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

Resilience, Availability, and Serviceability Evaluation in Software-defined Networks

Christopher Valentin Metter

Würzburger Beiträge zur
Leistungsbewertung Verteilter Systeme

Bericht 04/18

Würzburger Beiträge zur
Leistungsbewertung Verteilter Systeme

Herausgeber

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

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

Satz

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

ISSN 1432-8801

Resilience, Availability, and Serviceability Evaluation in Software-defined Networks

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

vorgelegt von

Christopher Valentin Metter

geboren in

Wertheim

Würzburg 2019

Eingereicht am: 14.11.2018
bei der Fakultät für Mathematik und Informatik
1. Gutachter: Prof. Dr.-Ing. Phuoc Tran-Gia
2. Gutachter: Prof. Dr. Andreas Kassler
Tag der mündlichen Prüfung: 17.01.2019

Danksagung

Nach nunmehr vier weiteren Jahren an der Universität habe auch ich es geschafft: Es gibt einen weiteren Dr. Metter. In dieser Zeit gab es einige Menschen die mich direkt oder auch indirekt bei meinem Vorhaben unterstützt haben. Dementsprechend möchte ich die Gelegenheit nutzen, um mich zu bedanken.

Zuallererst möchte ich mich bei Herrn Prof. Dr.-Ing. Phuoc Tran-Gia für die Möglichkeit einer Promotion am Lehrstuhl für Kommunikationsnetze nachzugehen, bedanken. Durch die Projekte und Kooperationen mit akademischen und industriellen Partnern, das Schreiben von diversen Publikationen, den ein oder anderen Blick über den akademischen Tellerrand und das Besuchen von internationalen Konferenzen bekam ich als Mitarbeiter am Lehrstuhl zahlreiche Möglichkeiten mich zu verwirklichen.

Mit Herrn Prof. Dr. Tobias Hoßfeld hat der Lehrstuhl einen würdigen Nachfolger aus seinem eigenen Kader erhalten. Ihm danke ich für die Möglichkeit meine Zeit am Lehrstuhl um die entscheidenden Monate zu verlängern und seiner Bereitschaft meiner Disputation als Prüfungsvorsitzender beizuwohnen. Ich bin gespannt wie sich der Lehrstuhl in den nächsten Wochen und Monaten unter seiner Leitung weiterentwickelt. Herrn Prof. Dr. Andreas Kassler danke ich für sein Interesse an meiner Arbeit und seiner Bereitschaft ein Zweitgutachten für meine Dissertation zu übernehmen. Des Weiteren bedanke ich mich bei Prof. Dr.-Ing. Samuel Kounev, der trotz seines vollen Terminkalenders bei meiner Disputation als dritter Prüfer kurzfristig zur Verfügung stand.

Besonderen Dank gebührt auch meinem Gruppenleiter Dr. Florian Wamser. Durch seine Motivation, zahlreichen Diskussionen, tollen Dienstreisen und den

ein oder anderen Eiskaffee hat er meine Zeit am Lehrstuhl maßgeblich mitgeprägt.

Bei den ehemaligen Leitern des *A-Teams*, den Doktoren Robert Henjes, Daniel Schlosser und Michael Jarschel, möchte ich mich für die Eingliederung in das Lehrstuhlleben im Zuge meiner Hiwitätigkeiten bedanken. Sie haben mich schon im Laufe meines Bachelor- und dann des Masterstudiums auf die Idee gebracht, dass eine Anstellung am Lehrstuhl interessant sein könnte. Bei Daniel hatte ich im Laufe eines Seminars im Jahr 2009 zum ersten Mal Kontakt mit „OpenFloW". Damals hätte ich nie gedacht, dass dieses „SDN"-Thema meine Zeit am Lehrstuhl bis heute so stark beeinflussen könnte. Michael hat mir schließlich im Laufe meiner Masterarbeit erste Einblicke in das „richtige" wissenschaftliche Arbeiten geben können.

Bei meinen ehemaligen Kollegen am Lehrstuhl für Informatik III bedanke ich mich für die gemeinsame Zeit, hilfreichen Diskussionen, gemeinsamen Arbeiten und dem ein oder anderen abendlichen Zeitvertreib. Mit einigen von euch werde ich sicherlich noch in Kontakt bleiben. Namentlich sind das die übrigen Gruppenleiter Dr. Thomas Zimmer, Dr. Matthias Hirth und Dr. Florian Metzger, meine ehemaligen Zimmerkollegen Dr. Michael Seufert und Anika Schwind, die restlichen Mitglieder der CAN-Gruppe Dr. Christian Schwartz, Dr. Valentin Burger, Dr. Lam Dinh-Xuan, Frank Loh und Christian Moldovan, die Administratorenkollegen Dr. Steffen Gebert, Nicholas Gray und Stefan Geißler sowie die restlichen Weggefährten Kathrin Borchert, Alexej Grigorjew, Dr. David Hock, Dr. Stanislav Lange, Dr. Anh Nguyen-Ngoc und Susanna Schwarzmann.

Besonderem Dank gilt natürlich auch Alison Wichmann, unserer guten Fee im Sekretariat. Sie hält mit ihrem einmaligen Humor den Lehrstuhl auf Kurs und hilft bei allen möglichen bürokratischen Irrwegen, die man an der Universität Würzburg zu gehen hat. Außerdem danke ich unseren Hiwis und Studenten; ohne ihren Beitrag könnten viele Dinge am Lehrstuhl nicht realisiert werden.

Zu guter Letzt geht mein Dank auch an meine Freunde und meine Familie. Sie haben für die willkommene Abwechslung zum manchmal stressigen Arbeitsall-

tag gesorgt, mir die nötige Unterstützung gegeben und mir den nötigen Rückzugsort geboten. Ohne euch hätte ich es nicht soweit geschafft.

Und natürlich danke an alle anderen, die nicht genannt wurden, es aber trotzdem verdient haben.

So long and thanks for all the fish.

– Douglas Adams (1952 - 2001)

Contents

1 Introduction

In recent years, the way in which communication networks are structured and managed has changed radically. With the financial pressures on network operators and the ever-changing demands on the network, many people want networks to become more agile, adaptable, and flexible to stand out from today's financially hard-to-maintain, rigid, and ossified legacy networks. This is caused by the rise of data centers with excessive virtualization usage like Amazon AWS, Google Cloud Computing, or Microsoft Azure. This requires the ability to quickly provision network connections that helps to meet increased network demands on the fly and save costs through greater efficiency. The fundamental idea, originating back to Martin Casado from Stanford University, is to enforce the decoupling of the foremost strictly connected data and control plane of switching elements. The data plane merely transports the data and is now independent of the control functions in the control plane, which is called Software-defined Networking (SDN). SDN is characterized by its benefits, namely the increased flexibility in configuration and management due to the softwarization of the control plane, the potential for resource usage optimization by its centrality of the control plane and the advanced monitoring features, and the vendor independence by design.

With the introduction of SDN in the late 2000s, not only a new research field has been created, but a paradigm shift was initiated in the broad field of networking. The programmable network control by SDN is a big step, but also a stumbling block for many of the established network operators and vendors. As with any new technology the question about the maturity and the production-readiness of it arises. Therefore, this thesis picks specific features of SDN and

analyzes its performance, reliability, and availability in scenarios that can be expected in production deployments.

1.1 Motivation

SDN is a new networking paradigm that overcomes various drawbacks of current communication networks. Unlike in earlier, rigid deployments of traditional networking deployments, the control and data plane of switching devices is decoupled and all control functions are centralized within the network controller. With this simple principle, SDN allows for the fine-granular control over packet forwarding and packet processing. Additionally, new networking scenarios are possible, facilitated by the introduction of increased control capabilities, dynamic modification of networking parameters, or the control of traffic flows on different granularities. Further, the applicability of these features is not restricted to a specific type of network, but can be used in access networks, data center networks, and wide area networks [8–10]. This versatility offers both opportunities and challenges in terms of performance and implementation. The network operators must balance these points to establish a high performance operational network.

The question in general is how traffic in SDN networks affects the SDN environment with controller, control plane, data plane, and switch. A performance analysis of the SDN technology is thus essential if the operator wants to understand the network in control and data plane, and yet, does not want to miss the improvements and advantages of the new technology.

Figure 1.1 shows the schematic architecture of SDN. On the bottom we see the infrastructure layer, where network devices, both hard- and software are found. In general, these devices are rather dumb and are only able to execute actions defined in their ruleset, the so-called flow table.

These tables are programmed via the Southbound API by the control layer of the SDN architecture: the so-called SDN controller. This piece of software, depicted in the center of the schematic, is dynamically programmable and cus-

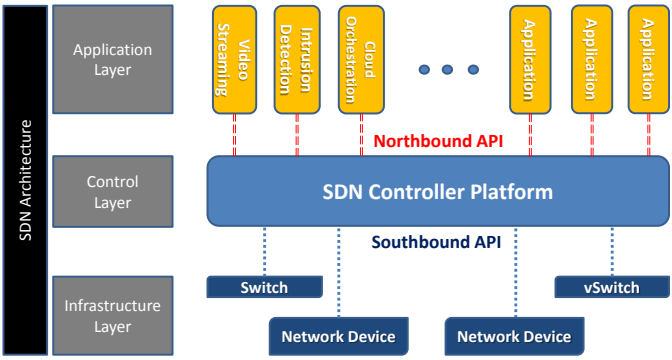


Figure 1.1: SDN architecture.

tomizable to the requirements of the network. Many of today's SDN controllers offer a core module with the basic functionality such as a protocol agnostic connectivity to network devices, creation of rules, and an API. To enhance the features of the controllers, additional modules are available. Examples are a Southbound API module implementing the OpenFlow protocol or a controller module replicating the Layer-2 switch functionality. In order to scale with the size of the network, many SDN controllers allow to physically distribute the controller whilst still maintaining a logically centralized view of the network.

In general, flow processing with SDN works like this: As soon as a packet enters a switching device, it is checked if a matching rule is installed in its routing table, the so-called flow table. If a match is found, the packet is processed following the associated actions. If no match is found, signaling traffic is created by forwarding the packet to the controller, and, thus, informing it about a table-miss. The controller now calculates the appropriate action for this flow and submits a new rule for this match back to the device. The device now installs that rule into its flow table and follows the actions of that flow for all further packets matching this flow.

Since the switches are only forwarding elements without any decision logic, they can route network packets efficiently but require the interaction with a controller that calculates the instructions in the form of rules. This switch-controller interaction is henceforth called signaling. The central controller must pay attention to the load by signaling in the network, so as not to interfere with the network performance due to control plane congestion. Scalability issues also foster the investigation of the SDN signaling load for productive deployments because they are a key metric for the performance of an SDN-enabled network. More specifically, the rate of signaling messages at various scenarios and stages of development is of interest for estimating performance. This applies both to small scenarios in which measurements can be taken, as well as to large, productive scenarios in which a large number of network flows have to be considered in order to explore the performance limits of an SDN deployment.

A second factor for the performance is the switch itself. As current generation network devices often lack the space to save an infinite amount of flow rules on the device, a so called time-out is introduced per stored flow. If a flow is inactive for a defined time, the flow is automatically deleted from the device, and, thus, frees table space. A new packet for a flow resets this timer, flows without a match in the table require further processing by the controller, thus creating additional signaling and load on the controller side and increasing the overall latency of that flow on the data perspective. This flow time-out is determined by the controller and has a huge influence on the signaling load between controller and device and the table occupancy on side of the networking device. Consequently, it is important to consider which rules should be stored in the flow table so that the performance of the SDN deployment cannot be degraded. Any table miss that causes additional signaling will affect performance.

Given a specific traffic mix, the choice of the time-out period affects the trade-off between signaling rate and table occupancy. On the one hand, small time-outs might result in unnecessary signaling traffic, control plane overload, and long response times for control actions [11]. On the other hand, long time-outs might cause a high flow table occupancy and table overflows. As a result,

providers have to adjust this parameter to enable a smooth and efficient network operation. The overall complexity of this problem including uncertainty of traffic mix and application characteristics does not allow a one-time optimization of the time-out period. Instead, the relevant system parameters have to be monitored in regular intervals and the time-out parameter has to be adjusted if necessary. To derive viable parameter settings in a timely manner, an appropriate abstraction is required.

For the commercial usage of a communication network the reliability and availability of an SDN deployment is of great concern to the operators. In particular, questions about fault detection and fault localization mechanisms of the new technology are important, since today, networks with a short update cycle and constant maintenance and modifications, need to provide techniques and mechanisms to ensure a guaranteed network operation and identify failures. Through the centralized nature of SDN, a big new point of failure has been introduced, namely the SDN controller, as it orchestrates the whole network. Next, the flexibility and availability of a central global knowledge offers many new possibilities and opportunities. In order to react in the most efficient manner, fast failure detection mechanisms are required.

From a technical point of view, this means that SDN is expected to provide measures and arrangements to deal with failures. Current state-of-the-art SDN controllers mostly build up on the limited detection capabilities of the southbound SDN protocol OpenFlow for their fast failure mechanisms. In SDN, after the switching device has detected a failure, a notification is sent to the controller, which, in turn, reacts according to its programming, i.e. if a link down event is detected by a switch, it reports that to the controller, which responds with new routing information for affected flows. This message contains the affected switch, the affected port of that switch and the new port status (link down, blocked or live).

Additionally, SDN offers capabilities for flow monitoring. For example, each OpenFlow-enabled switch offers counters, which store traffic information in different granularity, e.g. per-table, per-flow, and per-port (for example: byte/-

packet count per flow). The controller is able to query this data, and, if programmed to do so, react on them. Although offering basic statistics, these options also have several limitations. Frequent polling has to be used in order to monitor the dynamics of a flow. Moreover, in general, polling of statistics induces load on components, for example, a higher CPU load on the controller or a higher signaling traffic due to statistic packets. Therefore, these mechanisms are insufficient for large-scale, carrier-grade SDN deployments.

ONOS is one of the currently most common SDN controllers on the market. Its developers, the ON.Lab, present it as "*A new carrier-grade SDN network operating system designed for high availability, performance, scale-out*" [12]. In order to keep that bold statement, ONOS currently uses LLDP (Link Layer Discovery Protocol) packets to detect broken links. A typical questions that is often raised is whether the "five nines" of availability can be ensured. This expression describes the percentage an offered service is available throughout a year, in this case only a total downtime of 5.26 minutes is allowed. In order to comply with this, additional conditions are placed on the error detection. In order to limit the downtime, errors must be detected very quickly. This is currently one of the biggest challenges in SDN to enable the technology to be used in production environments. In the end, SDN currently lacks bot a specific and a fast error detection. Therefore, this monograph addresses the high availability claims of ONOS by analyzing the failure detection mechanism of the controller for challenging conditions in the data plane by using the means of mathematical analysis and measurements in a testbed.

Despite the paradigm change and all of its features that came with the introduction of SDN old problems still persist. Hence, another step in the evolution of networking was paramount. P4, short for *Programming Protocol-Independent Packet Processors*, attempts to provide a programmable interface that allows for flexible mechanisms that can parse packets and match against arbitrary fields at line rate. Instead of letting the underlying switching hardware *tell* the operator what it is capable to do and how, the operator now is able, thanks to a top-down design, to tell the hardware how it should behave and how it should switch pack-

ets, even after they are deployed. Furthermore, it is one of the prioritized goals to finally enable target-independence, both for hardware and software switches alike.

Many big data center operators have joined the P4 community and, therefore, the question whether this new technology is ready for production deployment is raised. The adding of tags, e.g. VLAN tags in order to enforce privacy and data security, is a very common task that is required in today's data centers and possible to realize with P4. This thesis runs an analysis of the processing performance of P4-capable hardware with varying configurations by launching measurements in a testbed.

All of these aspects and facets of SDN contribute to the production-readiness of this new technology. But with any new technology the questions arises whether new challenges are created, too. In general, in order to analyze the performance of a system generally three options exists: measurements in a testbed, simulation, and abstraction through the means of a model. For a detailed analysis of specific issues measurements in a small testbed are the tool of choice. Here, each environment parameter can be controlled and the effects of each configuration change and load on the system can be evaluated in a thorough investigation. For bigger deployments it is more feasible to switch to the means of simulation, where certain features can be abstracted in order to allow for a better manageability of the whole system under investigation. Depending on the requirements of the simulation its abstraction level can vary between the smallest detail and a very abstract representation of the system.

But, eventually, with scaling out the size of the network, the practicability, manageability, and feasibility both in time and money becomes disproportionately. Therefore, the means of an abstract model are the best option. From the perspective of performance analysis, analytical models are important to estimate and assess the performance of the control and data plane. In the end, they give guiding paradigms for managing, structuring, and deploying an SDN-enabled network. Closely linked to this, analytical models for SDN also enable a broader,

more holistic understanding of the factors that influence the performance of SDN networks.

1.2 Scientific Contribution

This thesis covers aspects that contribute to the production-readiness of SDN deployments. The first SDN topic is the performance impact of application traffic in the data plane on the control plane. Second, reliability and availability concerns of SDN deployments are exemplary analyzed by evaluating the detection performance of a common SDN controller. Thirdly, the performance of P4, a technology that enhances SDN, or better its impact of certain control operations on the processing performance is evaluated.

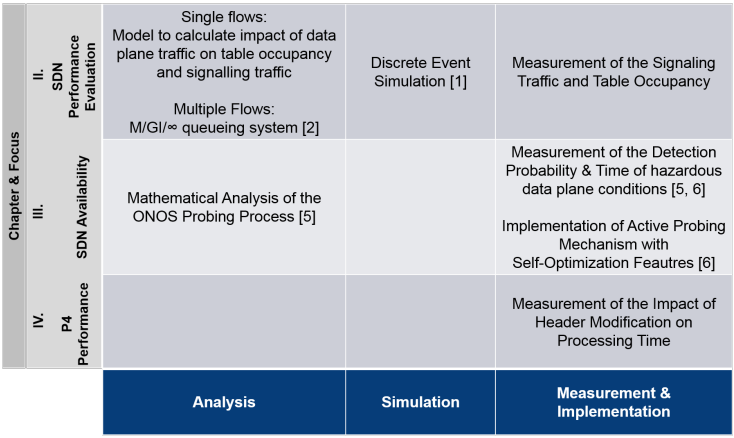


Figure 1.2: Topics discussed and methodology used in this thesis.

Figure 1.2 shows the various topics and methods presented in this thesis together with their scientific research publications. The y-axis shows the chapters and their focus of this monograph, the x-axis the used methodologies. Overall,

in order to cover a large spectrum of the covered topics, multiple methodologies are used which can be classified into mathematical analysis, simulations, and measurements and implementation. The respective focuses of the corresponding chapters are SDN performance evaluation, SDN availability, and P4 performance.

In the first part, the focus lies on the analysis of the impact of data plane traffic and the so called flow table time-out on the overall performance of SDN deployments. As, in general, networking devices do not have unlimited space for flow rules, the entries are usually provided with a time-out value describing the time after which inactive flows are automatically deleted from the table. As flows without a flow entry require the interaction with a controller which increases the processing latency and generates load on the controller, a trade-off between table occupancy and signaling load between controller and switch has to be found.

We create a model that allows us to calculate the effects of data plane traffic parameters and the flow table time-out with respect to the performance of an SDN deployment. At first, we use this model to understand the implications of these parameters for the case of a single TCP flow. Secondly, by adapting an M/GI/∞ queuing system, we extend this model to comprehend the relationships when multiple flows are in the system. Afterwards, the presented models are evaluated by a discrete event simulation and measurements in a testbed. The simulation focuses on package arrivals for each of the applications under investigation. The measurements, in turn, use a testbed that consists of one single SDN-enabled switch with a varying number of connected users. Connected to that switch is an SDN controller. As we are able to control the traffic parameters of the communication between the users in the data plane and the used time-outs in the switch, we are able to verify our model results by measuring the signaling traffic and the table occupancy.

In the second part, the availability of SDN deployments, more specific on the detection capabilities of one common open source SDN controller is of interest for the estimation whether SDN is ready for production environments. We

use the means of a stochastic analysis the detection mechanism of the ONOS controller in order to understand the relationship between the various configuration parameters and the detection time for the case of challenging network conditions, such as packet loss or jitter in the data plane. In order to verify the results measurements in a testbed are conducted.

This testbed consists of multiple switches arrange to a ring topology. Each of the switches is connected to one of two controller nodes that form a logically centralized, but physically distributed controller cluster. In this testbed we are able to control the network parameters, i.e. the data plane packet loss or jitter. Additionally, by measuring the controller-switch messages, we are able to verify our theoretical analysis of the probing mechanism. After having identified the shortcomings of the vanilla probing mechanism of ONOS, a more advanced active probing mechanism is designed, implemented and evaluated by measurements. Furthermore, as ever-changing network conditions pose a threat, a self-optimization feature is added which automatically optimized the probing process for the current network conditions.

Finally, one of the latest findings in the evolution of SDN is P4. Despite the benefits of the whole SDN development, still certain draw-backs exist. One of the more important shortcomings is the limitations of a deployment that comes with the choice of the switching hardware. Still, it is the usual case that the capabilities of the hardware determine the features that are usable in production environments, forcing the network manager into a so called bottom-up design. If a new protocol is introduced and the switch does not support processing the new header fields network administrators have no choice but to acquire new hardware or to pass on that protocol.

Here, P4 tries to change the approach by allowing for a top-down design. Now the network administrator is able to change the processing of packets by programs that determine the processing behavior of the switch. As this and other features sound very promising it is crucial to understand the possible performance implications of P4. Therefore, we conduct measurements in a testbed consisting of traffic generator and P4 hardware. While changing certain header

fields of incoming traffic, we measure the processing time within the hardware and attempt to understand the impact of the number and scale of the instructions on the overall performance.

1.3 Outline of the Thesis

After this introductory chapter the three mentioned topics are addressed in separate chapters. For each chapter a short introduction to the tackled challenge is given. In order to help understand the context of each topic, each chapter gives an overview about its backgrounds and related work. Afterwards, the problem of each chapter is analyzed and evaluated. Where applicable, the means of theoretical analysis, the creation of a model, verification by simulation or measurement in a testbed, is used. In the end of each chapter, the lessons learned are summarized. In the following, the organization of this monograph is described.

Chapter 2 addresses the impact of data plane traffic on the flow table utilization, and the trade-off of choosing the right flow-entry time-out value vs the imposed signaling load on the SDN controller. After the creation of models for single and multiple active parallel flows, it is evaluated by the means of simulation and measurement. Chapter 3 focuses on the reliability and availability aspects of SDN by analyzing the detection performance of the common ONOS SDN controller in hazardous situations in the data plane. After a theoretical analysis of the vanilla detection mechanisms and their vulnerability to packet loss and jitter in the data plane, a new active probing mechanisms is implemented and evaluated by the means of measurements in a test bed. As introduced earlier, P4 is the next step in the ongoing evolution of networking, and more precisely, SDN. Its backers claim that is enhance and simplifies many task, such as adding and removing tags, as it is common for example in data centers with VLAN tags. The impact of VLAN header addition and removal on the processing performance of TCP streams is evaluated in multiple configurations in Chapter 4. Finally, Chapter 5 concludes this thesis and provides an outlook to future research activities.

2 Analytical Model for SDN Signaling Traffic and Flow Table Occupancy

Software-defined Networking (SDN) is a new networking paradigm overcoming various drawbacks of current communication networks. The control and data plane of switching devices is decoupled and all control functions are centralized within the network controller(s). With this simple principle, new networking scenarios are possible, facilitated by the introduction of increased control capabilities, dynamic modification of networking parameters, or the control of traffic flows on different granularities. With this simple principle, a couple of new features like network programmability, the dynamic modification of network parameters or the control of traffic flows on different granularities are possible. Further, the applicability of these features is not restricted to a specific type of network, but can be used in access networks, data center networks, and wide area networks [8–10].

Old challenges can be addressed with SDN, such as the ossified network setup and operation, which represents a significant financial challenge for Internet service providers and a handicap for the development of cloud and Internet services. Consequently, operators are increasingly interested in including SDN principles within planning, dimensioning, and optimization of communications infrastructures. In production deployments, the question arises how network traffic affects the SDN environment with controller, control plane, data plane,

and switch. A performance analysis of the SDN technology is therefore essential.

From a technical perspective, currently, two modes of operation are discussed for SDN-enabled networks: proactive and reactive forwarding mode. In proactive mode, most of the traffic traversing the topology is assumed to be known, e.g., in a data center. Traffic flow rules can be pre-installed in the network. Reactive mode, in contrast, is ideal for highly dynamic traffic where no information on the traffic mix is previously known. In such a scenario, SDN rules are pushed to the switch based on incoming flows on the data plane. Typically, flow rules are then provided in a network-wide manner, i.e., controller traffic is not induced at each switch, but only at the first one. Thus, each new arriving flow at the switch triggers a signaling request to the controller. The controller defines a route through the topology and installs the new flow in the devices of the topology. Generally in SDN, multiple controllers per switch, with various controller architectures, are possible. Currently, multi-controller architectures are mostly used in order to increase the availability and reliability of the control plane. In order to reduce the possibility of contradicting commands sent from controllers to the switch, the OpenFlow protocol supports, e.g., a master- and slave role assignment of the connected controllers. Thus, a switch might have connections to multiple controllers with each of the connected controllers receiving updates from the data plane, but only the master controller is allowed to install new rules. In case a controller fails, the ownership of the affected switches is transferred to a controller which was formerly in slave role. In this chapter we will only work with one single controller instance. Nevertheless, the findings in this chapter are also applicable for multi-controller environments. Depending on the chosen controller architecture, only the signaling load on side of the switch increases, as multiple controllers need to be informed about changes.

To enable high performance of data plane switching decisions, flow rules are kept in a so called flow table with high-speed memory, such as ternary content-addressable memory (TCAM) and content-addressable memory (CAM). For example, a maximum flow table size on a switch is currently up to 1500 for HP

switches [13]. By modifying the matching process, e.g., by supporting only a limited number of header fields, the overall number can be increased to typically "some thousand rules". Since the flow table space is limited in terms of TCAM and CAM, unused flow rules should be removed. This is done with the help of a *time-out value* which can be configured at installation time of the flow. A flow is removed exactly when the inter-arrival times of packets within the flow exceed the time-out value. If the packet inter-arrival time is larger than the time-out, the flow entry will be discarded. In case of additional new packet arrivals of this flow, the controller is then again involved in the forwarding decision resulting in additional control plane traffic and waiting times for the data plane traffic.

Given a specific traffic mix, the choice of the time-out period affects the trade-off between signaling rate and table occupancy. On the one hand, small time-outs might result in unnecessary signaling traffic, control plane overload, and long response times for control actions [11]. On the other hand, long time-outs might cause a high flow table occupancy and table overflows. As a result, providers have to adjust this parameter to enable a smooth and efficient network operation. The overall complexity of this problem including uncertainty of traffic mix and application characteristics does not allow a one-time optimization of the time-out period. Instead, the relevant system parameters have to be monitored in regular intervals and the time-out parameter has to be adjusted if necessary. To derive viable parameter settings in a timely manner, an appropriate abstraction is required.

In this chapter, an analytical model is presented which can be used to analyze the outlined key performance parameters of SDN in reactive mode. It was initially presented and described in [5]. Based on various traffic mixes and applications, different scenarios in the reactive forwarding mode of the controller are investigated. With the model we study the control plane traffic between the controller and switch. Following the same principle, we also derive the utilization of the flow table in the switch. We use the model in the form for multiple TCP flows and evaluate the impact of different TCP flow characteristics and

SDN time-outs on the SDN controller traffic and the SDN flow table occupancy in a realistic scenario. We investigate a scenario in which network flows are installed upon packet arrival by the controller and are discarded after a certain time-out period. Using the example of four different applications, the signaling rate and the table occupancy is then determined for those four applications. The influence of the time-out on both parameters is shown. Finally, lessons learned are discussed for the dimensioning of the time-out value with respect to the applications.

In the last section of this chapter, the model is validated for a traffic mix consisting of several applications in the network. For this purpose, a discrete event simulation was implemented, which simulates package arrivals for each of the applications under investigation, and a testbed for measurements with applied components was created. The results show the signaling volume when several types of applications are active at the same time in the network. The same is evaluated for the flow table occupancy in order to obtain an overall view on the behavior of the two parameters.

The contribution of this chapter is threefold. Firstly, we provide the analytical model as a tool ready to use for a network operator to analyze control plane traffic and flow table utilization. Secondly, the model is validated using a discrete-event simulation and testbed measurements. Thirdly, we show the impact of traffic flow characteristics on control plane traffic for different scenarios. These scenarios include single flows, multiple concurrent flows from one application type, and multiple concurrent flows from different application types.

The findings of this chapter are based on content that is published in [5, 1] and is structured as follows. In Section 2.1, related work on SDN and network performance analysis is discussed. Section 2.2 presents the analytical SDN model as well as the used methodology. In Section 2.3, we evaluate the impact of different TCP flow characteristics and time-outs on the controller traffic and the flow table occupancy in realistic scenario. Section 2.4 shows the simulation and the measurements for validating the model. Additionally, the simulation is applied together with the model for an extended scenario, where several applications

with different characteristics are active simultaneously in the network. Finally, Section 2.5 concludes this chapter.

2.1 Related Work

This section features work and research with the focus on modeling switch-controller traffic and the impact of different time-out values.

The authors of [14] modeled the basic OpenFlow switch model based on M/M/1-S queues for switch, controller, and the interaction between these two elements, in order to analyze forwarding speed and blocking probabilities. Their results indicate, that the packet sojourn time in an OpenFlow-enabled network is mainly dependent on the controller. Mahmood *et al.* extended this work, as it is only viable for one single forwarding element and lacks correctness for highly bursty network traffic [15]. Jarschel *et al.* enhance the model of [14] to the case of multiple nodes by using an Open Jackson network in [16]. The presented model is used to evaluate an SDN system's average packet sojourn time. Afterwards, they validate their model by simulation for the case of serial topologies. Additionally, the impact of bursty traffic on the models output can be analyzed. [17] created an OpenFlow-based queuing model that provides the average packet sojourn time through a switch in large-scale OpenFlow networks. Their numerical analysis concludes that the packet sojourn time mainly depends on the packet processing capability of the controller.

