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

On Performance Assessment of Control Mechanisms and Virtual Components in SDN-based Networks

Anh Nguyen-Ngoc

Würzburger Beiträge zur
Leistungsbewertung Verteilter Systeme

Bericht 2/18

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

On Performance Assessment of Control Mechanisms and Virtual Components in SDN-based Networks

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

vorgelegt von

Anh Nguyen-Ngoc

aus

Würzburg

Würzburg 2018

Eingereicht am: 20.08.2018
bei der Fakultät für Mathematik und Informatik
1. Gutachter: Prof. Dr.-Ing. Phuoc Tran-Gia
2. Gutachter: Prof. Dr. Tobias Hossfeld
Tag der mündlichen Prüfung: 10.10.2018

Acknowledgements

I would like to express my great appreciation to those who have supported me and contributed to this dissertation as well as encouraged me during the whole time of my PhD studies and even before that.

First of all, my heartfelt thanks go to Prof. Dr.-Ing. Phuoc Tran-Gia, my supervisor, for providing me an opportunity that I can say "turn my life in a direction that I have never imagined", by accepting me as his PhD student. I have been extremely lucky to become a member of his Chair. Furthermore, he has given me the opportunity to cooperate with partners from both industrial companies and academic universities in several projects. I thank him for keeping my motivation strong to complete my dissertation with great support and kind advice. His valuable suggestions, constructive comments, and helpful guideline encourage me to try more day by day. I am genuinely grateful to him for placing his trust in me and sharing his exceptional scientific knowledge as well as his admirable qualities.

I would also like to extend my thanks to Prof. Dr. Tobias Hoßfeld for his willingness to become the second reviewer of my dissertation. His important suggestions and valuable feedback in very friendly discussions help me immensely improve the content of this work.

I wish to thank Prof. Dr. Reiner Kolla for his agreement to join the dissertation committee. He created a delightful atmosphere for the disputation. I truly appreciate his time and assistance.

I also very grateful to Prof. Dr. Harald Wehnes and Mrs. Wehnes for their kindness and encouragement when I was in Würzburg for the first time and during these last four years.

Many thanks to Ms. Alison Wichmann for taking care of an enormous number of my documents and administrative tasks. I have been received support from her for the enrollment of my study, paperwork related to business trips, and much helpful advice all the time.

I greatly appreciate the support of Prof. Dr. Thomas Zinner, my former group leader, for inspiring me to pursue the objectives of this dissertation. Without his support, inspiration, mentoring, peer pressure, and encouragement, this work would not have finished in time. I have learned and grown a lot while working with him, not only technical knowledge but also other skills.

Special thanks to Stanislav Lange, my group leader, for his enthusiastic support, availability and excellent suggestions. I really appreciate his willingness to discuss whenever I need it. I can not thank him enough for everything that he has done for making this work a possible one.

I have great pleasure of working with Nicholas Gray, Alexej Grigorjew, and Stefan Geißler in the fantastic NGN team. Thank you all for great support, interesting discussions, and positive feedback during my time here, especially in the writing phase of my dissertation.

My sincere thanks also go to Lam Dinh-Xuan, who took care of me when I had an accident with my leg last year. His encouragement and sharing have inspired me a lot to finish this work. Thank him also for many precious lessons from life that I have learned from him.

I very much appreciate the support of Dr. Florian Wamser for invaluable advice and useful experiences for the presentation of the dissertation. Besides, every conversation that related to football with him is absolutely fantastic.

I am hugely appreciative to all of my colleagues in the Chair of Communication Networks (Lehrstuhl für Informatik III), for giving me a chance to work in a scientific environment with pleasurable and friendly working atmosphere. Hence, I want to say thank to Dr. Matthias Hirth, Frank Loh, Kathrin Borchert, Anika Schwind, Susanna Schwarzmann, Christopher Metter, Dr. Florian Metzger, and Christian Moldovan for their enthusiasms and support, not only at work but also in the life outside the office.

I also truly thank former colleagues, Dr. David Hock, Dr. Steffen Gerbert, Dr. Michael Seufert, Dr. Valentin Burger, and Dr. Christian Schwarz for our sharing of sports passion, interesting discussions, and very nice time when we work together at the Chair.

I wish to thank Shpend Berani and Simon Raffeck for their hard work while doing their bachelor theses. They contribute a significant part for my research.

Last but not least, I want to especially thank my family, my friends, my professors at Hanoi University of Science and technology. Although they are not here with me in Germany, their encouragement is a tremendous inspiration to me throughout all my studies.

I could not have completed my research without the support of all these wonderful people. Thank you all!

