forall uv \in E. \tag{2.6}$$

The variable z is an upper bound on the lengths of primary and the backup path, i.e. it models the diameter

$$\sum_{uv \in E} f'^{st}_{uv} \cdot weight_{uv} \leq z, \tag{2.7}$$

$$\sum_{uv \in E} f''^{st}_{uv} \cdot weight_{uv} \leq z. \tag{2.8}$$

Forcing a Low-Diameter Spanning Tree: For each edge $e \in E$ we introduce a binary variable $x'_e \in \{0, 1\}$ indicating whether this edge is used by a primary path or not. This can be ensured by imposing the constraint (Equation 2.4) for the primary flow f' and x'. We also add a constraint to ensure that the subgraph

spanned by the primary flow forms a tree

$$\sum_{uv \in E} x'_{uv} \leq |V| - 1. \tag{2.9}$$

Objective Function: Minimize a linear combination of the diameter and the total length of the network

$$\min z + \sum_{uv \in E} x_{uv} \cdot weight_{uv}. \tag{2.10}$$

Minimum Diameter Tree, two-connected (G_{MDT^+})

The same constraints, variables and objective function as in the previous section are used. Additionally, a minimum diameter spanning tree and the $x_e = x'_e = 1$ for all edges e of this tree are computed.

2.3 Evaluation

Prior to evaluating the results of applying the *ADD* and *DEL* algorithms, we start with an evaluation of the effectiveness of the constructed subgraphs that serve as the starting point for the *ADD* algorithm. To evaluate the effects of providing the reduced topologies as input parameter to a network planning software, we reduce five realistic network topologies prior to supplying it to a network planning software according to the four minimal subgraphs presented in Section 14.

The evaluated networks are listed in Table 2.1. As resilience mechanisms require two-connected topologies, the original topologies are modified by removing the one-connected nodes so that the observed topologies are fully two-connected. The column *Size* of Table 2.1 lists the number of nodes ($|V|$) and edges ($|E|$) of the original topology, as well as the number of one-connected nodes being removed ($|V_A|$).

As exemplary network planning software MuLaNEO (Multi-Layer Network Engineering and Optimization) [43] is used for this evaluation. Based on a sup-

Network name and date		Size			Degree d		Geo
		$\|V\|$	$\|E\|$	$\|V_A\|$	avg.	max.	location
Commercial Network Topologies from [57].							
China Telecom	2010/08	20	88	18	4.40	14	CH
NTT	2011/03	25	112	22	4.48	11	Global
Research and Education Network Topologies from [57].							
CESNET	2010/06	19	60	26	3.16	8	CZ
GARR	2010/12	22	72	22	3.27	8	IT
Rediris	2011/03	18	60	1	3.33	10	ES

Table 2.1: Networks under study.

plied fiber topology and the Capex model from the IST NOBEL project [40], MuLaNEO plans the network.

Precomputed Subgraphs

Based on the four subgraphs G_{MSS}, G_{MST^+}, $G_{MDS}$and G_{MDT^+}, we evaluate the success of MuLaNEO in computing an accepted solution without applying the ADD algorithm.

Table 2.2 lists the intermediate results for the five networks. It can be seen that, compared to the original topology G_{ORIG}, Opex costs are reduced for all topologies with all subgraph types, as with each of them at least one edge is removed. However, in most cases, not all demands can be protected using disjoint backup paths. Therefor, except for the China Telecom topology, the returned solutions are almost never acceptable.

However, the cases when an accepted solution can be computed, Opex savings of 9 to 43% can be achieved compared to the original topologies G_{ORIG}.

		CESNET	China T.	GARR	NTT	Rediris
G_{ORIG}	Prot.	100,00%	100,00%	100,00%	100,00%	100,00%
	Opex	100,00%	100,00%	100,00%	100,00%	100,00%
G_{MSS}	Prot.	70,76%	100,00%	99,78%	66,17%	99,67%
	Opex	52,72%	56,59%	79,13%	37,47%	66,43%
G_{MST^+}	Prot.	98,83%	100,00%	100,00%	96,50%	99,67%
	Opex	66,64%	59,27%	91,11%	45,39%	66,43%
G_{MDS}	Prot.	98,54%	100,00%	99,35%	64,50%	99,67%
	Opex	59,65%	56,59%	85,15%	37,67%	66,43%
G_{MDT^+}	Prot.	100,00%	100,00%	100,00%	99,83%	99,67%
	Opex	75,05%	67,40%	87,290%	60,90%	67,71%

Table 2.2: Opex costs relative to G_{ORIG} and degree of protection (configurations not acceptable marked red).

Evaluations of Accepted Solutions

After evaluating the ability for a planning software to compute solutions based on the pre-computed subgraphs from Cplex, we apply the proposed methods ADD and DEL from Section 2.2.1. Using DEL, the original graph is reduced step by step. Using ADD, the minimal subgraphs are extended with additional edges until full protection is achieved. The results are shown in Figure 2.9.

It can be observed that all mechanisms lower the resulting Opex costs. The savings for the GARR topology (red) are with 9-18% already notable, although very low compared to NTT (yellow), which shows potential savings between 39 and 54% due to the high node degrees and thus big potential that links can stay unused. Regarding the resulting Capex, for most of the topologies (except Rediris and GARR based on G_{MDT^+}) an increase can be observed compared to

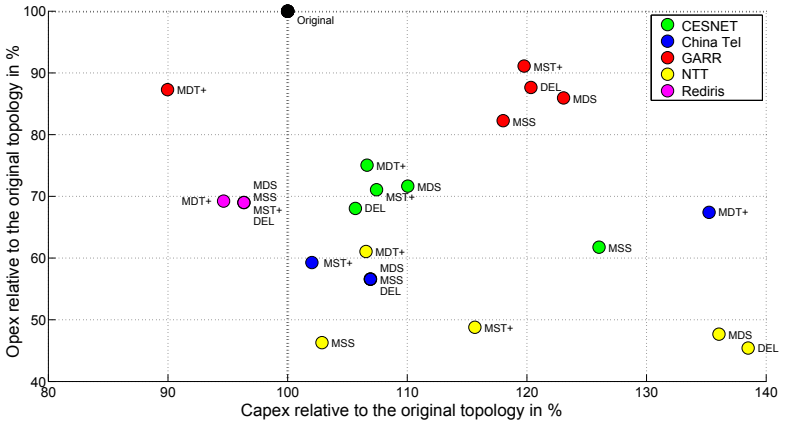


Figure 2.9: Comparison of reduced topologies with original one.

the original topologies. In these cases, aggregation of traffic requires more expensive equipment for the higher data rates. However, these value are results of the underlying Capex model and not the primary objective of this study. Furthermore, depending on the absolute Opex and Capex costs, this might be acceptable for operators, as higher initial costs can be returned through big savings during operation.

Furthermore, it can be seen that neither the DEL method, nor subgraphs enhanced with the ADD method, show clear advantages for any of the mechanisms. The (dis)advantages of the different subgraph types were previously explained, so that in cases like with the NTT topology, where the simplistic G_{MSS} results in lower Capex and Opex costs, this graph type might require closer inspection and comparison with the results of more expensive solutions like G_{MDT^+}, which tends to result in better network performance through shorter paths.

2.4 Lessons Learned

Multi-layer network planning is complex due to the large parameter space including technology choices of equipment, protocols, and protection mechanisms, as well as operator policies. The lack of comprehensive cost models, esp. for operational expenditures, makes it hard to draw universally valid conclusion from academic research perspective.

This first chapter suggested methods for reducing Opex costs during the planning process through increased energy efficiency. The link length was chosen as cost metric as an approximation. Link length contributes to the overall power consumption, as signal refreshers need to be placed around every 50-80 km link length.

The presented mechanisms avoid a modification of the network planning software and reduce the complexity of the planning software runs, as the size of the network topology used as input parameter is reduced.

By reducing the network topology, the traffic is aggregated to fewer links, as some of the links are removed and thus not available for the planning software.

The derived results indicate Opex savings between 9% and 54%. Depending on the applied methods and the observed topology, the Capex costs ranged between 90% and 140% compared to the original, unmodified topology. Furthermore, while not related to energy savings, unnecessary rent for dark fiber can be avoided, if the network topology still allows equipment to be set up that is able to route and protect all traffic demands.

A definite conclusion, which of the suggested subgraphs minimizes the energy consumption is not possible. Given the fact that the subgraphs show different characteristics regarding performance and resilience metrics, a closer inspection of the resulting topology is required for a particular network topology. However, it can be seen that the modification always leads to a reduction of the operational expenditures for the given topologies.

3 Management of Softwarized Networks

Software-based networks relocate the network control plane into, mostly centralized, software running on standard servers instead of decentralized directly on the network devices. As this software can be developed independent of the network device hardware and firmware, changes to the network can be conducted without independent of the device vendor or hardware replacements.

Such *Software Defined Networks* (SDNs) are nowadays typically found in the data center environments of the Facebooks and Googles. The amounts of data that these companies process, the scale at which they operate at and the need to innovate rapidly forced them to search for better solutions than traditional networking devices offer. For being flexible and able to tailor the network infrastructure to their special needs, they exploit the possibilities provided by softwareized networks, such as programmability and centralized control. These data centers are geographically limited regions and operated by a single institution, which is the owner of computing and networking infrastructure as well as of applications.

One real-world example outside of data centers is Google's *Wide Area Network* (WAN) backbone "B4" [25]. By using SDN and incorporation information from applications running inside their network, the link utilizations could be increased to around 95% on average. By using a custom controller software, Google is able to define and adapt their switches' forwarding behavior so that traffic steering and prioritization happens according to their own needs. A rule of thumb is that the development cycles of hardware devices are with around 18 months on average three times slower than those of software with around 6 months. Again, Google is able to roll out such softwarized networks, as thy own operate the complete

infrastructure including the fiber backbone.

However, an SDN-based network is not necessarily "better" than a traditional one, where every device decides about packet forwarding on its own. Through automation, the management of softwarized networks can be better operated and at a higher scale, similar to what has happened with applications running in the cloud. In contrast to traditional devices that mostly offer very vendor-specific interfaces, such as command line interfaces, SDN follows the API-driven cloud principle. These open, vendor-agnostic interfaces [27] allow a separation of concerns, easy replacement of particular building blocks, as well as interaction with other entities. Finally, this essentially allows everybody, from researchers to integrators and operators, to customize and modify the network behavior on a level that was previously only accessible to device vendors.

The remainder of this chapter is structured as follows: Section 3.1 introduces the most important aspects of softwarized networks and gives an overview over related work on this topic. During the past years, IT operations underwent major changes by adopting methods from software engineering, which lead to infrastructure being defined by code. As IT and network operations are merging through SDN, Section 3.2 describes how such methods can be adopted for the testing and deployment process in a softwarized networking environment. The usage of SDN features for network monitoring will be described in Section 3.3. The described concept will use the integrated byte counters offered by switches to detect the flows transferring most data in a network. Section 3.4 provides an example for service and network separation in a "Bring Your Own Device" scenario. Through the usage of an application running inside the SDN controller a more fine-grained and secure isolation is established. Finally, Section 3.5 describes the lessons learned.

3.1 Background and Related Work

This section continues with describing the topic Software Defined Networking. Further, relevant aspects from the field of software engineering will be introduced. Finally, previous work on the topic of network monitoring is described.

3.1.1 Software Defined Networking (SDN)

In the following, the SDN concept will be described in detail, as this technique is a foundation for most studies described in the following.

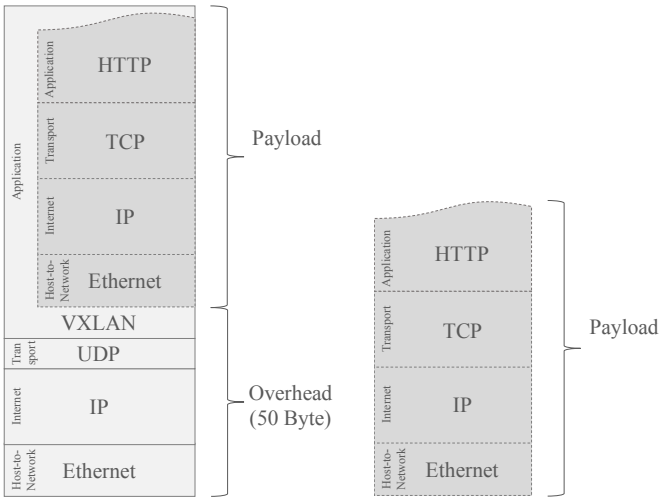
Overview

One of the basic differentiations between the wide range of SDN implementations is the one of an *underlay* versus an *overlay SDN*.

Overlay SDN. Based on existing network connectivity, *tunnels* are formed between end-hosts. Often, these tunnels are created using software switch instances running on compute hypervisors on both sides. Connected to these software switches are the network interfaces of virtual machines or containers. Through the tunnel, a virtual layer 2 network is provided over a layer 2 (switched) or layer 3 (routed) network spanning multiple physical hosts. The tunneling protocols used include *Generic Routing Encapsulation* (GRE) or *Virtual Extensible LAN* (VXLAN), which build directly upon IP respectively UDP. One such tunnel between two hypervisor instances is established per virtual network.

An exemplary HTTP packet transferred by a VM that is later sent over the network is depicted in Figure 3.1a. The inner packet is encapsulated in a VXLAN packet.

The benefit of such overlay SDN techniques is that they can run on top of existing Ethernet/IP-based infrastructures. As such traditional infrastructure is (usually) unaware of the tunneling protocol, decentralized non-SDN-based forwarding mechanisms are applied, i.e., shortest-path routing or *Equal-Cost Multi-Path* (ECMP). Using the latter, redundant paths between the two end hosts, e.g., inside a data center network, can be used to increase throughput, if multiple tunnels are established between the two hosts. The shortcomings of overlay SDNs include an additional overhead, i.e., 3% reduced packet size when using standard 1500 Byte *Maximum*



(a) Packet sent via VXLAN-based overlay SDN. *(b) Packet sent via underlay SDN.*

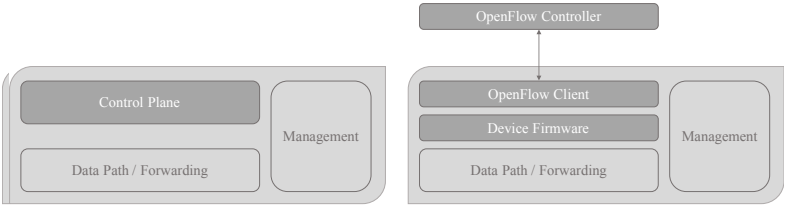
Figure 3.1: Protocol stack in VXLAN-based overlay SDN compared to OpenFlow-based underlay SDN.

Transmission Units (MTUs), as well as the loss of hardware offloading features, such as TCP checksum offloading, which leads to higher processing delays and CPU load.

Unterlay SDN. Through SDN-enabled networking devices, i.e., routers or switches, an underlay SDN allows to programmatically describe and modify the forwarding behavior of these devices through an external, logically centralized software. This entity, the *controller*, centralizes the previously distributed control plane. Figure 3.2 describes such separation and compares a traditional network element, e.g., a switch or router, with an OpenFlow-enabled [29] SDN switch. OpenFlow is today's most frequently used *Control Data Plane Interface* (CDPI), often referred to as *southbound* protocol.

Layer	Functionality	Realization
Data Plane	Packet forwarding	ASIC, FIB, flow table
Control Plane	Decides how to forward	MAC learning, STP, OSPF, BGP
Management Plane	Device configuration	SNMP, Netconf, SSH

Table 3.1: Overview over layers/planes of network devices.



(b) OpenFlow-based switch architecture.

Figure 3.2: Traditional network device architecture in comparison with OpenFlow-based device architecture. [58]

Defining the contents of the *Forwarding Information Base* (FIB) of network elements remotely increases flexibility for traffic engineering. Every flow, i.e., packets sharing common header data, precise decisions about the forwarding behavior can be made. However, underlay SDNs explicitly require support from the used hardware.

The remainder of this work focuses on OpenFlow-based underlay SDNs, as long as a concrete implementation is applied. Many of the introduced studies, however, are independent of a particular realization of the SDN.

3.1.2 Software Engineering Methods

Agile development has changed the way how software is developed. Instead of extensive manual tests prior the – very rarely happening – release, this style of software development applies automated testing, frequent releases, and continu-

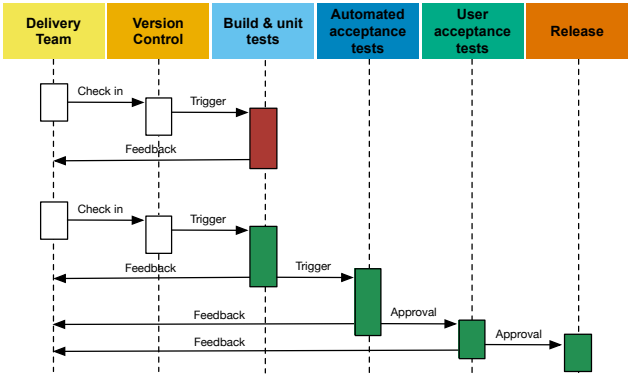


Figure 3.3: The deployment pipeline for software projects as suggested by [59]. Red: Test execution resulted in failure; green: tests succeeded.

ous feedback. Large parts of the software engineering world thus improved with regard to project success, development speed, and ability to incorporate changed requirements.

Given the new characteristics and potential of software-based networks, this work aims at exploiting these novel possibilities to improve performance and agility. An essential part is seen in the chance to learn from the software engineering world and at the core in the concept of *Continuous Delivery*, which will be introduced in the following.

Continuous Delivery

In order to mitigate the risk of broken software deployments while keeping a high pace of software releases, *Continuous Delivery* (CD) [59] introduces the concept of the *Deployment Pipeline*. The stages of such a deployment pipeline that are executed on a centralized server are described as follows:

Version control: Every change to the software is checked into a *Version Control System* (VCS), like *Git* or *Subversion*. The use of a VCS allows the team to

prevent overwriting changes of source code files, when different developers modify the same file in a short time. The traceability of changes by storing all historic versions of a software's source code gives the possibility to revert back to a state of the software that is known to work in case of regressions. After checking a change of the software into version control, the deployment pipeline is instantiated and the state of the software code passed through the following stages.

Build & unit tests: The centralized build server picks up the source code and creates an executable build artifact by compiling the source code. By creating the build on a centralized server, it can be assured that not only a single developer can compile and release software, but the whole team. Building only on PCs of developers introduces the risk of errors on other developer's machines caused by different configuration or compiler and library versions. After successfully creating the build and following the technique of *Test-Driven Development* (TDD), unit tests are executed against the compiled software. In case of any error during the build process or while running the unit tests, the pipeline is stopped and the developer informed about the problem. A fast feedback regarding any failure is important for the success of software development. If the stage is successfully passed, the next stage is triggered.

Automated acceptance tests: The particular steps in this stage are dependent on the actual implementation of the deployment pipeline. The result of this stage, however, is the knowledge that the created build artifact meets the specified acceptance criteria – or that they don't. These acceptance criteria range from functional and integration tests over to capacity tests, and longer-lasting source code analysis. Functional tests assure that the functionality of a software actually meets its specification. Integration tests assure that a part of the software works expected after integrating with other components. In order to execute all of these tests, a running version of the software is required, which is therefore deployed into a production-like infrastructure. This means that the environment where the tests are executed

matches the production environment in terms of version and configuration of operating system, libraries and installed software. If all tests can be executed successfully, the pipeline execution is continued and otherwise the delivery team is notified.

User acceptance tests: In order to verify the implementation of a new feature, the software is deployed into an environment accessible by all team members. The *Quality Assurance* (QA) sign-off when testers manually verify the functionality of the software, is the first human intervention after making the commit to the VCS. An extensive automated test suite that is executed in the previous steps and verifies the basic feature set of the software gives now the QA team time to focus on new features and exploratory testing. After this manual verification, the build artifact is ready to be released.

Release: The release of the software means that it is installed on the production servers or, in case of on-premise software, it is made available for customers to download. This stage can be either triggered manually, or after a button click in the software supporting the CD process.

The result of this is the knowledge about the state of the software, which could be either that it is verified to function or not. Through automated tests, the manual QA efforts are reduced and the duration that a change takes to pass through the deployment pipeline in order to be known as releasable is reduced to minutes or hours.

Besides to software development, the continuous delivery paradigm has been successfully applied to other areas. Modern server configuration management software follow the *infrastructure as code* paradigm, which allows to define the setup of a particular entity, mostly a cloud server including its application stack, as source code. Any change to the configuration inside the source code repository is only brought into production when the new configuration successfully passes the continuous delivery process.

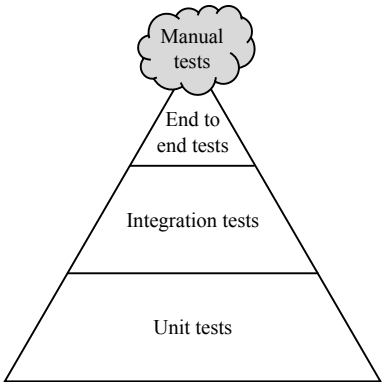


Figure 3.4: Sofware testing pyramid [60] in its variation from [61].

Software Testing

As discussed already for the CD pipeline, quicker software tests should run earlier than slower ones to allow fast feedback. A similar message is transported by the concept of the *Testing Pyramid* [60], as it is shown in Figure 3.4. It is made up of the following three types of tests.

Unit tests. As shown at the bottom of the pyramid, these tests should make up the majority of tests, as they are fast, focussed, and reliable. By testing the return values of a method for given input parameters, it can be checked very quickly with thousands of such tests per second. All calls to external systems, e.g., data bases or REST APIs, should be *mocked* so that none of these calls is ever made, but instead (defined) fake values are returned. Besides short execution time, unit tests also allow to narrow down the source of a test failure to a very small piece of the system.

Integration tests. Originally referred to as *Service tests*, unit tests combine multiple components together and evaluate the correctness of the business logic, e.g., a calculation of a complete tax declaration for supplied test in-

puts. This type of tests takes into account multiple subcomponents, making it hard to spot, which component is actually responsible for a wrong result. As external systems like data bases are often required to run these tests, they consume much more time and thus should make up only a smaller portion of all tests compared to the unit tests. However, such tests verifying the correct interoperability of multiple components, often created by different teams, are essential for verifying the overall correctness of the built system. A prominent example of such failure is the *Mars Climate Orbiter* satellite [62], which was lost in 1999 due to a unit mismatch. As the NASA established the metric system for all data, teams expected input to be in the newton-seconds unit. However, as one US team returned their value in pound-seconds, the overall system made wrong calculation leading to the satellite being lost in space. This mistake costed about USD 125 million.

End-to-end tests. Originally referred to as *UI tests*, end-to-end tests should make up the smallest part of all automated tests. While these tests are easy to comprehend, as they directly reflect the software's interaction with the user, they are hard to maintain, often unreliable and slow. In desktop or web applications, these tests mimic the user's mouse and keyboard inputs, e.g., through the use of testing tools like *Selenium* [63] or headless browsers like *PhantomJS* [64]. Such tests are hard to maintain, as smaller changes to the *User Interface* (UI) or differences in the rendering of different screen resolutions require updates to potentially many tests. Further, rendering differences or timing issues result in a high false-positive failure rate, which stresses developers and testers. Finally, such tests are slow, as they require everything *end to end*, from authentication, business logic, and finally rendering of the user interface. Organizations still practicing extensive UI tests end up with immense testing infrastructures, i.e., *salesforce.com*, which runs 50,000 VMs for testing 100,000 end to end tests multiple times per day. While not to this extend, end to end tests should be employed for testing the critical and user-interactive paths of the ap-

plication, e.g., user sign-up, but not the core functionality.

Manual tests finally should make up the smallest portion of tests, as they are *manually* executed by humans, compared to the other *automated* tests ran by machines. Traditionally and with grown software, manual tests often make up the largest part and sometimes take weeks prior a software release. Instead, manual testing should focus an *explorative* testing instead of repeatedly testing core logic.

Embedded in the continuous delivery pipeline, tests of all these fields help to ensure more stable software being released within shorter time. Fast unit tests ensure all building blocks work isolated by themself. More extensive tests that build upon the acceptance tests should then be made up of integration as well as end to end tests.

The authors of [65] argue for applying the software engineering concept TDD to the management of SDNs. In order to prevent a faulty network configuration to be deployed, a formal language called *Data Path Requirement Language* (DPRL) is introduced. Using the specifications made in DPRL, the compliance of an SDN controller against the specified rules can be verified. The authors provide a prototype implementation that builds upon *Mininet* and *Open vSwitch*.

While this work is certainly of significant importance for low-level network engineering and allows to verify the correctness of a *VNF Forwarding Graph* (VNF-FG), albeit its high complexity, it covers only one aspect. The goal of the work presented later in Section 3.2.2 is a more administrator-friendly, yet powerful, approach.

3.1.3 Network Infrastructure

While the separation of control and data plane is often seen as one of the main turning points in SDN, its openness regarding *Application Programming Interfaces* (APIs) also offers new possibilities. Network management applications now have one central endpoint to contact regarding all network matters, the northbound API of the controller.

This allows new approaches for network management. In the following, emphasis will be put on two aspects, network monitoring (cf. Section 3.3) as well as isolation of devices following the *Bring Your Own Device* (BYOD) concept (cf. Section 3.4).

Network Monitoring

This section covers relevant research work on the topic of elephant detection and SDN-based monitoring.

The relation between large and small (in terms of bytes), as well as short- and long-lived flows is discussed in [66]. The *elephant* flows, small in number, carry the biggest part of the entire traffic. In contrast, the *mice*, large in number, only contribute a small part to the total traffic volume. This has been observed in different data center measurement studies [67, 68]. Different approaches for elephant detection are summarized in the following.

Application-side labeling. One approach enabling application-specific flow monitoring is to label network flows within the application [69, 70]. Thus, every flow present in the network is labeled, e.g., with the type of application or the total amount of data to transfer. This approach, however, requires the modification of all involved applications as well as a dedicated trust relationship between the hosts and the network.

End-Host-Based detection. Instead of in the application, monitoring can also be performed in the operating system on the end-hosts. *Mahout* [71] detects elephant flows by monitoring socket buffers of end-hosts and is realized via a shim layer that marks packets which belong to an elephant flow before emitting them into the network. However, the network has to trust information received from the end-hosts.

Network-side flow statistics. Another approach, and probably the most common one, is to monitor within the network to keep statistics for every flow present at a given time. Statistics are exported from the network entities to the monitoring stations. Systems using this approach beside *sFlow* are [72]

and [73]. Providing such per-flow statistics for every flow in the network requires a severe amount of resources on the network elements, which might not be possible for all environments, especially data center or WAN networks. While packet sampling allows to apply this mechanism to even larger network environments, elephant detection becomes yet harder with higher sampling factors [74].

Programmability and packet matching in hardware switches offers multiple ways of using OpenFlow for monitoring. The most important approaches are described in the following.

NEC FlowSense [75] uses OpenFlow messages (*flow-removed* and *packet-in*) to measure the duration of flows. Furthermore, the amount of traffic measured via flow counters and inbound ports is logged in order to provide detailed information on link utilization.

OpenTM [76] leverages OpenFlow features for passive monitoring and is based on periodically querying flow statistics from selected switches. Furthermore, the authors investigate the trade-off between the load of the switches and the accuracy in terms of flow rate estimation. In contrast to the algorithm proposed in Section 3.3, FlowSense and OpenTM do not actively define flow rules besides the ones set up by the controller for forwarding traffic.

OpenSketch [77] offers a large feature set for flexible flow monitoring in SDN hardware. However, it relies on specialized, programmable hardware. In contrast, the approach described in Section 3.3 exploits existing features of OpenFlow hardware.

Currently available commercial SDN-based monitoring systems often do not build upon existing features. An out-of-band monitoring infrastructure comprising additional SDN switches is used by *Big Tap Monitoring Fabric* [78]. All production traffic is mirrored into the switching fabric, which then splits up the flows to

different monitoring servers. *Network Packet Brokers* (NPB) follow a similar approach. A central controller manages a certain number of NPBs, which analyze and forward the monitoring traffic through a dedicated network, the *vMesh* [79].

Isolation and BYOD

Programmable BYOD Security (PBS, [80]) applies SDN/OpenFlow to mobile devices using the *PBS-DROID* application. This lets the Android device run an OpenFlow-controlled switch with the smartphone apps connected to it. By running on the device itself, it allows per-app policies. Compared to the approach described in Section 3.4, PBS is able to work on a more fine-grained level, while the network infrastructure itself does not have to be changed. In contrast, the motivation behind the approach that we propose is to not require changes to the devices, but instead providing a high level of security through the support of the network infrastructure.

3.2 Management of Softwarized Networks

SDN lead to a shift away from hardware-centric networking towards open interfaces and software-driven network control. The controller running as software application on a standard server allows short innovation cycles for the network control plane. In contrast, traditional integrated devices require firmware updates to introduce new mechanisms or protocols. The pace of innovation can also be seen by the number of available OpenFlow controller implementations – and the number of projects that are stopped being developed any further.

Despite the availability of open source SDN controllers, it is unlikely that many network administrators outside change its core functionality. However, the plugin architectures of modern SDN controllers allows extensions of this centralized instance. This includes monitoring, routing, security, application awareness and quality of experience optimizations. Commercial offerings for such plugins are the business case for companies developing an open source controller base platform.