To demonstrate the correctness of their model, multiple measurements matching the results have been conducted. Azodolmolky *et al.* [18] present an analytical model to investigate the impact of composed application traffic on the interaction between switch and controller by using network calculus. The presented switch model captures the packet delay and the buffer length inside the SDN switch according to the parameters of a cumulative arrival process. In [19] an analytical network calculus model focusing on an upper bound for the processing delay of unknown flows arriving at a switch is created. The results of the model are validated by simulation. According to the authors of [20] their

model presents a faster way to predict the performance of SDN for any type of SDN deployment than benchmarking tools. In their paper, they utilize stochastic network calculus methods to analyze and evaluate the performance of an SDN deployment. To validate their results, they ran simulations and measurements in a testbed. Miao *et al.* [21] introduce an analytical model for the SDN architecture by using Markov-Modulated Poisson Process arrivals. According to their results, key performance indicators such as average latency and network throughput can be predicted by their model. An extensive OMNeT++ simulation experiment approves their results.

Beigi-Mohammadi *et al.* propose a model to study the efficiency and scalability of an application-aware software-defined infrastructure in [22]. After presenting the model, the authors conduct testbed measurements to verify the validity of their work. While being able to identify a bottleneck in the scalability of their cloud testbed, the authors do not consider the scalability of the flow table space used.

Analyzing different mechanisms of flow table updates, Liu *et al.* [23] study the impact of bandwidth and flow table size on the performance of flow updates. Both, qualitative and quantitative analysis of these trade-offs in several realistic network topologies are presented. In contrast to our work, the authors focus on the updates of already existing flows within a flow table. Therefore, possible flow table space problems are not considered.

All these papers do not analyze neither flow time-out, nor table occupancy. The impact of flow table time-out length on performance and table occupancy through measurements is analyzed in [24]. Additionally, multiple caching algorithms for a flow table are compared and evaluated. Their results indicate that, with an increasing time-out, the probability of an arriving packet triggering a request to the controller decreases exponentially, whilst the table size grows linearly. Depending on the characteristics of the data plane traffic, the authors are also able to identify good starting points for the time-out value: 5 and 10 seconds. These values, though, are not put in relation to the characteristics of the analyzed traffic. Based on their observations, the authors propagate a dy-

namically chosen time-out value. According to [25], one of the main scalability problems of SDN controllers is that the controller is often simply overwhelmed by the number of requests. To overcome this issue, the authors propose to adjust flow time-outs based on the mean inter-arrival time of packets per flow. Their results indicate that the dynamic modification of the time-outs, in dependence of the quality of their prediction, may decrease the controller load by almost 10%. Zhu *et al.* point out the importance of suitable time-outs for each flow as well as a load awareness of the flow table in [26]. They propose a mechanism to assign flow time-outs according to flow characteristics. Additionally, a feedback control to dynamically adjust the max time-out value according to the current load of the flow table is presented. Kim *et al.* [27] choose an LRU caching algorithm to reduce the table-miss rate of the switch. Thus, the controller load can be reduced. In [28] rule-caching is also used to increase the flow table-hits. Their design is based on four criteria: elasticity, transparency, fine-granular, and adaptability, and satisfies their requirements. Kuzniar *et al.* [29] describe important performance characteristics of flow tables from different manufactures by measurements. Their goal is to make controllers' use of flow tables more efficient. The main outcome of this work is that OpenFlow switches, although implementing the same OpenFlow protocol version, differ widely.

Several important work has been done in the area of modeling network traffic, which lays the foundations for analyzing the impact of flow entry time-out on the overall performance in SDN. Ciucu *et al.* [30] present an overview on the general usability and applicability of network calculus for modeling the performance of networks. In 1998 and 2000, Feldmann *et al.* presented fundamental work for modeling WAN traffic [31, 32]. The approach we are presenting in the next chapter features a universal analysis and is easily modifiable to a custom architecture. In order to imitate different general arrival processes, we adopt a two-moment substitution as proposed in [33], using Markovian arrival processes. In contrast to the Markov property, it has been shown that there is a long-range dependency in network traffic, as noted in [34]. Andersen *et. al.* created a model to represent these findings in superpositions of two-state Markov-modulated

Poisson processes [35]. Whitt *et. al.* also present a candidate for source traffic models [36, 37].

In [5] we introduced our model to analyze the impact of the flow table timeout value T_0 on the controller signaling rate of multiple applications, i.e. how many requests per second are triggered by an application, and the overall switch table occupancy, i.e. what is the percentage of time a flow is actually stored in the flow table. As shown in our results, the model produces valid output for the case of one single application generating one or multiple flows at the same time. In [1] we presented an enhanced version and validated the model with discrete event simulations. In this chapter we want to enhance those findings by showing measurements that underline the correctness of the model.

2.2 Modeling Concept, Methodology and Analysis

2.2.1 Scenario Description

For this analysis, we consider a single SDN switch, which is connected to a reactive SDN controller, cf. Figure 2.1. Multiple TCP flows are active in the network, thus, packet streams arrive at the SDN switch and have to be forwarded. The presented model can be applied for any flow-based traffic with known packet inter-arrival time. In this chapter we focus on TCP flows.

At the start of operation the SDN switch has no knowledge on how to handle any arriving packet. When the first packet of a TCP flow arrives, the SDN switch produces a flow table miss and sends a request on how to handle the new flow (`Packet_In` message) to the SDN controller. The SDN controller replies with a flow rule, which is stored in the flow table of the SDN switch. Any successive packet belonging to the same flow can now be processed by the switch independently. Due to the limitations in flow table size, flow rules cannot be kept forever in the switch. Therefore, current implementations of SDN switches discard entries after these entries have been rendered useless.

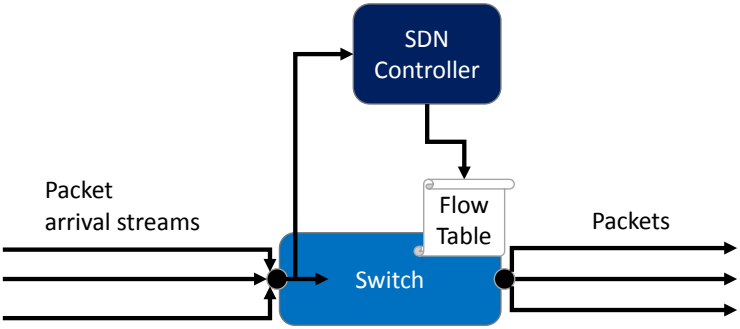


Figure 2.1: SDN switching model.

Accordingly, the arrival of a packet starts a time-out period T_0 for the given flow rule. If the next packet of the flow arrives before T_0, the time-out period is restarted. If no packet arrives within T_0, the flow rule is discarded by the switch. If another packet of this flow arrives after T_0, the packet will cause a flow table miss, and thus, the described procedure repeats. T_0 can be set to an arbitrary time-out value between 0 and `Integer max`. A value of 0 indicates an infinite idle time-out (no idle time-out condition), any other value a time-out value in seconds [38].

2.2.2 Single Flow Model

First, we investigate the situation in which a packet stream of a single TCP flow arrives at the SDN switch. Figure 2.2 illustrates the situation and introduces the used variables. We assume that the inter-arrival times A of the packets of the TCP flow follow a general independent distribution $A(t) = P(A \leq t)$. We divide the TCP flow into subflows, which are characterized as the time periods, from the setting of the flow rule to its time-out to the subsequent setting of the flow rule. This means, based on the time-out T_0, the subflow contains packets

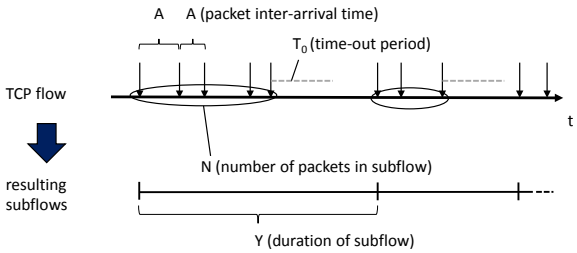


Figure 2.2: Illustration of resulting subflows and their characteristics, i.e., number of packets N and duration Y, based on the inter-arrival time distribution A of the TCP flow and the time-out period T_0 of the SDN switch.

with inter-arrival times smaller than T_0 until the time-out of the flow rule, and continues until the next packet starts a new subflow. The following parameters will be used and introduced in the following sections:

- N: The random variable of the number of packets in a subflow
- Y: The random variable of the duration of a subflow (in seconds)
- η_Y: The signaling rate, i.e., the inter-arrival time of requests towards the controller (in 1/seconds)
- ρ_Y: The table occupancy, i.e., the percentage of time a flow rule is present in the flow table

As a subflow starts with one packet and every subsequent packet belongs to the same subflow if it arrives within the time-out period T_0, or else the subflow ends, the number of packets in a subflow N follows a geometric distribution. For shortening reasons, we introduce α as the probability that the inter-arrival time is less or equal than T_0, i.e., $\alpha := P(A \leq T_0) = A(T_0)$. Throughout this chapter, we assume α to be lower than 1, otherwise the single resulting subflow

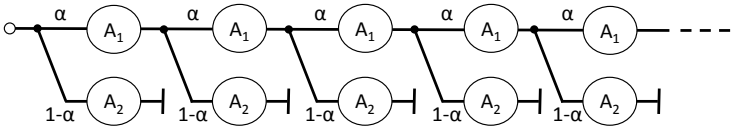


Figure 2.3: Phase diagram for the composition of Y.

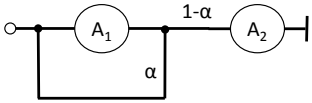


Figure 2.4: Feedback loop for subflow duration Y.

would be identical to the original flow. Eventually, the distribution of N is given by Equation 2.1

$$P(N = k) = \alpha^{k-1} \cdot (1 - \alpha), \quad k \in \{1, 2, \ldots\} \quad . \tag{2.1}$$

This insight helps to derive the duration of a subflow Y. We define $A_1 := \frac{A \cdot 1_{\{t \leq T_0\}}}{\alpha}$ as a random variable with the truncated conditional distribution, which gives the inter-arrival time of the packets in case the arrival is less or equal to T_0. Moreover, we define $A_2 := \frac{A \cdot 1_{\{t > T_0\}}}{1-\alpha}$ as the corresponding random variable in case the arrival is greater than T_0. Then, we consider the subflow duration Y as depicted in Figure 2.3. Y can be iteratively composed of A_1 phases, such that with probability α an A_1 phase is added to Y, until the subflow times out with a phase A_2 with probability $1 - \alpha$. Consequently, the random number of A_1 phases in Y follows the shifted geometric distribution $N' := N - 1$. Thus, Y can be written as a sum of a random number of random variables:

$$Y = A_{1(1)} + A_{1(2)} + \ldots + A_{1(N')} + A_2 \quad . \tag{2.2}$$

Figure 2.3 can be transformed into a feedback loop as depicted in Figure 2.4. This can be handled by means of standard control theory, which gives the Laplace transform $\Phi_Y(s)$ as presented in Equation 2.3

$$\Phi_Y(s) = \frac{(1-\alpha)\cdot\Phi_{A_2}(s)}{1-\alpha\cdot\Phi_{A_1}(s)} \quad . \tag{2.3}$$

We can now compute the moments of Y to obtain the expectation and coefficient of variation of Y in Equation 2.4 depending on A_1 and A_2. The signaling rate η_Y, indicating the rate of requests arriving at the controller, is thus the inverse of the average subflow duration. It can be seen that the characteristics of Y depend on higher moments of A_1 and A_2. Obviously, these moments are influenced by the characteristics of the packet arrival process A of the TCP flow and the threshold T_0 of the switch. We will investigate this relationship in detail in Section 2.3 by utilizing substitute arrival processes, which are introduced in Section 2.3.1.

$$\begin{aligned}
E[Y] &= -\Phi_Y'(0) = \frac{\alpha}{1-\alpha}E[A_1] + E[A_2] \quad , \\
\eta_Y &= \frac{1}{E[Y]} = \frac{1}{\frac{\alpha}{1-\alpha}E[A_1] + E[A_2]} \quad , \\
E[Y^2] &= \Phi_Y''(0) \\
&= \frac{2\alpha^2 E[A_1]^2}{(1-\alpha)^2} + \frac{2\alpha E[A_1]E[A_2] + \alpha E[A_1^2]}{1-\alpha} + E[A_2^2] \quad , \\
Var[Y] &= E[Y^2] - E[Y]^2 = \frac{\alpha E[A_1^2] - \alpha^2 Var[A_1]}{(1-\alpha)^2} + Var[A_2] \quad , \\
c_Y &= \frac{\sqrt{Var[Y]}}{E[Y]} \\
&= \frac{\sqrt{Var[A_2](1-\alpha)^2 + \alpha E[A_1^2] - \alpha^2 Var[A_1]}}{\alpha E[A_1] + (1-\alpha)E[A_2]} \quad .
\end{aligned} \tag{2.4}$$

The removal of each subflow from the switch table after the time-out T_0 has the characteristics of an on-off-process. The on-phase represents the time in which the flow rule is stored in the switch table, and its random variable Y_{on} can be computed by substituting A_2 in the above calculations with the deterministic random variable of the time-out T_0[1]. The off-phase, being the time in which the flow rule is not stored in the switch table, is given by the random variable $Y_{off} := A_2 - T_0$. Consequently, $Y = Y_{on} + Y_{off}$, and the switch table occupancy ρ_Y for a subflow Y, i.e., the percentage of time a flow rule is present in the flow table, can be computed by Equation 2.5

$$\rho_Y = \frac{E[Y_{on}]}{E[Y]} = \frac{\frac{\alpha}{1-\alpha}E[A_1] + T_0}{\frac{\alpha}{1-\alpha}E[A_1] + E[A_2]} \quad . \tag{2.5}$$

2.2.3 Composite Model - The Case with Multiple TCP Flows

After characterizing the subflows of a single TCP flow, we now transfer our findings to the case with multiple TCP flows. Typically, not all users in a network topology are simultaneously active. Based on the analysis in the previous subsection on arrival-rate and service time of a single user, we now draw conclusions about the number of simultaneous users in a system.

Therefore, we assume a memoryless arrival process of TCP flows with rate λ, each being active for a certain time following a general independent distribution B. Moreover, we assume that no TCP flow has to wait or is blocked, which resembles an M/GI/∞ queuing system. Thus, the number F of currently active flows follows a Poisson distribution given in Equation 2.6 with mean $E[F]$. The generated signaling traffic at the SDN controller is a superposition of all requests created by the subflows of the set of active TCP flows. This means, the total rate η at which requests are generated is the sum of the rates of each active TCP

[1]We substitute the notation T_0 for sake of simplicity, instead of formally defining a deterministic random variable T with $T(t) = P(T \leq t) := \Theta(t - T_0)$, where Θ refers to the Heaviside step function.

flow. In case all TCP flows follow the same characteristics and have the same signaling rate $\eta_{Y_i} = \eta_Y$, $\forall i \in F$, $E[\eta]$ can be computed directly from the expected number of active TCP flows $E[F]$.

$$\begin{aligned} P(F=k) &= \frac{(\lambda E[B])^k e^{-\lambda E[B]}}{k!} \quad , \\ E[F] &= \lambda E[B] \quad , \\ \eta &= \sum_F \eta_{Y_i} \quad , \\ \text{If } \eta_{Y_i} &= \eta_Y,\ \forall i \in F: \\ \eta &= F \cdot \eta_Y \quad , \\ E[\eta] &= E[F] \cdot \eta_Y = \lambda E[B] \eta_Y \quad . \end{aligned} \tag{2.6}$$

The occupancy of the flow table at the SDN switch, i.e., the number of entries in the table T, can be expressed as sum of a random number of indicator variables. They indicate for each of the F active flows whether it is in the on-phase, and thus, a rule is stored in the flow table. The distribution of the occupancy of the flow table in case of $F = k$ active flows, i.e., $P(T = m|F = k)$, can be expressed by means of the Poisson binomial distribution as presented in Equation 2.7, where F_m is the set of all subsets of m integers that can be selected from k greater than or equal to m integers. This formula can again be simplified with a binomial distribution in case all TCP flows follow the same characteristics, which also gives an expectation for T.

$$\begin{aligned} T &= \sum_F 1_{Y_{i,on}} \quad , \\ P(T=m|F=k) &= \\ &= \begin{cases} 0, & k < m \\ \sum_{M \in F_m} \prod_{i \in M} \rho_{Y_i} \cdot \prod_{j \in \overline{M}} (1 - \rho_{Y_j}), & k \geq m \end{cases} . \end{aligned} \tag{2.7}$$

$$
\begin{aligned}
&\text{If } \rho_{Y_i} = \rho_Y,\ \forall i \in k: \\
&\quad T = F \cdot 1_{Y_{on}} \quad , \\
&\quad P(T = m | F = k) = \binom{k}{m} \cdot \rho_Y^m \cdot (1 - \rho_Y)^{k-m} \quad , \\
&\quad E[T] = E[F] \cdot \rho_Y = \lambda E[B] \rho_Y \quad .
\end{aligned} \tag{2.7}
$$

2.2.4 Application Mixes Model

For application mixes ω, we again assume independent Poisson arrivals of flows with rate λ and exponentially distributed flow duration with mean $E[B]$. For each flow, the type of application i is modeled using uniformly distributed random variables according to the fixed flow probability $\omega_i,\ 0 \leq \omega_i \leq 1$, $\sum_i \omega_i = 1$. The results for the application mix η_ω, ρ_ω can then be computed by the weighted sum of the signaling rates η_i or table occupancies ρ_i of the single application model for each application i:

$$
\begin{aligned}
\eta_\omega &= \sum_i \omega_i \cdot \eta_i \cdot \lambda \cdot E[B] \quad , \\
\rho_\omega &= \sum_i \omega_i \cdot \rho_i \cdot \lambda \cdot E[B] \quad .
\end{aligned} \tag{2.8}
$$

In Section 2.3, the deduced characteristics for a single and multiple TCP flows will be evaluated in a realistic environment. In particular, the rate of requests at the SDN controller and the occupancy of the flow table at the SDN switch will be analyzed for the presented scenarios.

2.3 Evaluation

We evaluate the impact of different TCP flow characteristics and SDN time-outs on the SDN controller traffic and the SDN flow table occupancy in a realistic scenario. In the work of Gebert *et al.* [11], we find Table 2.1, which provides

inter-arrival times of TCP flow characteristics of four diverse mobile applications. It can be seen that the mean inter-arrival times of packets $E[A]$ can be as low as tens of milliseconds, e.g., in case of the music streaming service Aupeo, but can also extend to the order of some seconds, e.g., in case of browsing the social network Twitter. Also, the coefficient of variation c_A is application-specific and rather low in case of video chat application Skype. In contrast, very bursty arrivals with high c_A were measured for the game app Angry Birds and Aupeo. Thus, we will focus the evaluation on the observed ranges of $E[A]$ and c_A. To demonstrate the correctness of our results, they are cross-validated by measurement and simulation in Section 2.4.

Table 2.1: Mean inter-arrival time $E[A]$ and coefficient of variation c_A of packet arrivals for different applications [11].

Application	$E[A]$	c_A
Twitter	8.91	4.95
Skype	0.55	3.55
Aupeo	0.06	51.00
Angry Birds	0.66	24.09

2.3.1 Substitute Arrival Processes

To imitate different general arrival processes for packets of a TCP flow, we adopt a two-moment substitution as proposed in [33]. This means, to obtain a desired expectation $E[A]$ and coefficient of variation c_A of packet arrivals, the following substitute distribution functions are used:

Case 1: $0 < c_A \leq 1$

$$A(t) = \begin{cases} 0, & 0 \leq t < t_1 \\ 1 - e^{-(t-t_1)/t_2}, & t_1 \leq t \end{cases} \tag{2.9}$$

where $t_1 = E[A](1 - c_A)$ and $t_2 = E[A]c_A$.

Case 2: $1 < c_A$

$$A(t) = 1 - p \cdot e^{-t/t_1} - (1-p) \cdot e^{-t/t_2}$$
$$\text{where } t_{1,2} = E[A]\left(1 \pm \sqrt{\frac{c_A^2 - 1}{c_A^2 + 1}}\right)^{-1} \tag{2.9}$$
$$\text{and } p = E[A]/2t_1,\ pt_1 = (1-p)t_2.$$

The advantage of these substitute arrival processes is their mathematical tractability, thus, the moments of A_1 and A_2 can be calculated based on three parameters $E[A]$, c_A, and T_0 as presented in Equation 2.10. This allows to obtain the signaling rate at the SDN controller from Equation 2.4 and the flow table occupancy from Equation 2.5.

Case 1: $0 < c_A \leq 1$

$$E[A_1] = t_2 + T_0 + \frac{t_1 - T_0}{1 - e^{-(T_0 - t_1)/t_2}} \quad ,$$
$$E[A_1^2] = t_2^2 + \frac{(t_1 + t_2)^2 - (T_0 + t_2)^2 e^{-(T_0 - t_1)/t_2}}{1 - e^{-(T_0 - t_1)/t_2}} \quad ,$$
$$E[A_2] = T_0 + t_2 \quad ,$$
$$E[A_2^2] = (T_0 + t_2)^2 + t_2^2 \quad .$$

Case 2: $1 < c_A$ (2.10)

$$E[A_1] = \frac{p(t_1 - (T_0 + t_1)e^{-T_0/t_1})}{1 - pe^{-t/t_1} - (1-p)e^{-t/t_2}} \cdots$$
$$\cdots + \frac{(1-p)(t_2 - (T_0 + t_2)e^{-T_0/t_2})}{1 - pe^{-t/t_1} - (1-p)e^{-t/t_2}} \quad ,$$

$$
\begin{aligned}
E[A_1^2] &= \frac{p(2t_1^2 - ((T_0+t_1)^2 + t_1^2)e^{-T_0/t_1})}{1 - pe^{-t/t_1} - (1-p)e^{-t/t_2}} \cdots \\
&\cdots + \frac{(1-p)(2t_2^2 - ((T_0+t_2)^2 + t_2^2)e^{-T_0/t_2})}{1 - pe^{-t/t_1} - (1-p)e^{-t/t_2}} \quad , \qquad (2.10) \\
E[A_2] &= T_0 + \frac{pt_1 e^{-T_0/t1} + (1-p)t_2 e^{-T_0/t2}}{pe^{-t/t_1} + (1-p)e^{-t/t_2}} \quad , \\
E[A_2^2] &= T_0^2 + \frac{2pt_1(T_0+t_1)e^{-T_0/t1}}{pe^{-t/t_1} + (1-p)e^{-t/t_2}} \cdots \\
&\cdots + \frac{2(1-p)t_2(T_0+t_2)e^{-T_0/t2}}{pe^{-t/t_1} + (1-p)e^{-t/t_2}} \quad .
\end{aligned}
$$

2.3.2 Single Flow Model Analysis

Using the above described substitute arrival processes, we analyze the resulting SDN controller traffic and the SDN flow table occupancy for different packet arrival streams and time-outs.

Figure 2.5 shows the arrival rate η of requests originating from a single TCP flow on the y-axis. The mean packet inter-arrival time $E[A]$ is depicted on the x-axis, and the different curves depict the results for different coefficients of variation c_A ranging from 0 (black) to 51 (yellow). The time-out T_0 is set to 10 s, which is a typical default value set by SDN controllers, e.g., NOX [39], and is also used as the default value in the Stanford OpenFlow deployment and DevoFlow [40][2]. In the following the three investigated value ranges of c_A are discussed.

- $c_A = 0$:
  In the deterministic case, no flow ever times out when its packet inter-arrival time $E[A]$ is lower than T_0, and thus $\eta = 0$. However, if $E[A] > T_0$, every packet will start a new subflow, but because $\eta = 1/E[A]$ in

[2] The latest SDN controllers OpenDaylight and ONOS use different values in their default configuration: 1800 seconds and ∞, respectively.

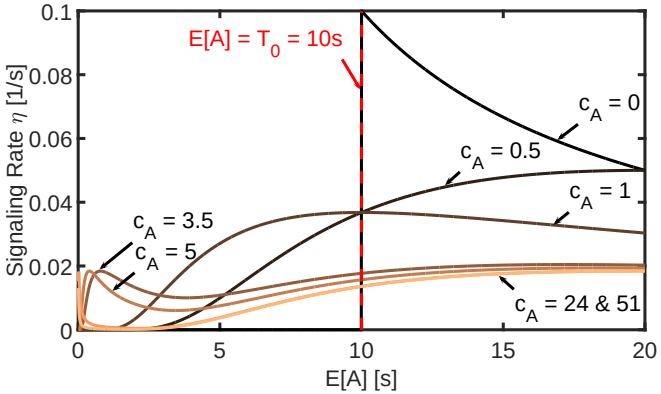


Figure 2.5: Arrival rate η of requests at SDN controller for fixed $T_0 = 10$ s depending on characteristics of TCP flow, i.e., the packet arrival process A.

this case, the traffic at the SDN controller will decrease with increasing $E[A]$. For very large $E[A] \gg T_0$ (not depicted), all curves will eventually approximate $\eta = 1/E[A]$, as each arrival is increasingly more likely to be larger than T_0 and start a new subflow.

- $0 < c_A \leq 1$:
  In the hypoexponential and exponential case, the signaling rate increases monotonically to a maximum before the rate merges ($0 < c_A < 1$, e.g., $c_A = 0.5$ at $E[A] = 20$) or converges ($c_A = 1$) towards $\eta = 1/E[A]$, respectively.

- $c_A > 1$:
  In the hyperexponential case, we observe that η has a first local maximum for small $E[A]$, then decreases to a local minimum, and increases to a second maximum before it eventually converges for large $E[A]$. The higher c_A, the more the first maximum is shifted to smaller $E[A]$, and

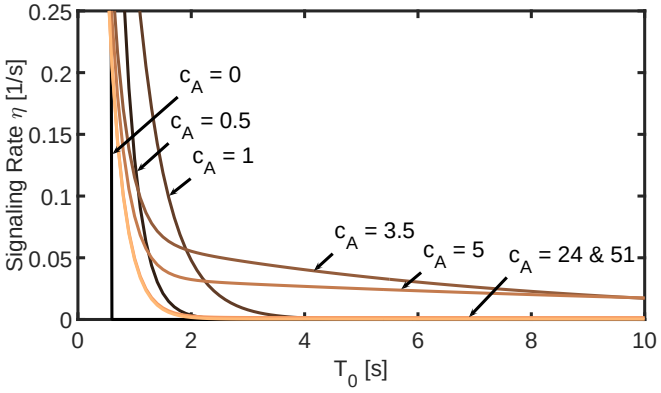


Figure 2.6: SDN controller traffic arrival rate η depending on SDN switch time-out T_0 for fixed $E[A] = 0.55$ s.

the smaller the first and second maxima. With increasing c_A, the curves converge, which can be seen from the overlap of $c_A = 24$ and $c_A = 51$ for $E[A] > 3$. The envelope of these curves gives the maximum η for $E[A] < T_0$ independent of c_A. Thus, we see that different TCP flow characteristics influence the arrival rate of requests at the SDN controller in case of a fixed T_0. In the interesting region for E[A] smaller or slightly higher than T_0, we observe that burstiness can decrease the SDN controller traffic for high $E[A]$, while burstiness increases for flows with small $E[A]$.

Taking a look at the impact of the time-out T_0, we fix $E[A] = 0.55$ s (cf. Skype in Table 2.1) in Figure 2.6. The time-out T_0 is varied on the x-axis and the different curves indicate again different coefficients of variation c_A of the packet arrivals in the TCP flow. All curves show a monotonically decreasing behavior towards 0 when T_0 becomes larger than $E[A]$. In order to discuss this evaluation step by step, a division after the three c_A ranges is made and discussed.

- $c_A = 0$:
  In the deterministic case, the signaling rate drops from a constant $\eta = 1/E[A] = 1.82$ to 0 at $T_0 = E[A]$, because no flow will time out if $T_0 > E[A]$.

- $c_A < 1$:
  Starting from this asymptotic curve, for hypoexponential arrival processes, the gradient will become smaller if c_A increases and the rates will converge slower towards 0.

- $c_A \geq 1$:
  In the exponential ($c_A = 1$) and hyperexponential cases, two intertwined effects cause the non-intuitive behavior visible in Figure 2.6 that the curves for very small and very high c_A show a fast convergence towards the asymptotic function, while the curves in between form an envelope and converge more slowly. First, when c_A surpasses from 1, the gradient of the curve transforms more quickly from a larger descent into a flatter slope. This will slow down the convergence towards 0, and can be seen when comparing the curves for $c_A = 1$, $c_A = 3.5$, and $c_A = 5$. At the same time, when c_A increases, the descent starts earlier, which brings the curves' points closer to the asymptotic function ($c_A = 0$). This will speed up again the convergence towards 0 for high c_A and can be observed when comparing the curves for $c_A = 5$ and $c_A = 24$. The envelope function of this group of curves constitutes an upper limit for the signaling traffic for given $E[A]$ and T_0.

Based on these two figures, several observations can be made concerning the dimensioning of T_0. As long as $E[A] < T_0$ the signaling rate of an application flow is acceptably small, especially for $E[A] \ll T_0$. The coefficient of variation c_A only seems to play a minor role in these constraints, especially for really high values of c_A the signaling load is negligible. Therefore, controller interaction for processing this flow is kept to a minimum. In general, a higher c_A of an applications flow renders lesser load on a controller.