Contents

1 Introduction **1**

1.1 Scientific Contribution 5

1.2 Outline of Thesis 8

2 Isolation of Virtual Networks in the context of SDN **11**

2.1 Background and Related Work 12

2.1.1 Software Defined Networking 13

2.1.2 Traffic Shaping and Traffic Policing 16

2.1.3 IEEE 802.3X Ethernet Flow Control 17

2.1.4 Related Work 20

2.2 Method Used 22

2.3 Evaluation results 26

2.4 Proof-of-Concept Implementation of Video Control Plane in the Case of DASH 35

2.4.1 Implementation of Video Control Plane 36

2.4.2 Performance of Network-assisted in VCP 40

2.5 Lessons Learned 46

3 Impact of the SDN Control Plane on the Data Plane Performance **49**

3.1 Background and Related Work 51

3.1.1 Operation of the SDN Control Plane 51

3.1.2 Methods of Sending FlowMod Messages 57

3.1.3 Performance Evaluation of SDN Components 58

3.2 Proactive Flow Installation Evaluation 61

3.2.1 Experimental Setup 61

3.2.2 Comparison of Switch Behavior 65
3.2.3 Accuracy Assessment of Measurement Mechanisms . . 67
3.2.4 Correlation Analysis of Measurement Mechanisms . . . 70
3.3 Influence of Control Plane Delay on Proactive Flow Installation 71
3.3.1 Sensitivity of Switches towards Control Plane Delay . . 73
3.3.2 Impact of Controller Implementation 77
3.4 Lessons Learned 79

4 Flow Monitoring Approaches in SDN Networks 83
4.1 Background and Related Work 86
4.1.1 Network Monitoring Approaches 86
4.1.2 Monitoring an SDN Network 89
4.1.3 Related Work 93
4.2 Performance of Adaptive Flow Monitoring in the ONOS Controller 96
4.2.1 Adaptive Flow Monitoring Algorithm 96
4.2.2 Evaluation Results 98
4.3 Selective Flow Monitoring 101
4.3.1 Measurement Setup 102
4.3.2 Selective Flow Monitoring Mechanism 103
4.3.3 Performance Evaluation of SFM 108
4.4 Lessons Learned 114

5 Conclusion 117

Acronyms 125

Bibliography and References 129

1 Introduction

In recent years, alongside the development of science and technology, computer networks have gained an important role in human life and changed the ways that people connect, entertain, or study. *Social networks* like Facebook or Twitter connect people globally. We exchange messages by using Over The Top (OTT) applications much more often than using Short Message Service (SMS) in mobile networks. *Smart TVs* connected to the Internet provide on-demand media instead of watching planned television programs and Internet links replace the transmission over radio waves. *eLearning* allows everyone everywhere to gain knowledge in many fields at any time with effective cost. Those are exemplary illustrations of the significant impact of computer networks on our lives. Also, this variety of user services and applications creates the diversity of traffic patterns in the networks and an enormous amount of data. Furthermore, each application has specific requirements to provide the best quality to the users. To this end, network infrastructures need to support these requirements promptly and stably. Consequently, an effective management approach and a stable system are required to adapt to a sharp rise in the volume of exchanged information as well as the requirements of applications.

However, the underlying network has suffered from limited innovation over the past decades in both management and configuration aspects. The legacy networks are mainly implemented in dedicated appliances, most functionalities within an appliance are implemented in dedicated hardware. This leads to several limitations that need to be taken into account. For example, adding a new device (switch, router, firewall) is time-consuming and is prone to errors. The reason is that configurational tasks have to be done separately, *device-by-device*, with both new and existing devices. This also makes it difficult to deploy new features such as routing algorithms, traffic engineering, or monitoring mechanisms without interrupting the operation of the whole network. Also, a variety of equipment in the system creates multi-vendor environments. This requires network administrators to have extensive knowledge of all devices from different vendors. Since each vendor has specific commands and syntax to execute the same function, it makes the scalability of networks more complex when using more proprietary equipment. Automatic configuration for flexibility demands of applications and services also are challenged due to this heterogeneous.

To overcome those issues, *virtualization* in computer networks has been more and more important in research. First, compute virtualization was introduced with simple hypervisors running on traditional servers and sharing resources such as CPU or RAM. Then, storage virtualization brought the ability to store data in the Internet without depending on the underlying hardware. On the next step, network functions were decoupled from proprietary hardware by using Network Functions Virtualization (NFV). Routers, switches, or load balancers can be replaced by a software instance on Commercial Off-The-Shelf (COTS) hardware. Additionally, Software Defined Networking (SDN) makes networks programmable and is able to change the behavior of physical devices in networks. It can easily adjust the configuration of devices or apply policies for traffic flows in the networks based on their characteristics to flexibly adapt changing needs. To accomplish this, the forwarding plane is separated from the central control plane, which centrally manages and decides how traffic passes through the network. Furthermore, SDN promises the cooperation of different vendors when using a common protocol to exchange information within a multi-vendor network. For example, OpenFlow-enabled switches can act as a layer 2 learning switch, a router or a firewall, depending on a script which is programmed by a network engineer. Although the switches are produced by different manufacturers, they support a common protocol (OpenFlow) that allows the devices to be programmable and work together in a network and under a common control entity.