Network Functions Virtualisation (NFV) in contrast aims at replacing *Network Functions* (NFs) provided by monolithic hardware middleboxes with software implementations running virtualized on standard servers, the *Virtual Applications* (VAs). An overdimensioning of resources as it is usually done with hardware can be avoided by applying elasticity mechanisms of nowadays cloud application stacks, including cluster and replication functionality.

Therefore, NFV promises slim software instances that provide a particular functionality, like applying *Deep Packet Inspection* (DPI), firewalling, or functionality of LTE mobile networks [7]. Again, shorter innovation cycles and an increased flexibility are a driver for research, vendors and operators to investigate the use of virtualised network functions. SDN is the preferred mechanism to pipe all or only specific flows, like all traffic on TCP port 80, through a specified set of network functions. The VNF Forwarding Graph [81] specifies, which network functions should be passed, e.g. mirrored traffic to a monitoring function, or to let it traverse an intrusion detection function.

Frequently modifying the network software and configuration of SDN controllers and the NFV infrastructure comes with the big risk of disrupting network connectivity. Besides bugs and breaking changes in software implementations, human-made configuration errors are reasons why traditional networks are progressing very slowly. The open issue is now, how virtualization of network elements and functions helps to allow frequent changes without unexpected outages.

An application of CD to softwarized networks will be discussed in the following, while special emphasis will be put on the testing part. The content of this section is taken from [5] and [22].

3.2.1 Continuous Delivery of Network Functions

Regardless of these new accomplishments, even after decades, the *Command Line Interface* (CLI) is still the best friend of network engineers when configuring switches, routers or other network devices. Furthermore, the network is configured decentralized directly on the devices. Thus, there is no simple way to test the complete network configuration before applying it device by device. The network

engineer has to make sure to always enter the correct commands, make no typing errors and hope for no unforeseen side-effects while configuring the network. This cumbersome process causes severe problems for the management of today's IT infrastructure, as the risk of breaking the production network results in security updates, e.g., for the infamous Heartbleed bug, being applied late or never to network elements [82, 83]. Automatizing particular tasks often requires the use vendor-specific interfaces, which does not allow to apply the same methods to devices of another vendor.

Installation of updates or configuration changes are nowadays often scheduled in maintenance windows, when users are informed beforehand that failures may occur. This approach tries to increase the *Mean Time Between Failure* (MTBF), as downtimes are avoided by deferring changes to be applied in batches – or until they become urgent. At the same time, the risk of failure when changes are applied is increased, as multiple changes are applied at the same time. Furthermore, also identifying the root causes of potential problems becomes harder, when more than one change is introduced to the network at the same time.

Software development projects faced similar problems before agile project management methods like *Scrum* were introduced to reduce the risks of late integration. Here, methods from *DevOps* [84], allow for fast feedback cycles and frequent releases in order to avoid misconception, extensive manual testing, as well as failures that are hard to identify due to the fact that many changes are applied to the system simultaneously. Instead, modern web companies release changes into production more often – in number of hundreds or thousands per day [85, 86]. These practices aim to increase availability by reducing the *Mean Time To Repair* (MTTR) instead of increasing the MTBF. Through automated test execution and deployments, the quality assurance efforts per change can not only be reduced, but also the time to release a fix into production, which might also be to revert a change, is effortless.

Some current network configuration management tools, allow for the automatized deployment of changes to network devices. However, the proprietary nature of the configuration interfaces of traditional network interfaces results in a high complexity and price for *Network Configuration Management* (NCM) tools. Thus,

many current networks are not using such tools and still maintained manually by the network engineers.

Recent developments in the area of SDN offer new opportunities for change. Besides the benefits of a better network performance, a simplified management and configuration of the network infrastructure is promised. Most research work on software defined networks, however, focuses on improving network performance or reliability by defining controllers architectures and roles of entities [87] or generally the more sophisticated management of network flows [88]. The life cycle of the introduced SDN entities, including provisioning and maintenance, could benefit from applying CD to SDN in order to not only benefit from the provided agility, but also to support the effortless and risk-free deployment of new networking software.

In the following, it will be described, how the concept of *continuous delivery* (cf. Section 3.1.2, [59]) can be applied to the area of softwarized networks. The main building blocks are (a) process automation, (b) automated tests, (c) availability of realistic test environments and (d) infrastructure as code.

Through *process automation*, network engineers are supported in their daily work – instead of being jammed by yet another tool to use and process to follow. The goal is to reduce the overhead of work that is required to make a configuration change. Instead of manually logging into all networking devices, the deployment will happen automatically on all affected entities. An *automated test suite* further supports the goal of reducing the overhead, in this case the additional work of manual testing. Furthermore, automated tests bring (high) confidence that any change that passed the automated tests will not be disruptive when deployed into production. The availability of *realistic test environments* that can be set up automatically is also a prerequisite of executing automated tests. Compared to traditional, hardware-centric networks, where devices have to be connected manually, the softwarization of networks now allows to automatically set up an environment matching the production environment through instantiation of virtual machines running the same software in the same versions. This allows the engineer or quality assurance teams also to manually test changes prior to rollout, without complicated setup of infrastructure. Finally, *infrastructure as code*

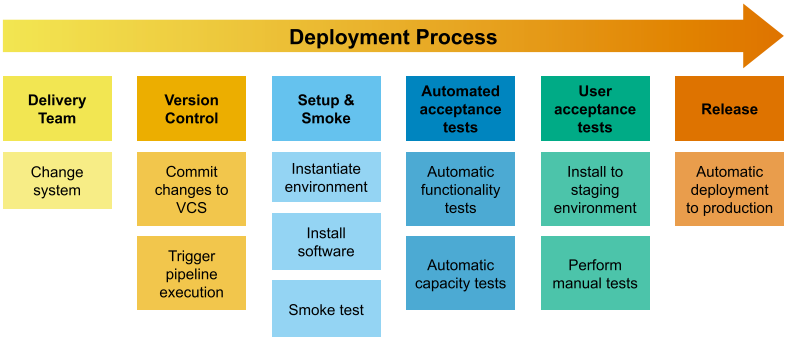


Figure 3.5: Suggested deployment pipeline for network functions.

ensures that the configuration of servers running a network function is not only documented in code, but also that this code can be applied to another system in order to reproduce the configuration of the production environment. This technique allows to instantiate new development and testing environments without manual intervention.

The stages of the resulting deployment pipeline are shown in Figure 3.5 and in the following illustrated using two typical functions for the data and control planes of an SDN/NFV-based network, (a) an SDN controller and (b) a *Virtualised Network Function* (VNF) applying traffic shaping.

SDN Controller Pipeline

The controller plays a critical role in an SDN network. Any fault in its execution would make the network "headless" and have a large impact on its operation. Outages resulting by an administrator supplying an invalid configuration, which brings the controller down for several seconds or minutes have to be urgently avoided. Therefore, the error-prone manual configuration on production devices has to be succeeded by an automatic process that is backed by automatic tests.

While the CD concept can be also applied for the software engineering task of developing the SDN controller software, this work focuses on the more frequent

case when an administrator uses an existing controller software and configures it according to own needs. If the controller software itself has to be compiled, a dedicated deployment pipeline following the original version of CD in terms of software engineering projects should be established. The case that is presented in the following uses the compiled artifact of an SDN controller software that could be also a purchased software, which is not available as source code, but only as binary artifact.

The following stages are suggested for the deployment pipeline applied to an SDN controller:

Version control: The version control repository for the SDN controller pipeline contains the configuration of the servers running the controller and how the controller has to be installed. This includes an exact specification of the controller version that has to be installed, as well as the configuration files of the controller software.

Setup & Smoke: As this pipeline does not involve any compilation of software, the focus is more on putting the infrastructure components together. Therefore, a virtual machine that matches the configuration of the production servers is provisioned. The controller software is downloaded from the defined source in the specified version and installed into the virtual machine. This ensures that the automatic deployment process works and the controller software and all dependencies can be downloaded and installed. Finally, the setup is completed by supplying the configuration files as specified in the version control repository.

Following the principle of fast feedback, this stage only includes tests that can be executed in a very short time frame, while catching many of the errors that are likely to happen. In [59] it is recommended to not exceed the 10 min mark. What should fit in this time frame are *smoke tests*, which only consist of testing, whether the controller software using the supplied configuration is able to start up or not.

Automated acceptance tests: The tests running in this phase ensure that the specified acceptance criteria for the software being deployed are met. A failure in any of the tests means that at least one criterion that is essential for the software is not fulfilled and thus the pipeline has to be stopped. Besides functional tests, which means that the software works as expected, the acceptance tests also include non-functional tests, verifying that certain performance or capacity requirements are met.

The functional test suite of an SDN controller should include at least the following checks:

a) *Accepts incoming southbound connections*: This check ensures basic functionality, namely that the controller is listening to the correct interface and port for incoming connections. This allows to quickly identify basic errors and helps to prevent them to fail more complex tests.

b) *Allows a switch to forward traffic*: Independent of an active or reactive setup of rules inside the switches' forwarding tables, the result should be that two stations connected to a switch can exchange network traffic.

c) *Accepts incoming northbound connections*: The whole benefit of SDN can only be exploited, if an integration into and coordination with the remaining IT infrastructure is happening [27]. Therefore, the functionality of the northbound API, nowadays usually implemented as a RESTful API, is as essential as the functionality of the southbound API is. This first northbound test ensures that a client is able to connect. Besides the correct configuration of the listening interface and port, this also verifies that the authentication is working correctly.

d) *Required feature set of northbound API is working*: Certainly, not all features of an SDN controller's northbound API will be used in a particular setup. However, the subset of functionality that is used should be verified to work correctly. If the production setup consists of a

cloud controller that interacts with the SDN controller, the correct functionality of that part of the REST API should be tested automatically. This ensures that especially updates of the controller software that might introduce changes or bugs are not deployed without the confidence that the used feature set is still working.

Other than functional tests, non-functional tests evaluate characteristics of a setup that cannot be directly described in a fashion that a certain response is expected for a certain request. One of these non-functional requirements is the ability to handle a certain number of connected switches as well as a specific rate of requests per second.

User acceptance tests: If desired, the administrator or a QA engineer can conduct further manual tests in the staging environment. Therefore, the tester can connect an own virtual switch, or even a hardware switch, to the controller software running in the staging environment. For convenience, additional virtual machines can be deployed and preconfigured so that the tester can focus on the actual testing, instead of struggling with setting up the test environment. The manual tests done here should not test absolutely critical functionality so that they have to be repeated prior to every deployment, thus during every run of the pipeline. Instead, such tests should then be automatized and executed in the previous stage. This allows the tester to focus on new functionality in detail.

After manual testing, the tester lets the pipeline either proceed to the next stage, or fails the pipeline and returns back to the delivery team to adjust.

Production Deployment: The final stage of the deployment pipeline triggers the production deployment. This step can either be done automatically after every successful execution of the pipeline, bundled into e.g. a single deployment per day, or manually triggered.

Traffic Shaping Network Function

NFV aims at replacing hardware middleboxes through VNFs. The deployment pipeline for a VNF that is inserted into the flow of traffic will be described in the following using a traffic shaping function. Besides giving more capacity to other network flows, traffic shaping or rate limiting is a common use case for mobile networks. After reaching the "flat rate" limit of e.g. 1 GB per month, the mobile operator is shaping the subscriber's traffic to a lower rate. Therefore, received traffic is queued inside the entity running the function and then forwarded with a reduced rate.

The suggested deployment pipeline for a TS-VNF and similar functions is as follows:

Version control: Similar to the pipeline of the controller, the version control repository for the traffic shaping VNF pipeline contains the configuration of the servers running the function. If the function is developed as source code, another deployment pipeline following the original CD for software projects [59] has to7 be set up.

Setup & Smoke: Again, the main focus of this stage is to ensure the successful deployment by installing the specified software into a production-like environment. A smoke test that verifies that the application implementing the VNF starts without errors.

Automated acceptance tests: The acceptance criteria for a VNF include criteria that are common for a certain type of network function and some that are specific for a particular function. Tests that are common e.g. for all middlebox VNFs ensure that incoming traffic is also sent back as specified, in order to reach the final destination or the next network function. Automated acceptance criteria that are specific to a traffic shaping function should include:

a) *Sent data matches received data:* Using a TCP data transfer it is checked that the Layer 7 payload sent out equals (bitwise) the pay-

load that is sent into the function under test. The aim of traffic shaping is not to falsify the data in any way. Implementation errors which result in data corruption can be detected this way.

b) *Traffic shaping is successfully applied:* In order to verify the core functionality, the throughput or duration of a data transfer is measured. By defining a certain threshold, deviations of some percentage from the defined bandwidth limit are not causing the test to fail, but ensure that shaping is applied.

c) *Only traffic that should be shaped is shaped:* Assuming that the shaping functionality allows to handle flows differently, this test ensures that bandwidth limits are only applied to the specified flows and not to others that should not be shaped. Therefore, a bandwidth limit for a certain flow criteria is defined. The duration of a data transfer through the network function that *not* matches the shaping rule is measured and the average throughput computed. If the measured throughput exceeds the shaping limit by a factor specified in the test, e.g. twice, the test succeeds and it can be assumed that the shaping is only applied to specified flows.

While the elasticity of network functions and the automatic up and down scaling should ensure that additional capacity is provided in case of increased resource requirements, automated capacity and performance tests are still of big importance. Heavily increased requirements of a certain functionality can cause various kind of trouble, including additional costs. Therefore, either static triggers with fixed limits, or a limit of divergence from previous runs ensures that a by far more resource-intense new implementation or configuration is not released into production.

User acceptance tests: Again, the network engineer or QA staff has the chance to manually verify functionality like already described in the pipeline described for the SDN controller pipeline in Section 3.2.1.

Production Deployment: As already applied in previous testing environments, the automated deployment is now happening into the production environment. Depending on the configuration of the pipeline, this stage is executed automatically after the QA sign-off in the previous stage or on the push of a button in the software supporting the continuous delivery process.

The deployment pipeline of an SDN controller and of a traffic shaping VNF as typical functions of SDN/NFV-enabled networks were used to illustrate how the agile deployment of such functionality into the production network should happen.

Discussion

In this section, the aspects that are worth noting to understand the reasons for introducing a continuous delivery process are discussed. Furthermore, open issues that should be tackled by the software development or network engineering community in order to further support the adoption of this concept are described.

Releasing more frequently. For agile software development, the "highest priority is to satisfy the customer through early and continuous delivery of valuable software" [89]. That quality and productivity is increased through agile methods in software engineering is shown by a survey in [90]. Also that more releases do not mean more bugs, instead that fixes are released faster, is shown in a study of *Mozilla Firefox'* long and short release cycles [91]. In the case of *Amazon.com*, it is reported that the number of outages triggered by software deployments was reduced by 75% within 5 years. The "safety net" provided by a deployment process backed by automated tests reduces stress for humans. Through the smaller amount of changes deployed simultaneously, the identification of root causes easier becomes easier in the case of negative side-effects.

Overhead of testing every change. The execution of numerous automatic tests often results in the feeling that CD would introduce a large over-

head into software development processes. In order to be able to identify the particular commit in the source code repository that introduces a regression, every single change executes numerous automated tests. The suggested pipeline execution includes the instantiation of multiple virtual machines and running functional tests with potentially a duration of several minutes.

Computing power is cheap. The compute power spent for automated tests is the replacement for manual tests repeatedly executed by humans. These manual tests, in the field of network software certainly similar to software development, otherwise bind a large capacity of human resources resulting in high operational costs. Compared to manually applying changes, the delay until a change passed through the deployment pipeline, is notably larger. However, manual configuration changes are considered bad practice, as they cannot be tracked (who changed what and when) or easily reverted and do not scale to a large number of devices. Therefore, the time for an automated deployment process has to be preferred over manual fire-and-forget changes in production.

Automated testing of networks. The work in [65] (cf. Section 3.1.2) can be seen as an important step into the right direction. TDD is an important building block for continuous delivery of networks. However, the suggested prototype implementation would still require the network engineer to write Python code. The next section will therefore focus on an administrator-friendly approach based on *Behavior-Driven Development* (BDD).

Metrics. An essential part of CD and the related DevOps practices is to monitor how the production infrastructure behaves. The collection of metrics in a DevOps context means more than just monitoring of bandwidth usage and QoS [92]. Instead, a data-driven culture relies on aggregating data from numerous sources together, in order to allow engineers, as well as business units to take decisions based on measured truth and not on assumptions

or feelings. While the change of network parameters does not necessarily result in a change of measured QoS metrics, it can affect performance of applications running in the network. The collection and aggregation of metrics can go so far that also the number of transactions in an online shop is monitored. In case of a large decrease of this metric, the metrics collected from the network side can be correlated as well as matched with times of deployments. If a certain behavior is seen after a point in time at which a change to the network or server infrastructure happened, this change is likely to be the cause for this changed behavior. Again, it is important that all changes are tracked in a version control repository to revert to previous states, as well as that data about the exact time when deployments happen are stored.

Feature flags [93] allow to change a particular behavior, like the availability of a feature, for a group of users, requests, or other types of aggregation. Transferred to networks, this could mean that a mechanism that should be evaluated can be tested under production conditions for only a number of users, devices, or flows. Metrics then allow to compare one implementation against the other. The flexibility of steering network traffic provided by SDN can be seen as an enabler for techniques such as A/B-testing, where two different implementations are compared regarding specified metrics.

3.2.2 Testing Methodology for Software-based Networks

In order to give network operators more confidence prior to rolling out changes into the production network, we suggest to provide an operator-friendly way of specifying test scenarios and acceptance criteria. Based on a human-friendly way to define these tests, network operations should be able to move forward quickly. All components of a software-based network are or, in case of switches, can be substituted by software. This offers the chance to create arbitrary network setups without dealing with physical equipment.

It is, however, the job of developers to provide the foundation on which operators can build upon. The benefits, if infrastructure projects would offer such

means, will be demonstrated in the following.

Test Specification

An example for a test specification in the domain-specific language *Gherkin* [94] could look as given in Listing 3.1. Gerkin allows to express the expected behavior of a system under certain preconditions. This simplified scenario is formulated in the domain language of an operator and allows to verify basic network connectivity, which is that two hosts can communicate with each other.

A slightly extended example to verify the network's behavior during a controller restart or outage is given in Listing 3.2. The aim is to ensure that already established network connectivity is further maintained, even if the controller becomes unavailable.

Another similar test could check wether two hosts which did not yet exchange data can connect with each other after the controller became unavailable. This allows to verify the correct implementation and configuration of the fallback to the *fail-save mode*.

Features that a testing framework could provide are tokens like `a large topology with loops` or `the flows take distinct paths`. By having this layer of natural-language style specification on top of the actual step definition, cf. Figure 3.6, the specifications are easy to formulate and – even more important – easy to read in case of a failure.

Listing 3.1: BDD specification of a reactive controller behavior

```
Given a network topology with two hosts
When host 1 pings host 2
Then the ping is successful.
```

Listing 3.2: BDD specification of a reactive controller behavior

```
Given a network topology with two hosts
And host 1 pings host 2
And the ping is successful
When the controller shuts down
```

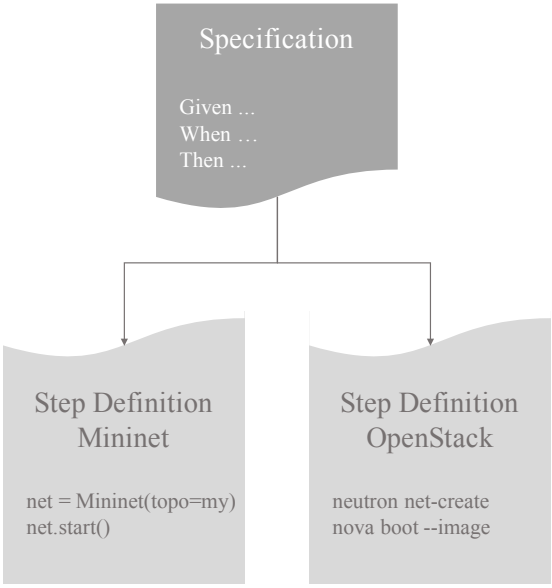


Figure 3.6: Different drivers can be used for implementing step definitions, i.e., Mininet or OpenStack. Both run the same specification.

```
And host 1 pings host 2
Then the ping is successful
```

Test Implementation

The previously described test specification formulated in Gherkin needs be translated into actual code. As motivated, these *step definitions* should not necessarily be created by the network operator, but can be offered by vendors, e.g., of the SDN controllers or third parties like the open source community.

The benefit of having this layer of abstraction between test specification and step definition can be seen in the possible implementations as illustrated in Figure 3.6. A BDD testing framework could offer different drivers that use different

types of testing infrastructure to verify the test specification.

We illustrate this idea based on the following two drivers, one using Mininet, the other OpenStack.

Mininet driver. Mininet is a lightweight network emulation tool running locally in a single Linux host. It uses Linux' network namespaces to emulate multiple hosts connected to one or more Open vSwitches connecting to the SDN controller. Through its Python API, Mininet allows to programmatically create topologies and execute commands on the emulated hosts.

Listing 3.3: Step Implementation using Mininet.

```
@given('a network topology with {numHosts} hosts')
def step_impl(ctxt, numHosts):
  ctxt.mn.addSwitch("s1")
  for num in range(numHosts):
    # create host names h1, h2, ..
    host = 'h' + num
    ctxt.mn.addHost(host)
    ctxt.mn.addLink(host, 's1')

@when('host {hst1} pings host {hst2}')
def step_ping(ctxt, hst1, hst2):
  h1 = MnHelper.getNodeFromName(ctxt.mn, hst1)
  h2 = MnHelper.getNodeFromName(ctxt.mn, hst2)
  packetLoss = ctxt.mn.ping((h1,h2))
  context.pingResult = packetLoss
```

As Mininet is very lightweight, it offers short setup times even for relatively large topologies (128 hosts and 128 switches set up in less than 10 seconds). The downside of Mininet is its characteristic of an emulation tool, which limits the scope of tests to the possibilities offered by Mininet. As BDD tests primarily focus at functional testing, the inappropriateness of Mininet for performance testing should not be a big disadvantage.

OpenStack driver. OpenStack as a full-blown cloud implementation offers APIs to instantiate virtual machines and virtual networks, as well as for orchestration. Assuming that an organization's production infrastructure runs in a (public or private) OpenStack cloud, it is desirable to run also the testing infrastructure in the same. With orchestration tools like OpenStack Heat [95] or Terraform [96], it is possible to define complete cloud infrastructures and instantiate them multiple times.

The actual test implementation logs into the VMs via SSH and executes test commands. These can range from `ping` to accessing the web server via HTTPS or running a bandwidth-consuming download.

While this reflects a test environment very similar to the production environment, it comes at a high cost. Although parallelized, the instantiation of the infrastructure takes several minutes. For provisioning, e.g., by using Chef [97] to install the SDN controller, another several minutes are would be consumed, until test execution can finally start.

Aim of introducing the two diametrically opposed drivers for BDD tests was to illustrate the difference between different test implementations for the very same test specification. Given the distinct execution times, step definitions using Mininet can be used for quicker tests, i.e., smoke tests. In contrast, a verification of critical network functionality in the acceptance stage should rely on more realistic environments, preferably matching the production environment.

3.3 SDN-based Flow Monitoring

In the following, we propose a detection mechanism for elephant flows based on network-side monitoring. As it implemented within the SDN network, no support by the end hosts or involved applications is required. Instead, it leverages already existing counters of OpenFlow switches that are automatically updated whenever a packet matching an existing flow rule passes the switch. Through iterative refinement of flow rules, the proposed algorithm allows narrowing down the aggregation level from coarse grained rules to very fine grained rules and counters.

Hence, the proposed *ZOOM* algorithm offers a very lightweight monitoring solution focused on elephant detection that requires no additional hardware. The content of this section is taken from [10].

3.3.1 ZOOM: Network Monitoring using SDN

This section describes the ZOOM algorithm, a lightweight approach leveraging packet counters in OpenFlow switches for elephant detection. After the algorithm is introduced, its accuracy is evaluated using a prototypical implementation.

Elephant flows are often defined by a certain, fix data volume they transfer, e.g. 100 MB. This work follows a slightly different approach based on [98]. Lan et al. define elephants as flows that carry more data than the mean of all flows plus three times the standard deviation. We base our definition on this metric while introducing a new variable called s_{ele} which enables us to define different elephant thresholds by using different values for s_{ele} in the following equation.

$$S = mean(\text{flowsize}) + s_{ele} \times std(\text{flowsize}) \tag{3.1}$$

Thereby, $mean(\text{flowsize})$ refers to the average size of all flows present in the system and $std(\text{flowsize})$ describes the standard deviation of the encountered flow sizes. The corresponding threshold for elephant flows is thus dependent on the traffic in the observed network and can thus be used independently of the monitored network.

The proposed algorithm is based on polling of flow statistics from OpenFlow switches. Such statistics are automatically maintained by the switches for each currently installed flow rule. Therefore, the main idea is to define flow entries that do not modify the forwarding behavior of switches, but still enable flow monitoring via packet counters. An iterative refinement of the IP address ranges matched by the flow rules then allows to narrow down the elephant flows. The match fields of these flow entries are set so that – in the simplest case – a binary search over the whole IP space is performed for source and destination IP addresses. Thus, the IP range of possible source and destination addresses is divided into sections,

each covered by one of the created flow rules, which then trigger statistics collection within the switch. By splitting up IP ranges into more than two parts, the algorithm can proceed faster, however, at the cost of creating more flow entries.

In contrast to NetFlow/sFlow-style monitoring, counters are not created for every single flow. Instead, the amount of data that needs to be processed by the algorithm is independent of the number of flows present in the system. In addition, if prefixes of the IP source or destination addresses are known, the run time can be shortened even further. In particular, the runtime grows linearly with the number of wildcard bits in the IP range that is to be searched. The following equation describes the runtime of the ZOOM algorithm.

$$\mathrm{t}_{\mathrm{ZOOM}}(n_{bit}, n_{flows}, n_{top}, t_{wait}) = \frac{n_{bit}}{\log_2(\frac{n_{flows}}{n_{top}})} \times t_{wait} \tag{3.2}$$

Thereby, $n_{bit} \in [0, 32]$ is the number of remaining wildcard bits in the IP range.

The ZOOM algorithm is listed in Algorithm 3, visualized in Figure 3.7, and described in the following.

a) **Initialization.** The algorithm's behavior can be adjusted with three input parameters. The first parameter n_{flows} defines the number of flow entries that are created per cycle. This defines the number of sections into which the remaining IP range is divided. While $n_{flows} = 2$ corresponds to a binary search, higher values allow faster advancing at the cost of more flow rules. How many of the n_{flows} sections covering the most traffic are treated as candidates to contain elephants and are thus further analyzed is defined by the n_{top} parameter. This also determines the total number of elephant flows contained in the output produced by the algorithm. Finally, t_{wait} defines the waiting time in seconds between the creation of flow rules and polling of corresponding statistics. Hence, it represents the interval during which passing traffic is monitored.

b) **Initial Flow Generation** As the algorithm searches for source and destination addresses of end-to-end flows, it is required to search the whole

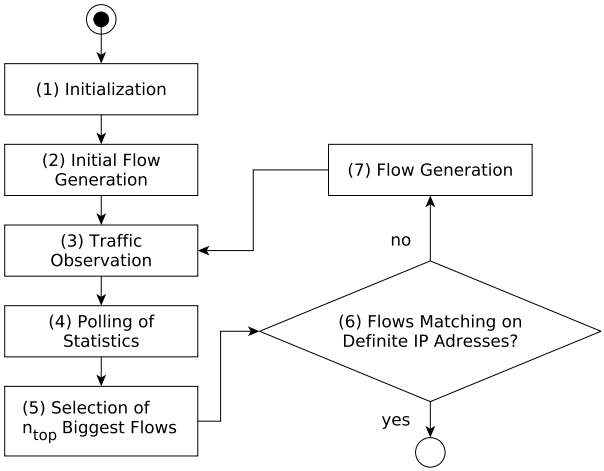


Figure 3.7: Flow chart of the ZOOM algorithm.

IP address range for both sources and destinations concurrently. Hence, the number of flow rules that are generated is n_{flows}^2. In this first step, n_{flows} flow entries covering all addresses from 0.0.0.0 to 255.255.255.255 are generated for source and destination respectively. If not the whole IP range needs to be covered, the initial flow rules change depending on the already known bits of the IP address range. For $n_{flows} = 2$, the resulting flow entries are given in Table 3.2. It can be seen that all fields except source and destination IP are wildcards. Instead of matching pairs of definite IP addresses, the set of possible pairs of source and destination addresses is partitioned into a number of n_{flows}^2 flow entries.

c) **Traffic observation.** During t_{wait}, while the algorithm pauses, the packet counters of the switch are automatically updated for packets matching one of the specified flow rules.