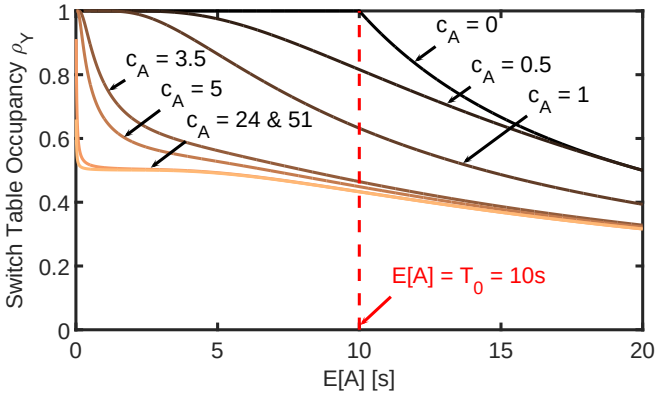


Figure 2.7: SDN switch flow table occupancy ρ for time-out T_0 = 10 s and different TCP flow characteristics.

The flow table at the SDN switch is populated by a flow rule, when the TCP flow is in an on-phase, i.e., a subflow of the TCP flow has not timed out. Figure 2.7 depicts the table occupancy ρ for an SDN switch with time-out T_0 = 10 s depending on the mean packet inter-arrival time $E[A]$, which is plotted on the x-axis, again for different values of c_A.

Two effects can be clearly seen. First, all curves are monotonically decreasing, such that a higher $E[A]$ leads to a lower occupancy ρ in the flow table. This is due to the fact that a higher $E[A]$ increases the probability of a flow time-out when the next packet arrives later than T_0. Larger $E[A]$ will also contribute to longer off-phases, which decreases the flow table occupancy. Second, the higher c_A, i.e., the more bursty the packet arrival process, the lower ρ, due to longer periods between two bursts, which will more likely cause a flow time-out and a long off-phase.

In the extreme case of $c_A = 0$, the occupancy ρ is 1 if $E[A] < T_0$, and decreases hyperbolically with $\rho = \frac{T_0}{E[A]}$ if $E[A] \geq T_0$, which is the asymp-

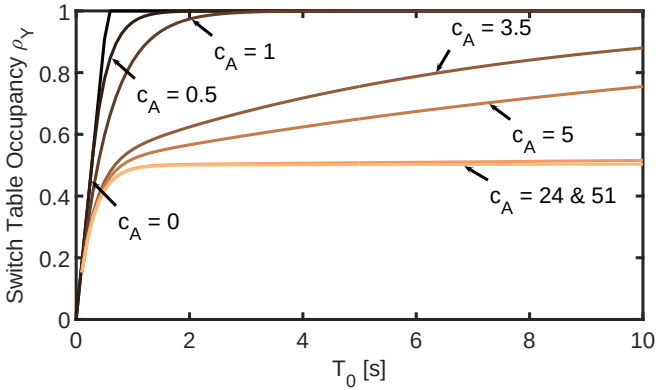


Figure 2.8: Impact of time-out T_0 on SDN switch flow table occupancy of a single TCP flow with $E[A] = 0.55$ s.

totic function in this plot. In the hypoexponential case ($c_A < 1$), the larger the deterministic share of the substitute process (i.e., the smaller c_A), the sooner convergence occurs. In the plot, the convergence for $c_A = 0.5$ is visible, when the curve overlaps with the asymptotic function for $E[A] > 18$. Eventually, the higher c_A, the earlier the drop of table occupancy and the more inert the convergence towards the asymptotic function.

Figure 2.8 depicts the impact of the time-out T_0 on the flow table occupancy of a single TCP flow for a fixed $E[A] = 0.55$ s. The x-axis shows the time-out T_0, and the y-axis presents the resulting occupancy ρ for different c_A. The smaller T_0, the more often TCP flows will time out and free the occupied space in the flow table. It can be seen that the choice of T_0 has more impact for TCP flows with small coefficient of variation. The resulting occupancies for small c_A range up to 1, i.e., there are time-outs T_0, for which the flow rule will never be discarded. For TCP flows with high c_A, the occupancy will increase very slowly for increasing T_0 since the flows are generally more likely to time out.

For example, a flow with $c_A = 51$ does not reach a higher occupancy than 50% throughout the investigated range of T_0.

Figures 2.7 and 2.8 demonstrate that a change in the parameter T_0 also has a significant impact on the switch table occupancy ρ. The general conclusion is that a lower T_0 value decreases the occupancy. Keeping the previous results for the signaling rate η in mind, it is to beneficially trade-off between signaling rate and table occupancy. For the values investigated, T_0 = 3 s offers a good solution: independent of the mean inter-arrival time E[A] of an application and its coefficient of variation c_A, both signaling load and table occupancy are at acceptable levels. A higher T_0 would lead to a higher occupancy, a lower value to higher signaling traffic towards the controller, which, in turn, may bring up another undesirable effect: overload at the controller.

2.3.3 Trade-offs between Signaling Load and Table Occupancy in the Case of Multiple Flows

At the SDN switch, multiple TCP flows arrive, which will contribute to the signaling rate at the SDN controller and the occupancy of the flow table. In this section, we will investigate this behavior and the resulting trade-offs for the four apps described above (cf. Table 2.1). From a previous work [41], we have taken several numerical values for the composite model. Based on an extensive measurement of Internet access in dormitories, the authors observed an arrival rate of TCP flows of $\lambda = 158.73\frac{1}{s}$ and a mean TCP flow duration of $E[B] = 234.95$s. The numbers from [41] are used in our composite M/M/∞ model to compute the average number of active TCP flows $E[F] = 37293.6$ in the evaluation scenario. In the following figures, we will consider the simple case that all TCP flows are from the same type of application.

Figure 2.9 shows the composite signaling rate η_{Total} at the SDN controller in the evaluation scenario depending on the time-out T_0. It can be seen that a very low time-out value T_0 will cause a significant amount of signaling traffic at the controller, which will put it at risk of overload. Especially, applications

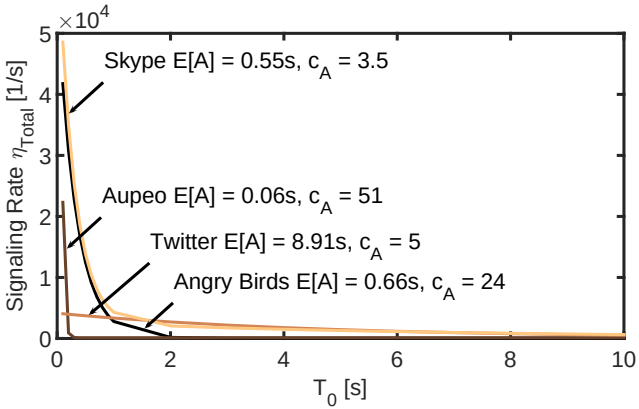


Figure 2.9: Composite signaling rate at SDN controller depending on flow time-out T_0 for the four applications in the evaluation scenario.

like Aupeo, Angry Birds, and Skype will often time out and start a new subflow, which results in frequent requests at the SDN controller. However, we see that, for the bursty applications Aupeo and Angry Birds, a large enough $T_0 \geq 2$ s will make sure that the SDN controller traffic becomes very low. The signaling rate caused by applications like Twitter and Skype will decrease more slowly when T_0 increases. Nevertheless, a higher time-out value T_0 generally decreases the traffic at the SDN controller. Thus, the default value $T_0 = 10$ s is a good choice to relieve the SDN controller.

Figure 2.10 investigates the flow table occupancy in the given scenario for different T_0. In general, the flow table occupancy increases with the time-out T_0, as TCP flows are discarded later from the table. This results in the monotonic increase of all curves in the figure. Low table occupancies can only be achieved by very low T_0. We see that the less bursty applications like Skype and Twitter continue to have a high gradient when T_0 increases. Thus, setting a high T_0 will cause a high table occupancy for these applications. In contrast, the occupancy

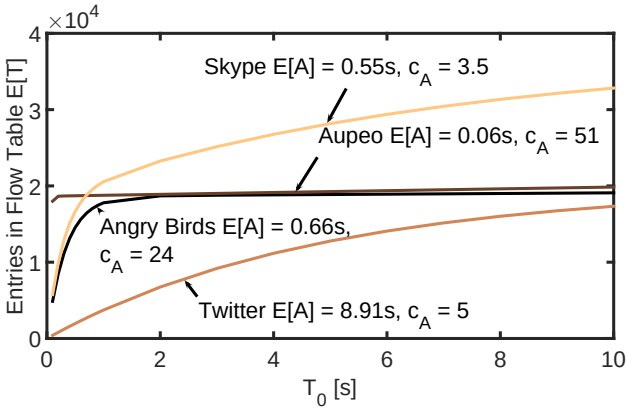


Figure 2.10: Flow table occupancy at SDN switch depending on flow time-out T_0 for the four applications in the evaluation scenario.

of bursty applications Angry Birds and Aupeo only increases very flatly for high enough $T_0 \geq 2$ s.

All in all, we see that a very low T_0 is required to reach a low occupancy of the flow table. However, this will cause a high signaling rate at the SDN controller. In turn, a high T_0 will cause a low signaling rate but a higher table occupancy. Still, some room for trade-off is left by setting T_0 to a value around 2-3 s. For the investigated applications Twitter and Skype, this will result in a reduction of the flow table occupancy compared to the default time-out of $T_0 = 10$ s, but will only cause a negligible increase of signaling at the SDN controller. Bursty applications like Aupeo and Angry Birds are not negatively affected considerably by such choice of T_0.

2.3.4 Application Mix Model

Usually, not only one application is active within a network but a whole mix of applications. Therefore, a mix of the four introduced applications is evaluated. In

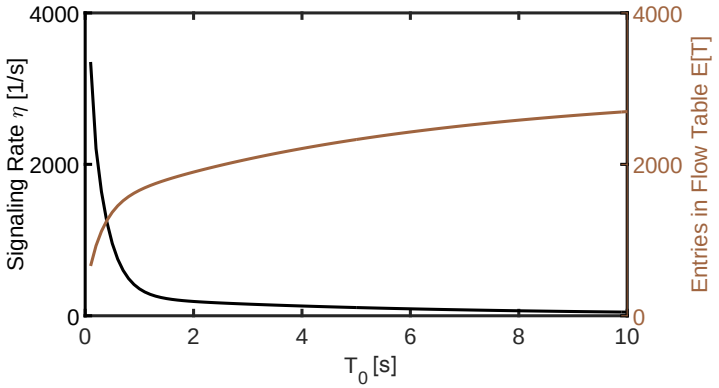


Figure 2.11: Signaling rate & flow table occupancy for varying T_0 values in the case of an application mix.

order to create a fix application mix, the number of downloads in the beginning of December 2016 from Google's Play Store [42] for each application have been taken and put into relation. The resulting share-ratio amongst the applications is depicted in Table 2.2. Twitter has a share-ratio of 32 %, Skype one of 35 %, Aupeo, though running out of service on December 16th 2016, a ratio of 15 %, and, last, Angry Birds a ratio of 18 %. In Figure 2.11, the signaling rate η_ω and the switch table occupancy ρ_ω of this previously mentioned application mix for a range

Table 2.2: Share-ratio of the four applications comparing their Google Play Store downloads

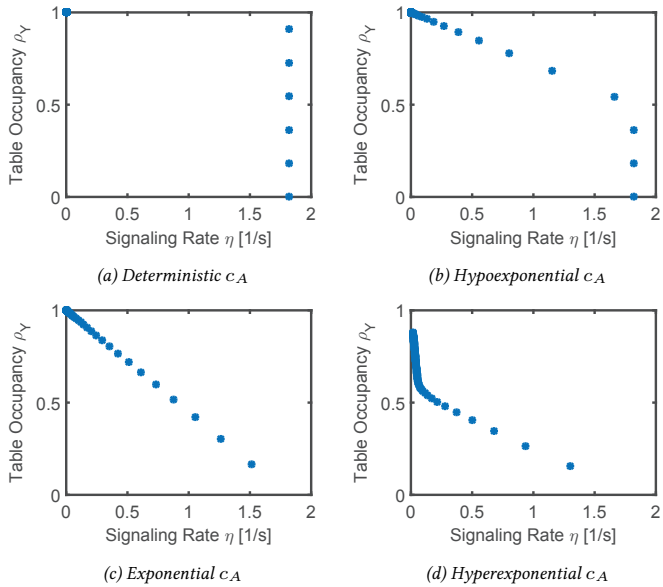
Application	Share-Ratio
Twitter	32%
Skype	35%
Aupeo	15%
Angry Birds	18%

of T_0 values are depicted. The x-axis shows T_0 ranging from 0 to 10 seconds, the y-axis on the left side the signaling rate from 0 to 4000 signals per second, and the y-axis on the right side the table occupancy from 0 to 4000 entries. As anticipated by the results for a single application, increasing the T_0 value has an enormous effect on the signaling rate towards the controller. With T_0 equal to 0, each flow times out for each arriving packet, and, thus, the signaling rate is here at its highest value of around 3300 signals per second. Higher T_0 values lead to a lesser ratio of packet inter-arrival times from this application stream, and, thus, a lower frequency of flow deletions from the flow table.

As the flow remains for a longer time inside the flow table, less signaling to the controller is required. Therefore, the signaling rate continuously decreases in an exponential manner to a value of 50 signals per second for T_0 = 10 s. For the switch table occupancy a coherent behavior is shown. With a small T_0 value below 1 second almost each new arriving packet of this application mix is arriving after the time-out has passed. Thus, the number of entries is low, e.g. only 650 for T_0 = 0.1 s. With a further increasing T_0 the flows remain for a longer time in the flow table. Therefore, the increase of the flow table occupancy flattens. Overall the table occupancy behavior has the resemblance of a logarithmic function. In the range of 0 to 1 seconds, the table occupancy heavily increases in this region from 600 to 1600. Beyond that the table occupancy only increases from 1600 to 2600 in the range of 1 to 10 seconds. In conclusion, for this application mix scenario, T_0 = 3 s is again the most suitable choice. With values below 3s, the signaling rate is too high, with values beyond 3s the table occupancy is only marginally increasing.

2.3.5 Lessons Learned for Dimensioning T0

Figure 2.12 summarizes the above findings by opposing signaling rate η and table occupancy ρ_Y for fixed $E[A] = 0.55$ s. Four qualitatively different cases are distinguished.



(a) Deterministic c_A

(b) Hypoexponential c_A

(c) Exponential c_A

(d) Hyperexponential c_A

Figure 2.12: Joint evaluation of signaling rate η and table occupancy ρ_Y for fixed $E[A] = 0.55\,s$ and $c_A = [0, 0.5, 1, 3.5]\,m$.

- $c_A = 0$:
  Figure 2.12a shows the deterministic case. If the time-out T_0 is larger than $E[A]$ the signaling rate is 0 and the table occupancy is 1. The smaller the time-out is set, the less the table occupancy. However, the signaling rate will not be affected by the choice of T_0.

- $0 < c_A < 1$:
  For the hypoexponential example $c_A = 0.5$, Figure 2.12b illustrates that the choice of T_0 influences both η and ρ_Y. Starting from $\eta = 0, \rho = 1$, a decreasing T_0 will slowly decrease the table occupancy, but faster in-

crease the signaling rate. When the signaling rate has reached its maximum, lowering T_0 will still decrease the table occupancy.

- $c_A \geq 1$:
  While Figure 2.12c shows a balanced behavior, it constitutes the transition to the hyperexponential case for which the example $c_A = 3.5$ is presented in Figure 2.12d. Here, we see that decreasing T_0 can significantly reduce the table occupancy ρ_Y, while only negligibly affecting the signaling rate η. Only if T_0 becomes too small, the signaling rate will increase. As most applications produce traffic with hyperexponential c_A this behavior can be exploited.

Recapitulating, a smaller T_0 decreases the switch table occupancy whilst increasing the signaling load offered to the controller. The biggest optimization potential can be observed especially for flows with small, hyperexponential c_A. If the time-out T_0 was optimized for these applications, the highest gain of table occupancy could be reached. Revisiting the results described above, a trade-off between these two metrics can be found for T_0 = 2-3 s: beyond that point, the switch table occupancy only marginally increases (on average), whilst the signaling load is at an acceptable value and decreases in small terms.

In general, an application-specific T_0 value would be preferable, though a smaller T_0 value already offers a good starting point. For setting an application-specific value, additional traffic characterization mechanisms have to be deployed within the network. A possible integration could start with a small T_0 value whilst the application is still unknown and not enough packets were yet received. After successful characterization, the T_0 value can be changed dynamically. Another influencing factor on the current T_0 values should be the overall table occupancy of a switch. If a flow table is full, no new flow can be installed. As most current SDN controllers do not have a failure handling for this case, they simply retry to install that flow for each incoming packet until the action completes or no more packets of a flow arrive at the switch or at the controller.

A beneficial factor for the controller load could also be to enable caching at the switches. As soon as a flow times out due to its T_0 value, it could be marked as `to delete`, but yet still left active within the flow table. Now, if a packet matching that flow arrives again at the switch, the packet can immediately be forwarded and the entry and its timer can be reset to the initial setting. This would reduce the controller load, however, the controller should be notified, such that it maintains a coherent view of the network. If the flow table is full, `to delete` entries can be deleted from the flow table and replaced by new rules. Caching algorithms for flow tables in an SDN environment have been investigated by various authors, as presented in Section 2.1. Nevertheless, an analytic approach has not been taken yet.

2.4 Validation of the Model

In order to validate the presented model, a discrete event simulation was implemented in MATLAB [43] and a testbed allowing measurements was created. This section features the implementation of both the simulation and the testbed. Afterwards, three different scenarios are evaluated and compared to the corresponding model values. The three scenarios include the case of a single flow being active in the system, multiple flows, and, finally, the case of an application mix, as introduced in Section 2.2.4. Due to the long duration of the measurements, only validation for the case of the single flow model was conducted. However, further validations are easy to catch up according to the guidelines of the previous sections.

2.4.1 Simulation Procedure

Based on the results from [11], packet arrivals were simulated for each application. This means, random packet arrival times were obtained from the substitute arrival processes presented in Section 2.3.1 for the given mean inter-arrival time $E[A]$ and coefficient of variation c_A. The packet arrivals of each flow allow to

exactly determine the signaling and the flow table occupancy at the SDN controller depending on the time-out T_0.

The simulation implements Poisson arrivals of flows with arrival rate $\lambda = 158.73\frac{1}{s}$ as measured in [41]. However, using exponentially distributed flow durations with the mean presented in [41] ($E[B] = 234.95\,\text{s}$) proved to be much to short to reach the desired $E[A]$ and c_A given in [11] within the simulation. Therefore, for the validation of the model, the mean of the exponentially distributed flow duration is set to $E[B] = 3600\,\text{s}$ (one hour), and a minimum flow duration of $900\,\text{s}$ (15 min) is introduced. These modifications allowed the average $E[A]$ and c_A of the flows, which were created by the substitute packet arrival processes described in [11].

In the validation scenario, the simulation duration is set to $50000\,\text{s}$ (13.89 hours). The presented results show the mean overall signaling rate and the flow table occupancy as observed by the SDN controller during five simulation runs. In addition to the mean results, also the 95% confidence intervals are shown. To account for the transient phase of the simulation, only the last $25000\,\text{s}$ of a simulation run have been evaluated. As all simulation runs converged to the stationary phase typically after around $10000\,\text{s}$ simulated time, the presented results can be assumed to provide a consistent view of the system in the long run. This long convergence time of the initially empty system occurred especially for simulation runs with a high coefficient of variation c_A. In these cases, the high burstiness of the packet arrivals and the resulting high variability of the flows' on- and off-phases slow down the convergence.

2.4.2 Measurement Procedure

Additional to the validation by simulation of the previous section, also a validation by measurements was performed. For this purpose a testbed was built and software packet generators, allowing to configure the packet inter arrival time and their coefficient of variation c_A, have been implemented. As vectors of packet inter-arrival times with some values of c_A are difficult to gen-

erate, as discussed in Section 2.4, we focused only on specific values, namely $c_A = \{0, 0.5, 1, 2\}$ The measurement duration was set to 20 min per parameter combination. The results show the mean results over 10 repetitions. Additionally, also 95% confidence intervals are shown.

Testbed

This section features the testbed, as depicted in Figure 2.13. The network devices required for the first measurements are emulated in Mininet [44]. This network consists of one Open vSwitch [45] which is connected to two hosts. Open vSwitch is an Open Source virtual software switch that supports the OpenFlow protocol. Host 1 is running a configurable TCP packet generator, host 2 the corresponding packet sink. The packet generator is a self-written small JAVA application allowing the generation of packet streams with a configurable mean inter-arrival time E[A] and coefficient of variation c_A. These inter-arrival times are calculated based on the two-moment formula of [33]. For the purposes of these measurements, we have chosen the RYU SDN controller as it allows for an easy adaption of the installed flow time-out period. The RYU SDN controller is connected to the virtual switch and only running a reactive forwarding application with a configurable time-out T_0. In order to measure all required parameters, two sensors have been installed: the first one is tcpdump [46], capturing the communication between switch and controller. The second one is a small program recording the flow table utilization of the switch each 0.5 s.

Measurement Process

In order to generate reliable, repeatable measurement results, the following measurement process was defined.

1. At first, a reset of the whole measurement environment is conducted, meaning that all running applications are shut down and no other component is accessing the Mininet topology.

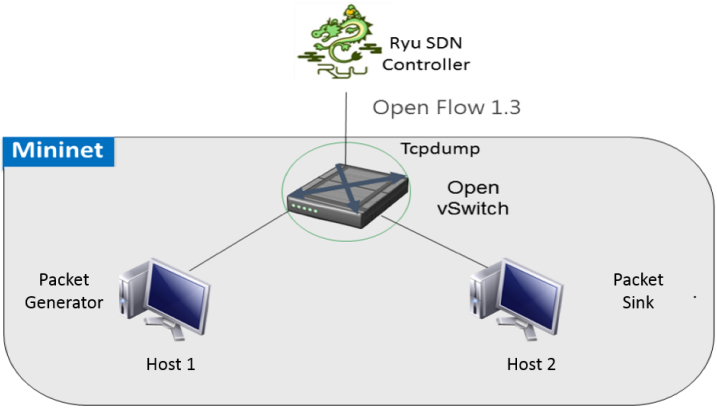


Figure 2.13: Testbed Architecture.

2. The second step is to configure the RYU SDN controller, start it up, and connect it to the software switch using the OpenFlow 1.3 protocol [47]. Depending on the current scenario, the flow time-out of RYU will be configured and the appropriate number of hosts will be started in the Mininet topology.

3. Third, the program recording the flow table utilization is fired up. Afterwards, the TCP traffic sink is started on the appropriate host.

4. Forth, the recording of the traffic between switch and controller via tcpdump begins.

5. Fifth, the TCP packet generator is started with the parameter configuration of the current scenario.

6. After the measurement duration, which is defined by the scenario, the TCP generator is stopped, Mininet is shut down, and the connection be-

tween Ryu and the switch is cut. Tcpdump exits and the measurement data is collected.

Now, the testbed is ready for another measurement run and the process starts over with step one.

2.4.3 Single Flow Model

For the single flow model, we reproduce the results presented in Figures 2.5 to 2.8. This means, we simulate one flow each with different mean packet inter-arrival times ranging from 0 to 20s for a time-out T_0 = 10 s. Additionally, we simulate one flow with $E[A] = 0.55$ (cf. Skype) and varying T_0 from 0 to 10 s. All presented coefficients of variation were used (0, 0.5, 1, 3.5, 5, 24, 51). Note that for this validation, the simulated single flows span the whole simulation duration, and thus, the application characteristics presented in [11] could be exactly replicated. As the curves obtained from the simulation are perfectly aligned with the respective curves in Figures 2.5 to 2.8, we conclude that the single flow model is accurate and refrain from presenting the plots.

The curves obtained from the measurements are shown in Figures 2.14 to 2.17. For the measurements the same scenarios as described for the simulation were validated, meaning that we generate one flow each with a packet mean inter-arrival time from 0 to 20 s with a step size of 0.5s and a time-out T_0 = 10 s.

For the scenario with varying T_0, we had to limit the range from 2 to 8 s. For T_0 values below 2s the communication between controller and switch did not show the expected behavior. Open vSwitch interprets a time-out value of T_0 = 0 s as T_0 = inf. For T_0 = 1 s, our investigations showed that the flow-mods sent from the controller to the switch had the right time-out value set and these entries were successfully added to the switches flow table. But, somehow, the Open vSwitch does not follow the set time-out value and internally uses another one. Therefore, rules were deleted from the switch flow table although packets have arrived within the time-out period. An educated guess is that the internal time-out clock of Open vSwitch does cope with this value of T_0. Yet, various

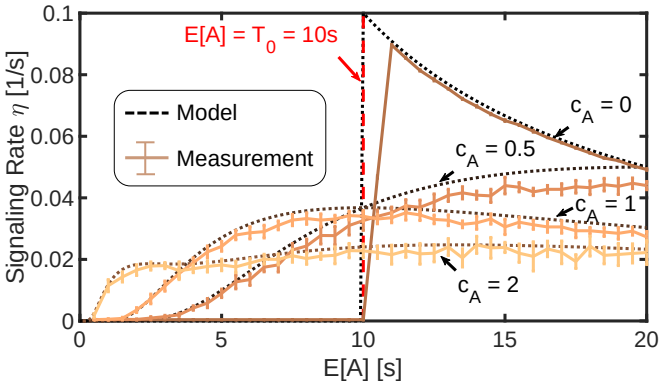


Figure 2.14: Measurement results for the impact of mean inter-arrival time on the controller signaling rate.

investigations did not show a deterministic behavior, therefore, we excluded this range from the results.

Figures 2.14 and 2.15 show the signaling rate and the flow table occupancy of the measurements for the varying mean packet inter-arrival time for the coefficient of variation values $c_A = \{0, 0.5, 1, 2\}$. The x-axis shows the inter-arrival time between 0 and 20 seconds, the y-axis the signaling rate between controller and switch or the flow table occupancy, respectively. The results for the different coefficients of variation with 95% confidence intervals are shown in different colors. The black line represents the results from the model, the dashed vertical red line at E[A] = 10 s depicts the configured idle time-out of 10 seconds. The results show that the measurement results verify the model, as they overlap with the model values. Moreover, the model results are often within the confidence intervals of the appropriate measurement results. This, combined with small size of the confidence intervals, validates the model.

The measurement results for the signaling rate and the flow table occupancy with a varying time-out value are depicted in Figures 2.16 and 2.17. The x-axis

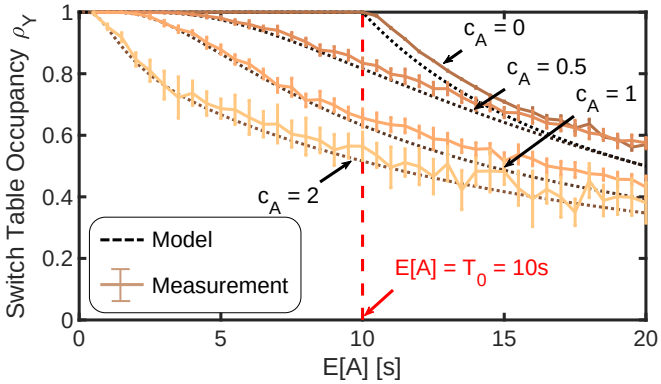


Figure 2.15: Measurement results for the impact of mean inter-arrival time on the switch table occupancy.

shows the time-out values T_0 from 2 to 8 seconds, the y-axis the signaling rate of controller and switch or the flow table occupancy, respectively. The measured coefficients of variation are shown in different colors. The intervals depicted are 95% confidence intervals. For each measurement result the corresponding model values are shown as a solid black line. Again, the measurement results match with the overall behavior of the model values. With increasing the T_0 value on the Open vSwitch, the deviation between model and measurements decreases and becomes almost invisible.

In general, the deviation between model values and measurement results can be traced back to the following reasons. First, the model values are generated for packet inter-arrival times between 0 and 20 s with a step size of 0.1 seconds, whereas the measurement step size has been 0.5 seconds. This is caused by the length of the measurements. In order to generate the results for one c_A it takes 20 minutes $*$ 40 values $*$ 10 repetitions $=$ 8000 minutes $\approx$ 5.55 days. An increased resolution of measurement results would have increases the measurement time by a factor of 5 and would not have been feasible.

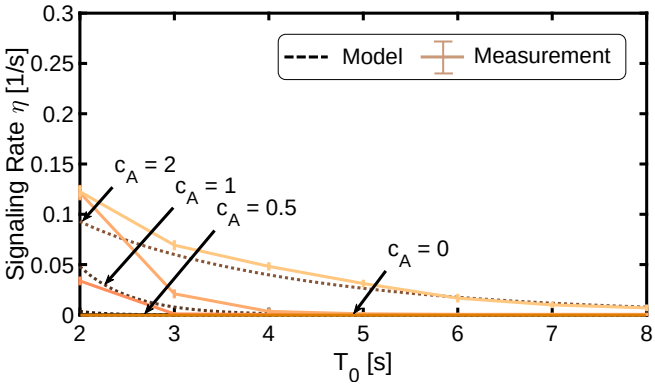


Figure 2.16: Measurement results for the impact of the time-out on the signaling rate.

Additionally, especially for higher c_A values, the measurement duration of 20 minutes is at a minimum. As discussed earlier, with high c_A values (i.e. $c_A > 5$) it may happen that the packet generator only generates a small number of packet inter-arrival times. Calculating the coefficient of variation of these three values often leads to a c_A value smaller than anticipated. This is not a fault of the implementation, as increasing the sample size would lead to the anticipated c_A value. To achieve better results, the measurement duration could be increased, respectively further increasing the total measurement duration. Moreover, increasing the number of repetitions per measurement would benefit the measurement accuracy. Another factor that has not been mentioned yet is the network delay being present to the usage of real network connections. This further contributes to the deviation between model and measurement.

