Julius-Maximilians-Universität Würzburg
Institut für Informatik
Lehrstuhl für Kommunikationsnetze
Prof. Dr. Tobias Hoßfeld

Performance Evaluation of Next-Generation Data Plane Architectures and their Components

Stefan Geißler

Würzburger Beiträge zur
Leistungsbewertung verteilter Systeme

Bericht 02/21

Würzburger Beiträge zur Leistungsbewertung verteilter Systeme

Herausgeber

Prof. Dr. Tobias Hoßfeld, Prof. Dr.-Ing. Phuoc Tran-Gia
Universität Würzburg
Institut für Informatik
Lehrstuhl für Kommunikationsnetze
Am Hubland
D-97074 Würzburg

Tel.: +49-931-31-86631
Fax.: +49-931-31-86632
email: tobias.hossfeld@uni-wuerzburg.de

Satz

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

ISSN 1432-8801

Performance Evaluation of Next-Generation Data Plane Architectures and their Components

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

vorgelegt von

Stefan Geißler

geboren in

Ansbach

Würzburg 2021

Eingereicht am: 15.11.2021
bei der Fakultät für Mathematik und Informatik
1. Gutachter: Prof. Dr. Tobias Hoßfeld
2. Gutachter: Prof. Dr.-Ing. Poul Einar Heegaard
3. Gutachter: Prof. Dr.-Ing. Wolfgang Kellerer
Tag der mündlichen Prüfung: 10.03.2022

Danksagung

Herzlich danken möchte ich an dieser Stelle all denjenigen, die mir auf dem Weg zur Promotion zur Seite standen und mich fachlich wie persönlich stets unterstützt und gefördert haben.

Zunächst und vor allem gebührt Dank meinem Doktorvater Herrn Professor Dr. Tobias Hoßfeld für die Betreuung meines Dissertationsvorhabens und die herausragende Zusammenarbeit. Ebenfalls möchte ich mich bei Herrn Professor Dr.-Ing. Phuoc Tran-Gia bedanken, der mich in den frühen Jahren meiner Promotion als Betreuer begleitet hat. Er war es, der mir im April 2016 überhaupt die Möglichkeit bot, das Promotionsvorhaben an seinem Lehrstuhl anzutreten und mir auch über seinen Ruhestand hinaus mit Rat und Tat zur Seite stand. Ich möchte mich bei beiden neben der ausgezeichneten fachlichen Unterstützung besonders für das respektvolle Miteinander und die wertvollen Erfahrungen herzlich bedanken.

Ebenfalls danke ich Herrn Professor Dr. Thomas Zinner, der mich als ehemaliger Forschungsgruppenleiter maßgeblich geprägt hat. Neben seiner fachlichen Kompetenz hat er mich besonders durch die von ihm geförderte Zusammenarbeit innerhalb der Gruppe stetig motiviert. In zahlreichen, teilweise auch hitzigen Diskussionen konnte ich mich sowohl fachlich als auch persönlich weiterentwickeln. Dafür und für all die Erfahrungen auch außerhalb der beruflichen Zusammenarbeit, möchte ich mich von ganzem Herzen bedanken.

Zudem möchte ich mich bei Herrn Dr. Florian Wamser bedanken, der mich als Gruppenleiter in der letzten Phase meiner Promotion unterstützt hat und durch steten Austausch und anregende Diskussionen ebenfalls zum Gelingen dieser Arbeit beigetragen hat.

Bei den Herren Dres. Steffen Gebert, David Hock und Matthias Hartmann möchte ich mich für die Betreuung während meiner Studienzeit und damit auch für den frühen Kontakt zum Lehrstuhl bedanken. Ebenfalls danke ich Herrn Dr. Michael Seufert, der mir während meiner Zeit stets mit fachlichem und kollegialem Rat zur Seite stand. Außerdem gilt mein Dank Herrn Professor Dr. Samuel Kounev und Herrn Dr. Aleksandar Milenkoski, die mich während meiner Masterarbeit am Lehrstuhl für Software Engineering betreut haben.

Für die immer angenehme, kreative und stets unterhaltsame Atmosphäre am Lehrstuhl sowohl während als auch außerhalb der Arbeitszeiten bedanke ich mich bei allen ehemaligen und aktuellen Kolleginnen und Kollegen. Besonderer Dank gilt dabei Herrn Dr. Stanislav Lange sowie Frau Dr. Kathrin Borchert, die sich über die Jahre mit mir ein Büro geteilt haben und stets wertvolle Ansprechpartner waren. Außerdem bedanke ich mich bei den weiteren ehemaligen und derzeitigen Gruppenleitern des Lehrstuhls, Herrn Prof. Dr. Matthias Hirth und Herrn Dr. Florian Metzger, für die gute Zusammenarbeit. Ebenfalls besonders erwähnt werden muss Frau Alison Wichmann für ihre unermüdliche Unterstützung bei der Projektverwaltung, der Abwicklung von Dienstreisen und allen weiteren organisatorischen Aufgaben, die im Hintergrund stattfinden – vielen Dank dafür.

Darüber hinaus möchte ich mich bei allen „Hiwis“ und Studierenden, mit denen ich über die Jahre zusammenarbeiten konnte, für viele wertgeschätzte Erinnerungen bedanken.

Ohne die stete Motivation und Förderung durch meine gesamte Familie – meine Eltern Luise und Richard sowie meine Geschwister Sonja und Bernd – wären mir Studium und Promotion nicht möglich gewesen. Dass sie mir bei jedweden Herausforderungen zur Seite stehen, weiß ich sehr zu schätzen. Zuletzt und ganz besonders danke ich meiner Frau Anja, die mir ein unschätzbarer Rückhalt ist und ohne die diese Arbeit in dieser Form wohl nicht entstanden wäre.

Contents

1 Introduction

Since the introduction of what is known today as the Internet, the number of active users has reached 5.2 billion in March 2021 [24], without even taking into account the influx of smart devices deployed in Industry 4.0, smart cities and the Internet-of-Things (IoT) in general. Additionally, the strain on network infrastructures in both Wide Area Network (WAN) and datacenter environments is amplified by the introduction of new, resource intensive applications such as video streaming or cloud gaming. In the years 2020 and 2021, the rise of remote working due to the COVID-19 pandemic has once again shown the importance of reliable and ubiquitous access to broadband internet in today's modern world.

All of these reasons — growing demand, increasing application requirements, increasing societal dependency — drive operators, industry and academia to constantly reinvent network infrastructures, systems and mechanisms with the goal to develop solutions to fulfill the ever-growing demands of users. Depending on the specific stakeholder, a multitude of different and competing criteria need to be taken into account. Network operators aim to achieve a high Quality of Service (QoS) by providing highly reliable network connectivity to their customers. Service operators aim for maximum user satisfaction, hence Quality of Experience (QoE). Finally, end-users expect reliable services using various, heterogeneous end-devices and access technologies. At the same time, the cost-efficiency of every component involved in these complex systems needs to be as high as possible.

In order to achieve these goals, several novel softwarization paradigms have been suggested and investigated in recent years. Initially, the goal of Software-Defined Networking (SDN) was to physically and logically separate the data

from the control plane and define open interfaces to enable vendor-agnostic programmability. To this end, control components have been removed from data plane devices and were instead aggregated at a logically centralized controller [25]. This kind of control-centralization enables new, dynamic network configuration and re-configuration as controllers are now able to maintain global knowledge of systems and make forwarding decisions based on this global knowledge. At the same time, data plane devices become increasingly more simple, as devices simply accept instructions from a local control entity and do not have to make their own decisions based on limited knowledge.

The next step in this softwarization of networks is the introduction of Network Function Virtualization (NFV). While data plane devices were largely expected to be dedicated hardware components in the early days of SDN, NFV aims to replace Application-Specific Integrated Circuits (ASICs) through software solutions running on commercial off-the-shelf (COTS) hardware. This usage of software over static middleboxes aims to improve the efficiency of networks, as software components can be dynamically scaled based on current demands. Furthermore, software tools are far easier to maintain and update than hardware middleboxes, further reducing the cost of operating networks.

However, processing network traffic in software is, in general, less performant than dedicated hardware appliances and comes with new challenges. On the one hand, novel monitoring approaches to reliably assess the performance of functions both before and after deployment are required [26]. On the other hand, mechanisms and models to reliably and accurately estimate the performance of software-based network functions are crucial for the development, dimensioning, and operation of software-based network infrastructures.

In addition to the performance aspects of software solutions, the interoperability with existing, potentially legacy, network infrastructures needs to be maintained in order to ensure reliable network operation. The operation of networks consisting of legacy devices, programmable hardware as well as software-based elements requires the investigation of new control plane mechanisms [27].

This monograph covers both technical and analytical mechanisms to assess the performance of software-based network functions as well as complex service-function chains in the area of microservice architectures. We propose a novel monitoring approach that enables low-overhead monitoring of high-performance software-based network functions. Subsequently, we apply our monitoring approach to acquire measurements with the purpose of developing a detailed, discrete-time model of a software router using state-of-the-art acceleration techniques. We show that the integration of software solutions into existing networks is possible while maintaining system performance. Through the abstraction of data plane components, a standardized SDN controller can be used to manage a heterogeneous landscape of data plane devices. Finally, we perform a detailed, simulative performance evaluation of a complex microservice architecture in the context of IoT. We propose simulation models for both the workload profile and the system architecture to allow investigations of various Key Performance Indicators (KPIs). In summary, we identify and contribute to answering the following research questions.

- How to monitor the processing performance of software-based network functions?
- How to predict critical KPIs of software-based network functions under varying load levels?
- How to ensure the interoperability of software solutions, programmable hardware and SDN-enabled data plane devices?
- How to model and assess the performance of complex microservice-based packet processing architectures?

1.1 Scientific Contribution

In the following paragraphs, the scientific contributions in each of the areas covered in this thesis are outlined. Note that a more detailed description of contributions and investigated research questions is presented at the beginning of each chapter, respectively. Figure 1.1 shows a selection of research activities in the context of this work. Thereby, single publications are categorized along the x-axis by their respective research subject, NFV, SDN or Applications in the context of NFV. Additionally, the y-axis shows the applied methodology, namely modeling, measurements and simulation. The figure also encodes the specific research area as the title of each box, whether the references are covered in this thesis through the used font color and the chapter that each of the topics is presented in by the circled number.

Performance Monitoring. In order to deepen our understanding of the performance characteristics of software-based network functions, sophisticated monitoring mechanisms that allow the assessment of relevant KPIs are crucial. To this end, in Chapter 2 of this monograph, we present a novel approach to monitor accurate packet processing times of network functions. Our approach of in-stack monitoring (Section 2.2) relies on mapping packet ingress and egress events directly in the used network stack in order to perform low-overhead monitoring during both development and after deployment. To this end, we develop KOMon, a proof-of-concept implementation based on the New API (NAPI) network stack used in the Linux Kernel. We evaluate both accuracy and overhead of our approach through measurements and provide a discussion regarding its application and limitations in practice.

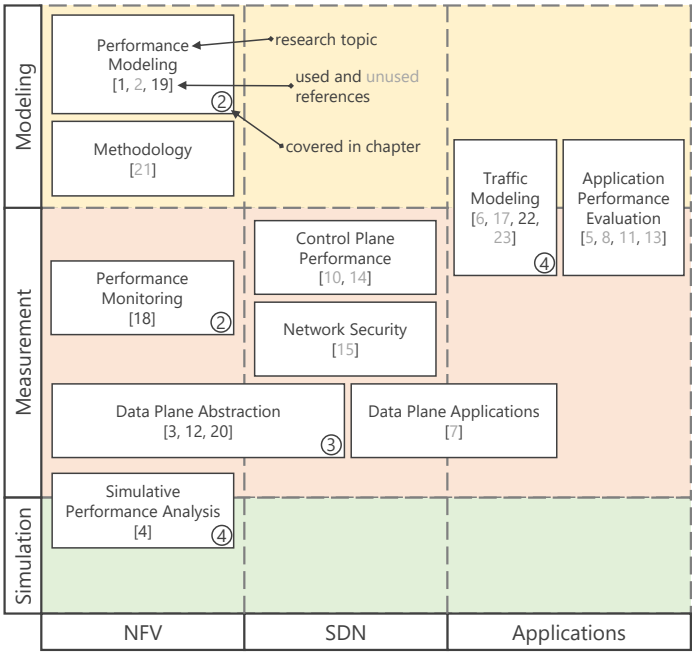


Figure 1.1: Categorization of research work conducted by the author. References noted in opaque font are not covered in this manuscript. The circled numbers at the bottom right annotate the chapter that covers the respective publications.

Performance Modeling. Subsequently, in Chapter 2 of this thesis, we apply our monitoring approach to obtain measurement values for a state-of-the-art software router. On the basis of the obtained processing time measurements, we develop an accurate, discrete-time model of Cisco's Vector Packet Processing (VPP) that is capable of predicting several crucial KPIs (Section 2.4). These include the queue size, response time, waiting time as well as packet loss probability under differing load levels and system parameters. We present a detailed

description of our model and validate its predictive accuracy using the previously obtained measurement values. The developed model in combination with the proposed monitoring approach enables both the accurate monitoring of network function performance and the prediction of relevant KPIs

Data Plane Abstraction. Following the performance evaluation of single component network functions, we introduce a data plane abstraction approach in Chapter 3. Here, the general concept of data plane abstraction through protocol translation is introduced and the design of our proof-of-concept implementation is outlined. The presented concept is based on the idea of actively intercepting control plane messages of various protocols between control and data plane. Before relaying the intercepted messages, various modifications to the content are made to transparently map between different data plane technologies and control plane applications. We thereby demonstrate the practicality of translating between OpenFlow [28] and P4 [29] as well as hardware and software solutions. We also demonstrate the possibility of emulating virtual devices towards the control plane, while realizing the data plane operations through multiple, potentially heterogeneous data plane devices. Finally, we conduct performance measurements and discuss the impact of our concept on the control plane performance.

Traffic Modeling. In order to perform accurate performance evaluation of a complex microservice system, the workload profile imposed on the system is established in Chapter 4. To this end, we perform an extensive analysis of a 31-day-long network trace containing more than 1.4 billion messages and over 7 terabyte of raw data. The trace has been obtained at the ingress of a real world Mobile Virtual Network Operator (MVNO) mobile network core. We dissect the trace contents regarding message distribution and identify a simple feature set that allows the classification of individual IoT devices based on their mobile signaling traffic (Section 4.2). Based on the identified features, we perform unsupervised machine learning through k-means clustering and evaluate the differ-

ences in device behavior between resulting clusters. Finally, we highlight critical properties of the aggregated arrival process resulting from the superposition of the signaling traffic of individual devices. We show that the Markov assumption commonly made in standardization and literature does not necessarily hold in practice, but can be restored through preliminary device classification.

Simulative Performance Analysis. Finally, after establishing a workload profile, Chapter 4 presents our simulation model and case study on the applicability of our model. Here, we outline the architecture of the real world system and present a suitable abstraction. Subsequently, we introduce both a signaling model that dictates the behavior of individual devices and a core network model that describes the available resources as well as their interactions (Section 4.3). After validating the accuracy of the proposed simulation model through measurements in both dedicated testing environments and a productive system, we perform a case study regarding system dimensioning. Finally, we evaluate several overload control mechanisms with regard to their capability to ensure efficient operation under extreme overload conditions. Our validation and investigation of overload control mechanisms shows that the developed simulation model is capable accurately of replicating conditions in the real world system. Hence, the model can be used to investigate complex extensions of the system to identify efficiency issues, thereby increasing the optimization potential of the system.

1.2 Thesis Outline

In the following, we present a brief outline of the contributions made in this thesis. Figure 1.2 presents an abstract view of the system evaluated in the following chapters. The annotations in read describe the chapter in which each part of the system is discussed. The figure shows data plane components and their interactions in green. Each network function can thereby be realized using software, programmable hardware or SDN-enabled whitebox switches. As is common in modern, softwarized networks, the control entity is realized using a logically centralized control and management unit, shown at the top of the figure.

Chapter 2 deals with the performance evaluation of single component software-based network functions. To this end, we conduct measurements and develop analytical models to advance our general understanding of the behavior of such network components.

After establishing accurate performance models of single network functions, Chapter 3 highlights the feasibility of integrating multiple, potentially technologically heterogeneous network functions into a single data plane. To this end, we develop an abstraction tool mediating between control and data plane. We show the feasibility of multi-component and multi-technology data plane solutions and discuss performance implications of our abstraction approach.

Finally, Chapter 4 covers the performance evaluation of a complex system of multiple software-based network functions at the example of a real world virtualized MVNO mobile core network. To achieve this, we establish a load profile through detailed analysis of an extensive network trace obtained from a productive mobile core network. We then apply the gained insights to develop a highly detailed simulation model of the real world mobile core system. We highlight the application possibilities of our model for bottleneck detection, system dimensioning as well as the investigation of systemic extensions like overload control mechanisms.

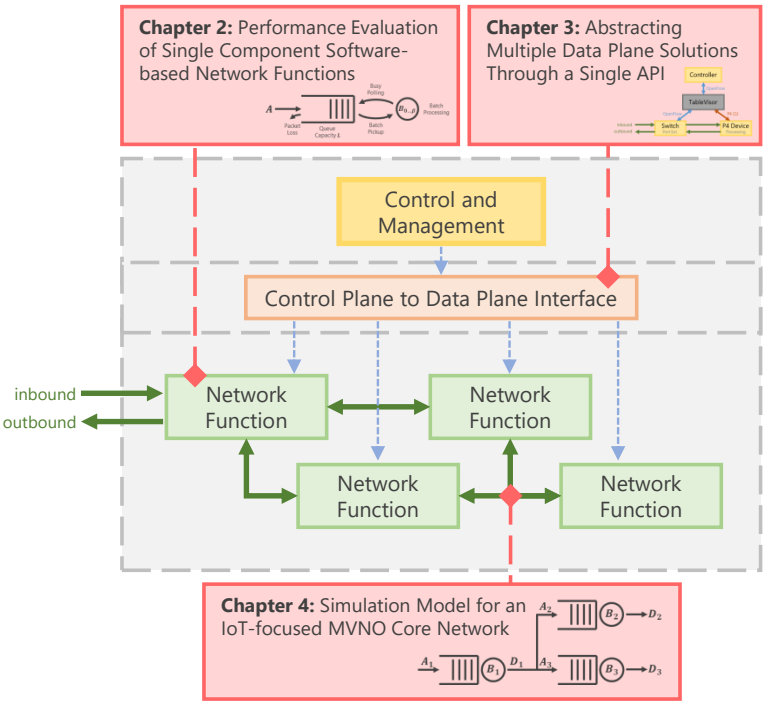


Figure 1.2: Abstract system architecture with annotations on which chapter of the thesis covers which aspect of the system.

2 Performance Evaluation of Single Component Software-based Network Functions

The concept of NFV has initially been introduced by Chiosi *et al.* in 2012 [30] in order to react to increasing demands for network operators to be able to quickly react to technological advancements and gain access to new revenue earning network services.

Up to this point, network services and functions are mostly realized via the development and deployment of dedicated, proprietary hardware appliances. These ASICs require significant capital investment both during development and deployment, as well as during operation. At the same time, the static nature of hardware components limits the ability to flexibly react to new and changing demands. In order to alleviate these issues and trigger the next evolution of networked systems, the paradigm of NFV promises to reduce equipment cost, simplify maintenance tasks and reduce power consumption. Simultaneously, the goal is to enable more flexible network operation and speed up the time to market.

The idea behind the NFV concept is the migration of network components and functions previously realized through proprietary middleboxes towards open software solutions that can be hosted in virtual environments using readily available COTS servers. This allows for dynamic resource scaling of these Virtualized Network Functions (VNFs) and enables far greater flexibility than is possible in hardware-based scenarios. However, in order to achieve these am-

bitious goals, several technical challenges require significant attention [30, 31]. Questions regarding management and orchestration [32], security [33], reliability and resilience [34], and finally performance [35] need to be addressed in order to achieve the ambitious goals promised by the NFV paradigm.

First, management and orchestration, deals with problems regarding the efficient operation of software-based network components. On the one hand, these include technical components such as the on-demand instantiation and destruction of VNF instances [36, 37] as well as their efficient placement within networks [38]. On the other hand, the integration of software solutions into existing networks poses significant challenges [39, 40], such as transparent interoperability with legacy equipment.

Second, the introduction of software into the network landscape inherently comes with additional attack surfaces exploitable by malicious parties [41]. At the same time, due to the reduced time to market and shorter cycles between component updates, software solutions are by nature less resilient against misconfiguration and implementation errors than their hardware counterparts [33].

This leads immediately to the third aspect, the reliability and resilience of VNFs. In environments in which five nines reliability is a mission critical factor, deployed components need to be sufficiently resilient against both hardware failures and software errors [41].

Finally, the performance penalty induced by moving from hardware to software is significant. In order to still be able to provide high performance services, new approaches for efficient, software-based packet processing need to be investigated [26, 35]. The performance aspect being the focus of this chapter, the following sections present a more fine-grained taxonomy of current research in this area.

Based on these observations, we identified the following research questions that are covered in this chapter of the monograph.

RQ2.1) How can the processing performance of generic network functions be monitored? This includes the measurement of the total raw processing times as well as the response or sojourn times of packets consisting of the processing as well as the waiting time in buffers and queues.

RQ2.2) Can this be achieved in a VNF-agnostic way while treating network functions as a black box without access to the VNF source code? This includes the elimination of VNF-specific methodology adaptations.

RQ2.3) Can the monitored packet processing times be exploited to develop accurate, generalizable performance models of state-of-the-art network functions? This includes the prediction of key performance indicators such as the expected loss rate, sojourn time and identification of maximal throughput.

In the following chapter, in order to address these research questions, we start in Section 2.1 by providing an overview over the process of handling packets in software when it comes to modern, state-of-the-art network functions. We cover common optimization mechanisms and provide insight into modern frameworks to accelerate software based processing. Furthermore, we provide an overview of selected literature that covers research work related to the contributions made in this chapter. Section 2.2 addresses RQ1 by presenting a novel approach to monitoring the processing performance of generic network functions without the need to access or modify the VNF code. We show both the accuracy of the proposed mechanism and the behavior under load by means of comparing reported values to baseline measurements obtained from experiments using an industrial grade hardware traffic generator in Section 2.3. Subsequently, Section 2.4 presents a general $Gi/Gi^{[x]}/1\text{-}\beta$ single arrival, batch processing queuing model with limited queue size β that allows the prediction of several KPIs such as the waiting time distribution or packet loss probability.

Once again, we present a detailed validation of the model output by comparing against measurements obtained in a dedicated testbed. Finally, Section 2.5 concludes the chapter and provides a discussion of lessons learned regarding the scientific contributions made in this chapter. The main contributions can be summarized as follows.

C2.1) A novel approach to measuring packet processing times of generic, black box network functions called in-stack monitoring.

C2.2) A general discrete-time queuing model able to predict several KPIs of modern network functions using state-of-the-art acceleration techniques.

These contributions have been published in the past and are condensed in this monograph based on the following scientific publications.

- Geißler, S., Lange, S., Wamser, F., Zinner, T., Hoßfeld, T.: "KOMon - Kernel-based Online Monitoring of VNF Packet Processing Times," in International Conference on Networked Systems (NetSys), 2019. [18]
- Lange, S., Linguaglossa, L., Geißler, S., Rossi, D., Zinner, T.: "Discrete-Time Modeling of NFV Accelerators that Exploit Batched Processing," in IEEE Conference on Computer Communications (INFOCOM), 2019. [19]
- Geißler, S., Lange, S., Linguaglossa, L., Rossi, D., Zinner, T., Hoßfeld, T.: "Discrete-Time Modeling of NFV Accelerators that Exploit Batched Processing," in ACM Transactions on Modeling and Performance Evaluation of Computing Systems (ToMPECS), 2021. [1]

2.1 Background and Related Work

In this section, we provide a concise overview of the current landscape regarding software packet processing as well as current research related to the contributions made in this work, namely the assessment of packet processing times of software systems in both live and testbed environments as well as the subsequent development of accurate performance models. This involves both the theoretical and practical performance evaluation of software solutions for efficient packet processing.

2.1.1 Software Packet Processing Overview

We start by describing the general processes involved in receiving and sending packets using software applications. Note that the information provided here is based on Linux Networking. However, due to the level of detail presented in this section, the information is in major parts applicable to other operating systems as well. Figure 2.1 depicts a schematic overview of the processes performed during both reception and transmission of packets. A more detailed, technical description of the process is presented in [42] and [43].

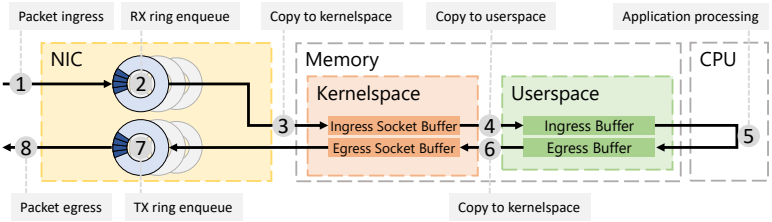


Figure 2.1: Schematic overview of processes involved in software packet processing.

The *packet ingress path* starts with a packet arriving at the Network Interface Card (NIC), depicted as (1) in the figure. Immediately after reception, the NIC controller moves the packet data into a pre-allocated memory region residing in kernel space called *RX ring* (2) using Direct Memory Access (DMA). Note that Figure 2.1 shows RX/TX rings within the NIC whereas technically, these memory regions resign in kernel space. This has been done to indicate the DMA process that does not involve the CPU. Subsequently, the NIC raises an interrupt to notify the CPU about newly available packets, which in turn copies the packet data (3) from the RX ring into a separate memory allocation in kernel space, represented by the *ingress socket buffer* in the figure. From there, user space applications can access the packet data by issuing a receive system call, triggering the CPU to copy the data from kernel to user space (4), where the application can finally perform packet processing (5). The *egress path* is traversed similarly, in the opposite direction. Applications issuing send system calls to move data from user to kernel space (6), and finally into the *TX ring* from where the NIC controller can then collect the packet and physically transmit it over the wire.

Based on this basic mechanism, all-software processing of network traffic has unleashed the possibility to rapidly deploy and update new protocols and features in both the control and the data plane. Particularly, ASICs still dominate the network *core*, where the network fabric performs simple processing like IP forwarding or MPLS switching at several terabits per second. In contrast, all-software stacks are gaining popularity at the network *edge*, where software can deliver feature-rich packet processing for a large variety of protocols at tens to hundreds of gigabits per second. Software routers have been introduced nearly two decades ago [44], but their adoption has been slow due to severe performance bottlenecks, which made the idea appealing but limited to research prototypes. Yet, the situation changed drastically in the last decade, with the introduction of the so-called “kernel-bypass” network stacks [45, 46] that started offering efficient low-level building blocks for multithreaded user-space processing of network traffic at line-rate. As a result, full-blown software stacks, enabling more complex use cases in the SDN and NFV areas started rising in

the software ecosystem. Open Virtual Switch (OVS) [47] and Vector Packet Processor (VPP) [48] are two examples.

To achieve high-speed processing, these software frameworks share commonalities [49] such as the use of lock-free multi-threading as well as the use of *poll-mode batched processing*. While the use of multi-threading allows horizontal scaling and makes each thread independent of the others, the use of *batching* is a distinctive characteristic of modern high-speed packet processing frameworks. Particularly, batching is used for both fetching packets from the Network Interface Card (NIC) by low-level drivers to reduce interrupt pressure [45, 46], and for processing batches of packets in higher-level applications to amortize framework overhead [48–51].

In the following, we provide details regarding several of these extensions and optimizations to the process of receiving, processing and transmitting packets and highlight aspects that are relevant in the context of NFV.

Polling and I/O Batching

Traditionally, the networking stack generates an *interrupt* every time a new packet is received by the NIC, signaling the CPU that all processing should stop in order to deal with packet I/O. Under heavy load, this mechanism is known to be very inefficient, leading to a livelock on the CPU [52]. To alleviate this issue different interrupt mitigation mechanisms have been introduced. One such mechanism is *polling* [53]. Here, at very high traffic rates, one or more CPU cores are being dedicated to continuously check for packets stored in the packet ring without raising any interrupts.

Polling mechanisms are typically coupled with *batching*. Meaning when the CPU polls a device, it gathers a group of contiguous packets in the ring and the whole batch is passed to the processing application. A similar procedure is executed during packet transmission, when packets scheduled to be transmitted are forwarded in batches. Batching is a powerful mechanism that speeds up overall processing, as it amortizes the fixed cost of the I/O process over multiple packets [50, 54] and is as such supported by modern networking stacks like DPDK [45] and netmap [46].

A maximum batch size β is usually defined to specify an upper limit on the number of packets to be taken by an atomic poll operation, so that the size of the polled batch can take any value in $[0, \beta]$. This is done to parametrize the trade-off between the processing efficiency of larger batches and the reduced jitter of smaller batches [55]. The impact of this value on the overall system performance under different circumstances is evaluated later in this work in Section 2.4.5.

Packet Ring and Receive Side Scaling

When packets are received at the NIC, they are written to a buffer, called *packet ring*, that is also accessed by the software to retrieve incoming packets. Writing happens without involving the CPU, using DMA, and does not involve costly memory copy operations. This memory area acts as a *circular queue* which means that when the input rate is higher than the processing rate, the oldest packets might be overwritten by new arrivals. Hence, unlike in classic FIFO queues, older packets are dropped when the buffer is full.

Modern NICs expose multiple RX/TX hardware queues for the same link to allow for more efficient, parallel packet processing. Modern software frameworks like DPDK [45], VPP [48] or netmap [46] can leverage Receive Side Scaling (RSS) [56] to bind different CPU cores to different of these RSS hardware queues. Thereby, incoming traffic is balanced across different RSS queues based on a hashing function, which allows parallelizing packet processing with the number of available CPU cores. Therefore, each CPU is assigned to a separate instance of the software router, managing its own specific RSS queue with its

own packet ring. Since RSS makes each thread independent, it is sufficient to analyze the performance of a single RSS queue as handled by a single core. Indeed, due to the lack of synchronization and locking issues, the aggregated system performance scales linearly with the number of cores [57]. Hence, for modeling purposes, it is sufficient to focus on a single RSS queue.

Compute Batching

More recently, the use of batching has been extended beyond packet I/O and has been applied to the processing of packets as well. Indeed, network function computation can similarly benefit from grouped processing, which is known as *compute batching*.

Shortly, when a VNF is executed on a batch instead of single, sequentially process packets, this allows sharing the overhead of the packet processing frameworks between multiple packets, e.g., all processing instructions are initialized once per batch rather than once every packet. Additionally, it increases the efficiency of the underlying CPU pipelines since the VNF code raises a single miss for the first packet in the batch, but is then subsequently cached in the L1 instruction cache for the remainder of the batch.

Whereas the actual implementation of compute batching differs among frameworks, for example when comparing the compute batching implementations of G-opt [51], DoubleClick [50], FastClick [49] and VPP [48], compute batching in general is an additional technique to increase performance of modern high-speed packet processing frameworks.

2.1.2 Experimental Research Work

The ecosystem of high-speed all-software packet processing has flourished in the last decade with both low-level building-blocks that use I/O batching (e.g netmap [46] and DPDK [45]), as well as high-level full-blown stacks that apply NFV functions with a compute batching paradigm [48–51]. Whereas such frameworks offer a similar set of features, comparison is difficult so that most

related work relies on extensive evaluation campaigns of a single tool – as we do in this thesis using VPP over DPDK.

Previous efforts aimed to evaluate a limited subset of the aforementioned tools [49, 58, 59]. For example, [58] focuses on accelerated low-level frameworks, namely netmap, DPDK, and PF_RING. The authors perform an experimental campaign assessing not only throughput, measured in Mpps, but also consider the impact of factors such as batch size or misses in CPU caches. Similarly, FastClick performance is evaluated over both DPDK and netmap in [49]. Finally, [59] experimentally compares NFV throughput with chains of heterogeneous functions using OVS-DPDK, SR-IOV, and FD.io VPP.

At the same time, significant work has been invested into the development of generic evaluation tools and frameworks that allow the establishment of performance profiles for various network functions, independent of their specific implementation or used platform.

In [60], the authors propose an offline solution to gather functional as well as performance data through static code analysis without execution of the VNF itself. This approach allows the evaluation of arbitrary workload characteristics in an offline manner. To make use of this methodology, however, the codebase of a VNF needs to be accessible, making it impossible to perform black box testing or to evaluate closed source network functions.

The authors of [61] propose another tool for offline performance benchmarking of virtual network functions. The Gym framework enables fully automated performance benchmarking of virtually any VNF type by allowing the user to define custom test cases suitable for the VNF that is to be tested. This and the support for different underlying virtualization platforms make this approach very flexible, while still limiting it to offline, dedicated performance benchmarking.

The aforementioned approaches focus on resource utilization, like CPU time or memory usage, in order to determine VNF performance levels. However, research has shown that this information might not always be sufficient to reliably identify performance bottlenecks [62]. To alleviate this issue, the approach pro-

posed in [63] suggests monitoring VM-to-VM communication to enable online performance monitoring of network functions. However, port mirroring of all incoming and outgoing packets is required by this solution, which imposes a significant performance overhead in a field where processing times and efficiency are crucial.

Another approach proposed by the SONATA-NFV project [64] involves using information exposed by the Linux Kernel to evaluate the performance of virtual network functions. However, the information exposed by the Kernel does not include NFV-specific metrics, like the processing time distribution of packets, and instead only provides generic information like interrupt counters, buffer levels and number of dropped packets. In this context, the mechanism proposed in Section 2.2 exploits a similar methodology of using the */proc/* file system to extend the information provided by the Kernel. Hence, it could be used to increase the monitoring capabilities of the SONATA framework.

A recent direction advocates for a more general approach at the evaluation of software routers, and for the availability for open and honest quantification of novel tools' performance. Authors in [65] propose a methodology to fairly assess the performance of several state-of-the-art software routers in different settings, while the online reports of [66] show the results of several throughput and latency measurements for the latest versions of VPP. Finally, pointed out in [67], the topic of fairly measuring the performance of software routers is delicate and difficult, which further proves the need for flexible approaches such as the one proposed in this thesis, alongside the classical experimental benchmarking.

Due to this lack of easily reproducible and widely applicable approaches that allow for accurate assessments of packet processing times as well as other KPIs, such as buffer fill levels, in both online and dedicated testing environments, we propose in-stack monitoring, a novel approach that can be applied to a wide field of network functions and implementation environments.

The top half of Table 2.1 summarizes the publications discussed in this paragraph and provides a qualitative comparison to the contributions of this chapter.

Table 2.1: Overview of related work in the field of VNF monitoring and modeling. The gray comments provide a short description of the publications covered in the respective row.

	Approach	Online Capable	VNF Agnostic	Overhead	IO Batching	Compute Batching	Response Time	Packet Loss	Queue Size
Monitoring	[60]	✘	✘	NA	Static code analysis				
	[61]	✘	✘	NA	Dedicated testbed				
	[63]	✔	✔	high	Port mirroring				
	[64]	✔	(✔)	low	Relies on kernel data				
Modeling	[68, 69]	Multi-function systems			✘	✘	✔	✔	✔
	[70]	CPU consumption			✘	✘	✘	✘	✘
	[71, 2]	NAPI processing			✘	✘	✔	✔	✔
	[72]	Mean values only			✔	✘	✔	✔	✔
	This Work	✔	✔	low	✔	✔	✔	✔	✔

2.1.3 Modeling Research Work

As outlined before, modern software packet processing frameworks leverage batching to improve processing efficiency. The theory of this mechanism has, in the form of *bulk* queuing systems, long been studied [73]. For Markovian bulk input $M^{[X]}/M/1$ and service $M/M^{[X]}/1$ systems, [74] provides closed form solutions under Poisson arrivals and exponentially distributed service times. Particularly, *bulk-input* Batch Markovian Arrival Processes (BMAP) have been well studied [75–77], and applied to study long-lived TCP connections [78, 79], model aggregated IP traffic [80] or describe parallel processing in cloud environments [81]. Similarly, models featuring batch arrivals and general independent arrival distributions have been proposed as well [82, 83].

Furthermore, several studies regarding *bulk-service* systems have been conducted in the past [84–87]. Similarly, previous work that takes batch-size dependent service times into account does exist as well [88–90]. However, the

complexity of the relation between batch-size and service time is limited in these studies. This relation between batch-size and service time can be arbitrarily complex for the model proposed in this work. Finally, [91, 92] both investigate $M/Gi_r^{(a,b)}/1$ queues and compute the queue size distribution at departure events. Note here that all of these studies contain at least one Markov component or are limited to basic performance indicators, as opposed to the $Gi/Gi^{[X]}/1-L$ study presented in this work.

Models in the context of NFV have also recently appeared [68–72]. In particular, queuing models are used in [68] and [69] to describe software-based networks. Similarly, the authors of [93–95] investigate the impact of autoscaling on 5G networks with both legacy equipment and VNFs as an $M/M/n$ system with variable n. All of these models adopt a global network view and strongly abstract the mechanisms of specific network elements by simply assuming a certain service rate, as opposed to this thesis, in which we provide a detailed model of a single VNF component. Under this perspective, studies closer to ours are [70, 71], which both aim at predicting virtual function performance on multicore systems. Yet, [70] does not take into account mechanisms like batch arrival or batch processing of packets, which both are crucial characteristics of modern NFV routers. In contrast, the authors of [71] assume fixed processing times, which we show not to hold true in practice, and omit a proper experimental validation. In [72], the authors extend previously proposed models for DPDK-based NFVs to take into account IO batching, but omit compute batching. Furthermore, the model only predicts mean values instead of full distributions.

In synthesis, while several models exist that take bulk arrival as well as batch service processes into account, evaluations of real world systems are missing. Furthermore, most solutions are based on the Markovian property of a system, which does not necessarily hold true in the real world. In addition, related approaches in the area of NFV often exploit a high level of abstraction by ignoring details of the software stack like batch processing, interrupt mitigation and busy polling mechanisms. Finally, proper validation of the model outputs based on a comparison to experimental results of a real NFV system is lacking so far. To this

end, we develop a discrete-time model that allows the prediction of several KPIs, such as the waiting time, sojourn time or packet loss probability, and validate the obtained predictions against a state-of-the-art software router implemented using the VPP acceleration framework.

The bottom half of Table 2.1 aggregates the discussed research works and indicates the differences to the contributions made in this chapter.

2.2 KOMon: In-Stack Monitoring of VNF Packet Processing Times

The ability to accurately measure the processing performance of network functions is crucial during both function development and live deployment. Being able to assess the impact of code changes on system performance is critical during early development and before rolling out changes. At the same time, it is important to monitor the performance of functions during operation in order to react to load fluctuations and detect anomalies.

However, reliable and accurate measurement of network function specific KPIs, like packet processing times, is not only relevant in live environments and for monitoring purposes. Instead, the processing time can be used as an input parameter for theoretical models ahead of deployment when it comes to predicting performance under certain circumstances [19, 1, 2]. Additionally, the softwarization of networks is often accompanied by the application of software development paradigms during network function development. Especially in the area of continuous integration and delivery, the availability of fast, reliable, and automatable mechanisms to obtain comparable performance metrics of a new version of a network application is required [96].