This thesis focuses on SDN architecture and its performance to determine the influence parameters on the operation of an SDN network.

In SDN networks, the separation between the control and data planes is one of the most important principles and is depicted in Figure 1.1. This figure provides a general view of an SDN network including connections between another SDN network and legacy network via *Westbound Application Programming Interface (API)* and *Eastbound API*, respectively. While the information between applications and SDN controller is exchanged through *Northbound API*, control and management tasks are implemented from the controller to SDN switches using *Soundbound API*. Decision making for packet forwarding is made by a centralized controller.

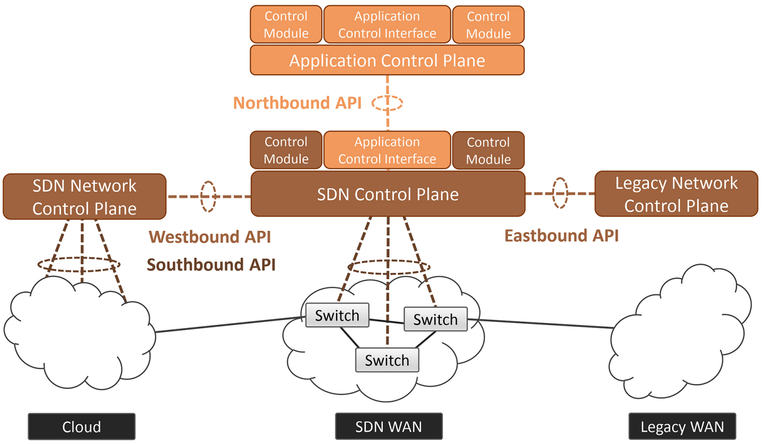


Figure 1.1: Standard interfaces in an SDN network [11].

SDN provides several benefits in terms of network management, control, and design. First, it is possible to automatically configure all network devices with more consistency and flexibility than legacy networks. In particular, the centralized controller has a global view of the network and secure connections to the devices, which allows administrators to provision of networks quickly without manual configuration. Second, SDN enables the ability to optimize data flows in the network [12]. A flow in SDN networks can be assigned multiple paths to its destination. Therefore, the traffic can be split across multiple nodes and has backup paths in case of failure. As a virtualization approach, SDN programmability brings opportunities to developers and network administrators to deploy any desired application. Furthermore, new network function or specific policy for packets can be tested in a part of the network, avoid interrupting the current operation. Finally, operational costs, as well as hardware expenses can be cost-effective while SDN replaces network function in the software instead of buying specialized physical hardware. Moreover, SDN is vendor-independent, hence, a variety of different devices can interoperate without conflict during the operation of the network.

Nevertheless, SDN also comes along with challenges including security, controller placement, performance, scalability, and reliability [13]. Before moving to an SDN-based architecture, network administrators need to make sure that network elements are capable of meeting their requirements with respect to those criteria. In this thesis, we focus on the performance of SDN networks towards a guideline for evaluating SDN performance before applying SDN paradigm. The evaluation covers both hardware and software perspectives, all SDN components, and the control mechanism of the SDN controller. First, we present a deeper understanding of the operation of SDN elements such as the mechanism of installing forwarding rules in SDN devices or approaches to monitor traffic flows implemented by the SDN controller. Then, influence factors on the performance of SDN networks including hardware devices and controller applications are determined. Finally, based on the measurement results of those parameters we show suggestions for decision making before utilizing SDN paradigm. It supports network administrators to decide not only appropriate forwarding hardware but also the instance of controller software and applications to obtain an efficient and stable system.

1.1 Scientific Contribution

This thesis aims at technical approaches to asses the performance of SDN networks covering all three layers of its architecture and components. First, corresponding influence parameters are determined. Second, the methodology for evaluating those parameters together with the proposed method for management and configuration SDN elements are described. Finally, discussion of evaluation results, which are obtained by applying testbed setups or emulation, is described.

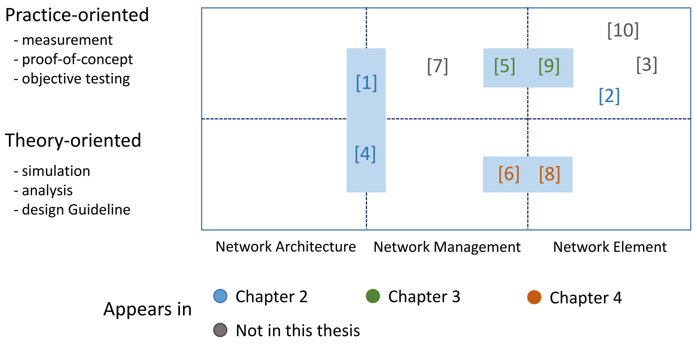


Figure 1.2: Contribution of this work as a classification of the research studies conducted by the author.