2.4.4 Multiple Flow Model

Figures 2.18 and 2.19 show the results of the validation of the multiple flow model, and correspond to Figures 2.9 and 2.10. In Figure 2.18 the results for both

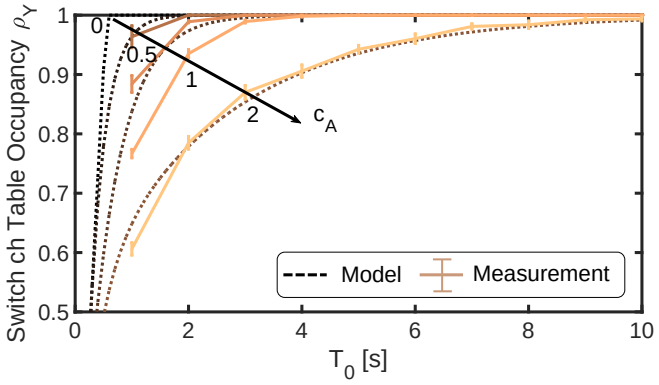


Figure 2.17: Measurement results for the impact of the time-out on the flow table occupancy.

simulation (solid) and model (dotted) for the signaling rate are shown. Here, the simulation is matching the model perfectly, as both results for all applications align. Additionally, the confidence intervals are between very small for low T_0 values and not visible for higher T_0 values, confirming the validity of our results. In Figure 2.19 results for the table occupancy in the case of multiple flows for the introduced applications are presented. For the applications with a small c_A (Skype and Twitter) both simulation and model align perfectly. For Angry Birds and Aupeo with a very high c_A of 24 and 51 the limitations of the simulation are visible. The alignment between them is lower and the confidence intervals are bigger. This deviation in the results can be explained by the challenges presented at the beginning of this Section.

2.4.5 Application Mix Model

To validate the model for a traffic mix of various applications, the application mix introduced in Section 2.3.3 is simulated. In Figures 2.20 and 2.21, the dotted

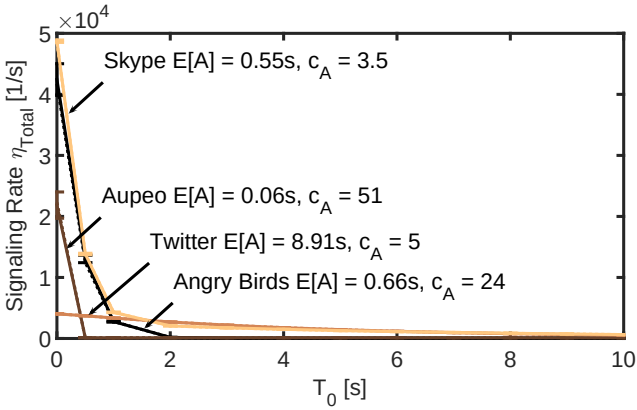


Figure 2.18: Validation of the multiple flow model for signaling rate.

line shows the model results, and the solid line the mean results of the simulation and 95 % confidence intervals after five runs. The small confidence intervals indicate a consistent behavior for all simulation runs. The model provides accurate results for which the deviation never surpasses 10 % throughout the whole parameter range of T_0. The remaining deviation can be attributed to the randomness within the single flows and the inadequacy to consistently reproduce the desired application characteristics for all flows, which propagate to the aggregated results. Still, the results confirm that the application mix model is sufficiently accurate to describe the signaling rate and table occupancy of an SDN controller.

In the following, it will be investigated how a variation of the application mix impacts the SDN controller. Instead of assuming one application mix with a constant flow ratio between the applications, we now evaluate application mixes of two applications for varying ratios. To better compare the impact of a specific application, we fix Twitter in the application mix, and mix it with the remaining three applications. For these scenarios we use the time-out $T_0 = 3$ s, which we

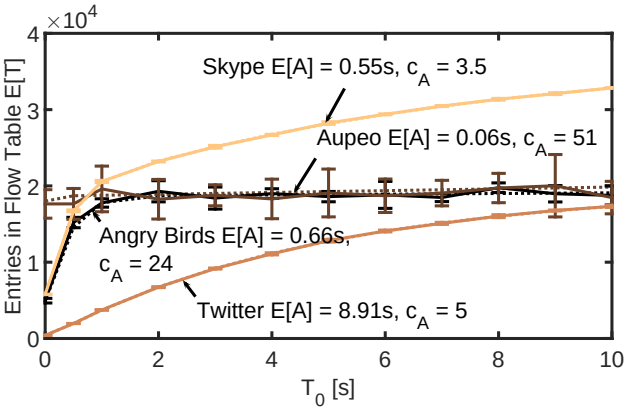


Figure 2.19: Validation of multiple flow model for table occupancy.

found earlier to be a good trade-off between signaling rate and table occupancy for application mixes in Section 2.3.4.

Figure 2.22 shows the resulting signaling rate at the SDN controller for the considered mixes of two applications. Thereby, the application mix is depicted on the x-axis, starting from 100 % Twitter flows and 0 % other app traffic, gradually decreasing Twitter until 0 % Twitter and 100 % other app traffic flows. The y-axis presents the signaling rate from 0 to 400 requests per second. Here, again, the dotted line displays the model results and the solid line the mean results and 95 % confidence intervals of our simulation after 5 runs. The Twitter values always start at 320 signals per second for the modeling results, and 350 signals per second for the simulation, respectively. The discussed inadequacy of the simulation, which only reproduces the desired application characteristics on average, reappears here. Increasing the ratio of the other applications Skype, Angry Birds, and Aupeo softens these effects, because these application characteristics can be more easily reached in the validation scenario. For example, increasing the ratio of Skype will decrease the signaling rate at the SDN controller. Both, simulation

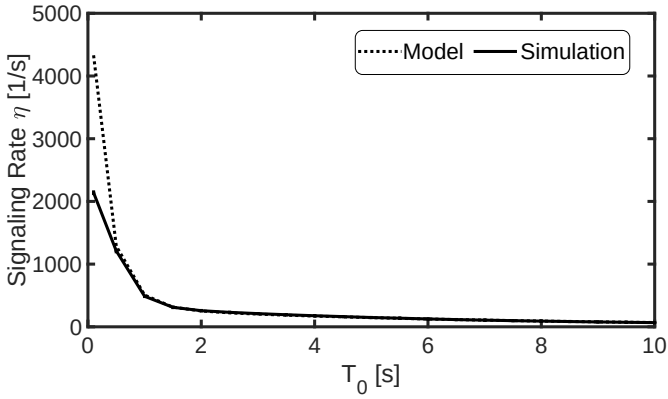


Figure 2.20: The impact of the T_0 value on the signaling rate η for the case of a mixed application stream.

and adapted model, produce a signaling rate of 280 per second, decreasing the deviation of the simulation down to almost 0 %. For Aupeo and Angry Birds, the same linear course of the signaling rate can be seen. The high coefficient of variation of these applications decrease the signaling rate close to 15. Thus, for the given time-out of T_0 = 3 s, a higher ratio of such traffic will reduce the load of the SDN controller.

The corresponding results for the total flow table occupancy is presented in Figure 2.23. The x-axis again depicts the application mix, the y-axis shows the overall flow table occupancy in the SDN switch. For Twitter-only traffic the model produces a result of 1300 entries in the flow table, which is slightly below the actual 1500 entries obtained by the simulation. Increasing the ratio of the other applications again decreases the deviation between model and simulation. Skype-only traffic results in around 3700 entries on average, while Aupeo and Angry Birds-only traffic will store almost 3000 entries in the flow table. As

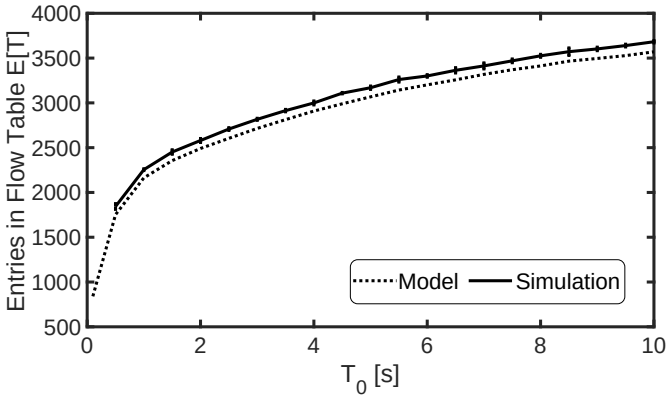


Figure 2.21: The impact of the T_0 value on the flow table occupancy for the case of a mixed application stream.

Twitter traffic uses the least flow table entries, again the trade-off between the signaling rate and the table occupancy is clearly visible.

To sum up, the simulative validation and the validation by measurements proved the accuracy and applicability of the single flow, the multiple flow, and the application mix models. Thereby, the small confidence intervals indicate a consistent behavior of the presented results, which well aligns with the models. Thus, as they allow to produce accurate results for a given evaluation scenario much faster, the analytical models can be used to replace simulative and measurement approaches. Moreover, the impact of different application characteristics and the resulting trade-off between signaling rate and table occupancy could be confirmed. As the trade-off can be tuned by setting the time-out T_0 properly, our analytical models are a useful tool for operators of SDN networks to find the optimal settings with respect to the specific application mixes in their networks.

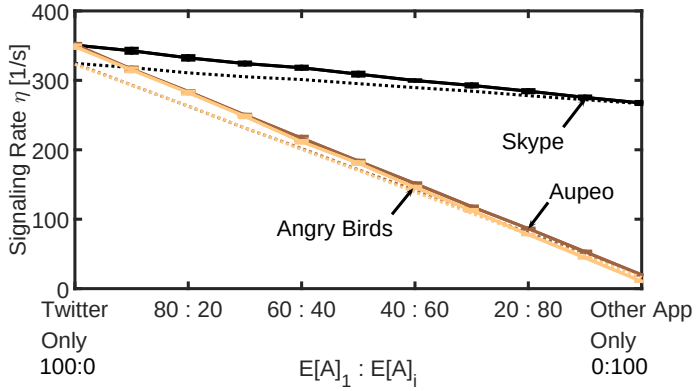


Figure 2.22: Signaling rate η of multiple application mixes with varying percentage.

2.5 Lessons Learned

In this chapter, an analytical model for SDN controller traffic and switch table occupancy is presented. The model focuses on the reactive operation mode of a controller. Incoming and unknown traffic at a switch therefore generates a request towards the controller. Based on this traffic, rules are created in the flow table of the switch, which specify the forwarding behavior. According to this flow table entry, further packets of this flow are processed by the switch only, and do not require any further controller interaction, i.e. generate no signaling traffic. To avoid table overflows, unused entries are removed after a predefined time-out period T_0. A time-out value is added to each flow entry. As soon as no packet has matched a flow for a duration of T_0, the flow is automatically removed from the table. Any further packet of this flow triggers a new request towards the controller.

As both the switch table occupancy and the controller signaling can render a limiting factor to the forwarding speed of a network, a trade-off between signal-

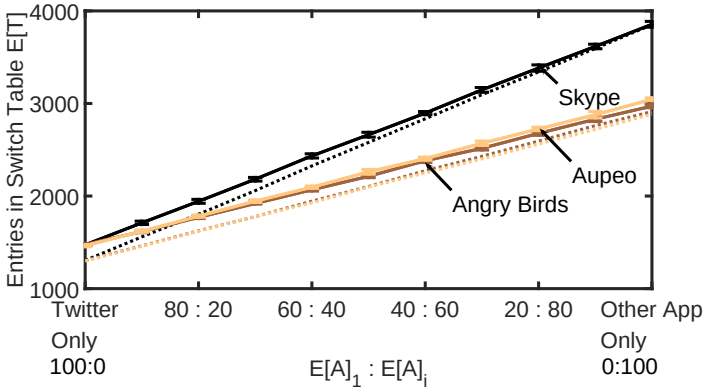


Figure 2.23: Flow table occupancy ρ of multiple application mixes with varying percentage.

ing rate and switch table occupancy has to be found. The results in this chapter deliver three main conclusions.

1. First, with our presented model it is possible to calculate the effects of the discussed parameters with respect to the performance of the SDN-based network. We start by modeling a single flow to understand its impact on the flow table occupancy and the resulting controller traffic. Based on these results, we adapt an $M/M/\infty$ queuing system and extend our model to understand the implications when multiple users, i.e. multiple flows, are in the system.

2. Second, the presented discrete-event simulations and the measurements validate the model in all scenarios, proving the accuracy and the applicability of our model. The impact of multiple application characteristics is additionally discussed.

3. Third, it is shown that application specific parameters, such as the inter-arrival time of packets $E[A]$ and its coefficient of variation c_A, have a non-negligible impact on both the signaling rate and the table occupancy.

However, the time-out value T_0 introduces an opportunity for trade-off. As long as $E[A] < T_0$ both factors are quite fine, independent of an applications coefficient of variation c_A. For applications with a high c_A, small T_0 are preferable, as the probability for time-out is comparatively high, which will relieve the flow table. High c_A values also introduce another effect to this analysis: At some point the inter-arrival time of packets is that high, that the flow table time-out independent of the set T_0 value, which would only increase the in-effective time of a flow-entry, as the only activity is the countdown of the timer. We observed that very bursty applications with high c_A are relatively unaffected by the choice of T_0 because of the likely long times between bursts. Thus, the highest optimization potential could be seen for flows with small c_A. Especially for these applications, T_0 should be set to improve the flow table occupancy, which can be reached by small T_0. This causes flows to time out and free the flow table space sooner, and thus, enables the processing of more applications within in the network at the same time.

Nevertheless, a too small T_0 has to be avoided as this will result in quickly increasing traffic at the SDN controller, which puts it at risk of overload. Consequently, our results show that the default value of T_0 = 10 s is too large. This is comparable to the findings of Zarek *et al.* in [24], which proposes a static time-out value of 5 s. Based on our observations, the best trade-off could be reached by decreasing T_0 down to 2-3 s. With these values, e.g., the flow table occupancy of Skype flows could be reduced by around 25%, while the controller traffic would only slightly increase.

The best results in terms of signaling rate and flow table occupancy could be achieved for an application specific T_0 value, e.g., as presented by Vishnoi *et al.* [48]. This, however, renders the requirement for identifying applications or collecting flow statistics based on the packet stream, which can pose quite new challenges. Additionally, the model itself can also be applied to a composed

network. However, the impact of switch-controller interaction on the packet inter-arrival times of a flow are not covered by the current model. How exactly this affects the accuracy in a composed network has to be covered by future research.

One might argue that the new generation of switches has much bigger flow table sizes, and, therefore, the importance of this work could decrease in the future. But, as the size of flow tables increase, more fine-granular flow rules are possible, and, thus, the flow table size could become an issue again.

3 Increasing the RAS Level in SDN Deployments

Software-defined Networking (SDN) is an increasingly important technology that breaks up the ossified structure in networking: it decouples the control from the data plane of network devices [8]. With this shift in networking, it is possible to centralize the control plane of many devices into one single software, the so called controller. This controller opens up many new possibilities, such as a flexible and programmable management station to steer, control, and monitor the network.

Providing high availability is and remains one of the biggest challenges of network management. For the ordinary user, this struggle is only visible through some magical promise of services with "up to x nines of availability" or is only discussed if a certain service, e.g. Google, is unreachable or users of a certain ISP are unable to access the Internet [49]. In the background, providers have to monitor their services for availability 24/7 in order to fulfill contracts and SLAs and not to lose money over it.

The process of monitoring the network, especially for larger deployments, is very complex and contains many pitfalls, for instance, the balance between granularity of information and their performance impact on the network. With the help of SDN this challenge can become easier, as it offers new methods, mechanisms and opportunities. One of the current most important open source controllers is the ONOS SDN controller [50]. According to its developers, it is a production ready controller that offers high availability due to its logically centralized and physically distributed architecture. But, as our investigations

show [6], it is unable to cope with hazardous network conditions such as sporadic or recurring packet loss on network links. It either does not detect hazardous conditions or only detects it after long time periods, failing all common network availability targets. From the technical point of view network monitoring needs configuration, planning, and constant adjustment in order to guarantee high availability. To enable high availability in the case of failures or errors of the network, at first a fast failure detection is required.

The contribution of this chapter is threefold: At first we present a stochastic analysis of the theoretical detection performance of the ONOS SDN controller towards link impairing effects. Second, we support our evaluation by measuring the detection performance of the ONOS controller for the case of packet loss in the data plane. Based on these results it becomes evident that ONOS is unable to reliably detect packet loss scenarios. Third, we present and design a detection extension to the ONOS SDN controller. It generates probing messages for each topology link that are induced into the data plane, and forwarded back to the controller. There, the message is processed and multiple statistics for each link can be derived, e.g. link delay or packet loss. If these values surpass configurable thresholds, a notification of an unusable link to the ONOS SDN controller is generated, triggering a rerouting of affected flows. Additionally, based on the current measured network conditions, it optimizes its probing process by, e.g. changing the time between two successive probing packets. Therefore, it always provides the desired detection performance while automatically balancing the load on the controller. Forth, we present an evaluation of this probing extension in multiple scenarios and compare it to the performance of the vanilla ONOS SDN controller. The findings of this chapter are also presented in [6, 7].

Section 3.1 presents related work and the background of the probing networks with SDN. Section 3.2 introduces a theoretical analysis of the detection mechanisms of the ONOS controller and verifies it with measurements. Afterwards, in Section 3.3, the active probing extension is presented and evaluated. Section 3.4 brings up the necessity to adapt to the current network conditions and introduces a suitable extension of the active probing application to cope with these

network conditions. Finally, Section 3.5 summarizes the content of this chapter and gives an outlook to further work on this topic.

3.1 Failure Detection in SDN

The remainder of this section is structured as follows: At first, a definition of existing approaches towards failure detection and failure tolerance is discussed. Secondly, background on detection mechanisms in the OpenFlow SDN protocol is presented. Finally, existing failure detection and recovery approaches are listed and discussed. The findings of this section form the basis for the further investigations.

3.1.1 Definition of Fast Failure Detection

Fast failure detection is a cornerstone for high available systems and ensures that a reaction to a failure can occur in a fast manner. In order to evaluate a *fast failure detection*, we first define common properties a fast failure detection mechanism should provide. Second, at first, it is a hard requirement to detect failures within defined time constraints. The failures the solution should be able to detect are *node, link, network or controller failure.* In our case "failure" includes anything that imposes a deviation from normal operation of the network, e.g. device outage, lossy link, or a misbehaving controller. Additionally, failure detection is a fundamental building block for ensuring fault tolerance in large-scale distributed systems. Its main objective is to reduce the time it takes to detect a failure. The challenge is to meet carrier-grade requirements with a detection time of less than 50ms without impacting the data traffic [51].

3.1.2 Background: Detection Mechanisms in the OpenFlow SDN Protocol

Current state-of-the-art SDN controllers mostly build up on the limited detection capabilities of the southbound SDN protocol OpenFlow for their fast failure

mechanisms [38, 52]. A fundamental part to detect link failures is the LINE protocol. This protocol detects a direct physical link between two adjacent components, and, therefore, is able to detect a physical change in the connection. In average, the detection time ranges from 50 to 150 ms, and is, therefore, not suitable for the provider requirements with less than 50 ms of detection time. In SDN, after the line protocol has detected a failure, a notification is sent to the controller. This message contains the affected switch, the affected port of that switch and the new port status (link down, blocked or live). Consequently, in an SDN-only network two messages are sent to the controller, one from each side of the link. Upon receipt, the controller reacts according to its internal specifications. As the line protocol is already in use for more than ten years, it also has some limitations. First, it is only able to detect a link failure at one local link. Second, it cannot detect packet loss or temporary partial failures within the network. Finally, it cannot detect network failures even if each of the used links is up and running according to the line protocol, e.g. if the routing between two end hosts in a network is not possible.

Additionally, OpenFlow offers capabilities for flow monitoring. Each OpenFlow-enabled switch offers counters, which store traffic information in different granularity, e.g. per table, per flow and per port (for example: byte/packet count per flow). The controller queries these statistics in regular intervals. Although offering basic statistics, these options also have several limitations. Frequent polling has to be used in order to monitor the dynamics of a flow. Moreover, in general, polling of statistics induces load on components, for example, a higher CPU load on the controller or a higher signaling traffic due to statistic packets. Therefore, these mechanisms are insufficient for carrier-grade SDN deployments.

3.1.3 Existing Failure Detection and Recovery Approaches

This section presents related work on the topic of SDN fast failure.

Reitblatt *et al.* [53] present a declarative language for writing fault-tolerant network programs. By using the fast-failover mechanisms provided by the OpenFlow protocol, it allows network administrator to specify paths through the network as well as the required level of reliability of these paths. Trinocular by Quan *et al.* [54] presented an outage detection system that uses active probing to understand reliability of edge networks. This approach learns the status of the Internet with probes driven by Bayesian inference.

Googles well-known B4 network presents probably one of the biggest SDN architectures [49]. Google uses OpenFlow to control relatively simple switches, which leverage control at network edges and allow for multipath forwarding. Additionally, they enable a dynamic bandwidth reallocation in case of link or switch failure. In order to increase the fast recovery performance, Adrichem *et al.* [55] introduced a failover scheme with per-link bidirectional forwarding detection (BFD [56]) sessions with preconfigured primary and secondary paths computed in advance by an OpenFlow controller. BFD sends probes with a configurable interval and as soon as three consecutive probes are lost, the link status is set to down. As long as all links are up, the primary path is always chosen for flow routing. As soon as BFD detects a link failure, their recovery process is divided into two steps: at first, the switch itself initiates a fast recovery based on preconfigured backup rules. Afterwards, the controller is notified and able to calculate and configure new optimal paths. Santos *et al.* used BFD and OpenFlow features for failure detection and resiliency in an mmWave meshed testbed [57]. Their results show the impact of the BFD probing interval on the failure detection time and its impact on the packet loss of active flows during a failover. In contrast to the presented approach of this chapter, BFD based mechanisms are only able to detect link failures.

Packet loss on a data plane link or even link delays are not covered by this approach. Scalability and fault management are of interest for Kempf *et al.* [58]. For fast recovery, they postulate that monitoring messages must be sent within a millisecond interval. They implemented their monitoring function on OpenFlow switches directly.

3.1.4 Probing Mechanisms

In general, two types of probing exist: active and passive probing. Passive probing determines the state of the network by relying on in-network traffic monitors. For example, by comparing the packet count statistics of a flow from two adjacent switches, the packet throughput and the packet loss can be computed. The advantage of this technique is that no additional measurement overload is generated in the data plane. The disadvantage is that it is often not as accurate and fast as the second probing mechanism: active probing. Furthermore, in SDN, these statistics have to be polled by a central mechanism, e.g. a plugin on the controller. This leads to additional load on the network and rises the problem of polling accuracy.

An active probing mechanism, in turn, inserts special probing packets into the networks data plane, routes them through the network, with its final destination again at the generator. According to the investigated metric (e.g. packet loss, packet delay, throughput, or congestion), the accuracy of the results and the resolution in time is higher than it is for passive probing. A non-neglectable disadvantage of the active probing approach is the induced measurement overload in the data plane by the measurement packets. In critical situations, such as a link overload, each additional packet that has to be transmitted via a link worsens the effect on the data plane traffic.

3.2 Failure Detection Analysis

This section is dedicated to the analysis of the performance of active probing. On the example of ONOS we evaluate the detection times, consequently, we need to start by considering the theoretical limits of its approach of time-out based probing. Afterwards, we measure the performance of our application in our testbed for the case of packet loss in the data plane.

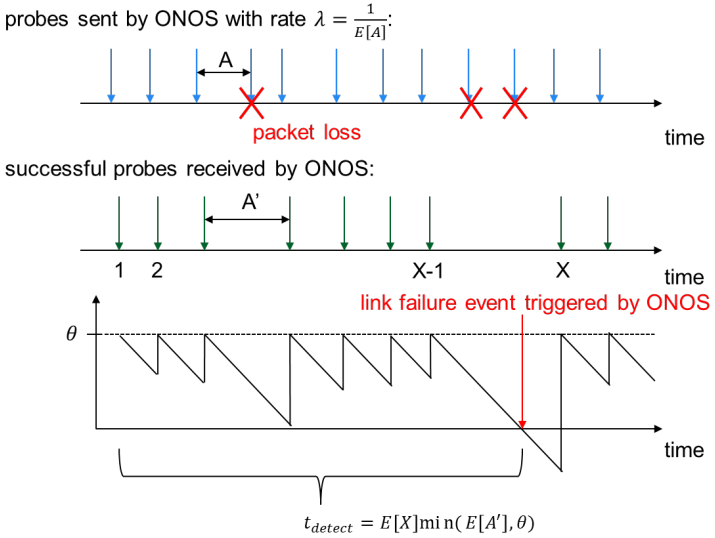


Figure 3.1: Probing process of one link.

3.2.1 Detection Time Analysis

In order to identify best probing rate, the current ONOS implementation is analyzed. To monitor a link ONOS sends probes with a fixed rate λ as shown in Figure 3.1. After each arrival of a probe a timer is reset to zero. The ONOS implementation detects a link failure and sends a link failure event if the timer exceeds a threshold θ (see Figure 3.1). To analyze the performance of the probing mechanism we use A as random variable (RV) for the inter-arrival time of probes. We consider the packet loss probability p as the probability that a probe is lost. Due to the loss of probes and the network dynamics the process of successful probes that arrive at the controller differs from the inter-arrival process of probes. Therefore, we use A' as RV for the inter-arrival time of successful

probes, with $E[A'] \geq E[A]$. The probability of a link failure event is then given by

$$p_{fail} = P(A' > \theta) = 1 - P(A' \leq \theta) = 1 - F_{A'}(\theta) \quad , \tag{3.1}$$

where $F'_A(t)$ is the cumulative distribution function of RV A'. The mean detection time of a failure is calculated by considering the number of successful probes X that arrive at the controller until the first successful probe comes late with probability p_{fail}. Considering that a failure event is triggered if $A' > \theta$, the mean detection time can then be calculated by

$$t_{detect} = E[X]min(E[A'], \theta) = \frac{1}{p_{fail}} min(E[A'], \theta) \quad . \tag{3.2}$$

Since X follows a geometric distribution with parameter p_{fail}, which has expected value $E[X] = \frac{1}{p_{fail}}$. If the probes are sent with a constant rate λ, as in the ONOS implementation, the inter-arrival time of probes A is $\frac{1}{\lambda}$ with probability 1. In this case the inter-arrival time of successful probes A' can be calculated by considering Y as RV for the number of probes that are sent to the controller until a probe successfully arrives at the controller with probability $(1-p)$. Y follows a geometric distribution with parameter $(1-p)$ and has the CDF $F_Y(k) = 1 - p^k$ and expected value $E[Y] = \frac{1}{1-p}$. Since the probes are sent with constant rate λ, we can calculate A' by $A' = Y \cdot A = Y \cdot \frac{1}{\lambda}$, with $E[A'] = E[Y] \cdot \frac{1}{\lambda} = \frac{1}{1-p} \cdot \frac{1}{\lambda}$ and $F_{A'}(t) = 1 - p^{\lfloor t \cdot \lambda \rfloor}$.

Hence, in this case the probability of a link failure is

$$p_{fail} = 1 - F_(A')(\theta) = p^{\lfloor \theta \cdot \lambda \rfloor} \tag{3.3}$$

and the mean detection time is

$$\begin{aligned} t_{detect}(p,\theta,\lambda) &= \frac{1}{p_{fail}} min(E[A'],\theta) \\ &= \frac{1}{p_{fail}} min(\frac{1}{1-p}\cdot\frac{1}{\lambda},\theta) \\ &= \frac{1}{p^{\lfloor\theta\cdot\lambda\rfloor}} min(\frac{1}{1-p}\cdot\frac{1}{\lambda},\theta) \quad . \end{aligned} \tag{3.4}$$

Based on the configuration parameters of ONOS, a calculation of multiple metrics is possible. In the following, two metrics will be evaluated: the mean detection times and the detection probability. The mean detection time describes the time interval between the configuration change of a link, e.g. from 0% packet loss to 5% packet loss, and the point ONOS is actually recognizing it. Link detection probability describes the probability ONOS is able to detect packet loss on a data plane link. Variable input parameters are the probing frequency λ, how often a probe is sent through a link per second, and the time-out θ, the number of probing packets that are required to be lost consecutively in order for ONOS to detect a link failure.

Figure 3.2 shows the probability p_{fail} of the ONOS detection module to recognize a change in the links status. The x-axis depicts the data plane packet loss probability, the y-axis the probability an event is recognized by ONOS. Three probing frequencies are shown. From left to right: 1/3 s, 1/2 s and 1/1 s. The time-out has been set to $\theta = 9\ s$. For the default probing frequency of 1/3 s, the results show the highest detection probabilities. If the probing frequency is increased, a higher number of consecutive probes have to be lost so that the timeout is exceeded. Hence, for a fixed time-out, the probability of ONOS to trigger a link failure event decreases with the probing frequency. In order to detect link failure events with high probability for high probing frequencies, the time-out has to be reduced with the probing frequency.

Figure 3.3 shows the probability of a link failure event p_{fail} dependent on the packet loss probability for a fixed probing rate 1/3 s and different time-outs 3 s,

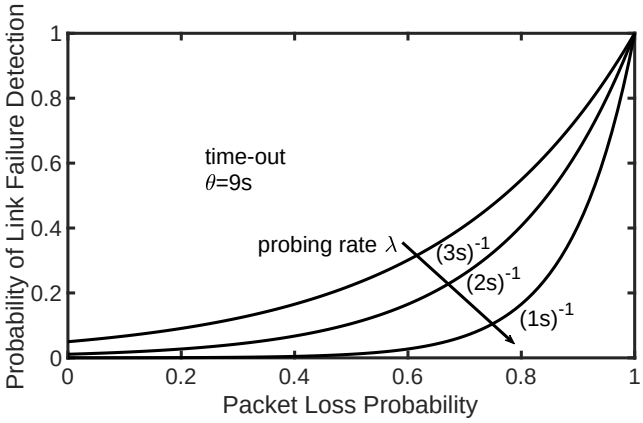


Figure 3.2: Calculus - detection probability for varying probing rate λ.