Input Port	Ethernet Type	Source MAC	Dest MAC	Source IP	Dest IP	ToS	Source Port	Dest Port	Protocol
*	0x800	*	*	0.0.0.0/1	0.0.0.0/1	*	*	*	*
*	0x800	*	*	0.0.0.0/1	128.0.0.0/1	*	*	*	*
*	0x800	*	*	128.0.0.0/1	0.0.0.0/1	*	*	*	*
*	0x800	*	*	128.0.0.0/1	128.0.0.0/1	*	*	*	*

Table 3.2: Exemplary flow entries matching the whole IP range (n_{flows} =2).

d) **Polling of statistics.** After t_{wait}, flow statistics are requested from the switch.

e) **Selection of n_{top} biggest flows.** Packet counters of previously retrieved flow statistics are evaluated and the n_{top} biggest flows regarding average bandwidth are selected for further processing.

f) **Termination condition.** The subsequent action is to check whether the termination condition is satisfied. This is the case if the section covered by each of the selected n_{top} biggest flows contains only connections between a definite source/destination IP address pair, i.e., if all 32 bits of the source and destination IP match fields are specified. If this termination condition is met, the identified n_{top} biggest flows are returned as result of the algorithm. As long as the IP address match still contains wildcard bits, execution continues.

g) **Flow generation ("Zoom In").** Again, a number of n_{flows} flow entries is defined. Using these, the source and destination IP ranges covered by the previously found n_{top} biggest flows are split up into $\frac{n_{flows}}{n_{top}}$ entries each. After removing all previously defined flow rules, the newly generated entries are pushed to the switch and the algorithm is repeated starting from Step 3.

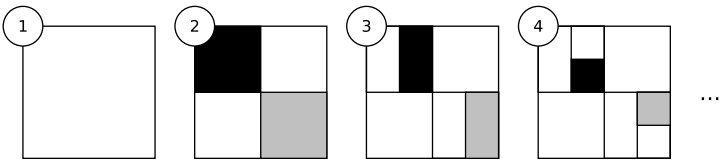


Figure 3.8: Example of refinement ("Zoom In" step) for $n_{flows} = 4, n_{top} = 2$.

Figure 3.8 illustrates the refinement process using $n_{flows} = 4$ and $n_{top} = 2$. Depicting the first 4 cycles of the algorithm, it can be seen that the $n_{top} = 2$ sections covering the biggest traffic (shown in black/gray) get iteratively refined by getting split into $\frac{n_{flows}}{n_{top}} = 2$ segments each.

Implementation

In order to evaluate the accuracy of the proposed algorithm, a proof-of-concept has been implemented as a module for the OpenDaylight controller (ODL).

The current implementation is focussed to statistics collection. In order to still allow correct forwarding of production traffic, the action of the flow entries defined by ZOOM should, e.g., set the *goto-table* action to a table containing the actual forwarding rules. This, however, is not relevant for evaluating the ZOOM algorithm's accuracy. Contrary to the proposed algorithm, our prototype implementation is limited with respect to legal values regarding the parameters introduced earlier. This leads to the following constraints:

$$n_{flows} \in \{1, 2, 4, 16\} \tag{3.3}$$

$$n_{top} \in \{1, 2, 4, 8\} \tag{3.4}$$

$$\frac{n_{flows}}{n_{top}} \in \{1, 2, 4\} \tag{3.5}$$

Furthermore, ODL does not allow creating flow entries that match the 0.0.0.0 IP address. Therefore, the proof-of-concept implementation of the ZOOM algorithm

has to create more than n_{flows} entries in the initial step (Step 1 in Figure 3.7). Instead of 0.0.0.0/1 and 128.0.0.0/1, a reasonably low number of matches[1] are defined. Afterwards, the algorithm continues as originally.

3.3.2 Evaluation

The results that are presented and discussed in the following are obtained by running the aforementioned OpenDaylight implementation. The accuracy is evaluated by replaying a publicly available traffic trace from the *Waikato Internet Traffic Storage* (WITS). Characteristics of the trace that is used to evaluate the ZOOM algorithm are listed in Table 3.3. In addition to the general trace information, the Table 3.4 shows statistics of elephant flows resulting from different elephant thresholds. The values of $s_{ele} \in \{1, 10, 30\}$ are selected as they show that the total number of elephants and s_{ele} behave roughly inversely proportional. The maximum value for $s_{ele} = 30$ is chosen since, due to the low elephant density, the accuracy decreases significantly for this threshold. Furthermore, the average number of active elephants at each point in time as well as the average duration of elephant flows is provided. Finally, the table shows the traffic contribution and the percentage of elephants compared to the total number of flows.

The evaluation is performed by using *tcpreplay* to replay the trace into an Open vSwitch connected to ODL running the ZOOM module. In order to calculate the accuracy of the algorithms, the experiment results are compared to the data available due to global knowledge about the traffic trace. This is obtained by processing the packet trace to obtain information on which flows (and elephants) are active at which time of the trace. These are then matched against the data obtained by the algorithm to calculate the accuracy of the results.

In order to conduct the measurements, the algorithm is started at different time offsets (5, 10, 20, 30, 40, 50 seconds) for every combination of parameters while replaying the traffic trace. In the following, the influence of different parameter settings on the accuracy of the ZOOM algorithm is evaluated.

The evaluation is performed by analyzing the accuracy of the results produced

[1] 1.0.0.0/8, 2.0.0.0/7, 4.0.0.0/6, 8.0.0.0/5, 16.0.0.0/4, .., 240.0.0.0/4

Algorithm 3: The ZOOM Algorithm.

Input: $n_{flows} \in \{2, 4, 16\}$, $n_{top} \in \{1, 2, 4, 8\}$
Assert $\frac{n_{flows}}{n_{top}} \in \{2, 4, 16\}$;
$S_{part} = \{n_{flows}$ partitions of source IP range$\}$;
$D_{part} = \{n_{flows}$ partitions of destination IP range$\}$;
Generate flow entries $E_{init} = S_{part} \times D_{part}$;
Push E_{init} into switches;
Sleep for t_{wait} seconds;
Set *foundFlowRuleToRefine = true*;
while *foundFlowRuleToRefine* **do**
    Collect flow statistics;
    Set *foundFlowRuleToRefine = false*;
    for $i = 1$ *to* n_{top} **do**
        Select i-th biggest flow;
        $S =$ Extract source IP range;
        $D =$ Extract destination IP range;
        if $|S| > 1$ *or* $|D| > 1$ **then**
            Set *foundFlowRuleToRefine = true*;
            $S_{part} = \{\frac{n_{flows}}{n_{top}}$ partitions of $S\}$;
            $D_{part} = \{\frac{n_{flows}}{n_{top}}$ partitions of $D\}$;
            Generate $E_i = S_{part} \times D_{part}$ flow entries;
        end
    end
    Remove previously installed flow rules;
    Push $E_1, \ldots, E_{n_{top}}$ to switches;
    Sleep for t_{wait} seconds;
end
Write results;
return;

Trace Metadata	
Trace URL	http://wand.net.nz/wits/ ... ispdsl/2/20100106-030946-0.dsl.php
Trace duration	1214 seconds
Traffic type	DSL Subscribers
Total number of flows	1,124,575
Concurrently active flows (min/avg/max)	3,641 / 18,916 / 20,919

Table 3.3: ISPDSL-II Traffic Trace Metadata.

Elephant Statistics	$s_{ele} = 1$	$s_{ele} = 10$	$s_{ele} = 30$
Total number of elephants	2,760	362	91
Average active elephants	1,062	186	56
Average duration (seconds)	443.9	595.0	715.4
Traffic contribution	77.6%	47.1%	28%
Count contribution	0.2%	0.03%	0.008%

Table 3.4: IPDSL-II Traffic Trace Elephant Statistics

t_{wait}	Mean Accuracy	Confidence Interval
1	0.3969	0.0224
2	0.4710	0.0201
5	0.5236	0.0167

Table 3.5: Influence of waiting time t_{wait} on the accuracy.

by the ZOOM algorithm. Accuracy thereby describes the percentage of retrieved results that are relevant and represents the precision in a standard precision and recall scenario. Hence, the accuracy represents the fraction of the detected flows that are considered true positives.

a) **Time of traffic observation (t_{wait}).** The study investigates the impact of the length of the monitoring interval t_{wait}, during which the flow entries are active in the switch and passing traffic is monitored. Therefore, we examine the influence of t_{wait} on the mean accuracy over all available parameter combinations.

Table 3.5 shows the mean accuracy as well as the 95% confidence interval of the ZOOM algorithm for $t_{wait} \in \{1, 2, 5\}$. It can be seen that the accuracy increases with growing t_{wait} for the evaluated trace. This can be explained by the influence t_{wait} has on the behavior of the algorithm. While low t_{wait} values decrease the overall runtime of the algorithm and thus allow for faster detection of elephants, the short observation time makes the algorithm vulnerable to short, bursty transmissions that are not elephants by definition. Higher values of t_{wait} on the other hand, increase the traffic monitoring interval and the algorithm becomes more robust against short-lived flows. As the flows that are to be detected are the ones that transmit large amounts of data, these flows also have a higher duration.

Hence, the parameter choice of t_{wait} controls the trade-off between short runtime and fluctuation resistance.

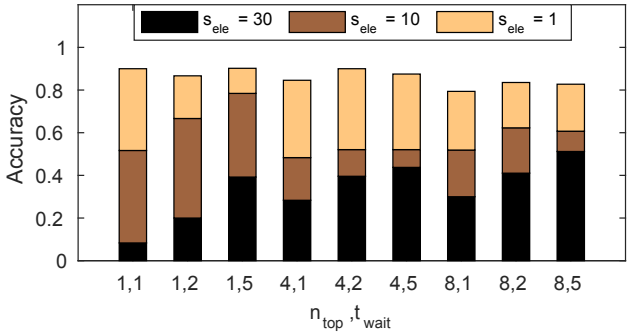


Figure 3.9: Influence of s_{ele} and t_{wait} on the accuracy.

b) **Elephant threshold (s_{ele}).** This section examines the influence of the elephant threshold s_{ele} on the accuracy of the ZOOM algorithm. As defined previously, the threshold s_{ele} describes the total amount of data a flow has to transmit during its lifetime in order to be considered as elephant. Figure 3.9 shows the data obtained by calculating the accuracy for $s_{ele} \in \{1, 10, 30\}$.

The figure shows that for a high elephant threshold the t_{wait} parameter has significant influence on the accuracy. For lower s_{ele} on the other hand, the influence is less decisive. This is due to the composition of elephants for different thresholds. As a high s_{ele} leads to a small set of large, long-living elephants that feature a high mean duration, the high t_{wait} value leads to increased accuracy as short-lived flows are filtered by the long observation period. On the other hand, decreasing s_{ele} leads to a larger amount of flows that are considered elephants. This reduces the mean duration of elephants which results in t_{wait} not having a significant influence. Due to the now larger set of elephants, the overall accuracy is increased.

c) **Number of requested elephant flows (n_{top}) and available flow rules (n_{flows}).** In the following, the relation between the ratio of n_{flows} and

n_{top} and the accuracy of the ZOOM algorithm is evaluated. Figure 3.10 shows the accuracy for different parameter combinations and elephant thresholds. Thereby, the parameter n_{top} is indicated by the line style while the color describes the value of t_{wait}. The x-axis describes the total number of elephants resulting from different elephant thresholds as described previously ($s_{ele} \in [1, 30]$). The data indicates that most parameter combinations achieve similar accuracies for similar elephant thresholds. Only the parameter combinations of $n_{top} = 1$ and $t_{wait} \in \{2, 5\}$ show significantly different behavior. This is most likely due to the higher granularity resulting from $n_{flows} = 16$ and $n_{top} = 1$. This fine grained segmentation reduces the risk of falsely selecting a flow aggregate of many small flows instead of a single elephant flow. As our proof-of-concept implementation does not support backtracking, a once deselected segment is never investigated again in further steps of the algorithm. We call this the *Aggregation Problem*. This may lead to false positives as the algorithm eventually selects an aggregate of many small flows with a large cumulative size over a single large flow.

Furthermore, the results show that if the elephant threshold is set too high, the accuracy of the algorithm decreases significantly. This can also be explained by the aggregation problem. As the number of elephants decreases with a growing threshold, the average size of non-elephant flows and thereby the extent of the aggregation problem increases even further.

This work introduces a lightweight algorithm for elephant detection which leverages built-in features of OpenFlow enabled switches. In particular, the algorithm combines the dynamic creation of flow entries and the counters maintained by switches to perform cost-effective elephant detection without introducing network overhead. Thus, the proposed mechanism does not require modification of software or any additional hardware. Furthermore, the amount of data that needs to be analyzed is independent of the amount of traffic present in the network.

The ZOOM algorithm is capable of detecting elephant flows and their source and destination IP addresses. To this end, the algorithm defines coarse-grained

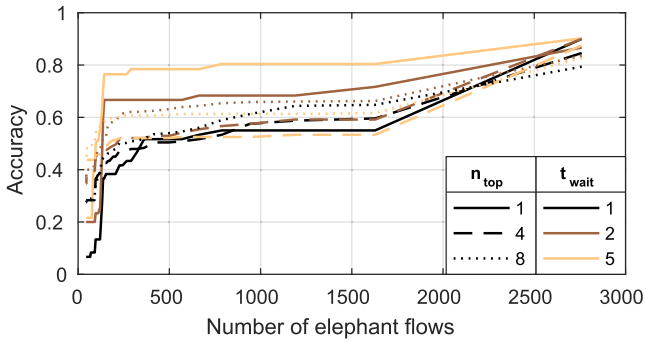


Figure 3.10: Accuracy for $n_{flows} = 16$, different n_{top}, t_{wait} and elephant thresholds.

flow entries in OpenFlow switches, monitors them, and iteratively refines them until they match specific IP addresses. Evaluations of our proof-of-concept implementation show that an accuracy of more than 80% can be achieved.

3.4 Service and Network Separation using SDN

One of SDN's main features is to offer flexible implementation of virtualized networks. This is also helpful for BYOD scenarios, where potentially insecure devices should be included in organizational IT infrastructure, none of the currently available solutions applies SDN to offer a fine-grained restriction of network access. Instead, as illustrated in Figure 3.11, commercial solutions either group connect all devices into a special VLAN that can be further restricted, or – in the case of more sophisticated *Mobile Device Management* (MDM) solutions – an agent runs on the smartphones or tablets that checks its trustworthiness.

S-BYOD, the solution introduced in this work, aims at solving the problem only on the network level. No modification of the user-owned device or restriction of privileges is needed. Instead, S-BYOD restricts each device into its own virtual network. After authentication, the user can explicitly enable corporate services for this specific device following the *sudo* principle of operating systems. Based

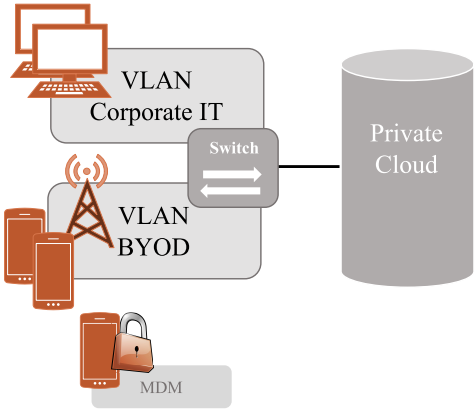


Figure 3.11: Traditional BYOD based on VLAN or MDM.

on an off-by-default approach, the number of services, to which a device running a potentially malicious app, can connect is reduced. To activate access to a particular service, an authenticated user can simply use the web portal of the BYOD solution.

The introduced BYOD solution is implemented as plugin for the ONOS controller. Its main innovation is the implementation of *personalized* virtual networks which are established and adjusted *on demand*. In contrast to other BYOD solutions, which usually group all devices together in a dedicated VLAN and thus do not allow to set up fine-grained policies, S-BYOD individually restricts network access for every device by the means of SDN. Further, these virtual networks are automatically adjusted according to the user's current requirements, i.e., when additional services are requested or when the user roams between access points, as well as when changes within the IT services infrastructure are made. The content of this section is taken from [9].

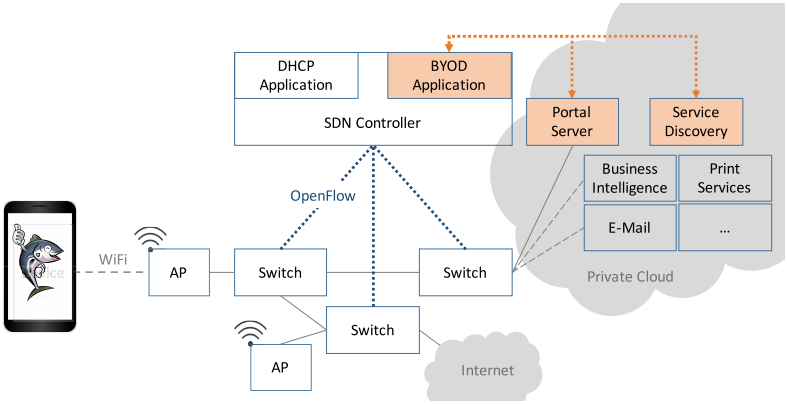


Figure 3.12: Corporate network setup with BYOD client.

3.4.1 Building Blocks

To achieve the outlined goal of SDN-based virtualized networks for BYOD users, the setup builds up on the following components and mechanisms, which are also shown in Figure 3.12:

OpenFlow-enabled switches. The wired network infrastructure is built using OpenFlow-enabled switches. This allows to programmatically establish and modify individual virtual networks, coordinated by the centralized controller.

ONOS SDN controller. ONOS ([99], described in Section 3.1.1) is a well-known controller for software defined networks and allows to modify the forwarding tables of switches using the OpenFlow protocol. By default, ONOS includes basic network services, including reactive forwarding and DHCP and offers APIs for SDN applications running inside the controller.

Wireless access points. Given the lack of OpenFlow-based wireless *Access Points* (APs), traditional APs are used, to which the BYOD users can con-

nect. These APs form an *Extended Service Set* (ESS), meaning they all advertise the same *Extended Service Set Identification* (ESSID) and are connected to the fixed, OpenFlow-based network. Towards the user, the ESSID looks like a WiFi network covering a large area, as several APs are used to increase coverage.

Corporate services. Users of the wireless network need to access business applications that are running in the corporate infrastructure. Such applications can include browser-based access via HTTPS, email services using IMAP/SMTP, or other IP-based protocols, e.g., for printing. Furthermore, Internet access might need to be available to a private device in the corporate environment. It is assumed that most of these applications are running within a private cloud environment in the corporate IT infrastructure.

Service discovery. As cloud-based applications scale accordingly to their usage, the virtual machines, on which an application is running, change over time, i.e., when using modern deployment techniques [59]. Hence, the IP addresses of these applications might change as well, if they are not served by a load balancer, but DNS load balancing is used. In order to allow different instances of cloud applications to connect to each other respectively to connect to their microservice instances [33], service discovery systems [100] are an essential part of modern cloud-based application architectures. This work makes use of *Consul* [101] as discovery service.

Captive portal. Finally, a web-based captive portal as shown in Figure 3.13 is used to authenticate and authorize users. This portal has access to the corporation's authentication services, i.e., *Lightweight Directory Access Protocol* (LDAP) or *Microsoft ActiveDirectory*, and further uses *Two-factor Authentication* (2FA), based on the *Time-Based One-Time Password Algorithm* (TOTP) [102] to prevent an attacker from accessing corporate services with stolen credentials. The captive portal communicates with the S-BYOD controller module using RESTful APIs. In the current implementation, the captive portal is implemented using *Meteor* [103] as client- and server-side

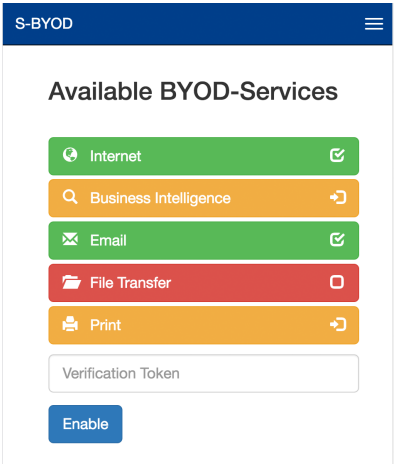


Figure 3.13: Screen shot of the captive portal to activate access to corporate services. Green: activated services, red: deactivated services, yellow: services to be activated after 2FA.

web framework. Figure 3.13 shows the portal, in which the authenticated user can explicitly enable particular applications, while every action that elevates privileges has to be authenticated using a 2FA verification.

3.4.2 OpenFlow Rule Setup

In order to allow basic network connectivity, redirect unauthenticated users to the captive portal, as well as to drop all unwanted traffic, the following rules are *proactively* defined by the ONOS controller in order of increasing priority:

Table-miss action drop all r_{miss}. This *deny all* default rule drops all unmatched packets by defining an all-wildcard rule without any actions.

ARP handling r_{ARP}. All ARP packets are forwarded to the SDN controller and processed by ONOS' *proxyarp* app.

DHCP service r_{DHCP}**.** All DHCP packets are forwarded to the SDN controller and processed by ONOS' DHCP app.

HTTP to controller $r_{HTTP\_to\_ctrl}$**.** For intercepting outgoing HTTP connections of any unauthenticated client, this low-priority rule redirects any TCP traffic to port 80 to the controller. This allows S-BYOD to intercept the HTTP connection and redirect the client to the portal web site for authentication.

DNS server connectivity r_{DNS}**.** As soon as a new client is detected, rules are provisioned to allow DNS traffic to the corporate DNS server. Therefore, persistent rules for each direction between a client device and the DNS server are installed throughout the path for both TCP- and UDP-based requests. Connectivity to the DNS server configured through DHCP is essential also for unauthenticated users, as otherwise no attempts to establish outgoing HTTP connections will be made, which is necessary for the portal redirection.

Portal connectivity r_{portal}**.** To enable connectivity between the client device and the captive portal via a secure HTTPS connection, the complete path is provisioned by one rule per direction between client and portal server.

Service connectivity $r_{service}$**.** As soon as a user has successfully authenticated at the portal and enabled a particular service, these rules permit network-side access to the IP addresses of the servers offering this service, e.g., email, web-based service, or any other protocol. Given the higher priority of rules, the previously applied rules to drop packets are overridden to successfully forward packets between the client and the destination server.

Internet connectivity $r_{external}$**.** All traffic that is routed outside of the layer 2 network, in which clients and the accessed applications reside, require connectivity to the default gateway. Therefore, once the user enables Internet connectivity in the portal, rules that allow traffic between the MAC addresses of the client and the default gateway are installed on switches along

the path. Given the gateway's configuration to not route traffic inside the OpenFlow-managed network, access to this MAC address does not allow the user to bypass security means which prevent accessing other internal devices.

All client-specific rules are installed once the new client is connected. All service-specific rules are installed once a user enables or disables application access in the captive portal. Once the BYOD service gets notified by the controller that a host disappeared from the network, all these rules are removed from the switches.

Once the controller receives packets originating from an unauthorized user accessing a web page via the $r_{HTTP\_to\_ctrl}$ redirection, it instructs the switch to rewrite the source and destination IP and MAC addresses to the portal using a OFPT_PACKET_OUT message. As the packets on the way back from the portal web server to the user also need to match the intercepted connection's addresses, further OFPT_PACKET_OUT messages also rewrite the endpoint addresses of this intercepted connection. As the portal only responds to the HTTP request with a HTTP *Location* redirect header to its resolvable address, no flow entries are set up in the switches for this short-lived connection.

To ensure high user satisfaction, wireless clients need to be able to roam between wireless access points, i.e., when moving from one room or building to another. The security mechanisms and rules installed by the BYOD solution therefore have to support this use case.

As the introduced S-BYOD relies mostly on static rules for performance and scalability reasons, a roaming client needs to be detected quickly and accordant rules need to be updated as well. RFC 5227 [104] introduces procedures for mobile clients to detect the correct setup of wireless networks spanning multiple access points ESS. This is achieved by sending an ARP packet to the previous gateway, once the client switches to the new AP. As this mechanism is implemented in most of the current operating systems, this also offers a safe way to detect the roaming device. In particular, the controller immediately notices this through the received ARP packets which originate from a different switch (port) and thus allows S-BYOD to update all flow rules and maintain connectivity for the roaming device.

3.4.3 Discussion

While ease of use is one important factor for user satisfaction, security of a BYOD implementation is an important aspect as well. Therefore, different mechanisms of S-BYOD that mitigate or still allow certain known attacks will be discussed in the following. Further, an estimation of the resource requirement in number of flows in OpenFlow switches will be provided.

IP address spoofing. To allow connectivity to the portal, every device connected to the WiFi has to receive an IP address from the DHCP service. By spoofing the IP address of another, already authenticated client, an attacker could gain access to internal services. Guessing such IP address is easy, as the used IP address range is known once connected to the network. Therefore, all client-specific flow rules $r_{service}$ and $r_{external}$ have to also match against the client's MAC address and thus lead to all traffic using a self-assigned IP address being dropped.

Wireless client isolation. In order to prevent communication between mobile devices and especially to prevent an attacker gaining knowledge of other (potentially authenticated) users' MAC addresses, the access points operate in *client isolation* mode. By activating such a setting, no incoming traffic of wireless clients is sent directly back to the air interface, but only to the wired connection. There, the wired OpenFlow-based network takes care of separation of clients.

Further implications of ARP/MAC spoofing will be discussed in later on. OpenFlow-enabled WiFi-APs, if they would exist, are therefore not even needed, as long as wireless clients never need to communicate directly.

Estimation of OpenFlow rules. One concern with large-scale OpenFlow deployments is the number of required forwarding rules. Therefore, the usage of flow table entries is discussed in the following. As all rules defined by S-BYOD use exact matches, i.e., MAC and IP addresses, these rules can be stored in *Content-addressable Memory* (CAM). In contrast, using wild-

cards, e.g., matching for `nw_dst= '192.168.0.*'`, would require expensive and very limited *Ternary Content-addressable Memory* (TCAM).

In real-world scenarios this is mitigated by utilizing multiple switches and thus, the client-specific rules are distributed over the switches. Then, only the switches on the path between a client and its connection endpoints receive the accordant rules. For the extreme case that all users are connected to only a single switch, the number of required rules is calculated as follows and provides a worst-case estimation:

$$n_{total} = n_{base} + n_{client} + n_{service} \tag{3.6}$$

With the following rules to provide the basic network setup:

$$\begin{aligned} n_{base} = & \underbrace{|r_{miss}|}_{1} + \underbrace{|r_{ARP}|}_{1} + \underbrace{|r_{DHCP}|}_{2} \\ & + \underbrace{|r_{discovery}|}_{2} + \underbrace{|r_{HTTP\_to\_ctrl}|}_{1} \end{aligned} \tag{3.7}$$

$r_{discovery}$ contains rules which are automatically added by the controller for discovery of links and broadcast domains.

For each of the c connected clients, 4 additional rules are added to enable basic connectivity, regardless of the client's authentication status:

$$n_{client} = c \cdot (\underbrace{|r_{DNS}|}_{4} + \underbrace{|r_{portal}|}_{2}) \tag{3.8}$$

Finally, to estimate the number of rules for enabled services, $s_{i,j} = 1$ denotes service j being enabled for client i and $|r_{service,j}|$ denotes the number of rules required for this service:

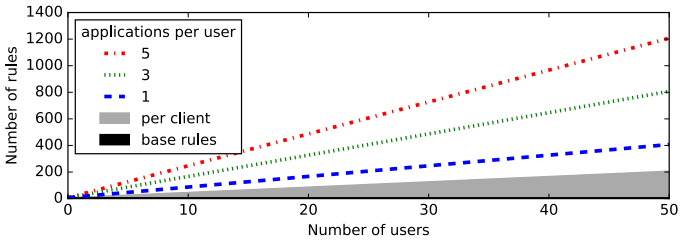


Figure 3.14: Amount of required rules in a single-switch setup, 2 IPs per application.

$$n_{services} = \sum_{i=1}^{c} \sum_{j=1}^{j_{max}} s_{i,j} \cdot |r_{service,j}| \tag{3.9}$$

Figure 3.14 depicts this relationship for different numbers of active BYOD clients and enabled applications per user, while assuming that every service is reachable through two IP addresses.

3.5 Lessons Learned

This chapter investigated effects of network softwarization with a focus on network management and monitoring. By shifting functionality from networking devices' firmware into software running on commodity hardware, flexibility and agility is improved and vendor-dependence reduced.

Section 3.2 stressed the importance of exploiting the benefits of softwarization also for deployment of configuration changes and new network software, such as the SDN controller or VNFs. In contrast to the traditional change processes, the adoption of the successful software engineering concept of *continuous delivery* also for network infrastructures helps to iterate in smaller steps and at higher pace. Such practice better fits to agile methods and are only possible due to the fact that the network is managed in the way how software and servers are managed.

Implementations for the "safety net", which are integrated into the continuous delivery pipelines, were proposed in the form of automated acceptance tests. In order to allow frequent changes, automated tests replace extensive manual testing, which is otherwise needed to ensure non-interrupted network operations.

Similar to the adoption of continuous delivery, the popular technique of *behavior-driven development* (BDD) and how it can be applied to network operations is described. BDD allows an network operator to express automated tests in its own domain language without the need to program.

Section 3.3 suggested a lightweight approach for elephant detection, a common task in the field of network monitoring. Based on existing features provided by SDN switches, byte counters for different flow aggregates were dynamically set up and evaluated. By refining these counters iteratively, it was possible to detect elephant flows with an accuracy of more then 80 % while applying the prototype implementation to a publicly available traffic trace.

Finally, the isolation of devices and services within a network was investigated for the "Bring Your Own Device" use case in Section 3.4. Compared to existing commercial approaches, the proof-of-concept implementation applies network-based separation. Therefore, no changes at the end devices are required, while security rules inside the network enforce more fine-grained separation. Based on the user's current demand, the virtual network is adjusted to include access to particular services, i.e., Internet access, business applications, or print services. The integration with these services follows the common design pattern of cloud applications by using a service discovery system. By subscribing to changes, the developed BYOD module within the SDN controller is always up to date - an essential requirement to keep the security rules provisioned to the network devices in sync with the IP addresses of the service endpoints that a users needs to access.