Figure 1.2 provides an overview of the contribution of this monograph with relevant publications. It covers different research areas and methods that have been focused on and used by the author during the course of this thesis. While the horizontal axis shows specific related areas of research, the implemented method is displayed on the vertical axis. Furthermore, different colors represent the contribution of the publication in particular chapter. Details of those areas are presented as follows.

The first part covers the area of network architecture. In this part, design of network-assisted approaches for video streaming called Video Control Plane (VCP) is presented. In order to obtain a dynamic adaptation of bandwidth for corresponding video resolution to optimize Quality of Experience (QoE), the interaction between video and network is required. This interaction is implemented by applying the SDN paradigm. On the one hand, we developed an API that allows programming SDN switches to keep track on network state and report to the SDN controller the information of video slices. The API contributes in threefold. First, using this API, the controller can add, remove, or configure dynamically virtual queues on network interfaces in a switch. Second, it provides the communication with both video server and video clients, hence, information like video bitrate or adaptation algorithms in the client are collected for management purposes. Finally, based on this API, the available bandwidth of specific video stream is allocated in an appropriate queue. On the other hand, we introduced an application that triggers control mechanisms such as quality adaptation, flow prioritization or bandwidth reservation for each video slice. It is shown that VCP strategies provide remarkable improvements in term of video quality fairness compared to the case in which VCP is disabled.

The second part focuses on ability to manage network resources and to guide traffic in the network depending on network state. The VCP mentioned above is also an example of researches in network management flied. Besides, we present the mechanisms that flows are monitored in SDN controller: *adaptive* and *selective* approaches. Our findings reveal that on the one hand, the *adaptive approach*, which monitors flows based on their lifetime, leads to high controller's resources usage and requires all flows to send their information. On the other hand, the *selective approach* allows querying particular flows as well as reduces the overload at the controller. Finally, the installation time of rules in SDN devices is investigated. To this end, we investigate the impact of delays between the control and data plane on the processing time of FlowMod messages, which are used to install forwarding rules. The investigation with different SDN switches with several measurement tools is performed. It is shown that each switch has its own behavior with different duration in which the controller and switch are in a synchronized state. Especially, a switch might inform the controller before it finishes installing all rules. These findings can help administrators either decide the appropriate device or incorporate the information to increase the reliability of the information exchange in the network. Further, the evaluation highlights software modules, which can be deployed in different controllers, gives a similar accuracy level in comparison with a commercial device when measuring the processing time. Therefore, such modules can be used as cost-effective approach and easily be obtained by similar measurements to evaluate a switch before actually deploying it in a network.

In the final part, the performance of individual elements in SDN networks is presented. On the data plane, the influence indicators on the isolation between virtual networks when sharing the same physical resource is identified. The isolation is typically performed by configuring buffers, rate limits, and queueing disciplines on incoming or outgoing interfaces. This part aims to addresses the following questions, e.g., how the isolation is implemented in SDN environment and to which extend the isolation is realized in state-of-the-art forwarding hardware. Hence, we study the isolation by implementing an investigation with OpenFlow switches in terms of packet loss in a network when a congestion occurs. Our evaluation highlights that a violation of the isolation between virtual networks is possible. Besides, the overall load on the outgoing switch port, the configured rate guarantees per virtual network also have a significant impact on the number of lost packets due to the violation. Nevertheless, with a specific combination of parameters and switch, the isolation is obtained properly. On the control plane, we confirm that the implementation details of the SDN controller also impacts the exchanged messages processing performance due to sender-side behavior. Furthermore, a benchmarking tool for SDN controller is deployed to evaluate the performance of an SDN controller in specific criteria. For example, the time that the controller needs to discovery different types of network topologies, how fast it reacts to a change in the network, or how long it takes to set a reactive path into switches. These guidelines can be used by either an administrator to select appropriate devices for his network or a controller developers to maximize compatibility and reliability of SDN components.

1.2 Outline of Thesis

This thesis consists of 3 main chapters which cover all components on the vertical view of SDN architecture. At the beginning of each chapter, a short introduction posing corresponding research questions is presented. Next, technical background information and related works regarding the topics are provided. Then, methodology to address the research question as well as the details of relevant evaluation are illustrated. Finally, each chapter ends with summarized lessons learned. These chapters are respectively introduced corresponding to the bottom-up study of the SDN architecture as visualized in Figure 1.3.