6 s and 9 s. The probability of a link failure event increases with the probability that a probe is lost. With increasing time-out the probability of a link failure event decreases, since a higher number of consecutive probing packets have to be lost so that the time-out is exceeded. If the time-out is set lower than the inter-arrival time of probes $\theta = \frac{1}{\lambda}$, a link failure event is triggered after each probe. Hence, to maximize the probability of a link failure event given a probing rate, the time-out is set to $\theta = \frac{1}{\lambda}$.

Figure 3.4 shows the mean detection time calculation for a variable probing frequency and two time-out values. On the x-axis the data plane packet loss probability is depicted, the mean detection time by ONOS in seconds is depicted on the y-axis. The solid line always depicts a probing frequency of 1/3 s, the dashed line a frequency of 1/2 s, and the dashed and dotted line a frequency of 1/1 s. Results in black have the time-out set to three times the probing frequency, for results in blue the time-out is set to one time the probing frequency. Apparently, ONOS mechanisms do not perform very well in this scenario. Only for data plane packet loss values larger than 45%, the mean detection time is less or

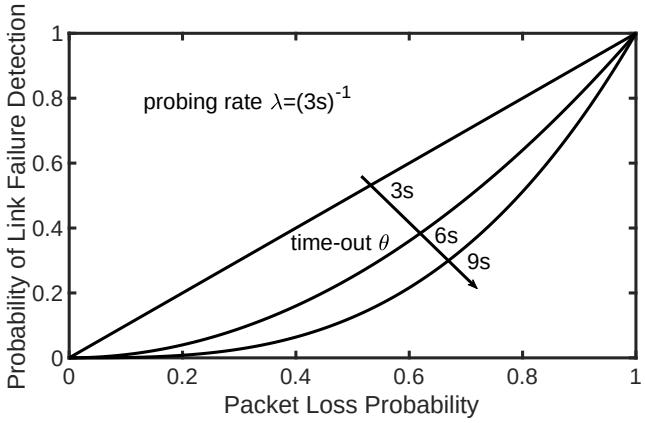


Figure 3.3: Calculus - detection probability for varying time-out θ.

equal to a minute. Increasing the packet loss further also decreases the detection time. But still at 100% packet loss the detection time is still at more than 10 s. Increasing the probing rate from 1/3 s to 1/2 s also decreases the detection time. Here, mean detection times of less than 60 s can be found for a packet loss value of circa 35%. For 100% packet loss the mean detection time is around 8s. Finally, for a probing frequency 1/1 s, the mean detection time decreases even further. Here, for packet loss values around 25% lead to a detection time of around 60 s. The final detection time for loss values around 100% is less than 5 seconds. Decreasing the time-out value to the probing frequency also drastically decreases the detection time. Here, already for values below 10% of packet loss, detection times below 60 s can be found. Again, increasing the probing frequency leads to better results, but this time the improvement is smaller in comparison. The evident kink at 70% is caused by the minimum function in equation 3.4.

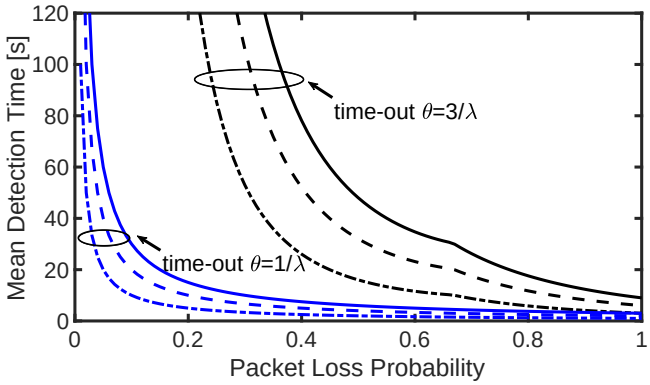


Figure 3.4: Calculus - mean detection time.

3.2.2 Practical Performance Evaluation

Figure 3.5 shows the testbed used for the measurements of this section. One physical server is running Mininet to emulate a SDN topology of four switches. The topology of the testbed is a ring topology with four switches. To each of the switches one simulated host is connected. Two physical servers form the ONOS controller cluster. The switches are load-balanced between these two nodes, i.e. one controller node controls two switches.

In order to test the detection capabilities of ONOS for the packet delay, packet loss is configured on the data plane link between Switch 1 and 2. To be able to determine the reactions and their delay, the signaling traffic between the controllers and their connected switches is recorded and evaluated after each run. Analyzing these traces allows us to calculate the mean reaction time and the detection probability of each scenario.

In this section we evaluate the detection capabilities of ONOS for the case of packet loss based on measurements. Finally, we compare these results with the predicted performance presented in Section 3.2.1.

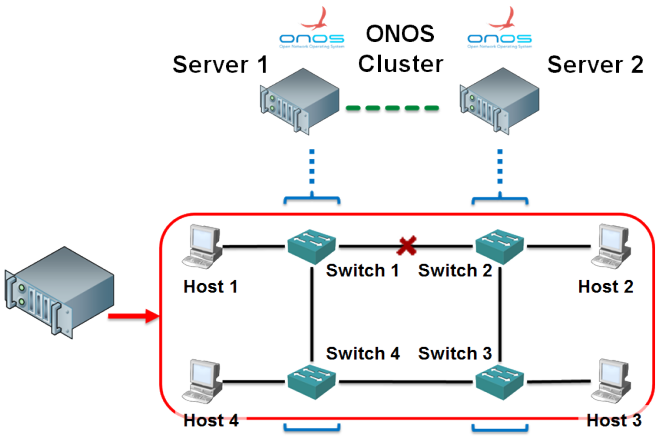


Figure 3.5: Testbed overview.

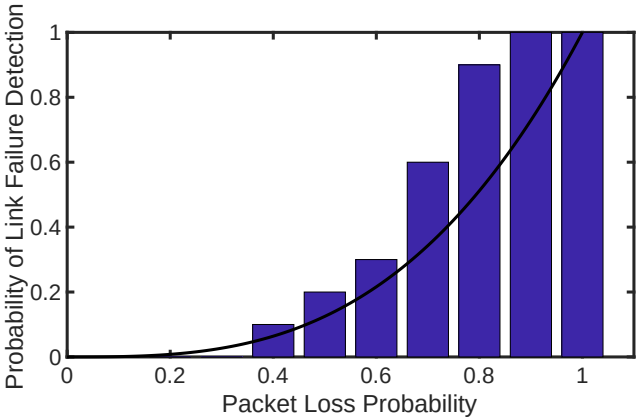


Figure 3.6: Number of successful detections for the ONOS controller for the case of packet loss.

In Figure 3.6 the detection probability of the ONOS controller for multiple packet loss values is depicted. The x-axis shows the configured packet loss in the data plane, the y-axis the detection probability after 10 measurement repetitions with a measurement duration of 120 seconds each. The blue bars visualize the measurement results, the black line show the according model results of Figure 3.2 with σ=9 s and λ=1/3 s for comparison. These measurement results only begin after a configured packet loss value of 30% as ONOS is unable to detect any change in the inter-connection of the connected switches for lower values. Taking a look at Figure 3.4 affirm these results, as the expected mean detection time for packet loss below 35% is beyond our measurement duration of 120 s. Beginning with 40% packet loss ONOS slowly and unreliably begins to detect the change in the link quality with a detection probability of 10%. Increasing the packet loss further to 50% and 60%, the detection probability rises to 20% and 30%. For packet loss values beyond 70% ONOS is able to reliably detect a change in the network conditions with a detection probability of 90%. After that the detection probability remains at 100%.

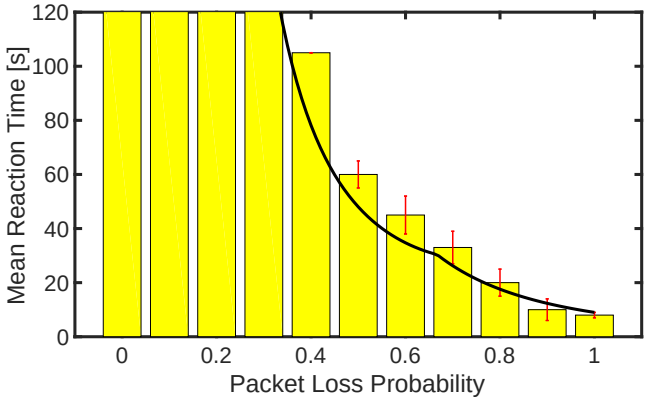


Figure 3.7: Mean reaction time of the ONOS controller for the case of packet loss.

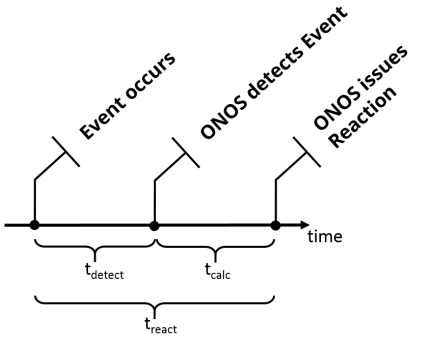


Figure 3.8: Composition of the measured reaction time.

Figure 3.7 shows the mean reaction time of the ONOS controller in this measurement scenario. The x-axis shows the range of measured packet loss values. The y-axis depicts the mean reaction time in seconds, respectively. Each result is shown as a yellow bar, additionally, 95% confidence intervals are shown in red. For the purpose of comparison, the calculated model values are shown as a black line. As these results have been derived from the same measurements as the ones from Figure 3.6, there are no results for the mean reaction time for 30%, depicted here as a bar filling the whole height of the figure. With a configured packet loss value of 40%, ONOS takes 105 s to detect a change in the data plane. For 50% packet loss, the reaction time decreases to around 60 s. A packet loss of 60% leads to reaction time of around 45 s. 33 s are required for a detection of 70% of packet loss in the data plane. Further increasing the packet loss value from 80% to 100% leads to a decrease of the reaction time to 20 s, 10 s, and 8 s. As the confidence intervals show, different variances of the reaction time have been recorded. Throughout the results, all confidence intervals are small, indicating a low variance of the results

Comparing these measurement results to results with the model in Section 3.2.1, it confirms our theoretical analysis. However, for the detection time an offset between measurements and model can be seen. This can be attributed

to the difference in the two displayed metrics: detection time vs reaction time. With the detection time t_{detect}, we express the time the internal detection mechanisms of ONOS require to realize that a link is exposed to hazardous conditions. The reaction time t_{react}, as depicted in Figure 3.8, actually can be expressed as a sum: $t_{react} = t_{detect} + t_{calc}$, where t_{calc} identifies the time ONOS requires to calculate a the reaction to the detected event. Within our testbed we are unable to measure t_{detect} directly, therefore, we only capture t_{react}.

3.3 Towards an Active Probing Application for the ONOS SDN Controller

As observed in the last section, ONOS, in its current version, is unable to meet service provider requirements in certain scenarios. As the approach of ONOS has several shown limitations, we n to create a new detection application for the ONOS controller that increases the detection performance for the case of packet loss in the data plane.

First, before designing the application, required features have to be defined. The new probing application should be able to detect packet loss and link delay in the data plane by using an active probing mechanism. As we rely on the ONOS controller, the app should be designed to interact with it using the Northbound API. This API offers a well-defined interface between additional applications, such as our detection mechanism and the connected network devices. The controller offers information on the status of the devices, for example flow traffic statistics, and transforms actions from an application into device-specific commands, e.g. the creation of a route between two hosts using multiple switch hops. As the controller is the "brain" of an SDN, and most network operation comes to a halt without it, the application should not generate a utilization overhead on the controller resources such as CPU load or memory utilization.

The remainder of this section features a description of the general concept of the application, and gives insights into the most important components, such

as the structure of the probing packets and how the algorithm is implemented. Finally, its functionality and performance is evaluated with measurements in a testbed.

3.3.1 Concept of the Application

Figure 3.9 shows the schematic concept of the application. On top, the active probing application is depicted, at the bottom the controller. After the activation of the probing application, the initial configuration is loaded and the current topology known to the controller is requested ❶. Based on this topology, the application now selects a new link between two data plane devices for the probing process ❷ and automatically generates a new probing packet ❸. This packet is then handed over to the controller via the Northbound API. The controller then injects this packet through its Southbound API into the data plane ❹. At the sink of the link, the probing packet is automatically redirected to the controller, which, in turn, passes this packet to the active probing application ❺. The application now processes this packet and updates the links properties, e.g. the packet loss and the link delay ❻. If one of these properties exceeds a threshold, for example the packet loss is greater than 10%, a notification is triggered and sent to the controller via the Northbound API ❼. The controller then is able to calculate an appropriate reaction, e.g. recalculating the routes of flows that traverse a link with bad quality.

3.3.2 Probing Packet Structure

For each link, an INFO3LLDP probing packet is sent each predefined probing interval. The packet structure expands the IEEE 802.AB LLDP standard [59]. The exact structure consists of an Ethernet frame, a number of mandatory LLDP TLVs (Type-Length-Value), an optional TLV list, here depicted as *LS3 Content* and finally an "end of LLDP" TLV, see Figure 3.10. Setting the destination MAC address of the Ethernet frame to a special multicast address, and the EtherType of this packet to 0x88cc ensures that network devices receiving this packet are

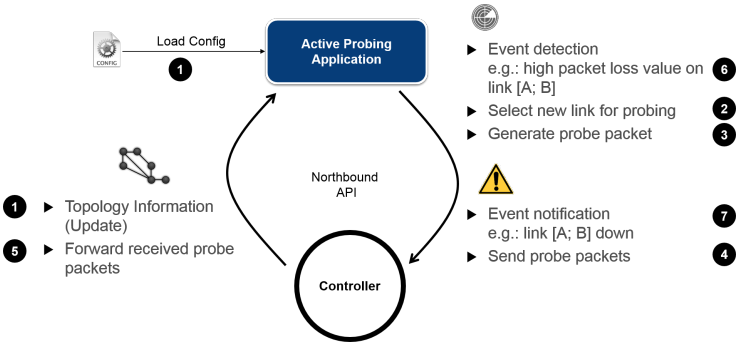


Figure 3.9: Schematic concept of the application.

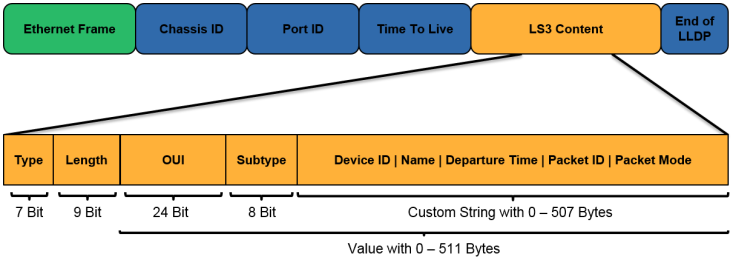


Figure 3.10: LS3LLDP packet structure.

able to handle it. Each of the TLVs, regardless of mandatory or optional, has three fields. Field 1 is the type of the TLV with 7 bits (e.g. 1 for chassis id or 2 for port id). Field 2 is 9 bits wide and describes the length of the following value. Field 3 is the actual value with a length ranging from 0 (minimum) to 511 (maximum) bytes. This structure is used for each TLV, even for the optional TLVs and the "end of LLDP" TLV. The mandatory TLVs include the chassis id, the port id, and the time-to-live. For the advanced purposes of our probing application, we use optional TLVs with a unique OUI (organizationally unique identifier), in order to identify as packets generated from our application, and multiple subtypes

followed by a value. These subtypes are *device id*, *name*, *departure time*, *packet id*, and *packet mode*. The device id identifies the source switch of this packet. The name field is used to identify this packet as an INFO3LLDP packet. The departure time field is set to the system time of the host of this packet at the time of creation and is used to calculate the transmission time of a packet upon reception. The packet id is incremented for each packet and is used to determine packet loss on a link, as explained later on. The effect of the packet mode will also be described below. In order to transmit multiple values (e.g. the whole set) via one single LLDP packet, for each information a new TLV is created and attached to the packet.

3.3.3 Implementation of the Application

As soon as the extension has been activated, it creates multiple internal data structures for each link. For each link of the topology, the following data is stored within the application:

- **Link History**: saves the history of probes, tracks packet loss and link delay
- **Outstanding**: a list that tracks probe packets that are currently in transmission
- **Requested**: a list that tracks requested *emergency* probe packets

Each entry of the *Link History* saves the following data: id of the source port including the switch, id of the target port (including the switch it belongs to), a list of the last 300 received probing packet ids, the timestamp of the last received packet, and a time-out which describes when the next probing packet is expected.

Figure 3.11 depicts an overview on the interconnections of the different parts of the active probing application. In total, there are five parts in the application: the *Sender*, the *Receiver*, *Update History*, *Control*, and *Emergency Probing*. In the

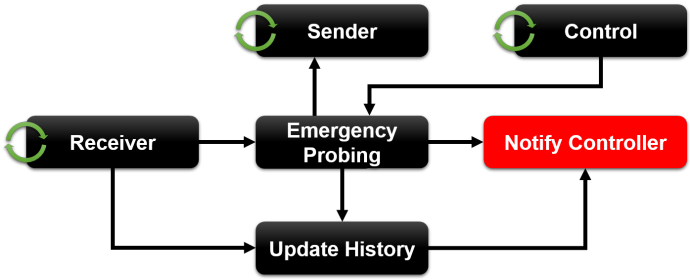


Figure 3.11: Interconnections between the parts of the application.

top middle of the figure there is the *Sender*. It regularly generates and transmits probing packets for each link of the topology. The *Receiver* is at the middle left of the figure. Its purpose is to receive incoming packets and decode their information. Depending on the packet type the information is passed on to *Update History*, or to *Emergency Probing*. *Update History* receives the information of one probing packet at a time and updates the corresponding statistics. If any threshold is exceeded, a notification to the controller is triggered, depicted by the red *Notify Controller* box in the figure. The *Control* part at the top right regularly checks for each link if the probing time-outs are within their limits. If a deviation from these thresholds is detected, the link is put under investigation in *Emergency Probing*, shown in the middle of the figure. Here, the *Sender* generates and transmits "emergency" packets. If these packets do not arrive within a time-out, the link under investigation is considered down and a notification to the controller is triggered. If these packets arrive, the investigation status is canceled and the probing packets are passed on to *Update History*. The exact functionality and procedure of each part is explained in detail in the following.

Sender Figure 3.12 shows the details of the *Sender*. Regularly for each link, a probe is generated and transmitted using the ONOS northbound API. In order to be able to track the probes, the time of its transmission is recorded and a new

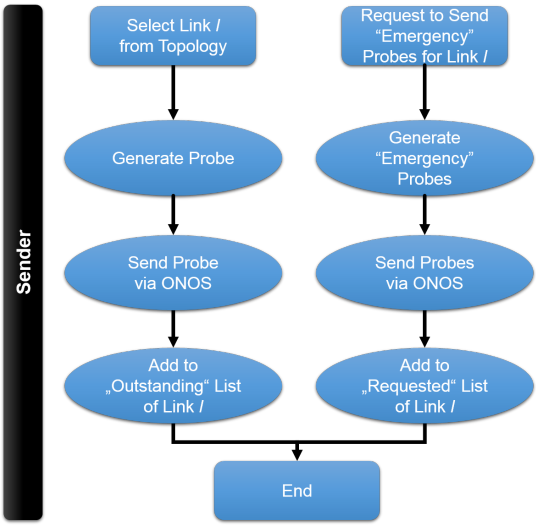


Figure 3.12: Flowchart of the sender.

entry containing the packet id is added to the *Outstanding* list of this link is added. The other function of this part is to generate emergency probing packets for a specific link on behalf of the *Emergency Probing* part. The procedure is similar to the first one, the only difference is that these packet ids are added to the *Requested* list.

Receiver & Update History In Figure 3.13 the internals of the *Receiver* are depicted. As soon as a probing packet is received by the probing application, the information of the packet is decoded and the reception time is recorded. Out of this information, a new link object is created. If this link object is found in the *Outstanding* list of a link, this object is removed from the list and the information is passed on to *Update History* procedure. If this object is not found in this list, a match is searched for in the *Requested* list. If a match is found, the information is

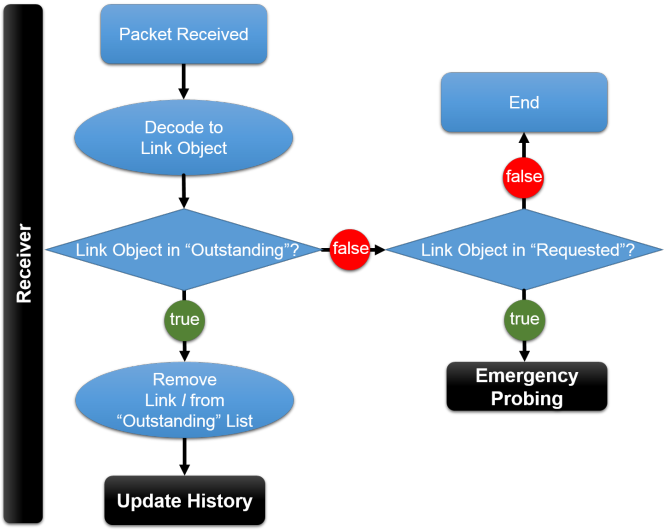


Figure 3.13: Flowchart of the receiver.

passed on to the *Emergency Probing* part of the application. If no match is found in both lists, the packet is discarded.

As soon as a probing packet is passed on to *Update History*, shown in Figure 3.14, the packet id is put into the list of the last received packet ids and the link latency now is calculated via the difference of reception and transmission time. Based on the list of received packet ids, the packet loss rate of a link can be calculated. The list always contains 300 entries, where the first entry is the id of the oldest received packet, the last entry indicates the id of the last received probing packet. As soon as a new packet is received, it is put at the end of the list and the first entry is removed. Now, based on the difference of the last packet id and the first id in the list, one can be calculate how many packets have actually been received.

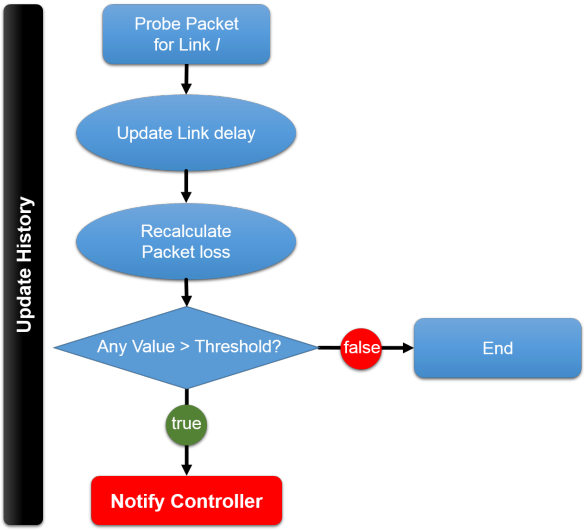


Figure 3.14: Flowchart of the history update procedure.

An example is illustrated in Figure 3.15. Here, the last packet id is 345, the oldest is 15. The difference of both ids is 330. As the length of the list is 300, the packet received rate is 300/330 = 0.91. Therefore, the packet loss rate is 1-(packet received rate) = 1 − 0.91 = 0.09.

Afterwards, it is checked whether all thresholds for all metrics are not exceeded. If a transgression is found a notification to the controller is triggered.

Control & Emergency Probing In the *Control* part of the application, as shown in Figure 3.16, each link is additionally regularly checked for exceeded time-outs. Therefore, the oldest entry of the *Outstanding* list of each link is checked for its transmission time. If the application does not receive a probing packet for a predefined time-out θ, the link is put under investigation and the *Emergency Probing* procedure, depicted in Figure 3.17, is triggered: more probe

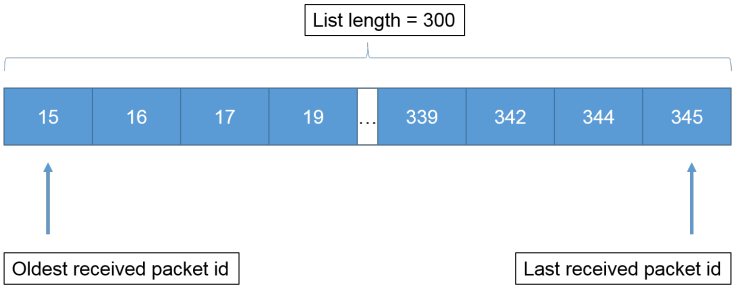


Figure 3.15: Exemplary packet history.

packets are sent via this link within a short period of time. If these packets are received within a predefined time interval, the above progress for the reception of a probing packet comes into play, c.f. *Receiver*, and this investigation is stopped and the probing process returns to its normal mode of operation. If not, this link is considered down and its updated status is forwarded to the controller which then is able to take appropriate actions, e.g. to adopt the routing of the installed flows.

3.3.4 Evaluation of the Benefits of the Active Probing Extension

After designing and implementing this application its functionality and performance has to be evaluated. Therefore, measurements comparing the detection performance of ONOS with its own detection application and with our new active probing application have been conducted. For this purpose the introduced testbed of Section 3.2.2 has been used again. The only difference is in the configuration of ONOS. For scenarios evaluating the active probing application, the vanilla detection application has been disabled and, accordingly, the new probing app has been installed and activated. The settings of the application were configured to a probing rate of 30 ms and a time-out θ of 100 ms. For the vanilla

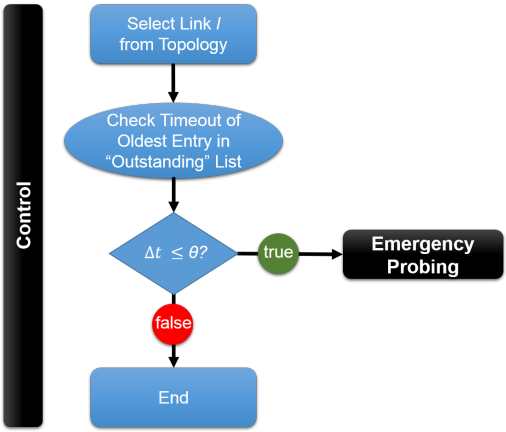


Figure 3.16: Flowchart of the control thread.

application the default values of a probing rate of 3000 ms and a time-out θ of 9000 ms were kept.

At first we will investigate the resource utilization of the application in comparison to a vanilla ONOS configuration. Afterwards, the enhanced detection performance is evaluated using different scenarios and configurations. The presented data represents the mean values of 10 repetitive measurement runs. If not mentioned separately, a packet loss value of 10% has been configured in the data plane.

Resource Utilization

Figures 3.18 and 3.19 show the resource utilization on side of the controller host with and without using the active probing application. Of course, the measurements contain the collection of the statistics from both servers. However, our investigations have shown that their behavior only differs marginally, and, therefore, it is sufficient to display only the results of one of the hosts. The red

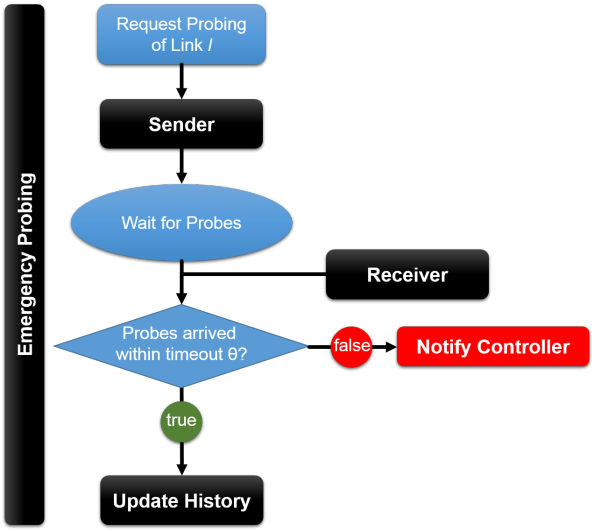


Figure 3.17: Flowchart of the emergency probing.

line displays the usage without the application, the blue line the usage with the application installed and activated. Figure 3.18 shows the mean CPU utilization during the whole time of the measurement. The x-axis depicts the measurement time in seconds, the y-axis the mean CPU usage in percent. Both measurements show only a small CPU usage varying between 1 and 3% throughout the measurement. Additionally, the measurement with the extension enabled is only marginal higher than the one without.

Figure 3.19 shows the mean memory usage with and without the active probing application. The x-axis shows the measurement time in seconds, the y-axis the memory usage in megabytes from 0 to 1200 MB. Again, only a small difference is noticeable: whereas the measurement without the probing application starts at 1000 MB and slowly decreases to a final value of 950 MB, the measurement with the app activated requires 1050 MB and slowly decreases to 1000 MB.

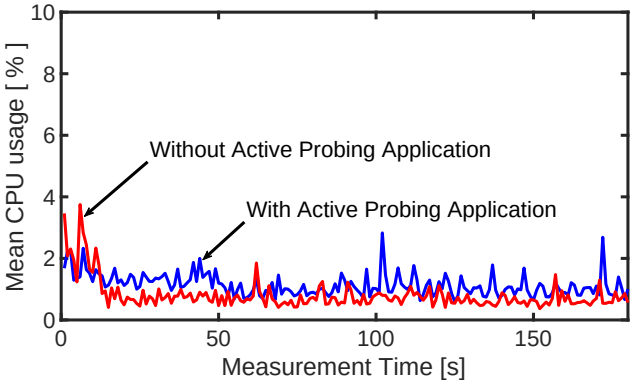


Figure 3.18: Mean CPU usage with and without the active probing application enabled.

Nevertheless, as the host of this controller instance has had a total of 16 GB of memory available, both memory usages do not carry any weight.

Looking at the resource utilization, the goal of not overloading the controller host with our application has been accomplished. For the case of the CPU load, the utilization more or less remains the same. For the case of the RAM load, the usage even is smaller than before. The reasons for this behavior are simple: Whereas the vanilla ONOS probing mechanism creates instances for each connected switch, our application only runs one instance per controller host. Therefore, less RAM is required.