4 Performance Measurements

Softwarized networks move functionality from specialized hardware devices into software applications running on server hardware. This shift has – still widely unknown – implications on the performance of these network functions. The reasons for this include the novelty of the concepts of *Software Defined Networking* (SDN) and *Network Functions Virtualisation* (NFV) and the rapid innovation happening in these ares, as well as the underlying hardware platform, which is using standard server hardware instead of specialized *Application-specific Integrated Circuits* (ASICs). Therefore, this chapter describes conducted performance benchmarks of key components of softwarized networks that allow to better design such infrastructures in the future.

Background information and related work will be introduced in Section 4.1 as a foundation for the following parts. Section 4.2 focuses on the network control plane and introduces the benchmarking tool *OFCProbe*. By emulating a specified number of network elements and a load level to apply, this allows to evaluate the performance of SDN controllers. The performance of this central component is critical, as delayed responses to the devices' requests will impair network performance. Additionally, the results of conducted performance benchmarking of two available controllers will be described.

In the following, two scenarios for evaluating the performance of different implementation types of data plane focussed *Virtualised Network Functions* (VNFs) will be described. First, we evaluate two different firewall implementations in Section 4.3. Firewalls are a typical network function used in nearly all networks, thus, a typical example of NFV applied to the domain of *Customer Premises Equipment* (CPE). We compare the traditional, hardware-based implementation, which is available as an expansion module for switches, with the software version, which

is available as virtual machine image. Both products are commercially available by the same vendor. Such investigation of provided capacities and performance metrics of these different deployment types allows to better estimate the number of software instances that are required to replace a single physical device.

In Section 4.4, we compare two software implementation variants of a prototypical *Serving Gateway* (SGW) VNF. Such SGW function is used in mobile networks, another typical use case for NFV. Originally, mobile network operators were the main driver behind the NFV concept. The SGW function acts as mobility anchor for the mobile subscriber and is responsible for forwarding all subscriber traffic towards the Internet. This results in a high traffic volume to be processed and forwarded by this VNF acting on the data plane. The two investigated implementation variants differ in the way how the software accesses the *Network Interface Card* (NIC) to send and receive packets. One uses the standard *Application Programming Interfaces* (APIs) of the operating system, while the other makes use of an acceleration software library, namely Intel *Data Plane Development Kit* (DPDK) [35].

These investigations help to better understand the influence of the characteristics of server hardware and operating systems on the network performance, both regarding control plane and data plane.

4.1 Background and Related Work

In the following, background information regarding the topics covered in later sections is described. Additionally, the most relevant related work is covered. At first, an overview over previous work regarding SDN controller performance is given. Then, previous work regarding packet processing performance of VNF instances – the part in the NFV concept that processes all traffic – is described. Finally, an overview over research work regarding firewall benchmarking is given, as such measurements have been conducted several times and help to better asses the results conducted for NFV-based firewalling in Section 4.3.

4.1.1 Controller Benchmarking

The *SDN Troubleshooting Simulator* [105, 106] addresses the problem of troubleshooting SDN control software that arises with the development of SDN platforms as network management services. These troubleshooting techniques are quite simple, as they are based on log inspection in the aim of finding relevant information. The simulator automatically identifies a minimal sequence of inputs to reproduce a given bug. The authors target troubleshooting of distributed controllers running complex applications with their tool.

The influence of the controller's performance on the overall network performance is investigates in [107]. They studied the NOX, NOX-MT, Beacon, and Maestro controllers each running with a Layer 2 switching application. Layer 2 switching is the common denominator, which is implemented by all tested controllers. As the resulting flow rules are installed on a per MAC address basis, this is the forwarding mechanism with the lowest request rate towards the controller, e.g., compared to flow installation on a per TCP flow level. The results provided in the following show that existing controllers perform better than predicted in the referenced literature. However, they also state that understanding the overall SDN performance remains an open research problem.

Cbench [108] was the first available controller benchmark and emulates a given number of virtual OpenFlow switches to measure different aspects of controller performance, specifically mean latency and throughput. As Cbench's functions are very limited and controllers have been further developed since its release, the results described in the [108] only give basic information on controller performance. Preferable features that still were missing in Cbench are, e.g., to be able to distribute the program on multiple cores or machines and to provide statistics for each virtual switch individually.

OFCBenchmark [109] creates a certain amount of virtual switches, which in turn generate independently configured messages and produce their own statistics. Scalability, detailed performance statistics, and modularity were the top design goals. With the OFCBenchmark distribution enabled, the benchmark can be spread over multiple hosts with each host running its own client. The virtual

switch, the key component of the tool, holds a simplified flow table, to be able to respond to controller requests, statistics counters and the connections to the controller.

The investigations with OFCBenchmark yielded important insight regarding the treatment of individual switches by the tested controllers [109]. Non-ideal design and implementation decisions became apparent, however, limit the benchmarking framework's scalability to large networks. This lead to the development of *OFCProbe*, which will be introduced in Section 4.2.

4.1.2 Packet Processing & VNF Benchmarking

By moving network functions out of specialized hardware appliances, which implement functionality using ASICs, a change in the performance characteristics is expected. The execution on a generalized hardware platform, either *Commercial off-the-shelf* (COTS) server hardware or programmable devices, potentially comes with an increased processing delay and reduced throughput. Depending on the usage scenario, the gained flexibility can overweight a reduced performance per instance and might be countered by horizontal scaling of multiple instances of the same VNF. To investigate the performance characteristics of VNF implementations, benchmarks have to be conducted.

In 1994, the need for well defined benchmarks of network interconnect devices was served by RFC 2544 [110]. It defines, how latency and throughput of a *Device Under Test* (DUT) should be measured and, thus, developed to a de facto standard. The defined guidelines, including key performance indicators and relevant benchmarking parameters, were used for benchmarks of network devices like routers, switches, or firewalls. Subsequent documents extend the methodology by tests [111], updates, and remarks [112, 113], adapting it to keep pace with the capabilities of network devices.

The increased performance of COTS hardware endorsed the change towards software-based solutions. Compared to the ASICs, this introduces additional architectural layers of abstraction, i.e., the operating system, its scheduler, as well as hardware drivers. The performance of these systems now depends on the hard-

ware and software implementation. Rather than performance modeling of black box hardware, benchmarking of software has moved into the focus. The additional layer was also used to implement further metering points in it. However, results of these white box measurements have to be considered carefully, as metering on the DUT may affect the tested behavior [114].

Furthermore, a recent IETF draft [115] discusses the novel challenges that are introduced with VNF benchmarking, e.g., phenomena like shared resources between multiple VNFs and their impact on the performance of individual VNFs. In [116], the performance of DPDK-accelerated switching and routing VNFs is evaluated with respect to throughput and latency. Intel's DPDK [35] offers APIs, which allow to circumvent the operating system's network stack and reduces the overhead per packet. The analysis shows that, when offered traffic to two Gigabit Ethernet physical interfaces, NFV-based approaches can achieve line rate throughput as well as a latency that does not impact performance in an enterprise network. Similarly, [117] describes a virtualized system using DPDK acceleration that achieves low latencies close to those when running non-virtualized on bare metal. The authors of [118] focus on benchmarking NFV infrastructures with respect to resilience and the effects of faults on their performance. Besides an implementation solely in software, other generally available hardware offers potential means for accelerated execution of VNFs. This includes *Field-Programmable Gate Arrays* (FPGAs) [119, 120] as well as *Network Processing Unit* (NPU) and *Graphics Processing Unit* (GPU) [121, 122], each resulting in different trade-offs between flexibility and performance.

Software platforms for VNFs are provided by [123] and [124], which enable fast packet processing on COTS hardware running VNFs. In the case of forwarding, both solutions can process packets at line rate on a 10 Gbps link while introducing latencies in the order of 50 µs.

4.1.3 Firewall Benchmarking

A performance comparison between a Cisco ASA 5505 hardware and the software-based Linux *iptables* [125] is presented in [126]. The work focuses on

three main performance metrics: throughput, latency, and concurrent connections. While the Cisco ASA outperforms Linux iptables in terms of throughput and latency, the latter is capable of handling bursts of packets better and achieves a higher number of concurrent sessions. The authors also show that the firewall performance does not only depend on hardware, but also on the implementation of the involved algorithms.

In [127], a comparison between various firewalls is provided with respect to different performance metrics and their ease of operation. The results show that the Cisco ASA outperforms the Checkpoint SPLAT and the open source packet filter available in OpenBSD, but lacks in terms of functionality and user interface. Nevertheless, all considered firewalls show basic intrusion detection capability and can block basic attack types.

An analytical approach for predicting the overall performance of a rule-based firewall in the context of different attack schemes is proposed in [128]. Taking into account performance indicators like throughput, packet delay, packet loss, and CPU utilization, the authors validate and verify their model by means of simulations and measurements.

A methodology for the specific case of firewall benchmarking is specified in RFC 3511 [129]. This document characterizes the performance of a firewall and describes formats for presenting benchmarking results. The measurements conducted in Section 4.3 follow this RFC as a guideline.

4.2 SDN Controller Performance

The key component in the SDN architecture is the controller, sometimes also referred to as the "networking operating system". The controller provides a platform for the operation of diverse network control and management applications. However, little is known about the stability and performance of current controller applications, which is a requirement for a smooth operation of the network–regardless of what shape the actual architecture of the controller cluster is.

In the following, we present the flexible OpenFlow controller performance analyzer *OFCProbe* as a follow up to previous experiences with *OFCBench* [109]. It

features a scalable and modular architecture that allows a fine-granular analysis of a controller's behavior and characteristics. It allows the emulation of virtual switches that each provide sophisticated statistics about different aspects of the controller's performance. The virtual switches are arrangeable into topologies to emulate different scenarios and traffic patterns. This way, a detailed insight and deep analysis of possible bottlenecks concerning the controller performance or unexpected behavior is possible. Key features of the implementation are a more flexible, simulation-style packet generation system as well as Java Selector-based connection handling. In order to highlight its features, evaluations with the Nox and Floodlight controllers in different scenarios are described. The content of this section is taken from [16].

4.2.1 Design Goals

Based on the lessons learned from the implementation OFCBench, we defined the following design goals:

Platform-Independency. In working environments one is often limited to certain operating systems and or hardware components. To bypass these requirements, we need a software that is executable on the most common system architectures.

Scalability. Scalability describes the software's ability to be run on multiple CPU cores, or even in a coordinated way on multiple hosts simultaneously. The tool should be multi-threaded in a meaningful way without excessive overhead leading to starvation, e.g., as with OFCBench, which suffered from launching one thread per emulated switch.

Modularity. Modularity is the key to be able to easily adapt to new OpenFlow controller versions, new scenarios and/or new types of measurement values. The separation of the program logic and the controller communication is also a central goal to simplify the adaptation of the tool itself to possible new communication protocol versions.

Performance Analysis. We want to investigate OpenFlow controllers and their performance by sending generated messages to the controller and recording the controller's responses. Performance has different meanings in this context. It is defined by throughput and latency in a best-effort situation, but it also means recording the controller's behavior in different scenarios.

Detailed Statistics. Detailed statistics include a set of features that permit the investigation of the OpenFlow controller's behavior, e.g., whether one is treating particular switches differently or changes its behavior over time.

4.2.2 Architecture of OFCProbe

An overview over OFCProbe's architecture is depicted in Figure 4.1. The *OpenFlow Connection Handler Module* is responsible for the connection to the controller. The generation of messages is handled by the *Traffic Generation Module*. Finally, there is a module for the storage of measurement values as well as a central management module, which handles the inter-module control messages. To realize platform independency, we chose Java in combination with the *OpenFlowJ* library [130] as programming environment. When starting, OFCProbe loads a configuration file and provides its values to all other modules. After that, the OpenFlow connection is established and, after a successful OpenFlow handshake, the traffic generation begins. Generated messages and their responses are traversing the configured statistics module, e.g., round trip time, packets per second, and are recorded in a data format that enables further analysis.

Modularity enables users and developers to easily adapt the experimental setup to their needs. This includes updated OpenFlow protocol versions,, but also a reconfiguration or the implementation of a new statistics module. A detailed description of each module follows.

OpenFlow Connection Handler Module. This module one key component of OFCProbe. It is in charge of establishing and holding the connection to the OpenFlow controller. Java's NIO Selector [131] is used for connection handling. With a selector, it is possible to handle multiple channels

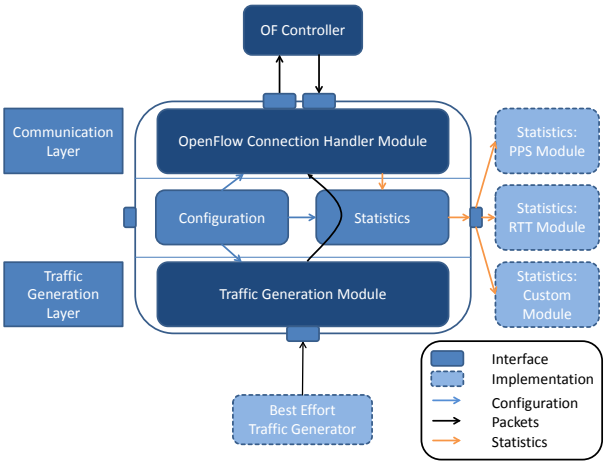


Figure 4.1: Architecture of OFCProbe.

in one thread, thus reducing the multithreading overhead when dealing with the sockets of hundreds of emulated switches. The module also contains the flow table implementation for the virtual switches. Apart from connection handling, the main task of this module is the acceptance of messages from the *Traffic Generation Module* and their subsequent encapsulation into OpenFlow messages, which are then sent to the controller. The replies from the controller are also handled here and transmitted to the connected statistics modules for further analysis.

Traffic Generation Module. The Traffic Generation Module is an event-driven scheduler/queue processor running within its own thread. The module has one queue in which all future events are stored. Each event consists of an event time, an event type and an associated virtual switch. When the event time is reached, the event is taken from the queue and based on its type, different actions are executed in the virtual switch.

A virtual switch goes through a life-cycle of events that is depicted

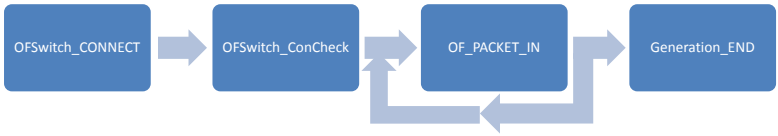


Figure 4.2: OFCProbe event chain.

in the state diagram shown in Figure 4.2. The first event is the `OFSwitch_Connect_Event`, which specifies when the virtual switch should start its connection establishment with the controller. Subsequently, an `OFSwitch_ConCheck_Event` for this virtual switch is scheduled to verify that connection is finally successfully established. Once this is done, the first `Packet_In_Event` is scheduled for time $t = 0$. When a `Packet_In_Event` is executed, the virtual switch's simulated packet queue is checked. If it is below a configured threshold, the queue is filled up to that threshold with new packet payloads to ensure that enough pre-generated workload is available. Then the next `Packet_In_Event` is scheduled in a pre-defined amount of milliseconds. This continues until the `Generation_End_Event` is fired, which terminates the experiment. The values for the inter-arrival time of `Packet_In_Events` and the target fill threshold of the switch queues are globally defined in the OFCProbe configuration or through a per-switch configuration. In the current version of OFCProbe, the payloads generated are TCP SYN packets either with static or randomized address values. For each TCP SYN packet, three headers have to be generated: the Ethernet and IP headers, with each a destination and source address, and the TCP header with a destination and source port. OFCProbe uses a pre-generated master TCP SYN packet that contains a correct TCP SYN packet with all fields except the address fields set. The remaining fields then are filled with generated addresses taken from the corresponding MAC, IP, or TCP generator.

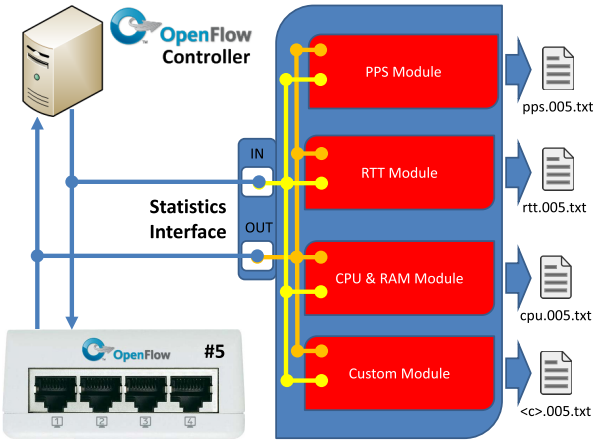


Figure 4.3: OFCProbe statistics module.

Statistics modules. Each outgoing OF Packet-In message and incoming OF Packet-Out message is provided by the connection handler module to all connected statistics module. This process is shown in Figure 4.3. With these messages and the corresponding arrival times, it is possible to measure, e.g., the throughput, the latency, or the inter-arrival time of subsequent messages to the controller. Currently, following statistics modules are available: *Packets Per Second* (PPS), *Round Trip Time* (RTT), *Inter-Arrival Time* (IAT), and a CPU and RAM utilization monitor for the controller.

4.2.3 Additional Features of OFCProbe

In this section, we describe the additional emulation features contained in OFCProbe.

Packet Capture Playback. Apart from the generated TCP SYN packets, another option is to provide the payload for the generated OF Packet-In mes-

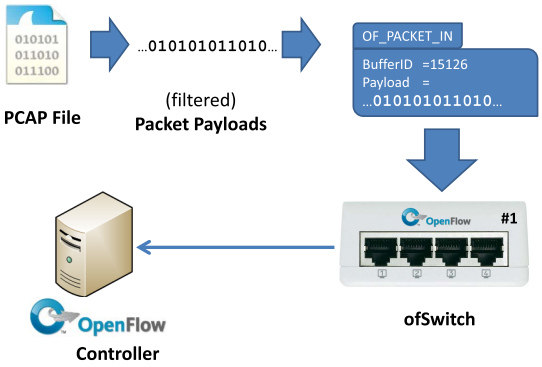


Figure 4.4: OFCProbe packet capture playback.

sages. To emulate a real network environment, it is also possible to specify a PCAP file containing a packet trace for each virtual switch individually. PCAP is a capture file format for network traffic that is used, e.g., by Wireshark [132]. These files are read by OFCProbe and each time a new Packet-In message is generated, the payload is taken from the PCAP file. Figure 4.4 illustrates this feature. Ideally, the PCAP file should only include the first packets of arriving flows.

Topology Emulation. Another feature of OFCProbe is the emulation of a network topology. Controllers employ mechanisms to detect the underlying topology between the connected OpenFlow switches.

The protocol that enables devices to detect these links is called *Link-Layer Discovery Protocol* (LLDP). The discovery process works as depicted in Figure 4.5. The OpenFlow controller sends an OF Packet-Out message containing an LLDP payload (1) with the information that it was injected into the topology at a certain switch and the action to flood it to all ports of the OpenFlow switch. The OpenFlow switch recognizes the flood action in the message and sends the LLDP payload out of all its ports (2). A con-

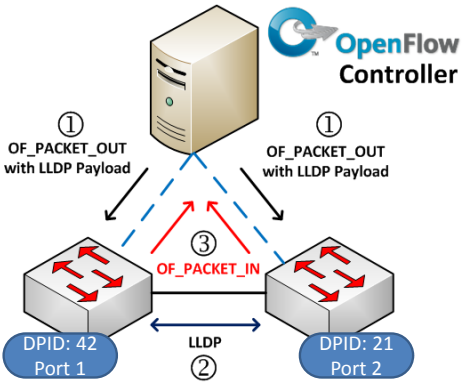


Figure 4.5: Topology discovery via LLDP.

nected OpenFlow switch queries the controller about packets that do not match any flow entry by encapsulating them into OF Packet-In messages and forwarding them to the controller (3). The controller then inspects the payload and reads from the contained LLDP packet information about the switch, from where this specific packet was injected. This way, the controller learns that there is a link between one port of the switch where the packet was injected and the one from where it received the packet. The topology emulated by OFCProbe can be defined in a separate topology initialization file.

ARPing Feature. As an enhancement to the topology emulation, we implemented what we call the *ARPing feature*. This feature allows to emulate a large number of devices connected to the ports switches within the topology. For every port in the emulated topology, which is not yet connected to another switch, a device with a unique MAC and IP address is generated. After the LLDP-based topology detection was executed by the controller, OFCProbe emulates an ARP request originating from every emulated end device to every other emulated end device. This is realized by sending an

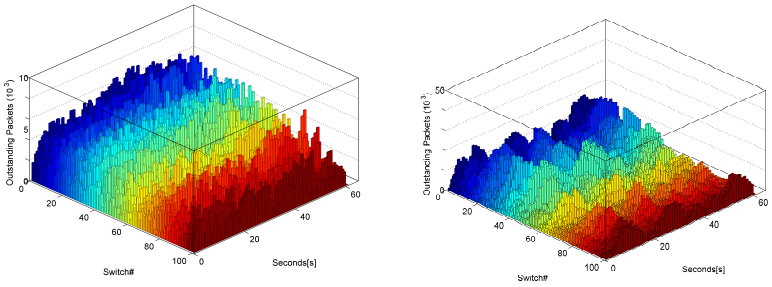
OF Packet-In message containing the ARP request to the controller. The controller's reaction is an OF Packet-Out message with the action to flood this packet out of all virtual switch ports except the ingress port. The ARP request is then queued in the next virtual switches that are connected to the original virtual switch in the emulated topology. There, the payload is again encapsulated in an OF Packet-In message and sent to the controller. This procedure continues until the ARP request has reached the virtual switch to which the corresponding end device is connected to. The virtual switch recognizes the target device as one of its connected devices and then generates an ARP reply in form of an OF Packet-In message to the controller. As a result, the controller is aware of the location, i.e., connected switch port, of the virtual device.

4.2.4 Evaluation of SDN Controller Performance

Based on the previously described objectives and created tooling, performance benchmarking of two SDN controllers will be conducted in the following. Therefore, OFCProbe will be used to evaluate the performance of the NOX [133] and Floodlight [134] controllers.

As stated before, OFCProbe is capable of an analysis on a per virtual switch level. This allows a granular inspection of how the controller treats each switch at a given time during the analysis and how this treatment changes. The number of outstanding messages that a controller has not yet answer is a result of an overload situation. This has a negative impact on network performance, as the packets which triggered the switch's messages to the controller are either buffered in the switch until a response is received, or – as soon as the switch buffer is fully occupied – are sent entirely to the controller. This worsens the overload situation even more, as the already overloaded control channel is now also used to store the payload data that does not fit into the switch buffer anymore.

Best-Effort Tests. To investigate the number of messages that a controller can process, the number of outstanding packets is evaluted. Figure 4.6a displays the number of outstanding packets, i.e., OF Packet-In messages that



(a) Outstanding packets (Floodlight)

(b) Outstanding packets (NOX).

Figure 4.6: Comparison of the numberer of outstanding packets for different controllers.

have not been answered yet, for the Floodlight controller for a run with 100 emulated switches.

The x-axis shows the number of the virtual switch, the y-axis the seconds passed since measurement start and the z-axis the number of outstanding packets in units of 10k packets. The controller's behavior appears to be quite fair as no switch has at any time a significantly higher amount of outstanding packets than the others with a maximum of outstanding packet values lower than 10k packets.

The outstanding packets for the NOX controller are shown in Figure 4.6b. The arrangement of the axes is the same as for Figure 4.6a. Here, certain inequalities between the switches and for each switch over the time can be observed. There are "waves" of outstanding packets in both x- and y-axis direction. This might be caused by the controller's implementation. The controller iterates over a list of connected switches, processing only one switch at a time. And while one switch is processed, the other connected switches have to wait for replies from the controller. Furthermore, the numbers of outstanding packets is notably higher than the numbers for Floodlight, having peaks with up to 30k outstanding packets.

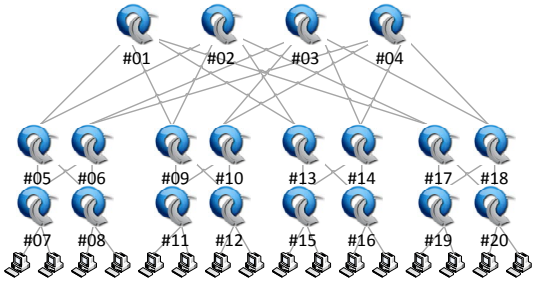


Figure 4.7: Fat-Tree topology, typical for data center networks.

Topology Emulation and ARPing. In this section, we investigate the controllers' behavior in a fat-tree topology setup. Such topologies are typical for data centers, one of the frequent use cases for software defined networks. The particular fat-tree topology considered in this experiment has 20 virtual switches with 4 ports each. As shown in Figure 4.7, the virtual switches are connected to the emulated host devices with their two free ports. Each of these emulated devices generates TCP_SYN packets for flows to all other host at every packet generation event. Therefore, 15x2 packets are generated per virtual switch per event. The inter-send time for switches #07, #11, #15 and #19 is set to 15 ms, switches #08, #12, #16 and #20 have a inter-send time of 40 ms. Figure 4.8 illustrates the outstanding packets for the ingress switches of the Floodlight controller. The x-axis shows the passed time since measurement start, the y-axis the outstanding packets in multiples of 10k packets, respectively. The switches with the inter-send time of 15 ms (type #1) between their packet generation events have a drastically higher number of outstanding packets than the switches with the inter-send time of 40 ms (type #2), which have almost no outstanding packets except for one.

A general characteristic is observable for virtual switches with notable outstanding packet counts. Within 10 seconds, their count rises above the

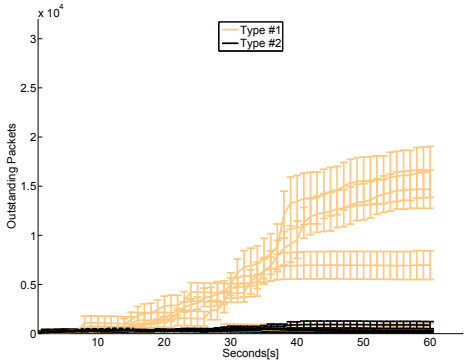


Figure 4.8: Outstanding packets for different traffic characteristics using Floodlight.

other switches, continuously climbing and reaching a mean value of 4k at 30 seconds from the start. At around 36 seconds, one switch settles at 6k outstanding packets, while the remaining switches constantly increase up to 15k packets.

As it can be seen, the confidence intervals are large for switches with a high number of outstanding packets. The reason for this is that the actual number of outstanding packets varies significantly between the different switches. Figure 4.9 shows a 3D representation to investigate the number of outstanding packets for each single switches. Here, again, the drastically higher outstanding packet numbers for the switches #07, #11 and #19 can be observed as they reach 20k, 10k and 8k outstanding packets. Switch #15 is an exception. The difference between the two sets of inter-send times is significant and observable. The other non-ingress switches all have next to no outstanding packets.

To summarize, Floodlight's switch handling appears to be fair in terms of performance as only switches with a high packet load have to wait for controller processing. All requests from the other switches are handled

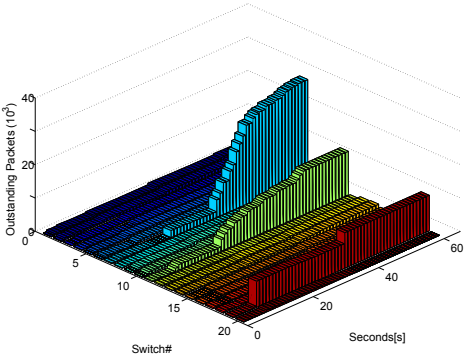


Figure 4.9: Outstanding packets per switch over time for Floodlight.

immediately.

4.3 Comparison of Hardware and VNF Implementations

The NFV paradigm promises higher flexibility, vendor-independence, and increased cost-efficiency for network operators. Its key concept consists of virtualizing the functions of specialized hardware-based middleboxes like load balancers or firewalls and running them on COTS hardware.

4.3.1 Network Function Under Test

In the following, we investigate the performance implications that result from migrating from a middlebox-based deployment to a NFV-based solution. This work focuses on the performance evaluation of a firewall which is commercially available both as a hardware entity as well as a VNF. In particular, Cisco's *Adaptive Security Appliance Service Module* (ASASM) [135] and its virtualized counterpart

Adaptive Security Virtual Appliance (ASAv) [136] are utilized in our experiments. Based on traffic statistics of a university campus network, the feasibility of the two deployment types in this environment is investigated. The content of this section is taken from [11].

4.3.2 Methodology

In order to investigate how a virtualized firewall compares performance-wise to its more expensive and inflexible hardware-based counterpart in a particular scenario, we evaluate and compare their performance in a dedicated testbed. First, the testbed setup is presented alongside the specifications of its hardware and software components. Second, the course of experiments for the aforementioned performance evaluation as well as parameters and performance indicators are described.

Figure 4.10 shows the main components of the testbed utilized in this work. It is comprised of two networks, an external and an internal network, which are separated by the firewall. The networks are represented by two switches that are connected to the firewall and different types of traffic are generated using two Spirent C1 [137] traffic generators. The Spirent C1 is a dedicated traffic generator equipped with four 1 GbE interfaces. Through its implementation using FPGAs, it allows highly accurate and reliable measurements of various performance indicators like packet delay.

The first traffic generator produces stateful TCP traffic using the Avalanche software. In this context, "stateful" refers to the fact that the firewall needs to keep track of the state of each connection according to the TCP state machine. This traffic is used to set the DuT under different load levels in terms of varying numbers of active connections. The second C1 device is controlled by the TestCenter software in order to benefit from the highly precise hardware-based delay measurement that, however, only support stateless traffic. Probe packets are sent once per second through a TCP connection that was previously opened by emulating the TCP handshake.