To this end we present a novel mechanism based around the idea of *in-stack monitoring*. This approach eliminates the need for port mirroring and allows online monitoring with minimal overhead by hooking into the network stack, which needs to be traversed by every packet destined for a hosted VNF. In the

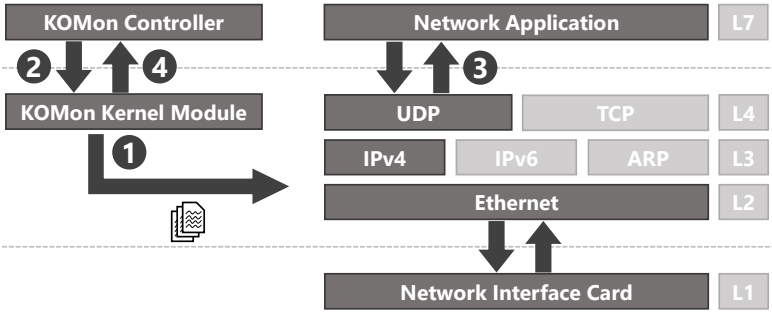


Figure 2.2: Architecture and stack interaction.

following, the architecture, monitoring logic, and a proof-of-concept implementation of KOMon [18] (**K**ernel-based **O**nline **Mon**itoring) is discussed, before a case study using an exemplary VNF is conducted. The code for both the monitoring tool and the VNF used in the case study can be found on GitHub[1]. In the presented use case, we use the network stack of the Linux Kernel v4.11 [97]. The in-stack monitoring approach can, however, be applied to other stack implementations such as DPDK [98], Snabb [99] or VPP [48] since the basic procedure is independent of the underlying network stack. However, the KOMon tool used in the case study is limited to the Linux Kernel stack and some re-implementation effort would be required to adapt the tool to another stack implementation.

The tool consists of two basic components. First, the KOMon Kernel module hooks into the network stack by injecting its monitoring code. Second, the KOMon controller runs in user space and is responsible for configuration and management tasks. Figure 2.2 shows this architecture and its interaction with the network stack. For reference, the right-hand side of the figure shows an abstracted view of the NAPI [100] stack, according to the ISO/OSI model.

[1] https://github.com/lsinfo3/KOMon

Before starting the monitoring procedure the tool requires **initialization (1+2)**. The module needs to be loaded into the Kernel, thus injecting the monitoring code directly into the network stack, thereby attaching its monitoring logic to existing Kernel functions of the network stack. In this use case, we attach two monitoring probes to the network stack responsible for handling UDP traffic. In order to obtain timestamps for incoming datagrams, we inject the monitoring point into the `__udp4_lib_rcv` function that is used to process UDP datagrams before sending them to a user space application. On the other hand, in order to gather timestamps of outgoing packets, we inject a second measurement point into the `udp_sendmsg` function that is used by a UDP socket to send datagrams. After the injection process, the user space controller configures sample size and monitoring interval and thereby defines how often and how many packets are being sampled by the Kernel module.

The Kernel module is now passive until it receives an initial trigger issued by the user space controller and the **monitoring loop (2+3+4)** is activated. After the controller triggers a monitoring interval, the Kernel module is activated, samples the previously configured number of consecutive packets, and calculates their response times. Therefore, at the first measurement point, the timestamps of incoming packets destined for the VNF to be monitored are stored in a ring buffer together with a hash of the packet payload. After the packets have been processed by the VNF and are ready to be sent out, the second measurement point monitors outgoing packets and compares payload hashes to the oldest packet still in the ring buffer. If the hashes match, the timestamps are simply subtracted and the response time is stored as a result. After the preconfigured number of packets have been monitored, the Kernel module returns to its passive mode in order to decrease overhead. The data obtained during the active monitoring phase can then be queried by the user space controller via a *procfs* interface that returns a list of measured response times. This mechanic of matching packets limits the functionality of the proof-of-concept implementation to VNFs that process packets in a FIFO manner and do not alter the payload of packets. The implications and challenges of payload altering and prioritizing

network functions are discussed in Section 2.3.3. It should further be noted that, from a modeling perspective, the times measured by this mechanism represent the response or sojourn times, since it includes at least part of the waiting time of each packet. A more detailed discussion is presented in Section 2.3.2.

Note that the highlighted nodes in Figure 2.2 correspond to monitoring UDP over IPv4 traffic. The Kernel module can, however, be attached to any protocol supported by the network stack with only slight modifications. This integration into the network stack consequently allows KOMon to monitor real VNF traffic and thus eliminates the need for injecting dedicated sampling traffic or traffic mirroring, further reducing the induced performance overhead. Furthermore, KOMon can do so without modifying or even accessing the VNF source code, as the complete monitoring logic is outsourced to the network stack. This provides crucial advantages over similar approaches, as it not only significantly broadens the spectrum of functions the approach can be used on, but also simplifies the deployment of the mechanism, as the monitoring logic is based on the execution platform instead of the particular network function.

In the following, we perform a detailed, measurement based case study with the goal to answer questions regarding both the accuracy of the proposed approach with respect to the replication of baseline values and its behavior in high load scenarios.

2.3 Evaluation of KOMon Monitoring Approach

Using the mechanism introduced before, we now perform a case study and show the accuracy of the monitoring approach by comparing values reported by the KOMon tool to ground truth values obtained from a specifically designed validation network function. To this end, a dedicated testbed as shown in Figure 2.3 has been designed, consisting of a Spirent C1 hardware traffic generator that is directly connected to a bare metal server equipped with an Intel Xeon E5420, 16 GB RAM, and 2x 1 Gbit NICs that is running Ubuntu 16.04.3 LTS and our modified version of Kernel 4.11.

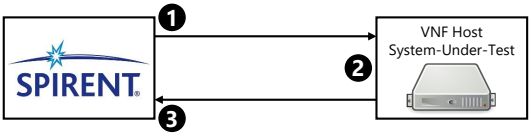


Figure 2.3: Testbed setup used during the evaluation.

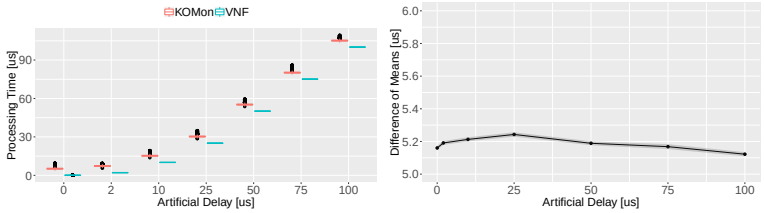
Thereby, the Spirent Testcenter C1 serves as the traffic generator (1) as well as the traffic sink (3). It generates a continuous stream of tagged UDP datagrams that can be uniquely identified via their payload and sends them to the system-under-test hosting the network function (2) over a direct 1 Gbps copper connection. The network function is a simple UDP relay with configurable processing delay. Furthermore, the VNF is able to report baseline processing delays for comparison. This test setup is used throughout the evaluation presented in this work. In the following, we evaluate the accuracy of the KOMon monitoring tool in different scenarios and evaluate the influence of various parameters on its performance. The data used in this evaluation is provided in the public GitHub repository.

2.3.1 Accuracy of Measurement Values

In order to evaluate the accuracy of the values reported by our tool, we conduct a series of measurements with increasing artificial delay values. During this set of experiments, the Spirent C1 generates a continuous stream of 1,000 UDP datagrams per second of size 128, that are received by the VNF, artificially delayed, and returned to the traffic generator.

Measurement Accuracy

First, we investigate the measurement accuracy of the KOMon monitoring tool, meaning the deviation of response time values reported by the tool from the ground truth. Figure 2.4a shows the time along the y-axis. The x-axis shows an



(a) Comparison of VNF reported values and values observed using KOMon for different artificial delays.

(b) Bootstrapped difference of means for different artificial delays.

Figure 2.4: KOMon evaluation at varying artificial delay values.

increasing artificial delay as it is added by the used network function. Thereby, values in red represent the data measured using the KOMon monitoring tool, while the blue data points show the baseline data reported by the VNF. The black markers in this plot mark outliers whose value is either smaller than $Q1 - 1.5 \cdot \text{IQR}$ or larger than $Q3 + 1.5 \cdot \text{IQR}$, where IQR is the interquartile range, Q1 is the 25% quantile and Q3 is the 75% quantile. In this scenario, both the VNF and KOMon sample 10 packets in negative exponentially distributed intervals with a mean of 0.2 seconds. Note that the artificial delay is configured to be the processing time of the network function and the load in this scenario is designed to be low enough to not induce waiting times. Hence, we can compare the processing time values of the VNF with the response time values reported by KOMon.

It can be seen that values monitored using KOMon exhibit a roughly constant offset over the baseline values reported by the VNF for all artificial delay levels. The differences in outliers are due to the fact that both the VNF and KOMon obtain the values through packet sampling, and it cannot be guaranteed that both tools sample the exact same packets. The general presence of these outliers is due to the general purpose operating system used to host the VNF that performs task scheduling that may affect a time measurement in the microsecond realm.

As the measurement points created by KOMon are located in Kernel space, the response time observed by the monitoring tool includes an offset formed by operations happening between the measurement points and the processing performed by the VNF, e.g. copying packet data from Kernel space to user space. This explains the slightly higher variance exhibited by KOMon reported values over the baseline data. In order to quantify this offset as well as validate the observation of the measurement accuracy being independent of the total processing time of the VNF, Figure 2.4b shows the bootstrapped difference of means for different levels of artificial delay. Thereby, 1,000 random samples, each consisting of 10 subsequent packets, have been taken from both the VNF-reported and the KOMon-generated data set, respectively. In order to soften the impact of scheduling on a general purpose operating system, the following figures only contain values up to the 99% quantile of the dataset. For each of the pairs of samples, the difference of means has been calculated before finally determining the 95% confidence interval for the resulting distribution of the difference of means.

Figure 2.4b shows the mean difference of means after 1,000 repetitions along the y-axis with the opaque area depicting the 95% confidence interval. The x-axis shows again the different levels of artificial delay. The first observation made in this figure is the fact that KOMon reported values are within a range of 0.2 us. Considering the fact that these values have been obtained by means of software based measurements, this difference falls well within the expected accuracy of our methodology. Additionally, none of the 95% confidence intervals exceeds a width of 0.03 us, further supporting the fact that KOMon exhibits a constant offset over the baseline values. Based on these values, we consider 5.2 us to be the overhead of KOMon reported values over the baseline. Note that the sample frequency as well as the number of packets in a single sample have no impact on the measured offset, since all packets have to traverse the part of the stack located between the measurement points of KOMon independently.

Estimation of Processing Time Distributions

Following the evaluation of its accuracy, we demonstrate the applicability of the proposed approach by using our KOMon tool to determine the processing time distribution of VNFs. To this end, we investigate the capabilities of estimating various processing time distributions. These include a negative exponential distribution, a uniform distribution that ranges from 0 to 2μ, as well as two normal distributions whose coefficients of variation are equal to 1 and 0.2, respectively. Furthermore, we vary the mean processing time of the VNF from 2 to 100 microseconds. Note that, as the normal distribution may assume negative values, we apply the left-sided pi-operator π_0 to the distribution, thereby accumulating negative probability mass on 0.

We first provide qualitative results of the evaluation in Figure 2.5. Each facet corresponds to one of the four processing time distributions and displays the empirical cumulative distribution function (ECDF) of the values that are reported by the KOMon tool and the VNF, respectively. While the data source is represented by the line type, differently colored curves denote different mean processing times μ as listed in the annotation in the first facet.

Two main observations can be made. First, the distributions that are obtained by means of the KOMon tool as well as the VNF are very similar regarding both their shape and values in all considered scenarios. Secondly, a constant offset between the curves can be observed. This offset has already been identified in the previous section and occurs due operations performed by the Kernel. By using the difference of means that is obtained during the calibration phase in conjunction with the KOMon-based values, we can correct this shift and estimate the mean processing time of the VNF as well as its coefficient of variation.

Results of this estimation are presented in Table 2.2. We show the mean values as well as the coefficient of variation of processing times that are reported by the VNF itself as well as the KOMon-based estimates. The latter are obtained by subtracting the calibration offset from KOMon-reported values. Finally, the table shows the theoretical coefficient of variation for each of the used distributions. In the presented case, this offset is equal to 5.2 microseconds.

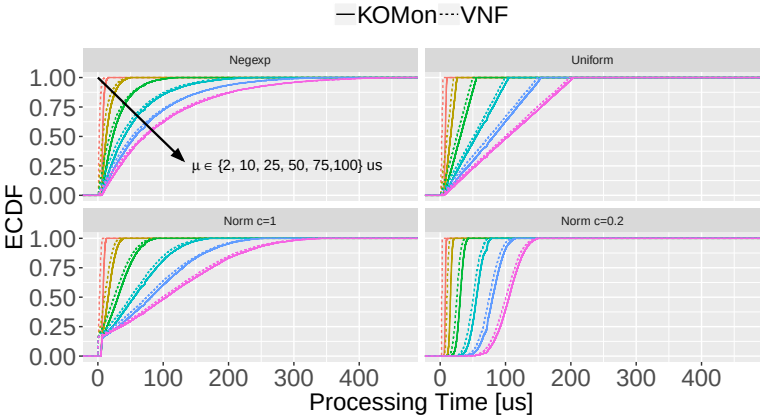


Figure 2.5: Empirical CDFs for different delays and different processing time distributions as sampled by the VNF as well as KOMon.

As evidenced by almost identical values in the context of all processing time distributions and average processing times, mean values can be estimated reliably. While this is also true for most scenarios in the case of the coefficient of variation, significant deviations are observed for $\mu = 2$ microseconds for all distributions. This behavior can be explained by the fact that in these scenarios, the offset is larger than the mean processing time and the small values are close to the resolution of software-based measurements. When it comes to the deviation between theoretical values and measurement values for the normal distribution with a coefficient of variation of 0.2, the inaccuracy can be explained by the inability of the network function to accurately reflect this small coefficient of variation. For this configuration, small technical inaccuracies due to, e.g., scheduling, reflect strongly in the resulting values. Hence, the deviation between theoretical and measurement values. However, when comparing VNF and KOMon-reported values this deviation disappears, and the results are in line with the remaining numbers.

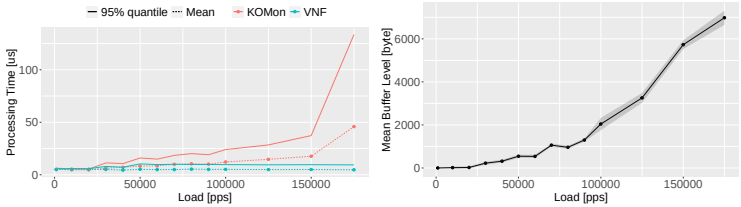
Table 2.2: Mean and coefficient of variation for different artificial delays and different processing time distributions.

Distribution	Delay μ [us]	Mean [us]		Coeff. of Variation		
		VNF	KOMon	VNF	KOMon	Theoretical
Negative Exponential	2	2.75	2.68	1.43	1.75	1
	50	50.91	50.59	0.99	0.99	1
	100	100.53	101.37	0.99	1.00	1
Uniform	2	2.73	2.61	1.39	0.87	0.58
	50	50.77	51.05	0.59	0.59	0.58
	100	100.69	101.02	0.58	0.58	0.58
Normal $c_v = 1$	2	2.89	2.79	1.59	1.49	0.80
	50	55.17	55.01	0.80	0.81	0.80
	100	109.74	109.57	0.80	0.80	0.80
Normal $c_v = 0.2$	2	2.73	2.62	1.22	0.71	0.2
	50	51.05	50.84	0.25	0.25	0.2
	100	100.88	101.05	0.22	0.22	0.2

To summarize, as has been observed in Figure 2.4, the monitoring values can be used to determine the packet processing time of a network function. The values obtained during the calibration show that the offset is statistically independent of the total processing time of the VNF and can be used to infer its real processing time.

Additionally, we have demonstrated that KOMon can be used to reliably reproduce the shape and estimate the mean as well as the coefficient of variation of a wide range of VNF processing time distributions. In particular, this estimation can be performed without VNF-reported values or access to the VNF code as only a system-specific constant, which can be obtained by means of a calibration run (cf. Figure 2.4), is needed.

Based on these measurements and observations, we are able to answer the question of measurement accuracy of our approach and conclude that the mechanism of in-stack monitoring can be leveraged to obtain highly accurate packet processing times while remaining network function agnostic (RQ2.1, RQ2.2).



(a) Comparison of VNF reported values and values observed using KOMon for different load levels.

(b) Socket buffer fill levels for different load levels.

Figure 2.6: KOMon evaluation at varying load levels.

2.3.2 Accuracy in High Load Scenarios

The second metric investigated here is the influence of different load levels on our tool's monitoring capabilities. Therefore, Figure 2.6a shows the mean as well as 95% quantiles of response time values monitored using KOMon as well as the processing time values reported by the VNF. Thereby, the x-axis shows the different load levels in packets per second. The y-axis presents the observed time in microseconds. Values obtained by KOMon are again reported in red while the blue values show baseline values reported by the VNF. In addition, the dotted lines represent the mean value while the solid curve depicts the 95% quantile. Similar to the scenario investigated before, the traffic generator produces a continuous stream of UDP datagrams of size 128 bytes. Both the VNF and KOMon sample batches of 10 consecutive packets in negative exponentially distributed intervals with a mean of 0.2 seconds. Instead of adding artificial delay, the VNF is configured to flood out packets as fast as possible.

Two observations can be made in this figure. On the one hand, it can be seen that the mean processing time reported by the VNF is nearly constant for all evaluated load levels between 1,000 and 175,000 packets per second, with 150,000 packets per second being the non-drop rate for the test setup as higher load leads to significant packet loss and is therefore not included in this evaluation.

This observation is quite intuitive as the VNF is processing packets as fast as possible for all load levels and, since no packet loss occurs even at 150,000 packets per second, it can do so even at high loads. The 95% quantile is, after a slight increase at around 30,000 packets per second, also mostly constant. On the other hand, the mean values obtained by KOMon show a continuous growth with the 95% quantile even exhibiting exponential growth behavior. This difference in observed response times is once again attributed to the operations taking place between the VNF and the monitoring points used by KOMon. Packets processed by the Linux Kernel network stack are in general queued two times on their way from the network interface card (NIC) to a user space application. Once at the NIC itself in a process called interrupt mitigation [101] that aims to decrease the overhead of sending an interrupt to the Kernel for every incoming packet, thus avoiding livelocks. Then, after having traversed most of the network stack, packets are queued a second time in the socket buffer of the socket opened by a user space application. This second buffer is located between the KOMon measurement points and the VNF and is filled to a different extent for different load scenarios. Figure 2.6b shows the mean buffer fill levels and 95% confidence intervals recorded during the load test presented in Figure 2.6a. The values have been obtained from `/proc/net/udp`.

The figure shows the different load levels along the x-axis and the mean buffer fill level along the y-axis. The opaque area depicts the 95% confidence interval. It can be seen that the buffer fill level develops similarly to the 95% quantile of the monitored values in Figure 2.6a, thereby exhibiting a correlation of 0.85.

In Summary, the information presented in Figure 2.6 shows that KOMon is able to take the queuing time of packets into account that not even the VNF itself can report and can thus be used to trigger measures like scale-up or scale-out ahead of time, thereby avoiding packet loss. This shows that KOMon can be used in live scenarios to monitor the current performance of network functions and is able to detect performance bottlenecks. Based on these measurements, and in combination with previous results, we can answer the question of applicability to both benchmarking and monitoring in live environments (RQ2.1).

2.3.3 Applications in Practice

In this section, we briefly discuss the packet matching problem and its impact on the proposed methodology and provide an outline regarding how it can be applied to different use cases.

The Packet Matching Problem

As described in Section 2.2, KOMon uses the hash value obtained from the packet payload for packet identification. This limits the functionality of the proof-of-concept in its current state to network functions that do not alter the payload in any way. In addition, in order to decrease the overhead induced by the monitoring logic as far as possible, we currently compare outgoing packets only to the oldest packet still in the incoming ring buffer, thus eliminating the possibility of per flow prioritization. The second issue of supporting non-FIFO network functions could be solved by comparing all packets currently in the system whenever a packet is sent out. This would lead to a slight increment in monitoring overhead while at the same time enabling more functionality. The initial problem, however, still remains. Network functions that alter payloads, drop, aggregate, or split up packets can currently not be monitored. This problem could be worked around by providing a VNF policy description ahead of time. The policy description can then be used to predict how the VNF is going to behave and KOMon can monitor for the expected result. This functionality, however, strongly depends on the type and functionality of the network function and is thus not part of the generic methodology proposed in this work.

Application during Network Function Development

Similar to the application of the approach in theoretical models, it can be applied during the development phase of a network function. Especially in the realm of continuous integration and continuous delivery (CI/CD) [102], automatable evaluations of application specific performance characteristics are crucial. To this end, KOMon can be seamlessly integrated into a build and evalu-

ation pipeline to perform automated performance evaluations of new versions of an application before its deployment, thereby ensuring no performance degradation, e.g., through added features. How the CI/CD paradigm can be applied to the networking realm is discussed in [96].

Application in Network Function Monitoring

Finally, the KOMon tool can be applied to monitor the performance of network functions in live environments by continuously gathering and evaluating samples. This allows for policing of the remaining resources of a network function instance. Upon reaching a certain threshold, scaling mechanisms can be triggered to provide additional resources before packet loss or response time explosion can occur. As was observed in Figure 2.6a, the 95% quantile could serve as a suitable indicator in most scenarios. In some cases other metrics extracted from the response time observations, e.g. entropy, might provide better results.

Application in Performance Modeling

One of the outcomes of the evaluation performed in Section 2.3.1 is that, for low loads, the reported monitoring values are very close to the real processing time of the VNF. In addition, we have shown that the offset included in the measurement is statistically independent of the magnitude of the processing times of the network function. This allows for KOMon to be used to determine the distribution of processing times as it is needed for theoretical performance models such as [19, 1, 2]. In addition, this offset can be eliminated from the values as it is possible to calibrate the system by performing measurements involving a network function of which the processing time is known. Hence, the offset that depends on the hardware or virtual environment can be calculated and taken into account by comparing the reported monitoring values to baseline measurements reported by a known network function. This exact procedure has been applied during the comparison of model and measurement results later in Section 2.4.5.

2.4 Discrete-time Modeling of Software-based Network Function KPIs

Based on the introduced mechanism for acquiring packet processing times and the shown ability to accurately replicate the processing time distributions of software-based network functions, we now apply the developed methodology to obtain the input parameters required for the development of an accurate system model. To this end, we provide an overview of the workflow and relevant system parameters when it comes to software packet processing. We continue with a detailed description of the model itself. Finally, the accuracy of the model is evaluated through a thorough set of measurements, thereby investigating the impact of different factors on the prediction accuracy of the model.

As a baseline for this work, we use VPP [48], an all-software framework that enables high performance packet processing by implementing several of the optimization and acceleration techniques introduced earlier in Section 2.1.1.

Of these software frameworks, numerous system implementations exist, and while some work recently started undertaking an experimental comparison of these implementations [49, 58, 59], to the best of our knowledge the model proposed in this thesis is the first system model that can explain and accurately predict the measurable system performance of such batch-based packet processors. Although a model for VNF response times is proposed in [71], its applicability is restricted to systems that process each packet individually. However, batching departs radically from such classic models where packets arrive independently and are independently buffered and treated. Indeed, batching not only correlates arrival and departure, but can also influence the average per-packet processing time. While queuing models that feature batched arrivals at the processing unit are not entirely new and have been used to better capture phenomena such as bursty TCP behavior [75, 76, 79, 80] or parallel processing in cloud environments [81], both the use case and the particular processing schemes differ significantly. Finally, modern systems for high-speed packet processing adopt several low-level techniques to speed up the processing time. The

efficiency of such techniques is severely affected by the experienced batch sizes. This in turn introduces a dependency between the processing efficiency, and hence the service time, and the batch size.

In the following, we present a brief overview of the processing workflow as well as parameters of such batch-based systems, before detailing the proposed model and finally presenting an evaluation of the prediction accuracy of said model.

2.4.1 Workflow and System Parameters

In a typical scenario, a software based packet processor binds one (or more) CPU(s) to one (or more) RSS queue(s). The CPU then enters a main loop in which the NIC is polled at every iteration. Consequently, the application collects a batch of packets and starts performing the packet processing. During this time new packets might have arrived at the NIC and are stored in the packet ring(s) until the next iteration. We now describe the most important metrics and parameters regarding the tuning and evaluation of our model and briefly highlight their respective interaction effects.

Batch Size

The first parameter is the batch size of the system, describing the number of packets that are processed in a single polling, processing and transmitting cycle. There is an intuitive correlation between the size of the batches and the input rate observed at the system. In the extreme case of input rate zero, the CPU keeps polling the NIC, but at every iteration it will find no packets, thus the average batch size is zero as well. The opposite extreme occurs if the input rate is higher than the processing rate. In this case, packets are written to the packet rings and the CPU cannot cope with the incoming rate. Hence, packets are overwritten and losses occur. Since the CPU will always find a full ring, it will intuitively pick up as many packets as possible, which is limited by the internal maximum batch size. Subsequently, the measured average batch size will

converge to the maximum batch size. The most interesting scenario happens in between the two extreme cases, where the input rate is between zero and the maximum sustainable rate of the software router. In this case, at every iteration the CPU will poll the device and find some amount of packets that will be processed. The number of packets observed at every polling event depends on the processing time of the batch collected in the previous cycle as well as the incoming packet rate and interarrival time distribution. In practice, this relation between the number of arrivals during a processing cycle in combination with the fact that the system becomes more efficient for larger batches, leads to an automatic feedback loop that helps maintain a stable equilibrium regarding the batch size. As already mentioned, smaller batches are processed less efficiently due to caching effects and the distribution of framework overhead in batch processing scenarios. Hence, small batches lead to an increase in batch size for the next cycle. At some point, the system reaches a state at which the processing efficiency has increased until the mean number of arrivals during a processing cycle equals the number of arrivals during the previous cycle. From that point, fluctuations in the batch size are mainly attributed to fluctuations in the arrival process. The impact of the maximum batch size is evaluated during the parameter study by investigating the impact of different values under varying circumstances in Section 2.4.5.

Processing Time

Next, we describe the processing time of batches. When an increase in the arrival rate occurs or the system has not yet reached equilibrium, the next batch will be larger, but the processing efficiency will be higher due to amortizing the fixed costs over a greater number of elements. Therefore, when a batch is larger, the per-packet processing time is smaller, impacting the size of the batch collected in the following polling cycle. Further, the processing time of batches is also affected by the network function that has to be applied to each packet. Some simple functions such as Ethernet switching intuitively require less processing than more complex functions such as IPSec or DPI. Finally, it needs to

be taken into account that packets within a single batch could potentially require differentiated processing and will hence exhibit varying processing times. In these scenarios, the processing of the batch is considered complete once the last packet of the batch has been processed. The processing time of different batch sizes is one of the input parameters of the proposed model. Section 2.4.5 provides insights into the tuning of the model based on observed values.

Packet Loss

Moving on, packet loss is one of the most crucial metrics when it comes to software routers as it relates directly to the QoS of a VNF. Typical software routers are able to sustain a rate of 12 to 14 millions packets per second (Mpps) [46, 49]. When sending 10Gbps traffic of minimum-sized packets on a wire, this translates into a rate of 14.88Mpps.[2] As mentioned before, losses occur if the packet arrival rate is higher than the packet processing rate as this leads to packets being overwritten in the receive side (RX) rings. However, losses may occur even at lower rates, because of the aforementioned dependencies between packet processing time, batch size and efficiency of the framework or in edge cases with arrival processes exhibiting large bursts. The packet loss probability is one of the output metrics generated by the proposed model and is evaluated for different scenarios during the parameter study.

Queue Size and Waiting Time

Finally, directly related to the packet loss probability as well as the sojourn time of the system is the queue size, meaning the number of available slots in the RX ring available for arriving packets. As mentioned before, loss occurs whenever a new packet arrives while all slots are occupied. In addition, the queue size impacts the waiting time experienced by arriving packets that are not collected in the next polling cycle. Instead, these packets have to wait for one or more full

[2] The minimum-size of a packet is 64 bytes, to which we should add the Ethernet header and trailers for 20 additional bytes.

processing times before they start processing. In this context, the queue size dictates the trade-off between potentially long waiting times and resilience against bursts of packets. The queue size distribution is one of the output parameters predicted by the proposed model. Similarly, the model is able to predict packet waiting times. However, due to technical limitations, we are not able to monitor waiting times in the technical system. Instead, we compare sojourn time distributions composed of the waiting time plus the processing time. Since the processing time is part of the model input, the only unknown factor remains the waiting time and the comparison is hence valid to establish the quality of our waiting time prediction.

Finally, it is important to note that, in the model, we represent the RX ring as a simple FIFO queue, meaning that, in lossy scenarios, the predicted waiting time will likely deviate from the measurement values. In a ring, old packets get overwritten and hence lost, while new arrivals are discarded instead of enqueued when it comes to the FIFO queue. The error is observed and detailed in Section 2.4.5. However, in lossy scenarios, the quantification of the waiting time distribution is only of limited value, since packet loss probability becomes the more meaningful parameter in that case.

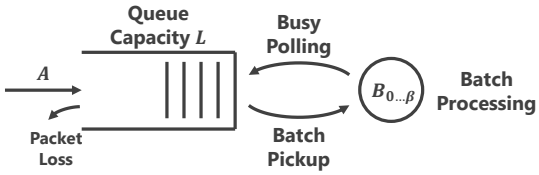


Figure 2.7: Schematic overview of model components and interactions.

2.4.2 System Model

In this section, we describe the queuing model that is used to evaluate the performance of batching-based packet processors. Figure 2.7 illustrates its main components, namely an arrival process with arbitrarily distributed packet interarrival times, a limited-capacity FIFO queue as well as a processing unit that regularly polls the queue, picks up limited-sized batches, and processes them with service times that depend on the batch size. We deliberately abstract the circular packet ring with a FIFO queue for the sake of tractability. Intuitively, this does not alter the system performance w.r.t. the amount of lost packets, only *which* packets are lost. Furthermore, experiments show that the model achieves a high level of accuracy despite this simplification (Section 2.4.5). In this section, after a brief overview of the system states that are captured by the model, we outline how to extract KPIs from the system steady state.

Discrete-Time Model

Before diving into the details we introduce definitions and notations as well as provide an outline of the model. Since the model developed in this work is based on a discrete-time approach, we need to discretize time into fixed intervals Δt. For the remainder of this work, we use $\Delta t = 10\,\text{ns}$, as it represents the most suitable resolution to describe the obtained measurement data. Note that the model resolution could be increased further by selecting a smaller Δt, at the expense of additional computational complexity.

Table 2.3: Notation.

Variable	Description
L	Queue capacity, equals 4096 if not stated otherwise.
β	Maximum batch size, equals 256 if not stated otherwise.
A, $a(k)$	Packet interarrival time.
B_i, $b_i(k)$	Service time of size i batches.
X_i, $x_i(k)$	Number of arrivals whose interarrival time is distributed according to A during an interval whose length is distributed according to $B_{min(i,\beta)}$ [103]. If τ is a constant, we implicitly apply the deterministic distribution with probability mass 1 at τ.
Q_n, $q_n(k)$	Queue size immediately before the n-th batch pick up.
Q, $q(k)$	Queue size at embedding times during steady state, before batch pick up. Hence, $Q = \lim_{n\to\infty} Q_n$.
Q_A, $q_A(k)$	Queue size at arrival times.
$\bar{Q}$, $\bar{q}(k)$	Queue size at random times.
V_n, $v_n(k)$	Batch size immediately before the n-th batch pick up.
V, $v(k)$	Batch size at embedding times during steady state. Hence, $V = \lim_{n\to\infty} V_n$.
S, $s(k)$	Batch service time at random times / among all batches.
p_{loss}	Packet loss probability.
W, $w(k)$	Waiting time.
D, $d(k)$	Processing time.

For the sake of readability, we provide an overview of the notation used in this manuscript in Table 2.3. The top half contains constants and random variables that constitute the model input, whereas outputs are listed in the bottom half. To disambiguate between random variables (RVs), distributions, and distribution functions, we use the following convention: uppercase letters such as A denote RVs, their distribution is represented by

$$a(k) =_{\text{def}} \mathrm{P}(A = k), \quad k \in [0, \infty)\,,$$

and the corresponding distribution function is defined as

$$A(k) =_{\text{def}} \mathrm{P}(A \leq k) = \sum_{i=-\infty}^{k} a(i), \quad k \in [0, \infty)\,.$$

In the proposed model, the system state at a given time is represented by the corresponding queue size Q_n at the time the n-th batch is polled from the NIC. As highlighted in Figure 2.8, all system events, such as packet arrivals as well as polling and batch processing, have a direct impact on the queue size. While each packet arrival leads to an increment of the queue size by one, polling by the processing unit decrements it by the number of packets that are picked up. The latter is limited by the maximum batch size which is denoted as β and the number of packets that reside in the queue at the time of the polling event. Finally, if an arriving packet finds the queue at its maximum capacity L, the packet is dropped. Hence, the *queue size distribution at the times of embedding during steady state* $q(k)$ can be used to derive relevant performance indicators of the modeled system such as the batch size distribution and the packet loss probability. Additionally, it is possible to derive the *queue size at arrival times* Q_A, which is required for computing the waiting and overall processing time distributions. To obtain event-independent system information, we also present a way of calculating the *queue size at random times* $\bar{Q}$.

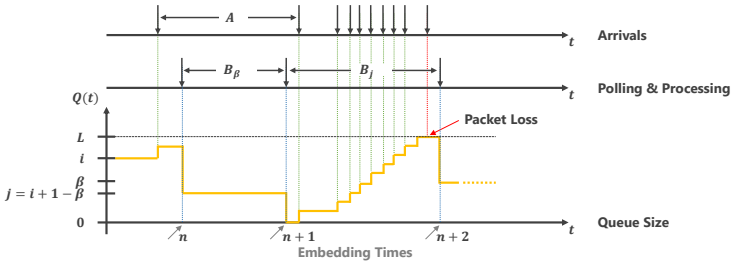


Figure 2.8: Exemplary state development of the model.

In order to derive the distribution of the queue size, we consider an embedded Markov chain whose embedding times are defined to be immediately before the busy polling events of the processing unit. Based on the queue size distribution $q_n(k)$ at these embedding times, we can derive the state probability distribution at consecutive embedding times by taking into account the current batch size and the number of arrival events during the corresponding service time. Finally, we use a fixed-point iteration in order to determine the queue size distribution at steady state $q(k)$. To this end, we leverage the recursive relationship in Equation 2.1 to compute the queue size distribution immediately before the $(n+1)$-st batch is picked up, based on the queue size distribution immediately before the n-th batch is picked up.

$$q_{n+1}(k) = \begin{cases} \sum_{i=0}^{L} q_n(i) x_i(k - (i - \min(i, \beta))) & \text{for } 0 \leq k < L, \\ \sum_{i=0}^{L} q_n(i) \sum_{j=0}^{\infty} x_i(L + j - (i - \min(i, \beta))) & \text{for } k = L, \\ 0 \text{ otherwise.} \end{cases} \tag{2.1}$$

The first case covers the probability to reach a state with a queue size that is below its capacity L. In order to calculate this probability, every possible previous value for the queue size i at the previous time of embedding is considered. Given i, the size of the batch that is processed between embeddings equals $\min(i, \beta)$ since the processing unit can pick up at most β packets. From this, we can derive the number of arrivals during the corresponding service time by means of X_i, which describes the number of arrivals with interarrival time A in an interval of length $B_{min(i,\beta)}$ [103]. Since embeddings are placed immediately before polling events, a queue size of k is reached when the number of arrivals during the service time is equal to the difference between k and $i - \min(i, \beta)$, the size of the queue immediately *after* the batch is picked up.

The special case of $k = L$ is calculated in an analogous fashion, but it is necessary to take into account packet loss, i.e., the arrival of packets beyond the queue capacity which also results in a queue size of L. Each number of lost packets is represented by the summation index j.

Under stationary conditions, the indexes n and $(n + 1)$ in (2.1) can be suppressed, i.e.,

$$q(k) = \lim_{n \to \infty} q_n(k).$$

Finally, we note a limitation of the outlined model, namely that the variability of the arrival process has to be reasonably smaller than batch service times. Otherwise, subsequent embedding times without new arrivals violate the assumption of independence between embedding times the embedded Markov chain is based on, and the number of arrivals is overestimated resulting in inaccuracies w.r.t. the KPIs. This is, however, a reasonable assumption taking into account the utilized traffic patterns.

Key Performance Indicators

Given the queue size distribution, the *batch size distribution* and *packet loss probability* can be derived according to Equations 2.2 and 2.3, respectively. While the former is representative of the system's efficiency, i.e., larger batches cor-

respond to lower per-packet processing times, a non-zero value of the latter is indicative of an under-dimensioned system. Furthermore, the queue size distribution at embedding times also allows calculating the queue size distributions at both arrival and random times. These statistics are representative of the system state encountered by arriving packets and a random observer, respectively. In particular, the former serves as the foundation for determining waiting and processing time distributions.

Batch Size Distribution If the queue size is lower than the maximum batch size β, the two are identical, i.e., the entire queue is emptied upon batch pickup, which is covered in the first case of (2.2). Queue sizes larger than β result in batch sizes of exactly β and are instead covered by the second case.

$$v(k) = \begin{cases} q(k) & 0 \leq k < \beta, \\ \sum_{i=\beta}^{\infty} q(i) & k = \beta, \\ 0 & \text{otherwise.} \end{cases} \tag{2.2}$$

Packet Loss Probability As noted in the description of (2.1), packet loss occurs when the number of arrivals during a service interval would lead to a queue size that exceeds the capacity L. Hence, we can describe the packet loss probability as the ratio of the expected number of arrivals beyond this threshold N_{Lost} and the expected total number of arrivals N_{Arrivals}:

$$\begin{aligned} p_{loss} &= \frac{\mathrm{E}\left[N_{\text{Lost}}\right]}{\mathrm{E}\left[N_{\text{Arrivals}}\right]} = \\ &= \frac{\sum_{i=0}^{L} q(i) \sum_{j=0}^{\infty} (j \cdot x_i(L + \min(i, \beta) - i + j))}{\sum_{i=0}^{L} q(i) \sum_{j=0}^{\infty} (j \cdot x_i(j))} \end{aligned} \tag{2.3}$$

Similarly to (2.1), we consider all possible queue sizes i and use the corresponding probability $q(i)$ as a weighting factor. For each number of lost packets j, we calculate the probability for the arrival of $(L + \min(i, \beta) - i + j)$ packets that are required for filling and exceeding the queue. For the expected total number of arrivals, we proceed in an analogous fashion but do not have to shift the distribution of the number of arrivals.

Queue Size Distribution at Arrival Times While the queue size distribution at the times of embedding allows us to compute the batch sizes that are picked up during busy polling events, we need a shift of perspective in order to determine the waiting time distribution of packets. In particular, the waiting time of a packet starts as soon as it arrives in the system and the time it spends in the system depends on its position in the queue.

Equation 2.4 shows how the queue size distribution as seen by arrivals, $q_A(k)$, can be computed. Based on the queue size at embedding times, it is possible to determine the resulting batch size as well as the distribution of the number of arrival events during the corresponding batch service interval. The probability of an arrival finding a certain queue size can then be computed via the ratio of the number of arrivals that find the queue at that specific level and the total number of arrivals.

$$q_A(k) = \frac{\sum_{i=0}^{L} q(i) \sum_{j=0}^{\infty} x_{b_{\min(i,\beta)},a}(j) \sum_{m=i-\min(i,\beta)}^{i-\min(i,\beta)+j-1} \mathbf{1}_{\{\min(m,L)=k\}}}{\sum_{i=0}^{L} q(i) \sum_{j=0}^{\infty} x_{b_{\min(i,\beta)},a}(j) \cdot j} \tag{2.4}$$

Waiting and Processing Time Distributions

Given the queue size distribution at arrival times, we can derive the distribution of the waiting time by decomposing it into two parts. First, the time between a packet's arrival and the next batch pick-up event. Second, zero or more service times of size-β batches, depending on the packet's position in the queue.

We obtain the first component by considering the recurrence time of the overall batch service time R_S, i.e., the time a random observer has to wait to encounter a batch pick-up event. To this end, we derive the distribution of observed batch service times $s(k)$ by weighting the batch size-dependent service times with the occurrence probabilities of the corresponding batch sizes as follows.

$$s(k) = \sum_{i=0}^{\beta} v(i) \cdot b_i(k). \tag{2.5}$$

For the second component, we leverage the fact that each possible queue size Q_A that can be encountered by an arriving packet can be mapped to a specific number of full batches that are serviced before it.