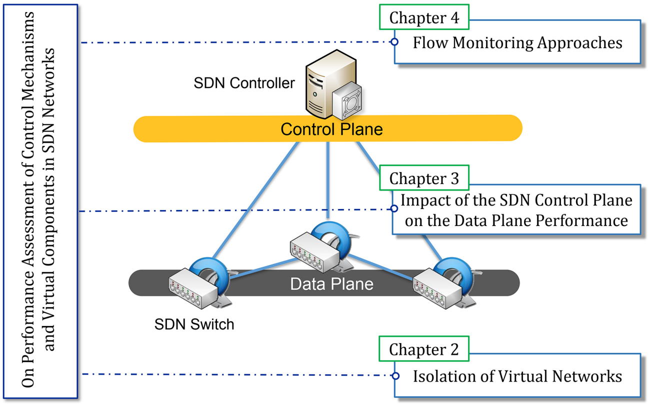

On Performance Assessment of Control Mechanisms and Virtual Components in SDN Networks
Chapter 4
Flow Monitoring Approaches
SDN Controller
Control Plane
Chapter 3
Impact of the SDN Control Plane on the Data Plane Performance
Data Plane
SDN Switch
Chapter 2
Isolation of Virtual Networks

Figure 1.3: Contents of the Thesis.

The next chapter focuses on the data plane with virtual networks in SDN environment and isolation between them. A general approach is studied to investigate the bandwidth guarantee while several networks share the same physical resource. Afterward, an implementation in specific SDN hardware is carried out to probe to which extend the isolation is realized in state-of-the-art hardware as well as whether congestion within one virtual network may affect the throughput performance of another. In addition, a proof-of-concept is given at the second half of Chapter 2. It leverages the previous investigation method in order to decide an appropriate OpenFlow switch for exchanging video streams in Dynamic Adaptive Streaming over HTTP (DASH) without influence between videos. Then, the design of network-assisted approaches, which perform by adapting network state according to video quality, is drawn.

In Chapter 3 the exchange information between SDN control and data planes is studied. To this end, an OpenFlow message which is used to install OpenFlow rule in switches is examined. In particular, the processing times of this message in different combinations of SDN controller instances and switches are presented to determine the impact of the SDN control plane on the data plane performance. The first part shows measurements which indicate how fast a switch installs flows when directly connects to a controller without network delay. Then, delays between the controller and the switches are emulated to provide an insight into the effect of control plane delays on the data plane flow installation. Moreover, we derive guidelines for choosing OpenFlow switches or controller instances for network administrator in term of flow installation time.

The application layer on top an SDN controller is investigated in Chapter 4 by comparisons between different flow monitoring approaches. First, Adaptive Flow Monitoring in ONOS controller, which is based on the lifetime of flows to decide relevant polling intervals, is implemented and compared to standard polling method. Then, an approach that relies on characteristics of flows and allows for flexible select specific flows to query their information is analyzed. Furthermore, we consider resources consumption of an SDN controller when applying the Selective Flow Monitoring as well as determining the influence parameters on flow monitoring.

Finally, Chapter 5 summarizes the presented work and concludes the major contributions.

2 Isolation of Virtual Networks in the context of SDN

In computer networks, when traffic flows are forwarded across a network, they simultaneously use network resources, such as link capacity and hardware equipment. In order to utilize these shared resources more effectively, Network Virtualization (NV) has been developed over the last decade. NV can be defined as "the sharing of network resources through the abstraction and isolation of network functionalities of the physical hardware" [14]. Since virtual networks share the same resources, in some circumstances, a network has to avoid the interference from others networks. For example, a specific type of traffic requires high throughput (video streaming, video conference, gaming); special policies applying for an individual network to achieve the privacy requirements; or a guaranteed amount of bandwidth of this network needs to be preserved to ensure Quality of Service (QoS). Therefore, it is important to identify virtual networks and isolate each network.

In legacy networks, Virtual LAN (VLAN) and Virtual Private Network (VPN) identify virtual networks through an ID (VLAN ID, VPN ID), then isolate the traffic between virtual networks by grouping computers into individual broadcast domains. On the other hand, Software Defined Networking (SDN), which is a state-of-the-art network technology, defines virtual networks based on flow-level. For instance, a network can be identified based on particular characteristics of the traffic, such as protocols, source/destination IP, MAC addresses, or application ports. In addition, SDN provides more flexible functions than VLAN and VPN, i.e., the flexible extensibility of networks and the effective programmability of the behavior of network devices through a central controller. However, it is not clear to which extent virtualized networks are isolated against each other in the context of SDN. For instance, how can the fairness among traffic flows in virtual networks be implemented and obtained as well as may a virtual network be affected by another due to congestion in one of them? In particular, the impact of several parameters like the specific configuration, the duration of the congestion, or the intensity of the overload in one virtual network may have an impact on the resource isolation and on the throughput of other virtual networks. Hence, this chapter describes a detailed investigation of isolation capabilities in a scenario while are multiple virtual networks sharing the same egress port of an SDN switch. This allows for evaluation of degree of the isolation between virtual networks, especially focusing on the granted bandwidth of a network.