As mentioned earlier, two different firewall deployments are analyzed in the

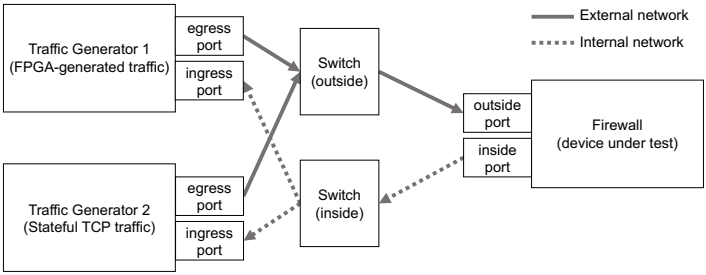


Figure 4.10: Hardware setup used in this work.

following and detailed in Table 4.1. The first component is the Cisco ASASM, a hardware solution. While Layer 3 and 4 processing is performed in ASIC respectively *Ternary Content-addressable Memory* (TCAM), the general purpose Xeon CPUs of the ASASM are used for tasks like VPN, content inspections, and management. The Cisco ASAv ASAv30 runs virtualized using VMware ESXi on a Cisco *UCS* server and is configured as per vendor recommendations. While the data sheet of the ASASM provides basic information on the processing delay of the appliance, no information is available in case of the ASAv. This work sheds some light on the performance differences between these commercially available solutions.

In order to achieve realistic testing conditions, around 1300 rules are configured on the firewalls. These rules correspond to those installed on our campus firewall. Hence, the resulting measurements can provide insights regarding the feasibility of the two firewall deployment types in such a network.

Using the testbed setup described previously, the performance of the two firewall types is evaluated with respect to the achieved processing time. This section presents the methodology for investigations regarding the effects of different load levels on the processing time of TCP packets. The test scenarios were developed together with firewall administrators and campus network administrators at the computing center of our university.

The main test case investigates the processing times of TCP packets while the

Product	**Hardware**	**Software**
Vendor	Cisco	Cisco
Series	ASASM	ASAv
Model	WS-SVC-ASA-SM1	ASAv30 v9.4(1)3
Deployment Type	Physical, Switch Blade	Virtual
CPU	2x Xeon 5600 2.00 GHz, 6 cores	4 vCPUs
Memory	24 GB	8 GB
Pricing		
Perpetual *Source: ciscoprice.com*	$97,750	$15,980
On-demand *Source: Amazon AWS*	-	$1.39 per hour
Hardware Base Platform		
Platform	Catalyst 6509 Switch	Cisco UCS C220 M3
CPU	VS-S720-10G, 600 MHz	Xeon E5-2680, 8 cores
Memory	-	64 GB DDR3
Hypervisor	-	VMware ESXi 5.5
Specifications		
Max. throughput	20 Gbps	2 Gbps
Connections/sec	300,000	60,000
Concurrent conn.	10,000,000	500,000

Table 4.1: Devices under test.

firewall is exposed to various amounts of load in terms of open TCP connections. Before the measurement is started, the TCP connections are opened by utilizing Avalanche in order to simulate users in the internal network that request file downloads from servers in the external network (cf. Figure 4.10). After downloading a small 1 kilobyte file, the TCP connection is left open and users keep on requesting other files from different servers until the desired number of TCP con-

nections is established. In order to avoid completely idle connections, a 1 kilobyte file is requested via HTTP for each connection every 10 seconds. The total number of active connections is varied between 1,000 and more than 500,000 in order to cover a diverse set of scenarios. Finally, the TestCenter software is used to instruct the FPGA to generate one TCP packet per second and capture the resulting packet delays.

4.3.3 Benchmarking Results

As the firewall is the access gate to a network where all ingress and egress traffic passes, situations where it becomes a bottleneck need to be avoided. Especially the software implementation of the firewall raises the question, how performant it is compared with its hardware counterpart. Therefore, the performance of the two firewall deployment types, according to the methodology described before, is evaluated in the following. Each experiment lasts 5 minutes and is repeated 5 times.

In Figure 4.11, an overview over the distribution of processing times of the ASASM is provided. While the x-axis displays the processing times, the y-axis represents the fraction of observations in which the corresponding value was not exceeded. Furthermore, differently colored curves mark different load levels in terms of concurrent connections. For the sake of readability, only the most relevant range of x-values is depicted.

There are three main observations. First, the processing time has a low variability for all scenarios, i.e., an interquartile range of roughly 4 microseconds with values between 60 and 85 microseconds. Second, processing times show a strictly decreasing behavior with increasing numbers of concurrent connections. This phenomenon could stem from interrupt mitigation mechanisms similar to those of the *New API* [138], the network I/O API that is utilized by Linux operating systems. In order to decrease the overhead that results from each individual packet causing an interrupt, this mechanism accumulates packets until either a certain amount of packets is collected or processing is initiated by a timeout. Hence, lower load levels cause higher delays as packets need to wait for the timeout to

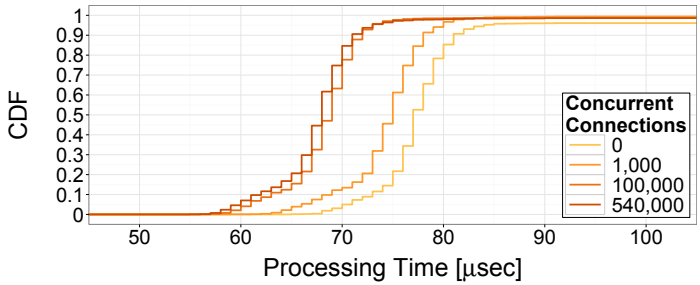


Figure 4.11: Processing times achieved by the ASASM hardware.

trigger processing, while packets exceed the threshold at an increasing rate in case of high load levels, resulting in lower delays. Finally, the small gap between the two highest load levels indicates a converging behavior with respect to processing times.

Similar to the previous figure, Figure 4.12 displays the distribution of processing times achieved by the virtualized firewall ASAv. Again, the x-axis is limited to the most relevant interval. In contrast to the maximum number of connections of 540,000 that was used in the context of the ASASM-related measurements, the maximum for the ASAv-related measurements is at 500,000. The reason for this parameter choice is a hard-coded capacity limit implemented in the ASAv software (cf. Table 4.1).

The ASAv exhibits two modes regarding the processing times. For low numbers of TCP sessions, higher delays and a significantly higher variance are observed. In these cases, the ASAv introduces packet delays between 100 and 800 microseconds, roughly ten times higher than when using an ASASM. When increasing the load, however, the variance and the absolute values decrease and stabilize in an interval between 100 and 350 microseconds. As in case of the hardware-based ASASM, there seems to be a mechanism that positively affects packet processing delays of the firewall at higher load levels.

In order to put the measurement results into the context of a real network, traf-

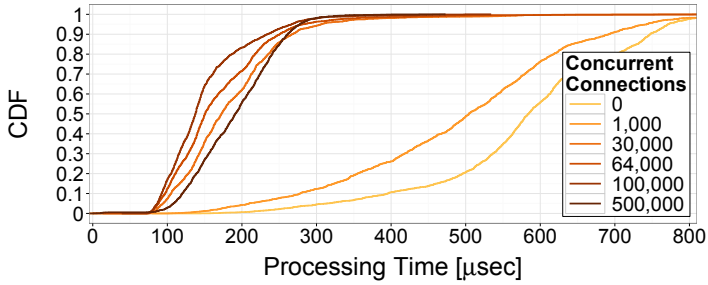


Figure 4.12: Processing times achieved by the ASAv.

fic measurements have been conducted in the university's campus network using NetFlow. During two working days of observation, the highest number of parallel connections at the Internet uplink with a capacity of 2×3.5 Gbps was found to be around 420,000 connections. Further, the highest number of new connections per second was found to be around 6,500-6,800, with two exceptions, during which 14,700 and 20,300 new connections were opened within one second, respectively. At the 10 G connection of a single building (computer science faculty), a maximum of around 177,000 concurrent connections with peaks of up to 4,400 new connections per second were observed.

Given the ability of ASAv to handle up to 500,000 concurrent connections, it seems to be a feasible approach for both scenarios. However, the total throughput of only up to 2 Gbps is the limiting factor, which would restrict the use of ASAv in the investigated network, e.g., to the protection of clusters of servers or buildings connecting only few users through smaller links.

While the ASAv does not offer such working mode, full-fledged VNFs should be able to operate in a cluster to allow horizontal scaling. Software firewall implementations, which support such scale out to overcome the capacity limitations of a single instance, could then also cope with higher throughput, i.e., the 2 Gbps as seen here.

4.4 Comparison of Software-based VNF Implementations

To benchmark the performance of a VNF, the challenges of classical benchmarking get extended by three additional problems: *complexity*, *abstraction*, and *concurrency*.

The increased system *complexity*: Software-based network devices introduce only a single layer of abstraction compared to dedicated bare metal devices. However, the use of (host) virtualization techniques and the execution in the cloud adds further abstraction layers that each introduce their own performance limitations, i.e., the hypervisor, the virtual switch, and the physical and virtual interfaces [114]. This work investigates whether hardware acceleration mechanisms, e.g., Intel's DPDK [35], improve performance or if the performance remains the same due to higher complexity.

The virtualization of a network function inherits the *abstraction* of the data plane from the executing hardware. Therefore, absolute performance values are less meaningful without an understanding of how they relate to the performance of underlying components. This, for instance, leads to the following question: If the processing power of the unit performing NFV is too low, how can it be accelerated if the employed cloud solution has doubled the amount of physical CPU cores?

In addition to increased complexity and a higher level of abstraction, the interpretation of results and application on real world setups gets even more challenging due to *concurrency*. In real setups, (virtual) network functions interact with other components, which may be other VNFs or the data plane.

We compare two implementations of a network function acting as a Serving Gateway (SGW) in the mobile core. After introducing the basic terminology used in this context, the experiment setup and details of the two alternative implementations are presented. Finally, the implementations are compared with respect to various performance indicators. The content of this section is taken from [17]

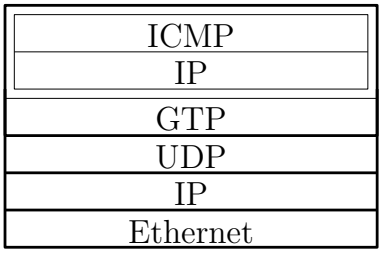


Figure 4.13: Structure of the GTP-u packets considered in this work.

4.4.1 Network Function Under Test

The LTE *Evolved Packet Core* (EPC) is comprised of various specialized components. The components' responsibilities range from purely control plane related tasks, e.g., in case of *Mobility Management Entity*s (MMEs), to combinations of control and user plane processing as in the case of SGWs. In this work, the user plane functionality of the latter is moved to a virtual network function. In particular, its task lies in transporting user data traffic through the LTE network and to serve as a mobility anchor for mobile subscribers. For this purpose, the traffic is encapsulated via the *GPRS Tunneling Protocol* (GTP). More specifically, the UDP-based *GTP-u* protocol is used. Figure 4.13 depicts the structure of *GTP-u* packets that are sent to the VNF discussed in this work. On top of UDP in the stack, a GTP header indicates the presence of an encapsulated IP packet that follows immediately. In order to identify GTP packets' membership to a particular tunnel, the GTP header includes a *Tunnel Endpoint Identifier* (TEID). In the following, GTP tunnels are also referred to as bearers.

Little information regarding the number of GTP-u packets that a traditional SGW can handle is publicly available. According to [139], the signaling load (GTP-c) of an LTE network is around 94,000 messages per second per million smartphone users.

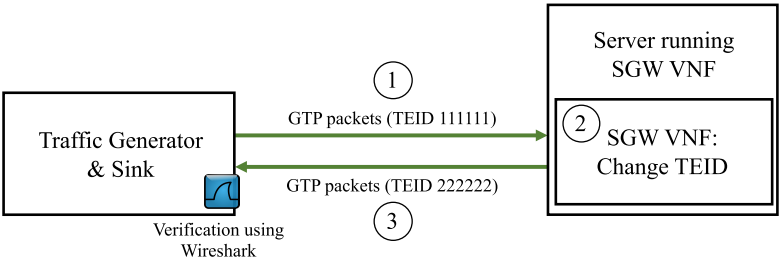


Figure 4.14: Testbed setup.

4.4.2 Methodology

Figure 4.14 shows the two main components of the testbed as well as the main steps of the experiment. First, GTP-u traffic is generated via the Spirent C1 traffic generator [137]. This traffic is processed by the VNF installed on a commodity server[1] running a Linux operating system (OS)[2].

After processing, the server forwards the altered GTP-u packets to the ingress port of the traffic generator chassis, where performance indicators are recorded by hardware capture cards. These include the one way delays, the received packet rate, as well as the amount of packet loss. In order to obtain the net processing times at the server, packet transmission times are deduced by performing a loop-back test, i.e., transmitting packets of different sizes from the generator's sending port to its receiving port. Furthermore, capturing packets at the receiving port of the traffic generator via Wireshark allows verifying that packet headers are modified as expected.

To measure the baseline performance, the SGWs are configured to change the TEID as well as the IP source and destination fields of incoming GTP-u traffic according to currently present bearer entries.

Both implementations of the SGW VNF are based on the *eXtensible OpenFlow data path daemon* (xDPd) ([140], version 0.7.5), which allows designing data path

[1] Intel Xeon E5-2620 v2 CPU at 2.10 GHz, Intel I350 NICs, 32 GB of RAM
[2] 64 bit version of Debian 7.7 (wheezy, kernel version 3.2.0-4-amd64)

elements and supports GTP traffic. In particular, the xDPd software is responsible for matching incoming packet headers against the installed flow table entries and, in the presented case, modifying their TEID. In the xDPd version used in this work, the matching algorithm consists of a loop that goes through each flow table entry and checks for a match. Hence, the runtime of the matching algorithm is linear in the number of flow table entries.

The first SGW VNF uses MMAP-based xDPd, where packets are copied to the OS user-space and thus, are expected to have longer processing times. In case of the used Linux OS, the network I/O API that is utilized by this implementation is the *New API* (NAPI) [138]. Hence, in the following, this implementation is referred to as the NAPI-based approach.

Second, a DPDK-accelerated implementation[3] is evaluated, which promises faster packet processing through reduced overhead for packet I/O [35, 141]. When running DPDK-based applications, a core mask can be specified in order to change the number of cores that are used. In all presented experiments, this parameter is set to 0x03, which corresponds to utilizing a total of two cores, one core for management and one for I/O on all ports. Increasing the number of cores did not affect the results, neither with respect to the processing times nor with respect to the maximum packet rate that can be handled without packet loss. This parameter may become relevant in the context of link capacities beyond 1 Gbps.

The systems are offered loads with varying packet sizes and rates. Each test run lasts 5 minutes and experiments for each set of configuration parameters are repeated 10 times in order to obtain confidence intervals for the key performance indicators. In order to find the performance limits for use in a practical context, the load is increased until packet loss occurs. Additionally, the influence of the number of present bearers on the resulting processing times and packet loss is investigated. When multiple bearers are registered on the server, the traffic generator sends GTP packets with corresponding TEIDs in a round robin fashion, i.e., one GTP packet for each bearer.

[3] Version 1.7.1 of the DPDK libraries is used.

4.4.3 Benchmarking Results

The occurrence of packet loss shall be minimized in order to provide a reliable service as well as fast processing of requests. When designing a network architecture containing SGW components, the operator has to consider feasible alternatives that meet the requirements of the particular use case. By utilizing the presented benchmarking methodology, it is possible to quantify the performance and limits of implementations and thereby assist the decision making process. In nearly all practical scenarios, packet loss needs to be avoided.

NAPI-based SGW. Figure 4.15 presents an analysis of the influence factors on the occurrence of packet loss as well as limits regarding the maximum load that can be handled without packet loss. While the x-axis denotes the number of packets that are sent to the server each second, the bars indicate the resulting packet loss percentage. Differently colored bars correspond to different packet sizes and whiskers represent 95% confidence intervals obtained from 10 experiment repetitions.

Low packet rates, i.e., 120,000 pps and less, are handled without packet loss. However, rates beyond roughly 130,000 pps result in a steady increase of loss. As the confidence intervals for both of the displayed packet sizes overlap and the corresponding mean values do not deviate from each other significantly, the packet rate is identified as the main influence factor on the packet loss rate. As discussed in [142], user space packet frameworks process packets of different sizes at an almost identical speed as only headers are copied and processed. Larger packet sizes are omitted due to the fact that these can be handled at line rate when only one bearer is installed. For example, a rate of roughly 89,000 pps corresponds to the capacity of the 1 Gbps link when a packet size of 1400 Byte is used. However, investigations regarding the processing times of packets show that not only packet rate but also packet size influences SGW performance.

After identifying the range of packet rates that can be handled by the NAPI-based implementation, benchmarking is performed with respect to

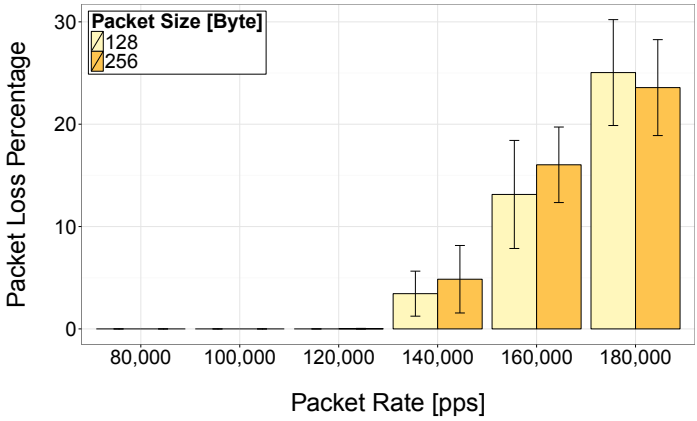


Figure 4.15: Packet loss in case of small packet sizes when using the NAPI-based SGW implementation.

processing times in this interval. Figure 4.16 displays mean processing times for packet sizes between 128 and 1400 Bytes, covering commonly observed values. The x and y-axis represent the packet rate and the mean processing time in microseconds, respectively. Again, 10 repetitions are performed in order to obtain confidence intervals and bar colors indicate the packet size. There are three main observations. First, very low packet rates, i.e., 100 and 1,000 pps, result in high processing times, roughly 230 and 110 μs, respectively. A possible explanation for this behavior is the interrupt mitigation mechanism that is part of the NAPI. In order to decrease the overhead that results from each individual packet causing an interrupt, this mechanism accumulates packets until either a certain amount of packets is collected or processing is initiated by a timeout. Second, the lowest processing times are observed for rates between 10,000 and 20,000 pps, corresponding to scenarios that are not affected by interrupt mitigation anymore and simultaneously do not expose the system to a high load. Finally,

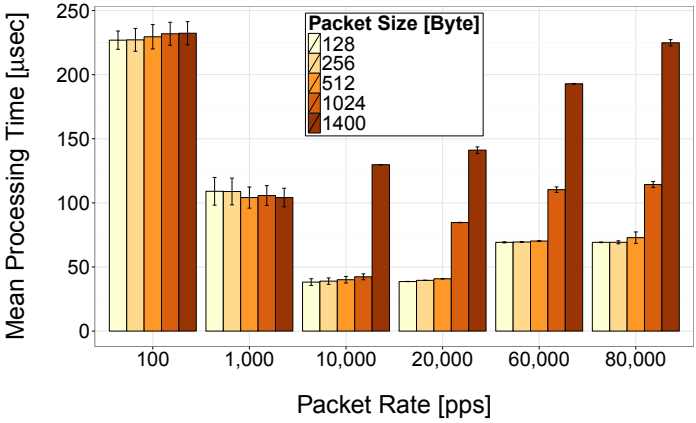


Figure 4.16: Influence of packet rate and size on mean processing times using the NAPI-based SGW.

the processing time increases steadily when the maximum rate approaches the previously determined limits. While the packet size does not have a significant impact on the processing times observed at low packet rates, higher rates have a larger effect on the processing times of bigger packets. This can be explained by the fact that combining a given packet rate with different packet sizes results in changes to the link utilization, which also poses an influence factor on processing times.

DPDK-enabled SGW. While the NAPI-based SGW implementation is capable of processing packets of 1400 Bytes in size at line rate, it suffers from packet loss when receiving small packets at rates beyond 130,000 pps. In contrast to this behavior, the DPDK-enabled approach can handle even small packets, i.e., 128 Bytes in size, at line rate. This corresponds to a packet rate of around 970,000 pps, or an increase by a factor of more than 7 when compared to the maximum tolerable rate achieved by the previous user-space approach.

In addition to the performance gain in terms of maximum packet rate without packet loss, DPDK significantly accelerates the processing of packets at the SGW. Figure 4.17 presents the processing times measured in various conditions with respect to packet size and packet rate. In order to fit the various combinations of these two parameters into one graphic, the x-axis displays the link utilization, ρ, which is calculated as the ratio between the bandwidth that results from a particular configuration and the link capacity of 1 Gbps. The y-axis shows the mean processing times and bar colors denote different packet sizes. As processing times of the SGW implementation are very stable for the scenarios under test, the narrow confidence intervals are barely visible. The observed processing times start at roughly 5 microseconds and do not exceed 8 microseconds, corresponding to a speedup by a factor larger than 8 when compared to the processing times of the NAPI-based solution ranging from 40 to 230 μs. Furthermore, the packet size is the main influence factor on the resulting processing times, as indicated by almost constant values among configurations sharing the same packet size parameter. An increase in packet size results in an increase in the mean processing time. Nevertheless, an additional effect is visible: an increase in the packet rate also results in a slight but consistent increase of processing times.

Scenarios Featuring Multiple Bearers. All results presented so far are based on environments that feature only one single bearer. While these provide important insights into the behavior of the different implementations and the influence factors on their performance, they are not sufficient for deriving practical guidelines with respect to the choice of implementation for a particular use case and dimensioning the system for a given load. Therefore, an investigation of the relationship between the number of bearers that are present in a system and various performance indicators is performed. Given a number of installed bearers, the maximum packet rate an implementation can handle without the occurrence of packet loss is empirically identified. Then, the processing times are measured for these

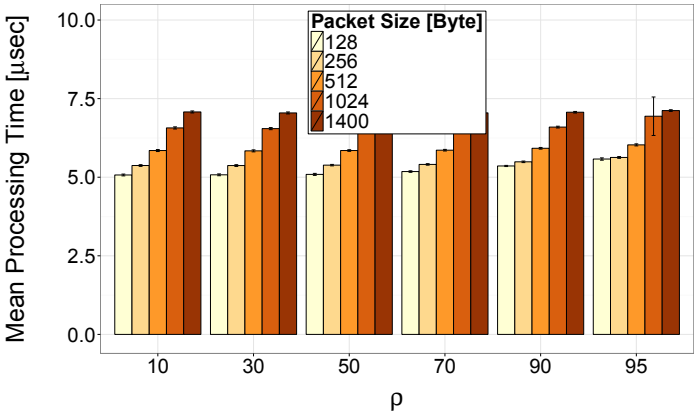


Figure 4.17: Processing times for different link utilizations and packet sizes using the DPDK-enabled SGW implementation.

scenarios.

Figure 4.18 provides an aggregated view on the packet rates that the SGW implementations discussed in this work can handle when different numbers of bearers are present. The numbers of bearers for which the rate limits are determined are between 1 and 400, as indicated by the x-axis. The y-axis shows the corresponding maximum packet rate that can be handled by a particular combination of SGW implementation and packet size, which are represented by line style and color, respectively. Solid lines denote the DPDK-enabled implementation, dashed lines denote the NAPI-based implementation. For the sake of clarity, only the two extreme values for the packet size are shown, i.e., 128 Byte and 1400 Byte.

In case of both implementations, a steady decrease of the maximum tolerable load is observed when the number of bearers is increased. The main reason for this behavior lies in the matching algorithm used by the xDPd software which both implementations have in common. As discussed be-

fore, the matching procedure that is utilized has linear time requirements with respect to the number of flow rules in the SGW's table. Given the fact that increasing the number of bearers in the system corresponds to increasing this number of flows, it follows that a growing number of bearers also causes higher processing times. Consequently, the process of matching incoming packets becomes the system's bottleneck and affects the maximum packet rate that can be handled without the occurrence of packet loss.

When using the NAPI-based approach, both packet sizes yield almost identical values for the resulting maximum packet rate. This is consistent with the results from without DPDK, where no significant influence of the packet size on the maximum tolerable load is observed. Only for numbers of bearers below 50, the maximum rate for small packets exceeds that for big packets. This stems from the fact that the maximum packet rate is not only limited by the processing time of the server, but also by the link capacity, i.e., 1400 Byte packets are processed at line rate for up to 10 bearers.

In contrast, the gap between the two curves corresponding to the DPDK-enabled solution is significantly larger and closes only after 200 bearers are present in the system. As discussed previously, the DPDK-accelerated SGW can process packets of all considered sizes at line rate when one bearer is installed. Hence, the initial gap represents the rate limitation due to the link capacity. With an increasing number of bearers, however, the portion of the total processing time that is caused by the packet matching routine outweighs that caused by the network I/O API and the maximum rate decreases. After the maximum rate for small packets drops below the rate required for line rate with large packets, the rate limitation is not caused by the link capacity anymore and thus, both curves overlap.

For all considered packet sizes and numbers of bearers, the DPDK-enabled SGW implementation outperforms the NAPI-based approach in terms of maximum packet rate. While the advantage of the former is especially pronounced in the case of small packets and low numbers of bearers, there is

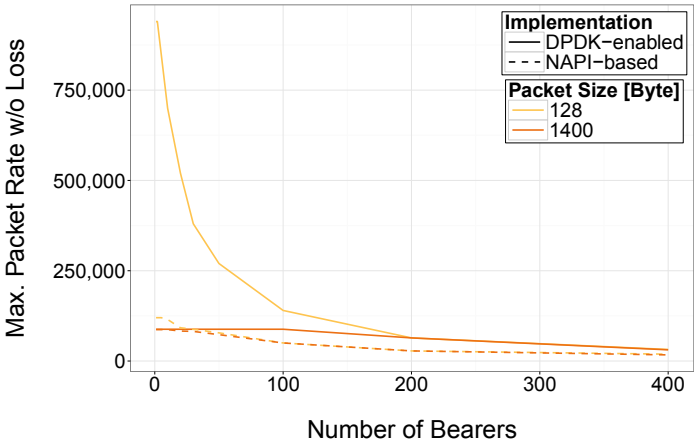


Figure 4.18: Influence of the number of installed bearers on the maximum packet rate without the occurrence of packet loss.

still an improvement by a factor of almost two when considering the highest number of bearers. Independent of the network I/O mechanism, the maximum tolerable rate decreases due to the data plane software that is responsible for packet matching.

Although the rate limits determined in the previous paragraphs provide upper bounds for the load that can be applied to the systems without causing packet loss, the resulting processing times may not be feasible in practice. Thus, Figure 4.19 highlights the processing times of the SGW VNFs when facing these circumstances. Like in the previous figure, the x-axis shows the number of bearers that are present in the system. The y-axis displays the mean processing time observed when packets arrive at the maximum rate for a particular combination of network I/O API, packet size, and number of bearers. In addition to the bar color representing the aforementioned configuration, whiskers denote 95% confidence intervals that are obtained from 10 experiment repetitions.

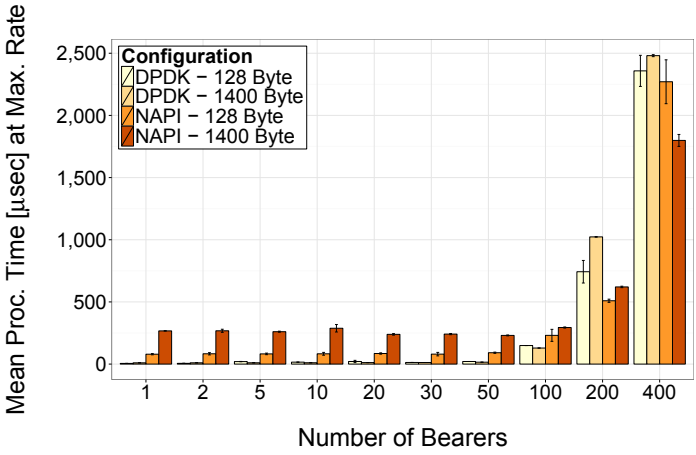


Figure 4.19: Influence of the number of bearers on the processing time at the highest rate that can be handled without loss.

Until a number of 100 bearers is reached, the DPDK-accelerated solution not only outperforms the NAPI-based approach with respect to the maximum tolerable packet rate, but also regarding the mean processing time of individual packets. In this interval, the former achieves processing times below 20 microseconds while the latter has processing times of roughly 100 and 250 microseconds for packet sizes of 128 and 1400 Byte, respectively. As soon as 200 or more bearers are installed, however, the processing times of the DPDK-enabled approach are higher than those of the NAPI-based implementation. The main reason for this behavior is that the packet rate and thus, the total amount of table lookups differ for the two implementations. As discussed in the context of Figure 4.18, the DPDK-enabled SGW is capable of processing almost two times more packets.