Subsequently, the distribution of the corresponding duration for servicing the respective number of batches can be obtained by convolving b_β with itself. In terms of random variables, the waiting time can therefore be expressed as

$$W = R_S + \sum_{i=1}^{\left\lfloor \frac{Q_A}{\beta} \right\rfloor} B_\beta, \tag{2.6}$$

where R_S denotes the recurrence time of $s(k)$.

Finally, the total time a packet spends in the system can be calculated as the sum of the waiting time W and the service time S:

$$D = W + S. \tag{2.7}$$

Queue Size Distribution at Random Times

While the perspective of individual packets that arrive at the system can be useful when calculating performance indicators such as the waiting time, the queue size distribution at random times provides generic steady-state system information that is independent of specific events.

In order to determine the queue size distribution at random times, we reason about the possible development of the queue between two times of embedding. We illustrate the development between the n-th and $(n+1)$-st embedding time in Figure 2.9. Let the queue size equal Q_{init} immediately before a batch of packets is picked up for processing. Depending on Q_{init}, the resulting batch size of the t-th batch k_t is between 0 and β. During the corresponding batch service time B_{k_t}, a number of X_{k_t} arrival events take place and sequentially increase the queue size. The relative time the queue spends on each level is proportional to the interarrival time A for most packets. Exceptions include the first and last level since they are interrupted by the current and subsequent batch pick-up events, respectively. Hence, their duration is proportional to the recurrence time of the interarrival time, R_A. Other exceptions include the case of no arrivals and arrivals that find the queue fully occupied. In those cases, the queue spends the entire time on the same level or a prolonged time on the maximum level, respectively. The recurrence time R_A [104] of a RV A can be computed as

$$r_A(t) = \frac{1}{E[A]} \cdot (1 - A(t)). \tag{2.8}$$

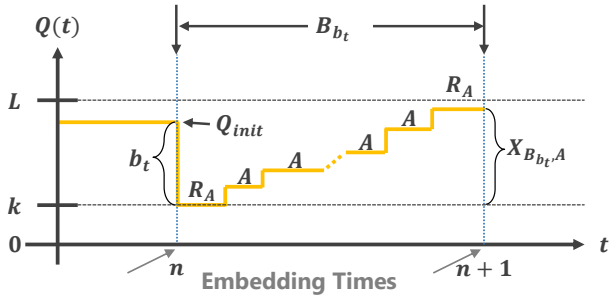


Figure 2.9: Exemplary queue size development between two embedding times and the contributing interarrival as well as recurrence times.

We use Equation 2.9 to calculate $\bar{q}(k)$, i.e., the probability of observing a queue size of $0 \leq k < L$, as follows. $\mathbf{1}_{\{condition\}}$ thereby represents the indicator function, assuming 1 if the condition is true, 0 otherwise. The first term considers reaching queue size k immediately after the batch pick-up. In this case, the queue size either remains equal to k the entire time if no additional arrivals take place during the batch service time, or it stays at size k for a time proportional to the recurrence time if there are additional arrivals. The second term deals with cases in which the queue size is reduced to a value lower than k when a batch is picked up, and a queue size of k is reached either as an intermediate state or as the final state prior to the next pick-up event. Depending on this, the time spent at a queue size of k is proportional to either the interarrival time A or its recurrence time R_A. Note the two indices in $x_{j,A}$ as opposed to the single index defined in Table 2.3. $x_{j,A}$ describes the more general case without the limitation of j to β, and thus represents the distribution of the number of arrivals with interarrival time A that occur in an interval of length j.

$$
\begin{aligned}
\bar{q}(k) = & \sum_{i=0}^{L} q(i) \cdot \mathbf{1}_{\{i-min(i,\beta)=k\}} \cdot \sum_{j=0}^{\infty} b_{min(i,\beta)}(j) \\
& \cdot \left(x_{j,A}(0) + \sum_{l=1}^{\infty} x_{j,A}(l) \cdot \frac{\mathrm{E}\left[R_A\right]}{(l-1)\,\mathrm{E}\left[A\right] + 2\,\mathrm{E}\left[R_A\right]} \right) \\
& + \sum_{i=0}^{L} q(i) \cdot \mathbf{1}_{\{i-min(i,\beta)<k\}} \cdot \sum_{j=0}^{\infty} b_{min(i,\beta)}(j) \\
& \cdot \left(\sum_{l=k-i+1}^{\infty} x_{j,A}(l) \cdot \frac{\mathrm{E}\left[A\right]}{(l-1)\,\mathrm{E}\left[A\right] + 2\,\mathrm{E}\left[R_a\right]} \right. \\
& \left. + \; x_{j,A}(k-i) \cdot \frac{\mathrm{E}\left[R_A\right]}{(l-1)\,\mathrm{E}\left[A\right] + 2\,\mathrm{E}\left[R_A\right]} \right) \\
& \text{for } 0 \leq k < L.
\end{aligned}
\tag{2.9}
$$

For the special case of $k = L$, we need to account for the fact that the proportion of time the queue spends at its maximum occupancy may be larger due to the occurrence of packet loss. We derive the corresponding term in Equation 2.10:

$$\bar{q}(L) = \sum_{i=0}^{L} q(i) \cdot \sum_{j=0}^{\infty} b_{min(i,\beta)}(j) \sum_{l=L-i+min(i,\beta)}^{\infty} x_{j,a}(l) \cdot \frac{\mathrm{E}[R_A] + (L-l-1)\,\mathrm{E}[A]}{(l-1)\,\mathrm{E}[A] + 2\,\mathrm{E}[R_A]}. \tag{2.10}$$

2.4.3 Experimental Setup

To validate our model, we instrument a testbed operating a real NFV software router following the IETF benchmarking guidelines [105]. This section describes our hardware and software setup as well as the scenarios we examine to assess the accuracy of our model. We point out that, to assist reproducibility of our work, all the experimental data we collect is available at [106].

Hardware Setup

The testbed, as depicted in Figure 2.10, consists of two COTS Desktop PCs, equipped with two Intel 82599ES dual port NICs operating at 10 Gbps. Each node has an i7-2600 processor, running at 3.40 GHz. Each processor has 3 levels of cache hierarchy, ranging from 32 KB for the L1 to 8 MB for the L3. Both machines are equipped with 16 GB of memory.

One node is used as the Device Under Test (DUT) and the other for traffic generation (TX) and reception (RX). The DUT receives traffic from one input linecard, performs the packet processing, and then proceeds with the forwarding to the designated output port. We conduct our measurements at the TX and RX side in order to assess the packet ingress and egress rate as well as packet loss. Additionally, we measure directly within the DUT in order to obtain batch sizes

and per-batch processing time. In order to ensure reproducibility and eliminate operating system scheduling, we run the DUT on a single CPU core attached to a single RSS queue, as is typically done in stress-test conditions.

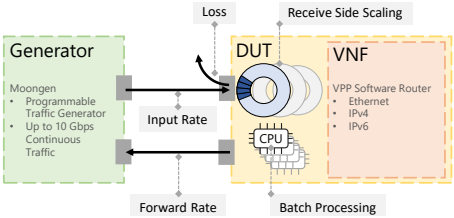


Figure 2.10: Schematic overview of the hardware testbed used to obtain VNF measurements.

2.4.4 Software Setup

DUT To validate the model, we select a state-of-the-art NFV software stack that employs batched processing. In particular, we conduct experiments with the Vector Packet Processor (VPP) [48]. VPP implements VNFs as software components (*nodes*) that can be linked together in a specific configuration (*forwarding graph*). A specific input node (*dpdk-input*) polls the line-card for new packets, grabbing a batch (*vector*) from the ring for processing. Notice from Table 2.4 that VPP compute-batches may aggregate several DPDK I/O-batches, as the maximum VPP batch size is larger than DPDK's. VPP then processes all packets in the vector node-by-node instead of traversing the graph packet-by-packet. Hence, in addition to sharing the framework overhead over the batch, only the first packet triggers fetching of processing code in the L1-instruction cache of the CPU, whereas processing of subsequent packets benefits from L1-instruction cache hits [55, 107].

Table 2.4: Experimental configuration parameters.

	Parameter	Value
HW	NIC	Intel 82599ES dual-port 10 Gbps
	CPU	i7-2600@3.4 GHz
	Caches L1/L2/L3	32 KB/256 KB/8 MB
DUT	Software router	VPP 19.04
	Number of CPU cores	1
	Number of RSS queues	1
	Memory allocated	4 GB
	Size of input queue (pkts)	$L = \{1024, 2048, 4096\}$
	Max DPDK batch size (pkts)	32
	Max VPP batch size (pkts)	$\beta = \{64, 256, 512, 1024\}$
TX/RX	Traffic Generator	MoonGen
	Rate span [min:inc:max]	$[0.5 : 0.5 : 10]$ Gbps
	Hi/Lo rates	10 Gbps / 2.5 Gbps
	Packet sizes	$\{64, 128, 256, 512, 1024\}$ B
	Arrival rate process	Constant bit-rate (CBR)
	Data points per configuration	138k
	Functions	{ XC, Eth, IPv4, IPv6 }
	Scenarios	Homogeneous vs Heterogeneous

Also notice that this process naturally introduces branches, as packets may trigger different functions implemented in different nodes of the forwarding graph. This requires splitting the original heterogeneous batch into smaller homogeneous batches for the subsequent nodes. This is expected to change the operational point of the NFV router, as not only the splitting process incurs an additional overhead, but also since the framework overhead is now shared over a smaller batch, and the code heterogeneity increases the L1-instruction cache miss rate. It is thus important to assess experimental performance under realistic scenarios involving multiple functions. Furthermore, we investigate the impact of varying maximum batch sizes as well as the size of the RX ring for different load levels.

TX/RX For traffic generation and reception, we use MoonGen [108], a state-of-the-art programmable tool capable of sustaining 10 Gbps line-rate. MoonGen also provides APIs to perform basic measurements at the TX/RX side. For example, it is possible to access the NIC's hardware counters to precisely measure the number of packets transmitted and received, which allows to derive the experimental forwarding and loss rates for comparison with the model.

Typically, a single DUT thread on a single RSS queue under commonly considered NFV workloads is able to sustain a rate of 12–14 Mpps [46, 49]. As such, when sending 10 Gbps worth of traffic at minimum-sized 64 Bytes packets on a wire, corresponding to a rate of 14.88 Mpps, we expect the system to be in a lossy regime. As such, we assess the system performance for different rates, ranging from 0.5 Gbps to 10 Gbps with a step increment of 0.5 Gbps. For the sake of illustration, we also consider two exemplary operational points, representing a high-rate (10 Gbps) and a low-rate (2.5 Gbps) regime. Additionally, we assess the system performance for differently sized packets, ranging from 64 Bytes to 1024 Bytes.

Scenarios

We consider two VNF cases, in which the router is stressed with either homogeneous traffic that triggers the same function or heterogeneous traffic that activates a mixture of functions. We select popular functions in the NFV ecosystem that allow us to focus on different components of the framework. We use the simplest function to investigate I/O batching and introduce different types of lookup and data structures to provide instances of compute-batching with different complexity.

Homogeneous Cross-Connect Function In this scenario a single VNF, usually referred to as cross-connection (XC), is applied to all packets, representing the baseline of homogeneous functions in an NFV router. In this case, the VPP DUT is configured to take all the packets from one input interface and immediately forward them to a fixed output interface. Notice that for the XC VNF, no

computation is needed on the headers of the transferred packets since the DUT simply moves batches from the input to the output NIC. Therefore, this scenario helps assess whether the model faithfully reproduces the impact of I/O-batching.

We generate our workload using a MoonGen script that sends a stream of packets at a fixed rate, namely copies of a templated UDP packet. Notice that for such a simple VNF, the type of traffic does not affect the processing time. Since neither processing nor branching happens, XC performance represents an upper bound for the performance of the NFV router.

Heterogeneous Eth/IPv4/IPv6 Functions As pointed out in [59], as network traffic is heterogeneous, NFV routers need to handle a mixture of different functions. We therefore consider the case of three different functions that operate on the same traffic batch. Specifically, we investigate three functions with different sizes of inputs (48, 32, and 128 bits), lookup types (exact vs longest-prefix match), and data structures (hash tables vs tries). In particular, we generate traffic that triggers the following operations, in increasing order of complexity: (i) a 48 bit exact-match Ethernet lookup, (ii) a 32 bit IPv4 longest-prefix match lookup using a trie structure, and (iii) a 128 bit IPv6 longest-prefix match lookup that performs a lookup over multiple hash tables for different netmask lengths.

For the sake of simplicity, our experiments are performed with an even split of the functions, i.e., each of the above traffic types consume $\frac{1}{3}$ of the bandwidth, so that each function activates with probability $\frac{1}{3}$, resulting in different function breakdowns across batches. We point out that more complex scenarios (e.g., featuring an uneven split, a larger set of functions, or longer chains) are within the capability of the model, but are out of the scope of the research presented here.

2.4.5 Modeling vs Experimental Results

Before we validate our model via experimental results from the homogeneous and the heterogeneous traffic scenarios, we discuss several options that are available for tuning the model inputs. These options represent different trade-offs in terms of the resulting prediction accuracy, the model's general applicability, as well as the amount of measurements that are required prior to its application. Using appropriate model settings, we then compare model-based performance predictions with our measurements. In particular, we focus on the batch size and the waiting time.

Model Tuning Options

As detailed in Section 2.4.2, the model input consists of the queue capacity L, the maximum batch size β, the distribution of packet interarrival times $a(k)$, and the size-dependent distributions of batch processing times $b_i(k)$. For the purpose of model tuning, we fix the values for L and β at 4096 and 256, respectively.

While the mean packet interarrival time $\mathrm{E}[A]$ can be determined from the applied rate, our model provides a degree of freedom by allowing to set an *arbitrary distribution* to reflect aspects like the traffic's burstiness. To this end, we consider a total of four distributions that have varying degrees of variation. In particular, these include (i) the Poisson distribution whose coefficient of variation equals $\frac{1}{\sqrt{\mathrm{E}[A]}}$, (ii) the geometric distribution with a coefficient of variation of $\frac{1}{\sqrt{q}}$ with $p = 1 - q$ being the success probability, and (iii)-(iv) negative binomial distributions whose parameters are set to achieve coefficients of variation equal to 0.5 and 2, respectively.

Batch-dependent Processing Time Distribution Furthermore, we use our measurements to obtain $\mathrm{E}[B_i]$, the mean size-dependent batch service times. Similarly to the packet interarrival time, we can use different distributions to model the behavior of the processor. However, all conducted measurements yielded a very low degree of variation when considering a particular combi-

nation of applied rate and the corresponding per-size batch service time. Hence, we use Poisson distributions for the service time as our experiments have shown that the low variability provides a close fit to the measurement data.

Interpolating Processing Times Additionally, the model might require service time distributions for batch sizes that did not occur in the measurements. In order to provide these missing distributions, the mean service times for the remaining values of the batch size are required. We obtain these by means of linear fitting. In particular, we interpolate the missing mean service times based on the means of observed values. The Poisson distributions for the service time are then generated with measurement-based means where available and with fitted means otherwise. This fitting procedure can be done either globally or on a per-rate basis. This choice represents at trade-off between the overhead for per-rate measurements of the service time, risking overfitting the model to a particular scenario, and a possible improvement w.r.t. the resulting accuracy. Again, based on our experiments, we use a single, global linear fit in order to generate mean processing time values for batch sizes not observed during measurement runs. The remainder of the presented results are based on this method of interpolating the missing data.

Arrival Process Distribution In contrast to the fitting strategy, the chosen distribution of the arrival process does not have a significant impact on the *mean batch size* returned by the model, which we can capture with the relative error (RE) of the normalized difference of means which is defined as

$$\mathrm{RE}(P, Q) = \frac{|\mathrm{E}[P] - \mathrm{E}[Q]|}{\mathrm{E}[P]}.$$

Therefore, we extend our evaluation and compare the *batch size distributions* that are returned for different arrival processes. To this end, we compare the batch size distribution returned by the model under different arrival processes to distributions observed in the testbed. Note that we do not modify the arrival pro-

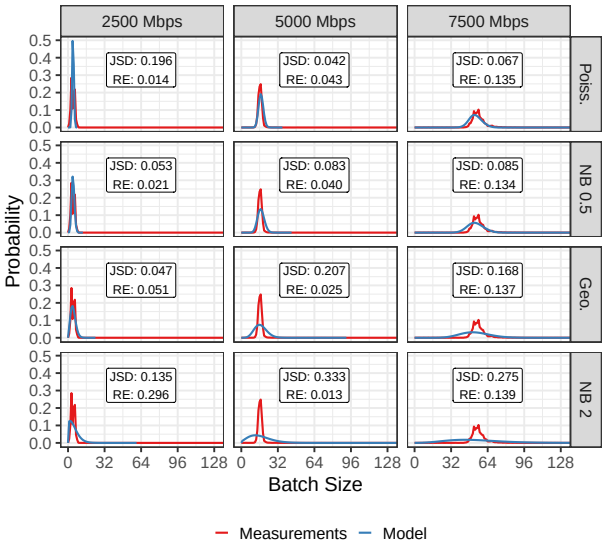


Figure 2.11: Batch size distributions for different rates and arrival processes. Cross-connect scenario with $\beta = 256$, $L = 4096$, and 64 B packets.

cess in the measurement setup, but are interested in identifying an appropriate interarrival time distribution for the model. We quantify the difference between the distribution that is returned by the measurements and the model by means of the Jensen-Shannon divergence (JSD) which is symmetric and bounded and allows to equally weight differences among two distributions $p(k)$ and $q(k)$ over their full support, and is defined as

$$\mathrm{JSD}(P,Q) = \sum_{k=0}^{\infty} \left(\frac{1}{2} p(k) \ln \frac{p(k)}{\frac{1}{2}p(k) + \frac{1}{2}q(k)} + \frac{1}{2} q(k) \ln \frac{q(k)}{\frac{1}{2}q(k) + \frac{1}{2}p(k)} \right).$$

For three exemplary rates that represent a low, a medium, and a high load as well as our four arrival distributions in increasing order of coefficient of variation, Figure 2.11 displays the batch size distribution obtained by means of measurements and our model. Given the batch size on the x-axes, the y-axes represent the corresponding probability while annotations provide the JSD and RE values. Note that all rows show the same values in case of the measurement-based distributions, since we only vary the interarrival time distribution used in the model.

When inspecting the distributions obtained by the measurements, we can observe that there is usually one peak around which the main portion of the probability mass is centered. This can be explained by the fact that there is an equilibrium between the per-packet service time that is achieved in the context of a particular batch size and the mean packet interarrival time. Hence, the number of arrivals during the service time of a batch is nearly constant. In the case of higher rates, shorter interarrival times lead to larger mean batch sizes which, in turn, allow for larger fluctuations in terms of the number of arrivals during the corresponding service time.

When comparing the subfigures column-wise, we observe that while these peaks are also reconstructed by all model variants, their dispersion increases significantly with the coefficient of variation of the chosen arrival distribution. Similarly to the previous argument, the higher variance of packet interarrivals leads to a wider range in terms of the number of arrivals during a service period. Finally, the best match regarding both the shape of the resulting distributions and the achieved JSD measure is achieved when using arrivals that follow a Poisson distribution.[3] This is also in line with the settings of the MoonGen traffic generator that is set to send packets at a constant rate. Since it is a software-based generator, minor fluctuations of the corresponding sub-microsecond interarrival times are to be expected. Therefore, we use interarrival times that follow a Poisson distribution for the remainder of this work.

[3]Note that arrivals do not follow a Poisson process, but exhibit interarrival times according to a Poisson distribution.

Table 2.5: Packet loss probability for different rates. Cross-connect scenario with $\beta = 256$, $L = 4096$, and 64 B packets.

Rate [Mbps]	8000	8500	9000	9500	10000
Measurements	0.97%	6.69%	11.58%	16.44%	20.17%
Model	1.09%	6.78%	11.56%	16.30%	20.13%

As already noted, the mean batch size takes on a constant value of 256 for rates of 8 Gbps and above. In these high-load regimes, packet loss begins to occur since the number of arrivals during the batch service time exceeds 256 and the queue fills up steadily. In Table 2.5, the actual packet loss that is reported in the measurements is compared to the model's predictions. Rates below 8 Gbps are omitted since they are equal to 0 in both cases. For the remaining rates, the model accurately predicts the occurrence and quantity of packet loss which increases linearly with the applied load.

In summary, our model achieves a very high accuracy for both batch size and packet loss in the cross-connect scenario, faithfully modeling I/O batching over a wide range of arrival rates, including overload scenarios that result in packet loss, thereby contributing to the answer of RQ2.3.

Prediction of Key Performance Indicators

Having demonstrated the model accuracy in the context of the simple cross-connect scenario and having identified appropriate model settings, we present the results of evaluations with the more complex scenario featuring mixed traffic in this section. Additionally, we investigate the impact of parameters such as the maximum batch size, the queue size, and the packet size on the system behavior as well as the accuracy of the model.

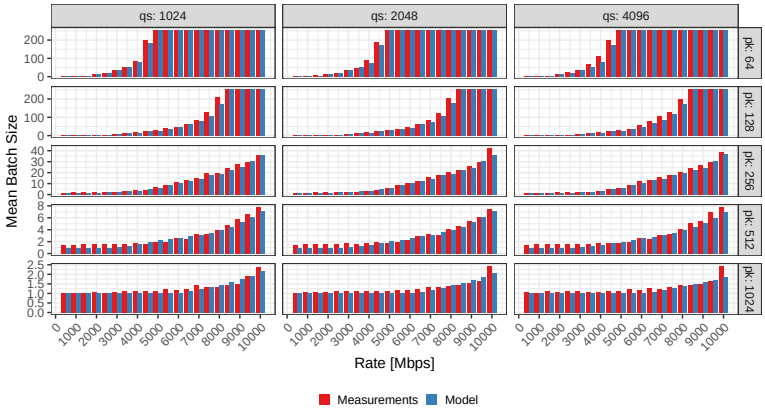


Figure 2.12: Average batch size obtained via model and measurements given different packet (pk) and queue sizes (qs). Maximum batch size $\beta = 256$ and mixed traffic.

Batch Size Distribution Since the overhead for processing a batch of packets is shared between packets within a batch, the batch size distribution constitutes an important measure of system efficiency. We present a comparison between the mean batch sizes reported in our measurements and the predictions of our model in Figure 2.12. Each subplot depicts a combination of queue and packet size and displays the applied rate on the x-axis and the corresponding mean batch size on the y-axis.

First, we can extract insights regarding the system behavior. We observe that the maximum batch size is attained earlier in the context of the more complex mixed traffic scenario than in the cross-connect scenario. While the top right subplot of Figure 2.12 shows that this already happens at a rate of 5 Gbps with mixed traffic, a queue size of 4096, and 64 B packets, the maximum batch size is reached starting at a rate of 8 Gbps when using the same parameters in the cross-connect case.

Furthermore, increasing the packet size leads to a decrease of batch sizes as evidenced by the development of batch sizes along vertical sequences of subplots. This can be explained by the fact that processing happens per header rather than per byte and therefore an increase in packet size results in a decrease of the packet-rate at the same bitrate. From the vertical sequences of subplots, we can derive that the queue size does not have a significant impact on the batch size. To explain this, we can consider the two extreme operational regimes the system can be in. If it is in the loss-free regime, the queue is emptied on each batch pickup event and never runs full. In conditions with packet loss, the queue tends to fill up regardless of its size, leading to exclusively full batches. The behavior in between these two scenarios is, in general, depending on the arrival process and, more specifically, its burstiness. Hence, while the maximum queue size does not have an immediate impact on batch sizes, it should not be ignored since it does affect waiting times and can allow the system to withstand packet bursts.

Finally, comparing measurement values with model-based estimates demonstrates that despite the increased complexity of the scenario, the model is still capable of reliably predicting the mean batch size. In 90% of scenarios, the difference w.r.t. mean batch size is 2 or less. Nevertheless, some outliers with larger differences are present. These tend to occur on two occasions. First, during the transition from the loss-free regime to the lossy regime. Second, when the applied rate equals the maximum rate of the link. Both constellations represent conditions with an increased sensitivity where deviations are amplified.

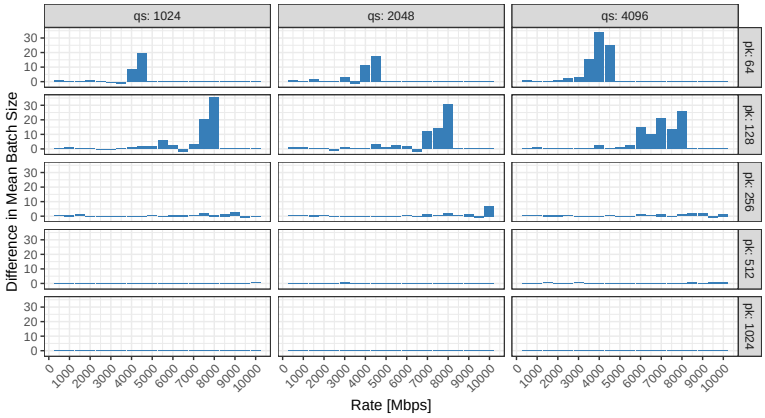


Figure 2.13: Difference in mean batch sizes obtained via model and measurements given different packet (pk) and queue sizes (qs). Maximum batch size $\beta = 256$ and mixed traffic. Values greater than 0 correspond to the mean batch size in the measurements being larger than the one returned by the model.

The transition between loss-free and lossy regimes can be observed in Figure 2.13, which shows the difference between the experimentally measured batch sizes and the predictions of the model. Considering the first row (obtained with mixed traffic of 64 B packets) we observe that at high-rate, the difference is zero: the model correctly predicts the saturation of the system which will always retrieve batches of the maximum size β. At low rate, the difference between the model and the measurements is very low, and it increases as we approach the state change between a loss-less and a lossy regime. When the system load reaches this state change (between 3.5 Gbps and 5.0 Gbps) we observe that the model underestimates the actual size of the batches. This is due to non-linear effects introduced by the implementation of the VNF router, as the program tries to privilege larger batches in order to minimize the overhead of the framework, thus causing a discrepancy of up to 30 packets for the batch size.

Similarly, this also explains the behavior observed in the second row of Figure 2.13. Interestingly, we observe that when the packet size is 64 B (first row), the interval of the state change is [3.5, 5.0] Gbps, which translates into an interval of [5.2, 7.4] Mpps (millions of packets per second). When the packet size is 128 B, the state-change interval is [6.0, 8.5] Gbps, which translates into an interval of [5.1, 7.2] Mpps. Therefore, although the second interval is larger in bitrate, it is comparable in terms of processed packets per second. Finally, as the scenarios with packet size greater than 256 B never show a change of regime, we do not observe a significant difference between the measurements and the model.

In summary, the model accurately captures the mean batch size as well as the batch size distribution, even when the scenario complexity is increased by changing the VNF behavior, traffic mix, or parameters such as the packet size, queue size, and maximum batch size. The compatibility with these scenarios is maintained without modifications to the model, highlighting its general applicability.

Waiting and Processing Time Distributions While the batch size can serve as an efficiency indicator, large batches can also adversely affect the waiting and processing time of packets. Hence, these metrics should be considered when evaluating the performance of VNFs. With our model, we can derive the waiting and processing time distributions, and we validate the results in this section.

Since performing per-packet delay measurements in the DUT would interfere with VNF performance - especially at high packet rates - it is not a feasible strategy for obtaining ground-truth latency data. Instead, we measure the total per-packet latency between the egress and ingress of the MoonGen traffic generator and compare it to the processing time determined via our model. Due to this measurement setup, we expect two main sources of mismatch between the experimental results and our model. Whereas the model targets internal DUT processing, measurements are taken externally. Hence, the measurements include DUT processing as well as additional delays induced by the traffic generator and other overheads. Similarly, propagation and transmission delay between the generator and the DUT are not explicitly accounted for in the model.

As such, we expect the model to underestimate delay-related KPIs when compared to measurement values. However, due to the nature of the aforementioned overhead, this difference should result in a constant, scenario-specific *fixed delay offset* which can be addressed with appropriate calibration. In our case, this fixed delay offset encompasses the MoonGen processing time for packet handling, the round-trip propagation, and transmission delays on wire. We quantify this offset by means of a simple cross-connect setup in which the DUT simply executes DPDK L2 forwarding to minimize processing. In this scenario, we once again use MoonGen to generate a continuous stream of packets, that are forwarded back to the source by the L2 forwarding VNF. In our specific testbed configuration, this overhead amounts to roughly 5 microseconds and is later used in Figure 2.15 to present adjusted model predictions.

For different rates on the x-axis, Figure 2.14 shows a comparison of the mean latency in microseconds as obtained from measurements and our model. Horizontally and vertically arranged subplots illustrate the effects of changes to the queue size and packet size, respectively. We limit the y-axis to a maximum of 10 μs in order to show details during the loss-free operational regime. As soon as the load increases and the system transitions into the lossy regime, the latency increases significantly due to congestion at the queue.

We make three main observations. First, for all shown scenarios, both the measurement- and the model-based values follow the same trend, i.e., latency increases happen at the same locations and with a similar slope. Furthermore, the offset between the two curves remains in a narrow range around 5 μs. Second, there is a clear effect of the packet size on the waiting time. Following subfigures along the vertical axis, we can observe that the latency starts increasing earlier in the case of small packets. This is in line with previous observations about higher packet rates when using smaller packets at the same bitrate. Third, almost no difference is observed regarding the mean latency in scenarios that differ only in the queue size. This effect is caused by limiting the y-axis and showing only the loss-free portion of the scenarios. In those, no congestion at the queues takes place and the queue is emptied on each batch pick-up.

Beyond these rates, the queue size does play a role and limits the maximum waiting time, i.e., rather than having to wait longer, packets are dropped in the case of a smaller queue size.

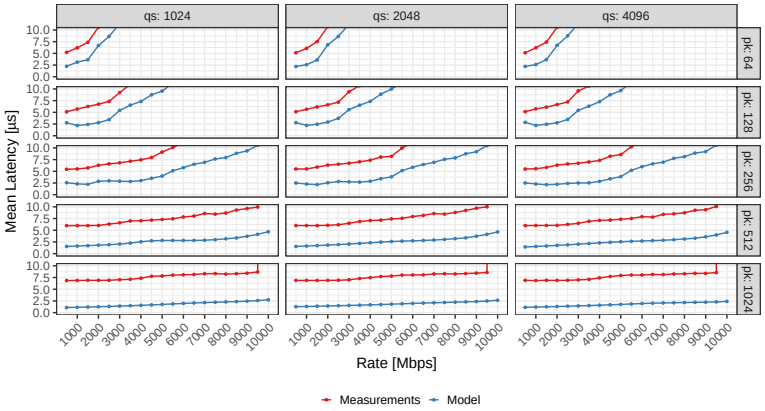
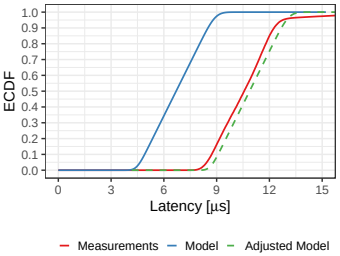
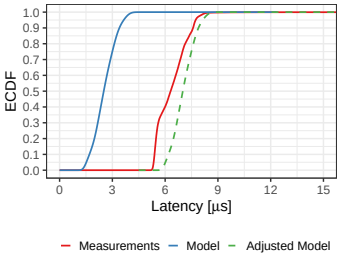


Figure 2.14: Mean latency measured in the technical system and reported by the model under different packet (pk) and queue sizes (qs). Maximum batch size $\beta = 256$ and mixed traffic.

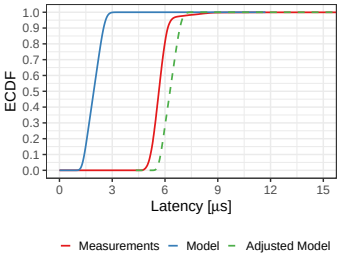
Despite the offset regarding the mean latency, we investigate the latency distribution to check whether the model can faithfully reproduce the general system behavior, e.g., regarding the shape of the distribution. To this end, we present cumulative distribution functions for three different parameter combinations in Figure 2.15. Each subfigure corresponds to one parameter combination, and we vary the traffic conditions, queue size, packet size, and applied packet rate. The measurement values are shown in red, whereas the green and blue lines indicate the model values with and without calibration, respectively. In particular, the adjusted model curves are obtained by using the mean offset from the calibration measurements to shift the original distributions that are returned by the model.



(a) Mixed traffic, $\beta = 256$, 64 B packets, $L = 4096$, 2000 Mbps.



(b) Mixed traffic, $\beta = 256$, 256 B packets, $L = 2048$, 2000 Mbps.



(c) Cross-connect, $\beta = 256$, 64 B packets, $L = 4096$, 3000 Mbps.

Figure 2.15: Latency distributions in different scenarios obtained via measurements and model. Dashed green curves are obtained by shifting the original model curves by a constant offset that is obtained via calibration measurements.

In case of all three combinations, we can see that the shape of the distribution is retained and that in case of both the measurements and the model, the overall processing time roughly follows a uniform distribution. This is consistent with the fact that packets experience different waiting times depending on their time of arrival relative to the next batch pick-up event and arrive at the system at a near-constant rate.

When comparing the first two subplots, we notice that the latency ranges differ despite the respective scenarios having the same applied rate of 2000 Mbps. This phenomenon is explained by the different packet size which in turn affects the batch size and therefore batch service time, leading to a lower total processing time in case of the scenario with larger packets in Figure 2.15b. Furthermore, the latency values shown in Figure 2.15c are even lower and lie within a narrower range. This stems from the simple cross-connect scenario in which a higher rate can be sustained due to more heterogeneous packets and absence of table look-ups during packet handling.

Finally, when looking at the adjusted model values obtained by adding the constant offset obtained during calibration measurements, the close fit of the model can be seen.

In summary, the results presented in this subsection highlight that the model is also capable of accurately reproducing the behavior of VNFs in terms of the latency that is experienced by packets. Furthermore, this capability persists throughout numerous parameters and therefore shows that the model generalizes well. Based on this, we are able to answer RQ2.3 and conclude that the developed model can reliably predict critical KPIs of modern software network functions.

2.5 Lessons Learned

Designing and operating high performance network functions is a complex and expensive task. The development and testing of high performance hardware appliances require immense financial as well as time expenditures and the subsequent maintenance of deployed devices is a very laborious task. Still, even after the developmental and operational overhead, hardware appliances remain relatively inflexible in today's fast-changing network landscape. The rate at which new services need to be deployed, new protocols need to be supported and systems need to be scaled to meet growing demands require new, more dynamic solutions.

One of these solutions that promises increased flexibility, more cost-efficient operation and reduced time-to-market is NFV. The paradigm aims to revolutionize the way networks operate by replacing monolithic, proprietary ASICs with, in comparison, lightweight and dynamic software solutions. This migration towards software allows for the application of well known practices during development, such as continuous integration [102] to significantly speed up and simplify the process of developing new as well as updating existing network functions while at the same time reducing management overhead and financial expenses.

However, the introduction of software components raises several new challenges, the most crucial one being the reduced performance of software tools when compared to highly specialized ASICs. To address this issue, new mechanisms to both monitor and predict the performance of these software tools need to be investigated in order to be able to assess system performance during both development of new functions and after function rollout. Similarly, mechanisms to solve systems dimensioning, parameter optimization as well as bottleneck detection need to be investigated.

To this end, we identified and investigated three distinct research questions in this chapter.

RQ1) How can the processing performance of generic network functions be monitored?

RQ2) Can this be achieved in a VNF-agnostic way while treating network functions as a black box without access to the VNF source code?

RQ3) Can the monitored packet processing times be exploited to develop accurate, generalizable performance models of state-of-the-art network functions?

To address these, we designed a novel approach to enable accurate, low overhead monitoring of software based network functions called *in-stack monitoring* (RQ1). Thereby, components of the existing network stack are exploited to provide precise measurements of the processing time of a generic network function. In the context of a case study we have shown that the approach is well suited for both performance assessment during development and monitoring after live deployment as the tool is both able to capture the accurate processing time distribution during calibration and detect performance degradation during online operation, while remaining network function agnostic (RQ2). Furthermore, we developed a discrete-time model that allows the prediction of several KPIs when it comes to software based network functions, such as the packet loss probability and waiting time distribution (RQ3). To this end, we applied the aforementioned measurement approach to a state-of-the-art packet processing framework, called VPP, and constructed an accurate model on basis of the obtained measurement data. We have shown that model predictions are highly accurate for all relevant KPIs.

The combination of these tools enables both the highly accurate monitoring of network function performance and the prediction of KPIs during system dimensioning and configuration, thereby unlocking additional optimization potential. Furthermore, the nature of these tools allows their integration in modern

software development pipelines, thereby reducing validation overhead during function development and in return reducing the time-to-market. At the same time, our proposed monitoring approach in combination with the capability to perform detailed parameter studies using the introduced model can be used to quickly adapt function parameters or perform resource scaling based on current demand. Ultimately, the mechanisms introduced in this work can be exploited in several ways to better understand the performance characteristics of network functions and, depending on their exact application, can be used to advance closer to the goal of NFV, to replace hardware with software.

3 Abstracting Heterogeneous Data Plane Solutions Through a Single API

In current and emerging networks, the performance of packet processing components is one of the most crucial parameters. As discussed before, the raw performance of pure software solutions is often times inferior when compared to dedicated hardware appliances. However, not all tasks involved in network operation require the flexibility of software solutions. Instead, tasks like layer2 forwarding, access control or simple header modifications may well be performed by programmable hardware solutions in order to benefit from a trade-off between the performance of hardware and the flexibility of pure software solutions. Thus, either one, or a combination of multiple different, not necessarily homogeneous, data plane solutions, may be used to realize a given task. Depending on several factors such as required throughput, latency or flexibility, but also how often configuration changes are expected or if dynamic resource scaling is required different solutions or technologies may be best suited to solve a given task. However, the combination and integration of differing data plane technologies raises new challenges regarding both the performance overhead induced by the concept of a mixed technology data plane and control solutions that allow the configuration of such systems.

On one end of the spectrum, pure software solutions allow for highly flexible setups both regarding scale and their supported feature set. These software solutions, however, often come with a performance penalty over hardware ap-

pliances. These dedicated ASICs, on the other end of the spectrum, feature exceptional performance for a very narrow and ossified feature set.

In addition to the pure software and pure hardware solutions, the introduction of technologies like SDN [109] with its flexible configuration protocol OpenFlow [28] and programmable hardware solutions like P4 [29] allows for a middle ground, providing increased flexibility through programmability while maintaining ASIC-like performance.

On the one hand, hardware data plane solutions are, by definition, equipped with a specific capacity for processing, storage and other relevant tasks. For example, an SDN switch with a total Ternary Content-Addressable Memory (TCAM) capacity of N entries and a maximum number of M flow tables will, regardless of potential optimizations, run into a bottleneck if more than N rules need to be installed or more than M flow tables are required. A common way for developers to deal with this problem is by creating explicit mitigation strategies for missing capabilities or bottlenecks. However, this is a redundant, complex and time-consuming task, which leads to additional development effort and cost in the best case, and to feature abandonment and prevention of innovation in the worst. On the other hand, performance limitations of software may impede the adoption of more flexible data plane solutions. These issues, in return, lead to longer development times and release cycles which adversely affect the adoption of SDN as well as NFV as mainstream technologies.

To this end, we discuss the concept of data plane abstraction through an intermediary translation agent, sitting between the control and data plane, and present a proof-of-concept implementation that enables the aggregation of multiple hardware as well as software-based data plane solutions to transparently emulate multi-technology data plane devices. Hence, we identify and investigate the following research questions.

RQ3.1) Are multi-technology data plane solutions feasible using existing technologies? This includes the definition of an abstraction concept that allows the integration of multiple data plane technologies into flexible packet processing pipelines.