Detection Performance

In order to determine the detection rate and the detection time, the threshold for the application has been set to the corresponding packet loss value configured in the data plane. As soon as a the *Update History* procedure notifies the the controller, the successful detection is logged and the detection time is calcu-

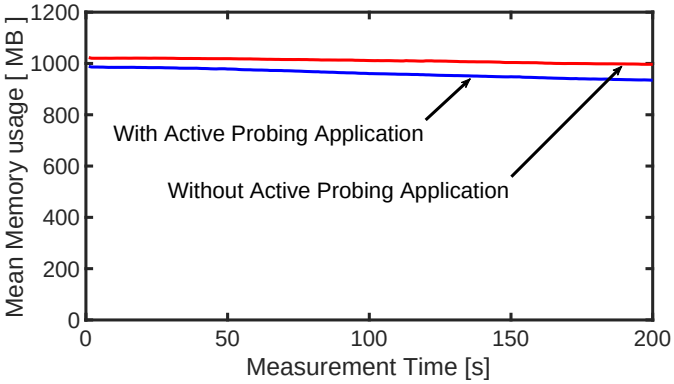


Figure 3.19: Mean RAM usage with and without the active probing application.

lated. Figure 3.20 shows the detection rate of the active probing interval after 10 runs with a measurement time of 120 seconds each. The x-axis displays multiple configured data plane packet loss values, ranging from 3 to 50%. The y-axis displays the detection rate in percent. For this scenario, the packet loss threshold of the application has been set to the corresponding packet loss value configured in the data plane. As soon as For the lowest packet loss values of 3 and 5%, the probing application offers a detection rate of 10%. For 10% and 15% packet loss the detection rate increases to 70%. A detection rate of 90% has been measured for the packet loss values between 20 and 30%. Beyond that packet loss rate the application is in 100% of the cases able to detect the configured packet loss in the data plane. There is a trade-off between detecting all packet loss rates and generating not too much additional traffic.

In Figure 3.21 the reaction times of the active probing application for multiple packet loss values are depicted. The x-axis shows the packet loss values ranging from 3 to 50%, the y-axis the reaction time in seconds. Beginning with a detection time of around 96 seconds for 3%, the detection time continuously

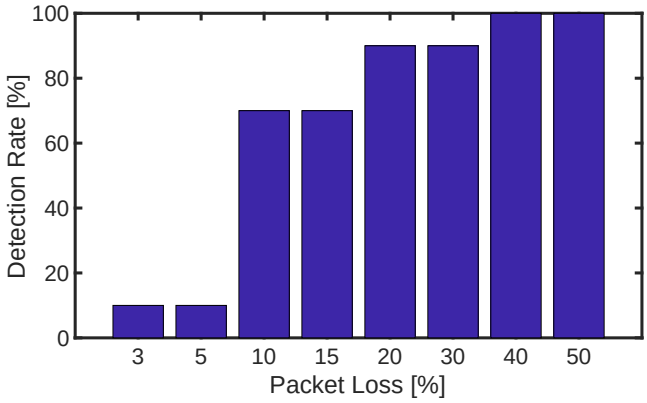


Figure 3.20: Detection rate of the active probing application.

decreases with an increasing packet loss value. For 5% the reaction takes place within 85 seconds. 10% of packet loss is detected within 47 seconds, 15% within 42 seconds, 20% after 40 seconds. Roughly 37 seconds required to detect 30% of packet loss, 29 seconds for 40%, and, finally, 50% of packet loss are detected within 19 seconds.

In summary, the results of Figures 3.20 and 3.21 demonstrate the increased detection performance of the active probing application. In terms of detection rate, the new application surpasses the performance of the native ONOS detection mechanism. Our model, and the provided measurements, in Section 3.2 show a detection probability of 10% for a packet loss probability of 40%. The new application, in comparison, already offers this detection rate for packet loss values of 3%. Additionally, the detection times have decreased as well. Before implementing the active probing mechanism, the detection time for packet loss value of 40% was around 100 seconds. After the successful implementation of the app, the detection of this link status is already possible after 29 seconds. Reviewing the resource utilization on side of the controller host also reveals that the goals

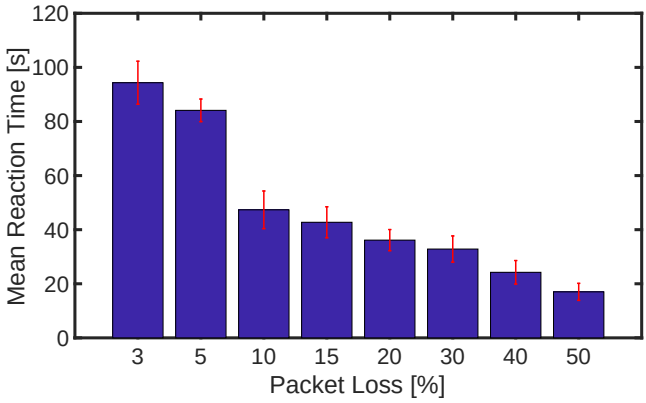


Figure 3.21: Reaction times with the active probing application enabled.

in terms of overhead have been kept. The new application offers a better detection performance whilst requiring the same or even less CPU or RAM usage.

3.4 Adapting to Change

The proposed mechanisms for the active probing application does not yet consider one parameter that requires consideration: the constant change in the network, especially the variance in the packet inter-arrival times, i.e. the jitter. The remainder of this section focuses on how jitter does impact the probing process of the native ONOS detection mechanisms, how the new active probing application is also affected by it. Finally, a mechanism to cope with this is proposed and its performance is evaluated.

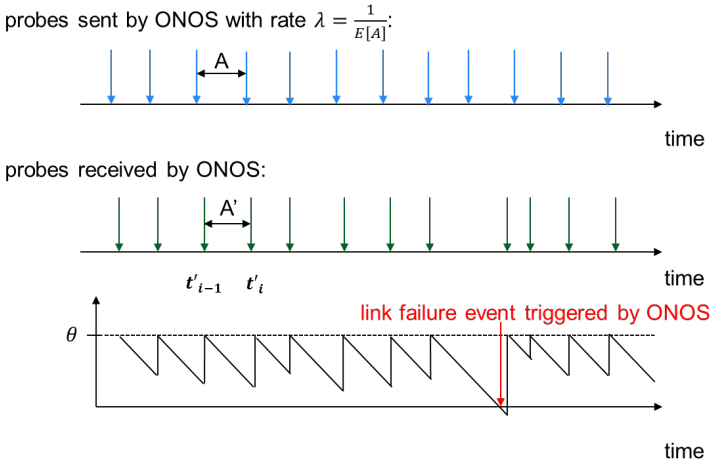


Figure 3.22: Probing process with false positive caused by jitter.

3.4.1 False Positive Failure Rate

The results from Section 3.2.1 show that the mean detection time of a link failure indicated by packet loss depends on the probing rate λ and the time-out θ. As shown in Figure 3.22, the jitter on the link scatters the arrivals of the successful probes at the controller. On a link with no packet loss the inter-arrival time of two successful probing packets A' can also exceed the time-out, due to jitter on the link. Hence, in this case a false positive link failure event is triggered even if no packet is lost. These false positive link failure events need to be avoided, as they would interrupt the operation of a healthy system. Especially, if the probing frequency is high, a too low time-out θ can lead to false positive link failures due to jitter.

In order to evaluate the rate of false positive link failures, we implement a Monte-Carlo simulation that simulates the probing process of one link. The simulation time is T, probes are sent with rate λ and are dropped with probability

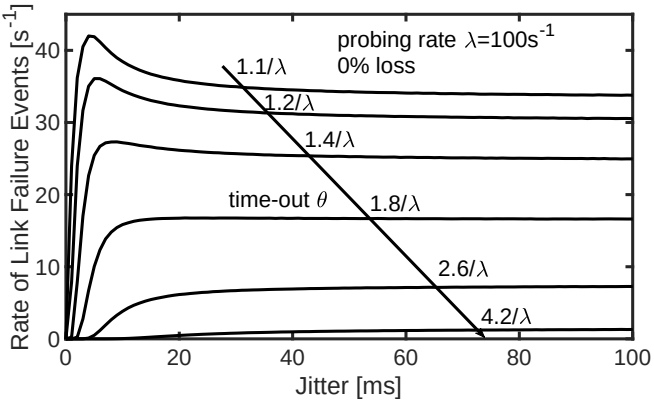


Figure 3.23: Rate of false positive link failure events caused by jitter.

p. Each probe is delayed by a normal distributed random time with parameters $(0, \sigma)$. The probing time-out is θ.

As performance metrics we consider the rate of link failure events

$$f_{fail} = \frac{1}{T} \sum_{i>0} X_i,$$
$$\text{where } X_i = \begin{cases} 1 & t'_i - t'_{i-1} > \theta \\ 0 & \text{else} \end{cases} . \tag{3.5}$$

In case of $p = 0\ \%$, f_{fail} is also the rate of false positive link failure events.

Figure 3.23 shows the rate of failure events for 0% packet loss, i.e., the rate of false positive link failure events caused by jitter. The probing rate is $1/100\ s^{-1}$, and the time-out is varied relative to the average inter-arrival time of probes $\frac{1}{\lambda}$. The results show that a small time-out θ leads to high false positive rate. This leads to a high probability of flapping of the link status. With increasing jitter a

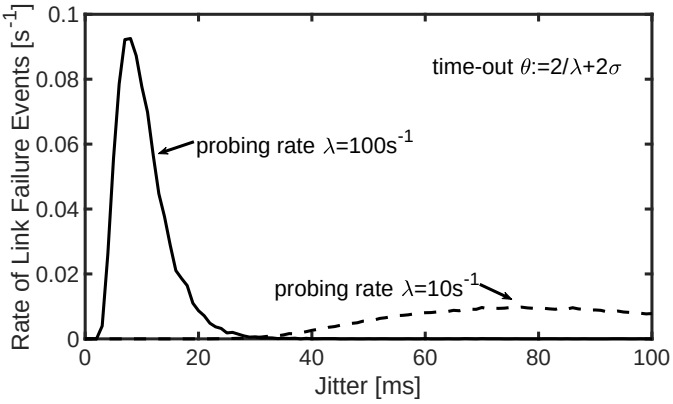


Figure 3.24: Rate of false positive link failure events for Equation (3.6).

longer time-out θ is necessary to keep the rate of false positive link failures low. Hence, for effective operation with a low rate of false positives, the time-out θ has to be set depending on the jitter on the link. As rule of thumb the time-out θ can be set using a margin of 2 times the jitter:

$$\theta := 2E[A] + 2\sigma = 2/\lambda + 2\sigma \quad . \tag{3.6}$$

Figure 3.24 shows the rate of false positive link failures for the rule of thumb setting for 100 and for 10 probes per second. The result shows that using the rule of thumb setting the less than one false positive link failure event is triggered every 10 seconds.

The minimum value for the time-out θ can be calculated by evaluating the inverse cumulative distribution function of a normal distribution $N^{-1}_{1/\lambda,\theta}(p)$ with mean $\frac{1}{\lambda}$ and standard deviation σ depending on the tolerated false positive rate $\bar{f}_{fail}$:

$$\theta_{min}(\lambda, \sigma, \bar{f}_{fail}) = N^{-1}_{1/\lambda,\sigma}(1 - \bar{f}_{fail}) \quad . \tag{3.7}$$

3.4.2 Automated Probing Rate Adaptation in the App

As shown in the previous Section 3.4.1 the time-out θ has to be increased depending on the jitter on the link to fulfill a certain target rate of false positive link failure events. To fulfill a certain target detection time, as shown in Section 3.2.1, the time-out θ can be reduced or the probing rate λ can be increased, which in turn affects the rate of false positive link failures. Hence, there is a trade-off between a low rate of false positive link failure events and a low failure detection time. To keep the load on the controllers low, the probing rate has to be minimized. Therefore, we propose an algorithm for a self-optimized process that meets the target rate of false positive link failure events $\bar{f}_{fail}$ and the target mean detection time $\bar{t}_{detect}$ given the tolerated packet loss rate $\bar{p}$ and minimizes the controller load.

Algorithm 1 Probing Rate Adaption

procedure PROBING RATE ADAPTION
input parameters:
    tolerated packet loss rate $\bar{p}$
    tolerated false positive rate $\bar{f}_{fail}$
    target detection time $\bar{t}_{detect}$
variables:
    jitter σ
    probing frequency λ
    time-out θ
start:
determine $\theta(\lambda, \sigma, \bar{f}_{fail})$ based on Equation 3.7
calculate expected detection time $t_{detect}(\bar{p}, \theta, \lambda)$ based on Equation 3.4
set probing frequency $\lambda := \lambda \cdot t_{detect}(\bar{p}, \theta, \lambda)/\bar{t}_{detect}$
go to *start*
end procedure

At first, the new time-out θ is calculated according to Equation 3.7. Afterwards, by using the new time-out, the new expected detection time is calculated according to Equation 3.4. Finally, the new probing frequency is set to the old probing frequency times the division of the new expected detection time by the detection time at the beginning of this iteration. Afterwards, the process restarts at the beginning, and, therefore, is always adapting to the current network situation.

3.4.3 Integration with the Active Probing Application

In order to take advantage of the proposed auto-adapting algorithm, it has to be implemented and integrated with the already existing active probing application. The currently only missing information inside the application is the current network jitter. The probing frequency and the time-out are both configuration parameters that can be read from the application. The target variables tolerated packet loss rate $\bar{p}$, tolerated false positive rate $\bar{f}_{fail}$, and the target detection time $\bar{t}_{detect}$ are input parameters that will be loaded upon start-up of the application.

The app already measures the one-way latency of each connected link with each transmitted probing packet. Therefore, to determine the network jitter, only a new data field has to be created that stores the difference in link delay from one measurement point to the next one. The remainder of the implementation is putting together the data and implementing Algorithm 1.

Figure 3.25 shows the new optimization cycle of the application. The active probing extension measures the packet delay, the packet loss and the jitter of a link of the network topology. These values are then put into the algorithm. Based on the predefined configuration parameters, the algorithm then calculates a new probing frequency and a new probing time-out for the current environment conditions. This output is then used to modify the actual probing frequency and the probing time-out of the active probing extension. As network environment parameters such as link delay, network jitter, and packet loss tend to change over time, this whole optimization process is repeated regularly.

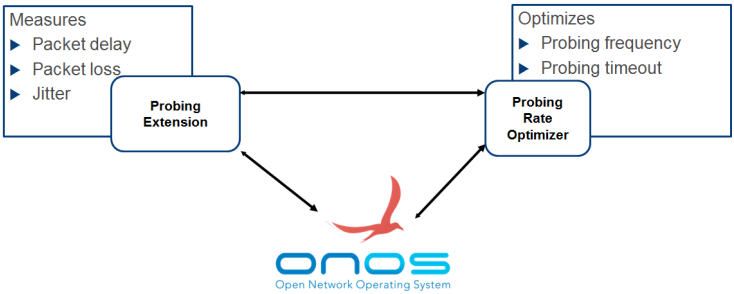


Figure 3.25: Implementation of the self-adaption mechanism.

3.4.4 Evaluation

In order to highlight the advantages of the self-optimization algorithm, measurements with and without activated self-optimization have been conducted. In order to proof the functionality of the auto-optimization feature of the active probing application new measurements have been performed. Again, the testbed of Section 3.3.4 has been used, now with the additional auto-optimization feature.

At first the actual adaption of the probing rate under changing jitter levels is investigated. In this scenario, the probing interval for the probing extension without the optimization has been set to 100 ms and the time-out-factor to 3. Figure 3.26 shows the jitter interval in orange, the probing interval for measurements without the optimization algorithm enabled in blue and the probing interval for measurements with the optimization algorithm enabled in red. In this scenario, the jitter level changes each 50 seconds. For the first 50 seconds the jitter level is at almost 0 ms. For the time interval of 50 to 150 seconds, the jitter is increased to 10 ms for 50 to 100 seconds, and to 20 ms for 100 to 150 seconds. Afterwards, the jitter level is relaxed to 0 ms, before it is increased to 50 ms for seconds 200 to 250. After another 0 ms jitter phase from 250 to 300 seconds,

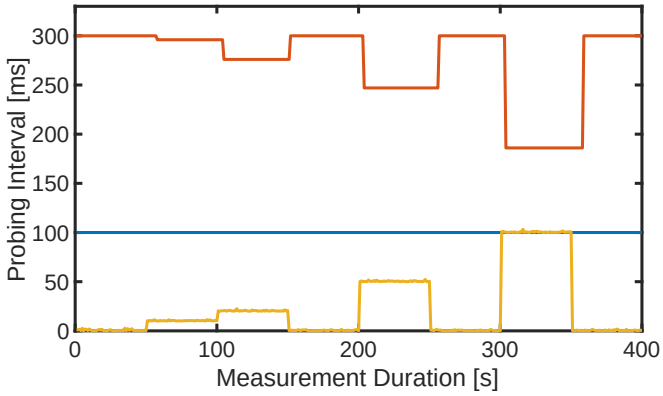


Figure 3.26: Impact of jitter on probing interval.

the jitter is increased to 100 ms for 50 seconds. For the remaining measurement time the jitter is again reduced to 0 ms.

With the probing interval of the probing extension without the optimization the change in the jitter level does not impact it in any way. The probing interval is constantly at 100 ms. The optimized version, in contrast, shows an adoption to the changed networking environment. As the probing extension has been set to a desired detection time of 300 ms, the probing interval at first is at 300 ms and the time-out factor at 1.0, leading to a time-out value of 300 ms. With the detection of the rise in jitter, the algorithm now changes the probing interval to 296 ms and the time-out factor to 1.043. This small change can be a drastic difference in the actual reaction time if a failure happens with such environment parameters. As introduced in Section 3.4.1, a small change in the jitter level can have a huge impact if the probing extension does not react to it. As soon as the jitter increases, the time-out factor increases as larger time-outs lead to reduced false positive rate of link failure events. Additionally, as demonstrated later on, this change in the probing frequency has an impact on the resource utilization

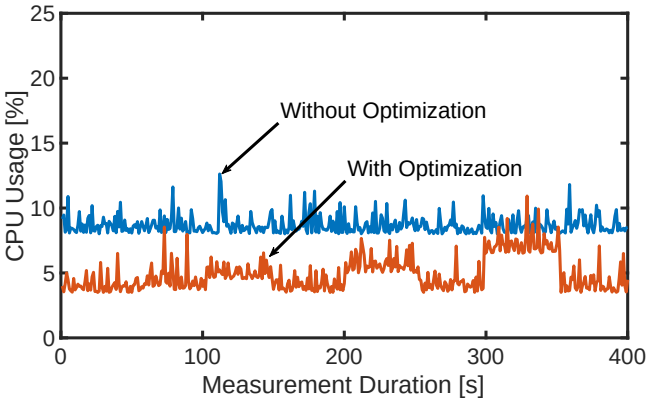


Figure 3.27: CPU usage both with and without the optimization algorithm enabled.

of the controller host. The rise of the jitter level from 10 to 20 ms at 100 seconds leads to a changes of the proving interval to 276 ms and only has a minor impact on the time-out factor. For the reset of the jitter to 0 ms, the interval and the time-out factor again are set back to 300 ms and 1.0. Increasing the jitter level to 50 ms decreases the probing interval to 247 ms and the time-out factor to 1.259. For the final jitter level of 100 ms the optimization result sets the probing interval to 186 ms and the factor to 1.689. Throughout the whole measurement the adaption time of the optimization mechanism is in mean 3 seconds. This factor is influenced by two factors: First, the probing extension has to detect the jitter level, which takes some time, as probing packets have to be transmitted, received and analyzed. Second, as we use multiple controllers, a synchronization between these two nodes is required. For this setup, the synchronization time has been set to 5 seconds. Decreasing these values is possible, but would not necessarily increase the detection performance.

Figure 3.27 shows the CPU usage for above scenario. The CPU usage for the controller host without the optimization algorithm enabled is shown in blue,

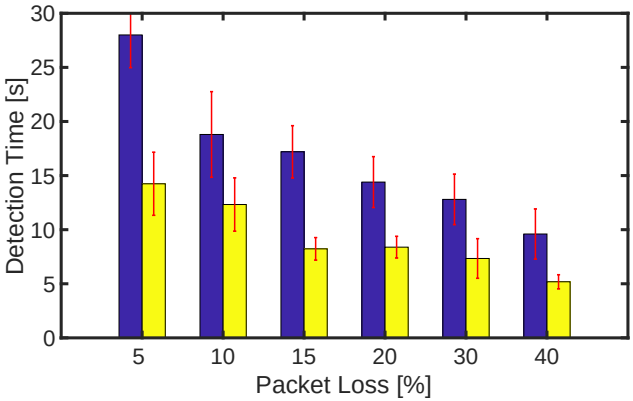


Figure 3.28: Reaction times to packet loss events both with and without the optimization algorithm enabled.

the usage with the optimization Algorithm enabled is shown in red. Here, the previously mentioned difference in the CPU usage is visible. Without the optimization, the CPU usage is always in between 8 and 11%, with the exception of a few peaks. With the optimization enabled, the CPU usage is lower throughout the measurement, with values between 3 and 8%. Each time the probing interval is increased, the CPU load is reduced, and vice versa. The memory utilization for both with and without the optimization has also been analyzed. But, as the measurement showed little to none difference in the results, they are not shown here. The memory utilization was at levels that have been shown in Section 3.3.4.

In order to measure the detection time for the case of link failure in the data plane, the scenario settings have been changed. The optimization algorithm calculates a probing interval of 208 ms and a time-out factor of 1.2. For the non-optimized variant, a probing interval of 100 ms and a time-out factor of 3 is chosen. As the actual packet loss in the data plane has no impact on the optimization process, the calculated parameters remain constant. In Figure 3.28 the

reaction times for both with and without the optimization algorithm are compared. For this scenario, the topology is set up and running without any packet loss for 150 seconds. Afterwards, as introduced in Chapter 4.1, packet loss is added to the link between switch 1 and switch 2. Finally, the network is set back to 0% packet loss for the next measurement. The x-axis of Figure 13 shows multiple packet loss values from 5 to 40%, the y-axis shows the detection time to a packet loss event in seconds. The values for the probing extension without the optimization algorithm are depicted in blue, the results with the optimization algorithm in yellow, respectively. In red 95% confidence intervals are shown.

For the 5% packet loss value, the non-optimized probing extension requires 28 seconds to detect a change in the data plane link, whereas the optimized variant reduces the detection time to 14 seconds. For 10% packet loss, the detection times of both variants are reduced. Here, the probing extension without the optimization enabled requires 19 seconds to detect a rise in the packet loss of a link. Enabling the optimization algorithm reduces the detection time to 12 seconds. Increasing the packet loss to 15% would decrease the detection times to 17 and 8 seconds for the active probing extension with disabled and enabled optimization algorithm. For 20%, the reaction time only decreases marginally for both optimization settings: the packet loss is detected in 15 seconds without the optimization, and stays at 8 seconds with the optimization. Increasing the packet loss to 30% and 40% further decreases the reaction times. The standard probing extension without any optimization algorithm requires 12 and 10 seconds. Enabling the optimization mechanism decreases these times to 7 and 5 seconds. Overall, the measurements present the expected results. In all cases increasing the packet loss decreases the reaction time.

Furthermore, the optimization algorithm constantly outperforms the non-optimized version of the probing extension. The decrease of the reaction time for increasing packet loss can be explained by the mechanism of the active probing extension. In order to detect packet loss on a link three consecutive packets have to be lost. Therefore, increasing the packet loss increases the probability of three consecutively dropped packets on a link. This phenomena has been intro-

duced an explained in Section 3.2. It also explains the performance gain of the optimization algorithm. The optimization algorithm analyzes the configuration parameters and calculates a suiting probing interval and time-out. In order to optimize the reaction time for the case of packet loss, it reduces the time-out. With this setting less than two consecutive packets have to be lost to trigger a detection, which has a higher probability of three consecutively dropped packets.

Overall, the performance gains of the optimization mechanism are as expected. With normal network environment parameters, close to 0% packet loss and 0 ms of jitter, the algorithm relaxes the probing interval and, therefore, requires less CPU resources. Furthermore, under harder network conditions, the optimization algorithm detects a challenging situation and decreases the probing interval in order to increase the detection performance.

3.5 Lessons Learned

Software-defined Networking is gaining momentum since its introduction at the end of the last decade. Through its decoupling of the control and the data plane of network devices, it is possible to configure the network in a very flexible and central manner. These advantages lead to an interest by service providers. But, before migrating to a new technology, providers have to be ensured that, despite the possible advantages, their reliability and availability requirements are met.

One of the currently most important SDN controllers is ONOS. According to the ON.Lab, the company developing this controller, it is very reliable and production ready. In this chapter we theoretically analyzed the capabilities of the detection mechanisms of ONOS and verified this analysis by exemplary measurements. According to the presented results, the implemented mechanisms only provide an unreliable detection and is therefore unable to fulfill the requirements of service providers. For example, it can take more than one minute to detect packet loss values of 50% in the data plane.

Reducing or disabling the threshold of failure mechanisms is a possible approach to increase the fast-failure reaction performance. Usually, most detection and reaction mechanisms have implemented safety margins, so that the controller does not already react to the slightest possible failure in the network. For example, the standard probing application of the ONOS controller only reacts to three consecutively lost probing packets. The reason for this behavior is that there always is a very small packet loss and jitter within a network. Therefore, it is only logical to take this in to mind when designing detection algorithms as so-called false-positive failure events can lead to congestion on the calculated backup links, which, in turn, would trigger a real link failure on that link. However, by decreasing these safety thresholds or even disabling them, a reaction to real failure can also be faster detected.

The approach presented in this chapter introduces an active probing mechanism for the ONOS SDN controller as a Northbound API application. It generates, transmits, and receives probing packets through and from the network. Therefore, it is possible to derive network data on a link level, e.g. the packet loss rate, or the link delay. The evaluation performed in a testbed shows that this application is able to increase the detection performance, both in rate and time, whilst only marginally increasing the load on the controller resources, such as CPU or RAM.

An additional feature presented in this chapter is to implement a self-adapting active probing mechanism into the ONOS controller that adapts to the changes in the network conditions, e.g. jitter. With a history over a number of active probes, a packet loss and jitter detection has been implemented. Furthermore, in order to trade-off controller load to detection performance, the time between two succeeding packets and the internal time-outs are adapted to the current network conditions. An algorithm that optimizes the whole probing process during run-time has been presented. This algorithm analyzes the current network environment parameters and calculates new probing intervals and probing time-outs, which have a large impact on the detection performance. Afterwards, the algorithm is implemented as part of the already existing probing extension.

Measurements demonstrating the difference between an enabled and a disabled optimization algorithm are presented. Due to the implementation of the algorithm, the probing extension is able to react to changes in the network environment parameters, such as jitter. Furthermore, under normal environment parameters, the probing interval will be increased and, therefore, the CPU utilization on side of the controller can be reduced.

Another approach to increase the fast-failure performance of the ONOS SDN controller is to implement a *hyperreactive* controller reaction. In regular detection and reaction mechanisms some application is analyzing the current network environment, e.g. through active probing, and, as soon as certain thresholds are reached, a link is registered as failing and the controller tries to reroute all flows traversing that said link. As sooner or later a failure in a network will occur, there are two approaches to further increase the reaction performance: 1) Pre-calculating backup routes, or 2) Reducing or disabling the thresholds. By pre-calculating backup routes the controller assumes that sooner or later a link failure will occur and sooner or later it will have to calculate alternative routes for flows.

Valuable time could be lost if this recalculation is done on-demand, i.e. only after a link failure is detected by the controller. Therefore, for each link, alternative backup paths are already pre-calculated and stored by the controller. After a link failure happens, the controller is now able to react to this failure faster by using the previously pre-calculated route. A further addition for this approach could be to actually install the backup paths inside the flow tables of the switches. Then, the controller only has to send a command to the affected switches to replace the active routing with the backup routes. With this approach more reaction time can be saved.

4 The Impact of Header Modification on the P4 Processing Performance

With the rise of Software-defined Networking (SDN) in both campus and data center networks, more and more network engineers use it due to its flexibility and appreciate its benefits. Especially in data centers the offered flexibility, programmability, and centrality of SDN is indispensable. With current state-of-the-art SDN technology, it is possible to dynamically populate the tables of the connected data plane devices through a flexible and logically centralized control plane via a vendor-independent protocol. That means that the network engineer can, by programming the SDN controller, dynamically react to events in the network and ever changing application requirements. Furthermore, through the central knowledge being available in SDN, it is also possible to harden the network against upcoming challenges, e.g. link failures or frequently recurring traffic spikes. But, when trying to flexibly change the data plane, still strict boundaries apply. With most networking hardware, especially those with high throughput, it is simply not possible to adapt the matching mechanics to new protocols. The currently most common control plane protocol OpenFlow does not allow for the dynamic definition of its flow table entry attributes that can be used for flow rule matching.

Despite common SDN-protocols that support features and functionality of SDN at suitable networking devices, such as the OpenFlow protocol, there is still a heterogeneity between device-support for different vendors, both in feature

and functionality. Very few vendors have successfully implemented full capabilities of the latest stable version, 1.5.1, or even one of the earlier versions. One of the reasons for this circumstance is certainly the short time spans between the major OpenFlow versions. Between the beginning of 2011 and the end of 2014 five versions of the OpenFlow specification have been released. Additionally, with OpenFlow devices are only able to match against specified header fields. This means that for each supported data plane protocol each field that should be supported as an attribute of a possible flow table entry, e.g. destination IP or TCP source port, requires a definition in the OpenFlow specification. Therefore, as the support for more data plane protocols has been added, the number of fields grew with each version. OpenFlow 1.0 only had 12 fields defined in its specifications, with version 1.4 the number already increased to 41 fields. This is another downside as only a few vendors are able to deliver programmable ASICs at networking devices that are capable of dealing with all of those fields in the fast-path of a switch. Otherwise, the packets have to be processed by the slow-path of a switch, meaning that each packet has to be processed by the CPU, drastically reducing the processing latency.