4.5 Lessons Learned

This chapter described performance benchmarking of the most important components of softwarized networks. Performance aspects of the SDN control plane were investigated in Section 4.2. At first, we introduced and described *OFCProbe*, a tool for performance an consistency testing of OpenFlow SDN controllers. By emulating the control plane part of OpenFlow switches, it allows to emulate a large number of switches that are sending control messages to the controller. Using different network topologies, two SDN controllers were evaluated. It was shown that one of them treated the connected switches disparate. While some of them received their response faster, others had to wait longer. Such controller behavior can lead to hard to detect performance issues in parts of the network.

Software-based data plane elements were evaluated in Sections 4.3 and 4.4. While the first part investigated an entity that is typical for *Customer Premises Equipment* (CPE), the second one described benchmarking of a mobile network entity.

A firewall was chosen as a typical CPE network function example. By investigating two products offered by the same commercial vendor, the performance characteristics of both deployment types was compared. The hardware appliances, as well as the software appliance, available as virtual machine image were benchmarked regarding their processing delays while set under different load conditions. It was found that the software version increased the processing delay per packet compared to the hardware appliance. The processing delays were typically between 100 and 350 µs, compared to 60 to 85 µs for the hardware. Depending on the actual usage scenario, such an increase in delay can be neglected.

The differences between two implementation variants of the same VNF in the case of a mobile network entity was described in Section 4.4. The evaluation of a *Serving Gateway* (SGW), an entity that forwards all subscriber traffic towards the Internet, has shown the resulting performance spectrum between using the network APIs of the operating system and when using a specialized acceleration library. In this case, the use of Intel DPDK resulted in a seven-fold increase of packet throughput, as well as a reduction of forwarding delays by factor eight. However,

it was also seen that other influence factors can have a negative impact on the performance. As soon as the supplied workload contained traffic of many different user sessions, the throughput decreased significantly. Starting with roughly 200 simultaneous users, the difference between the standard implementation and DPDK vanished as a result of an unfavorable implementation.

The conducted benchmarks gave insights into the different deployment and implementation types of VNFs. Especially when using frameworks that optimize the application's network access, high throughput per VNF instance can be achieved so that it sounds realistic to replace most of the hardware appliances with a limited, yet dynamically adjustable, number of software instances.

The findings with both, unfair behavior of SDN controllers, as well as bad implementation choices when handling many user sessions in parallel, stress the need for such performance evaluations being applied on a regular basis during development. An integration of OFCProbe, as well as an automation of data plane benchmarking, into a continuous delivery pipeline as it was described in Section 3.2 is desired, as any software release or configuration change might result in an unexpected performance impediment.

5 Model for Packet Processing

The previous chapter covered softwarized networks with a focus on performance benchmarking of the particular instances. The idea to process every single packet on a generic server platform instead of carefully crafted hardware devices requires the server hardware and the software running on top to be highly optimized.

To estimate and forecast the effects the processed traffic, conducting benchmarks as in the previous chapter offers often only limited insights. In order to run such benchmarks, the system has to actually exist in this configuration, as well as all parameter settings need to be executed, which can be a lengthy process.

In contrast, analytical models allow to predict a system's performance even before it is actually built. By introducing abstractions, the behavior of a system can be estimated by taking different implementation options into account or forecasting its scalability under various traffic patterns. The introduced model helps to understand the impact of performance-relevant parameters on these metrics to allow an adequate dimensioning and a proper performance prediction. By evaluating "what-if" scenarios, the system performance can be evaluated before implementing one of the various acceleration mechanisms or under the assumption of a traffic characteristic as it is expected in the future.

Based on the previously conducted performance benchmarking, this chapter introduces a first analytical model to estimate the performance of *Virtualised Network Function* (VNF) running on a general purpose server and operating system. This model allows to estimate the throughput and packet loss rate for given service time distributions of a VNF. The content of this section is taken from [8].

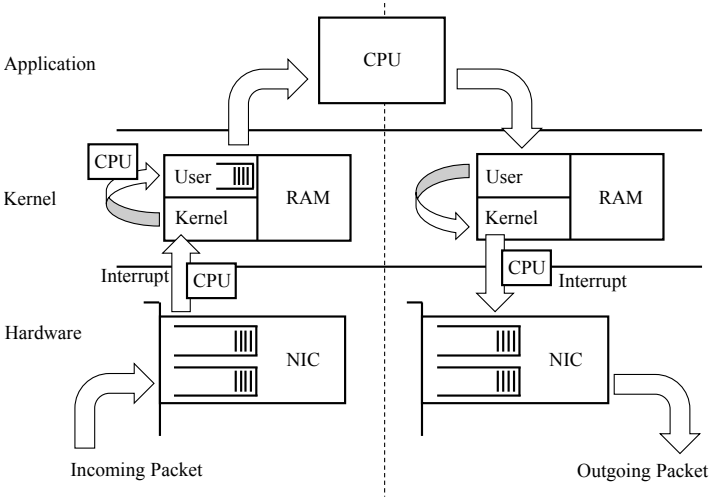


Figure 5.1: Packet processing in a server.

5.1 Background and Related Work

In the following, an introduction into the problem space of packet processing in generic hardware, as well as the field of optimization techniques that allow faster packet processing, is given.

5.1.1 Packet Processing in COTS Hardware

Applications processing network traffic send and receive data packets through functions provided by the operating system's kernel. This procedure is shown in Figure 5.1, from the packet being received at the incoming *Network Interface Card* (NIC), processing of the packet in the software application and finally the packet leaving at the outgoing NIC. Accordingly, packets traverse a complex chain of forwarding steps between the NIC, the kernel, and the software application resulting in a specific delay overhead.

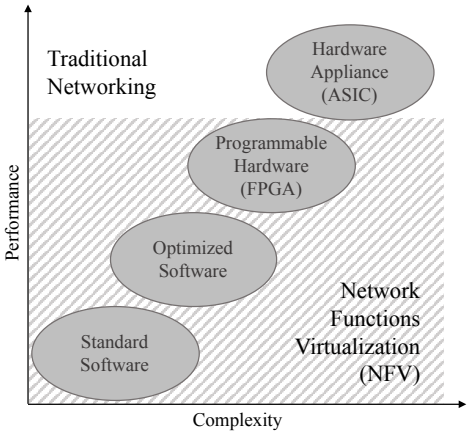


Figure 5.2: Implementation choices for instances of network functions.

One major contributor to these delays are copy operations between the memory of the NIC, the kernel space and the user space. In the case of insufficient *Central Processing Unit* (CPU) resources, the system drops packets exceeding the buffer in user space. To overcome these issues and to speed up packet processing to enable large-scale *Network Functions Virtualisation* (NFV) deployments, numerous approaches and techniques exist besides the standard implementation using the operating system's networking stack.

The solution space for the implementation of a network function's data path is illustrated in Figure 5.2. Starting from the full software implementation on the lower left side to the full hardware implementation on the upper right side, the efficiency and performance, as well as the complexity and thus the costs of the implementation increase. In between are programmable hardware devices (i.e. NetFPGAs) as well as optimized software implementations. The reduced performance per single instance when applying the NFV paradigm is compensated by more fine-grained scale out possibilities, when multiple instances are used per network function depending on the actual load.

5.1.2 Interrupt Moderation

The overhead of reading one packet from the NIC and making it accessible to the application requesting the packet might sum up to a livelock [143]. In such situation, the CPU is effectively busy with packet handling instead of executing the program that processes incoming data. The trigger to read a packet from the NIC originates in the interrupt signal that is sent by the network card after assembling data receiving from the wire. The routine of the NIC driver within the operating system's kernel executes with higher priority than the application processing the data, thus interrupting other workloads from being processed.

In order to avoid such livelocks and to reduce the overhead of packet processing in a server, several approaches that apply interrupt moderation have been introduced on operating system side as well as in networking hardware.

The networking stack (*New API* (NAPI), [143]) in the Linux kernel disables interrupt handling for interrupts related to receiving packets, once the first packet is processed. Followed by that, the NIC queue is polled in assumption that multiple packets arrived in a burst. After a certain number of packets have been processed, or a timeout occurs, interrupts are re-enabled and the process restarts once the next packet arrives.

Hardware-based implementations are offered in many server network adapters. The actual feature set varies between different chipsets. For receive as well as transmit directions, the NIC can hold back interrupts until either a pre-configured number of packets are received or sent, or until a pre-configured time since the first packet starting the batch passed by. Further options allow to define a threshold to differentiate between a low and a high traffic load and to specify options for both of these conditions. Finally, some NICs offer adaptive modes, in which they change their behavior based on the current receive rate.

The effects of interrupt moderation and the reduction of end-to-end delay has been subject of several studies already. The influence on passive and active network measurements is investigated in [144]. By identifying packet bursts, effects of interrupt moderation can be considered when running capacity and delay measurements using commodity NICs. A similar methodology is applied in Sec-

tion 5.2.2 in order to estimate the processing time within the application.

By increasing the interrupt rate, more context switches occur in the CPU, when switching between interrupt handling and data processing. Every context switch comes at a certain cost, especially when code and data are evicted from the CPU caches. [145] estimates a time of 3-4.5 µs for a pure context switch without any computation on a multi-core system and 1.3-1.9 µs when the processes are pinned to one CPU core. When the CPU's cache lines are not filled, experimental results show context switch delays of 2.2-2.9 µs, when the process is pinned to a specific core and a simple program that writes memory pages is used. With virtualization, the time for context switches is reported to be increased 2.5-3-fold. As a rule of thumb, the author estimates 30 µs for a context switch in real-world scenarios. In contrast, the delays seen in the following are mostly based on one single program being executed, resulting in much lower overhead, as the contents of CPU caches are usually not evicted by code or data of other applications.

Latencies of network communication between two servers are studied in detail in [146]. The authors investigate the contributing factors to latencies in Ethernet-based TCP/IP connections and try to achieve a minimal end-to-end latency. Using a modified Linux kernel, the authors make use of nanosecond-precision timers offered by CPUs to break down the packet transmit and receive latencies for a 1 Gbps and a 10 Gbps Ethernet NIC. Based on measurements and estimations, this study indicates a total receive latency of 7.747 µs for a 1 Gbps card. The main contributors (more than 1 µs) are *Interrupt cause register read requirement*, *SoftIRQ*, *Wakeup application to process socket information*, as well as the example application identifying and acknowledging the received data (*ACK the pong received by the remote sender*).

5.1.3 Acceleration Techniques

Besides interrupt latencies, also the process of copying data from the NIC into the kernel's memory and once again to the memory segment of the application increases packet processing overhead.

Multiple techniques are available, e.g., Netmap [142], ClickOS [124], Intel

DPDK [35], or VPP [147] to bypass the kernel completely during packet reception, use shared memory buffers to avoid additional copy operations, process packets in batches, or replace the entire network stack. Accordingly, these mechanisms usually speed up specific parts of the stack. An extensive measurement study on the performance of several of the aforementioned mechanisms in case of packet forwarding is conducted in [148].

However, the above-mentioned studies have several drawbacks. First, the focus on simple network functions like pure packet forwarding obscures the influence of the processing time spent in the user space on the total processing time. This component, however, might account for the majority of the total processing time. Second, measurements are conducted for very specific use cases and cannot be generalized in order to obtain a holistic evaluation of the proposed mechanisms. Finally, it is impossible to determine the feasibility of an approach without identifying its key performance indicators. Therefore, a model for analyzing the packet processing performance on COTS hardware is required. In addition to providing the capability to derive key performance indicators, model parameters can be tuned in order to represent different acceleration techniques and quantify their effects in the context of different use cases.

Based on such evaluations, it could be decided, which technique offers a good trade-off between complexity of implementation and speedup for a specific network function. As seen in Section 4.4, operating modes of network functions exist, in which the overhead of packet handling, and therefore the speedup gained by techniques like DPDK, might become negligible.

While different network functions have different characteristics in terms of their behavior and requirements, there is a common ground when it comes to evaluating their performance. RFC 2544 [110] provides benchmarking guidelines for networking devices such as routers, switches, or firewalls. These guidelines include key performance indicators for various *Device Under Tests* (DUTs) as well as the methodology for measuring latency and throughput of these devices. Several additional documents were released in order to take into account the increased set of features and capabilities of network elements [111, 112].

When attempting to evaluate the performance of virtualized network func-

tions that run on COTS hardware, additional challenges need to be addressed. Due to the additional abstraction layer introduced by the softwarization of network functions, system performance does not only depend on the underlying hardware but also on the particular VNF implementation, cf. Section 4.4. In contrast to ASIC-based packet processing, effects like scheduling and caching also impact the predictability and reliability of software solutions. Furthermore, interdependencies between VNF instances running on the same physical substrate can affect the system's behavior [115, 116].

In [116], the performance of DPDK-accelerated switching and routing VNFs is evaluated with respect to throughput and latency. The analysis shows that, when offered traffic to two physical Gigabit Ethernet interfaces, NFV-based approaches can achieve line rate throughput as well as a latency that does not impact performance in an enterprise network. Similarly, [117] shows a DPDK-accelerated virtualized system that achieves performance levels that are close to that of non-virtualized systems in terms of latency. The authors of [118] focus on benchmarking NFV infrastructures with respect to resilience and the effects of faults on their performance. Further options for hardware acceleration for VNFs include *Field-Programmable Gate Arrays* (FPGAs) [119, 120] as well as *Network Processing Unit* (NPU) and *Graphics Processing Unit* (GPU) resources [121, 122], each resulting in different trade-offs between flexibility and performance.

[123] and [124] present platforms that enable fast packet processing on COTS hardware running VNFs. In the case of forwarding, both solutions can process packets at line rate on a 10 Gbps link while introducing latencies in the order of magnitude of 50 μs.

5.2 Analytical Model for Packet Processing in NFV

In order to understand the process of packet processing within a Linux x86 system, an abstracted description is provided in the following. This process, which starts with receiving a packet on the wire and ends with the processed packet being sent over the wire, is also depicted in Figure 5.1.

Read from media: The network interface card reads data from the transmission media by interpreting electrical or optical signals within the MAC layer and transforms it into packets.

Store in receive queue: These packets are saved into a receive queue implemented in hardware inside the NIC. Multiple such queues can exist and, based on hashing, packets can be distributed among these queues.

Trigger interrupt: In the most simple case, the NIC triggers an interrupt signal to notify the CPU about the arrival after every received packet. Interrupt moderation techniques, which are under study in this work, aim at reducing the number of interrupts by processing multiple packets at once. Depending on the capabilities and configuration of the network card, this batch processing mechanism can be triggered by a timeout, by accumulating a specified amount of received packets, or a combination of both. Some NICs also offer adaptive modes, which adjust timers and batch sizes according to the current packet rate.

Read packet from NIC: As soon as the interrupt is sent, the CPU stops other work in order to load and execute the interrupt service routine of the NIC driver. This code then fetches the batch of packets from the network card. This process, which results in a *context switch* of the CPU, is rather costly as CPU registers first need to be loaded with new code and data. Additionally, this also purges other applications' code/data and thus introduces overhead. The overhead caused by a interrupts can also lead to livelocks, if all CPU time is spent with interrupt handling. It can be reduced by avoiding interrupts for every single packet at the cost of additional delay.

Store packet in buffer: The packet data is stored in a buffer in RAM, until an application requests them for processing. The size of this buffer is limited to a fixed number of bytes[1]. If the application cannot catch up with reading packets, the kernel drops packets. The process of copying packet data from kernel space to user space takes additional time per packet.

[1] in Linux, `net.core.rmem_max = 131071 bytes`

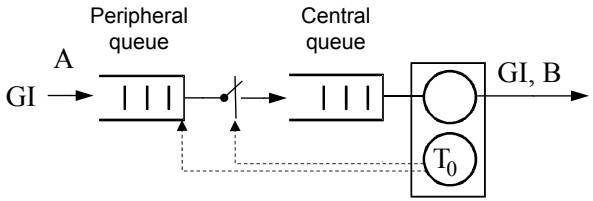


Figure 5.3: Queueing Model.

Process packet in application: While the application processes the packet, it blocks the CPU. Ideally, this part would accumulate for the largest portion of CPU time, as this is actually executing the VNF's code.

Send packet: After processing, the packet traverses the same way backwards, until it is finally sent to media. The NIC informs the operating system about this by means of another interrupt.

5.2.1 Model

Abstract Server Model and Performance Metrics

The queuing model used for the performance analysis of the system outlined in Section 5.2 is depicted in Figure 5.3. It is a generalization of the clocked approach introduced by [149]. The generation of packets follows an arbitrary distribution A. The packets are stored in a peripheral queue which is assumed to have infinite size. Incoming packets are transferred in a batch to the central queue after a time interval τ initiated by the first packet after a batch transfer. The inner queue is then modeled as a $GI^{[X]}/GI/1-L$ system and evaluated by means of discrete-time analysis. Distributions of the batch sizes and burst interarrival times are derived in the next subsection.

Model of the Peripheral Queue (NIC)

In the peripheral queue, which represents the network interface card, packets are aggregated. The resulting batch is then forwarded to the central queue, which represents the CPU/software.

For the remainder of this work, we use the following notation to distinguish between random variables (RVs), their distributions, and their distribution functions. A random variable is represented by an uppercase letter, e.g., X. The distribution of X is denoted by $x(k)$ and is defined as

$$x(k) = P(X = k),\ -\infty < k < \infty.$$

Furthermore, the distribution function of X is written as $X(k)$ and is defined as

$$X(k) = \sum_{i=0}^{k} x(i),\ -\infty < k < \infty.$$

Finally, $\mathrm{E}[X]$ denotes the mean of X and $*$ refers to the discrete convolution operation, i.e.,

$$a_3(k) = a_1(k) * a_2(k) = \sum_{j=-\infty}^{\infty} a_1(k-j) \cdot a_2(j).$$

The following distributions are used for modeling the peripheral queue:

- $a(k)$: distribution of the packet interarrival time.
- $r_a(k)$: distribution of the packet recurrence time.
- $\tau(k)$: distribution of the duration of the aggregation interval.
- $u_n(k)$: distribution of unfinished work in the system before the arrival of the n-th batch.
- $o(k)$: distribution of the interrupt processing delay.

- $s(k)$: distribution of the interarrival time between batches.
- $x(k)$: distribution of the batch size.
- $f^{(j)}(k)$: distribution of the time between the start of an aggregation interval and the arrival of the j-th packet. Since the aggregation interval starts with the arrival of a packet, this time equals the sum of j interarrival times. The corresponding random variable is referred to as $F^{(j)}$.
- $w_i(k)$: distribution of the waiting time of the i-th packet in the peripheral queue.

The first packet arriving after a burst transferal initiates a new aggregation interval. All packets arriving in this time frame are transferred to the inner queue at the end of this interval. Based on the work in [150] and [23], the batch size distribution $x(k)$ can be computed as follows.

$$\begin{aligned} x(k) &= \tau(0)\delta(k) \\ &\quad + \sum_{m=1}^{\infty} \tau(m) \sum_{i=0}^{m-1} \left(f^{(k)}(i) - f^{(k+1)}(i) \right), k = 0, 1, ... \end{aligned} \tag{5.1}$$

The equation allows calculating the number of arrival events in an arbitrarily distributed time interval. The special case, in which no arrivals are observed in an interval of length 0, is covered by the first term. The function δ is defined in Equation 5.2. For the remaining interval lengths, the law of total probability is used in the second term in order to calculate the conditional probability $x(k|m)$. It can be derived from the relationship shown in Equation 5.3.

$$\delta(k) = \begin{cases} 1 & k = 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.2}$$

$$
\begin{aligned}
x(k|m) &= \mathrm{P}\left(F^{(k)} < m \leq F^{(k+1)}\right) \\
&= \mathrm{P}\left(F^{(k)} < m\right) - \mathrm{P}\left(F^{(k+1)} < m\right) \\
&= \sum_{i=0}^{m-1} \left(f^{(k)}(i) - f^{(k+1)}(i)\right), m > 0
\end{aligned}
\tag{5.3}
$$

Since the first packet after a transfer initiates the next aggregation interval, the batch interarrival time s can be calculated as the sum of the recurrence time of one packet, i.e., r_a, and the duration of the aggregation interval τ:

$$
s(k) = r_a(k) * \tau(k) \tag{5.4}
$$

Since the first packet in a batch triggers the timeout, the waiting time of consecutive packets is reduced. In particular, the waiting time of the i-th packet in the peripheral depends on the arrivals of the $i-1$ packets before it. Hence, the distribution of its waiting time can be computed as follows:

$$
w_i(k) = \pi_0 \left[\tau(k) * \underbrace{a(-k) * \cdots * a(-k)}_{(i-1) \text{ times}} \right] \tag{5.5}
$$

Model of the Central Queue (CPU/software)

We model the inner queue as a $GI^{[X]}/GI/1-L$ queue, i.e., a system with batch arrivals and bounded delay. The waiting time of packets is limited to a maximum value of L, i.e., jobs which arrive and would have to wait longer than $L-1$ are rejected. Our analysis extends the work presented in [151] by introducing batch arrivals. A similar notation, as presented in the following, is used:

- $u_{n,b_i}(k)$: distribution of unfinished work in the system before the arrival of the i-th packet of the n-th batch.
- $B_{n,i}$: RV for the service time of the i-th packet of the n-th batch.

- p_b: average blocking probability per packet.
- $\pi_0(\cdot)$: sweep operator which sums the probability mass of negative unfinished work in the system and appends it to the state for an empty system.

$$\pi_0(x(k)) = \begin{cases} x(k) & k > 0 \\ \sum_{i=-\infty}^{0} x(i) & k = 0 \\ 0 & k < 0 \end{cases}$$

- $\sigma^m(\cdot)$: operator which truncates the upper part of a probability distribution function.

$$\sigma^m(x(k)) = \begin{cases} x(k) & k \leq m \\ 0 & k > m \end{cases}$$

- $\sigma_m(\cdot)$: operator which truncates the lower part of a probability distribution function.

$$\sigma_m(x(k)) = \begin{cases} 0 & k < m \\ x(k) & k \geq m \end{cases}$$

The development of the batch arrival process is illustrated in Figure 5.4. Observing the packets of the n-th batch arrival, the i-th packet of the burst is accepted if the current unfinished work in the system is less than $L - 1$. In case the packet is accepted, the unfinished work is increased by the amount of work $B_{n,i}$ that is required to process the packet. Otherwise, the packet as well as the remaining packets of the current batch are rejected.

The following recursive relationship can be used in order to compute the amount of unfinished work in the system:

$$u_{n,b_1}(k) = u_n(k) \tag{5.6}$$

$$u_{n,b_{i+1}}(k) = \sigma^{L-1}\left[u_{n,b_i}(k)\right] * b_{n,i}(k) + \sigma_L\left[u_{n,b_i}(k)\right] \tag{5.7}$$

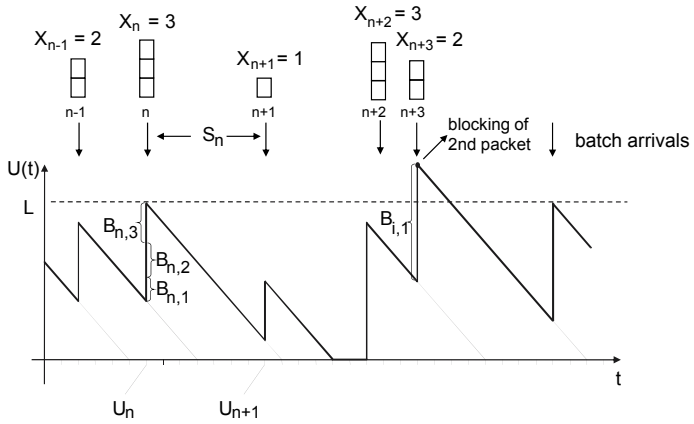


Figure 5.4: Exemplary system development for $GI^{[X]}/GI/1-L$ with bounded delay.

Hence, the remaining unfinished work in the system at the arrival of the next batch can be computed as:

$$u_{n+1}(k) = \pi_0 \left[\left(\sum_{i=1}^{\infty} x(i) \cdot u_{n,b_i}(k) \right) * s_n(-k) * o(k) \right] \tag{5.8}$$

In this calculation, the interrupt overhead o is added to the batch interarrival time s due to the fact that for each batch, the CPU has to devote time to handle this interrupt instead of processing packets.

An algorithm for calculating the workload prior to the i-th arrival can now be derived. It can be used for both stationary and non-stationary traffic conditions. Under stationary conditions, the index n and $(n+1)$ in these equations can be suppressed, cf. Equation 5.9. Furthermore, we assume that the packet service time is independent of a packet's position within the batch. Hence, the RV B_n refers to the service time for packets in the n-th batch. Similarly to Equation 5.9, the index n can also be suppressed under stationary conditions, resulting in RV B.

–

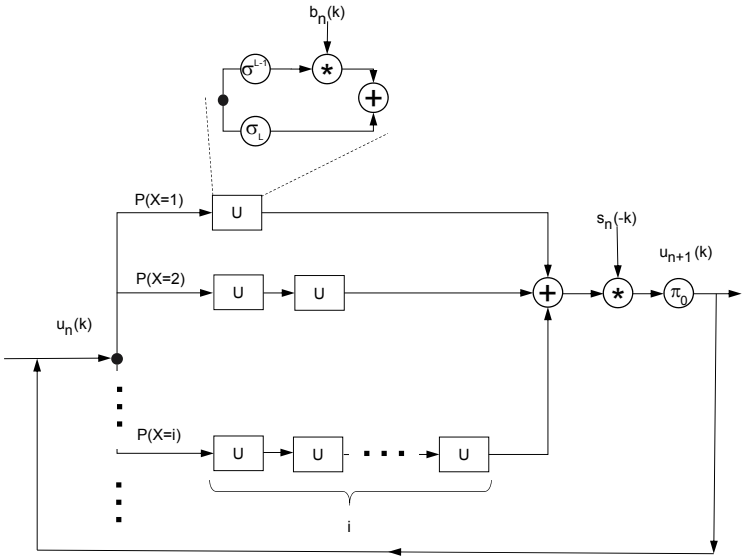


Figure 5.5: Computational diagram for $GI^{[X]}/GI/1-L$ with bounded delay.

$$\begin{aligned} u(k) &= \lim_{n\to\infty} u_n(k) \\ u_{b_i}(k) &= \lim_{n\to\infty} u_{n,b_i}(k) \end{aligned} \tag{5.9}$$

The computational diagram of the system is depicted in Figure 5.5. Depending on the batch size X, the unfinished work after a batch arrival can be determined by following the corresponding path through the diagram. Each of the X phases in such a path represents the relationship from Equation 5.7. Finally, the batch interarrival time s_n is taken into account and the sweep operator π_0 is used in order to ensure that a proper probability distribution is returned.

It is also possible to quantify the load ρ of the central queue. This is achieved by

calculating the ratio between the amount of work that arrives within a given time interval and the amount of work that is processed in this interval. In particular, we observe that the amount of work that arrives within a batch interarrival time depends on the batch size and the packet service time (cf. Equation 5.10). Note that both the batch size and the batch interarrival time are affected by the packet interarrival time (cf. Equations 5.1 and 5.4).

$$\rho = \frac{\mathrm{E}[X]\,\mathrm{E}[B]}{\mathrm{E}[S]} \tag{5.10}$$

Finally, the packet loss probability in statistical equilibrium can be computed as follows:

$$p_b = \sum_{i=1}^{\infty} \left(\frac{1}{i} x(i) \cdot \sum_{j=L}^{\infty} u_{b_i}(j) \right) \tag{5.11}$$

Depending on the batch size and the amount of unfinished work added by each packet within the batch, the blocking probability for the latter packets within the batch increases.

Combined Model

Using the two models described in Section 5.2.1 and Section 5.2.1, it is possible to determine the distribution of the total processing time. It is comprised of the waiting time in the peripheral queue, the waiting time in the central queue, and the service time in the latter. The waiting time in the central queue can be calculated from the unfinished work in the system and a packet's position in its batch. Hence, the following equation can be used to calculate the distribution of the total processing time of the i-th packet in a batch, d_i:

$$d_i(k) = w_i(k) * u(k) * \underbrace{b(k) * \cdots * b(k)}_{i \text{ times}} \tag{5.12}$$

Consequently, the distribution of the total processing time for all packets can be determined via conditional probabilities:

$$d(k) = \sum_{i=1}^{\infty} \mathrm{P}(X = i) \cdot d_i(k) = \sum_{i=1}^{\infty} x(i) \cdot d_i(k) \tag{5.13}$$

5.2.2 Applicability of the Proposed Model

In order to assess the goodness of fit of the introduced model, measurements are conducted in a testbed and compared with the model's predictions. In the following, the components of this testbed are described alongside the methodology for accurately measuring the CPU processing times as well as the results of the comparison.

Testbed Setup

The testbed setup is depicted in Figure 5.6. The *Serving Gateway* (SGW) application (cf. Section 4.4) runs on the DUT, a server[2] running a recent Linux version[3] equipped with a four-port NIC. Similar to Section 4.4, *GPRS Tunneling Protocol* (GTP) traffic is generated using a hardware traffic generator[4]. In order to evaluate per-packet processing times, wiretaps that duplicate all traffic are placed between the traffic generator and the receiving NIC of the DUT, as well as between the emitting NIC of the server and the traffic sink, which is again the traffic generator. The wiretaps are connected to a hardware capture card[5], which provides nanosecond precision timestamping of received traffic.

The processing time of the server is measured by calculating the time between a packet's arrival at the first wiretap and its arrival at the second wiretap. The packets at the two wiretaps are matched based on a unique 40 byte signature that the traffic generator adds to every packet. In order to verify that the traffic generator emits packets at equidistant times and at the correct rate, the interarrival times seen at the first capture card are inspected.