RQ3.2) Can a proof-of-concept be realized using existing, state-of-the-art control plane and data plane solutions? This includes the practical integration of readily available solutions to proof the realizability of the designed concept.

RQ3.3) What are the performance and scalability limitations of the proposed concept? This includes a detailed discussion as well as measurement study regarding the capabilities of the proposed approach.

In order to address these research questions, we provide a detailed taxonomy of related work as well as the required background for the remainder of this chapter in Section 3.1. Section 3.2 introduces the concept of data plane abstraction developed in this work and outlines the design goals behind the abstraction architecture before introducing example use cases. Section 3.3 presents a detailed analysis of the performance of our proof-of-concept implementation. Finally, Section 3.4 discusses limitations of the proposed approach before Section 3.5 summarizes the insights and results from this chapter. The main contributions can be summarized as follows.

C2.1) The design and realization of a novel approach for the integration of multiple data plane technologies.

C2.2) A detailed investigation regarding the performance and discussion of limitations of the proposed approach.

These contributions have been published in the past and are condensed in this monograph based on the following scientific publications.

- Geißler, S., Herrnleben, S., Bauer, R., Grigorjew, A., Jarschel, M., Zinner, T.: "The Power of Composition: Abstracting a Multi-Device SDN Data Path Through a Single API," in IEEE Transactions on Network and Service Management (TNSM), 2019. [20]

- Geißler, S., Gebert, S., Herrnleben, S., Zinner, T., Bauer, R., Jarschel, M.: "TableVisor 2.0: Towards Full-Featured, Scalable and Hardware-Independent Multi Table Processing," in IEEE Conference on Network Softwarization (NetSoft), 2017. [3]

3.1 Background and Related Work

Many network applications rely on sophisticated packet processing and advanced pipelining, thereby combining multiple processing steps to realize complex network functions. On the one hand, common SDN examples include source address validation [110], d-dimensional packet classification [111], wildcard rule caching [112], controller modularization or hierarchical network management [113]. On the other hand, virtual network functions are expected to be composed of several, atomic elements such as header reads, header writes, state tracking [35] as well as more complex operations like database lookups or cryptographic operations.

The available hardware and software solutions to realize these use cases, however, may not necessarily support all required capabilities or provide sufficient resources. We call this phenomenon a “mismatch” between control plane requirements and data plane capabilities. Here, we can differentiate between functional limitations such as missing features like monitoring counters or support for specific protocols and non-functional limitations such as memory capacity or maximum throughput.

There are different approaches to deal with this problem. The easiest solution is to simply develop purpose built applications or appliances that provide the required capabilities or use devices with high overall flexibility. Programmable switches in combination with OpenFlow or P4-enabled devices are promising candidates here. Another, more realistic, approach is to accept — and maybe even encourage — heterogeneous infrastructures and then hide the heterogeneity with unified and silicon-independent APIs. P4 Runtime is a promising recent development that does exactly that. Finally, there are various works that try to

improve flexibility and scalability of the data plane itself, e.g., with respect to pipeline processing (see Table 3.1).

There are, however, two fundamental problems that cannot be solved with just unified APIs or improved data plane solutions: the conceptual limits of flexibility and the resource constraints of individual devices with respect to both performance and scalability.

3.1.1 Conceptual Limits of Flexibility

It is not likely that current — or future — device generations provide sufficient flexibility in the long run. Various enhancements to the OpenFlow protocol and corresponding devices clearly demonstrate this and while current developments in the area of programmable devices have alleviated this issue, the inherent limitations of single devices will, by definition, not be solved through these advances. [114] extended the match-action abstraction to support autonomous stateful decision-making. [115] added a new API to allow autonomous generation of packets. [116] proposed approximation techniques to enable the application of otherwise excessive data plane procedures. [117] tackled the problem of slow flow table entry installation. Even more important, completely new control plane requirements may emerge that cannot be realized with a new configuration file, pipeline template or firmware update, e.g., in-network support for distributed machine learning and time-sensitive networking [118]. As a result, it is simply not realistic to control every possible device with a unified API such as P4 Runtime, especially if we are talking about devices with bleeding-edge capabilities often used in the research community.

Instead, we need a way to efficiently deal with different existing control channel protocols. The proxy-layer architecture in conjunction with the new protocol translation concept introduced in Section 3.2 is a first step in this direction. The method of transparently processing OpenFlow messages is not particularly new. Similar techniques are used for network virtualization [119–121], to realize hypervisor functionality [122], to interoperate with non-SDN legacy net-

work equipment [123] or to transparently deal with flow table limitations [124]. The novelty of our approach is that developers can easily create and deploy their own translation application for every possible control channel protocol, without extensive changes to the network operating system or the control apps, which simplifies rapid prototyping and research work. Furthermore, our approach allows the utilization of resources of more than one device, which is discussed in detail in the next section.

3.1.2 Constraints of Individual Devices

The resource and capability constraints of individual devices are equally important. Programmable switches are, like every other hardware device, limited to the resources and capabilities of a single device. Control plane requirements going beyond what is provided by a single device cannot be satisfied, even if two cooperating devices could easily fulfill the request. Existing work to alleviate this limitation can be distinguished into two categories: software-based solutions that do not touch the switch hardware and hardware-based solutions. We present prominent related work for both categories and explain how previous work differs from our approach. A summary of this taxonomy can be found in Table 3.1. The limitations of existing solutions clearly shows the gap in literature that is addressed in this thesis.

Software-based Approaches

Most software-based solutions focus on either pipeline flexibility or scalability aspects. We start by elaborating on flexibility by comparing the studies in category C1 in Table 3.1 and discuss software-based scalability solutions later on.

Several works try to address the problem of pipeline flexibility. SDFTP [113] introduces software-defined flow table pipelines with an arbitrary number of stages and adaptive table sizes. A special mapping logic is used to embed virtual software tables into the hardware tables of the switch. FlowAdapter [125] is a middle layer between the hardware and software data plane that provides

Table 3.1: Comparison between TableVisor and selected approaches from related work

	Category	Approach	Scope			Features			
			Pipeline Flexibility	Device Scalability	Network Scalability	Control Plane Transparency	Exploit Shared Resources	No Infrastructural Changes	Separation of Concerns
SW	C1	[113]	✔	✘	✘	✘	✘	✔	✘
		[125]	✔	✘	✘	✔	✘	✔	✘
		[126]	✔	✘	✘	✘	✘	✔	✘
	C2	[127]	✘	✔	✘	✔	(✔)	✘	✘
		[124]	✘	✔	✘	✔	✔	✔	✘
		[128], [129]	✘	(✔)	✔	✘	✔	✔	✘
HW	C3	[130]	✔	(✔)	✘	✘	✔	✘	✘
		[131], [132]	✔	(✔)	✘	✔	✘	✘	✘
	This work		✔	✔	(✔)	✔	✔	✔	✔

support for multi-stage pipeline processing by properly mapping rules onto existing hardware capabilities. FlowConverter [126] tries to generalize the above ideas and presents an algorithm that can translate between different forwarding pipelines. Table Type Patterns (TTP) [133] for OpenFlow also introduce flexibility by allowing the controller to negotiate pipeline details with hardware switches. ALIEN HAL [134] focuses not directly on pipeline flexibility, but follows similar design principles as FlowAdapter and TTP by using a hardware abstraction layer to realize OpenFlow capabilities on legacy network elements. Furthermore, previously proposed approaches like Frenetic [135] and Pyretic [136] revolve around high level languages for programming collections of network switches. However, these approaches are largely limited to OpenFlow and,

like other previously proposed solutions, don't address the limitations imposed by the constraints of singular hardware devices. Instead, these approaches focus on a simplification of the programming interface of SDN-enabled devices.

Since the approach presented in this thesis is also software-based, there are similarities to the above approaches, primarily with respect to the basic motivation. However, there are three important conceptual differences. (1) Existing solutions for software-based pipeline flexibility are, by design, limited to the resources of a single switch, i.e., the approach can only be used if there is enough free space left in at least one of the hardware tables, which imposes inherent restrictions with respect to scalability. Our solution copes with this important challenge by combining the resources and capabilities of multiple devices into one emulated device. (2) Our approach is used in a fully transparent fashion as neither the controller nor the applications have to be modified, and the approach can be used out-of-the-box in any OpenFlow based network. While some approaches like FlowAdapter and HAL have similar properties, others sacrifice transparency. SDFTP [113], for example, introduces a new southbound interface for all table operations which requires non-trivial changes in the control plane. (3) Our approach does not solely focus on flexible pipeline processing but rather considers it as one single use case among a broader set of different applications. The application engine presented in Section 3.2.2 can be seen as a generic platform to transparently include different functions. Following this approach, we can combine pipeline flexibility with use cases from other research domains such as TCAM space optimizations, control channel logging or protocol conversion. Even the integration of complex VNFs performing arbitrary packet processing tasks could be realized, as long as the interfaces are compatible or instructions can be translated by our shim layer.

Category C2 in Table 3.1 is addressing device as well as overall network scalability. Solutions for device scalability try to cope with limited capacities and capabilities of a single device, e.g., by adding a virtual switch with high table capacity [117, 127]. However, this requires infrastructural changes and is associated with a performance degradation for all flows that are forced to use the

slow path via the virtual switch. Performance characteristics and limitations of virtual switching were intensively studied in the recent past [137–139]. Even if only a fraction of the traffic is affected, this might be inapplicable for many scenarios [127]. Others try to achieve similar results by exploiting spare resources of co-located devices [124]. However, unlike our approach, these approaches do not consider pipeline processing.

Solutions for overall network scalability usually consider flow tables and TCAM space as a shared resource. Palette [128], DIFANE [140] and One-Big-Switch (OBS) [129] are prominent examples. The general idea of gathering shared resources under a unified abstraction is similar to our solution. However, these solutions have a fundamentally different scope and try to abstract the whole infrastructure (i.e., every switch), while we aim for a more localized solution, focusing on smaller sets of switches. In addition, the aforementioned solutions introduce policy abstractions that change how networks are used and programmed, which impedes transparency to and compatibility with legacy applications. The high level of abstraction introduced by these changes is a blessing and a curse at the same time. It shields application programmers from low level details but makes it difficult, if not impossible, to realize proper pipelining, because the pipelining itself is not covered by the individual abstractions. Looking at the various use cases that are difficult to realize without explicit, application controlled pipelining [110–113], we argue that this kind of abstraction is not necessarily a panacea. As a result, we designed our solution as a transparent proxy layer without changing the interface that is used for pipelining.

Hardware-based Approaches

Category C3 in Table 3.1 compares prominent examples of hardware-based solutions. The core idea here is to provide programmable network devices with freely definable packet processing pipelines. The FlexPipe architecture of Intel's FM6000/FM7000 series [141] allows programmable parsing of incoming traffic based on a TCAM/SRAM/MUX structure. Protocol Oblivious Forwarding [142] introduces a generic flow instruction set to make the data plane

protocol-oblivious. Reconfigurable Match-Action Table (RMT) [132] proposes a model for re-configurable match tables and enables dynamic, in-field reconfiguration of the data plane. Recently, dRMT [143] introduced a new RMT architecture, in which memory as well as compute resources are disaggregated and moved to a general pool that can be accessed by all pipeline stages over a crossbar. ESwitch [131] proposes a novel architecture that is able to generate efficient machine code for SDN switches based on packet processing templates inspired by the OpenFlow pipeline.

The limitations of programmable switches are twofold. First, they require special hardware currently not available in large quantities. While this may change in the future, it is more likely that such devices will complement existing infrastructures, rather than completely replace them. So the control plane has to either deal with this expected device heterogeneity directly, by differentiating between programmable and non-programmable devices in the applications, or use some kind of abstraction as is exploited by the approach presented in this thesis. Second, even if we assume that every device in the network supports the same degree of programmability, these devices will still be equipped with fixed resources, say a total TCAM capacity of N entries and a maximum number of M flow tables. Existing solutions such as RMT [132] and ESwitch [131] can boost these numbers, so that N and M might be much larger compared to currently used OpenFlow switches. NOSIX [130] is the only hardware-based approach that we are aware of that follows a similar design paradigm by exploiting shared resources. Similar to our approach, NOSIX envisions a *lightweight portability layer* for the control plane and can make use of special-purpose tables in different switches. However, the approach requires special hardware support and breaks transparency.

Programmable Data Plane

The programmable data plane concept describes the idea of programmatically reconfiguring the packet processing mechanisms provided by data plane devices at runtime [144]. The concept thereby applies to both software solutions and

hardware devices and has gained significant traction through the introduction of the separation of control and data plane introduced in SDN [25]. The splitting of formerly joint components into a separate, intelligent control unit and simple, dumb data plane devices allows for far greater flexibility and enables new applications in modern networks [109, 145].

Especially the introduction of OpenFlow [28] has led to the adoption of the programmable data plane concept as a mainstream technology in networking as a standardized protocol for the configuration of widely available programmable switches enabled the adoption in both industry and research [146]. With OpenFlow, it is possible to issue well-defined instructions to compatible hardware or software switches that modify their internal match-action tables and hence manipulate their respective forwarding behavior [146]. A detailed description of these controller-to-switch and switch-to-controller instructions can be found in the OpenFlow specification [147].

Naturally, the programmable data plane concept has since been well researched and new, more powerful approaches have emerged. Covering all developments in the area at this point in this thesis would be unreasonable. Instead, a broad overview of developments in the area of programmable data plane devices is provided by Bifulco *et al.* [144].

However, two developments that warrants dedicated coverage are Programming Protocol-independent Packet Processors (P4) [29] and NFV. P4 aims to extend the match-action functionality provided by, e.g., OpenFlow to enable the processing of arbitrary header fields, and hence arbitrary protocols. Furthermore, P4 provides basic arithmetic operations and dissolves the ossified lookup tables of early programmable switches by providing a set of resources that can be programmatically allocated as needed. A detailed description of the P4 programming language can be found in the respective specification [148]. In this context it is important to differentiate between P4 as a hardware platform and OpenFlow as a control channel protocol, as P4-enabled devices still need some form of control API to push match-action rules into their tables. However, it is clear that the control capabilities of OpenFlow only cover a small subset of

what P4 devices can do. Different control approaches are hence required. To this end, we cover the translation between OpenFlow and a proprietary P4 control mechanism in Section 3.2.

The concept behind NFV, as already discussed in Section 2.1, covers the processing of simple network tasks like packet forwarding but also complex operations such as Deep Packet Inspection (DPI) or encryption. To this end, software frameworks like DPDK [45], VPP [48] or FastClick [49] are used to develop high-performance software solutions. In addition, the concept of network function offloading has been introduced to further increase the processing capacity of network functions [26, 35, 149, 150]. However, the integration of heterogeneous data plane technologies comes with new challenges regarding interoperability with respect to both their configuration and manner of processing [151]. To this end, we investigate the feasibility of integrating heterogeneous data plane technologies by providing a unified interface towards the controller. This aspect is covered in Section 3.2.3.

3.2 TableVisor

This section covers the proposed proof-of-concept implementation of a data plane abstraction service. We start by presenting the design concept of TableVisor, describe available functionality, and conclude with possible use cases that can be realized by applying its features. Note that the proof-of-concept implementation is available as an open source project via Github[1].

3.2.1 Design Principles

When it comes to the design choices made for our proposed approach, we need to differentiate between the goals we want to achieve and the design principles needed to realize these goals. To this end, we present a brief overview of the aforementioned goals that are described in more detail in the following sections.

[1] https://github.com/lsinfo3/JTableVisor

First and foremost, TableVisor should be able to do two things, with the first one being the aggregation of features and capacities provided by multiple, heterogeneous data plane devices into a single emulated device. The second major design goal is the integration and seamless interoperability of differing data plane technologies such as P4, whitebox switches or software solutions. In order to achieve these goals, a number of design principles needed to be defined.

First, TableVisor is expected to work in a fully transparent manner, meaning that both control plane and data plane are unaware of the abstractions happening by our service. To achieve this, we need to seamlessly translate between different protocols and control mechanisms and can hence use both unmodified data plane and control plane solutions. Second, our service needs to work fully placement agnostic, meaning that TableVisor can be deployed on an arbitrary host somewhere within the network in order to allow the integration into more complex network environments as well as interoperability with a multitude of different data plane technologies. Finally, TableVisor needs to respect the separation of concerns [152], meaning it may not act as a control plane entity itself and is not allowed to make control decisions while at the same time it is not a data plane entity and is not allowed to interact with data plane traffic. Instead, our service merely mediates between the two planes and translates messages to ensure interoperability. The control plane is still in full control of forwarding decisions while the control plane is still handling all the traffic processing and forwarding.

3.2.2 Architecture

TableVisor has a modular structure in order to allow the addition of extensions and further functionalities in the future. It comprises a central core and three main logical layers — the upper layer endpoint, the application engine, and the lower layer endpoint — as shown in Figure 3.1. The separation between upper and lower layer endpoints as well as the application engine allows a clear separation of responsibilities with regard to the workflow of TableVisor.

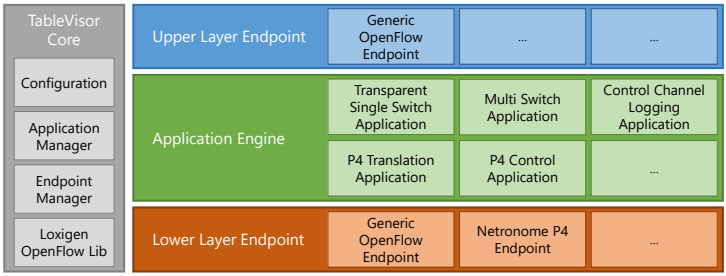


Figure 3.1: TableVisor Architecture.

This is realized through Java APIs exposed by the TableVisor Core which enables easy implementation of additional endpoints or application engine modules.

The *upper layer endpoint* is responsible for the communication with the controller and handling of protocol specific mechanisms (e.g., keep alive messages). The endpoint parses control protocol messages into a workable data structure that can be processed by the application engine and vice versa. The addition of further upper layer endpoint implementations allows TableVisor to work with control plane instances that are not OpenFlow-capable, e.g., legacy network management systems.

The *application engine* is responsible for passing messages through all loaded applications. An application specifies whether and how a message is processed, e.g., rewriting of table IDs or actions. TableVisor comes with a number of preselected applications, including basic applications such as for maintaining the OpenFlow Control Channel, but also applications that change the representation and behavior of the virtual switch.

The applications themselves are structured as an ordered pipeline. Control messages are passed through all loaded applications in a predefined order, which is specified at a global scope during their implementation. Applications can be loaded and unloaded in the configuration file.

If an application is loaded, all messages are passed to it, unless blocked by an earlier application. The application order of messages from the upper and the lower layer endpoints are opposed, i.e., the *last* application that is passed by a control message from the upper layer endpoint is traversed *first* by messages from the lower layer, and vice versa. The feature set supported by the current version of the implementation is listed in Table 3.2.

The *lower layer endpoint* maintains the communication paths towards data plane devices. It parses, encodes and decodes messages that are to be sent to or received from the respective data plane devices. This could be OpenFlow messages, implemented by the *Generic OpenFlow Endpoint*, P4 control messages, managed by the *Netronome P4 Endpoint*, or additional novel or existing protocol implementations.

By exploiting upper and lower level endpoints, this architecture can achieve complete transparency towards both data plane and control plane. Therefore, it allows the usage of standard controller implementations as well as data plane devices without the need for modifications.

3.2.3 Features

The application set of TableVisor is focused around three major feature sets, namely, control channel translation, table capacity extension and device aggregation. The main tool used for their implementation is the mapping of global table IDs, as seen by the controller, to local devices and tables, while handling all their intermediate communication. The individual feature domains, as directly applicable by the use cases, are explained in the following.

Control Channel Translation

The control channel translation functionality encompasses everything that involves the modification of control messages for compatibility reasons. This includes not only the translation between different protocol families, but also different versions of the same protocol, as detailed in the following.

Table 3.2: TableVisor Application Engine Modules.

Transparent Single Switch Application Transparently forwards OpenFlow messages between a single switch and the control plane. Can be used in conjunction with other applications to add functionality, such as the control channel logging application that monitors an OpenFlow control path.
Multi Switch Application Aggregates multiple switches into a single pipeline that is presented towards the controller as a single multi-table switch. Allows the realization of multi-table use cases and exploits the heterogeneity inherent to the landscape of OpenFlow-enabled hardware switches.
Control Channel Logging Application Monitors OpenFlow control messages between the lower and upper layer endpoints. This can be used in conjunction with other applications either for debugging or live monitoring purposes.
P4 Control Application Provides OpenFlow functionality towards the control plane and constructs OpenFlow messages based on statistics and rule sets provided by proprietary P4 management tools.
P4 Translation Application Provides translation functionality between OpenFlow in the control plane and proprietary control protocols in the data plane.

OpenFlow device heterogeneity. The idea of control channel translation originally evolved around the problem of device heterogeneity in the early OpenFlow hardware implementations, as different switches exhibit different feature sets and device specific behavior with respect to the supported OpenFlow versions, default table numeration, and general conformability with the standard. This problem is most prominent when comparing early implementations of different vendors. TableVisor can be used in such situations to alleviate the problem of protocol mismatches and different expectations between con-

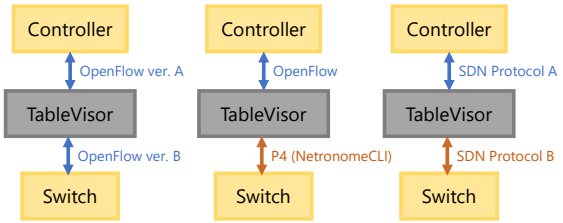


Figure 3.2: Illustration of the control channel translation features between specific OpenFlow versions, and generally between different control plane protocols.

trol and data plane. The general instructions of various SDN controllers can be tailored to the needs of specific devices' behavior, and their replies can be generalized to be understood by a standard-conformant software implementation of OpenFlow. This is shown on the left in Figure 3.2, where TableVisor translates between different OpenFlow versions.

P4 device integration. Going one step further, TableVisor features a semi-automatic translation between the OpenFlow protocol and the P4 running configuration. Therefore, the translation application expects an OpenFlow packet from the upper layer endpoint. As both protocols are based on a similar match-action architecture, the translation engine must match the intents contained in the original message to the available table names, action names, and header fields in the P4 program. Note that, as with each translation, the expressiveness of the resulting control channel is limited by the intersection of possible command sets from both languages. In this case, OpenFlow supports a subset of P4 capabilities, hence the level of control is limited by the highest supported OpenFlow version in the controller and the actually deployed P4 program. However, if necessary, unused OpenFlow header fields may be used to address sophisticated headers in P4 programs. In many scenarios, the required operations can be triggered implicitly in the P4 control flow.

TableVisor leverages the existing P4 program in order to learn the respective mappings. Therefore, the program is annotated with the corresponding ta-

ble IDs, header field names as well as action names and parameters. TableVisor parses these annotations and stores their respective mappings. Listing 1 contains an example snipped of a P4 program with a simple mapping from the OpenFlow table ID `0` to the P4 table name `acl_tbl`.

```
// @TV table 0
table acl_tbl {
reads { ...
```

Listing 1: Annotated P4 program excerpt as example table mapping: acl_tbl ↔ table 0.

Similarly, the header field names can be mapped to their OpenFlow counterpart inside the `reads` block of the table. Note that this mapping might be applied locally for this specific table, which enables the use of different mappings in different tables, if desired. Our current implementation applies a global mapping to reduce administrative overhead during configuration. Listing 2 maps OpenFlow's `in_port` to the corresponding P4 metadata field, and the `ipv4_dst` to our defined destination address header field name.

```
// @TV table 2
table routing_tbl {
reads {
// @TV field in_port
standard_metadata.ingress_port: exact;
// @TV field ipv4_dst
ipv4.dstAddr: exact;}
actions { set_dst_mac; }}
```

Listing 2: Annotated P4 program excerpt with two example field mappings: in_port and ipv4_dst.

Finally, mapping actions requires special care. They do not only contain their own name, but also a list of parameters that may be supplied by the controller at runtime. In addition, OpenFlow allows the execution of multiple actions at the

same time, while P4 is limited to a single, custom defined action for each table entry. Therefore, in Listing 3, a single P4 action `set_dst_mac` is mapped to multiple OpenFlow actions, and vice versa. Note here that the `GOTO_TABLE` instruction is called explicitly in OpenFlow, while the table transition is defined separately in the `control` block in the P4 program.

```
// @TV action SET_FIELD_ETH_DST ETH_DST=mac
// @TV action GOTO_TABLE_123
action set_dst_mac(mac) {
modify_field(eth.dst, mac);
}
```

Listing 3: Annotated P4 program excerpt for action mapping: set_dst_mac is associated with SET_FIELD_ETH_DST and GOTO_TABLE.

These name mappings are utilized by the translation application to create a JSON string. As our current implementation is designed for the Netronome Agilio CX SmartNIC[2], this JSON string is then passed into the `RTECLI` interface by the respective lower layer endpoint which pushes the new flow rules to the SmartNIC that is associated with it. The entire example mapping, along with further explanation, can be found in the GitHub repository.[3] This implementation is shown in the middle of Figure 3.2.

Further extensions. Note that the concept can be further extended and generalized to support additional protocols by adding new endpoints, both in the upper and lower layer. With little more effort, new applications could handle the inclusion of additional devices well beyond the match-action abstraction, for example to enable control of hardware accelerated security features in legacy networking devices or complex software network functions performing arbitrary packet processing tasks like DPI or application layer firewalling. The general translation between different implementations of SDN protocols is illustrated on the right in Figure 3.2.

[2]https://www.netronome.com/
[3]https://github.com/lsinfo3/JTableVisor/tree/master/example2

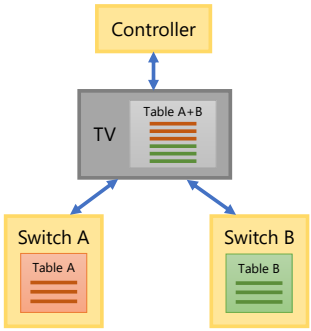


Figure 3.3: Illustration of the table capacity extension feature.

Table Capacity Extension

In early deployments of OpenFlow switches, hardware accelerated table capacity was very limited, with some models only allowing a few hundred OpenFlow rules stored in their TCAM.[4] Some use cases require large amounts of flow rules [153], which then need to be handled explicitly by the network administrator or the SDN control plane deployment. With the TableVisor abstraction, multiple hardware tables can be mapped to a single, virtual table and presented in a single device to the control plane. In this scenario, TableVisor would handle dependencies between the different match-action rules, such as implicit header matches due to higher priority flow rules. This concept has been discussed in detail in [20].

Figure 3.3 provides a visual example of the intended use. TableVisor presents a single table hosted by a single device towards the controller. The individual match-action flow rules are split between Switch A and Switch B in order to provide the necessary TCAM space, thereby simplifying the implementation of control plane applications.

[4] https://support.hpe.com/hpsc/doc/public/display?docId=c04217797&lang=en-us&cc=us

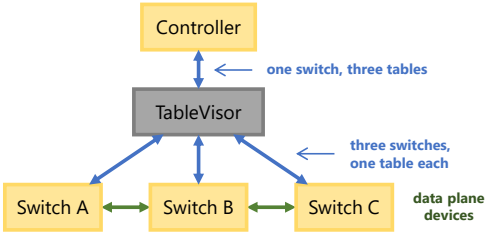


Figure 3.4: Illustration of the device aggregation feature.

Device Aggregation

Similarly to the table capacity extension, some features of a single deployed switch are often not sufficient for a task. Simply deploying a "bigger" device would often solve problems with respect to number of ports, number of tables, or even supported features. However, such hardware is not always available, or may be too expensive for the task at hand. TableVisor features a cost-efficient way to aggregate features from multiple networking devices in a single, virtual device. This is especially useful if these devices feature very different characteristics, e.g., a "dumb" forwarding device with many ports and a smart router, VNF, or programmable NIC.

The general approach towards the different aggregation features is shown in Figure 3.4. The capabilities of multiple devices (Switch A, B and C) are presented as a single switch with a mix of all of their capabilities, tables, and ports towards the control plane, based on the intended use case.

Port extension. If a single device does not provide sufficient ports to connect to all of its neighbors, but features all of the required functionality, TableVisor can be used to extend the port number by seamlessly integrating another device. The devices would best be connected via dedicated trunk links at higher rates due to the increased amount of traffic between these two switches. Alternatively, multiple ports can be spent for the inter-device communication, depending on the situation.

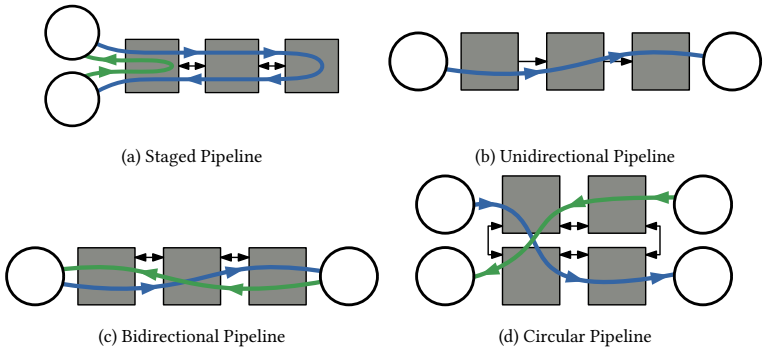
Extension of number of tables. In many cases, SDN forwarding devices perform multiple actions based on multiple matching criteria. To prevent an exponential explosion of flow rule numbers [154], multiple tables can be used in succession for these tasks. However, not all devices support multiple tables natively. TableVisor can be used to aggregate existing devices and present their tables next to each other in a single, multi table device to the controller. In particular, it would handle the mapping of global table IDs to local, device-specific tables, as well as ensure interoperability, e.g., by translating `GOTO_TABLE` messages to their respective `OUTPUT` actions if the desired table is located on another switch.

Aggregation of features. Finally, not all tables of an SDN switch provide the same capabilities. For example, only a limited number of hardware accelerated rules is often able to push or pop header fields at line rate in the data plane. Sometimes, the required characteristics for a specific task are not simultaneously supported by a single device, e.g., the required number of ports and a specific set of actions, but can possibly be provided by sophisticated hardware such as P4 devices like Barefoot's Tofino hardware.[5] In these cases, TableVisor's mapping can be used for a seamless combination of separate devices with the required capabilities. This way, the required characteristics can easily be recreated by available, cheap hardware in a brownfield deployment or for rapid prototyping of new concepts as well as to emulate devices for research purposes.

3.2.4 Supported Topologies

This section covers the different underlying topologies the TableVisor approach is able to leverage for its data plane abstraction. Figure 3.5 shows the four most common pipeline structures. Note that these topologies represent common design patterns. The TableVisor approach is not limited to these types of topologies and can be adapted to the specific use case.

[5] https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch.html



(a) Staged Pipeline (b) Unidirectional Pipeline

(c) Bidirectional Pipeline (d) Circular Pipeline

Figure 3.5: Available topologies for TableVisor abstraction.

Staged Pipeline. Figure 3.5a shows the staged pipeline, in which each element provides additional features or TCAM space. All hosts or uplink switches, represented by circles in the figure, are connected to the first stage of the pipeline. From there, packets only need to enter the stage providing the capabilities needed for their specific processing requirements. This minimizes the overhead induced by the abstraction while still enabling more complex use cases. The abstraction is realized by emulating multiple tables, each represented by at least one switch of the pipeline.

Unidirectional Pipeline. An extension of the staged pipeline is shown in Figure 3.5b. The *unidirectional pipeline* allows hosts or up-link devices to connect to both ends of the pipeline. However, packets may only traverse the pipeline in one direction. A direct communication between network elements connected to different pipeline ends is thus not possible in this scenario. Instead, this emulation type is especially useful in VNF offloading scenarios that only need to handle unidirectional traffic.

Bidirectional Pipeline. The third extension supported by the TableVisor concept is the bidirectional pipeline illustrated in Figure 3.5c. This layout allows the connection of hosts or up-link switches to either the first or last switch

of the pipeline respectively. Simultaneously, the direction of paths through the pipeline is arbitrary in this case and packets can enter as well as leave the switch aggregate at both ends. This structural layout further increases the capabilities of the emulated switch at the cost of further management complexity. This pipeline type is best used in VNF offloading cases that require bidirectional traffic, e.g., in request-response scenarios.

Circular Pipeline. The final pipeline type supported by TableVisor is the circular pipeline shown in Figure 3.5d. Thereby, data plane devices are arranged in a ring topology and hosts as well as up-link switches can be connected to every element of the pipeline. This significantly increases the total number of ports available for the emulated switch. Hence, this emulation type can be used to deploy use cases in which a large number of ports is required. The drawback of this setup is the need for internal forwarding rules in order to ensure that traffic still reaches its intended destination. Depending on the specific use case, this may, in the worst case, lead to one forwarding rule per connected host or up-link device, significantly reducing the effective TCAM space available for the application.

Finally, these different topologies can be used by multiple TableVisor instances in a single network deployment, as shown in Figure 3.6. In this case, the controller identifies five devices in the network, indicated by the control channel connections. The switches connected to each of the respective TableVisor instances are abstracted into a single emulated device. In addition to the TableVisor instances, additional data plane devices can be used without the abstraction layer.

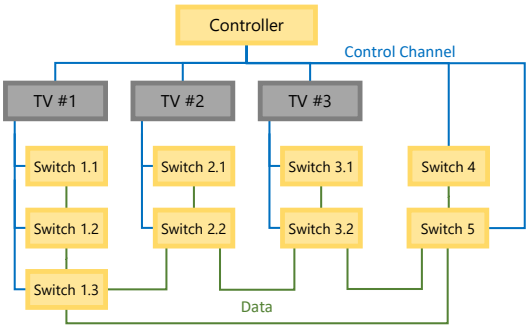


Figure 3.6: Schematic depiction of a network using multiple TableVisor instances.

3.2.5 Use Cases

In the following section, we present two specific exemplary use cases that show how the features presented in the Section 3.2.3 can be leveraged to design complex scenarios using simple and affordable network components. We start by detailing an access control list (ACL) setup comprised of multiple, non-expensive and highly available single table switches and how the device aggregation feature is used to realize this use case. Furthermore, we demonstrate how control channel protocol translation can be used to realize powerful use cases using common network components by describing an MPLS label edge router realized through the incorporation of non-expensive, P4-enabled devices into the data plane.

ACL Multi Table Switch

The first example we detail is the realization of an emulated multi-table switch performing access control as well as forwarding. A scenario like this can be especially useful in brownfield deployments in which new applications and devices have to co-exist with legacy solutions in both control as well as data plane.

The idea here is to combine multiple widely available and affordable single table switches to enable a single, multi table application. In this specific use case, the usage of multiple forwarding tables alleviates the common problem of flow table explosion [154] with legacy SDN devices. This problem essentially describes a combinatorial problem occurring whenever a single or multiple actions need to be performed based on combinations of two or more input values, like ACL rules and MAC addresses. In a single table scenario, all ACL rules would need to be recombined with all possible output MAC addresses, leading to $N \times M$ flow rules in total. When moving to a multi table scenario, the total number of required flow rules to achieve the same functionality is reduced to $N + M$. Thereby, the first table can perform the access control lookups and an independent second table is used to perform the L2 forwarding task. Figure 3.7 shows a schematic setup of TableVisor realizing this multi table switch by combining two single table devices.

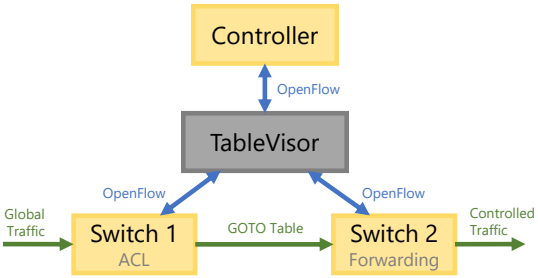


Figure 3.7: Schematic ACL setup using two single table devices.

More specifically, we leverage the device aggregation feature discussed in Section 3.2.3, to combine two, potentially heterogeneous, data plane devices into a single, emulated switch featuring two tables. This emulated device is then transparently presented to the controller as a regular hardware device. Hence, neither the devices, nor the controller are required to provide any specialized functionality to be used in this environment. Instead, all involved parties communicate

using their version of the OpenFlow protocol. Due to the control-channel protocol translation functionality of TableVisor, the two switches as well as the controller can thereby speak different versions of OpenFlow, or even a completely different protocol. At the time of writing, TableVisor supports OpenFlow versions 1.0 and 1.3 as well as the proprietary control API used by Netronome P4 devices used in the example use case in the next section. The processing required for this level of transparency, from control as well as data plane point of view, is fully handled by TableVisor.

Through the message processing performed by TableVisor, the system is able to emulate a multi-table switch towards the control plane without the need for modification of the controller or the involved data plane devices. This increases the reusability of legacy devices in brownfield deployments and allows researchers to quickly prototype control plane applications for which expensive hardware devices would be required, otherwise.

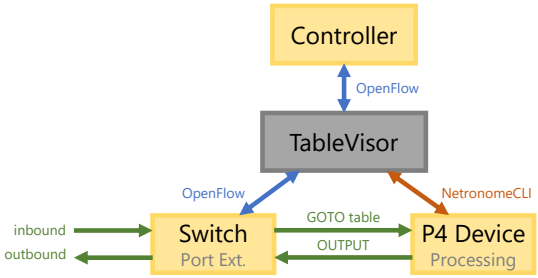


Figure 3.8: Schematic MPLS lable edge router setup using a P4 device.

P4 Label Edge Router

The second example application we discuss in this work is the realization of an MPLS label edge router using a regular OpenFlow device in combination with a two port P4 PCIe extension card. The goal here is to extend the functionality of a simple, affordable and widely available OpenFlow device by adding the

programmability of a single P4 device. Thereby, we are able to leverage the port capacity of the OpenFlow switch as well as the processing flexibility of the P4 device and combine them into a single, powerful, emulated data plane device towards the controller. The general setup of this application is shown in Figure 3.8. It is essentially a combination of the *port extension*, *feature aggregation* and *P4 device integration* features described in Section 3.2.3.

Here, the switch communicates with TableVisor using regular OpenFlow. The P4 device, a Netronome Agilio CX SmartNIC 2x10G, is controlled via the proprietary CLI tool shipped with these kinds of PCIe extension cards. TableVisor handles the translation between the OpenFlow protocol used to connect to the controller and the proprietary control interface of the card. This control channel translation mechanism allows the transparent integration of the P4 device into the data plane without the need for modifications of any part of the system. As the SmartNIC only features two ports, the OpenFlow switch is instructed to output all `GotoTable` instructions at one port connected to the NIC and forward packets coming from the other port connected to the NIC. Information required for the forwarding process can be included using the metadata functionality of TableVisor, as described in [20]. The P4 device then performs all the heavy lifting, like pushing and popping MPLS headers, essentially using the OpenFlow switch as a port replicator. Note that the Netronome devices used in this use case feature two 10 GbE ports while the OpenFlow switch only features 1 GbE regular ports and two 10 GbE uplink ports.[6] The implementation of this use case can be found in the GitHub repository accompanying this work.[7]

Note that the P4 device in this scenario could also be replaced by a VNF performing the processing tasks. This again simplifies rapid prototyping and seamless integration of complex networking functionality without modification of the control plane in a brownfield deployment.

[6] HP 2920-24G + 2x10G
[7] https://github.com/lsinfo3/JTableVisor

To summarize, we have introduced the concept of transparent data plane abstraction, realized by a dedicated shim layer between the network control plane and diverse data plane solutions. Furthermore, the features provided by our proof-of-concept implementation are outlined and we show that the integration of generic OpenFlow capable devices with P4-enabled programmable switches and NICs is possible. Applications such as port, table or feature aggregation have been outlined and exemplary use-cases have been realized using our proof-of-concept implementation. Based on the presented insights, we can conclude that multi-technology data plane solutions are indeed feasible and can be realized using already available technology on both control and data plane as well as using existing communication protocols like OpenFlow (RQ3.1, RQ3.2).