The reason behind this is the limitations of the availability of fast memory for the match tables, as with each combination of fields for a table entry, the memory requirements per entry rise. As not every vendor is able to cope with these requirements, differences in the behavior, feature set, and programmability between switches of different vendors and their implementation of the OpenFlow protocol were introduced. As a result of that network administrators and managers have to target their controller programming with its policies and services to the underlying hardware. This is a contradiction to one of SDNs principles: the vendor independence.

Despite the increasing number of fields, it still does not suffice for production deployment. For example, VXLAN support still is unsatisfactory and NVGRE is not supported at all. With the OpenFlow approach the only solution is to add further fields to the specification, which would increase the differences between

the devices of multiple vendors even further as the adoption time differs significantly.

Therefore, the idea to implement an interface that allows for flexible mechanisms that can parse packets and match against arbitrary fields at line rate became one of the cornerstones whilst designing the next step in the evolution of SDN. This is the problem P4 attempts to solve. P4 hereby stands for *Programming Protocol-Independent Packet Processors* which describes the goals of this new approach. Being able to specify by programs how a switch processes packets offers protocol independence. Instead of the former bottom-up design of SDN, e.g. with OpenFlow, where the network hardware is *telling* the operator what and how it is able to process packets, P4 allows for a top-down design where the operator is able to *tell* the hardware how he wants it to behave and how he wants to switch packets. Additionally, the aim is to create a target independent solution that is suitable for both hardware and software switches. In contrast to the OpenFlow approach, network engineers can change the way the network devices process packets after they are deployed.

This difference in the approach is visualized in Figure 4.1. On the top, the well-known SDN control plane is presented, at the bottom a target switch. With OpenFlow it is only possible to install and query rules from the switch. P4 adds another layer of configurability: it allows the user to modify the parser and table configuration. Nevertheless, OpenFlow is not rendered useless with this approach. It can still be used to populate the tables dynamically during operation.

The first public P4 related document is the paper at SIGCOMM in 2014 [60–62]. Shortly after this publication, the first P4 specification, today known as $P4_{14}$, draft has been released. To the initial draft, authors from Barefoot, Google, Intel, Microsoft, the universities of Princeton and Stanford contributed. Since then many new companies, both networking device developers, engineers, and users, have joined the P4 community. All of the publications, including the language specifications and code examples, are available as open source.

With P4, network devices still should be built with ASICs, but they should now become more flexible. Recent research has shown that such flexibility can

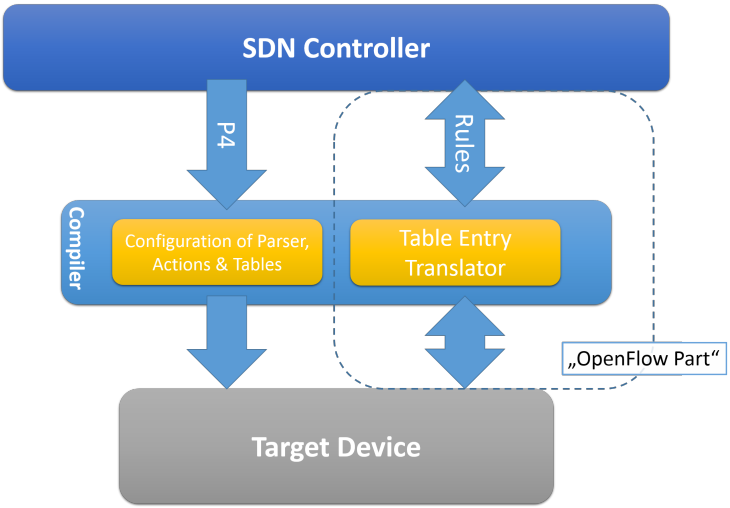


Figure 4.1: Depiction of the enhanced features and possibilities of P4 in comparison to OpenFlow.

be achieved with programmable ASICs. That this is not in contrast with the requirement for processing speed has been shown in previous publications [63].

In this chapter, the performance impact of modifying packets with P4-enabled hardware is analyzed. One of P4s promoted features is the possibility to modify packets at line rate. One proposed application by the P4 community is to add information, e.g. the processing latency of the packet itself, to packet headers during the processing in a switch. When continuously monitoring this information for the whole network, it is possible to identify processing bottlenecks without any additional controller involvement [64]. The contribution of this chapter is to analyze the performance impact of packet header modifications by adding and removing VLAN headers to a TCP packet stream. The analysis focuses on the

processing latency within the switch and the impact on the resulting network bandwidth.

This chapter is structured as follows. In Section 4.1, P4, including its architecture and current development directions, is introduced. Section 4.2 presents the measurement methodology, including a description of the testbed and the hardware devices used. The evaluation in Section 4.3 presents the measurements and discusses the implications of the results. Finally, Section 4.4 concludes this chapter and gives an outlook.

4.1 Background on P4

The P4 language specification consists of multiple pillars that, in conjuncture, allow for the promised flexibility whilst still maintaining line-rate speeds. This section, therefore, focuses on each of these fundamental parts. At first, the abstract P4 forwarding model is presented and explained on the example of PISA (Protocol-Independent Switch Architecture) that allows for the flexible realization of P4 programs in hardware. Second, the current state of P4, including related work is discussed. Finally, information on the VLAN protocol and its header is given as these headers will play a crucial role in the evaluation section of this chapter.

4.1.1 P4 Forwarding Model

An abstract representation of the P4 forwarding model is shown in Figure 4.2. In general, a packet has to run through the following parts of a P4 enabled switch (from left to right): A parser, which extracts header information from the packet. Here, the network administrator is able to freely program which parts of an incoming packet are recognized and in which order they are placed. After parsing the extracted information, it is processed in the programmable match-action tables which define what to do with which part of the packet and in which order. Furthermore, the egress port(s) is/are determined here. Finally, for the egress

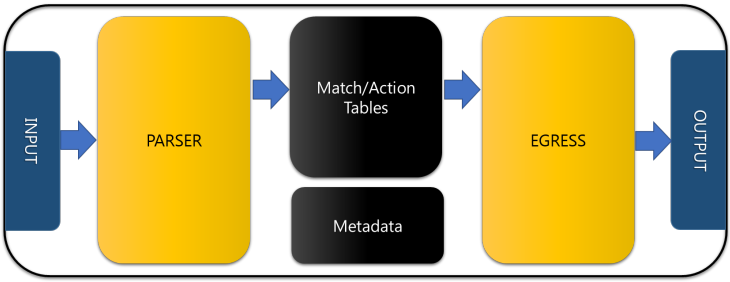


Figure 4.2: PISA architecture: abstract forwarding model.

queues it is possible to declare how an output packet will be structured before it is put on the wire. Additionally, a switch stores the so called packet metadata with the packet between ingress and egress. Internally, they are treated just as extracted header information. This data contains additional information, e.g. on the ingress port, the enqueuing time within the switch, or the queue length at packet arrival. This information can be used during the processing of the packet.

In order to start up operation of a P4 switch the following steps have to be completed. At first, the P4 device-independent program defining the headers including their fields, their order, and possible actions to take on them, has to be written. Second, a device-specific compiler transforms the code into a firmware. This firmware is then loaded onto the device. Finally, based on the defined tables with their matches and actions, entries can now be added to the table. This can be done by hand with static entries, or, similar to the OpenFlow operation, on the fly from a central controller. Now the switch is ready for operation.

For example, a simple Layer-2 switch would therefore have a parser that is able to extract Ethernet header fields. The match/action tables would most likely be filled with MAC-to-port entries, e.g. destination MAC `de:ad:be:ef:ba:11` to egress port 4. The corresponding code defining the structure of the Ethernet header can be found in Listing 4.1. In P4 each defined header is required

Listing 4.1: Ethernet header definition in P4

```
1   header_type eth_hdr {
2     fields {
3         dst : 48;
4         src : 48;
5         etype : 16;
6     }
7 }
```

to have a `fields` attribute that contains an ordered list of header fields including their length in bits. These fields are unsigned, and their position within a whole packet is determined by adding the width of all previous fields, including those of preceding headers. Here, for the case of the Ethernet header, at first the destination MAC address with 48 bits is defined and is followed by 48 bits of the source MAC address. The EtherType field with 16 bits concludes the header. Listing 4.2 shows two possible actions that could be applied to an Ethernet packet. Lines 1-3 show a simple drop action that could be used as a default action in an environment where all MAC addresses are supposed to be known. In the lines 5-7 a forwarding action is defined. Applying this action to packets, sets the metadata field `egress_spec` to the value of `prt`. This value is read in the egress queue and transmits the packet on the specified output port. The action `fwd_and_add_label` is an example on how to add a VLAN header to an Ethernet packet. At first, the egress port is set. Afterwards, with the `add_header` call, a VLAN header, that needs to be defined as the Ethernet header example, is added. The following lines 15-18 with the `modify_field` command change field values of the VLAN header to those of the input parameters. Finally, in Line 19, the EtherType field is set to the corresponding VLAN value, indicating that the Ethernet header is followed by a VLAN header.

Besides the already introduced actions, called primitives, P4 allows to copy the contents of one field to another via `copy_field`, to pop a header including

Listing 4.2: Exemplary P4 actions

```
1  action drop_act() {
2      drop();
3  }
4
5  action fwd_only(prt) {
6      modify_field(standard_metadata.egress_spec, prt);
7  }
8
9  action fwd_and_add_label(prt, pcp, dei, vid) {
10     modify_field(standard_metadata.egress_spec, prt);
11     add_header(vlan);
12     modify_field(vlan.etype, eth.etype);
13     modify_field(vlan.pcp, pcp);
14     modify_field(vlan.dei, dei);
15     modify_field(vlan.vid, vid);
16     modify_field(eth.etype, ETHERTYPE_VLAN);
17 }
```

all of its fields via `remove_header`, to increment or decrement via `increment`, and to calculate checksums over fields, e.g. an IPv4 checksum, via `checksum`.

The filling of the match&action tables of P4 is strongly dependent on the used hardware. For example, the Netronome P4 cards which will be used for the experiments of this chapter use a JSON like file format that can be transmitted as a so-called `user config` during run time to the card. An example can be found in Listing 4.3. This user config sets the default rule, which matches every packet that is not matched by another rule, to the `fwd_and_add_label` action of Listing 4.2 and sets the parameters to the defined values.

4.1.2 Current State and Related Work

This section gives an overview on the general directions of P4-related research and later on, focuses on other research on the impact of processing latency on the networking performance

Listing 4.3: Exemplary P4 user config

```
{
    "tables": {
        "in_tbl": {
            "default_rule": {
                "action": {
                    "data": {
                        "dei": { "value": "0" },
                        "pcp": { "value": "1" },
                        "vid": { "value": "0x123" },
                        "prt": { "value": "p1" }
                    },
                    "type": "fwd_and_add_label"
                },
                "name": "default"
}}}}
```

With the introduction of $P4_{16}$ in May 2017 [65], a long-term evolution of $P4_{14}$, the architecture has been relaxed. The same basic building blocks and concepts still apply, but the semantics have been defined more formally. This renders the operation of the devices more convenient and the language architecture separation has been enforced. The fundamentals of P4 still remain the same and all devices that support $P4_{16}$ are also able to execute programs written in $P4_{14}$.

Although still in its infancy, the number of research directions is already quite all-embracing. Also, the number of P4-enabled devices is increasing rapidly.

With the introduction of the INT (In-band Network Telemetry) framework Kim *et al.* [64] presented a way to query, analyze, and detect switch-internal processing information such as queue size, and processing latency. With the help of these information, they showcase a scenario to detect congestion on a per switch level.

Beyond the scope of this paper, the development of the INT framework continued and is today the topic of one of the working groups of the P4 project. The current draft specification [66] states that the goal is to improve network

monitoring and management without the expensive intervention of the control plane. In an INT-enabled network environment, devices add "telemetry instructions" to the packets. These tell the switch what information to collect and to add to the packet data. Network devices are divided into two categories: traffic sources and sinks. INT traffic sources execute these instructions and add it to the packet, if applicable, INT traffic sinks retrieve this data and optionally refer them to a special data processing unit. This way, the packets "report" themselves the state of the network they "observed" while traversing the network.

P5 from Abhashkumar *et al.* presents a way of using knowledge about applications that are deployed in the network [67]. One of the benefits of this approach is that it is able to detect and remove inter-feature dependencies between tables. This increases the pipeline efficiency by improving the pipeline concurrency. Furthermore, if the network traffic of an application never uses some switches of the network topology, those switches do not require the installation of application-specific features on said switch which increases the free resources which, in turn, increases its efficiency. Experiments of the authors show a possible efficiency increase of up to 50%.

HyPer4 is an approach that tries to virtualize the programmable data plane in order to allow for the storing of multiple programs on the same P4 device [68]. The benefit of this technique is that multiple programs can be run in parallel or as hot-swappable snapshots. In contrast to plain P4, programs that are currently inactive are even modifiable without interrupting running ones. After evaluating the performance, the isolation and possible limitations of their proposal, the authors conclude that their HyPer4 is deployable on ASIC-based hardware.

One of the remaining gaps in the feature set of the P4 specifications is tackled by Jepsen *et al.* [69]. The authors aim to create stateful processing, or what is required to achieve it with the current P4 spec. An implementation of their approach is able to run the Linear Road benchmark [70] and exceeds the throughput of any prior work in that direction.

Another approach that tries to bring statefulness to the P4 data plane is presented by Luo *et al.* [71]. By piggybacking information to the traffic, they allow

for the migration of state information without updates from the control plane. Their first implemented prototype shows that this is already possible to realize.

Gelberger *et al.* discuss the influence of the flexibility brought by SDN on the performance [72]. Metrics to qualify and quantify this performance and include throughput, latency, and jitter which are compared between OpenFlow-enabled SDN and ProGFE [73], which is based on IETFs ForCES [74]. They are also interested in performance penalties whilst using more complex SDN functions. Their conclusion is that SDNs flexibility comes with the price of performance. In contrast to our research focus, this paper focuses on OpenFlow-enabled SDN without any P4-enabled enhancements.

Whippersnapper Dang *et al.* introduce a P4 language benchmark suite [75]. Rather than only targeting switching devices and their networking performance, Whippersnapper also takes the compilers into consideration. With the help of their synthetic approach, it evaluates the main components of the P4 language. The authors argue that their research is helping innovation in the P4 context. They also demonstrate the usefulness of their benchmark by comparing latencies of multiple header modifications between four P4 systems. Although also covering the impact of header modification on the processing latency, Dang *et al.* do not differentiate between single packets, bursts of packets or a packet stream. This is covered by our scenario selection.

4.1.3 Background on VLAN

The term Virtual Local Area Network (VLAN) describes a logical division of the physical network into multiple subsections. As these subsections are isolated from each other it is often used to secure crucial data from forbidden access. For example, a common VLAN application is to isolate the VoIP traffic that is routed through a LAN from the remaining data traffic.

When talking about VLAN technology today, most of the time the discussion is about the IEEE 802.1Q standard [76]. In order to enable VLAN in a network, hardware that supports this standard is required. This hardware then is able

Figure 4.3: VLAN header in the Ethernet frame.

to apply so called VLAN tags to incoming traffic based on certain rules. The rule-set includes simple mapping of input-port-to-vlan-tag or more dynamical mappings, e.g. based on MAC addresses or after user authentication. In most deployments the VLAN tag is removed (also 'popped') at the destination switch, as not all client network interfaces support VLAN tagging.

The VLAN tags of the IEEE 802.1Q standard consists of four data fields with a total length of 32 bits that are added to the Ethernet frame, as depicted in Figure 4.3. The `protocol id` uses two bytes and is set to the default value of 8100, indicating the usage of 802.1Q. The `pcp` and `dei` are indicators for the priority of this traffic and are 4 bit wide each. Finally, the `vid`, the VLAN identifier is 12 bit wide, allowing for the creation of $2^{12} - 2 = 4096 - 2 = 4094$ VLANs. The VLAN ids 0 and 4095 are reserved and cannot be used. Devices that are within the same VLAN id group can transmit data to each other but are unable to communicate with devices from another VLAN id.

Today, especially in data center deployments, where it makes sense to isolate the traffic of one consumer from the traffic of another consumer or even management traffic, the number of 4094 possible VLAN ids is often too small. Therefore, new approaches that enable for the utilization of more VLANs have been created. Two currently competing standards are VXLAN (Virtual Extensible LAN) and NVGRE (Network Virtualization Generic Routing Encapsulation). Both of these approaches have their own advantages and disadvantages. Behind both approaches stand different interest groups, namely Cisco and VMware for VXLAN and Microsoft, Intel, and Dell for NVGRE. Therefore, it is most likely

that the questions which standard will be enforced, will be a political one. Nevertheless, VXLAN looks to be the certain winner, as, in contrast to earlier years where Microsoft could dictate the protocols in use by their servers, simply the number of virtual machines running on VMware hosted servers are already in favor for VXLAN.

The VLAN header addition or removal will be used as an example for the P4 performance as its application is a very common usecase. Therefore, evaluating the P4 performance in this scenario can derive a valuable insight in the practicability of P4 in existing deployments.

4.2 Methodology and Testbed Setup

This section introduces the methodology used during the measurements in order to analyze the packet modification performance. At first, the testbed setup is presented. Second, the various scenarios are discussed.

4.2.1 Testbed Setup

The testbed, as depicted in Figure 4.4, consists of two inter-connected components: a Spirent traffic generator and a NIC by Netronome that is P4-enabled.

The Spirent traffic generator [77] is a hardware device that is, among other things, able to generate packets of well-known protocols with a rate of 10 GBit/s. In this testbed it is used to generate random data that is encapsulated into a random IPv4 packet.

The P4-enabled card is a Netronome Agilio CX with two 10 GBit/s interfaces [78]. According to its vendor, the card supports the full $P4_{14}$ language specification and has 2 GB DRAM for lookup tables, which allows tables with "*millions of entries*" [79].

The measurement procedure of the test will be as follows: The Spirent will generate packets and transmit them to the first port of the Netronome NIC. Depending on the scenario the packet does or does not include VLAN tag(s).

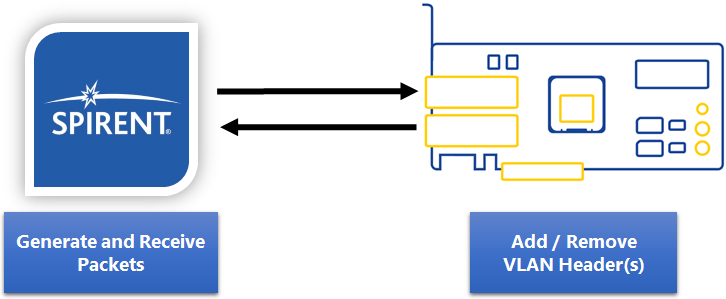


Figure 4.4: Testbed overview.

The Netronome NIC will then apply the defined action and forward the packets back to the Spirent via the second port. The Spirent receives the packets and calculates some basic statistics, such as average processing latency and number of transmitted and received packets including their length. These statistics are then saved to file and evaluated.

4.2.2 Scenarios

As the target of this evaluation is to measure the impact of the header modifications on the processing latency of a P4-enabled network device, a baseline of non-modifications has to be established. The only two actions that do not modify any header of a packet are forward and drop. As drop does not return any packets and, therefore, does not generate any valuable statistics at the Spirent, forward is the only option left. In order to compare the effects of adding or removing multiple headers or just a single VLAN header, the second baseline is to measure the addition and subtraction of a single VLAN header.

Furthermore, in order to add or remove multiple VLAN tags with P4, multiple options are possible. The special use case with multiple VLAN tags is also found in existing data centers. Figure 4.5 exemplarily depicts the possible options when adding two headers. Adding more tags or removing tags works analogously.

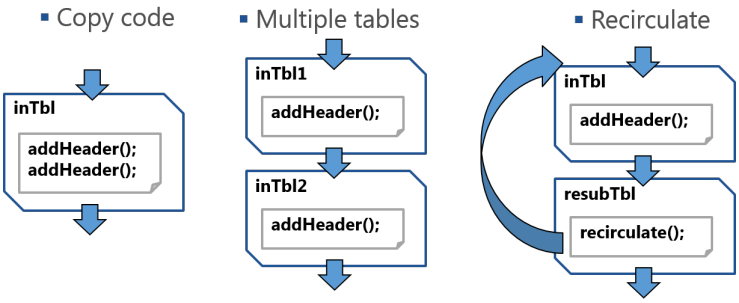


Figure 4.5: Options to add multiple tags to a packet with P4.

A very simple, but viable option is to simply copy and paste the statements of the defined action. Instead of simply executing the `addHeader()`-action once, it is now executed twice in order to add two VLAN tags.

Another option is to define multiple tables that each execute one `addHeader()`-action. According to the P4 specifications, functional identical tables that only differ in their name are possible. The additional table only costs memory for the table declaration and not for the action declarations as multiple tables can be associated with the same action(s).

The third, most complex operation is to 'loop' the packet. By design, P4 does not allow looping constructs as they are undesirable in hardware pipelines. This is also a decision to have a fixed upper bound of the processing time of a packet within a network device.

For this at least two tables are required. The first one executes `addHeader()` whereas the second one executes `recirculate()` which takes the extracted packet header information and puts it again at the beginning of the packet processing pipeline, namely the parser. This primitive accepts a list of fields that specifies which field should keep their values during the reparsing of the header fields. Here, the packet metadata comes into play: it allows for the tracking of the exit condition of this 'loop'. This 'looping', however, is expected to be as-

sociated with a performance penalty. As a rule of thumb, recirculating a packet n times cuts the throughput by a factor of n. For example, the ruleset of this variant would be to create the following matching entries in the tables:

1. inTbl: add VLAN header to each incoming packet from port #1, add recircFlag and set it to 1, and set output port to #2,
2. resubTbl: if recircFlag equals 1 and output port is #2, recirculate to inTbl,
3. inTbl: if recircFlag equals 1 and output port is #2, add VLAN header and set recircFlag to 2,
4. resubTbl: if recircFlag equals 2 and output port is #2, do nothing,
5. enqueue packet on egress queue of port #2.

Despite the capabilities of P4 to enable bursty traffic, the behavior in full bandwidth traffic is also crucial. Therefore, a second series of scenarios will be investigated. Using the introduced testbed, the Spirent traffic generator will generate 10 GBit/s streams of data and the Netronome NIC will add or remove VLAN headers. Evaluated will be the processing latency, the number of lost packets, and the bandwidth received by the Spirent from the P4 NIC.

4.3 Evaluation

After setting up the testbed and specifying the scenarios, the evaluation of the measurements is done. At first the bursty scenario is evaluated, then, the full bandwidth scenario is analyzed.

Figure 4.6 shows the processing latencies for multiple scenarios of adding or removing VLAN tags and their corresponding average processing latencies. For all scenarios the color coding is the same: blue bars depict the average value of a burst-size of 1 packet, red bars of 100 packets, and yellow bars 1000 packets. This means, that the depicted value of the yellow bar is the average from 1000 repetitions of an average processing latency of 1000 packets, for example. Due to the limited data recording features of the Spirent traffic generator, no extended

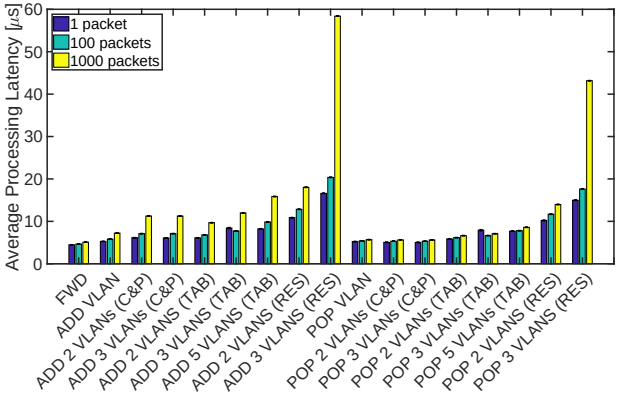


Figure 4.6: Average processing latency when adding and removing multiple VLAN headers.

statistical evaluation of the measurement runs is possible, as, for example, a measurement run of with a burst size of 1000 packets only provides three values: average, minimum and maximum of the desired metric. The shown scenarios are those that have been introduced earlier in Section 4.2. At first the baseline with plain forwarding is shown. Afterwards, all possible combinations for adding 1, 2, and 3 tags are presented. Finally, the results for removing (here: popping) 1, 2, and 3 VLAN headers are given. For all results 95% confidence intervals are shown that have been calculated over the data of 1000 measurement runs per configuration. The inter frame gap for the burst scenarios has been set to 120 ns.

With an average latency of 4.48, 4.66, and 5.09 µs only forwarding a packet is the fastest processing option within the Netronome P4 card. Already noticeable is the presence of a difference by 14 % between processing a single packet and a burst of 1000 packets.

Adding one VLAN header to a single packet results in a processing latency of 5.36 µs. When processing a burst of 100 packets the latency increases to 5.83 µs,

for 1000 packets to 7.26 µs, respectively. Removing a single VLAN header shows comparable results with a processing latency of 5.18 µs, and increases to 5.41 µs and 5.71 µs. Here, already, differences between adding and removing of the tags are evident. Especially for the case of the burst size set to 1000, the deviation is 1.55 µs, with the benefit being on the removing side.

For the scenario of simply copy-pasting the statements the number of modified headers seems not to have a large impact on processing performance. Whilst adding two headers, the latency ranges from 6.15, to 7.11 and 11.26 µs, with the difference between the values of two and three headers being less than 0.10 µs. For removing the header, a similar behavior is observable. Here, the values range from 5.05, over 5.32, to 5.63 µs. Again, removing the headers is faster than adding them.

In the multi-table scenario, the previously observed trends continue. Adding two headers requires 6.12 µs of processing for one single packet, and 6.82 µs for 100 packets, and, finally, 9.70 µs for bursts of 1000 packets. The margins when adding three headers instead of two are a bit taller than before: for one single packet 8.42 µs of processing time pass, whereas 100 packets are processed within 7.75 µs. For 1000-packet-bursts the processing time again rises to 12 µs. 5.88, 6.17, and 6.64 µs are the latencies for removing two headers, 7.92, 6.64, and 7.01 µs are the processing times for three headers, accordingly. The results for removing two and three headers confirm the general tendencies of the previous results. But, this time, the results for adding and removing three headers from a single packet are larger than those for the other configurations. This is a very odd observation that is also found in the results for removing two headers.

Using the looping-construct with recirculate, the longest processing times can be found for each category. Adding two headers to a single packet on average takes about 10.88 µs of processing time. For the case of 100-packet burst, the time increases to 12.84 µs, and is surpassed by the value for a burst with 1000 packets of 18.03 µs. Adding three headers to a packet increases the times even further: For a single packet 16.6, for a burst of 100 packets 20.37, and for 1000-packet-bursts 58.4 µs are required in average for processing within in the

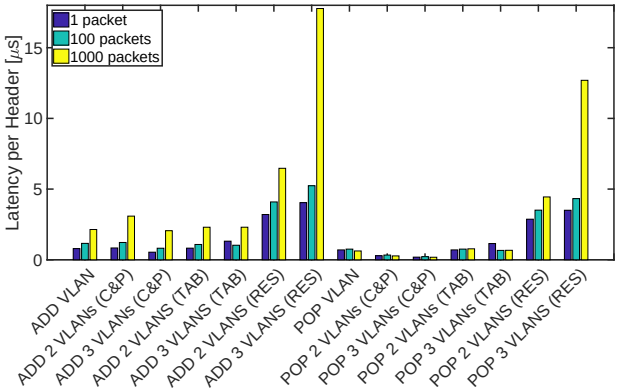


Figure 4.7: Average processing latency per VLAN header.

P4 NIC. Removing the headers ranges from 10.22 over 11.68, to 13.98 µs for the case of two headers, and from 14.99, over 17.64, to 43.17 µs for the case of three headers. These values, especially for 1000-packet-burst of adding and removing three headers with the recirculate method, show a large deviation from all other scenarios.

The reason for this behavior could be manifold. The increased processing time is not unexpected per se for this scenario. As introduced in Section 4.2, this looping workaround with the *recirculate*-command is one of the most ineffective ones and no guarantees about the processing performance can be given. What instead is astonishing is the jump in processing time between single to batch-processing by a factor of almost 4. For all other measurement scenarios, the factor was less than 2.

Here, the processing latency per added VLAN header is plotted for all previously mentioned scenarios. The results are calculated by taking the processing latency of one scenario, e.g. adding two VLANs via multiple tables, subtracting

the latency of the forward only baseline and divide it by the number of headers modified. This is expressed in Equation 4.1.

$$\begin{aligned} \textit{latency per header}_i &= \frac{latency_i - latency_{fwd}}{count(\#headers)} \quad , \\ \text{where } i &:= \{\text{All Scenarios}\} \backslash \{FWD\} \quad . \end{aligned} \tag{4.1}$$

Throughout the scenarios and their burst configurations, except for the looping variant, the latencies are almost constant per added or removed header. Adding a header in the single-packet-burst scenarios takes between 0.53 and 1.32 µs, for 100-packet-bursts between 0.84 and 1.16 µs, and for 1000-packet-bursts between 2.01 and 3.01 µs. Removing a header is, as previously observed, less costly. In the single packet scenario the latency varies from 0.19 to 1.15 µs, in the 100 packet scenario from 0.22 to 0.75 µs, and in the 1000 packet scenario from 0.18 to 0.77 µs. The results for looping variant have been excluded from these previous results, as they are, again, drastically slower than all the other results. The factors vary between 3 and 9 times the latencies of the other scenarios for adding heading headers, for removing headers it varies between 2.5 and 50.

Figure 4.8 shows the average processing latency for the configuration with full 10 GBit/s. The x-axis shows the various available options to add or remove one, two, or three VLAN headers to and from a packet stream. The results depicted in this figure show only small deviations from the previously observed behavior in Figure 4.6 for the configuration using packet bursts. Overall, the difference between each configuration option, has become smaller and the overall processing latency has increased. For example, the result for forwarding a packet without any modification remains at values around 5 µs. Only the high spikes that have been observable for the looping variant with 1000 packets do not appear in this configuration.