[2] Intel Xeon E5-2620 v2 CPU at 2.10 GHz, Intel I350 NICs, 32 GB of RAM
[3] 64 bit version of Debian 7.7 (wheezy, kernel version 3.2.0-4-amd64)
[4] Spirent TestCenter C1
[5] Endace DAG 7.5G2 Gig Ethernet

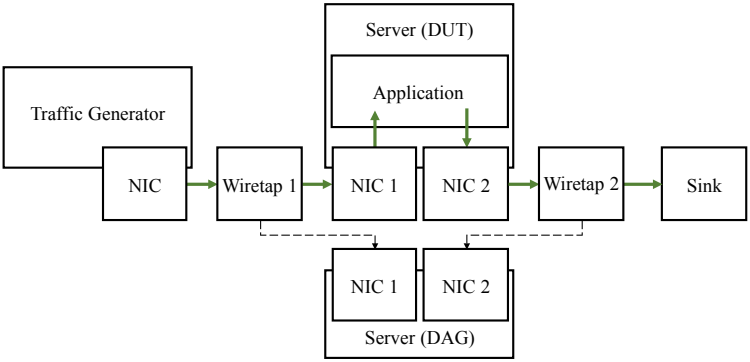


Figure 5.6: Testbed setup consisting of the DUT running the SGW application, a traffic generator, and a server with a DAG capture card.

The delay, how long the NIC buffers incoming packets, is adjusted using the `ethtool` command. In this context, <N> represents the number of the NIC and <T> reflects τ, i.e., the number of microseconds to wait after the first incoming packet:

```
# ethtool -C eth<N> rx-usecs <T>
```

Listing 5.1: Command to modify aggregation interval of NIC.

Estimating CPU Processing Time

In order to determine the processing time of the application code at a per-packet granularity, measurements using `tcpdump` are conducted. The time, when `tcpdump` captures a packet is on the kernel level, right after the interrupt is handled in incoming direction (from NIC 1), i.e., before the packet is copied by the kernel driver code to the egress NIC (cf. Figure 5.1).

One such exemplary measurement displaying the time between a packet's arrival at the receiving and the sending side is shown in Figure 5.7a. The two batches

of packets each show an increasing processing time, as the first packet is processed first by the CPU and the last (10th) packet is processed after all others in this batch. Therefore, the difference in the processing delay between consecutive packets equals $B_{n,i}$, the waiting and processing time in the application.

Given the application used in our experiments, a prototypical VNF implementation of a mobile network SGW, the measurements result in a distribution of processing times with a mean of 8.336 µs. This empirical distribution is used in the following after capping it at the 90% quantile (16 µs) to remove outliers, resulting in a mean of 7.25 µs. This distribution is shown as the red curve in Figure 5.7b and was picked as a representative from multiple measurements. The gray CDFs, as well as the corresponding means (dashed lines) show the CPU processing times of other measurements and highlight the variations between the different runs.

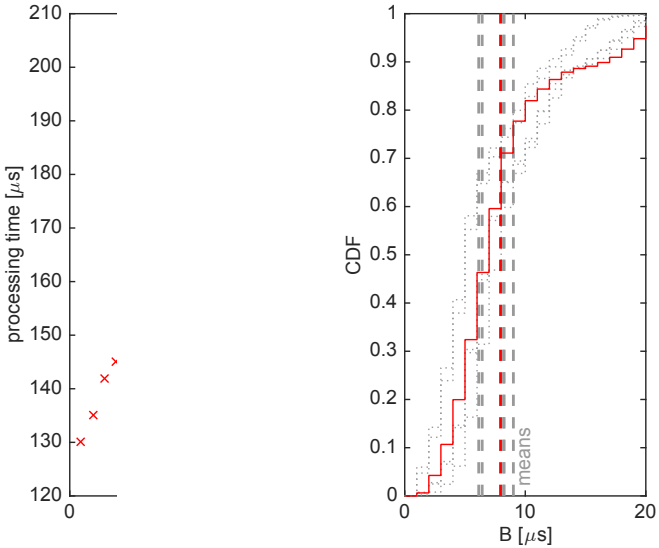
Comparison of Model Predictions and Measurements

In order to demonstrate the applicability of the proposed model, we compare its results with measurements. For that, we conduct five independent measurement runs for constant interarrival times between 5 and 12 µs. Each measurement run lasts one minute, and the aggregation interval is set to $\tau = 200\,\mu s$.

The size of the central queue, denoted by L, corresponds to 5,200 µs of unfinished work. Based on the measured mean service time at the CPU $\mathrm{E}[B] = 8.336\,\mu s$, the inner queue size of $L_{byte} = 131,071$ byte as defined by the operating system, and the packet payload of 210 byte, L computes as follows:

$$L = \mathrm{E}[B] \cdot \frac{L_{byte}}{210} = 5200\,\mu s$$

Based on the measurements, we compute the mean processing times and the corresponding confidence intervals on a 95% confidence level, as well as the packet loss probability. Additionally, we compute the mean processing times and the packet loss probability using the analytical model. As service time distribution, we take the empirically measured service time from Figure 5.7. As interrupt overhead, we use $o = 4\,\mu s$, which is based on the values reported in [146].



(a) Function service time of the CPU for packets captured at kernel level.

(b) Distributions of measured function service times and mean values for different runs.

Figure 5.7: Differences in processing times of single packets seen in the networking stack (kernel level) allow deriving the CPU service time per packet.

Figure 5.8a shows a comparison of the measurements and values obtained from the analytical model. Error bars denote the 95% confidence intervals from five measurement runs. The bars indicate the mean processing time per packet according to the model.

For packet interarrival times below 8 µs, the error bars overlap with the mean values from the model, indicating the applicability of the model. For larger interarrival times, only a slight difference is observed between the model's prediction and the measurements. A possible explanation for this phenomenon is the high degree of variance regarding the empirical function service times shown in Figure 5.7b.

In an analogous fashion, Figure 5.8b shows the applicability of the model with respect to to the packet loss rate. Except in the case of $\mathrm{E}[A] = 7\,\mu s$, the error bars overlap with the values from the model. Based on the huge error bar seen in the previous figure, this interarrival time roughly corresponds to the maximum rate that the server can handle and the first occurrence of packet loss can be observed. For larger interarrival times, the model and measurements both indicate zero packet loss.

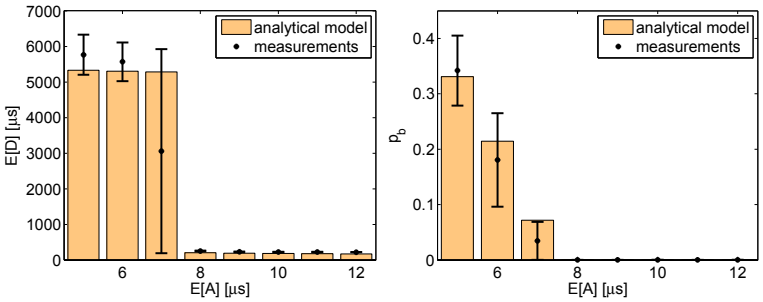
The occurrence of packet loss for an average interarrival time below 8 µs is consistent with the definition of the system load ρ in Equation 5.10 and the mean service time at the CPU of 7.25 µs that is obtained after removing outliers. Since the average batch size $\mathrm{E}[X]$ can be determined by means of τ and $\mathrm{E}[A]$, and the recurrence time in the context of very low interarrival times is negligible, the system load can be approximated as follows:

$$\rho = \frac{\mathrm{E}\left[X\right] \cdot 7.25}{\mathrm{E}\left[S\right]} = \frac{\tau \cdot 7.25}{\mathrm{E}[A](\mathrm{E}[R_a] + \tau)} \overset{\mathrm{E}[R_a] \ll \tau}{\approx} \frac{7.25}{\mathrm{E}[A]} \tag{5.14}$$

Hence, in the context of mean interarrival times below 7.25 µs, the system load is larger than 1, and packet loss occurs. Furthermore, the actual load is slightly higher due to the fact that the same CPU core also handles the interrupts that are caused by outgoing packets. These amount to roughly 20,000 IRQs per second in our scenarios.

Evaluation

In this section, we investigate the behavior of the packet processing server based on the introduced model. In this context, we focus on the total processing time D and the packet loss probability p_b. The influence of the mean packet interarrival time $\mathrm{E}[A]$ and the length of the aggregation interval τ are studied. At first, coarse-grained analyses of the resulting mean processing times and packet loss ratios for different interarrival time distributions and aggregation interval lengths are presented. Afterwards, we investigate the impact of these two influence factors for a particular packet interarrival time distribution on the distribution of processing times.



(a) Average processing times. *(b) Packet loss.*

Figure 5.8: Comparison of analytical model and measurements for $\tau = 200\,\mu s$.

Impact of the Arrival Process The sensitivity of the modeled system to different distributions of the packet interarrival time A is studied based on four different distributions, namely deterministic (det), Poisson (pois), geometric (geo), and negative binomial (nbin). For det, pois, and geo, the distributions are characterized solely by $\mathrm{E}[A]$, resulting in a constant coefficient of variation of 1. The parameters p and r of nbin are adjusted so that $\sigma = \mu$ holds true. This results in a constant coefficient of variation equal to $\sqrt{3}$.

Impact on Mean Processing Times Figure 5.9 presents the mean packet processing time D that results from different combinations of the distribution of packet interarrival time and its mean. While the x-axis displays the mean packet interarrival time, the y-axis indicates the average packet processing time. Additionally, line colors represent different values of the aggregation interval length τ and line styles correspond to the four distribution types.

In most cases, the curve shape is composed of three phases. First, small packet interarrival times result in high processing times that stem from long waiting times in the central queue. As soon as the average interarrival time exceeds τ, in most cases, each batch is comprised of only one packet. As this packet initiated a new aggregation interval, it has to wait until the timer ends after τ. Due to the

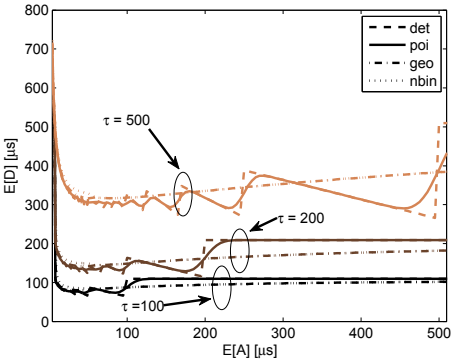
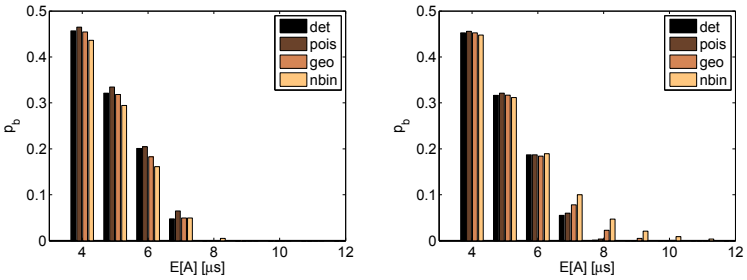


Figure 5.9: Effects of different values of $\mathrm{E}[A]$ *and different aggregation intervals* τ *on the mean processing time* $\mathrm{E}[D]$.

low rate, the unfinished work at the central queue (the CPU) is low or oftentimes zero, resulting in immediate processing of the packet. Since in this case the processing time in the central queue is relatively low compared to the waiting time in the peripheral queue, the total processing time is mostly influenced by τ. For interarrival times that follow a deterministic or a Poisson distribution, most aggregation intervals contain exactly one packet, resulting in processing times that are slightly higher than τ. In contrast, the negative binomial and geometric distributions lead to bursts of packet arrivals that result in lower mean processing times. After the first packet of a batch starts the aggregation interval, consecutive packets still arrive within the interval and thus, have a lower waiting time in the peripheral queue.

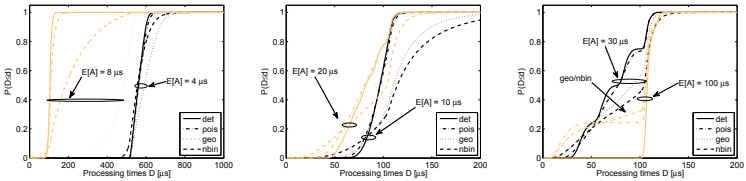
For interarrival times that are lower than τ, but do not lead to queuing at the central queue, expected batch sizes for all distributions are larger than one. Therefore, the mean waiting time in the peripheral queue decreases and thus, the mean overall processing time $\mathrm{E}[D]$ also decreases.

Although the figure might suggest that decreasing τ, i.e., reducing the interrupt moderation, leads to lower processing times, this is only true until reaching a



(a) $\tau = 200\ \mu s$. *(b)* $\tau = 500\ \mu s$.

Figure 5.10: Packet loss depending on $\mathrm{E}[A]$ *for different* τ.



(a) Low interarrival times. *(b) Low interarrival times.* *(c) High interarrival times.*

Figure 5.11: Processing time distributions for varying packet interarrival times and different interarrival distributions in case of $\tau = 100\ \mu s$.

break-even point. Then, the overhead per packet caused by interrupt handling and context switches accounts for the majority of CPU time.

Impact on Packet Loss As described previously, the processing time increases with the number of packets per second, as packets experience a waiting time at the central queue. As this queue is limited by L (cf. Section 5.2), packet loss occurs once this limit is exceeded as described in Equation 5.11. In the following, the impact of the mean and distribution of interarrival times on the packet loss probability is evaluated.

Figure 5.10 depicts the packet loss probability for the four different distributions depending on different mean processing times and lengths of the aggregation interval. It can be observed that the Poisson distributed interarrival times result in the highest packet loss ratio when the system operates at a high load. The assumption behind applying interrupt moderation techniques is a certain burstiness of traffic. Hence, the packet loss ratio is up to 8% lower for nbin than for geometrically distributed arrivals in the case of $\tau = 200\,\mu s$ depicted in Figure 5.10a. Due to the higher degree of burstiness of the former, longer idle times after τ finished occur and thus fewer interrupts are triggered.

As described in Section 5.2.2, the CPU load exceeds 1 when $\mathrm{E}[A]$ falls below 7.25 µs (cf. Equation 5.14). This fits with the observed packet loss at $\mathrm{E}[A] \leq 7\,\mu s$ for all distributions. In case of nbin, packet loss occurs even at $\mathrm{E}[A] = 8\,\mu s$ due to the higher burstiness of the traffic.

However, it is questionable whether this system can be operated in overload conditions with a packet loss ratio of more than 5 %, which occurs for interarrival times of 7 µs and less, corresponding to more than 142,857 packets per second. Thus, the lower rate with $\mathrm{E}[A] = 8\,\mu s$, when no packet loss occurs for all distributions *except* nbin, is more interesting. The reason for this behavior is, again, its burstiness and higher variation, resulting in short overload situations that lead to packet loss. In contrast, the other distributions result in more equally spaced arrivals.

For the largest aggregation interval of 500 µs, this effect is visible even for higher values of $\mathrm{E}[A]$. Caused by the higher expected number of packets per batch ($\tau / \mathrm{E}[A]$), the probability that packets are dropped in the central queue is increased, resulting in a higher packet loss ratio.

Processing Time Distributions for Varying Interarrival Times In addition to studying the influence of the arrival process on the mean processing time, we also investigate its effect on the distribution of the processing time. Figure 5.11 shows the CDFs of the processing time D given an aggregation interval of $\tau = 100\,\mu s$ combined with different arrival processes and values for the mean interarrival time $\mathrm{E}[A]$.

For the lowest mean interarrival time of $4\,\mu s$ shown in Figure 5.11a, i.e., the scenario with the highest system load, the highest processing times are observed. Furthermore, the distribution of processing times in this scenario has a low variance and similar values independent of the arrival process. This can be explained by the combination of the very high load and the fact that the system drops packets that encounter a full queue. In contrast, the distribution of the processing time in the context of $\mathrm{E}[A] = 8\,\mu s$ differs significantly across different distributions of the interarrival time. On the one hand, the relatively stable det and pois distributions result in a narrow range of processing times which is significantly lower than for $\mathrm{E}[A] = 4\,\mu s$. On the other hand, the higher degree of variation of the geo and nbin distributions result in a larger variety of batch sizes which, in turn, yield wide intervals of different processing times.

A further decrease of the processing times is observed for the medium interarrival times seen in Figure 5.11b. In these scenarios, the distributions resulting from the nbin and geo distributions are closer to each other and begin to converge. This phenomenon can be explained by the evolution of the two arrival processes. For higher values of $\mathrm{E}[A]$, the coefficient of variation of geo approaches 1, i.e., that of the nbin distribution used in this work. Simultaneously, the r parameter of the nbin distribution approaches 1. Since the geometric distribution is a special case of the negative binomial distribution with $r = 1$, the aforementioned convergence can be explained.

Finally, Figure 5.11c displays the processing time distributions in case of $\mathrm{E}[A] = 30\,\mu s$ and $\mathrm{E}[A] = 100\,\mu s$, respectively. When the mean interarrival time equals the aggregation interval τ, only size 1 batches are processed in case of a deterministic arrival process. In combination with the fact that arrivals initiate the aggregation intervals, the processing time is dominated by the waiting time in the peripheral queue. For $\mathrm{E}[A] = 30\,\mu s$, batches consist of four packets, hence the distribution consists of four segments with similar shapes corresponding to a packet's position within a batch. The processing time distributions that result from a geometric and a negative binomial distribution converge further when $\mathrm{E}[A]$ is increased and overlap in case of $\mathrm{E}[A] = 100\,\mu s$. Processing times resulting from interarrival times that follow a Poisson distribution are lower and

closer to those of det rather than geo and nbin which have a significantly higher degree of variation.

5.3 Lessons Learned

The introduced model allows to estimate the impact of performance-relevant parameters of VNFs. By reducing the complexity of the system by taking abstracting, the model is able to compute system's treatment of every processed packet. The resulting processing time distribution and packet loss probabilities allow to estimate the effects of particular implementation aspects on the characteristics of the emitted traffic.

The validation setup, where measurements of the implementation were conducted with microsecond-precision, demonstrated the applicability of the model to the real VNF. Based on this, adjustments of input parameters like packet interarrival distributions and the processing time within the VNF together with different interrupt frequencies allows to estimate the effects with regard to overhead and short processing times.

It was found out that there is a "sweet spot" in such system, right between a high frequency of interrupts, which lead to low delays but lots of wasted CPU time and, on the other side, larger delays caused by more infrequent interrupts and larger batch sizes in consequence. The model allows to evaluate such "what-if" scenarios, without running the complete benchmark and even before altering the implementation.

Further parameter adjustment allows to predict the system behavior in case of acceleration techniques like Intel's DPDK or Cisco's *Vector Packet Processing* (VPP) being applied. Applying the model allows to compare heterogeneous network function implementations and selecting the appropriate technique for a specific use case.

6 Conclusion

Increasing traffic volumes in data center and wide area networks increase the expenditures of operators. Traditionally, new hardware was acquired once utilization exceeded some boundary, like 50% link utilization. In order to avoid, or at least delay, such investments in more powerful hardware, a few companies started to apply novel mechanisms on their own, instead of relying on device vendors. Network softwarization implements logic in form of software running on servers instead of as firmware. This allows operators to develop and deploy custom solutions tailored to their needs within short time. This allows setups that make far better resource utilization without impairing quality of service.

Software companies like Google and Facebook are leading this trend of softwarizing network devices and appliances. They developed new business models that are consumed by billions of users and generate extreme amounts of data, which these companies process. Software-based networks help them to not only reduce expenditures, but also to be more agile with making changes to the infrastructure to serve the applications running on top.

Recently, especially through the advent of *Network Functions Virtualisation* (NFV), more and more operators of telecommunication networks see the benefits of software-based networking as well. Not only that telcos want to reduce their dependency on proprietary software, they also want to exploit new fields of income by offering new services to their customer. These operators also lead the corresponding standardization activities within the *European Telecommunications Standards Institute* (ETSI), as well as reference implementations within the collaborative *OpenNFV* project.

Prior to a wide-spread deployment of software-based networks using techniques such as NFV and *Software Defined Networking* (SDN), several issues have to

be resolved. Many of these issues are related to performance and novel management methods that allow a much higher pace of risk-free changes to the network infrastructure.

This monograph contributes to solving both types of issues that prevent a widespread deployment of software-based networks. Chapter 2 investigated, how the physical infrastructure can be optimized for higher resource utilization and increased energy efficiency. With SDN and NFV on the horizon, operators hesitate to invest in the underlying physical network topology with all its fiber strands. Further, capacity planning in advance and path planning prior to network deployment become less critical and too inflexible, given the steadily changing characteristics of modern networks. Therefore, it was investigated, how the planning process of wide area networks can be tailored more towards energy efficiency, without actually changing the software responsible for this planning process. As such tooling is very complex and expensive, the goal was to influence the resulting outcome to be more energy efficient, by only modifying the input parameters to the planning software. Based on multiple subgraphs of the original fiber topology, it was evaluated, how the active length of all fiber links could be reduced. By avoiding planning runs for subgraph topologies that lack criteria like two-connectedness, the overall duration of the planning process can be reduced.

The effects of different deliberately generated subgraphs were evaluated using multiple realistic network topologies and an existing and publicly available network planning software. The effects have shown that savings between 9% and 54% are possible when using the link length as cost metric.

Novel approaches to network management were described in Chapter 3. By applying successful methods from software engineering, like continuous delivery and test-driven development, changes to network infrastructures can be applied more rapidly and with increased confidence. Automated tests aid as a safety net that prevent unexpected side-effects to occur, which is a very likely issue in complex network setups. The utilization of such methods now becomes possible, as networks become less dependent on physical instances and instead make use of virtualization, allowing to have realistic duplicates of the production infrastructure available at very low costs and on the push of a button.

In the case of defects, the mind-changing practices of continuous delivery help to solve problems faster than with traditional approaches focusing on increasing *Mean Time Between Failure* (MTBF) [152]. Instead of aggregating changes and applying many of them at once, novel approaches bring small pieces of change very quickly into production. This eases the root-cause analysis to figure out the source of the failure and helps to deploy a fix way faster, i.e., reducing the *Mean Time To Repair* (MTTR) instead. The first part of Chapter 3 describes guide lines, how the deployment process for an SDN controller as well as a *Virtualised Network Function* (VNF) should look like.

Furthermore, Chapter 3 describes two techniques that exploit the flexibility and programmability of software-based networks. For conducting elephant detection, a particular network monitoring problem, it was described, how counters in hardware devices are created dynamically and refined iteratively. Compared to traditional approaches, this provides a very light-weight solution, which can be further tuned towards the particular needs of the network administrator or application consuming the output.

Micro segmentation constraints network devices into their own virtual network, restricting access to only explicitly permitted resources. The described implementation for a "Bring Your Own Device" use case employs SDN and NFV to dynamically set up virtual networks. In contrast to previous approaches, this neither relies an on agent installed on the user-owned device, nor does it group all devices into a common access group. Instead two factor authentication is used to verify authenticity of requests that result in white listing of services, e.g., to a email, a business application, or the Internet. Changes in the IP addresses under which rapidly changing cloud services are available are communicated to the network infrastructure through the very same service discovery mechanisms, as they are used within the application themselves.

Performance aspects of software-based networks are investigated in Chapter 4, which further contributes tooling and guide lines for benchmarking of such systems. Conducted evaluations include a comparison of hardware and software implementations of a firewall, where it could be seen that the by far more expensive hardware offers more predictable processing times in the order of microseconds.

Nevertheless, the software variant, which induces sub-millisecond jitter on the emitted traffic might still be applicable to a wide range of use cases.

Similar, benchmarking studies of SDN controllers using a custom-built benchmarking software help to evaluate the performance of the network control plane prior to a deployment. Given the criticalness of controller for the overall network stability, such capacity tests should be integrated into the previously described deployment pipeline.

Finally, Chapter 5 introduces an analytical model for the packet processing in a server running the *Linux* operating system. The queuing system implements a clocked transfer, which initiates the transfer of work - packets from the queue within the *Network Interface Card* (NIC) - into operating system buffers, from where the VNF application can read them. The transfer is triggered by an interrupt, itself coming with an overhead. The presented model allows to estimate the sweet spot between frequent interrupts resulting in low delays and less frequent ones resulting in lower CPU overhead. Triggering too many interrupts can finally lead to starvation of CPU resources and thus packet loss.

The introduced model was validated using a prototypical VNF in a test bed running hardware traffic generation and highly precise time stamping. After empirically determining the service time distribution of the VNF, different settings for the interrupt delay, as well different packet arrival rates were evaluated. It was observed that the model correctly estimates both, the processing time of the server running the VNF as well as the packet loss. This model could be further extended to analyze other implementation choices of a VNF, i.e., packet processing frameworks like *netmap* or *DPDK*.

The combination of mechanisms described in this monograph now finally brings long-desired features, like better scalability, increased flexibility, as well as shorter development cycles. Backward compatibility can be maintained by keeping communication with end hosts, e.g., hosts in the Internet, unchanged. The network itself, however, can still be optimized under the surface. In contrast, applications within the own network or those of cooperating peers can interact with the network to exploit the full benefits, including cost effectiveness, better user satisfaction, and higher agility.

Acronyms

2FA Two-factor Authentication. 89, 90

AP Access Point. 88, 89

API Application Programming Interface. 3, 9, 11, 55, 98, 101

ASASM Adaptive Security Appliance Service Module. 114, 116

ASAv Adaptive Security Virtual Appliance. 115, 116

ASIC Application-specific Integrated Circuit. 6, 97, 100, 116

BDD Behavior-Driven Development. 69

BGP Border Gateway Protocol. 1

BYOD Bring Your Own Device. 56, 86, 87

CAM Content-addressable Memory. 93

CD Continuous Delivery. 50, 52, 59, 61–63, 66, 68, 69

CDPI Control Data Plane Interface. 4, 48

CE Carrier Ethernet. 18

CLI Command Line Interface. 59

COTS Commercial off-the-shelf. 100, 114

CPE Customer Premises Equipment. 5, 97, 133

CPU Central Processing Unit. 137, 138

DOCSIS Data Over Cable Service Interface Specification. 15

DPDK Data Plane Development Kit. 6, 9, 98, 101

DPI Deep Packet Inspection. 5, 59

DPRL Data Path Requirement Language. 55

DSL Digital Subscriber Line. 15

DUDL Dynamic Upper Dynamic Lower. 26

DUFL Dynamic Upper Fixed Lower. 26

DUT Device Under Test. 100, 101, 140, 151

ECMP Equal-Cost Multi-Path. 47

EPC Evolved Packet Core. 122

ESS Extended Service Set. 89, 92

ESSID Extended Service Set Identification. 89

ETSI European Telecommunications Standards Institute. 5, 6, 163

FDM Frequency Division Multiplexing. 13

FEC Forward Error Correction. 17

FIB Forwarding Information Base. 3, 49

FPGA Field-Programmable Gate Array. 101, 115, 118, 141

FUFL Fixed Upper Fixed Lower. 26

GPON Gigabit Passive Optical Networks. 15

GPU Graphics Processing Unit. 101, 141

GRE Generic Routing Encapsulation. 47

GTP GPRS Tunneling Protocol. 122, 151

IAT Inter-Arrival Time. 107

IP Internet Protocol. 1, 18, 19, 26

IPFRR IP Fast Reroute. 23

ISP Internet Service Provider. 14

LAN Local Area Network. 13, 18

LDAP Lightweight Directory Access Protocol. 89

LER Label Edge Router. 18

LISP Locator-Identifier Split. 1

LLDP Link-Layer Discovery Protocol. 108, 109

LSR Label Switch Router. 18

MDM Mobile Device Management. 86

MILP Mixed Integer Linear Program. 20, 26, 27, 36, 37

MME Mobility Management Entity. 122

MPLS Multi-Protocol Label Switching. 18, 19

MST Minimum Spanning Tree. 32

MTBF Mean Time Between Failure. 60, 165

MTTR Mean Time To Repair. 60, 165

MTU Maximum Transmission Unit. 47

NAPI New API. 124, 138

NCM Network Configuration Management. 60

NF Network Function. 59

NFV Network Functions Virtualisation. 5–8, 59, 66, 97, 98, 114, 137, 163

NIC Network Interface Card. 11, 98, 136–139, 143, 144, 151, 152, 166

NPU Network Processing Unit. 101, 141

OADM Optical Add-Drop Multiplexer. 17

OCh Optical Channel. 17, 18

OMS Optical Multiplex Section. 17, 18

OPEX Operating Expense. 11

OPNFV Open Platform for NFV. 6

OTN Optical Transport Network. 17

OTS Optical Transmission Section. 17

PCEP Path Computation Element Protocol. 1

PPS Packets Per Second. 107

QA Quality Assurance. 52, 65, 67, 68

RAID Redundant Array of Independent Disks. 23

ROADM Reconfigurable Optical Add-Drop Multiplexer. 23

RTT Round Trip Time. 107

SDH Synchronous Digital Hierarchy. 18

SDM Space-Division Multiplexing. 13

SDN Software Defined Networking. 3–9, 11, 45, 46, 55, 58, 59, 61–64, 67, 68, 70, 74, 86, 87, 95, 97, 102, 110, 163, 165, 166

SGW Serving Gateway. 98, 133, 151, 153

SLA Service Level Agreement. 20, 22

SONET Synchronous Optical Networking. 18

SRG Shared Risk Group. 24

TCAM Ternary Content-addressable Memory. 94, 116

TDD Test-Driven Development. 51, 55, 69

TDM Time-Division Multiplexing. 13, 18

TEID Tunnel Endpoint Identifier. 122

TOTP Time-Based One-Time Password Algorithm. 89

UI User Interface. 54

VA Virtual Application. 59

VCS Version Control System. 50, 52

VNF Virtualised Network Function. 5, 6, 8, 9, 11, 62, 66, 68, 95, 97, 98, 100, 120, 133–135, 153, 161, 165, 166

VNF-FG VNF Forwarding Graph. 55

VPP Vector Packet Processing. 161

VXLAN Virtual Extensible LAN. 47

WAN Wide Area Network. 13, 16, 25, 45

WDM Wavelength-Division Multiplexing. 13, 26

xDPd eXtensible OpenFlow data path daemon. 123, 124, 129

Bibliography and References

Bibliography of the Author

Journal Papers

[1] S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, D. Hock, M. Jarschel, and M. Hoffmann, "Heuristic approaches to the controller placement problem in large scale SDN networks", *IEEE transactions on network and service management - special issue on efficient management of SDN and NFV-based systems*, vol. 12, no. 1, pp. 4–17, Feb. 2015.