3.3 Performance Evaluation of TableVisor

In order to evaluate the implications of TableVisor in different use cases, we conducted extensive delay measurements with respect to different control plane interactions. We evaluate scenarios in which we measure the controller's interaction with the data plane devices, specifically:

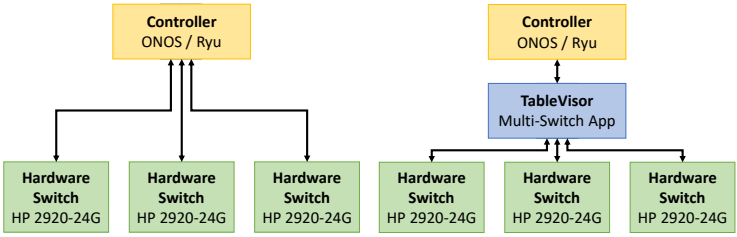
1. The way TableVisor influences the overall control channel delay of a bulk of operations, namely `FlowMod` installations targeted to multiple switches,

2. The influence on the delays of individual messages, for example `FlowStatsRequests`, during regular operation.

Thereby, the influence of the number of installed `FlowMods` as well as the number of connected switches is investigated with both software and hardware switches as well as varying controllers.

The three main OpenFlow instructions we use during the presented measurement study are `FlowMods` that modify the match-action rules installed in a switch flow table, `FlowStatsReqeuests` that query statistics about installed

flow rules from connected switches and `BarrierRequests` that instruct the switch to issue a `BarrierReply` once all previously issued instructions have been processed. The latter is used purely for monitoring purposes in order to trigger a notification once a device has completed all issued instructions. The exact message flows for the scenarios investigated in this thesis are shown in Figures 3.11 and 3.12.



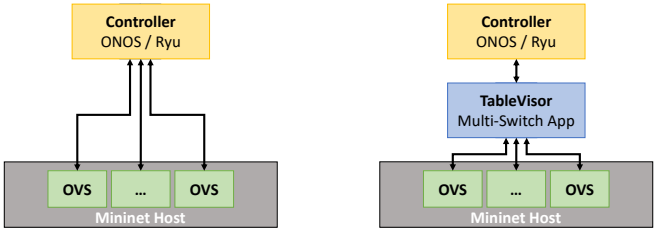
(a) Without TableVisor (b) With TableVisor

Figure 3.9: Performance overhead measurements with hardware switches.

Figures 3.9 and 3.10 provide an overview of the experiment setups for the performance comparisons. Here, Figure 3.9 represents the topology use in a staged pipeline scenario with one to three hardware switches, as well as two hosts[8] for the controller and TableVisor itself. The switches are either connected directly to the controller, or to the TableVisor instance, which acts as a single multi-table switch towards the controller. In order to investigate the impact of horizontal scaling of the pipeline, a similar topology with up to 50 switches has been emulated using Mininet, as shown in Figure 3.10.

In particular, we measure the `FlowMod` installation times with and without TableVisor by sending multiple `FlowMod` messages followed by a single `BarrierRequest`. Thereby, without TableVisor, each switch receives its `FlowMods` directly from the controller. In the scenario with TableVisor, all

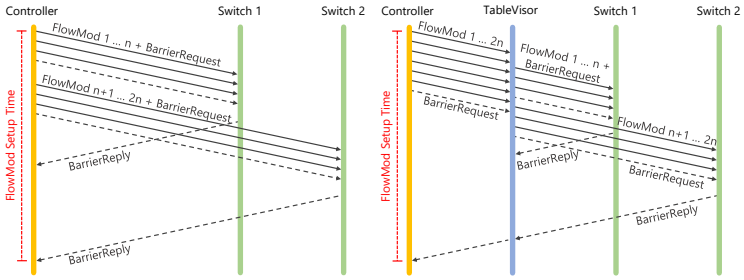
[8] Controller: Intel Xeon X5650 @2.67GHz with 36GB RAM, TableVisor: Intel Xeon D-1548 @2.00GHz with 32GB RAM.



(a) Without TableVisor (b) With TableVisor

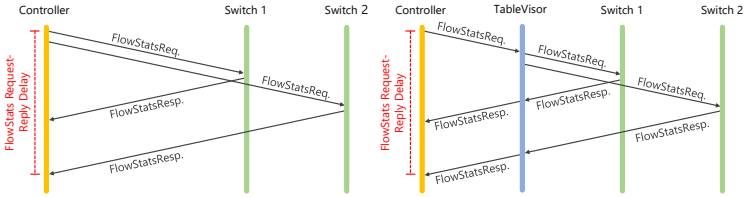
Figure 3.10: Mininet scenarios for switch number performance tests.

`FlowMods` are sent to the TableVisor instance, which then distributes the rules to its underlying hardware devices acting as tables in the emulated switch. The installation time is then given by the time difference between the first `FlowMod` and the last `BarrierReply` as seen on the controller host. This measurement methodology has been validated in detail in [155]. Figure 3.11 shows an exemplary message flow in this measurement scenario in case of two connected switches.



(a) Without TableVisor (b) With TableVisor

Figure 3.11: Message flow for `FlowMod` installation with two switches.



(a) Without TableVisor (b) With TableVisor

Figure 3.12: Message flow for `FlowStatsRequest` response time with two switches.

For all such experiments, 10 repetitions of each configuration were performed, and all displayed results show the mean installation time with a 95% confidence interval. The `FlowMods` used during the evaluation consist of a match on ethertype 0x800 (IP), the TCP protocol as well as a randomly chosen TCP port. Additionally, the priority of the `FlowMod` is chosen at random between 1 and 255.

For the single-response delay measurements, we captured 30 seconds of ONOS' regular operations after installing the `FlowMods`. During this time, ONOS sends `FlowStatsRequest` messages to every connected switch every 5 seconds. For each such request, we measure the time difference between the `FlowStatsRequest` and the corresponding `FlowStatsReply` message as perceived by the controller. Note that TableVisor does not aggregate multiple FlowStatsReplies into a single message, but passes each of them separately to the controller by leveraging the `ReplyMore` flag. Figure 3.12 shows again the exemplary message flow in case of two connected switches with and without TableVisor. Based on all such request-reply-delays observed during the 30-second interval, we evaluate their mean values and 95% confidence intervals.

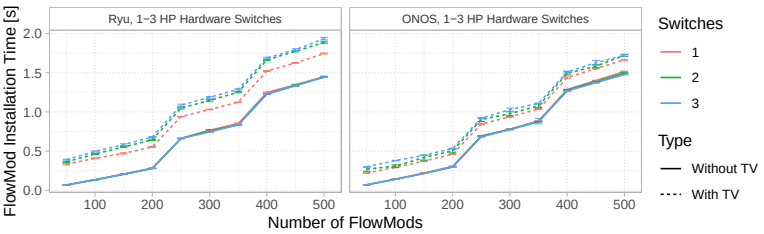


Figure 3.13: FlowMod setup times for 2 controllers (Ryu, ONOS), 1 to 3 hardware switches, with and without TableVisor, and 50 to 500 FlowMods.

3.3.1 FlowMod Setup Times

Figure 3.13 shows the `FlowMod` installation times with one, two, and three hardware switches, using the Ryu controller on the left, and the ONOS controller on the right. On the x-axis, the number of `FlowMods` that every switch receives is given, while the y-axis displays the corresponding setup times in seconds. The color indicates the number of switches used in the experiment. Solid lines indicate the bare controller-switch-scenario, and dashed lines refer to the same scenario including TableVisor.

In both controllers' scenarios, the installation times without TableVisor, as shown by the solid lines, are very similar. They range from 0.1 to 1.5 seconds, and the number of connected switches only has a minor impact on the measurements. The steep increments after 200 and 350 `FlowMods` are expected to originate from the switches' underlying hardware setup, i.e., the time it takes the switches to process the `FlowMods` increases with their TCAM utilization [156]. With ONOS, TableVisor causes an additional delay of roughly 0.2 up to 0.25 seconds in the control plane throughout all switch counts and `FlowMod` numbers. With Ryu, this additional delay shows a slight linear increase with `FlowMods` after connecting multiple switches. This is due to Ryu generating and sending the messages through a single thread while ONOS uses multiple threads to send messages. Figure 3.14 shows the difference of means between scenarios with and

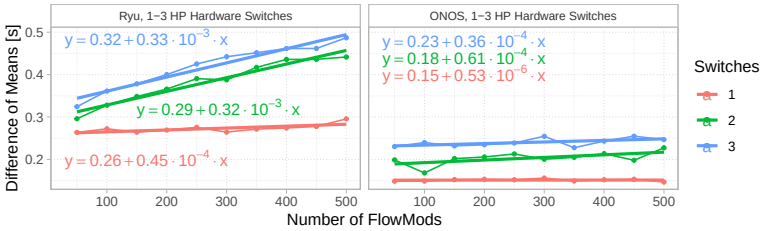


Figure 3.14: Linear regression of difference of means between with and without TableVisor for FlowMod setup times for 2 controllers (Ryu, ONOS), 1 to 3 hardware switches and 50 to 500 FlowMods.

without TableVisor and confirms these observations. The linear regression formulas provided in Figure 3.14 show that, especially when using the production ready controller ONOS, the additional delay caused by the TableVisor proxy layer is dominated by the inherent `FlowMod` installation times of hardware switches, even for higher loads.

The `FlowMod` installation times obtained from the software scenario are presented in Figure 3.15. Hereby, the x-axis shows the number of OVS instances connected to either the controller or TableVisor, and the y-axis contains the respective setup times. The graph is split into groups for each of the both controllers, Ryu and ONOS, and each of the `FlowMod` counts of 50 and 250, while the color indicates whether the switches were connected directly or TableVisor was used.

When using Ryu, the `FlowMod` installation times increase linearly with the number of connected devices, both in the 50 and 250 FlowMods case, peaking at 0.25 and 1.25 seconds, respectively. In low-load scenarios, the use of TableVisor causes an additional latency of up to 0.13 seconds. This overhead is decreasing with increasing load on the setup, and after 15 switches with 250 `FlowMods`, it becomes negligible compared to the bottleneck Ryu introduces. This increment in the performance of TableVisor is likely due to the Java JIT compiler that is able

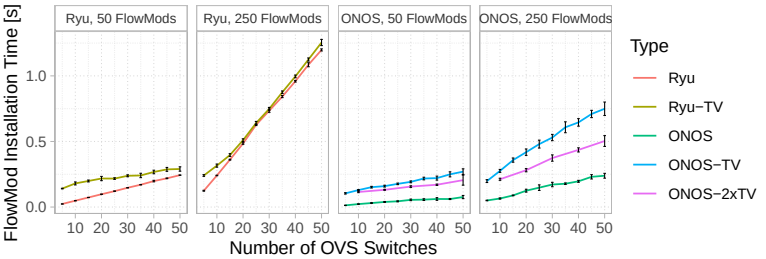
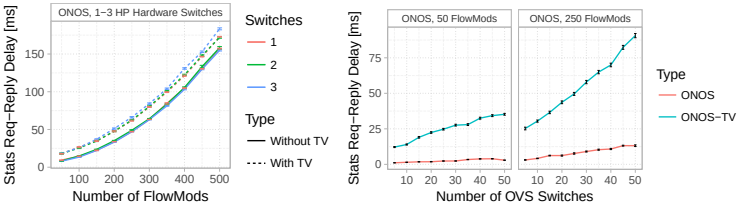


Figure 3.15: FlowMod setup times for 2 controllers (Ryu, ONOS), with and without TableVisor, with 50 and 250 `FlowMods` per switch, and 5 to 50 software switches.

to perform various optimizations at runtime. The results obtained in the ONOS scenarios show a much smaller installation time due to more efficient processing via multi-threading. With 250 `FlowMods`, ONOS setup times peak at only 0.25 seconds without TableVisor, and rise up to 0.75 seconds with TableVisor. This additional delay is not only caused by the processing time of TableVisor, but also by ONOS itself as it generates and transmits the `FlowMod` messages in a much slower pace when connected to TableVisor, presumably due to its internal multi-threading structure as we only present a single virtual switch to the controller. Note that, when the abstraction of large device numbers is desired, the devices can be split into multiple TableVisor instances. Consider the scenario indicated by the purple line in Figure 3.15, in which two TableVisor instances each handle 50% of all connected devices. Here, the installation times peak at 0.5 seconds when both instances handle 25 switches, which effectively halves the overhead introduced by our concept in that case.



(a) Delays with 1-3 hardware switches.

(b) Delays with 5-50 software switches.

Figure 3.16: FlowStats Request-Reply delays for hardware and software switches with the ONOS controller.

3.3.2 FlowStats Request-Reply Delay

Figure 3.16 presents the delays for the individual FlowStats Requests with up to three hardware switches as well as up to 50 OpenvSwitches. In the hardware measurement in Figure 3.16a, the x-axis depicts the number of installed flow rules, while the y-axis shows the measured delay between `FlowStatsRequest` and FlowStatsReply. The latency without TableVisor ranges from roughly 10 ms to 155 ms and is mostly independent of the switch count. The additional delay introduced by TableVisor is steady around 10 ms to 15 ms throughout all investigated `FlowMod` counts for one and two switches, rising much slower compared to the overall latency. With three switches, this difference peaks at 25 ms with 500 `FlowMods`, which equals a 16% overhead compared to the pure ONOS case.

Finally, Figure 3.16b presents the `FlowStatsRequest` delays with an increasing number of software switches. In this case, the y-axis again shows the measured delay of the request-response pair, while the x-axis shows the number of connected software switches. While the pure ONOS measurements appear to be nearly constant for 50 `FlowMods`, they increase linearly from 3 ms to 13 ms when the FlowStats Replies contain 250 `FlowMods` each. The TableVisor measurements show a notable overhead of 12 ms to 34 ms with 50 `FlowMods`,

and 22 ms to 76 ms with 250 `FlowMods`, respectively, peaking at a mean response time of 89 ms with 50 connected switches and 250 `FlowMods`. However, it should be noted that these results apply to this specific scenario, and do not only comprise the processing delay of TableVisor. Albeit the mean latency remains quite stable throughout the different samples, the delays of individual switches vary a lot when using TableVisor. Most notably, it should be considered that all software switches are located on the same host in the Mininet scenario. When TableVisor receives the `FlowStatsRequest` message, it immediately sends 50 copies of it towards the switches. However, when ONOS is directly connected to them, the individual requests are spaced out throughout a certain period, allowing the Mininet host to spread the workload and reply to individual requests faster. If we compare the time difference between the first `FlowStatsRequest` and the last `FlowStatsReply`, similarly to the previous `FlowMod` installation time measurement, we observe 0.175 seconds with TableVisor and 0.202 seconds with ONOS directly connected to 50 switches in the 250 `FlowMod` scenario. Overall, although individual messages receive an overhead delay from TableVisor during such a bulk update, the total time until the controller receives all updates is even shortened by this approach.

3.3.3 Data Plane Overhead

As, in our measurements, TableVisor only interacts with the control plane of OpenFlow devices, the above evaluation does not consider data plane performance. Here, we evaluate the latency introduced in the data plane by concatenating multiple data plane devices into a single pipeline. To do so, we measure the end-to-end delay of the emulated pipeline using a Spirent C1 Testcenter as the traffic generator.

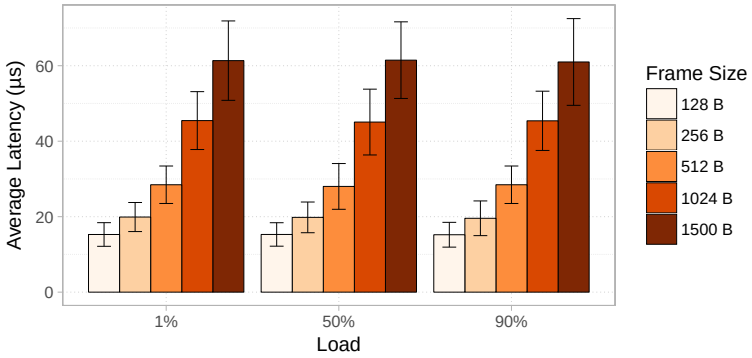


Figure 3.17: Data plane delay of an emulated multi-table switch for different frame lengths and load levels.

Figure 3.17 shows the average delay in a pipeline comprised of four identical HP 2920-24G switches along the y-axis. Different load levels, in percent relative to the maximum of 1 Gbps, and frame lengths are indicated along the x-axis and by differently colored bars, respectively. The whiskers describe 95% confidence intervals. It can be seen that both the delays and the confidence intervals are independent of the load and are constant for all three load levels. The frame length, on the other hand, has significant influence on the end-to-end delay, which indicates that a large portion of the time is used to write the information to the transmission media while the processing of packets is independent of the packet size.

Note that this evaluation is independent of the TableVisor implementation. This particular scenario is similar to the multi-stage ACL deployment presented in Section 3.2.5 to alleviate shortcomings in the devices' capabilities. Although the setup is achieved by our TableVisor proof-of-concept, there is no further interaction after preparing the configuration as the data plane traffic does not trigger actions in the control plane.

In this section, we have investigated both the impact of the additional abstraction layer on control plane performance and on data plane performance. As is intuitively expected, data plane performance is fully independent of the use of TableVisor and is only dictated by the number of devices a packet has to pass through. When it comes to control plane performance, our measurements have shown that depending on the deployed controller application, the overhead induced by our approach varies based on specific configuration, but remains within suitable levels even for large configurations with up to 50 software switches connected to a single TableVisor instance. Based on these observations, we can answer RQ3.3 and conclude that the developed proof-of-concept, while inducing additional control plane delay, operates well within the time scales expected from directly connected hardware devices and scales well with the total number of abstracted devices. A detailed discussion of these results is presented in Section 3.4.

3.4 Challenges and Limitations

In this monograph, we propose the concept of data plane device aggregation and provide a proof-of-concept implementation of our approach. In order to detail required considerations and limitations of TableVisor, we discuss the most important points regarding the operation of TableVisor. Furthermore, we explain current limitations of the implementation as well as general limitations of the approach and describe trade-offs that come with the application of TableVisor. Where applicable, we provide potential mechanisms to alleviate the limitations due to the device abstraction.

Performance Characteristics. The performance with respect to additional control plane delay as well as data plane latency when using our approach has been described in the previous section. However, there are some aspects that need to be taken into account when it comes to real world deployments of TableVisor, including research and development scenarios.

Regarding data plane performance, one has to consider the limitations of all devices involved in a TableVisor emulated switch. As traffic needs to traverse

multiple devices, the total delay to go through the emulated switch will be the sum of individual device delays. This effectively limits the number of stages used in a specific deployment in order to not exceed the data plane delay requirements. However, as observed in Figure 3.17, the relevant timescales are in the order of tens of microseconds. Additionally, in the real world, it can be expected that this issue only impacts very specific use cases, as we have shown in Section 3.2.5 that the size of aggregates can be kept low, while still able to solve complex use cases. Furthermore, the available bandwidth between physical devices needs to be taken into account as the traffic of multiple input ports may need to be passed on to a second stage. However, depending on the use case and the available hardware devices, the impact of this limitation can be alleviated by mechanisms like providing multiple uplink ports between stages as a trade-off between externally available ports and internal bandwidth capacity.

When it comes to control plane performance, the response time needs to be taken into account in addition to the feature set of emulated devices. The response time of messages is, in general, limited by the slowest device in the aggregate. However, TableVisor's use of the `SEND_MORE` flag allows modern controllers to start processing replies before all switches answer to a request, resulting in faster control plane updates.

Finally, it needs to be considered that TableVisor is in many cases not a drop-in solution for already deployed networks. Instead, it is well suited to combat heterogeneity of single devices as well as to provide missing functionality by adding a carefully selected set of devices in combination with TableVisor's aggregation features. However, the actual devices to be combined need to be selected carefully as TableVisor allows the aggregation of nearly arbitrary devices. TableVisor does currently not provide any form of sanity checks or implicit optimization, therefore it is possible to create emulated devices with unexpected behavior, e.g., unevenly sized tables or specialized tables that lack basic features. This needs to be taken into account during deployment and must be avoided either by using a suitable set of switches or worked around on the controller or application side.

Fault Tolerance. When assessing the fault tolerance of a proxy-based abstraction approach, both the failure probability of the proxy itself and the underlying abstracted devices are relevant. As for TableVisor itself, additional components in the system increase the overall failure probability, as it potentially adds another single point of failure to the control channel. To alleviate this issue, TableVisor was designed without the need to track run-time state to enable fast re-initialization in case of software failures. An outage of TableVisor, from both the controller's and the devices' view, appears as a short disconnect of the control channel and no data is lost in this case. Additionally, the stateless nature of TableVisor enables mechanisms like fast fail-over through hot standby to minimize downtime. These mechanisms may be implemented in future extensions of this work. If hosted on similar hardware as the actual SDN controller, TableVisor is subject to the same hardware failure conditions as the controller itself.

As for failures in the data plane, due to the abstraction, problems of individual data plane devices result in the outage of the entire abstracted device chain. This is a limitation of the proposed approach and can not easily be worked around. Potential approaches to alleviate this issue include the use of backup devices or the dynamic deployment of OVS instances to take over from a failed device.

Scalability. Finally, when it comes to the scalability of the TableVisor approach, multiple aspects that are related to the previously mentioned points have to be taken into account. First, the introduction of TableVisor into the control channel affects the control plane performance of emulated devices. This has been evaluated in detail in Figures 3.13 and 3.15. The evaluation has shown that, when using hardware switches, TableVisor induces a near constant offset regarding control plane delay. We have also seen that the behavior strongly depends on the controller used. In general, our measurements have shown that TableVisor scales linearly with the number of devices in the evaluated scenarios with up to 50 switches and 250 FlowMods per switch. We assume that this linear scaling holds true for larger scenarios. However, this needs to be verified by means of additional measurements in the future. However, as with the data

plane delay mitigation, we assume that in many real world applications, like the examples shown in Section 3.2.5, the number of devices will be limited. Hence, the results shown in Figure 3.13 should be considered for real world use cases.

Overall, it needs to be taken into account that the emulation features provided by TableVisor come with certain trade-offs regarding the control plane and data plane performance, and may also impact other aspects of a network such as resiliency and fault tolerance. Some of these can be worked around using the controller, while others can be limited to a certain extent, e.g. by restricting the number of hardware devices used in a single TableVisor emulated device.

3.5 Lessons Learned

In this chapter we presented TableVisor, a transparent proxy layer that allows the emulation of hardware-accelerated data plane devices towards a standard SDN controller. On the one hand, this functionality allows the aggregation of devices towards the controller to simplify its view of the network and reduce overhead, e.g., through topology discovery. On the other hand, it enables more powerful and more flexible use cases through the introduction of hardware-accelerated pipeline processing using multiple data plane devices with differing technologies. In this context, our proof-of-concept enables the integration of P4 hardware into an OpenFlow controlled network. The generic abstraction functionality of TableVisor allows the application of the approach as a tool during rapid prototyping and the emulation of state-of-the-art devices for research purposes in addition to the realization of new, more complex use cases. Hence, we are able to answer research questions RQ3.1 and RQ3.2, and conclude that the integration of multi-technology data plane solutions can be realized using existing mechanisms, protocols and technologies.

We performed an extensive performance evaluation to investigate the impact of our approach on the control plane performance. Measurements involving hardware devices have shown that TableVisor introduces a constant additional delay in the control channel that is independent of the respective workloads.

Extending the evaluation to software data plane solutions has shown that our approach scales well with the number of abstracted devices. Finally, a detailed discussion regarding the limitations with respect to performance, fault tolerance and scalability allows us to answer RQ3.3 and conclude that the concept of data plane abstraction can be exploited to realized highly complex network functions and processing pipelines while maintaining high performance processing.

Our investigation has shown that TableVisor is a suitable tool to realize not only complex new use cases using a combination of hardware devices, but also to support the rapid development promised by the SDN paradigm. In constantly changing and developing networks, the high flexibility provided by TableVisor allows SDN application developers, network orchestrators and researchers to realize use cases that face limitations using single, dedicated hardware devices.

4 Simulation Model for an IoT-focused MVNO Core Network

After the evaluation of single-component software-based network functions in Chapter 2 and the study on data plane abstraction presented in Chapter 3, the final chapter of this monograph covers the performance evaluation of complex multi-component queuing systems. More specifically, we investigate a large scale microservice architecture consisting of a multitude of virtualized cloud services.

However, when considering the analytical or numerical modeling of such systems, the list of constraints quickly limits the practical application of many approaches. Methodologies like Jackson Networks [157, 158] in which all service time distributions need to be negative exponentially distributed and all events need to be processed on a first-come-first-serve basis do provide product-form solutions to open queuing networks. However, these constraints often do not hold in practice. Even with the extensions provided by Gordon *et al.* [159] their application for modern queuing networks remains limited. Further extensions by Baskett *et al.* [160] and Gelenbe [161, 162] do provide solutions under less or more flexible constraints but still require substantial abstractions when dealing with real world systems. Finally, approaches using the renewal approximation in combination with discrete-time analysis, as seen in Chapter 2, can be used to describe complex queuing networks [163, 21].

Due to these limitations, other approaches to evaluate complex queuing networks are required. Especially in the context of modern microservice architectures in which a multitude of heterogeneous components interact with each

other, we reach the limits of the aforementioned approaches [164–166]. Hence, many investigations resort to simulative approaches when it comes to systems of this level of complexity [167–169].

Similarly, we develop a scalable simulation model in order to investigate a complex system of queuing elements comprised of inter-dependent VNFs. The system analyzed in this work represents the core network of an MVNO that has been modeled in close cooperation with the development team of the operator. In this context, both the load profile applied to the system and the individual components of the microservice architecture are evaluated in detail in order to develop an accurate simulation model. Finally, an extensive set of measurements are conducted to validate our model.

More specifically, we first investigate a large scale dataset containing the 2G/3G signaling traffic of over 270,000 unique IoT devices. These data points have been monitored in cooperation with aforementioned MVNO and contain the raw signaling traffic as it arrives at the core network. We perform a detailed investigation of this dataset in order to assess key characteristics of both the aggregated traffic and per device message flows. These findings are then used to define an abstract behavioral model for IoT devices that is subsequently used as a load profile. Based on this behavioral model and detailed knowledge of both the network domain and the core network architecture, we then develop a detailed, simulative model of the network core components and perform a case study on overload control to show both the scalability and accuracy of our simulator.

Based on the classification of the workload and the development and validation of the simulation model, we identify the following research questions.

RQ4.1) Can IoT devices be characterized based solely on their signaling traffic? This includes the investigation of the traffic of single IoT devices as observed at the modeled core network.

RQ4.2) How can the aggregated signaling traffic as seen by the core network by characterized? This includes the analysis of the aggregated traffic resulting from a superposition of multiple IoT device signaling flows.

RQ4.3) How accurate can a system of this complexity be simulated while maintaining practical feasibility? This includes both the accuracy and scalability of the developed simulation model.

RQ4.4) Can our simulation model be applied to answer both questions of dimensioning, such as the detection of bottlenecks, and evaluate possible extensions of the system ahead of deployment? This includes the applications of the developed simulation model and whether the previously validated accuracy can be maintained while extending the simulation through new features.

In order to address these research questions, we provide related work as well as the required background for the remainder of this chapter in Section 4.1. Section 4.2 subsequently covers the analysis of the obtained dataset and lays the groundwork for establishing our device behavior model used later in Section 4.3 as a load profile. In the same section, we also introduce our simulation model. A detailed case study on different overload control mechanisms is conducted in Section 4.4. Finally, Section 4.5 summarizes the lessons learned in the context of this work. The main contributions can be summarized as follows.

C2.1) Development of a simple feature set that allows classification of IoT devices based on their signaling behavior.

C2.2) Identification of key characteristics of aggregated signaling traffic of IoT devices.

C2.3) Development and validation of a detailed, extendable, simulation model of a complex microservice architecture.

These contributions have been published in the past and are condensed in this monograph based on the following scientific publications.

- Geißler, S., Wamser, F., Bauer, W., Krolikowski, M., Gebert, S., Hoßfeld, T.: "Signaling Traffic in Internet-of-Things Mobile Networks," in 2021 IFIP/IEEE International Symposium on Integrated Network Management (IM), 2021. [22]

- Geißler, S., Wamser, F., Bauer, W., Gebert, S., Kounev, S., Hoßfeld, T.: "Simulating Fully Virtualized IoT-centric Mobile Core Networks," Under submission, 2021. [4]

4.1 Background and Related Work

The adoption of the cloud computing paradigm has paved the way for system architectures to become increasingly complex. The possibility of on-demand scaling, both vertically and horizontally, provided the basis for dynamically changing infrastructures. Where legacy systems have been realized by large, monolithic components, modern applications and infrastructures are often built in the form of microservices. These systems are comprised of a multitude of tiny software services that act in unison to perform complex tasks [170]. To achieve this, each component of a microservice architecture represents a highly specialized actor that is responsible for one very specific task. These tasks range from simple queue management, e.g. keeping track of waiting requests, to more complex operations such as decryption and encryption or database queries. In general, microservices are small application that can be deployed, scaled and tested independently of the remaining infrastructure [171]. Through these properties, the microservice concept promises to improve scalability, streamline deployment of new services and reduce time to market while simultaneously increasing reliability [172]. A popular example of the application of these principles is Chaos Monkey by Netflix [173, 174], a software tool that systematically introduces link

or network failures. Thereby, the idea is that the injection of realistic, manageable failures simulates real world operation and helps identify breaking points of a live production environment. Similar concepts have been employed by Amazon and Google [175] as well as Microsoft [176] and Facebook [177]. This adoption of the microservice paradigm by industry giants shows how powerful this technology is in practice.

At the same time, the move from singular, monolithic applications towards a distributed microservice architecture introduces new challenges. Specifically, the complexity of the infrastructure as a whole is essentially moved from the application layer into the networking layer [171]. To address these new challenges, several approaches to assess the performance of single microservices as well as whole microservice architectures have been proposed [167–169, 178–180]. Our work builds upon this research body and introduces a detailed simulation model of a state-of-the-art microservice architecture. We later use said model to explore overload control mechanisms to improve resilience against internal as well as external failures.

4.1.1 Internet-of-Things Traffic Classification

Previous research in the area of IoT traffic falls into two categories. Investigation of traffic characteristics and datasets related to IoT deployments on the one hand and research regarding the signaling efforts of both machine to machine (M2M)-centric mobile networks and common mobile networks on the other hand. Studies investigating the traffic characteristics of IoT devices at a large scale, i.e. evaluating global scenarios including roaming devices, are still lacking from the general literature. Further, while all the following studies investigate traffic patterns of IoT devices, they all focus on data plane traffic. With this work, we aim at filling the gap created by the lack of a detailed model of the signaling behavior of IoT devices.

Regarding research on IoT traffic characteristics, the authors of [181] have compiled a taxonomy of available traffic patterns observed in IoT networks and investigated the applicability of the Poisson approximation in the context of IoT traffic. Further research in the area of IoT traffic modelling compares human generated traffic to M2M traffic [182, 183], proposed source traffic [184] and traffic models [23] as well as classification mechanisms for smart devices using WiFi [185] or general communication networks [186]. Finally, there exist studies analyzing M2M communication [187]. However, all the above focus on data plane traffic. Hence, when it comes to analyzing signaling traffic of mobile IoT devices beyond a single Mobile Network Operator (MNO), to the best of our knowledge, no other studies are available at the time of writing.

4.1.2 Mobile Network Architectures

Several works have been conducted investigating various aspects of mobile core networks and MVNOs in particular.

Bedhiaf *et al.* [188] investigate different MVNO architectures w.r.t their deployment time and overall suitability for virtualization. The authors of [189] perform an extensive, crowdsourced measurement study and investigate aspects of inaccurate billing and performance discrimination by examining user plane traffic flows as well as several external data sources.

When it comes to the simulation of mobile networks, previous research has looked into simulating radio access systems and signal propagation [190, 191]. Metzger *et al.* [192] investigate the performance of a virtual GGSN. Samoilenko *et al.* [193] investigate the impact of the presence of IoT traffic on human communication in LTE environments. However, the simulation proposed in this work is, at the time of writing and to the best of our knowledge, the first protocol level simulation of a mobile core architecture in the context of IoT.

4.1.3 Overload Control for Mobile Networks

Aside from the microservice-specific mechanisms mentioned earlier, overload or congestion control mechanisms mainly date back to the beginning of radio networks. Originally, overload control was done at the air interface or radio access network of mobile networks with radio resource management [194–197]. The challenge here was to efficiently assign subscribers and devices to radio resources of a cell. However, with the introduction of smaller or even cell-free architectures for machine-type communications as well as the new role of virtual operators, air interface based overload control went out of the focus in favor of signaling overload control in the mobile core [198, 199].

Common overload mechanisms for core networks are simple dropping or probabilistic dropping [198]. More advanced overload mechanisms are specifically designed to deal with bursty IoT traffic [198] or leverage explicit signaling [199]. A detailed dissection of overload and congestion control mechanisms for both mobile core and radio access networks is provided by Ferdouse *et al.* [200]. Adaptive and selective overload control mechanisms have been presented in [201]. Here, admission control rejects only specific requests that, for example, lead to bottlenecks or otherwise reduce system efficiency. Similarly, analytical approaches for overload scenarios have been developed in the past. Tran-Gia [202] devised a discrete-time approach based around the remaining unfinished work in a queuing system. Thereby, throttling in the form of rejecting newly arriving events is applied as soon as a specific threshold of unfinished work is reached. Further analytical models are for example presented in [203].

4.1.4 Simulation Methodology

Finally, when it comes to the design and implementation of simulation models, applying the correct methodology is crucial in order to obtain accurate and generalizable results. First, a suitable level of abstraction is required to enable the assessment of relevant KPIs while keeping simulation complexity and runtime in check [204, 205]. To this end, the simulation model presented in this chap-

ter is represented by an event based simulation on protocol level. This means that every single simulated device tracks its own state for the protocol used during mobile signaling. On the one hand, this leads to a substantial amount of signaling events that need to be simulated. On the other hand, this allows the investigation of the interaction between the system and the signaling behavior of devices.

This leads directly to the methodology decision of realizing the simulation in the form of a model-driven simulation instead of trace-driven simulation [206]. Both approaches have been used extensively in the past [207–210]. However, as one of the points of research is the interaction between system and device behavior, and trace-driven simulation essentially dictates the behavior of devices ahead of time, the choice is clear. By developing a model that determines the signaling load observed at the system, how retries are handled or what happens in case of rejected requests, the interaction details can be captured and evaluated under different circumstances.

4.2 Classification of Internet-of-Things Signaling Traffic

Assessing the traffic carried in a communication network is an important task in order to operate a network successfully. In particular, with new types of traffic that have fundamentally different characteristics, such as the traffic occurring in the context of the Internet of Things (IoT) [181, 184–186, 211], those assessments are essential to plan and operate networks accordingly.

With IoT traffic, a number of new characteristics arise in the networks. By 2022, it is expected to have 18 billion IoT devices, 1.5 billion of which use cellular connectivity [212, 213]. These devices span heterogeneous areas such as industrial, healthcare, residential, automotive, sports, and entertainment. Between 2016 and 2022 there is an average growth of 21% per year, driven by new use cases [213]. This growth in numbers as well as the heterogeneity raises the ques-

tion of the scalability of the underlying infrastructure. Further, the characteristics of traffic are fundamentally changing with IoT, especially given the growth in machine-to-machine communications [214]. For 2020 about 41% or 12.86 billion IoT devices are installed as smart home devices [215]. The traffic generated by smart home IoT devices differs from the traffic generated by conventional devices [183]. In general, a mixture of machine-driven and event-driven traffic patterns is expected for upcoming IoT traffic [181].

From a technical perspective, this change in mobile traffic is met by a different handling of IoT devices in the networks compared to previous mobile use cases. Newly emerging IoT MVNOs run specialized service platforms providing worldwide device connectivity by leveraging already existing infrastructure of MNOs [216]. By providing global coverage to their customers through roaming agreements, MVNOs can provide their service without setting up a dedicated physical cellular network infrastructure. This is technically implemented with the IP exchange network (IPX) through which all physical and virtual providers that roam with each other are connected [216]. It is a separate network parallel to the Internet, so that mobile devices around the world can register with a physical, local mobile network, and their traffic is forwarded to the respective provider responsible for their SIM card, in case the provider has a roaming agreement with the given physical network.

This technical setup results in a mixture of traffic with unknown characteristics in the mobile core of MVNOs, since traffic comes from different locations and devices worldwide, different applications. In addition, different customers are being aggregated through the usage of the IPX network [216, 217]. This leads to unknown signaling traffic in the control plane, which results in significant uncertainties for the operation of networks [218, 219]. Overall, technical questions such as the volume of expected signaling traffic, the identification of certain classes of devices, or the identification of possible approaches to system optimization become important.

To provide a first step towards these issues, we aim at characterizing IoT signaling traffic for mobile networks (i) from a mobile network operator's point of view, (ii) at device level, and (iii) for data connection establishment.

A large scale dataset is obtained by monitoring signaling transport (SIGTRAN) as well as 3GPP GPRS Tunneling Protocol (GTP) signaling traffic of over 270,000 IoT devices in cooperation with an MVNO that provides global IoT connectivity through over 500 roaming partners in 192 countries. In this part of the thesis, we dissect the signaling behavior of devices using 2G/3G network connectivity and provide a broad overview regarding the signaling volume for both protocol stacks and a detailed evaluation regarding the occurrence of specific signaling patterns in the case of IoT traffic. Furthermore, we identify features characterizing the signaling behavior of single IoT devices and perform a device classification based on the identified signaling characteristics.

We evaluate the signaling behavior exhibited by devices of different device classes and show that the devices of different classes exhibit statistically significant differences regarding their signaling traffic. Finally, we dissect the arrival process of new data connections as a proxy for system load and present approaches to model the aggregated arrival process of the observed devices. We show that the Markov assumption, widespread in standardization and literature [181, 23, 220, 221], regarding the aggregated arrival process for data connections does not apply in reality by default, but can be restored through additional modeling steps. The Markov assumption should, based on the theorem of Palm-Khintchine, apply here, as the aggregated arrival process is the result of the superposition of a sufficiently large number of independent sources. The methodology used to obtain the results as well as the structure of the following section is shown in Figure 4.1.

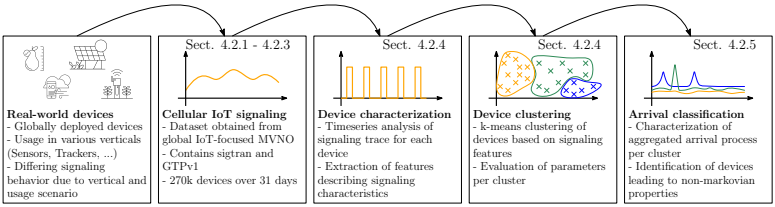


Figure 4.1: Methodology used to obtain the results from the creation of a dataset to the characterization of important key figures for IoT signaling traffic in mobile networks.

4.2.1 Data Description and Processing

The raw data collected in this work consists of all SIGTRAN and GTP messages between Home Location Register (HLR) and Visitor Location Register (VLR) as well as between HLR and Visitor Serving GPRS Support Node (VSGSN) in a roaming scenario. Consequently, all messages belong to the ETSI/3GPP Mobile Application Part (MAP) protocol as part of the SS7 signaling system (ITU-T Q.700-series) or the 3GPP GPRS Tunneling Protocol (GTP) respectively. This includes signaling for *authentication* (SAI), *network attachment* (UL, UL_GPRS), *data connectivity* (PDP_CREATE, PDP_UPDATE, PDP_DELETE) as well as *mobility*.

Table 4.1 provides a list of messages captured and parsed in the scope of this work. The table also shows the corresponding dialog classification based on message sequences.