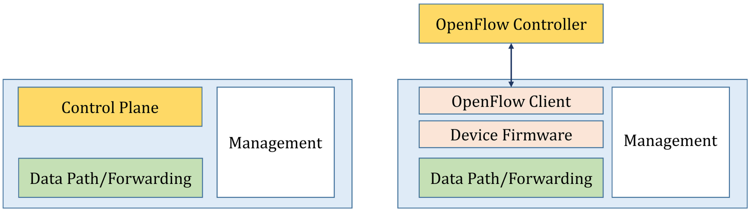
The content of this chapter is based on [1, 2] and is organized as follows. Section 2.1 introduces an overview of SDN and related work on the isolation between virtual networks, as well as two popular techniques that are used to ensure the granted bandwidth of a virtual network. Then, a study regarding the isolation in SDN environments, which discusses the extent of the isolation between virtual networks is realized in state-of-the-art hardware, is presented in Section 2.3. Afterwards, a specific use case with regards to the isolation between video streaming slices for Dynamic Adaptive Streaming over HTTP (DASH), is provided in Section 2.4. Finally, Section 2.5 describes the lessons learned on this topic.

2.1 Background and Related Work

This section presents the technical background and highlights the relevant works with regards to the isolation between virtual networks. Among these, Section 2.1.1 focuses on the concept of SDN. Then, details of traffic shaping and traffic policing to achieve a bandwidth limitation are presented in Section 2.1.2. Afterwards, an overview of the IEEE 802.3X flow control mechanism is described in Section 2.1.3. Finally, Section 2.1.4 introduces previous research regarding the virtual networks' isolation issues.

2.1.1 Software Defined Networking

In the following, basic principles of Software Defined Networking (SDN) are introduced, then an overview of the SDN architecture is given.



(a) Traditional network device architecture.

(b) OpenFlow-bases device architecture.

Figure 2.1: Comparison between the architecture of a traditional network device and an OpenFlow-based device. [15]

SDN Principles. In [11], basic principles of SDN are outlined. The first principle of SDN is *the separation* of control and data plane, which is a significant difference in comparison to traditional networks (or non-SDN networks). In traditional networks, the architecture of a typical forwarding device consists of three main components as shown in Figure 2.1a. First, the *Management Plane* provides methods to access and configure the device by using Simple Network Management Protocol (SNMP), Terminal Network (Telnet), or Secure Shell (SSH). Second, the *Data Plane* refers to packet forwarding based on information in a forwarding table or a Forwarding Information Base (FIB). Finally, the *Control Plane* determines how packets should be forwarded through routing protocols like Open Shortest Path First (OSPF) or Routing Information Protocol (RIP). All three components are located in the same physical hardware, meaning every time a new rule is added or changed, all participating devices have to be configured the same rule individually.

In contrast, SDN allows an administrator to configure the devices more flexibly by decoupling the control plane from the data plane and moving it to an entity called the *SDN Controller*, as depicted in Figure 3.8b. While the data planes are still implemented at every switch, the control planes are centralized at a dedicated machine. It indicates the second principle - *logically centralized controller*, which provides a more effective way to interact and configure network devices. The SDN Controller is a software platform and represents the Control Plane in the traditional network devices with the similar functions as mentioned above. The communication between those planes is implemented by using control protocols like OpenFlow [16]. OpenFlow is an example for the principle *open interfaces* of SDN, which implies the flexibility and adaptability development of software and needs to be public and open to community definition. Morover, this principle provides the ability of equipments from different vendors to interoperate.

Another key principle of SDN is the *programmability* that allows not only the behavior of network devices but also the operation of the whole network can be programmed as desired. For example, depending on what the controller demands, the device can act as a layer 2 switch, a router applying for layer 3 traffic, as well as the automatic gathering network statistics via scripts without the interaction via Command Line Interface (CLI) or SNMP can be implemented.

SDN Architecture. The aforementioned separation between the control and data plane in SDN creates the 3-layer architecture as Figure 2.2 illustrates. The lowest layer is the infrastructure layer, or the data plane, which consist of interconnected forwarding devices. Such devices contain *Flow Tables*, which have information of matched fields, counters and instructions for every flow in order to forward packets, instead of FIB in legacy switches.

The middle layer is the control layer that communicates with the data plane layer through a standardized interface, also called the southbound interface. In

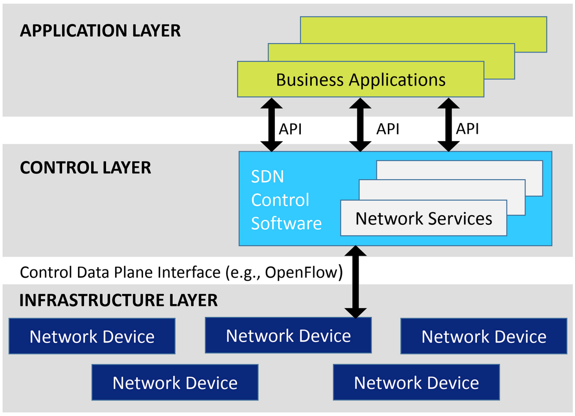


Figure 2.2: SDN architecture. [17]

order to program and manage the forwarding devices, control protocols like Openflow [16] or ForCES [18] are used. In this thesis, OpenFlow, which is standardized by the Open Networking Foundation (ONF) [1], is investigate since the used devices support this protocol. Beside the big amount of open source controllers [19], there are also many commercial controllers, which are developed by vendors like Cisco, HP, IBM, VMWare in order to optimize the operation of vendor hardwares and their own controllers.