[2] C. Lorenz, D. Hock, J. Scherer, R. Durner, W. Kellerer, S. Gebert, N. Gray, T. Zinner, and P. Tran-Gia, "An SDN/NFV-enabled enterprise network architecture offering fine-grained security policy enforcement", *IEEE communications magazine*, vol. 99, Jan. 2017.

Conference Papers

[3] S. Gebert, R. Pries, D. Schlosser, and K. Heck, "Internet access traffic measurement and analysis", in *4th international conference on traffic monitoring and analysis (TMA)*, Vienna, Austria, Mar. 2012.

[4] S. Gebert, D. Hock, M. Hartmann, J. Spoerhase, T. Zinner, and P. Tran-Gia, "Including energy efficiency aspects in multi-layer optical network design", in *5th international conference on communications and electronics (ICCE)*, Da Nang, Vietnam, Jul. 2014.

[5] S. Gebert, C. Schwartz, T. Zinner, and P. Tran-Gia, "Continuously Delivering Your Network", in *IEEE/IFIP international symposium on integrated network management (IM)*, Ottawa, Canada, May 2015.

[6] S. Gebert, M. Jarschel, S. Herrnleben, T. Zinner, and P. Tran-Gia, "TableVisor: An emulation layer for multi-table OpenFlow switches", in *4th european workshop on software defined networks (EWSDN)*, Bilbao, Spain, Sep. 2015.

[7] S. Gebert, D. Hock, T. Zinner, P. Tran-Gia, M. Hoffmann, M. Jarschel, E.-D. Schmidt, R.-P. Braun, C. Banse, and A. Koepsel, "Demonstrating the optimal placement of virtualized cellular network functions in case of large crowd events", in *ACM conference on SIGCOMM*, Aug. 2014.

[8] S. Gebert, T. Zinner, S. Lange, C. Schwartz, and P. Tran-Gia, "Performance modeling of softwarized network functions using discrete-time analysis", in *28th international teletraffic congress (ITC)*, Würzburg, Germany, Sep. 2016.

[9] S. Gebert, T. Zinner, N. Gray, R. Durner, C. Lorenz, and S. Lange, "Demonstrating a personalized secure-by-default bring your own device solution based on software defined networking", in *28th international teletraffic congress (ITC)*, Würzburg, Germany, Sep. 2016.

[10] S. Gebert, S. Geissler, T. Zinner, A. Nguyen-Ngoc, S. Lange, and P. Tran-Gia, "ZOOM: Lightweight SDN-based elephant detection", in *1st international workshop on programmability for cloud networks and applications (PROCON)*, Würzburg, Germany, Sep. 2016.

[11] S. Gebert, A. Müssig, S. Lange, M. Krieger, T. Zinner, N. Gray, and P. Tran-Gia, "Processing time comparison of a hardware-based firewall and its virtualized counterpart", in *8th EAI international conference on mobile networks and management (MONAMI)*, Abu Dhabi, United Arab Emirates, Oct. 2016.

[12] N. Gray, T. Zinner, S. Gebert, and P. Tran-Gia, “Simulation framework for distributed SDN-controller architectures in OMNeT++”, in *8th EAI international conference on mobile networks and management (MONAMI)*, Abu Dhabi, United Arab Emirates, Oct. 2016.

[13] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia, “Pareto-optimal resilient controller placement in SDN-based core networks”, in *25th international teletraffic congress (ITC)*, Shanghai, China, Sep. 2013.

[14] D. Hock, S. Gebert, M. Hartmann, T. Zinner, and P. Tran-Gia, “POCO: A framework for the pareto-optimal resilient controller placement in sdn-based core networks”, in *IEEE network operations and management symposium (NOMS)*, Krakow, Poland, May 2014.

[15] D. Hock, M. Hartmann, S. Gebert, T. Zinner, and P. Tran-Gia, “POCO-PLC: Enabling dynamic pareto-optimal resilient controller placement in SDN networks”, in *Computer communications workshops (INFOCOM WKSHPS)*, Toronto, Canada, Apr. 2014.

[16] M. Jarschel, C. Metter, T. Zinner, S. Gebert, and P. Tran-Gia, “OFCProbe: A platform-independent tool for OpenFlow controller analysis”, in *5th IEEE international conference on communications and electronics (ICCE)*, Da Nang, Vietnam, Aug. 2014.

[17] S. Lange, A. Nguyen-Ngoc, S. Gebert, T. Zinner, M. Jarschel, A. Koepsel, M. Sune, D. Raumer, S. Gallenmüller, G. Carle, and P. Tran-Gia, “Performance benchmarking of a software-based LTE SGW”, in *2nd international workshop on management of SDN and NFV systems (ManSDN/NFV)*, Barcelona, Spain, Nov. 2015.

[18] S. Lange, S. Gebert, J. Spoerhase, P. Rygielski, T. Zinner, S. Kounev, and P. Tran-Gia, “Specialized heuristics for the controller placement problem in large scale SDN networks”, in *27th international teletraffic congress (ITC)*, Gent, Belgium, Sep. 2015.

[19] C. Metter, S. Gebert, S. Lange, T. Zinner, P. Tran-Gia, and M. Jarschel, "Investigating the impact of network topology on the processing times of SDN controllers", in *7th IFIP/IEEE international workshop on management of the future internet*, Ottawa, Canada, May 2015.

[20] A. Nguyen-Ngoc, S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, and M. Jarschel, "Investigating isolation between virtual networks in case of congestion for a pronto 3290 switch", in *Workshop on software-defined networking and network function virtualization for flexible network management (SDNFlex)*, Cottbus, Germany, Mar. 2015.

[21] ——, "Performance evaluation mechanisms for flowmod message processing in OpenFlow switches", in *IEEE sixth international conference on communications and electronics (ICCE)*, Ha Long City, Vietnam, Jul. 2016.

Technical Reports

[22] S. Gebert, C. Schwartz, T. Zinner, and P. Tran-Gia, "Agile management of software based networks", University of Wuerzburg, Tech. Rep. 493, Jan. 2015.

[23] S. Gebert, T. Zinner, S. Lange, C. Schwartz, and P. Tran-Gia, "Discrete-time analysis: Deriving the distribution of the number of events in an arbitrarily distributed interval", University of Wuerzburg, Tech. Rep. 498, Jun. 2016.

[24] F. Metzger, S. Gebert, K. Salzlechner, A. Ratetseder, P. Romirer, and K. Tutschku, "Signaling load and tunnel management in a 3G core network", University of Wuerzburg, Tech. Rep. 484, Sep. 2012.

General References

[25] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat,

"B4: Experience with a globally-deployed software defined wan", in *ACM conference on SIGCOMM*, Hong Kong, China, 2013.

[26] M. Jarschel, A. Basta, W. Kellerer, and M. Hoffmann, "SDN and NFV in the mobile core: Approaches and challenges", *It-information technology*, vol. 57, no. 5, pp. 305–313, 2015.

[27] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer, "Interfaces, Attributes, and Use Cases: A Compass for SDN", *IEEE communications magazine*, vol. 52, Jun. 2014.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks", *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.

[29] Open Networking Fundation, "Software-Defined Networking: The New Norm for Networks", *ONF white paper*, 2012.

[30] *ONOS project - Open Network Operating System*. [Online]. Available: `http://onosproject.org/`.

[31] *OpenDaylight project*. [Online]. Available: `http://www.opendaylight.org`.

[32] "Network Functions Virtualisation - Introductory White Paper", 2012. [Online]. Available: `https://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

[33] S. Newman, *Building microservices*. O'Reilly Media, 2015, ISBN: 9781491950357.

[34] *Open Platform for NFV (opnfv)*. [Online]. Available: `https://www.opnfv.org`.

[35] Intel, *Intel data plane development kit (DPDK)*. [Online]. Available: `http://dpdk.org`.

[36] B. Dormon, *How the internet works: Submarine fiber, brains in jars, and coaxial cables*, Ars Technica, May 2016. [Online]. Available: http://arstechnica.com/information-technology/2016/05/how-the-internet-works-submarine-cables-data-centres-last-mile/.

[37] "ITU-T recommendation G.872: Architecture of optical transport networks", International Telecommunication Union, Tech. Rep., 2001.

[38] "ITU-T recommendation G.709: Interfaces for the optical transport network", International Telecommunication Union, Tech. Rep., 2001.

[39] "ANSI IEEE 802.3 standard", IEEE, Tech. Rep., May 1998.

[40] *IST Project NOBEL*, http://cordis.europa.eu/projects/rcn/71203_en.html.

[41] S. Orlowski and R. Wessäly, *An integer programming model for multi-layer network design*, 2004.

[42] M. Duelli, X. Qin, and M. Menth, "Greedy design of resilient multi-layer networks", in *6th EURO-NF conference on next generation internet (NGI)*, Jun. 2010.

[43] M. Duelli, E. Weber, J. Ott, and X. Qin, "MuLaNEO: Planung und optimierung von mehrdienstnetzen", *PIK-Praxis der Informationsverarbeitung und Kommunikation*, vol. 34, no. 3, pp. 138–139, 2011.

[44] M. Duelli, *MuLaNEO: design and provisioning of resilient multi-layer networks*. [Online]. Available: https://sourceforge.net/projects/mulaneo/.

[45] M. Shand and S. Bryant, "IP fast reroute framework", IETF Secretariat, Internet-Draft draft-ietf-rtgwg-ipfrr-framework-13, Oct. 2009. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtgwg-ipfrr-framework-13.txt.

[46] J.-P. Vasseur, M. Pickavet, and P. Demeester, *Network recovery: Protection and restoration of optical, SONET-SDH, IP, and MPLS*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, ISBN: 012715051X.

[47] B. Rajagopalan, D. Pendarakis, D. Saha, R. S. Ramamoorthy, and K. Bala, "IP over optical networks: Architectural aspects", *IEEE communications magazine*, vol. 38, no. 9, pp. 94–102, Sep. 2000.

[48] K. Hinton, J. Baliga, M. Z. Feng, R. Ayre, and R. Tucker, "Power consumption and energy efficiency in the internet", *IEEE network*, vol. 25, no. 2, pp. 6–12, 2011.

[49] C. Lange, D. Kosiankowski, R. Weidmann, and A. Gladisch, "Energy consumption of telecommunication networks and related improvement options", *IEEE journal of selected topics in quantum electronics*, vol. 17, no. 2, pp. 285–295, 2011.

[50] A. P. Bianzino, C. Chaudet, D. Rossi, and J.-L. Rougier, "A Survey of Green Networking Research", Institut TELECOM, TELECOM ParisTech, Paris, France, Tech. Rep., 2010.

[51] L. Chiaraviglio and A. Cianfrani, "On the effectiveness of sleep modes in backbone networks with limited configurations", in *20th international conference on software, telecommunications and computer networks (SoftCOM)*, 2012.

[52] A. Muhammad, P. Monti, I. Cerutti, L. Wosinska, P. Castoldi, and A. Tzanakaki, "Energy-efficient WDM network planning with dedicated protection resources in sleep mode", in *IEEE global telecommunications conference (GLOBECOM)*, Dec. 2010.

[53] F. Idzikowski, S. Orlowski, C. Raack, H. Woesner, and A. Wolisz, "Dynamic routing at different layers in IP-over-WDM networks — maximizing energy savings", *Optical switching and networking*, vol. 8, no. 3, pp. 181–200, 2011, Special Issue on Green Communications and Networking.

[54] F. Idzikowski, "Power consumption of network elements in ip over wdm networks", Telecommunication Networks Group, Technical University Berlin, Tech. Rep. TKN-09-006, Jul. 2009.

[55] M. Pióro and D. Medhi, *Routing, flow, and capacity design in communication and computer networks*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004, ISBN: 0125571895.

[56] *IBM ILOG CPLEX optimizer*. [Online]. Available: `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`.

[57] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo", *IEEE journal on selected areas in communications*, vol. 29, no. 9, pp. 1765–1775, 2011. [Online]. Available: `http://www.topology-zoo.org/`.

[58] G. Ferro, *What is OpenFlow?* [Online]. Available: `http://content.ipspace.net/get/What%20Is%20OpenFlow.pdf`.

[59] J. Humble and D. Farley, *Continuous delivery: Reliable software releases through build, test, and deployment automation*, 1st. Addison-Wesley Professional, 2010.

[60] M. Cohn, *Succeeding with agile: Software development using scrum*. Addison-Wesley Professional, 2009.

[61] A. Scott, "Introducing the software testing ice-cream cone (anti-pattern)", *Watirmelon - a software testing blog by alister scott*, Jan. 2012. [Online]. Available: `https://watirmelon.com/2012/01/31/introducing-the-software-testing-ice-cream-cone/`.

[62] R. Lloyd, "Metric mishap caused loss of NASA orbiter", *CNN.com*, Sep. 1999. [Online]. Available: `http://edition.cnn.com/TECH/space/9909/30/mars.metric.02/index.html`.

[63] D. Haeffner, *The selenium guidebook*, 2013. [Online]. Available: `https://seleniumguidebook.com/`.

[64] A. Beltran, *Getting started with PhantomJS*. Packt Publishing, 2013.

[65] D. Lebrun, S. Vissicchio, and O. Bonaventure, “Towards test-driven software defined networking”, in *Network operations and management symposium (NOMS)*, May 2014.

[66] N. Brownlee and K. Claffy, “Understanding internet traffic streams: Dragonflies and tortoises”, *IEEE communications magazine*, vol. 40, no. 10, pp. 110–117, Oct. 2002.

[67] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild”, in *10th ACM SIGCOMM conference on internet measurement (IMC)*, ser. IMC ’10, Melbourne, Australia, 2010, pp. 267–280.

[68] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, “The nature of data center traffic: Measurements & analysis”, in *9th ACM SIGCOMM conference on internet measurement (IMC)*, ACM, 2009, pp. 202–208.

[69] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia, “SDN-based application-aware networking on the example of YouTube video streaming”, in *2nd european workshop on software defined networks (EWSDN)*, Oct. 2013, pp. 87–92.

[70] E. Nordström, D. Shue, P. Gopalan, R. Kiefer, M. Arye, S. Ko, J. Rexford, and M. J. Freedman, “Serval: An End-Host Stack for Service-Centric Networking”, in *9th USENIX symposium on networked systems design and implementation (NSDI)*, 2012, pp. 85–98.

[71] A. R. Curtis, W. Kim, and P. Yalagandula, “Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection”, in *IEEE infocom*, 2011, pp. 1629–1637.

[72] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, “Hedera: Dynamic flow scheduling for data center networks”, in *7th USENIX conference on networked systems design and implementation (NSDI)*, vol. 10, 2010, pp. 19–19.

[73] N. Farrington, G. Porter, S. Radhakrishnan, H. H. Bazzaz, V. Subramanya, Y. Fainman, G. Papen, and A. Vahdat, "Helios: A hybrid electrical/optical switch architecture for modular data centers", *ACM SIGCOMM computer communication review*, vol. 41, no. 4, pp. 339–350, 2011.

[74] T. Mori, M. Uchida, R. Kawahara, J. Pan, and S. Goto, "Identifying elephant flows through periodically sampled packets", in *4th ACM SIGCOMM conference on internet measurement (IMC)*, ACM, 2004, pp. 115–120.

[75] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, "FlowSense: Monitoring network utilization with zero measurement cost", in *Passive and active measurement*, Springer, 2013, pp. 31–41.

[76] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic matrix estimator for openflow networks", in *10th international conference on passive and active measurement (PAM)*, Zurich, Switzerland, 2010.

[77] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with OpenSketch", in *10th USENIX symposium on networked systems design and implementation (NSDI)*, 2013, pp. 29–42.

[78] Big Switch Networks, *Big Tap Monitoring Fabric Ver 4.5 Datasheet.* [Online]. Available: `http://bigswitch.com/sites/default/files/big_tap_monitoring_fabric_v4.5.pdf`.

[79] *The VSS unied visibility plane*, 2015. [Online]. Available: `http://www.vssmonitoring.com/unified-visibility-plane/pdf/Unified-Visibility-Plane-Whitepaper.pdf`.

[80] S. Hong, R. Baykov, L. Xu, S. Nadimpalli, and G. Gu, "Towards SDN-defined programmable BYOD (bring your own device) security", in *Network and distributed system security symposium (NDSS)*, Feb. 2016.

[81] ETSI, "Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV", ETSI GS NFV 003 V1.2.1, Dec. 2014.

[82] TrustedSec, *CHS hacked via heartbleed vulnerability*, Aug. 2014. [Online]. Available: `https://www.trustedsec.com/august-2014/chs-hacked-heartbleed-exclusive-trustedsec/`.

[83] G. Cluley, *Heartbleed blamed for hack that put 4.5 million patients at risk*, Aug. 2014. [Online]. Available: http://grahamcluley.com/2014/08/heartbleed-chs-hack/.

[84] E. Mueller, *The agile admin: What is devops?* [Online]. Available: http://theagileadmin.com/what-is-devops/.

[85] J. Douglas, *Deploying at GitHub*. [Online]. Available: https://github.com/blog/1241-deploying-at-github.

[86] J. Jenkins, "Velocity culture (the unmet challenge in ops)", in *O'reilly velocity conference*, Jun. 2011.

[87] R. Ahmed and R. Boutaba, "Design considerations for managing wide area software defined networks", *IEEE communications magazine,*, vol. 52, no. 7, pp. 116–123, Jul. 2014.

[88] H. Kim and N. Feamster, "Improving network management with software defined networking", *IEEE communications magazine*, vol. 51, no. 2, pp. 114–119, Feb. 2013.

[89] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, *Manifesto for agile software development*, 2001. [Online]. Available: http://www.agilemanifesto.org/.

[90] A. Ahmed, S. Ahmad, N. Ehsan, E. Mirza, and S. Sarwar, "Agile software development: Impact on productivity and quality", in *International conference on management of innovation and technology (ICMIT)*, Jun. 2010, pp. 287–291.

[91] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams, "Do faster releases improve software quality? an empirical case study of Mozilla Firefox", in *IEEE working conference on mining software repositories (MSR)*, Jun. 2012, pp. 179–188.

[92] J. Roche, "Adopting devops practices in quality assurance", *Communications of the ACM*, vol. 11, no. 9, Nov. 2013.

[93] R. Harmes, *Flipping out*, Dec. 2009. [Online]. Available: `http://code.flickr.net/2009/12/02/flipping-out/`.

[94] *Cucumber wiki: Gherkin*. [Online]. Available: `https://github.com/cucumber/cucumber/wiki/Gherkin`.

[95] O. Foundation, *OpenStack Heat*. [Online]. Available: `https://wiki.openstack.org/wiki/Heat`.

[96] HashiCorp, *Terraform*. [Online]. Available: `https://www.terraform.io/`.

[97] Chef, Inc., *Chef*. [Online]. Available: `http://www.chef.io/chef/`.

[98] K.-c. Lan and J. Heidemann, "A measurement study of correlations of internet flow characteristics", *Computer networks*, vol. 50, no. 1, pp. 46–62, 2006.

[99] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. Parulkar, "ONOS: Towards an open, distributed SDN OS", in *3rd workshop on hot topics in software defined networking (HotSDN)*, Chicago, Illinois, USA, 2014.

[100] G. Kousiouris, D. Kyriazis, T. Varvarigou, E. Oliveros, and P. Mandic, "Taxonomy and state of the art of service discovery mechanisms and their relation to the cloud computing stack cloud computing stack", in, D. P. Kyriazis, T. A. Varvarigou, and K. G. Konstanteli, Eds. Information Science Reference - Imprint of: IGI Publishing, 2012, ch. 11.

[101] HashiCorp, *Consul*. [Online]. Available: `https://www.consul.io/`.

[102] D. M'Raihi, S. Machani, M. Pei, and J. Rydell, *TOTP: time-based one-time password algorithm*, RFC 6238 (Informational), Internet Engineering Task Force, May 2011.

[103] *Meteor*. [Online]. Available: `https://www.meteor.com/`.

[104] S. Cheshire, *IPv4 Address Conflict Detection*, RFC 5227 (Proposed Standard), Internet Engineering Task Force, Jul. 2008.

[105] C. Scott, A. Wundsam, S. Whitlock, A. Or, E. Huang, K. Zarifis, and S. Shenker, "Automatic troubleshooting for SDN control software", *Online*, 2013.

[106] ——, "How did we get into this mess? isolating fault-inducing inputs to sdn control software", University of California at Berkeley, Tech. Rep. UCB/EECS-2013-8, 2013.

[107] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks", in *USENIX workshop on hot topics in management of internet, cloud, and enterprise networks and services (Hot-ICE)*, 2012.

[108] R. Sherwood and K.-K. Yap, *Cbench controller benchmarker*. [Online]. Available: `http://sourceforge.net/projects/cbench/`.

[109] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A flexible OpenFlow-controller benchmark", in *European workshop on software defined networks (EWSDN)*, Darmstadt, Germany, Oct. 2012.

[110] S. Bradner and J. McQuaid, *RFC2544: Benchmarking Methodology for Network Interconnect Devices*, IETF, 1999.

[111] R. Asati, C. Pignataro, F. Calabria, and C. Olvera, *RFC26201: Device Reset Characterization*, IETF, 2011.

[112] S. Bradner, K. Dubray, J. McQuaid, and A. Morton, *RFC6815: Applicability Statement for RFC 2544: Use on Production Networks Considered Harmful*, IETF, 2012.

[113] *Y.1564: Ethernet service activation test methodology*, ITU-T, 2011.

[114] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance Characteristics of Virtual Switching", in *IEEE international conference on cloud networking (CloudNet)*, 2014.

[115] A. Morton, "Considerations for benchmarking virtual network functions and their infrastructure", Internet-Draft draft-morton-bmwg-virtual-net-03, 2015. [Online]. Available: `http://www.ietf.org/internet-drafts/draft-morton-bmwg-virtual-net-03.txt`.

[116] Overture, Brocade, Intel, Spirent, and Integra, *NFV performance benchmarking for vCPE*, Executive Summary, 2015.

[117] J. DiGiglio and D. Ricci, *High performance, open standard virtualization with NFV and SDN*, White paper, Intel Corporation and Wind River, 2013. [Online]. Available: `https://www.intel.eu/content/dam/www/public/us/en/documents/white-papers/communications-virtualization-snd-nfv-paper.pdf`.

[118] D. Cotroneo, L. De Simone, A. Iannillo, A. Lanzaro, and R. Natella, "Dependability evaluation and benchmarking of network function virtualization infrastructures", in *IEEE conference on network softwarization (NetSoft)*, 2015.

[119] S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack", in *IEEE annual international symposium on field-programmable custom computing machines (FCCM)*, 2014.

[120] X. Ge, Y. Liu, D. H. Du, L. Zhang, H. Guan, J. Chen, Y. Zhao, and X. Hu, "OpenANFV: Accelerating network function virtualization with a consolidated framework in Openstack", in *ACM conference on SIGCOMM*, 2014.

[121] Z. Bronstein, E. Roch, J. Xia, and A. Molkho, "Uniform handling and abstraction of NFV hardware accelerators", *IEEE network*, 2015.

[122] L. Nobach and D. Hausheer, "Open, elastic provisioning of hardware acceleration in NFV environments", in *International conference and workshops on networked systems (NetSys)*, 2015.

[123] J. Hwang, K. Ramakrishnan, and T. Wood, "NetVM: High performance and flexible networking using virtualization on commodity platforms", in *USENIX symposium on networked systems design and implementation (NSDI)*, 2014.

[124] J. Martins, M. Ahmed, C. Raiciu, *et al.*, "ClickOS and the art of network function virtualization", in *11th USENIX symposium on networked systems design and implementation (NSDI)*, Apr. 2014, pp. 459–473.

[125] G. N. Purdy, *Linux iptables pocket reference*. O'Reilly & Associates, 2004, vol. 1.

[126] J. Xu and W. Su, "Performance evaluations of cisco ASA and linux iptables firewall solutions", Master's thesis, Halmstad University, 2013.

[127] C. Sheth and R. Thakker, "Performance evaluation and comparative analysis of network firewalls", in *International conference on devices and communications (ICDeCom)*, 2011.

[128] K. Salah, K. Elbadawi, and R. Boutaba, "Performance modeling and analysis of network firewalls", *IEEE transactions on network and service management*, 2012.

[129] B. Hickman, D. Newman, S. Tadjudin, and T. Martin, *RFC3511: Benchmarking methodology for firewall performance*, IETF, 2003.

[130] S. University, *OpenFlowJ - OpenFlow software library in Java*, 2010. [Online]. Available: `https://openflow.stanford.edu/bugs/browse/OFJ`.

[131] Oracle, *Java NIO selector*, 2013. [Online]. Available: `http://docs.oracle.com/javase/7/docs/api/java/nio/channels/Selector.html`.

[132] W. Foundation, *Wireshark - go deep*, 2013. [Online]. Available: `http://www.wireshark.org/`.

[133] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an operating system for networks", *SIGCOMM computer communications review*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[134] BigSwitchNetworks, *Floodlight*. [Online]. Available: http://www.projectfloodlight.org/floodlight/.

[135] Cisco Systems, *Cisco catalyst 6500 series, 7600 series ASA services module*. [Online]. Available: http://www.cisco.com/c/en/us/products/interfaces-modules/catalyst-6500-series-7600-series-asa-services-module/index.html.

[136] Cisco Systems, Inc, *Cisco adaptive security virtual appliance (ASAv) data sheet*. [Online]. Available: http://www.cisco.com/c/en/us/products/collateral/security/adaptive-security-virtual-appliance-asav/datasheet-c78-733399.html.

[137] Spirent Communications, Inc., *Spirent testcenter C1 portable layer 2-7 system*. [Online]. Available: http://www.spirent.com/-/media/Datasheets/Broadband/PAB/SpirentTestCenter/STC_C1-Appliance_Datasheet.pdf.

[138] J. H. Salim, R. Olsson, and A. Kuznetsov, "Beyond softnet", in *5th annual linux showcase & conference*, 2001.

[139] *Signaling is growing 50% faster than data traffic*, White Paper, Nokia Siemens Networks, 2012. [Online]. Available: http://networks.nokia.com/system/files%20/document/signaling_whitepaper_online_version_final.pdf.

[140] BISDN, *The openflow extensible datapath daemon (xdpd)*. [Online]. Available: https://github.com/bisdn/xdpd.

[141] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO", in *ACM/IEEE symposium on architectures for networking and communications systems (ANCS)*, 2015.

[142] L. Rizzo, “Netmap: A novel framework for fast packet I/O”, in *21st usenix security symposium (USENIX Security)*, Aug. 2012, pp. 101–112.

[143] R. Love, *Linux kernel development*, 3rd. Addison-Wesley Professional, 2010.

[144] R. Prasad, M. Jain, and C. Dovrolis, “Effects of interrupt coalescence on network measurements”, in *4th international workshop on passive and active measurement (PAM)*, 2004, pp. 247–256.

[145] B. Sigoure, *How long does it take to make a context switch?*, Nov. 2014. [Online]. Available: `http://blog.tsunanet.net/2010/11/how-long-does-it-take-to-make-context.html`.

[146] S. Larsen, P. Sarangam, R. Huggahalli, and S. Kulkarni, “Architectural breakdown of end-to-end latency in a TCP/IP network”, *International journal of parallel programming*, vol. 37, no. 6, 2009.

[147] Cisco Systems and Intel Corporation, *NFV Partnership*, Joint Whitepaper, http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cisco-nfv-partnership-paper.pdf, 2015.

[148] T. Barbette, C. Soldani, and L. Mathy, “Fast userspace packet processing”, in *11th ACM/IEEE symposium on architectures for networking and communications systems (ANCS)*, Oakland, California, USA, 2015, pp. 5–16.

[149] D. Manfield, P. Tran-Gia, and H. Jans, “Modelling and performance of inter-processor messaging in distributed systems”, *Performance evaluation*, vol. 7, 1987.

[150] P. Tran-Gia, *Zeitdiskrete Analyse verkehrstheoretischer Modelle in Rechner- und Kommunikationssystemen - 46. Bericht über verkehrstheoretische Arbeiten*, 1988.

[151] ——, “Discrete-time analysis technique and application to usage parameter control modelling in ATM systems”, in *8th australian teletraffic research seminar*, Melbourne, Australia, Dec. 1993.

[152] A. Brown, N. Forsgren, J. Humble, N. Kersten, and G. Kim, *2016 state of devops report*. [Online]. Available: `https://puppet.com/resources/white-paper/2016-state-of-devops-report`.

ISSN 1432-8801