The messages contained in the raw dataset are assembled into *Dialogs* using Apache Spark. To this end, a state-machine keeps track of the current signaling state of each device throughout the dataset and matches messages belonging to the same dialog. Here, a dialog is defined as a single signaling interaction between serving and home network and each dialog consists of all messages related to the initial request. A dialog is considered finished when the correspond-

ing response has been captured. Hence, in accordance with the specifications of the SS7 signaling system as well as the GTP protocol, the dialog assembly process generates the dialog types presented in Table 4.1.

Table 4.1: Dialog types generated by assembling corresponding messages. op code in brackets. Number of insertSubscriberData Requests $n \in [1, \infty[$.

Dialog	Abbreviation	Contained Messages
sendAuthenticationInfo	SAI	sendAuthenticationInfo Request(56)→ sendAuthenticationInfo Response(56)
updateLocation	UL	updateLocation Request(2)→ $n\times$(insertSubscriberData(7) Request→ insertSubscriberData Response(7))→ updateLocation Response(2)
updateGprsLocation	UL_GPRS	updateGprsLocation Request(23)→ $n\times$(insertSubscriberData Request(7)→ insertSubscriberData Response(7))→ updateGprsLocation Response(23)
cancelLocation	CL	cancelLocation Request(3)→ cancelLocation Response(3)
create pdp context	PDP_CREATE	createPDPContext Request(16)→ createPDPContext Response(17)
update pdp context	PDP_UPDATE	updatePDPContext Request(18)→ updatePDPContext Response(19)
delete pdp context	PDP_DELETE	deletePDPContext Request(20)→ deletePDPContext Response(21)

These dialog types have been selected for further study as they, as well as their corresponding errors, contribute 95% of the total observed traffic volume. All remaining dialogs have been combined into groups labeled *OTHER* in the case of SIGTRAN and *PDP_OTHER* in the case of GTP. These dialogs are mostly related to SMS transmission and interrogation requests, which have not been evaluated in this work.

4.2.2 Dataset Overview

The dataset has been collected between 01.01.2020 and 31.01.2020. Figure 4.2 shows a timeseries over the whole month with the number of million dialogs per hour depicted along the y-axis. It can be seen that the timeseries exhibits gradual growth in signaling traffic over the monitored period. The blue line shows a linear regression with a constant of 0.71 million dialogs and a coefficient of 0.00042 million dialogs per hour. Furthermore, the data exhibits a cyclic pattern with 31 peaks, exactly the number of observed days. On January 8th an operator outage lead to a significant signaling incident, inducing roughly fourfold signaling traffic for about 20 minutes before returning to baseline. The specific reason for the incident in the visited network is unclear. In total, signaling traffic from 346 different mobile networks in 192 countries has been observed during the measurement period.

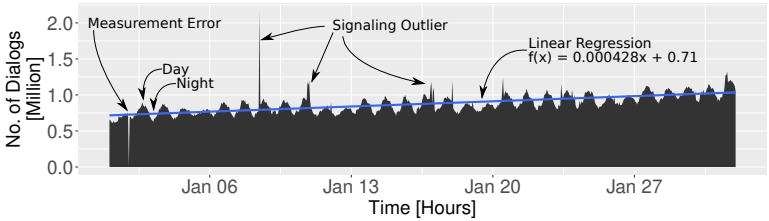


Figure 4.2: Number of signaling dialogs over time in January 2020.

Table 4.2: Dataset overview and key data points.

Devices	Operator	Country	Type	Messages Abs.	Messages Frac.		Dialogs Abs.	Dialogs Frac.	
SIGTRAN									
274,184 (100%)	346	192	Error	14,400,550	0.010	0.587	3,849,272	0.006	0.551
			Reject	198,425,402	0.140		99,212,701	0.152	
			Success	595,878,305	0.419		235,523,346	0.361	
			Unknown	26,165,873	0.018		20,829,090	0.032	
GTPv1									
230,602 (84.1%)	191	152	Error	154,987,535	0.109	0.413	77,593,654	0.119	0.449
			Success	431,698,276	0.304		215,848,695	0.331	
			Unknown	186	0.000		9	0.000	

Table 4.2 presents an overview of key characteristics regarding the dataset. Between the two monitored signaling types GTP and SIGTRAN, roughly 270,000 devices have been observed, 84.1% of which have established at least one GTP tunnel in the observed time frame. In total, devices have generated around 1.4 billion signaling messages, 72.3% of which relate to successful signaling procedures. 14% are rejections by the system, which include disabled SIM cards as well as networks blocked by configuration. 11.9% are related to error responses for establishing PDP context, which include technical errors as well as devices without remaining quota. After assembling the raw data into dialogs, about 650 million dialogs could be identified, 69.2% of which have been successful, 12.5% have failed and 15.2% have been rejected by the system. A more detailed dissection of errors and rejected messages is provided in Section 4.2.3.

4.2.3 Global IoT Statistics for the Dataset

This section summarizes general IoT statistics for the dataset before presenting a detailed decomposition of the dataset. Finally, we present temporal correlations within the dataset and show common signaling patterns that have been observed.

General Statistics

On average, 172,000 unique IoT devices have been observed per day. A device is counted as active if either a successful updateLocation (UL) dialog after authentication or PDP context creation (PDP_CREATE) has occurred. Broken down to hours, roughly 55,000 devices are active on average. In total, all devices generate an average traffic volume of 20 million signaling dialogs per day, 875000 per hour, or 244 dialogs per second within our dataset.

With regard to modeling purposes, the average number of signaling dialogs per day and device amounts to 103 with a standard deviation (sd) of 1,485 and a coefficient of variation (c) of 14.4. The high coefficient of variation indicates high heterogeneity among devices.

Out of the 230,000 devices using data connectivity, on average, a device establishes 21.9 connections (sd: 84.7, c: 3.86) per day. The average duration is 2,890 seconds (sd: 20,205, c: 6.99). The high variation here is a further indicator for highly heterogeneous behavior of the devices active in the dataset.

Errors and Rejected Dialogs

Of the 1.4 billion signaling messages, 27.7% relate to unsuccessful signaling procedures, with errors and rejected messages contributing 12.5% and 15.2%, respectively. An error is defined as an invalid sequence of signaling messages, such as incomplete or out of order interactions. As it is nearly impossible to identify the reason for incomplete dialogs, the observed errors are not evaluated in more detail at this point.

Some dialogs have been actively rejected by the home network. These contain mostly requests from devices equipped with SIM cards that have not been activated, are no longer active or are not allowed to establish data or phone connectivity. In this context, two reasons for rejected dialogs are prevalent.

Inactive SIMs. Of 650 million dialogs, roughly 13.4% are rejected due to devices with inactive SIM cards. These dialogs are generated by 8.3% of the devices (23,002 devices) that have at least one of their dialogs rejected with *Unknown-Subscriber*. Hence, this relatively small number of devices is responsible for 13.4% of total signaling dialogs.

Invalid Roaming Attempts. Accordingly, 11 million dialogs (1.7%), generated by 15.2% of the devices (41,874 devices), are rejected due to invalid roaming partner selection. This occurs if a device selects a visited network that, e.g. due to policy reasons or customer configuration, cannot be used as a roaming partner. These dialogs fail with the message `RoamingNotAllowed`.

Table 4.3: Dialog composition of signaling trace.

Protocol	Dialog Type	Abs.	Frac.
SIGTRAN	SAI	172,618,050	0.26
	SAI_REJECT	87,962,499	0.13
	UL	31,013,855	0.05
	CL	21,343,648	0.03
	OTHER	20,829,090	0.03
	UL_REJECT	11,049,617	0.02
	UL_GPRS	10,547,793	0.02
	UL_GPRS_ERROR	1,726,272	0.00
	UL_ERROR	1,200,272	0.00
	SAI_ERROR	860,920	0.00
	UL_GPRS_REJECT	200,585	0.00
	CL_ERROR	61,808	0.00
GTPv1	PDP_CREATE	93,915,264	0.14
	PDP_DELETE	93,236,365	0.14
	PDP_CREATE_ERROR	77,556,648	0.12
	PDP_UPDATE	28,697,066	0.04
	PDP_DELETE_ERROR	36,009	0.00
	PDP_UPDATE_ERROR	997	0.00
	PDP_OTHER	9	0.00

Sequence of Messages

Table 4.3 decomposes the dataset according to the observed dialog types as well as how much of the total signaling volume each type contributes. The last column shows the arrival rate for each dialog type. Note that arrivals during the incident shown in Figure 4.2 have been removed here.

The table shows the significant portion of messages attributed to inactive devices (SAI_REJECT) as well as invalid roaming attempts (UL_REJECT). Furthermore, it can be seen that a large fraction of errors is related to GTP context cre-

ation. These errors occur mostly due to the GGSN rejecting the device (10.7%) or `APN Congestion` (1%).

Temporal Correlation of Signaling Dialogs.

In order to improve the understanding of device behavior, Table 4.4 shows all dialog sequences observed in the dataset that contribute at least 1% to the total number of dialog sequences. A dialog sequence is thereby defined as a sequence of dialogs without a significant pause inbetween dialogs. More specifically, the timeseries of each device has been divided into bins of one minute with a device being active if at least one dialog occurs within each bin. The resolution of one minute has been selected as it coincides with the timeout for activity in the monitored mobile core. Based on this activity diagram, a sequence is defined as all dialogs occurring in bins with activity without there being a bin without activity inbetween.

Table 4.4 shows that a significant portion of the resulting sequences consists of three or fewer dialogs with the majority only featuring a single dialog. We can also observe that a significant fraction of sequences occurs due to devices closing an already established PDP context, directly followed by the creation of a new tunnel, meaning devices don't establish a tunnel, send data and close the tunnel. Instead, devices close and reestablish PDP contexts, send their data and leave the tunnel open until the next iteration.

Table 4.4: Dialog sequences with at least 1% contribution (73% of total dialogs).

Dialog Sequence	Abs.	Frac.
SAI_REJECT	46,265,888	0.157
SAI	32,634,374	0.110
PDP_DELETE	19,907,081	0.068
SAI_REJECT→SAI_REJECT	17,088,982	0.058
PDP_CREATE	14,012,643	0.048
PDP_DELETE→PDP_CREATE	13,215,440	0.045
PDP_CREATE→PDP_DELETE	8,477,924	0.029
SAI→SAI	8,376,366	0.028
UL	8,059,256	0.027
PDP_UPDATE	6,094,100	0.021
SAI→PDP_CREATE	5,682,473	0.019
PDP_CREATE_ERROR	5,046,938	0.017
UL_REJECT	4,624,730	0.016
SAI→UL	4,315,840	0.015
UL_GPRS	4,051,486	0.014
SAI→PDP_CREATE→PDP_DELETE	3,710,986	0.013
SAI→PDP_DELETE→PDP_CREATE	3,705,860	0.013
SAI→SAI→PDP_CREATE	3,170,721	0.011
PDP_CREATE_ERROR→PDP_CREATE_ERROR	3,123,481	0.011
UL_REJECT→SAI_REJECT	2,937,871	0.010

4.2.4 Device Classification

As already observed earlier, the signaling behavior differs significantly between devices. Hence, in the following section, we establish a set of device features extracted from purely evaluating signaling traffic and show that devices can be clustered using the k-means algorithm. In the feature set, the *error rate* as well as *reject rate* denote the fraction of dialogs resulting in errors or being rejected by

the system, respectively. Further, the grade of *periodicity* of a device is defined as the sum of the autocorrelation values of the three most significant lags observed while calculating the autocorrelation for lags between 1 and 1500 in minutes. Thus, this evaluation is able to identify periods of up to 24 hours for devices with up to three significant periods.[1] Figure 4.3 shows autocorrelation values of a synthetic process exhibiting a strong autocorrelation at lags $15, 30, 45$ and 60. In this example our periodicity metric would equal the sum of the three highest values, hence *periodicity* $P = 0.69 + 0.48 + 0.33 = 1.5$. Note that periodicity values greater 1 can occur, and the metric is bounded between 0 and 3, as potentially all three values could be equal to 1.

Although further features have been evaluated, our investigations have shown that this minimal feature set is enough to classify devices observed in the trace. Note that Section 4.2.5 introduces an additional feature to further refine the clustering performed here.

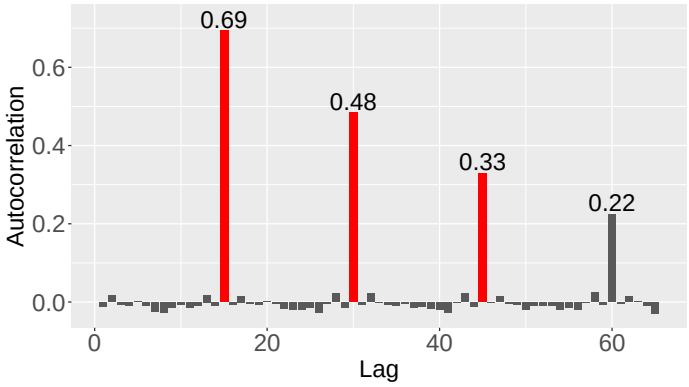


Figure 4.3: Exemplary autocorrelation plot of synthetic, periodic process with period $p = 15$.

[1] Devices with single period p also exhibit high autocorrelation values at $2p$ and $3p$.

Table 4.5: Features used for device classification.

Featurename	Description
Error Rate	Percentage of dialogs resulting in errors.
Reject Rate	Percentage of dialogs resulting in reject.
Periodicity	Sum of autocorrelation of three most significant lags.

Evaluating the within-cluster sum of squares using the elbow-method, $k = 5$ has been decided to be a suitable number of clusters to perform the k-means algorithm on. The resulting clusters correlate with the expected outcome according to expert knowledge contributed by the MVNO. In order to visualize the results of the clustering, Figure 4.4 shows the biplot resulting from the primary component analysis and plotting the two most significant primary components against each other. Each point represents one device and the colors represent the assigned cluster. Additionally, the arrows and labels indicate the influence of the original features on the shown primary components. This visualization allows a visual identification of the relation between features and the resulting clusters. We identify the five clusters as *Non-Periodic* (Green), *Semi-Periodic* (Pink), *Periodic* (Blue), *High Error Rate* (Red) and *High Reject Rate* (Yellow).

Figure 4.5 shows the distribution of devices among those five identified clusters. It can be seen that a significant portion of devices exhibits non-periodic or semi-periodic behavior, with the other classes containing only 16% of devices altogether.

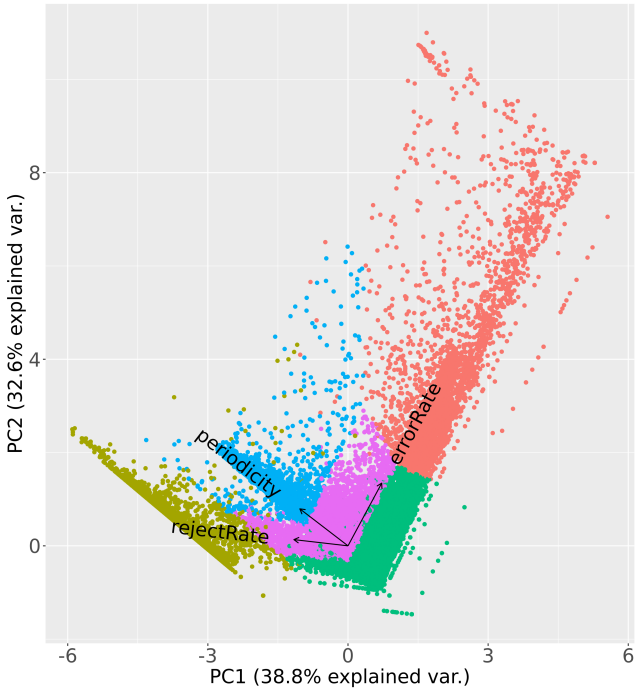


Figure 4.4: Biplot of two most significant primary components and influencing original features.

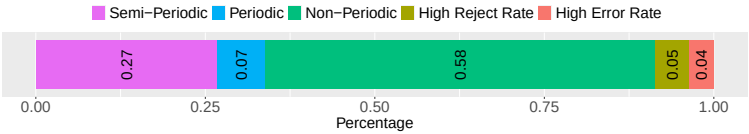


Figure 4.5: Devices classified by signaling behavior via k-means.

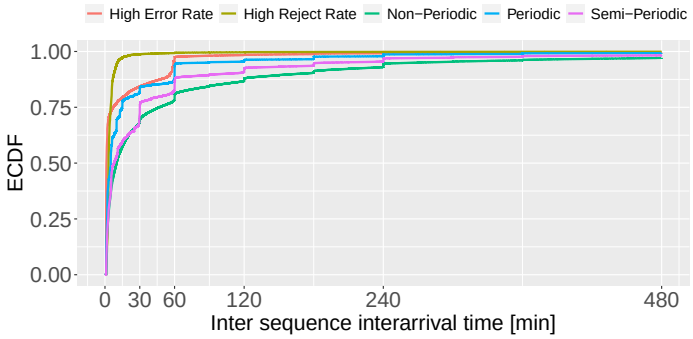


Figure 4.6: ECDF of inter sequence interarrival times by device class.

In order to validate the classification with respect to differing device behavior within the identified classes we compare signaling characteristics between classes. Figure 4.6 shows the empirical cumulative distribution function (ECDF) of inter sequence interarrival times, meaning the distribution of the time between activity phases per device over all devices within one cluster. It can be seen that the different device classes exhibit differing behavior regarding their sequence interarrival times. Periodic devices show clear peaks at interarrival times of 30, 60, and 120 minutes. The two sided Kolmogorov-Smirnov (KS) test confirms statistically significant differences regarding the sequence interarrival times.

Analogously, Figure 4.7 shows the ECDF of the GTP context duration for the same set of device classes. Note that the classes *High Error Rate* and *High Reject Rate* have been omitted here, since their signaling behavior is strongly influenced by their high number of erroneous and rejected dialogs and a comparison to regular devices regarding their GTP context durations would be invalid. Additionally, the y-axis has been limited to $[0.7, 1]$ to make the effect more visible. The KS test once again shows statistically significant differences for each pair of distributions.

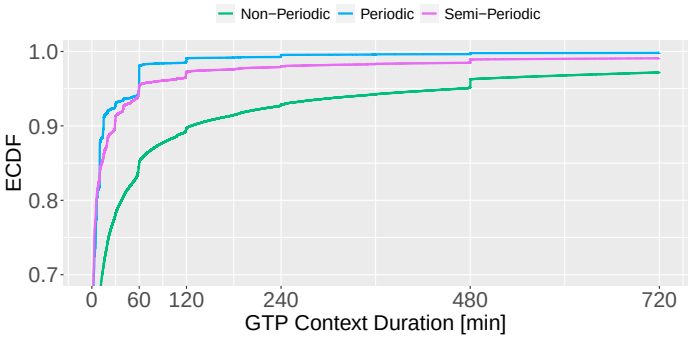


Figure 4.7: ECDF of GTP context duration by device class.

While less prominent, periodic devices again show peaks at context durations 30, 60 and 120 minutes, indicating a correlation between the two values.

Finally, when it comes to the distribution of dialog types used by devices of each class, with the exception of *high error rate* and *high reject rate*, no significant difference between classes could be observed. In the case of the error and reject classes, the fraction of rejected and erroneous dialogs is, by definition, higher compared to other classes.

4.2.5 Aggregated PDP Context Arrival Process

Finally, in order to better understand the underlying arrival process of the devices resulting from the performed classification, we examine the aggregated arrival process of newly established data connections. To this end, we dissect the arrival process of PDP_CREATE dialogs. This subset of messages has been selected as it acts as a proxy for the general system load for MVNOs since a successful PDP_CREATE requires successful SAI, UL, and UL_GPRS dialogs beforehand.

As already stated before, large parts of literature and standardization [181, 23, 220, 221] assume Markov properties when dealing with the aggregated arrivals in large scale IoT environments. The Markov assumption is suitable as the superimposed traffic of an infinite number of sources exhibits memorylessness, according to Palm-Khintchine [222]. In the following, we demonstrate that the assumption does not hold true in reality due to the presence of time synchronous devices, but can be restored through additional device classification and filtering. Similar observations have been made in the past, e.g. [185]. Note that the synchronous behavior of devices is expected to stem from firmware implementation specifics rather than actual synchronization between independent devices. Devices seem to be programmed to transmit data at fixed times, instead of fixed intervals, resulting in pseudo synchronous behavior in the aggregated traffic.

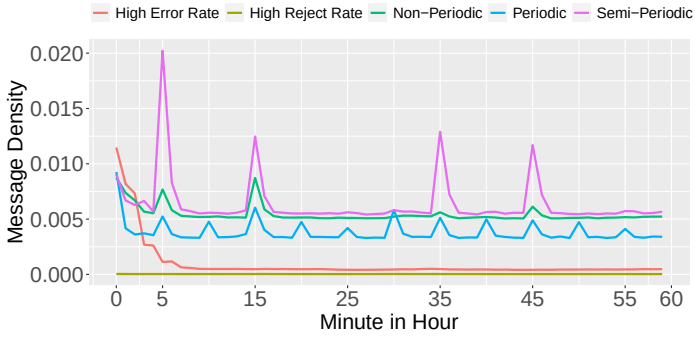


Figure 4.8: Message probability density function of PDP_CREATE dialogs for the probability that messages occur at a specific minute within any hour during the entire trace.

To this end, Figure 4.8 shows the message density over one hour for all device classes for the whole trace. Hence, the y-axis shows the probability of an arbitrarily selected dialog happening within the corresponding minute along the x-axis. The plot is shown to visualize the synchronized behavior of devices within the trace. Although devices are expected to not synchronize their signaling behavior with other devices, clear synchronization patterns can be observed. Specifically, increased density between 0 and 5 minutes as well as peaks at minutes 0, 5, 15, 35 and 45 can be observed. As these devices contribute a significant portion of traffic, this behavior violates the assumption of memorylessness, as in a process exhibiting the Markov property, the message density would need to be constant. Note here that asynchronous, periodic devices would still lead to a memoryless superposition. The issue here is the time-synchronous behavior of devices, violating the independence between devices.

However, based on the message density function of single devices, we are able to identify time synchronous devices and hence divide the arrival process in synchronized and non-synchronized devices. To this end, we examine the maximum message density for messages of a specific device, as is shown in Figure 4.8 for all devices. This classification is based on the assumption that the message density function of a non-periodic, non-synchronized device would follow a uniform distribution. Figure 4.9 shows the ECDF of the maximum message density over all devices with at least 30 activity phases. This limitation is introduced as the density value is not significant for devices with less than one activity per day.

Based on the distinct knee observed in the figure, devices with a maximum message density of at least 0.075 are classified as synchronized. This results in 23% of devices being classified as synchronized. Note that this classification has been performed in addition to the clustering performed earlier, so each device can be classified by both their signaling behavior and their synchronicity. Figure 4.10 shows the same plot as Figure 4.8 with devices split into synchronized and non-synchronized classes.

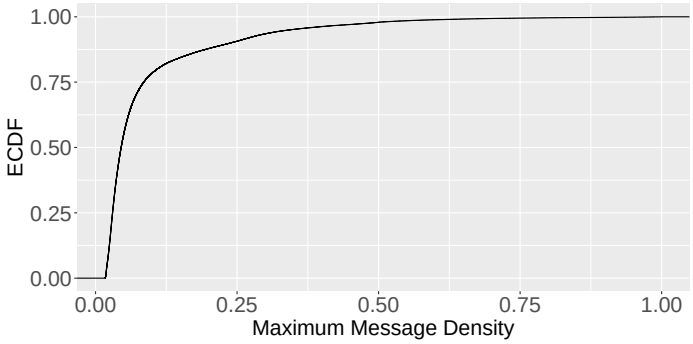


Figure 4.9: ECDF of maximum message density over all devices with at least 30 activity phases.

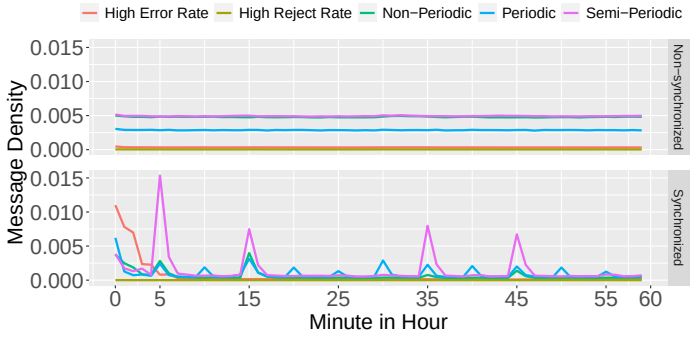


Figure 4.10: Message probability density of PDP_CREATE dialogs over minutes within any hour of the trace per synchronization class.

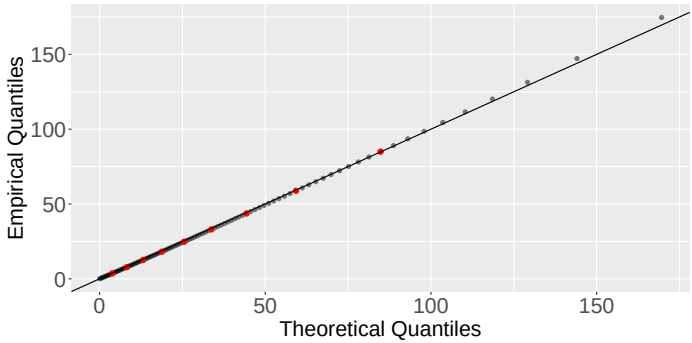


Figure 4.11: Q-Q-plot of interarrival times of non-synchronized devices.

It can be seen that non-synchronized devices now exhibit a uniform message density distribution while synchronized devices feature the peaks observed before as well as low baseline density. Based on these observations we assume the arrival process of non-synchronous devices to exhibit memorylessness and consequently negative exponentially distributed interarrival times. To this end, Figure 4.11 shows the Q-Q-plot of the empirical interarrival times of all non-synchronized devices, irrespective of device class, and the corresponding negative exponential fit. The red marks show the 10% to 90% quantiles, the gray marks show the 1% to 99% quantiles. It can be seen that the interarrival times of the aggregated process of all non-synchronized devices can be closely approximated using an exponential distribution as it is the case for Markov processes. Furthermore, the interarrival times exhibit no significant autocorrelation with the largest observed value being 0.017 for lags between 1 and 1000.

In the preceding sections, we presented the results of our analysis of a 31 day IoT signaling trace containing more than active 270,000 IoT devices of an MVNO operating worldwide. We found that about 84% of the observed 2G/3G devices use data connectivity with the remaining 16% only using network connectivity for phone and text messaging as they do with circuit-switched services. We have shown that

devices exhibit significant differences regarding their signaling behavior and extracted features that allow the modelling of different device classes based on the rate of erroneous and rejected dialogs as well as the periodicity of devices. We have shown that devices identified by this classification mechanism exhibit statistically significant differences when it comes to their signaling behaviour. Based on these findings, we can answer RQ4.1 as we have shown the feasibility of classifying IoT devices based on their signaling behavior.

Finally, by evaluating the aggregated arrival process of new data connections, we have shown that the often assumed memorylessness does, surprisingly, not hold true in reality due to the presence of time synchronous devices, but can be restored through additional classification and filtering steps. Hence, we can answer RQ4.2 by providing means to model aggregated signaling traffic in IoT-focused 2G/3G environments.

4.3 Simulation of an IoT-centric MVNO Core Network

The rapid adoption of Internet-of-Things (IoT) in both industry and everyday life is leading to an ever-increasing number of connected devices. Network operators and providers react to this development with the ratification and implementation of Machine to Machine (M2M) IoT platforms that, in connection with mobile networks, enable global and demand-oriented access of IoT devices to the Internet [217]. Providers invest considerable effort towards the transition from conventional communication to machine-controlled mass device connectivity in order to support the enormous number of devices, open up new profitable business cases, and give companies more configuration options and more flexible use of the mobile networks.

While initial forecasts regarding the number of devices have been overestimating the expected growth [223], more recent predictions expect up to 18 billion deployed IoT devices within the next few years [212]. Of these globally

deployed devices, approximately 10% are expected to rely on mobile connectivity for data transmissions [213]. The volatility in the number of devices and tight economic reasons are encouraging providers and operators to take steps to operate IoT networks efficiently. The ability to mitigate short-term overload and general flexibility, e.g., during booms and downturns with millions of devices, has therefore become an indispensable measure and must be ensured by suitable overload control mechanisms [224]. This aspect specifically, and a general understanding of the scalability, performance and resilience of these IoT platforms in general, are still subject of active research [22, 193, 217].

Recent studies show that IoT devices differ significantly from regular smartphone-based traffic both regarding their signaling and their payload transfer behavior [181, 184–186, 211]. Thus, networks in general and the deployed overload control mechanisms in particular must be able to mitigate short-term load peaks, excessive signaling traffic, or abnormal behavior of machines. Characteristics of machine-type communication become particularly problematic under unexpected circumstances, e.g., in the event of regional or global network failures, connection resets, or general loss of connection. In such scenarios, the reaction of devices is entirely dictated by their programming, firmware or configuration. In many cases, as opposed to humans, devices that lose connectivity will continuously try to re-establish connectivity, thereby triggering workload in the control plane.

This behavioral difference becomes even more evident when taking into account recent developments in the area of MVNOs that run centralized instances of the mobile core without deploying their own radio access network (RAN). Instead, MVNOs rely on roaming agreements with physical MNO, thereby exploiting existing infrastructure in combination with the global IP exchange network (IPX) [216] to carry signaling traffic between visited networks and the mobile core of the MVNO. This setup allows a single MVNO to operate a single, centralized instance of the mobile core and still provide global connectivity to its customers. However, this also introduces multiple points of failure for both MNOs and MVNOs that may lead to extreme overload scenarios. Being depen-

dent on the infrastructure of many, globally distributed physical operators as well as a number of signaling carriers to ensure transmission of signaling traffic between the visited and home network introduces the risk of regional outages resulting in signaling storms [225] that can quickly overload the home network core.

To this end, in this work we introduce a detailed, protocol level simulation framework developed on base of a real world, virtualized MVNO core network. The proposed simulation model contains, on the one hand, a signaling model dictating how IoT devices behave with respect to their signaling message sequence and timeout and retry behavior. On the other hand, we present a detailed MVNO core model describing the processing of received signaling messages by the core network using network and component simulations with real processing times obtained from a productive MVNO mobile core network that processes IoT traffic from 540 radio access networks with IoT devices in 180 countries.

Furthermore, we present a case study of how the proposed simulation framework can be applied to perform bottleneck detection and investigate the impact of resource scaling within the core network. Finally, we present a detailed investigation of various overload control mechanisms that allow the continuous survival of the core network under extreme overload conditions. We identify several performance metrics beyond a mechanism's mitigation capability and compare the suggested mechanisms regarding, e.g., resource utilization and blocking probability from device point of view. The results presented in the following sections help categorize overload control mechanisms and highlight the need for MVNOs to deal with the upcoming IoT traffic by defining appropriate control mechanisms in mobile cores. Furthermore, to best of our knowledge, this is the first protocol level simulation of a real world MVNO core network in the context of IoT traffic.

4.3.1 Technical System Description

As a base for the simulation model, we use a state-of-the-art virtual mobile core implementation that is, at the time of writing, in production and serving upwards of 170,000 unique devices and 21 million signaling dialogs on a daily basis. In the following, we present the key technical aspects of this system and highlight central properties relevant for the development of our simulation model.

The system is designed from the ground up to operate in the cloud, adhering to the cloud native paradigm [226, 227] and the principles of NFV. All involved services are implemented in software and running on Amazon Web Services (AWS). Besides advantages of fine-grained scalability, this dramatically lowers efforts compared to hardware development and enables in-house development of unique selling points without dependency of a large vendor. By leveraging multiple AWS Regions, a globally distributed, multi-layered control plane for GTP has been developed [228] and is in production since 2015.

Leveraging the AWS public cloud ecosystem enables the mobile core system to operate globally and scale virtually limitless to meet the required capacity. The latter, as well as resiliency against component failure, however, require more than just the underlying cloud architecture can provide.

Instead, paradigms from the area of serverless computing [229] are used to allow dynamic scaling, optimize resource utilization and allow easy recovery after component failure. More specifically, all components of the architecture are developed as an actor based model [230, 231] on top of Akka [232]. The advantage of the actor model is a simplification of applications handling a high number of concurrent tasks, in contrast to object-oriented programming that often requires locking of data structures. While actors communicate via immutable messages, every actor processes only one message at a time. Until a message can be processed, it waits in the actor's mailbox. Communication across multiple hosts is transparently and resiliently given by the Akka Cluster extensions.

Figure 4.12 shows a schematic overview of the full system and its environment, starting with IoT devices deployed in the field on the left, the visited network used for roaming to establish mobile connectivity and the corresponding

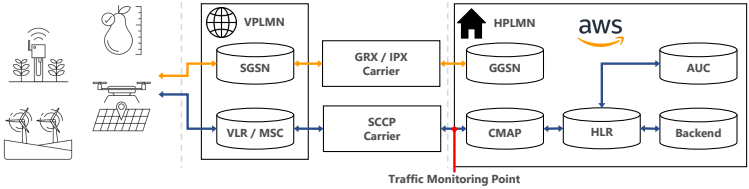


Figure 4.12: Schematic overview of the technical system used as the basis for the simulation model.

signaling carriers in the middle, and the virtual core network on the right. Note that only services relevant for this work are shown in the image. The path taken by GTP-c signaling messages is shown in orange, the blue path indicates the path taken by SIGTRAN signaling messages. The devices on the left as well as the signaling carrier part in the middle are not the focus of this work and are only included for completeness' sake. Instead, we focus on the right part of the figure, representing the virtual mobile core platform as well as its components. Note that this work focuses only on the SIGTRAN processing path shown in blue as well as the four components involved therein. The CAP-MAP Router (CMAP) component presents the ingress and egress nodes for both signaling requests and responses, respectively. It receives messages and translates them into the data structure used by the subsequent components before performing forwarding the request to the next component in the chain. The HLR is, as specified in [233], involved in keeping the state of mobile subscribers and handles signaling interactions related to, e.g., authentication or updating the location of devices. To this end, the HLR delegates tasks to the two remaining components. Authentication Center (AUC) is thereby responsible for the authentication of devices. The Backend represents the interface towards persistent storage and is involved in both querying and updating device profiles. Finally, it is important

to note that both the HLR and the Backend component feature caching capabilities that allow requests to be processed directly without the need for task delegation. Thereby, the HLR caches device profiles for 20 minutes while the Backend cache removes entries after 24 hours. The cache size is thereby only limited by the available system memory.

Each component is realized by reserving a specific number of CPU cores dedicated for the specific component. This resource reservation for individual components ensures that all tasks can be performed at any time without the need for inter-task scheduling. However, within each of the four core network components the actor based programming model realized by the Akka framework creates a serverless pipeline. This means all resources available within each component's pool can be freely distributed among all processing tasks of the specific component.

The implications for the developed simulation and the resulting abstractions inferred from these system properties are detailed in the following section.

4.3.2 Simulation Model Description

After detailing the technical characteristics of the real world system used as a basis for the developed model, this section presents a description of the simulation model. Figure 4.13 shows a schematic representation of the system. The simulation consists of two core components, namely the *signaling model* dictating the behavior of simulated IoT devices and the *MVNO core model* representing the components in the virtual cellular core dealing with processing signaling messages. Both models are based on preliminary studies of a production-ready state-of-the-art virtual core network for IoT devices and applications. The code for all simulation scenarios presented in this work is publicly available on GitHub.[2]

[2] https://github.com/lsinfo3/MVNOCoreSim

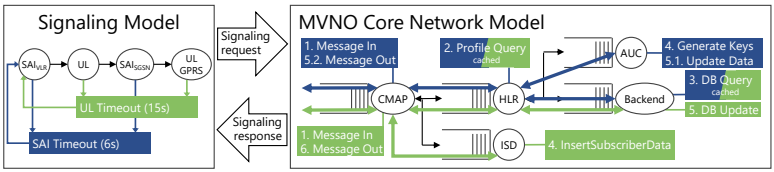


Figure 4.13: Schematic overview of the simulated system consisting of the signaling model and the MVNO core model.

Signaling Model

The signaling model represents an abstract representation of the signaling traffic observed between visited and home networks resulting from roaming IoT devices. It dictates characteristics such as the sequence of signaling interactions as well retry and timeout behavior. Note here that due to the technical properties of the real world system, our signaling model does not actually model signaling messages used by IoT devices, but rather the messages triggered by the VLR and Serving GPRS Support Node (SGSN) of the visited network that are then issued towards the mobile core network, as is the case during roaming.

In order to keep the number of parameters of the simulations conducted in this work in check, we define a singular signaling model that is used for all simulation scenarios throughout this work. The model used is a representation of a scenario in which devices attach to the system, assuming the system holds no prior information about the respective devices. This model has been chosen as it reflects the behavior that occurs after a system outage that leads to loss of connectivity and the subsequent attempt to reattach to the mobile network as well as completely new devices that connect to the system for the first time. Note that in the following, we refer to devices not as physical IoT devices, but abstract objects generating signaling load.

In this model, the VLR starts by issuing a *sendAuthenticationInfo* (SAI) message request used for authentication at the core network. Based on the origin of this message, we call this first message SAI_{VLR}. Upon success, an *updateLo-*

cation (UL) message is issued to update information about the device, such as the visited operator and location, in the HLR database. At that point, the device would be able to make calls and transmit text messages. However, since a majority of devices use data connectivity [22], we assume all interactions also perform the equivalent attach procedure for GPRS connectivity consisting of another SAI message, called SAI_{SGSN}, followed by an *updateGprsLocation* message (UL_GPRS). After successful completion of this sequence of messages, we consider the attachment procedure complete and the corresponding element leaves the simulation.

In addition to the message sequence, the signaling model defines two separate timeout paths, indicated in blue and green in the signaling model part of Figure 4.13. The green path shows the timeout started after an UL message is issued. Based on expert knowledge provided by the operator, if the respective interaction is not completed within 15 seconds, the system assumes loss of connection and returns to the beginning of the signaling sequence. Analogously, the same occurs after not receiving a response to a SAI message for 6 seconds.

Naturally, and as already mentioned, the signaling model used to obtain the simulation results presented in the next sections is only an abstract representation of how the signaling traffic observed at this point in the system behaves in reality. However, the developed simulation tool is explicitly not limited to such abstract models but can also incorporate much more detailed models as well as a combination of various different models to simulate different signaling patterns. Especially taking into account the findings in [22], meaning the combination of Markov arrivals and time-synchronized behavior, this combination of models will be required to obtain workloads that match reality.

MVNO Core Model

The second component of the simulation is the MVNO core model that dictates the interaction between services within the mobile core, i.e. the path messages take through the system, as well as their capacities and processing performance. Furthermore, this component will be responsible for implementing any specific

system extensions or mechanisms, as we will demonstrate later in Section 4.4. The right side of Figure 4.13 depicts the simulated core components. The annotations indicate the processing steps performed during the processing of SAI (blue) and UL and UL_GPRS (green) messages.

In general, the MVNO core model is defined by a network of queuing components, their capacities and processing performance as well as the paths messages take through the queuing network. Table 4.6 lists the processes messages go through on their path through the core as well as their key parameters. Note that the provided capacities are per component and all steps performed by the same component share this capacity.

Table 4.6: MVNO core model components and parameters.

					Processing Time	
Step	Name	Component	Description	Capacity	Mean	Coeff. of Var.
1	Message In	CMAP	Parsing messages	8 Processors	1659 µs	0.07
2	Profile Query	HLR	Get device profile	64 Processors	1739 µs	0.06
3	DB Query	Backend	Get device profile from DB	128 Processors	1877 µs, 70482 µs	0.06, 0.68
4	Insert subscriber data	ISD	Outgoing interaction with visited network	∞ Processors	5998 µs	1.43
5	DB Update	Backend	Update database content	128 Processors	13115 µs	0.25
4	Generate Keys	AUC	Generate keys for authentication	8 Processors	2886 µs	0.04
5.1	Update Data	AUC	Update device data in DB	8 Processors	4960 µs	0.03
6, 5.2	Message Out	CMAP	Transmission of response	8 Processors	749 µs	0.14

The capacities as well as the processing times used for these operations are modeled after the real world system and have been obtained through dedicated measurements. Although only the mean processing time values are presented in Table 4.6, the simulation input actually consists of a vector of processing time samples obtained from the real world system. These samples are also included in the accompanying GitHub repository. Furthermore, all queues are assumed to hold an infinite number of elements, as the real world system is only limited by memory capacity and never actively rejects messages without overload control or congestion mechanisms.