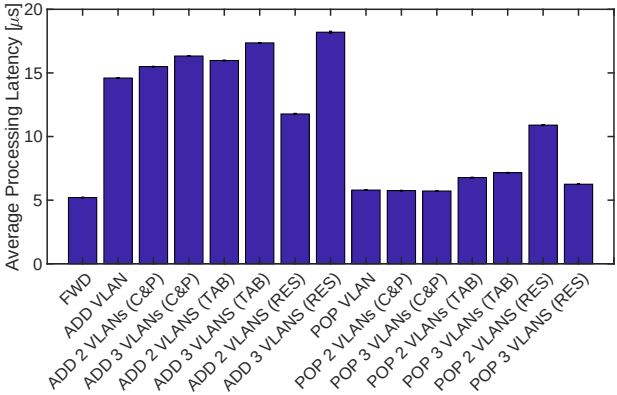


Figure 4.8: Average processing latency.

Figure 4.9 shows the average receive bandwidth at the Spirent traffic generator. For most of the scenarios the P4 card from Netronome is able to keep the configured transmit bandwidth of 10 GBit/s. For the scenarios removing the VLAN header(s) the bandwidth shows small deviations. These are caused by the fact that by removing a VLAN header, the overall size of packet is reduced slightly, namely 4 Byte per VLAN header. This results in a slightly off bandwidth on the receiver side. Only for the looping variants huge deviations can be found. For adding two or three VLAN headers whilst using the recirculate command reduces the bandwidth to 3.1 and 2.4 GBit/s. Removing two VLAN headers shows comparable results with a reduction of the receive bandwidth to 2.9 GBit/s. This general behavior has been expected, as it has already been introduced that the recirculate command comes with a performance penalty.

Only the result for removing three VLAN headers in this configuration is odd, as the bandwidth rises again to 6.7 GBit/s. Investigating this scenario further reveals that the P4 card is unable to cope with this scenario properly. Only half of all the packets received at the sink have all the VLAN header removed, for

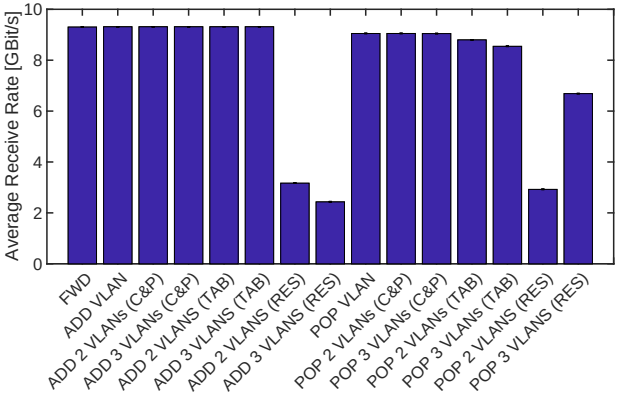


Figure 4.9: Average receive rate.

some only one or two headers are removed, for some none. This can only mean that the internals of the card work not as expected under heavy load. Presumably whilst the card is busy removing the header of some packets, more packets arrive, and are buffered. As soon as this buffer overflows, the packets are not simply dropped but somehow circumvent the matching instructions and are simply forwarded to the destination port without the proper count of recirculations.

In Figure 4.10 the average rate of packets lost per scenario is depicted on the y-axis. Throughout the measurements a low packet loss rate between 0.01% and 0.04% can be noticed. Only the packet loss rate of the recirculation results show a deviating behavior. Here, the packet loss rate is between 0.16 and 0.61%. This behavior is in line with the results for the receive bandwidth.

Altogether, the presented results for the full bandwidth scenario confirm the results for the scenarios with packet bursts. Overall, three general observation can be made from these processing latency results. First, adding or removing VLAN headers introduces latency to the processing in the data plane as the baseline scenario in which the packets are only forwarded from one port to the other

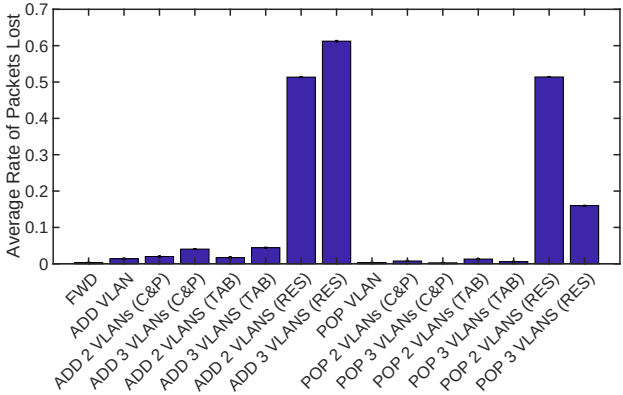


Figure 4.10: Average packet loss rate.

shows the lowest latency results. Second, the introduced latency is, except for the looping-variant via recirculate, independent of the method used. The results depicted in Figure 4.7 confirm this assumption. Third, the recirculate/resubmit command introduces a lot of overhead. Therefore, the usage of this command should only take place for special, time-insensitive use-cases.

4.4 Lessons Learned

P4 is a new and upcoming evolution of the whole SDN movement. It allows for a top-down design where the network operator is able to instruct the hardware how to behave and how to switch packets. Put into concrete terms, this means that a network operator is now also able to program the parsing unit of a switch, whilst still maintaining line rate bandwidths. Furthermore, P4 claims to be able to modify any of these programmable header fields at line rate. Additionally, it is the goal of P4 to create a target-independent solution that is suitable for both hard- and software switches alike.

As it is always the case with new technology, the applicability of this in production deployments requires research of the performance, the reliability, and the availability. The goal of this chapter is to analyze the impact of header modifications on the processing performance at the example of the VLAN header. As the action to add or remove a VLAN header is a very common use-case for networking devices in data centers the results can be used as an indicator whether the deployment of P4 is already useful and, within in some limitations, what to take in mind when designing a P4 program. The analysis focuses on the processing latency within the switch and the impact on the bandwidth of modified TCP packet streams.

Our performance investigations show that the seemingly simple task of adding or removing VLAN headers to TCP packets already contains several pitfalls. Already adding or removing a single VLAN header introduces delay to processing of a packet stream within a device. Except when using the `recirculate` method, the introduced latency is mostly independent of the method used for the header modifications. Finally, the `recirculate` command is, as expected, rather complex and introduces a lot of delay. Additionally, the packet loss probability is at the highest levels for this action. Therefore, the usage of this command should be limited to low-bandwidth and time-insensitive packet streams.

Dang *et al.* already evaluated the impact of header modifications on the processing performance in more general scenarios [75]. Their overall observation that more adds/removes result in a higher processing latency are in line with our observed behavior. Nevertheless, they observe that removing a header is slower that adding a header, which is in contradiction to our results. The authors explain this behavior with a non-optimization in their P4 compiler, which adds additional stages for each remove action. Additionally, compared to Dang's work, in this thesis more actions and scenarios are evaluated.

Altogether, the development of SDN towards P4 is a necessary step in the evolution of software-defined networking. Nevertheless, P4 is not a jack of all trades, and, therefore, thoughtful development and deployment of P4 programs

is required as well as comprehensive performance evaluations. The evergrowing community is driving the development of P4 into new directions and fills gaps. Among other working groups, especially the INT framework [64] offers interesting opportunities. Through its in-place addition of switching information, such as queue length or processing time per network device, it is possible to have even deeper insights into the performance and status of the network than before. Nevertheless, as this approach is also a novel one, the performance implications of this technique needs to be analyzed and considered - especially when it its deployed outside the safe world of a testbed.

5 Conclusion

With its simple principle of decoupling the control plane in networking devices from the data layer, Software-defined Networking (SDN) has brought innovations back into networking. After almost one decade of SDN, it is fair to say that this principle has led to a radical shift in networking thinking. New business models are opening up, and with SDN, flexibility is entering the data networks. As the control plane can now be centralized in software, the so-called SDN controller, network operators are able to steer the entire network in a more flexible and agile way than before. Yesterday's legacy networks required each device on the network to be accessed individually and according to the vendor and model in its own way. This has been a major challenge, especially for large topologies that have evolved over time, resulting in high operational costs. Thanks to the paradigms of SDN, operators are now able to change the policies of an entire global network by changing only a few lines in their controller configuration. Thanks to its paradigms, SDN enables vendor independence and, therefore, allows network administration the management of the networking without having to think about the capabilities of the underlying network hardware. However, as with any emerging technology, SDN still has to prove that it is already production ready and able to provide carrier-grade reliability and availability.

In summary, this thesis focused on reliability and availability challenges of SDN deployments in production environments by conducting a performance analysis and investigating currently common SDN error detection mechanisms. Finally, P4, the latest and most recent evolution of the SDN technology was investigated by evaluating the influence of the P4 program implementation on the data plane processing performance.

In SDN packet processing is normally done via a matching of incoming packets against the so-called flow table. This flow table can be filled in two different ways. The pro-active way assumes that most or all traffic is known at the start-up of the switch and flow table rules are installed before the first packet of a flow arrives at a switch. This operation mode is most likely used in environments where most sources of traffic are known, e.g. in a data center. In reactive mode, the controller only responds to incoming traffic because it is assumed to be unknown before start. If no match can be found, the packet is usually forwarded via a notification message to the controller which, in turn, calculates the further actions for this packet stream. The decision is then sent to the switch, e.g. by installing a new rule, containing the match and the associated action, into the flow table. As the flow table space of current generation hardware SDN switches is very limited in general, these rules are annotated with a time-out value. If a flow is inactive for a certain amount of time, the flow entry is removed from the table and, thus, frees the space for other rules. This creates signaling overhead that poses a new challenge in SDN-enabled networks. Especially with switches with small flow tables much signalization is generated, which corresponds to a performance metric of SDN. This results in a trade-off between stored flows in the table and signaling traffic, which is subject of Chapter 2.

With the means of an analytic model the relationship between the characteristics of a data plane traffic, the flow table occupancy, and the control plane signaling is evaluated. As both, flow table occupancy and signaling rate between switch and controller, can become a bottleneck in the performance of the whole network, a trade-off has to be found. Therefore, a model for a single flow is presented in order to understand its impact on the mentioned metrics. From there, an $M/M/\infty$ queuing system is adapted in order to enhance our model for multiple simultaneous flows. Afterwards, our results have been validated by the means of discrete-event simulations and measurements in a testbed. Furthermore, it became evident that traffic parameters, such as the mean inter-arrival time and its coefficient of variation c_A have an impact on both mentioned metrics. On the one hand, the choice of the time-out has a relatively small impact

for very bursty applications with high c_A as they are more likely to have long inter-arrival times between two traffic bursts. On the other hand, a large optimization potential for applications with a small c_A has been discovered. With a small time-out for these applications, their flow time out and free the flow table space sooner, and thus, more applications within the network can ben processed.

With the time-out value for the flow table entries a powerful tool is given to each network administrator. Setting this value right is one of the challenges of SDN deployments and is best done on a per-application level. As the characteristics of each application data flow may vary enormously, it is impractical to set a network-wide static default value. With such a choice, too much unnecessary signaling load would be generated, and, thus, the performance of the whole network would suffer. Nevertheless, this approach would open the new challenge of a reliable application detection based on flow traffic characteristics.

In Chapter 3, the detection capabilities of the common SDN controller ONOS for challenging network conditions, e.g. packet loss in the data plane, is evaluated. After theoretically analyzing the built-in mechanisms and verifying these results through measurements in a testbed, it became evident that this controller lacks detection performance under these conditions. The presented results show that the implemented algorithms are unreliable and therefore unable to keep the carrier-grade reliability claims of the ONOS developers. For example, it can take more than one minute to detect packet loss values of 50% in the data plane.

Our approach is to develop and implement an active probing application that uses the features of the Northbound API of ONOS. With the means of generated probing packets that are sent through the network, the application is able to determine the current metrics for links in the data plane, e.g. packet loss or one way delay. In order to further increase the detection performance of this application, also a self-adapting mechanism is introduced that adjusts the configuration of the probing mechanism and, therefore, hardens it against further hazardous network conditions such as jitter. Our evaluations by measurements in a testbed have shown that the detection performance is increased without generating a noticeable overhead on side of the controller resources, namely

CPU and RAM. For example, with the application enabled it is already possible to detect packet loss values of 5%, while the native detection mechanism is only able to detect packet loss values beyond 40%. Furthermore, as soon as the network environment conditions relax and become less challenging, the algorithm is able to adopt its probing strategy, and thus reduces the CPU utilization.

Finally, Chapter 4 of this monograph focuses on a recent evolution of SDN: P4. It enables the programming of processing mechanisms of hardware networking devices and, therefore, offers new flexibilities for network operators. The question whether this new technology is already ready for production deployments is addressed by performance measurements of a common data center task: adding and removing of VLAN headers. This simple use-case already showcases that there are many pitfalls when designing and implementing P4 programs. The shown approaches of adding multiple VLAN headers present a huge gap between the fastest and slowest options. For instance, for a traffic burst of 1000 packets the fastest option is able to add 3 VLAN headers within 11.26 µs, while the slowest option using a looping-construct takes more than 58.4 µs. Keeping these results in mind, two statements about the P4 technology can be made: At first, it is already usable in deployments, and, but secondly only with the knowledge of how to realize the desired actions.

Using the results of this monograph, several aspects of the production readiness of SDN deployments can be evaluated, analyzed, and improved. As introduced, there are still many pitfalls for a reliable network. With SDN, network operators have many opportunities and parameters to adjust the functionality, performance, and availability of their networks. As SDN is still evolving and developing, e.g. shown by the introduction of P4, the interest of the networking community all over the world is undaunted. Short-development cycles, being one of the advantages of SDN, contribute to the realization of new approaches, insights, and features. Especially the P4 technology will bring a lot of new inputs to this ever-changing field.

Bibliography and References

Bibliography of the Author

Journal Papers

[1] C. Metter, M. Seufert, F. Wamser, T. Zinner, and P. Tran-Gia, "Analytical Model for SDN Signaling Traffic and Flow Table Occupancy and Its Application for Various Types of Traffic", *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 603–615, Sep. 2017, ISSN: 1932-4537. DOI: 10.1109/TNSM.2017.2714758.

Conference Papers

[2] M. Jarschel, C. Metter, T. Zinner, S. Gebert, and P. Tran-Gia, "OFCProbe: A platform-independent tool for OpenFlow controller analysis", in *5th IEEE International Conference on Communications and Electronics (ICCE 2014)*, Da Nang, Vietnam, Jul. 2014, pp. 182–187. DOI: 10.1109/CCE.2014.6916700.

[3] C. Metter, S. Gebert, S. Lange, T. Zinner, P. Tran-Gia, and M. Jarschel, "Investigating the impact of network topology on the processing times of SDN controllers", in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, Ottawa, Canada, May 2015, pp. 1214–1219. DOI: 10.1109/INM.2015.7140469.

[4] L. Iffländer, F. Wamser, C. Metter, P. Tran-Gia, and S. Kounev, "Performance Assessment of Cloud Migrations from Network and Application Point of View", in *Proceedings of 9th EAI International Conference on Mobile Networks and Management (MONAMI 2017)*, Melbourne, Australia, Dec. 2017.

[5] C. Metter, M. Seufert, F. Wamser, T. Zinner, and P. Tran-Gia, "Analytic Model for SDN Controller Traffic and Switch Table Occupancy", in *12th International Conference on Network and Service Management (CNSM)*, Montreal, Canada, Oct. 2016, pp. 109–117. DOI: 10.1109/CNSM.2016.7818406.

[6] C. Metter, V. Burger, Z. Hu, K. Pei, and F. Wamser, "Evaluation of the Detection Capabilities of the ONOS SDN Controller", in *7th IEEE International Conference on Communications and Electronics (IEEE ICCE 2018)*, IEEE, Hue, Vietnam, 2018, pp. 1–6.

[7] ——, "Towards an Active Probing Extension for the ONOS SDN Controller", in *2018 28th International Telecommunication Networks and Applications Conference (ITNAC) (ITNAC 2018)*, Sydney, Australia, Nov. 2018.

General References

[8] M. Jarschel, T. Zinner, T. Hossfeld, P. Tran-Gia, and W. Kellerer, "Interfaces, attributes, and use cases: A compass for SDN", *Communications Magazine, IEEE*, vol. 52, no. 6, pp. 210–217, Jun. 2014, ISSN: 0163-6804. DOI: 10.1109/MCOM.2014.6829966.

[9] B. A. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks", *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 3, pp. 1617–1634, 2014.

[10] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[11] C. Schwartz, T. Hossfeld, F. Lehrieder, and P. Tran-Gia, "Angry Apps: The Impact of Network Timer Selection on Power Consumption, Signalling Load, and Web QoE", *Journal of Computer Networks and Communications*, 2013.

[12] ON.Lab, *ONOS - A new carrier-grade SDN network operating system designed for high availability, performance, scale-out.* https://onosproject.org/, [Online; accessed 12.11.2018], 2018.

[13] OpenFlow v1.3 Administrator Guide, *HP Switch Software*, 2015.

[14] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and Performance Evaluation of an Openflow Architecture", in *Proceedings of the 23rd International Teletraffic Congress*, International Teletraffic Congress, 2011.

[15] K. Mahmood, A. Chilwan, O. N. Østerbø, and M. Jarschel, "On The Modeling of OpenFlow-based SDNs: The Single Node Case", *CoRR*, vol. abs/1411.4733, 2014. arXiv: 1411.4733. [Online]. Available: http://arxiv.org/abs/1411.4733.

[16] K. Mahmood, A. Chilwan, O. Østerbø, and M. Jarschel, "Modelling of OpenFlow-based software-defined networks: the multiple node case", *IET Networks*, vol. 4, no. 5, pp. 278–284, 2015.

[17] B. Xiong, K. Yang, J. Zhao, W. Li, and K. Li, "Performance evaluation of OpenFlow-based software-defined networks based on queueing model", *Computer Networks*, vol. 102, pp. 172–185, 2016.

[18] S. Azodolmolky, R. Nejabati, M. Pazouki, P. Wieder, R. Yahyapour, and D. Simeonidou, "An analytical model for software defined networking: A network calculus-based approach", in *2013 IEEE Global Communications Conference (GLOBECOM)*, IEEE, 2013, pp. 1397–1402.

[19] A. G. Osgouei, A. K. Koohanestani, H. Saidi, and A. Fanian, "Analytical performance model of virtualized SDNs using network calculus", in *2015 23rd Iranian Conference on Electrical Engineering*, IEEE, 2015, pp. 770–774.

[20] C. Lin, C. Wu, M. Huang, Z. Wen, and Q. Zheng, "Performance evaluation for SDN deployment: an approach based on stochastic network calculus", *China Communications*, vol. 13, no. Supplement, pp. 98–106, 2016.

[21] W. Miao, G. Min, Y. Wu, H. Wang, and J. Hu, "Performance Modelling and Analysis of Software-Defined Networking under Bursty Multimedia Traffic", *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 12, no. 5s, p. 77, 2016.

[22] N. Beigi-Mohammadi, H. Khazaei, M. Shtern, C. Barna, and M. Litoiu, "On Efficiency and Scalability of Software-Defined Infrastructure for Adaptive Applications", in *2016 IEEE International Conference on Autonomic Computing (ICAC)*, Jul. 2016, pp. 25–34. DOI: 10.1109/ICAC.2016.39.

[23] Y. Liu, Y. Li, Y. Wang, Y. Zhang, and J. Yuan, "On the resource trade-off of flow update in software-defined networks", *IEEE Communications Magazine*, vol. 54, no. 6, pp. 88–93, 2016.

[24] A. Zarek, Y. Ganjali, and D. Lie, "Openflow timeouts demystified", *Univ. of Toronto, Toronto, Ontario, Canada*, 2012.

[25] T. Kim, K. Lee, J. Lee, S. Park, Y.-H. Kim, and B. Lee, "A Dynamic Timeout Control Algorithm in Software Defined Networks", *International Journal of Future Computer and Communication*, vol. 3, no. 5, p. 331, 2014.

[26] H. Zhu, H. Fan, X. Luo, and Y. Jin, "Intelligent timeout master: Dynamic timeout for SDN-based data centers", in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 734–737. DOI: 10.1109/INM.2015.7140363.

[27] E. Kim, S. Lee, Y. Choi, M. Shin, and H. Kim, "A flow entry management scheme for reducing controller overhead", in *16th International Conference on Advanced Communication Technology*, Feb. 2014, pp. 754–757. DOI: 10.1109/ICACT.2014.6779063.

[28] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, "Rule-Caching algorithms for Software-Defined Networks", Citeseer, Tech. Rep., 2014.

[29] M. Kuźniar, P. Perešíni, and D. Kostić, "What You Need to Know About SDN Flow Tables", in *Passive and Active Measurement: 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings*, J. Mirkovic and Y. Liu, Eds. Cham: Springer International Publishing, 2015, pp. 347–359, ISBN: 978-3-319-15509-8. DOI: 10.1007/978-3-319-15509-8_26. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-15509-8_26.

[30] F. Ciucu and J. Schmitt, "Perspectives on network calculus: no free lunch, but still good value", in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, ACM, 2012, pp. 311–322.

[31] A. Feldmann, A. C. Gilbert, and W. Willinger, "Data networks as cascades: Investigating the multifractal nature of Internet WAN traffic", in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 28, 1998, pp. 42–55.

[32] A. Feldmann, "Characteristics of TCP connection arrivals", *Self-similar network traffic and performance evaluation*, pp. 367–397, 2000.

[33] H. Gold and P. Tran-Gia, "Performance analysis of a batch service queue arising out of manufacturing system modelling", *Queueing Systems*, vol. 14, 1993.

[34] T. Karagiannis, M. Molle, and M. Faloutsos, "Long-range dependence ten years of Internet traffic modeling", *Internet Computing, IEEE*, vol. 8, no. 5, pp. 57–64, 2004.

[35] A. T. Andersen and B. F. Nielsen, "A Markovian approach for modeling packet traffic with long-range dependence", *IEEE Journal on Selected Areas in Communications*, vol. 16, no. 5, pp. 719–732, Jun. 1998, ISSN: 0733-8716. DOI: 10.1109/49.700908.

[36] N. Duffield and W. Whitt, "A source traffic model and its transient analysis for network control", *Stochastic Models*, vol. 14, no. 1-2, pp. 51–78, 1998.

[37] K. Sriram and W. Whitt, "Characterizing Superposition Arrival Processes in Packet Multiplexers for Voice and Data", *IEEE Journal on Selected Areas in Communications*, vol. 4, no. 6, pp. 833–846, Sep. 1986, ISSN: 0733-8716. DOI: 10.1109/JSAC.1986.1146402.

[38] OpenFlow Switch Specification, *Version 1.4.0*, 2013.

[39] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[40] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "DevoFlow: scaling flow management for high-performance networks", in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 41, 2011, pp. 254–265.

[41] S. Gebert, R. Pries, D. Schlosser, and K. Heck, "Internet Access Traffic Measurement and Analysis", in *Traffic Monitoring and Analysis (TMA) workshop co-located with the 13th Passive and Active Measurement (PAM) conference*, Vienna, Austria, Mar. 2012.

[42] Google, *Google Play Store*, https://play.google.com/store, [Online; accessed 12.11.2018], 2018.

[43] MathWorks, *Matlab - The language of technical computing*, http://www.mathworks.de/products/matlab/index.html, [Online; accessed 12.11.2018], 2018.

[44] Lantz, Bob and O'Connor, Brian, *Mininet - An instant virtual network on your Laptop (or other PC)*, http://mininet.org/, [Online; accessed 12.11.2018], 2018.

[45] Linux Foundation, *Open vSwitch*, http://openvswitch.org/, [Online; accessed 12.11.2018], 2018.

[46] The Tcpdump team, *tcpdump, a powerful command-line packet analyzer*, http://www.tcpdump.org/, [Online; accessed 12.11.2018], 2018.

[47] OpenFlow Switch Specification, *Version 1.3.0*, 2012.

[48] A. Vishnoi, R. Poddar, V. Mann, and S. Bhattacharya, "Effective switch memory management in openflow networks", in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ACM, 2014, pp. 177–188.

[49] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, *et al.*, "B4: Experience with a globally-deployed software defined WAN", *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 3–14, 2013.

[50] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, *et al.*, "ONOS: towards an open, distributed SDN OS", in *Proceedings of the third workshop on Hot topics in software defined networking*, ACM, 2014, pp. 1–6.

[51] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, "OpenFlow: Meeting carrier-grade recovery requirements", *Computer Communications*, vol. 36, no. 6, pp. 656–665, 2013.

[52] B. Heller, "OpenFlow Switch Specification, Version 1.0.0", OpenFlow Consortium, Tech. Rep., 2009.

[53] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "Fattire: Declarative fault tolerance for software-defined networks", in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, ACM, 2013, pp. 109–114.

[54] L. Quan, J. Heidemann, and Y. Pradkin, "Trinocular: Understanding internet reliability through adaptive probing", in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 43, 2013, pp. 255–266.

[55] N. L. Van Adrichem, B. J. Van Asten, and F. A. Kuipers, "Fast recovery in software-defined networks", in *Software Defined Networks (EWSDN), 2014 Third European Workshop on*, IEEE, 2014, pp. 61–66.

[56] D. Katz and D. Ward, "Bidirectional forwarding detection (bfd)", RFC Editor, RFC 5880, Jun. 2010.

[57] R. Santos, H. Ogawa, G. Khanh Tran, K. Sakaguchi, and A. Kassler, "Turning the knobs on OpenFlow-based resiliency in mmWave small cell meshed networks", in *Globecom Workshops (GC Wkshps), 2017 IEEE : 5G Testbed*, 2017, ISBN: 978-1-5386-3920-7. DOI: 10.1109/GLOCOMW.2017.8269214.

[58] J. Kempf, E. Bellagamba, A. Kern, D. Jocha, A. Takacs, and P. Sköldström, "Scalable fault management for OpenFlow", in *2012 IEEE International Conference on Communications (ICC)*, Jun. 2012, pp. 6606–6610. DOI: 10.1109/ICC.2012.6364688.

[59] IEEE Computer Society, "Station and Media Access Control Connectivity Discovery", *IEEE Std. 802.1ab*, vol. IEEE Standard for Local and Metropolitan Area Networks, pp. i–204, 2009.

[60] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, *et al.*, "P4: Programming protocol-independent packet processors", *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[61] R. Ozdag, "Intel® Ethernet Switch FM6000 Series-Software Defined Networking", p. 5, 2012.

[62] G. Pongrácz, L. Molnár, Z. L. Kis, and Z. Turányi, "Cheap silicon: a myth or reality? picking the right data plane hardware for software defined networking", in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, ACM, 2013, pp. 103–108.

[63] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN", in *ACM SIGCOMM Computer Communication Review*, ACM, vol. 43, 2013, pp. 99–110.

[64] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes", in *ACM SIGCOMM*, 2015.

[65] P4 Language Consortium and others, *The P4 16 Language Specification, version 1.0.0*, 2017.

[66] C. Kim, P. Bhide, and E. Doe, "In-band network Telemetry (INT)", P4 Language Consortium, Tech. Rep., 2009.

[67] A. Abhashkumar, J. Lee, J. Tourrilhes, S. Banerjee, W. Wu, J.-M. Kang, and A. Akella, "P5: Policy-driven optimization of P4 pipeline", in *Proceedings of the Symposium on SDN Research*, ACM, 2017, pp. 136–142.

[68] D. Hancock and J. Van Der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane", in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, ACM, 2016, pp. 35–49.

[69] T. Jepsen, M. Moshref, A. Carzaniga, N. Foster, and R. Soulé, "Life in the fast lane: A line-rate linear road", in *Proceedings of the Symposium on SDN Research*, ACM, 2018, p. 10.

[70] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: a stream data management benchmark", in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, VLDB Endowment, 2004, pp. 480–491.

[71] S. Luo, H. Yu, and L. Vanbever, "Swing State: Consistent Updates for Stateful and Programmable Data Planes", in *Proceedings of the Symposium on SDN Research*, ACM, 2017, pp. 115–121.

[72] A. Gelberger, N. Yemini, and R. Giladi, "Performance Analysis of Software-Defined Networking (SDN)", in *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, Aug. 2013, pp. 389–393. DOI: 10 . 1109 / MASCOTS.2013.58.

[73] R. Giladi and N. Yemini, "A programmable, generic forwarding element approach for dynamic network functionality", in *Proceedings of the 2nd ACM SIGCOMM workshop on Programmable routers for extensible services of tomorrow*, ACM, 2009, pp. 19–24.

[74] L. Yang, R. Dantu, T. Anderson, and R. Gopal, "Forwarding and Control Element Separation (ForCES) Framework", RFC Editor, RFC 3746, Apr. 2004.

[75] H. T. Dang, H. Wang, T. Jepsen, G. Brebner, C. Kim, J. Rexford, R. Soulé, and H. Weatherspoon, "Whippersnapper: A p4 language benchmark suite", in *Proceedings of the Symposium on SDN Research*, ACM, 2017, pp. 95–101.

[76] *IEEE Std 802.1Q-2014 - IEEE Standard for Local and metropolitan area networks – Bridges and Bridged Networks*, IEEE Standard. [Online]. Available: https://standards.ieee.org/findstds/standard/802.1Q-2014.html.

[77] Spirent Communications, *TestCenter*, https://www.spirent.com/, [Online; accessed 12.11.2018], 2017.

[78] Netronome Systems, Inc, *Agilio CX 2x10GbE*, https://www.netronome.com/products/agilio-cx/, [Online; accessed 12.11.2018], 2017.

[79] ——, *Netronome NFP-4000 Flow Processor*, https://www.netronome.com/media/documents/PB_NFP-4000.pdf, [Online; accessed 12.11.2018], 2017.

ISSN 1432-8801