At the top is the application layer running network applications. This layer provides network features, i.e. routing strategies, security and manageability. The northbound interface refers to the interface between software applications and the control layer. Furthermore, this layer can have a global view of the network and use that information to deploy appropriate instruction to the control layer. The applications use an API such as Representational State Transfer (REST) to exchange the information with the control plane. For example, an application might use REST APIs to send an HTTP GET/PUT/DELETE message to

[1]`https://www.opennetworking.org/`

the SDN controller, so it can execute management or configuration tasks, e.g., collecting statistics information, flow insertions or flow removals, respectively.

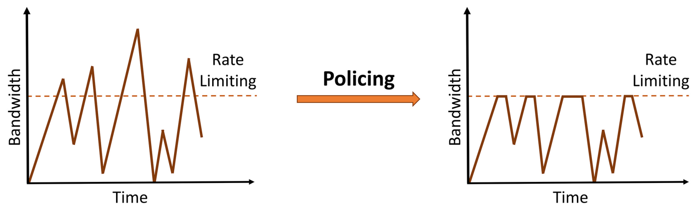
2.1.2 Traffic Shaping and Traffic Policing

When a customer signs a contract with an Internet Service Provider (ISP), there are several terms that are described in a Service Level Agreement (SLA). Basically, the SLA is a combination of commitments between service providers and their customers to satisfy one or several requirements for the services that are used by the customers. In addition to the utilization of a connection, SLA indicates a bandwidth that the customer desires, e.g., a connection having a maximum bandwidth of 100 Mbps and a minimum rate at 30 Mbps. However, the link from an ISP to the customer's location is normally a high-speed fiber connection and the capacity might be up to Gigabits per second that can contains multiple 100 Mps lines. In addition, multiple users can also share the same connection from the ISP with different applications. To use the transmission link effectively, there are two techniques are applied to restrict the bitrate of a physical connection to or to limit traffic rate in order to avoid exceeding bandwidth, traffic policing and traffic shaping, respectively.

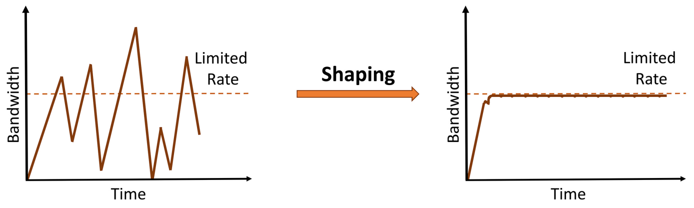
Figure 2.3 illustrates the main difference between these techniques when the traffic reaches a bandwidth threshold. On the one hand, *Traffic Policing* either drops or re-marks packets that exceed the committed rate. These tagged packets can be eliminated if congestion occurs. Hence, the shape of the output bandwidth appears as a saw-tooth as shown in Figure 2.3a.

On the other hand, *Traffic Shaping* is known as a traffic conditioning tool that allows packets to be sent at a configured speed. To ensure that the transmission rate does not exceed the defined rate, Traffic Shaping reduces the speed by retaining the additional packets in queues and delaying them for later transmission. Consequently, the shape of the output bandwidth is a smooth line, as shown on the right side of Figure 2.3b.

The other differences between *Traffic Policing* and *Traffic Shaping* are listed



(a) Traffic Policing.



(b) Traffic Shaping.

Figure 2.3: Traffic Policing vs. Traffic Shaping. [20]

in Table 2.1. While *Traffic Policing* is usually implemented on the provider side in order to limit access to resources when using a part of a high-speed link, or limit the traffic of certain application. *Traffic Shaping* is mostly performed on the customer side to slow the traffic rate and avoid exceeding the traffic which is in compliance with the SLAs. In this chapter, *Traffic Shaping* is evaluated on the egress interface by means of measurements on different OpenFlow switches in order to investigate whether a limited rate of a virtual network may be affected by the throughput performance of another virtual network when congestion occurs.