Based on these operations, we can now define the paths of the three signaling messages used in the signaling model. Note that the *insert subscriber data* (ISD) operation scales with an infinite number of processors. This is done as the processing for this outgoing interaction is performed by the visited network which, for simplicity reasons, is represented as a dummy component with infinite capacity, based on the assumption of similar response times independent of the number of requests the MVNO is issuing. This also means that events never have to wait before being processed by the ISD component.

As already detailed in the signaling model, *sendAuthenticationInfo* messages occur in two variants, SAI_{VLR} and SAI_{SGSN} which are processed mostly identical. The blue annotations in Figure 4.13 depict the involved processing steps. Incoming messages are parsed by the CMAP component (1), routed to the HLR (2) where the actual message processing is performed. At this point, the difference between the two types occurs due to caching. Namely, in order to process the initial SAI_{VLR} message the HLR needs to query data from the Backend component (3). Depending on the cache of the database, this step exhibits two different processing time distributions, hence we also show two mean values in Table 4.6. Additionally, the response will be cached at the HLR for subsequent SAI messages belonging to this device, meaning subsequent messages may skip the DB query (3) entirely. Finally, the AUC generates the authentication keys (4) and performs the processing required for authentication (5.1). In parallel a reply is sent via the CMAP back to the visited network (5.2). Note that steps (5.1) and (5.2) are performed in parallel.

When it comes to UL and UL_GPRS messages, the green annotations in Figure 4.13 depict the involved processing steps. Here, it is important to note that, due to the device behavior employed in this work, all UL and UL_GPRS messages generate a cache hit at the HLR (2) and can hence skip the DB query (3). However, the DB Update (5) has to be performed either way.

In this work, both HLR and Backend caches exhibit infinite space and are configured to retain elements for 20 minutes in the case of the profile query operation and 24 hours in the case of the DB query operation.

Finally, in order to keep the complexity of the core model in check, a number of abstractions over the real system have been made during the development of the simulation model. These mainly revolve around the representation of individual components. As is typical for cloud native architectures, the real system realizes each component as a group of dedicated instances, each equipped with a certain capacity, i.e. CPU cores. As an example, the AUC component is realized using 4 instances with 2 CPU cores each for a total of 8 processors. In this setup, dictated by implementation specifics of the services, each instance features its own queue. For the simulation model, we abstract this behavior and just implement a single instance with a single queue and the total number of processors instead. This abstraction will, based on the multiplexing gain in queuing systems, overestimate the performance of the real system.

Further, we introduce an additional abstraction with respect to the processing path that is presented in Figure 4.13. Namely, due to technical limitations, we were not able to obtain accurate processing time distributions for some steps along the processing path. More specifically, some processing steps take such little time, that their processing times are well below the margin of error of time measurements in software environments. Due to these inconsistencies and to avoid overfitting, the simulation model is omitting these processing steps. One example for this is the forwarding of the response of a SAI message by the HLR — between (4) and (5.2) in Figure 4.13. This abstraction will, in general, lead to an underestimation of the response times for these message types. However, as the processing times for the omitted steps are negligible and are within the range of 100 microseconds, the effect on the overall results is expected to be negligible as well.

4.3.3 Validation

Based on the simulation model introduced in the previous section, we now validate the model by comparing results to measurements taken from real systems. We compare the results of a dedicated simulation run with values obtained from a dedicated testing environment deployed in AWS provided by a real MVNO. Furthermore, more validation data was recorded during the productive use by the MVNO and compared with the values from the test system and the simulation.

Simulation Accuracy

We start by validating the processing time values reported by the simulation against both measurements from the testing and the production environment. The results of the productive dataset have been collected in November 2020. The dataset was sampled from all customers from 193 countries and consist of 1 601 442 messages and dialogs. After post-processing, 206 971 message flows could be isolated to reveal the processing times of components in the MVNO core network per dialog interaction. For the testbed measurements, the signaling model was configured to generate new device arrivals with geometrically distributed interarrival times with a rate of 2 devices per second. Both the testing and the simulation environment as well as the production system use the capacities and processing values presented in Table 4.6.

Figure 4.14 shows the response times for both cache misses and cache hits for SAI messages as well as cache hits for UL messages. It is important to note here that, due to the modeled device behavior and time scales, both *updateLocation* and *updateGprsLocation* messages always result in a cache hit, as they require a successful SAI to be issued in the first place, which triggers a cache miss and hence fills the cache. Moreover, the SAI_{SGSN} message, cf. Figure 4.13 also always results in a cache hit for the same reason.

The figures show the response time along the x-axis and the empirical CDF along the y-axis. All subplots show the values obtained from the testing environ-

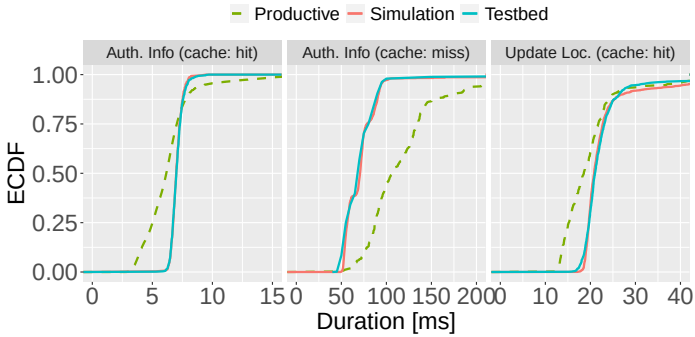


Figure 4.14: Validation of processing times of dialogs in testbed and simulation; additionally, values of productive environment.

ment in blue, from the productive one in green, and the distributions resulting from the simulation in red. The simulated values correspond to the CDFs of the testbed measurements for all measured dialogs. It can also be seen that the values from the productive system are of a similar order of magnitude, but have slightly different values. This is due to the different load and configurations in Amazon AWS compared to the testbed. Note that the measurement values shown for the productive system may contain waiting times, as the measurement points include those. The testbed values have been obtained for load levels that effectively eliminate waiting, hence isolating the service times.

Furthermore, the Kolmogoroff-Smirnoff-Distance (KSD) and the Jensen-Shannon-Divergence (JSD) have been derived between the testbed and simulation distributions to better show that the CDFs match each other in numbers. For SAI_{VLR} the one-sample KSD with the simulation samples is 0.097 with a p-value of 0.94, meaning there stands nothing against the hypothesis that the samples are drawn from the same distribution. The JSD is at 0.04 for SAI_{VLR}. For SAI_{SGSN} with cache hits the KSD is at only 0.030 with a p-value of 0.99, the JSD assumes 0.004. The KSD for *updateLocation* is at 0.064, JSD with 0.0251.

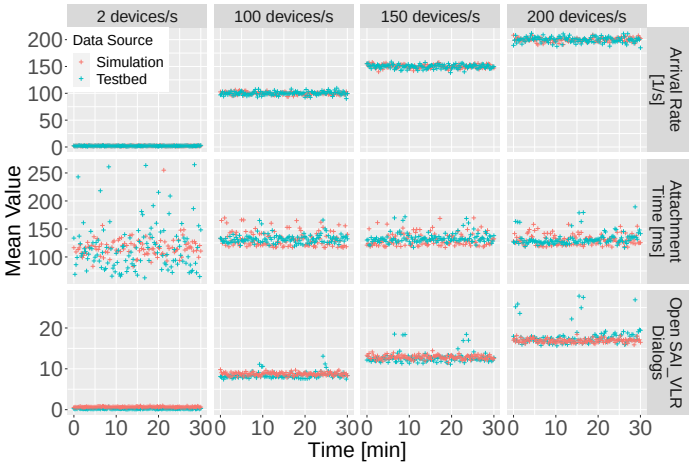


Figure 4.15: Validation of configured arrival rate, attachment time and number of open SAI_{VLR} dialogs.

Figure 4.15 highlights three central KPIs and compares the results obtained via simulation to values observed in the testing environment. Thereby, horizontal facets indicate the configured arrival rate of new devices while vertical facets state the evaluated metric. Each facet shows the mean metric value obtained for 15 second intervals over time for simulation and measurement runs of 30 minute duration. Note that the simulation time has been set to 40 minutes and the first and last 5 minutes have been trimmed to obtain stationary results. Similarly, the figure shows a 30-minute snapshot of a measurement run from the testbed.

The first row depicts the monitored arrival rate of new devices and shows that both the testbed and simulation have been subjected to a similar load between 2 and 200 arrivals per second. More interesting, the second row shows the attachment time of devices, meaning the time it takes for each device to complete their attachment cycle as described in Section 4.3.2. It can be seen that the simulation results align with the observed measurement values for arrival rates of 2, 150

and 200 devices per second when it comes to the magnitude of values. The peaks observed in both the attachment time and the open SAI_{VLR} portion of the figure are explained by fluctuations of the testbed performance due to it running in a public cloud environment. More specifically, a closer investigation has shown that the database underlying the Backend component took longer to respond to queries. Unfortunately, due to the nature of the monitoring tool used in the testbed, it is unknown if single large outliers or a general increase in response time occur at these points in time. In order to not overfit this specific system, the simulation neglects these fluctuations. However, such changes in processing power can be reflected in the simulation by modeling the processing times of components as a function of time, if desired. In fact, even the dynamic adaptation of CPU capacities could be realized with minimal work. The fluctuations observed for 2 arrivals per second are explained by the low number of samples in each 15 second interval. Similarly, the last row of Figure 4.15 shows the mean number of open SAI_{VLR} dialogs. A dialog is thereby considered open as long as it is being processed by or waits for processing by any of the core components depicted in Figure 4.13. Again, the figure shows a close match between simulation and measurement values for all load levels over the whole run.

Based on these observations as well as the dialog level response times evaluated in Figure 4.14, we can conclude that the proposed simulation model provides a close representation of the real world system when it comes to the KPIs reported by the simulation.

Simulation Functionality

Further, in order to show the correctness of the implementation of the signaling model as well as the path of messages through the core on a functional level, Figure 4.16 shows a waterfall plot of an exemplary device that represents the timeline of processing steps of this device along the x-axis. The y-axis shows the different signaling dialogs used by the device. The colored bars indicate the resource that is occupied by this specific device at the specific point in time. The vertical lines indicate special events such as cache hits and cache misses.

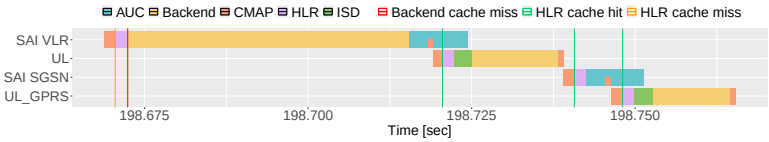


Figure 4.16: Waterfall plot of an exemplary device and its signaling behavior.

The figure shows the sequence of occupied components for each signaling message. For example, the SAI_{VLR} shown in the first row arrives at the CMAP component, is then processed by the HLR, where a cache miss occurs. Based on that, a database query is triggered and the Backend component gathers the required information before the AUC starts generating the entries required for authentication. Finally, the CMAP component transmits the response while the AUC simultaneously updates database entries. Note the missing occurrence of the HLR between the Backend and the AUC, that is omitted in the simulation due to the reasons elaborated earlier in Section 4.3.1. Analogously, the other signaling interactions are shown in the further rows of the plot, each of which behaves as described in Figure 4.13.

4.4 Case Study - Dimensioning and Overload Control

After demonstrating the close fit between the real world system and the simulation model, this section presents a case study evaluating different aspects of the system to show the capabilities of the developed simulation model. We start with a bottleneck detection and show the possibility of scaling individual core components. Following, we develop and compare different mechanisms that can be applied to mitigate system overload and ensure system survival under overload conditions.

4.4.1 Bottleneck Detection

In order to establish a baseline for the maximum load the system can handle without failing and in order to detect which component acts as the bottleneck, we perform 10 simulation runs with a duration of 5 minutes for an increasing arrival rate of new devices. The first 60 seconds of each run have been trimmed from the data to remove the transient phase and let the system reach a steady state. For these runs, we evaluate the mean load for each of the MVNO core components, as described earlier in Section 4.3.2.

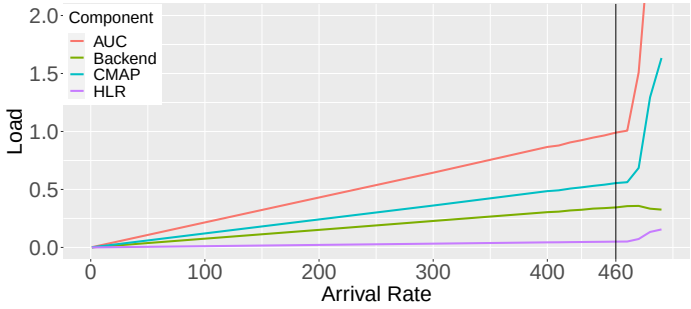


Figure 4.17: Component load under increasing arrival rates of new devices.

Figure 4.17 plots the arrival rate of new devices in arrivals per second along the x-axis against the encountered load along the y-axis. The colors indicate the core services as introduced in Section 4.3.2. Note that confidence intervals have been omitted here, since the variation between simulation runs is negligible. We identify the load level at which the first core component exceeds its maximal processing capacity as the baseline for the following scenarios. The vertical indicator line shows the highest arrival rate that leaves all components with a system load below 1. The component with the highest load in this case, and hence the bottleneck, is AUC with a load of 0.99. The figure also shows different behavior of components when increasing the arrival rate over the maximum

value of 460 arrivals per second. This spike is explained by the device behavior introduced in Section 4.3.2. As, due to overload, the queues of components start to fill and devices start to run into timeouts resulting in an infinite retry loop. This leads to an additional increase in signaling messages over time, as more and more devices try, and fail, to attach to the system. Due to this behavior, the system enters a *livelock*, as all components work at maximum capacity, but due to the long waiting time, devices retry before the system can respond.[3] Note further that the absolute values shown in the plot are to be considered as an indicator for overload, as the system is unstable for these load levels and queue sizes grow with increasing simulation time. At the same time, the Backend component exhibits a decline regarding its load. This is due to the fact that the Backend service is only involved in processing UL and UL_GPRS messages. However, since the AUC is already overloaded, devices fail during authentication and the rate of UL messages drops.

4.4.2 Resource Scaling

After identifying the maximum rate the system can handle without running into overload at a rate of 460 new devices per second and identifying the AUC as being the bottleneck holding back the system, we now rescale the AUC component from 8 processors as shown in Table 4.6, to twice the capacity with 16 processors and repeat the bottleneck detection regime for the rescaled system.

Figure 4.18 again shows the load of individual core components. It can be seen that rescaling the AUC has shifted the bottleneck to the CMAP component as now this is the first component to exceed a load of 1 at 850 new arrivals per second.

Based on this, for the remainder of this work, we consider an arrival rate of 840 the highest load without running into overload and an arrival rate of 850 the lowest load that already leads to overload behavior.

[3] *Note:* This behavior can also be observed in the real production system if no measures are taken.

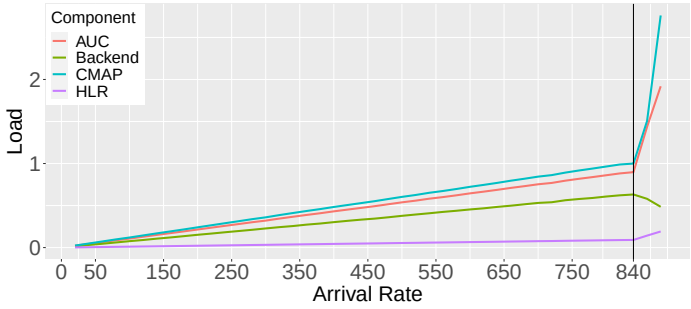


Figure 4.18: Component load under increasing arrival rates of new devices with rescaled AUC.

4.4.3 Dedicated Overload Mechanisms

As resource scaling is only feasible up to a certain point, we now demonstrate the capability to evaluate specific mechanisms regarding the modeled system by defining and implementing various overload control mechanisms that allow the system to keep working while under severe overload. Table 4.7 summarizes the mechanisms evaluated in the following.

Using the information obtained earlier, we can subject the system to various levels of overload by increasing the arrival rate of new devices beyond the identified rate of 850 devices per second. To this end, we first introduce a set of three different overload mechanisms and compare their performance regarding system as well as device parameters.

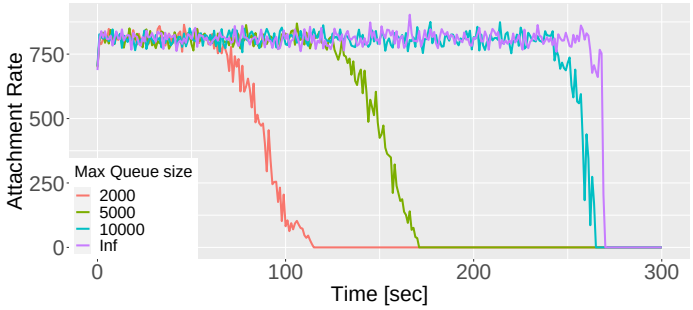
Simple Dropping The first mechanism employed and examined here, *Simple Dropping*, aims at reducing the total waiting time of messages and thereby reducing the probability for timeouts that in return lead to a retry by the device. To this end, we limit the maximum queue size of each component in order to drop messages that can not be processed in time, thereby limiting the amount of unnecessary work performed by the system.

Table 4.7: Summary of overload control mechanisms.

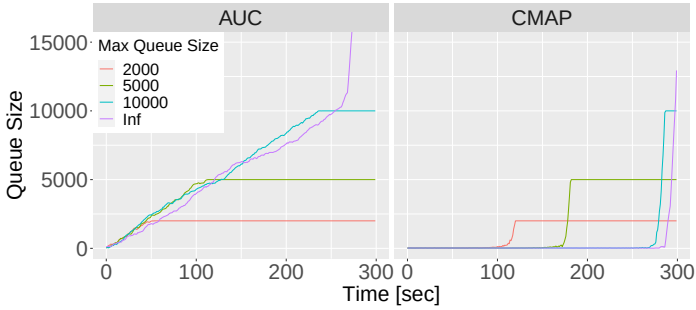
Name	Queueing	Dropping	Prioritization	Explicit Signaling
Baseline	✔	✘	✘	✘
Simple Dropping	✔	✔	✘	✘
MSU Policing	✔	✔	✔	✘
Device Policing	✔	✔	✔	✔

Figure 4.19 shows the attachment rate as well as queue size for the AUC and CMAP components over time for an arrival rate of 850 new devices per second. Note that the queue size of other components is omitted here, since it remains zero over the whole duration. The attachment rate in Figure 4.19a represents the total number of devices that are able to successfully complete their attachment cycle per second. The colors indicate the respective configuration of the maximum queue size, with *Inf* indicating the baseline case with an unlimited queue size. Note that no transient phase has been trimmed for these plots as the system never reaches steady state. It can be seen that, for the baseline scenario, the rate stays largely linear around the 840 devices per second until the queues fill up to a certain threshold, after which the aforementioned retry livelock occurs and the attachment rate quickly declines to zero. When looking at the baseline curve in Figure 4.19b, it can be seen that at the point of the decline of the attachment rate, the queue size of the AUC component exhibits exponential growth, as devices start to retry in addition to the arrival of new devices. Eventually, close to the 300-second mark, the load on the system even exceeds the capacity of the CMAP component and its queue starts to fill up as well.

Moving to the simple dropping scenarios shown in red, green and blue, it can be seen that the limitation of the maximum queue size has no positive effect on the survivability of the system. In fact, the opposite occurs and the system



(a) Attachment rate

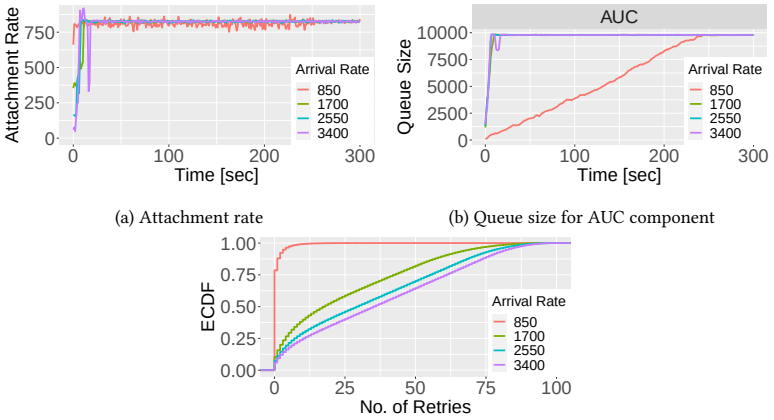


(b) Queue size for AUC and CMAP components (y-axis limited to 15,000)

Figure 4.19: Performance Metrics over time for different maximum queue sizes.

enters the retry livelock even sooner, depending on the configured maximum queue size. This can be explained by the fact that, in order for devices to not issue retries, it is required to successfully process the four previously mentioned signaling messages. However, as the system is overloaded, the maximum configured queue size is eventually reached and a significant fraction of messages gets dropped. Hence, the probability of four consecutive messages of a single device not being dropped is close to zero, leading to the exact same behavior observed in the baseline scenario.

MSU Policing The logical evolution to the simple dropping mechanism, *MSU policing*, limits the number of messages that are allowed to be processed concurrently. Thereby, each message type m is subject to its own quota Q_m, meaning that at one point in time, a maximum of $Q_{SAI_{VLR}}$ messages may be processed, while at the same time a maximum number of Q_{UL} may be in the system as well. Note that 'being processed' here also includes messages currently waiting at one of the core components. This mechanism allows the system to perform a form of prioritized dropping, as it can reject messages of new devices, while still processing messages of devices that have already started their attachment cycle in time.



(a) Attachment rate

(b) Queue size for AUC component

(c) ECDF of retry count per device

Figure 4.20: Performance metrics over time when applying MSU policing.

Figures 4.20a-b) again show the attachment rate as well as queue size of an exemplary simulation run. Note that the queue size for the CMAP component is omitted here since it remains zero over the simulated duration. Further note that the confidence intervals of the steady state attachment rate over 10 simulation repetitions are once again negligible with values of 2.5 for an arrival rate of 850

and lower than 0.7 for all other arrival rates, respectively. We hence only show one exemplary simulation run. The color now indicates the arrival rate of new devices, as we increase the arrival rate beyond the identified overload threshold of 850 devices per second to stress the system even further. Figure 4.20a shows that the attachment rate, after an initial transient phase, remains constant at around 840 devices per second, even after the queue of the AUC component has filled (cf. Figure 4.20b). The quotas Q_m for all message types have been set to 5,000 messages in this scenario as a queue of 5,000 SAI_{VLR} and 5,000 SAI_{SGSN} at the AUC component allows the system to still process all waiting messages in time. In combination with the prioritized dropping, the UL and UL_GPRS messages of devices can also be processed in time. However, additional authentication messages beyond the threshold of $Q_{SAI_{VLR}} = 5000$ get rejected. This leads to near optimal resource utilization, as all messages processed by the system eventually lead to a successful attachment. Other combinations the Q_m quotas have been evaluated as well and have resulted in similar behavior, which is why only one parameter set is shown here.

Additionally, Figure 4.20c shows the ECDF of the number of retries a device has to perform before eventually being accepted for processing. Intuitively, the number of required retries grows with the arrival rate and the highest observed retry count was 100.

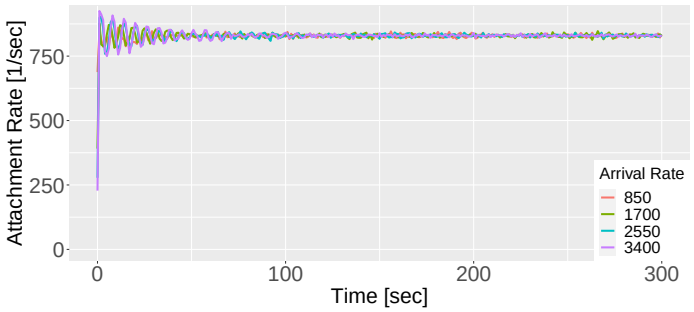
Device Policing with Explicit Congestion Signaling Similar to the MSU policing mechanism, device policing also artificially limits the number of messages allowed to be concurrently present in the system. However, instead of tracking the number of messages, the mechanism keeps track of the total number of devices having at least one message in the system. Hence, the quota Q_D dictates the number of devices allowed to have messages currently being processed by the system. In addition, we now introduce a mechanism that is able to explicitly notify devices about system congestion and can suggest a retry window based on currently unfinished work. Note that a mechanism like this would require an extension of the protocol stack to support the explicit conges-

tion notification. Here, whenever Q_D is exceeded, the system rejects all messages of devices not currently being processed and notifies them about a point in the future at which a retry should be attempted. By this, the system gains more fine-grained control over the arrival rate of messages and can optimize resource utilization without excessive blocking of messages. Note that this scenario introduces changes to the signaling model introduced earlier. Instead of pure timeout-driven retries, devices are now able to dynamically change their retry behavior. Also note that a mechanism like this would require an extension of current protocols and assume devices that respect the feedback of the system. Especially the latter is unlikely to hold true in the real world. However, we still include this mechanism as a potential solution that optimizes both system utilization and message reject rate, thereby reducing the load on both the visited network and the signaling carrier in a roaming scenario. Furthermore, this can lead to reduced energy consumption for the IoT devices as excessive retries are reduced.

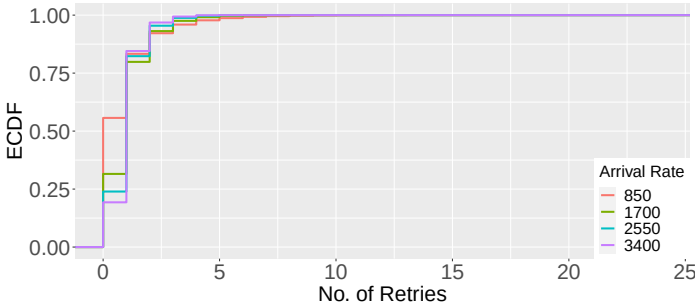
In the scenario presented here, the system is configured to propose retry times based on the fact that it can process 840 devices per second without resulting in overload. Hence, the timer t_r suggested to devices is computed as

$$t_r = \lceil \frac{n_u}{840} \rceil$$

with n_u being the number of unattached devices actively trying to connect to the system. This results in an arrival rate of exactly 840 devices per second when only taking into account retrying devices. In addition, the system has to deal with new devices. However, as the source for the overload is expected to be resolved eventually, and we want to ensure maximum resource utilization, a guaranteed arrival rate of 840 devices will yield the best resource utilization as it guarantees a high system load without resulting in overload.



(a) Attachment rate



(b) ECDF of retry count per device

Figure 4.21: Performance metrics over time when applying device policing with explicit signaling.

Figure 4.21 again shows the attachment rate as well as the ECDF of the number of retries for different arrival rates of new devices. The confidence interval of the steady state attachment rate over 10 simulation repetitions is below 0.75 for all evaluated arrival rates. Thus, we again show one exemplary simulation run. It can be seen that the attachment rate is as consistent as in the MSU policing scenario, while the number of required retries is significantly reduced, as the maximum observed value has declined from 100 to now 25 with the 99% quan-

tile being only 3 in the case of 3400 arrivals per second. However, the time to attach, meaning the time between the first message attempt and the successful completion remained similar with a mean time to attach of 236 seconds for MSU policing and 229 seconds for device policing with explicit congestion signaling.

In order to improve our understanding of the behavior of IoT-focused mobile core systems, we have presented a protocol level simulation framework of a real world MVNO mobile core and have shown that the proposed simulation is able to accurately reproduce several KPIs of the underlying real world system. The implementation used in this work is freely available on GitHub and can easily be extended to include more complex device models or investigate other aspects beside the ones examined in this work. We can hence conclude that we are able to faithfully model the behavior of such a complex system using our simulator and thus answer RQ4.3.

We have shown the source of the issue in overload scenarios within the retry behavior of devices and proposed two different solutions to ensure consistent recovery from overload phases. Both MSU policing and device policing have been shown to ensure high resource utilization by allowing the system to operate at maximum capacity, even under sustained overload. We were furthermore able to identify bottlenecks regarding the scaling of the system and were able to evaluate aforementioned overload control mechanisms. Thereby, we can answer RQ4.4 and conclude that our simulator can be used to perform case studies regarding possible extensions to the system and can furthermore be applied to identify scalability issues under varying circumstances.

4.5 Lessons Learned

In this chapter, we present both our analysis of a large scale dataset containing 2G/3G signaling traffic of over 270,000 IoT devices that has been obtained and analyzed in close cooperation with a MVNO. In this context, we have first shown that it is feasible to characterize IoT devices based on the signaling behavior observed at the mobile core. Furthermore, we have shown that a classification of IoT devices based on a simple feature set is possible and devices

can be distinguished based on their signaling traffic (RQ4.1). Finally, we have investigated characteristics of the aggregated stream of signaling messages arriving at the mobile core and have established that the Markov assumption does not necessarily hold in practice, despite having a virtually unlimited number of traffic sources. However, we have also proposed additional filtering and processing steps to reestablish the Markov property by distinguishing between synchronous and non-synchronous devices (RQ4.2). This observation has been applied to develop a simplistic device behavior model to apply as a load profile for our mobile core simulation model developed in the second part of this chapter.

To this end, we have designed and implemented a detailed simulation model that allows the evaluation of several key performance indicators of the mobile core network. We perform a broad set of simulation runs and validate the simulation results against measurement conducted in a dedicated testing environment hosted and operated by the MVNO. By comparing the results we can conclude that our simulation is able to faithfully represent the core network and replicate critical key performance metrics such as the attachment rate of incoming devices, the number of concurrently open signaling requests as well as the time it takes new devices to attach to the system (RQ4.3). Further, we have conducted a case study and investigated multiple approaches of overload control to ensure system survivability in the event of extreme overload. Our results show that, based on the applied device behavior model, prioritized dropping in combination with message type specific queue size limits allows the system to maintain optimal efficiency even under extreme overload of up to four times its maximum processing capacity (RQ4.4).

5 Conclusion

Today, our everyday life is inextricably linked to the ubiquitous usage of technological devices. Not only are we using smartphones, smart televisions or digital assistants during our leisure time and around our living spaces, but also continuously interact with a vast array of heterogeneous devices and applications during our professional life. In both areas, basically every application, use case and intended usage pattern is designed around the availability of reliable and fast network connectivity. Applications like video streaming, video conferencing, cloud gaming, but also emerging platforms such as the Internet-of-Things (IoT) or Industry 4.0 continuously drive the need for faster, more resilient, easier to manage and more flexible network architectures. The importance of the widespread availability of broadband access became especially clear during the COVID-19 pandemic that started in 2020. Within days, large parts of the working population all around the globe started working remotely due to travel restrictions and work-from-home orders. Luckily, wide area as well as access networks were able to handle the sudden spike in usage. This can, in no small part, be attributed to the efforts of operators, industry and academia, who constantly strive to improve network efficiency and resiliency and develop concepts to support the growing resource demand of new users as well as applications with increasing complexity.

One of these efforts, the softwarization of networks, is also covered in this monograph. Specifically, as the title of this thesis states, we focused on the performance evaluation of next-generation data plane architectures and their components. The performance evaluation of network components and distributed systems has been an established research area for many years and contributions

towards the improvement of network performance have been made plenty in the past. However, due to the speed at which network architectures evolve and the number of new, groundbreaking advancements achieved in recent years, we were able to identify a broad yet specialized set of research questions that were not answered in literature before. Approaches and methods of simulative and analytical system modeling were combined with extensive measurement studies to investigate a wide range of research topics related to the performance of software-based network functions. More specifically, we were able to answer the research questions outlined in the introductory section of this monograph as follows. Note that a more detailed discussion of research questions as well as the related contributions can be found in the respective chapters.

Starting in Chapter 2, we answered the question of how to monitor the processing performance of software-based network functions by proposing a novel monitoring mechanism that exploits the processing done by the network stack used by VNFs. To this end, we designed the monitoring concept itself and implemented a proof-of-concept that has been published as an open-source tool. The proposed mechanism, called in-stack monitoring, leverages the already available code infrastructure of the network stack to intercept timestamps of incoming and outgoing messages and compute the processing time of packets. We validated the accuracy of our approach through measurements in a dedicated testbed using an industrial grade traffic generator. Finally, we discussed limitations and outlined application scenarios of the in-stack monitoring mechanism. We have shown that our approach can be applied to obtain highly accurate packet processing times while remaining network function agnostic and inducing only minimal overhead. Even more, through the exploitation of already existing packet parsing infrastructure provided by the network stack, the approach can be transferred to basically any arbitrary networking implementation that is capable of parsing and processing packets. Thus, the concept itself, supported by our proof-of-concept implementation, is a valuable tool for operators, industry and academia and builds the basis for future research in the area of performance monitoring of software-based network functions.

We answered the question of how to predict critical KPIs of software-based network functions by developing a discrete-time model of a state-of-the-art software router. To this end, we applied our developed monitoring mechanism to obtain a dataset of the processing performance of said software router under different load levels as well as configurations. We developed a suitable abstraction and defined a discrete-time queuing model that allows the prediction of critical performance metrics, such as the waiting time, queue size and packet loss probability. Using parts of the measurements during development and parts during validation, we showed that our model is capable of accurately predicting the relevant performance characteristics based on easily obtainable input parameters. The generalizability, as evaluated for two different types of network functions using the same underlying framework, makes the developed model a valuable asset for both practical problems, like dimensioning or performance prediction, and future research. By providing a point of reference for the performance of network functions, our model can be used to evaluate performance metrics of network functions that will be developed in the future. Simultaneously, the model can be practically applied by operators, developers and academia today in order to assess and predict the performance of systems without the need for physical deployment. This relevance in both practical and research areas as well as current and future developments highlights the value of a generalizable, analytical model. In combination with the previously discussed monitoring mechanism, we provide a full set of tools to be used by various stakeholders to successfully assess the performance of software-based network functions.

Regarding the integration of software solutions into existing network architectures, as discussed by the third research question, Chapter 3 presents our contribution in the area of network function interoperability. To answer this question, we developed a data plane abstraction mechanism that allows the transparent translation between different control plane protocols. In addition to the introduction of the concept itself, we presented a proof-of-concept implementation that allows the transparent integration of regular SDN-enabled whitebox switches, software solutions as well as proprietary P4 programmable

hardware devices into the same network while using a unified control plane protocol, namely OpenFlow. We evaluated the performance impact of the data plane abstraction on the control plane performance and discussed opportunities and limitations of the concept. The proof-of-concept implementation has been published as an open-source tool. The contribution in this area is the investigation of the data plane abstraction concept itself. By highlighting the general feasibility and impact on control plane performance, the investigation we conducted is largely independent of the current state of the art. The same concept can be applied to new and currently unknown control plane protocols as well as data plane devices. At the time of writing, we work with the OpenFlow protocol and translate between different protocol versions as well as proprietary control mechanisms employed by P4 devices. This type of translation, however, is in no way a limitation of the concept. Instead, next generation devices and control protocols developed in the future can be subsumed in a single network in the same way. The same holds true for the feature aggregation discussed in Chapter 3. While new devices will likely solve the problems today's data plane devices have, the methodology of combining multiple, heterogeneous devices to solve problems emerging in the future remains valid, independent of the capabilities of single devices. Hence, the concept of data plane abstraction can be considered a timeless contribution and will remain valid, even as the underlying technology evolves.

Lastly, Chapter 4 answers the question of how to assess the performance of complex microservice-based packet processing architectures. In this context, our contribution is twofold. First, we examined an extensive dataset obtained through measurements in a real world IoT-focused MVNO core network in order to infer a realistic workload profile for the system. Here, we identified a simple feature set that enables the differentiation between types of devices solely through analysis of their mobile signaling traffic. Subsequently, we leveraged those insights during the development of a detailed simulation model of both the signaling workload and the mobile core microservice architecture. After the validation of our proposed simulation model, a series of case studies re-

garding bottleneck detection, resource scaling, and overload control have been conducted. We have shown that, using our approach, we are able to evaluate possible extensions of the architecture and have identified optimization potential currently being realized by the MVNO. An implementation of the simulation model as well as the required input data have been published for the community to use and extend in the future. Similarly, the model can be applied to future network architectures as well. As opposed to trace driven simulations that depend on accurate measurements reflecting the current system state, our model-based approach can be easily scaled and adapted to future signaling workloads and core architectures beyond 5G. At the same time, the current model implementation is a valuable tool for academia and MVNOs, as it provides a platform for parameter studies and system extensions. As we have shown, the simulation can be used to investigate the impact of system modifications without the need to implement modifications ahead of time. Thereby, operators can evaluate different mechanisms such as overload control, autoscaling or load balancing using a fraction of the time and cost it would take to develop proof-of-concept implementations.

The concepts, mechanisms and tools discussed throughout this monograph not only individually cover crucial parts of the landscape of software-based network functions, but, if seen in conjunction, outline a state-of-the-art workflow in evaluating the performance of complex, interconnected systems consisting of heterogeneous data plane components. By applying our approaches, a full performance estimation of individual components as well as a multi-component system can be achieved. Furthermore, we highlight the concept of data plane abstraction that allows the integration of heterogeneous data plane solutions into the same system. By publishing both the scientific insights and the developed tools, our contributions advance the state of the art regarding the performance evaluation of software-based network functions and serve as a basis for future work in this area. Here, several points remain as open areas to be addressed in the future. While this monograph focused purely on performance aspects, our contributions build the basis for further research beyond the scope of this work.

In the area of reliability, questions regarding the continuity of the provided processing performance need to be investigated. By combining our model with concepts known from the area of reliability and survivability modeling, new models covering the reliability and availability of software-based network functions need to be developed. It is well known that software is seldom perfect, the underlying hardware fails and unforeseen events can severely impact the operation of network functions. By investigating these aspects, we can strengthen our understanding of the general reliability of software solutions and the impact failures have on hybrid or pure-software systems. For example, the data plane abstraction approach introduced in this monograph can be applied to provide a transparent software-fallback in case of hardware failure, thereby allowing the system to keep functioning at reduced performance instead of outright failing. The implications, opportunities and limitations of such a mechanism remain to be investigated.

Similarly, questions regarding elasticity and scaling in the context of dimensioning software-based network architectures need to be addressed. Even today, many architectures are dimensioned towards the worst case, thereby wasting a significant amount of energy and money that could be used more effectively otherwise. Even as dynamic scaling is by no means a novel concept, its application to software-based network components has only gained importance in recent years. Questions regarding when to perform scale-out and when to perform scale-up need to be revisited and results obtained in the context of cloud elasticity need to be transferred to the networking domain. In this context, scaling mechanisms can be combined with our simulation model to investigate the impact of dynamic scaling using different scaling mechanisms on system performance and availability. In combination with the previously mentioned aspects of reliability and survivability, mechanisms that enable rapid scaling under unforeseen load spikes to ensure system survivability need to be examined. These aspects, among others, in the area of elasticity and scaling remain to be the subject of future research work.

These areas and more remain to be investigated in the future when it comes to improving our understanding of the implications of moving from hardware to software. Without knowing what developments the future holds and what technological advancements will be coming to the networking domain, the investigation of the current state of the art and the identification of generalizable models, concepts and ideas is the best way to prepare for what is to come. With the contributions presented in the monograph, we have laid the groundwork for impactful and relevant research in the area of network softwarization in the years to come, even beyond the performance aspects covered in this thesis.

Bibliography and References

Bibliography of the Author

Journal Papers

[1] S. Geißler, S. Lange, L. Linguaglossa, D. Rossi, T. Zinner, and T. Hoßfeld, "Discrete-time Modeling of NFV Accelerators that Exploit Batched Processing," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, 2021.

[2] T. Zinner, S. Geißler, S. Lange, S. Gebert, M. Seufert, and P. Tran-Gia, "A Discrete-time Model for Optimizing the Processing Time of Virtualized Network Functions," *Computer Networks*, 2017.

[3] S. Geißler, S. Herrnleben, R. Bauer, A. Grigorjew, T. Zinner, and M. Jarschel, "The Power of Composition: Abstracting a Multi-device SDN Data Path Through a Single API," *IEEE Transactions on Network and Service Management*, 2019.

[4] S. Geißler, F. Wamser, W. Bauer, S. Gebert, S. Kounev, and T. Hoßfeld, "Simulating Fully Virtualized IoT-centric Mobile Core Networks," *Under submission*, 2021.

Conference Papers

[5] S. Geißler, T. Prantl, S. Lange, F. Wamser, and T. Hoßfeld, "Discrete-time Analysis of the Blockchain Distributed Ledger Technology," in *31st International Teletraffic Congress (ITC 31)*, 2019.

[6] T. Zinner, S. Geißler, F. Helmschrott, and V. Burger, "Comparison of the Initial Delay for Video Playout Start for Different HTTP-based Transport Protocols," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017.

[7] S. Gebert, S. Geißler, T. Zinner, A. Nguyen-Ngoc, S. Lange, and P. Tran-Gia, "Zoom: Lightweight SDN-based Elephant Detection," in *28th International Teletraffic Congress (ITC 28)*, 2016.

[8] T. Zinner, S. Geißler, F. Helmschrott, S. Spinsante, and A. Braeken, "An AAL-oriented Measurement-based Evaluation of Different HTTP-based Data Transport Protocols," in *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2017.

[9] M. Flittner, R. Bauer, A. Rizk, S. Geißler, T. Zinner, and M. Zitterbart, "Taming the Complexity of Artifact Reproducibility," in *Proceedings of the Reproducibility Workshop*, 2017.

[10] A. Nguyen-Ngoc, S. Lange, S. Geißler, T. Zinner, and P. Tran-Gia, "Estimating the Flow Rule Installation Time of SDN Switches when Facing Control Plane Delay," in *International Conference on Measurement, Modelling and Evaluation of Computing Systems*, 2018.

[11] S. Schwarzmann, T. Zinner, S. Geißler, and C. Sieber, "Evaluation of the Benefits of Variable Segment Durations for Adaptive Streaming," in *Tenth International Conference on Quality of Multimedia Experience (QoMEX)*, 2018.

[12] S. Geißler and T. Zinner, "Tablevisor 2.0: Hardware-independent Multi Table Processing," in *KuVS-Fachgespräch Fog Computing 2018*, 2018.

[13] S. Geißler, S. Lange, F. Wamser, and T. Hoßfeld, "Deriving Youtube Playout Phases from Encrypted Packet Level Traffic," in *30th International Teletraffic Congress (ITC 30)*, 2018.

[14] A. Nguyen-Ngoc, S. Raffeck, S. Lange, S. Geißler, T. Zinner, and P. Tran-Gia, "Benchmarking the ONOS Controller with OFCProbe," in *IEEE Seventh International Conference on Communications and Electronics (ICCE)*, 2018.

[15] L. Iffländer, S. Geißler, J. Walter, L. Beierlieb, and S. Kounev, "Addressing Shortcomings of Existing DDoS Protection Software Using Software-Defined Networking," in *Proceedings of the 9th Symposium on Software Performance*, 2018.

[16] V. Skwarek, J. Reher, S. Geißler, and T. Zinner, "Automation und Digitalisierung: Potenzial und Einsatz von Blockchain-und Distributed-Ledger-Technologien in der Automatisierungstechnik," 2019.

[17] F. Loh, S. Geißler, F. Schaible, and T. Hoßfeld, "Talk to Me: Investigating the Traffic Characteristics of Amazon Echo Dot and Google Home," in *IEEE Eighth International Conference on Communications and Electronics (ICCE)*, 2021.

[18] S. Geißler, S. Lange, F. Wamser, T. Zinner, and T. Hoßfeld, "KOMon — Kernel-based Online Monitoring of VNF Packet Processing Times," in *International Conference on Networked Systems (NetSys)*, 2019.

[19] S. Lange, L. Linguaglossa, S. Geißler, D. Rossi, and T. Zinner, "Discrete-time Modeling of NFV Accelerators that Exploit Batched Processing," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*, 2019.

[20] S. Geißler, S. Herrnleben, R. Bauer, S. Gebert, T. Zinner, and M. Jarschel, "TableVisor 2.0: Towards Full-featured, Scalable and Hardware-Independent Multi Table Processing," in *IEEE Conference on Network Softwarization (NetSoft)*, 2017.

[21] S. Geißler, S. Lange, P. Tran-Gia, and T. Hoßfeld, "Discrete-time Analysis of Multicomponent GI/GI/1 Queueing Networks," in *International Conference on Networked Systems (NetSys)*, 2021.

[22] S. Geißler, F. Wamser, W. Bauer, M. Krolikowski, S. Gebert, and T. Hoßfeld, "Signaling Traffic in Internet-of-Things Mobile Networks," in *IFIP/IEEE International Symposium on Integrated Network Management (IM)*, 2021.

[23] F. Wamser, P. Tran-Gia, S. Geißler, and T. Hoßfeld, "Modeling of Traffic Flows in Internet of Things Using Renewal Approximation," in *Advances in Optimization and Decision Science for Society, Services and Enterprises*, 2019.

General References

[24] I. W. Stats, "Internet Users in the World by Regions," *Miniwatts Marketing Group*, 2018.

[25] H. Kim and N. Feamster, "Improving Network Management with Software Defined Networking," *IEEE Communications Magazine*, 2013.

[26] X. Fei, F. Liu, Q. Zhang, H. Jin, and H. Hu, "Paving the Way for NFV Acceleration: A Taxonomy, Survey and Future Directions," *ACM Computing Surveys (CSUR)*, 2020.

[27] J. Xie, D. Guo, Z. Hu, T. Qu, and P. Lv, "Control Plane of Software Defined Networks: A Survey," *Computer communications*, 2015.

[28] N. McKeown *et al.*, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM computer communication review*, 2008.

[29] P. Bosshart *et al.*, "P4: Programming Protocol-independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, 2014.

[30] M. Chiosi *et al.*, "Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges and Call for Action," in *SDN and OpenFlow world congress*, 2012.

[31] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network Function Virtualization: Challenges and Opportunities for Innovations," *IEEE Communications Magazine*, 2015.

[32] M. Ersue, "ETSI NFV Management and Orchestration - An Overview," *Presentation at the IETF*, 2013.

[33] S. Lal, T. Taleb, and A. Dutta, "NFV: Security Threats and Best Practices," *IEEE Communications Magazine*, 2017.

[34] V.-G. Nguyen, A. Brunstrom, K.-J. Grinnemo, and J. Taheri, "SDN/NFV-based Mobile Packet Core Network Architectures: A Survey," *IEEE Communications Surveys & Tutorials*, 2017.

[35] L. Linguaglossa *et al.*, "Survey of Performance Acceleration Techniques for Network Function Virtualization," *Proceedings of the IEEE*, 2019.

[36] X. Fei, F. Liu, H. Xu, and H. Jin, "Adaptive VNF Scaling and Flow Routing with Proactive Demand Prediction," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, 2018.

[37] J. G. Herrera and J. F. Botero, "Resource Allocation in NFV: A Comprehensive Survey," *IEEE Transactions on Network and Service Management*, 2016.

[38] S. Lange, A. Grigorjew, T. Zinner, P. Tran-Gia, and M. Jarschel, "A Multi-objective Heuristic for the Optimization of Virtual Network Function Chain Placement," in *2017 29th International Teletraffic Congress (ITC 29)*, 2017.

[39] J. Costa-Requena *et al.*, "SDN and NFV Integration in Generalized Mobile Network Architecture," in *2015 European conference on networks and communications (EuCNC)*, 2015.

[40] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "NFV: State of the Art, Challenges, and Implementation in Next Generation Mobile Networks (vEPC)," *IEEE Network*, 2014.

[41] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, and A. Meddahi, "NFV Security Survey: From Use Case Driven Threat Analysis to State-of-the-Art Countermeasures," *IEEE Communications Surveys & Tutorials*, 2018.

[42] A. K. Chimata, "Path of a Packet in the Linux Kernel Stack," *University of Kansas*, 2005.

[43] P. Emmerich, M. Pudelko, S. Bauer, S. Huber, T. Zwickl, and G. Carle, "User space network drivers," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019.

[44] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. Kaashoek, "The Click Modular Router," *Operating Systems Review*, 1999.

[45] Intel, *Intel Data Plane Development Kit (DPDK)*. [Online]. Available: `http://dpdk.org`.

[46] L. Rizzo, "netmap: A Novel Framework for Fast Packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, 2012. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rizzo`.

[47] B. Pfaff *et al.*, "The Design and Implementation of Open vSwitch," in *USENIX NSDI*, 2015.

[48] FD.io. "VPP Whitepaper." (2016), [Online]. Available: `https://fd.io/wp-content/uploads/sites/34/2017/07/FDioVPPwhitepaperJuly2017.pdf`.

[49] T. Barbette, C. Soldani, and L. Mathy, "Fast Userspace Packet Processing," in *ANCS*, 2015.

[50] J. Kim, S. Huh, K. Jang, K. Park, and S. Moon, "The Power of Batching in the Click Modular Router," in *Asia-Pacific Workshop on Systems*, 2012.

[51] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the Bar for Using GPUs in Software Packet Processing," in *NSDI*, 2015.

[52] R. Love, *Linux Kernel Development*. Pearson Education, 2010.

[53] L. Rizzo, "Device Polling Support for FreeBSD," in *BSDConEurope Conference*, 2001.

[54] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated Software Router," in *SIGCOMM*, 2010.

[55] L. Linguaglossa, D. Rossi, S. Pontarelli, D. Barach, D. Marjon, and P. Pfister, "High-speed Data Plane and Network Functions Virtualization by Vectorizing Packet Processing," *Computer Networks*, 2019.

[56] T. Herbert and W. de Bruijn, *Scaling in the Linux Networking Stack*, 2011. [Online]. Available: `https://www.kernel.org/doc/Documentation/networking/scaling.txt`.

[57] T. Barbette, G. P. Katsikas, G. Q. Maguire Jr, and D. Kostić, "RSS++ Load and State-aware Receive Side Scaling," in *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, 2019.

[58] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of Frameworks for High-performance Packet IO," in *ANCS*, 2015.

[59] N. Pitaev, M. Falkner, A. Leivadeas, and I. Lambadaris, "Characterizing the Performance of Concurrent Virtualized Network Functions with OVS-DPDK, FD.IO VPP and SR-IOV," in *ACM/SPEC International Conference on Performance Engineering*, 2018.

[60] F. Rath *et al.*, "SymPerf: Predicting Network Function Performance," in *Proceedings of the SIGCOMM Posters and Demos*, 2017.

[61] R. V. Rosa, C. Bertoldo, and C. E. Rothenberg, "Take Your VNF to the Gym: A Testing Framework for Automated NFV Performance Benchmarking," *IEEE Communications Magazine*, 2017.

[62] A. S. Rajan *et al.*, "Understanding the Bottlenecks in Virtualizing Cellular Core Network Functions," in *IEEE International Workshop on Local and Metropolitan Area Networks*, 2015.

[63] P. Naik, D. K. Shaw, and M. Vutukuru, "NFVPerf: Online Performance Monitoring and Bottleneck Detection for NFV," in *IEEE Conference on NFV and SDN*, 2016.

[64] *SONATA-NFV Project.* [Online]. Available: `https : / / github . com / sonata – NFV / son – monitor – probe / wiki / VNF – monitoring` (visited on 04/24/2019).

[65] T. Zhang, L. Linguaglossa, M. Gallo, P. Giaccone, L. Iannone, and J. Roberts, "Comparing the Performance of State-of-the-art Software Switches for NFV," in *ACM CoNEXT*, 2019.

[66] fd.io, *CSIT-2005 - Technical Report*, 2020. [Online]. Available: `https : / / docs . fd . io / csit / rls2005 / report / vpp % 5C % 5Fperformance%5C%5Ftests/` (visited on 10/25/2020).

[67] V. Fang, T. Lvai, S. Han, S. Ratnasamy, B. Raghavan, and J. Sherry, "Evaluating Software Switches: Hard or Hopeless?" *Berkeley Technical Report No. UCB/EECS-2018-136*, 2018.

[68] G. Faraci, A. Lombardo, and G. Schembra, "A Building Block to Model an SDN/NFV Network," in *IEEE International Conference on Communications (ICC)*, 2017.

[69] A. Lombardo, A. Manzalini, V. Riccobene, and G. Schembra, "An Analytical Tool for Performance Evaluation of Software Defined Networking Services," in *IEEE Network Operations and Management Symposium (NOMS)*, 2014.

[70] K. Suksomboon, M. Fukushima, S. Okamoto, and M. Hayashi, "A Dilated-CPU-consumption-based Performance Prediction for Multicore Software Routers," in *IEEE NetSoft Conference and Workshops (NetSoft)*, 2016.

[71] 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)*, 2016.

[72] Z. Su, T. Begin, and B. Baynat, "Towards Including Batch Services in Models for DPDK-based Virtual Switches," in *2017 Global Information Infrastructure and Networking Symposium (GIIS)*, 2017.

[73] I. W. Kabak, "Blocking and Delays in M(x)/M/c Bulk Arrival Queueing Systems," *Management Science*, 1970.

[74] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris, *Fundamentals of Queueing Theory*. John Wiley & Sons, 2018.

[75] D. M. Lucantoni, "New Results on the Single Server Queue with a Batch Markovian Arrival Process," *Communications in Statistics. Stochastic Models*, 1991.

[76] D. Manfield and P. Tran-Gia, "Analysis of a Finite Storage System with Batch Input Arising out of Message Packetization," *IEEE Transactions on Communications*, 1982.

[77] V. Q. Rodriguez and F. Guillemin, "Cloud-RAN Modeling Based on Parallel Processing," *IEEE Journal on Selected Areas in Communications*, 2018.

[78] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP Throughput: A Simple Model and its Empirical Validation," *ACM SIGCOMM Computer Communication Review*, 1998.

[79] E. Altman, K. Avrachenkov, and C. Barakat, "A Stochastic model of TCP/IP with Stationary Random Losses," *ACM SIGCOMM Computer Communication Review*, 2000.

[80] A. Klemm, C. Lindemann, and M. Lohmann, "Modeling IP Traffic using the Batch Markovian Arrival Process," *Performance Evaluation*, 2003.

[81] F. Guillemin, V. Q. Rodriguez, and A. Simonian, "A Processor-Sharing Model for the Performance of Virtualized Network Functions," in *2019 31st International Teletraffic Congress (ITC 31)*, 2019.

[82] M. L. Chaudhry and U. C. Gupta, "Queue-length and Waiting-time Distributions of Discrete-time GI X/Geom/1 Queueing Systems with Early and Late Arrivals," *Queueing Systems*, 1997.

[83] M. L. Chaudhry, U. C. Gupta, and V. Goswami, "Modeling and Analysis of Discrete-time Multiserver Queues with Batch Arrivals: GIX/Geom/m," *INFORMS Journal on Computing*, 2001.

[84] M. L. Chaudhry and U. Gupta, "Analysis of a Finite-buffer Bulk-service queue with Discrete-Markovian Arrival Process: D-MAP/Ga, b/1/N," *Naval Research Logistics (NRL)*, 2003.

[85] W. B. Powell and P. Humblet, "The Bulk Service Queue with a General Control Strategy: Theoretical Analysis and a new Computational Procedure," *Operations Research*, 1986.

[86] M. L. Chaudhry and S. H. Chang, "Analysis of the Discrete-time Bulk-service Queue Geo/GY/1/N+ B," *Operations Research Letters*, 2004.

[87] N. T. Bailey, "On Queueing Processes with Bulk service," *Journal of the Royal Statistical Society: Series B (Methodological)*, 1954.

[88] A. Banerjee, U. C. Gupta, and S. R. Chakravarthy, "Analysis of a Finite-buffer Bulk-service Queue under Markovian arrival Process with Batch-size-dependent Service," *Computers & Operations Research*, 2015.

[89] G. L. Curry and R. M. Feldman, "An M/M/1 Queue with a General Bulk Service Rule," *Naval research logistics quarterly*, 1985.

[90] M. F. Neuts, "Transform-free Equations for the Stationary Waiting Time Distributions in the Queue with Poisson Arrivals and Bulk Services," *Annals of Operations Research*, 1987.

[91] M. L. Chaudhry and J. Gai, "A Simple and Extended Computational Analysis of M/G j (a, b)/1 and M/G j (a, b)/1/(B+ b) Queues Using Roots," *INFOR: Information Systems and Operational Research*, 2012.

[92] S. K. Bar-Lev, M. Parlar, D. Perry, W. Stadje, and F. A. Van der Duyn Schouten, "Applications of Bulk Queues to Group Testing Models with Incomplete Identification," *European Journal of Operational Research*, 2007.

[93] Y. Ren, T. Phung-Duc, Y.-K. Liu, J.-C. Chen, and Y.-H. Lin, "ASA: Adaptive VNF Scaling Algorithm for 5G Mobile Networks," in *Proceedings of the IEEE International Conference on Cloud Networking (CloudNet)*, 2018.

[94] Y. Ren, T. Phung-Duc, J.-C. Chen, and Z.-W. Yu, "Dynamic Auto Scaling Algorithm (DASA) for 5g Mobile Networks," in *Proceedings of the IEEE global communications conference (GLOBECOM)*, 2016.

[95] T. Phung-Duc, Y. Ren, J.-C. Chen, and Z.-W. Yu, "Design and Analysis of Deadline and Budget Constrained Autoscaling (DBCA) Algorithm for 5G Mobile Networks," in *Proceedings of the IEEE international conference on cloud computing technology and science (CloudCom)*, 2016.

[96] S. Gebert, C. Schwartz, T. Zinner, and P. Tran-Gia, "Continuously Delivering your Network," in *International Symposium on Integrated Network Management (IM)*, 2015.

[97] *Linux Kernel Source Code.* [Online]. Available: `http://kernel.org`.

[98] *Data Plane Development Kit.* [Online]. Available: `http://dpdk.org/` (visited on 04/26/2019).

[99] *SnabbCo, "Snabb: Simple and Fast Packet Networking.* [Online]. Available: `https://github.com/snabbco/snabb` (visited on 04/26/2019).

[100] T. L. Foundation, *Wiki: NAPI.* [Online]. Available: `https://wiki.linuxfoundation.org/networking/napi`.

[101] K. Salah and A. Qahtan, "Implementation and Experimental Performance Evaluation of a Hybrid Interrupt-handling Scheme," *Computer Communications*, 2009.

[102] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, 2006.

[103] 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," Tech. Rep., 2016.

[104] W. J. Stewart, *Probability, Markov Chains, Queues, and Simulation*. Princeton university press, 2009.

[105] S. Bradner and J. McQuaid, *RFC2544 Benchmarking Methodology for Network Interconnect Devicesgoo*, 1999. [Online]. Available: `https://www.ietf.org/rfc/rfc2544.txt`.

[106] [Online]. Available: `https://github.com/lsinfo3/2020-tompecs-vpp-data`.

[107] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, and D. Rossi, "High-Speed Software Data Plane via Vectorized Packet Processing," *IEEE Communications Magazine*, 2018.

[108] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A Scriptable High-speed Packet Generator," in *IMC*, 2015.

[109] N. Feamster, J. Rexford, and E. Zegura, "The Road to Sdn: An Intellectual History of Programmable Networks," *ACM SIGCOMM Computer Communication Review*, 2014.

[110] B. Liu, J. Bi, and Y. Zhou, "Source Address Validation in Software Defined Networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[111] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel, "Split: Optimizing Space, Power, and Throughput for TCAM-based Packet Classification Systems," in *Proceedings of the ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, 2011.

[112] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "CAB: A Reactive Wildcard Rule Caching System for Software-Defined Networks," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[113] X. Sun, T. E. Ng, and G. Wang, "Software-Defined Flow Table Pipeline," in *Cloud Engineering (IC2E) on IEEE International Conference*, 2015.

[114] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch," *ACM SIGCOMM Computer Communication Review*, 2014.

[115] R. Bifulco, J. Boite, M. Bouet, and F. Schneider, "Improving SDN with InSPired Switches," in *Proceedings of the Symposium on SDN Research*, 2016.

[116] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the Power of Flexible Packet Processing for Network Resource Allocation," in *14th USENIX Symposium on Networked Systems Design and Impl. (NSDI)*, 2017.

[117] R. Bifulco and A. Matsiuk, "Towards Scalable SDN Switches: Enabling Faster Flow Table Entries Installation," in *ACM SIGCOMM Computer Communication Review*, 2015.

[118] *Official website of the IEEE 802.1 Time-Sensitive Networking task group*, Accessed: 2019-06-18. [Online]. Available: `https://1.ieee802.org/tsn/`.

[119] R. Sherwood *et al.*, "FlowVisor: A Network Virtualization Layer," *OpenFlow Switch Consortium, Tech. Rep*, 2009.

[120] R. D. Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, "VeRTIGO: Network Virtualization and Beyond," in *European Workshop on Software Defined Networking*, 2012.

[121] A. Al-Shabibi *et al.*, "OpenVirteX: Make Your Virtual SDNs Programmable," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[122] X. Jin, J. Gossels, J. Rexford, and D. Walker, "CoVisor: A Compositional Hypervisor for Software-Defined Networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015.

[123] I. Alawe, B. Cousin, O. Thorey, and R. Legouable, "Integration of Legacy Non-SDN Optical ROADMs in a Software Defined Network," in *IEEE International Symposium on Software Defined Systems*, 2016.

[124] R. Bauer and M. Zitterbart, "Port Based Capacity Extensions (PBCEs): Improving SDNs Flow Table Scalability," in *28th International Teletraffic Congress (ITC 28)*, 2016.

[125] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, "The FlowAdapter: Enable Flexible Multi-Table Processing on Legacy Hardware," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

[126] H. Pan, G. Xie, Z. Li, P. He, and L. Mathy, "FlowConvertor: Enabling Portability of SDN Applications," in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017.

[127] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Infinite CacheFlow in Software-Defined Networks," in *Proceedings of the third workshop on Hot topics in software defined networking*, 2014.

[128] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing Tables in Software-Defined Networks," in *INFOCOM, Proceedings IEEE*, 2013.

[129] N. Kang, Z. Liu, J. Rexford, and D. Walker, "Optimizing the "One Big Switch" Abstraction in Software-Defined Networks," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013.

[130] M. Yu, A. Wundsam, and M. Raju, "NOSIX: A Lightweight Portability Layer for the SDN OS," *ACM SIGCOMM Computer Communication Review*, 2014.

[131] L. Molnár *et al.*, "Dataplane Specialization for High-performance OpenFlow Software Switching," in *Proceedings of the 2016 ACM SIGCOMM Conference*.

[132] P. Bosshart *et al.*, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," in *ACM SIGCOMM Computer Communication Review*, 2013.

[133] T. O. N. Foundation, *OpenFlow Table Type Patterns, Version 1.0*, 2014.

[134] D. Parniewicz *et al.*, "Design and Implementation of an OpenFlow Hardware Abstraction Layer," in *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*.

[135] N. Foster *et al.*, "Frenetic: A Network Programming Language," *ACM Sigplan Notices*, 2011.

[136] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic," *Technical Report of USENIX*, 2013.

[137] M. Shahbaz *et al.*, "PISCES: A Programmable, Protocol-Independent Software Switch," in *Proceedings of the 2016 ACM SIGCOMM Conference*.

[138] A. Beifus *et al.*, "A Study of Networking Software Induced Latency," in *International Conference and Workshops on Networked Systems (NetSys)*, 2015.

[139] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, "Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis," *Journal of Network and Systems Management*, 2018.

[140] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable Flow-Based Networking with DIFANE," *ACM SIGCOMM Computer Communication Review*, 2010.

[141] R. Ozdag, "Intel® Ethernet Switch FM6000 Series-software Defined Networking," vol. 5, 2012.

[142] H. Song, "Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

[143] S. Chole *et al.*, "dRMT: Disaggregated Programmable Switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[144] R. Bifulco and G. Rétvári, "A Survey on the Programmable Data Plane: Abstractions, Architectures, and Open Problems," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018.

[145] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer, "Interfaces, Attributes, and Use Cases: A Compass for SDN," *IEEE Communications Magazine*, 2014.

[146] A. Lara, A. Kolasani, and B. Ramamurthy, "Network Innovation using OpenFlow: A Survey," *IEEE communications surveys & tutorials*, 2013.

[147] A. Nygren *et al.*, "OpenFlow Switch Specification Version 1.5. 1," *Open Networking Foundation, Tech. Rep*, 2015.

[148] P. L. Consortium *et al.*, "P416 Language Specification," *Retrieved Oct*, 2017.

[149] A. Nakao, "Software-defined Data Plane Enhancing SDN and NFV," *IEICE Transactions on Communications*, 2015.

[150] K. Yamazaki, T. Osaka, S. Yasuda, S. Ohteru, and A. Miyazaki, "Accelerating SDN/NFV with Transparent Offloading Architecture," in *Open Networking Summit 2014 (ONS 2014)*, 2014.

[151] D. Moro, G. Verticale, and A. Capone, "Network Function Decomposition and Offloading on Heterogeneous Networks with Programmable Data Planes," *IEEE Open Journal of the Communications Society*, 2021.

[152] E. W. Dijkstra, "On the Role of Scientific Thought," in *Selected writings on computing: a personal perspective*, 1982.

[153] A. X. Liu, C. R. Meiners, and E. Torng, "TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs," *IEEE/ACM Transactions on Networking (TON)*, 2010.

[154] T. O. N. Foundation, *The Benefits of Multiple Flow Tables and TTPs*, 2015.

[155] A. Nguyen-Ngoc, S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, and M. Jarschel, "Performance Evaluation Mechanisms for FlowMod Message Processing in OpenFlow Switches," in *IEEE Sixth International Conference on Communications and Electronics (ICCE)*, 2016.

[156] M. Kuźniar, P. Pereš'ıni, and D. Kostić, "What you need to know about SDN flow tables," in *International Conference on Passive and Active Network Measurement*, 2015.

[157] J. R. Jackson, "Networks of Waiting Lines," *Operations research*, 1957.

[158] J. Jackson, "Jobshop-like Queueing Systems," *Management Science*, 1963.

[159] W. J. Gordon and G. F. Newell, "Closed Queuing Systems with Exponential Servers," *Operations research*, 1967.

[160] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *Journal of the ACM (JACM)*, 1975.

[161] E. Gelenbe, "G-networks by Triggered Customer Movement," *Journal of applied probability*, 1993.

[162] E. Gelenbe and J.-M. Fourneau, "G-networks with Resets," *Performance Evaluation*, 2002.

[163] G. Hasslinger and E. S. Rieger, "Analysis of Open Discrete-time Queueing Networks: A Refined Decomposition Approach," *Journal of the Operational Research Society*, 1996.

[164] M. Villamizar *et al.*, "Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015.

[165] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when Moving from Monolith to Microservice Architecture," in *International Conference on Web Engineering*, 2017.

[166] C. Esposito, A. Castiglione, and K.-K. R. Choo, "Challenges in Delivering Software in the Cloud as Microservices," *IEEE Cloud Computing*, 2016.

[167] Y. Sun, L. Meng, P. Liu, Y. Zhang, and H. Chan, "Automatic Performance Simulation for Microservice based Applications," in *Asian Simulation Conference*, 2018.

[168] A. Van Hoorn, A. Aleti, T. F. Düllmann, and T. Pitakrat, "ORCAS: Efficient Resilience Benchmarking of Microservice Architectures," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018.

[169] S. Wang, Z. Ding, and C. Jiang, "Elastic Scheduling for Microservice Applications in Clouds," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[170] N. Dragoni *et al.*, "Microservices: Yesterday, Today, and Tomorrow," *Present and ulterior software engineering*, 2017.

[171] J. Thönes, "Microservices," *IEEE software*, 2015.

[172] J. Vučković, "You Are Not Netflix," in *Microservices*, 2020.

[173] A. Basiri *et al.*, "Chaos Engineering," *IEEE Software*, 2016.

[174] M. A. Chang, B. Tschaen, T. Benson, and L. Vanbever, "Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[175] J. Robbins, K. Krishnan, J. Allspaw, and T. Limoncelli, "Resilience Engineering: Learning to Embrace Failure," *Communications of the ACM*, 2012.

[176] H. Nakama, "Inside Azure Search: Chaos Engineering," *blog, Microsoft*, 2015.

[177] Y. Sverdlik, "Facebook Turned Off Entire Data Center to Test Resiliency," *Data Center Knowledge*, 2014.

[178] A. Sriraman and T. F. Wenisch, "μ Suite: A Benchmark Suite for Microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018.

[179] M. Grambow, L. Meusel, E. Wittern, and D. Bermbach, "Benchmarking Microservice Performance: A Pattern-based Approach," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020.

[180] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark Requirements for Microservices Architecture Research," in *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, 2017.

[181] F. Metzger, T. Hoßfeld, A. Bauer, S. Kounev, and P. E. Heegaard, "Modeling of Aggregated IoT traffic and its Application to an IoT Cloud," *Proceedings of the IEEE*, 2019.

[182] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang, "A First Look at Cellular Machine-to-Machine Traffic: Large Scale Measurement and Characterization," *ACM SIGMETRICS performance evaluation review*, 2012.

[183] I. Cvitić, P. Zorić, T. Kuljanić, and M. Musa, "Analysis of Network Traffic Features Generated by IoT Devices," in *XXXVII Simpozijum o novim tehnologijama u poštanskom i telekomunikacionom saobraćaju*, 2019.

[184] M. Laner, P. Svoboda, N. Nikaein, and M. Rupp, "Traffic Models for Machine Type Communications," in *ISWCS 2013; The Tenth International Symposium on Wireless Communication Systems*, 2013.

[185] A. Sivanathan *et al.*, "Classifying IoT Devices in Smart Environments using Network Traffic Characteristics," *IEEE Transactions on Mobile Computing*, 2018.

[186] H. Tahaei, F. Afifi, A. Asemi, F. Zaki, and N. B. Anuar, "The Rise of Traffic Classification in IoT networks: A Survey," *Journal of Network and Computer Applications*, 2020.

[187] F. Malandra, S. Rochefort, P. Potvin, and B. Sansò, "A Case Study for M2m Traffic Characterization in a Smart City Environment," in *Proceedings of the 1st International Conference on Internet of Things and Machine Learning*, 2017.

[188] I. L. Bedhiaf, O. Cherkaoui, and G. Pujolle, "Third-generation Virtualized Architecture for the MVNO Context," *annals of telecommunications-annales des télécommunications*, 2009.

[189] Y. Li *et al.*, "Understanding the Ecosystem and Addressing the Fundamental Concerns of Commercial MVNO," *IEEE/ACM Transactions on Networking*, 2020.

[190] D. B. Johnson, "Validation of Wireless and Mobile Network Models and Simulation," in *DARPA/NIST network simulation validation workshop*, 1999.

[191] A. P. Jardosh, E. M. Belding-Royer, K. C. Almeroth, and S. Suri, "Real-world Environment Models for Mobile Network Evaluation," *IEEE Journal on Selected Areas in Communications*, 2005.

[192] F. Metzger, C. Schwartz, and T. Hoßfeld, "GTP-based Load Model and Virtualization Gain for a Mobile Network's GGSN," in *2014 IEEE Fifth International Conference on Communications and Electronics (ICCE)*, 2014.

[193] R. Samoilenko, N. Accurso, and F. Malandra, "A Simulation Study on the Impact of IoT Traffic in a Smart-city LTE Network," in *2020 IEEE 31st Annual International Symposium on Personal, Indoor and Mobile Radio Communications*.

[194] N. Xia, H.-H. Chen, and C.-S. Yang, "Radio Resource Management in Machine-to-Machine Communications — A Survey," *IEEE Communications Surveys & Tutorials*, 2017.

[195] S. Duan, V. Shah-Mansouri, Z. Wang, and V. W. Wong, "D-ACB: Adaptive Congestion Control Algorithm for Bursty M2M Traffic in LTE Networks," *IEEE Transactions on Vehicular Technology*, 2016.

[196] N. Jiang, Y. Deng, A. Nallanathan, X. Kang, and T. Q. Quek, "Analyzing Random Access Collisions in Massive IoT Networks," *IEEE Transactions on Wireless Communications*, 2018.

[197] R. Ratasuk, N. Mangalvedhe, and A. Ghosh, "Overview of LTE Enhancements for Cellular IoT," in *2015 IEEE 26th annual international symposium on personal, indoor, and mobile radio communications (PIMRC)*, 2015.

[198] A. Ksentini, Y. Hadjadj-Aoul, and T. Taleb, "Cellular-based Machine-to-Machine: Overload Control," *IEEE Network*, 2012.

[199] M. Gotin, D. Werle, F. Lösch, A. Koziolek, and R. Reussner, "Overload Protection of Cloud-Iot Applications by Feedback Control of Smart Devices," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, 2019.

[200] L. Ferdouse, A. Anpalagan, and S. Misra, "Congestion and Overload Control Techniques in Massive M2M Systems: A Survey," *Transactions on Emerging Telecommunications Technologies*, 2017.

[201] M. Welsh and D. E. Culler, "Adaptive Overload Control for Busy Internet Servers.," in *USENIX Symposium on Internet Technologies and Systems*, 2003.

[202] P. Tran-Gia, *Analysis of a Load-driven Overload Control Mechanism in Discrete-time Domain*. IBM Thomas J. Watson Research Division, 1988.

[203] P. Tran-Gia and T. Hoßfeld, *Performance Modeling and Analysis of Communication Networks: A Lecture Note*. BoD–Books on Demand, 2021.

[204] D. B. Webster, M. L. Padgett, G. S. Hines, and D. L. Sirois, "Determining the Level of Detail in a Simulation Model — A Case Study," *Computers & industrial engineering*, 1984.

[205] T. J. Teorey and A. G. Merten, "Considerations on the Level of Detail in Simulation," in *Proceedings of the 1st symposium on Simulation of computer systems*, 1973.

[206] A. J. Smith, "Trace driven Simulation in Research on Computer Architecture and Operating Systems," in *Proceedings of the Conference on New Directions in Simulation for Manufacturing and Communications*, 1994.

[207] J. P. Kleijnen, B. Bettonvil, and W. J. Van Groenendaal, "Validation of Trace-driven Simulation Models: Regression Analysis Revisited," in *Proceedings of the 28th conference on Winter simulation*, 1996.

[208] Y. Joo, V. Ribeiro, A. Feldmann, A. C. Gilbert, and W. Willinger, "TCP/IP Traffic Dynamics and Network Performance: A Lesson in Workload Modeling, Flow Control, and Trace-Driven Simulations," *ACM SIGCOMM Computer Communication Review*, 2001.

[209] P. E. Heegaard, "GenSyn-a Java Based Generator of Synthetic Internet Traffic Linking User Behaviour Models to Real Network Protocols," in *ITC Specialist Seminar on IP Traffic Measurement, Modeling and Management*, 2000.

[210] P. E. Heegaard, B. E. Helvik, and R. Andreassen, "Application of Rare Event Techniques to Trace Driven Simulation," in *Proceedings of the Winter Simulation Conference, 2005.*, 2005.

[211] A. Sivanathan *et al.*, "Characterizing and Classifying IoT traffic in Smart Cities and Campuses," in *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2017.

[212] G. M. D. T. Forecast, "Cisco Visual Networking Index: Global Mobile Data Traffic Forecast Update, 2017–2022," *Update*, 2019.

[213] P. Cerwall, P. Jonsson, R. Möller, S. Bävertoft, S. Carson, I. Godor, *et al.*, "Ericsson Mobility Report," *On the Pulse of the Networked Society. Hg. v. Ericsson*, 2017.

[214] T. Taleb and A. Kunz, "Machine Type Communications in 3GPP Networks: Potential, Challenges, and Solutions," *IEEE Communications Magazine*, 2012.

[215] Statista, "The Internet of Things (IoT)* Units Installed Base by Category from 2014 to 2020 (in billions)," Tech. Rep., 2018. [Online]. Available: `https://www.statista.com/statistics/370350/internet-of-things-installed-base-by-category/`.

[216] A. Lutu, B. Jun, F. E. Bustamante, D. Perino, M. Bagnulo, and C. G. Bontje, "A First Look at the IP EXchange Ecosystem," *SIGCOMM Comput. Commun. Rev.*, 2020.

[217] A. Lutu, B. Jun, A. Finamore, F. Bustamante, and D. Perino, "Where Things Roam: Uncovering Cellular IoT/M2M Connectivity," *arXiv preprint arXiv:2007.13708*, 2020.

[218] C. Schwartz, F. Lehrieder, F. Wamser, T. Hoßfeld, and P. Tran-Gia, "Smart-phone Energy Consumption vs. 3G Signaling Load: The Influence of Application Traffic Patterns," in *2013 24th Tyrrhenian International Workshop on Digital Communications-Green ICT (TIWDC)*, 2013.

[219] G. Gorbil, O. H. Abdelrahman, M. Pavloski, and E. Gelenbe, "Modeling and Analysis of RRC-based Signalling Storms in 3G Networks," *IEEE Transactions on Emerging Topics in Computing*, 2015.

[220] R. R. Tyagi, F. Aurzada, K.-D. Lee, and M. Reisslein, "Connection Establishment in LTE-A Networks: Justification of Poisson Process Modeling," *IEEE Systems Journal*, 2015.

[221] 3GPP, "GERAN Improvements for Machine-Type Communications (MTC)," *Technical report (TR)*, 2014.

[222] C. Palm, "Intensitätsschwankungen im Fernsprechverker," *Ericsson Technics*, 1943.

[223] A. Nordrum *et al.*, "Popular Internet Of Things Forecast of 50 bIllion Devices By 2020 Is Outdated," *IEEE spectrum*, 2016.

[224] A. Čolaković and M. Hadžialić, "Internet of Things (IoT): A Review of Enabling Technologies, Challenges, and Open Research Issues," *Computer Networks*, 2018.

[225] Y. Choi, C.-h. Yoon, Y.-s. Kim, S. W. Heo, and J. A. Silvester, "The Impact of Application Signaling Traffic on Public Land Mobile Networks," *IEEE Communications Magazine*, 2014.

[226] D. Gannon, R. Barga, and N. Sundaresan, "Cloud-native Applications," *IEEE Cloud Computing*, 2017.

[227] N. Kratzke and P.-C. Quint, "Understanding Cloud-native Applications after 10 Years of Cloud Computing - A Systematic Mapping Study," *Journal of Systems and Software*, 2017.

[228] M. Giess and D. Sisalem, "Multi-layered Distributed GTP-C Processing," E.U. Patent EP3806412, April 14, 2021.

[229] I. Baldini *et al.*, "Serverless Computing: Current Trends and Open Problems," in *Research Advances in Cloud Computing*, 2017.

[230] R. J. Byrd, S. E. Smith, and S. P. deJong, "An Actor-based Programming System," in *Proceedings of the SIGOA conference on Office information systems*, 1982.

[231] P. Haller and M. Odersky, "Scala Actors: Unifying Thread-based and Event-based Programming," *Theoretical computer science*, 2009.

[232] Akka, *Akka - Build Concurrent, Distributed, and Resilient Message-driven Applications for Java and Scala*. [Online]. Available: `https://akka.io/` (visited on 05/28/2021).

[233] T. ETSI, "129 002 V10. 2.0 (Apr. 2011) Digital Cellular Telecommunications System (Phase 2+)," *Universal Mobile Telecommunications System (UMTS)*,

ISSN 1432-8801