2.1.3 IEEE 802.3X Ethernet Flow Control

An impact of the IEEE 802.3X Ethernet flow control [21] on the isolation between VNs is also taken into account. IEEE 802.3X is a mechanism allowing a

Table 2.1: Comparison between traffic policing and traffic shaping [20]

Criteria	Traffic Policing	Traffic Shaping
Objective	Drop (or remark) excess packets above the committed rates	Buffer and queue excess packets above the committed rates
Token Refresh Rate	Continuous based on formula: 1 / committed information rate	Incremented at the start of a time interval.
Applicable on Inbound	Yes	No
Applicable on Outbound	Yes	Yes
Bursts	Propagates bursts, does no smoothing	Controls bursts by smoothing the output rate
Advantages	Controls the output rate through packet drops. Avoids delays due to queuing	Typically avoids retransmissions due to dropped packets
Disadvantages	Drops excess packets, throttling TCP window sizes and reducing the overall output rate of affected traffic streams	Can introduce delay due to queuing, particularly deep queues
Optional Packet Remarking	Yes	No

network device to send an Ethernet frame, called Pause frame, to tell its neighbor that it is experiencing overload, e.g., when the device is receiving data faster than it can handle.

Figure 2.4 depicts an example of the operation of the IEEE 802.3X flow control, the details are as follows:

① At the beginning, the transmitter sends data to the receiver, however,

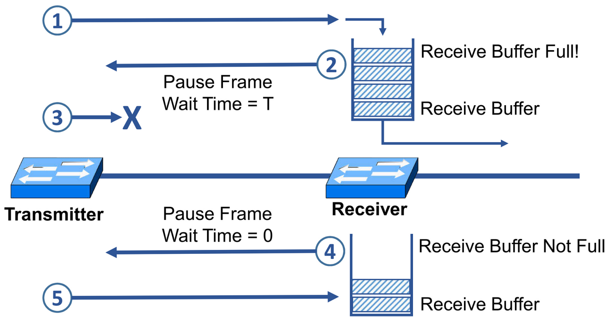


Figure 2.4: IEEE 802.3x Flow Control [22].

congestion occurs due to no space left in the receiver buffer on the interface that connected to the transmitter. This phenomenon usually happens, when the traffic is sent from a high-speed interface to a lower speed interface.

② To avoid discarding the arrival frames if the transmitter keeps sending data, the receiver dispatches a pause frame with a certain waiting time T to the transmitter to inform the transmitter stops sending frames for a certain period of time T.

③ When the Pause frame is received, the sender stops the traffic flow for a certain duration, which is specified in the frame, and buffers the packets until the receiver is ready to accept them again.

④ If the receiver buffer is empty before the waiting time is over, it informs the transmitter by sending another pause frame with a waiting time value of 0.

⑤ Finally, the transmitter restarts the transmission, either due to of receiving a pause frame with the waiting time of 0 or the waiting time is expired. The purpose of flow control is to prevent packet loss by handling the input buffer congestion.

2.1.4 Related Work

An important feature of virtual networks is to completely isolate each network, while they still share the same common physical resources. Therefore, the isolation becomes a very crucial criterion when applying Network Virtualization paradigm. As consequence, several solutions have been proposed to deal with the isolation issue. Bhatia *et al.* introduced *Trellis* [23] as a scalable platform which provides isolation between networks on the wide-area Virtual Network Infrastructure facility (VINI) [24]. Using Trellis, virtual networks are able to create their own topologies, routing protocols, and forwarding tables. Thus, interference between virtual networks can be prevented. Another research [25] uses Xen-based [26] virtualization to aim at providing variable cloud services on a physical network. Each virtual network is assigned a specific virtualized network ID to ensure the integrity of service content. Also based on the Xen plaftform, the authors in [27] propose *XNetMon*, in which the control mechanism and data forwarding are decoupled. The *XNetMon* monitors the use of shared resources and executes a punishment for the virtual networks which violate the isolation.

In addition, in term of isolation with reagards to bandwidth allocation between virtual networks, the isolation is typically realized by configuring buffers, rate limits, and queueing disciplines on incoming or outgoing interfaces. Hence, additional delays and jitter may be introduced due to waiting times in buffers at the switching fabric, the incoming, or the outgoing interface [28]. In case of a First-In-First-Out queueing discipline, buffering at the ingress results in head-of-line blocking and reduced throughput [29]. To overcome this issue, virtual output queues at the ingress have been proposed and implemented in order to achieve a high throughput of approximately 100% [30].

Explicit rate limits or guarantees for specific virtual networks, however, are typically configured at the outgoing queue. Although the crossbar fabric of a switch meets the Clos condition [31] and is thus non-blocking, congestion may happen due to several input ports sending to the same output port. Accordingly, the question arises, whether congestion in one virtual network, realized by rate guarantees on an outgoing interface, may also influence the performance of other virtual networks. Particularly, the impact of several parameters like the specific configuration, the duration of the congestion, or the intensity of the overload in one virtual network may have an impact on the resource isolation and on the throughput of the other virtual networks.

The authors of [32] present a theoretical model of a multi-core router that supports multiple VNs simultaneously and compare it with a simple store-and-forward router model that uses a single processor for all incoming traffic. Due to isolation as well as VN-specific scheduling and processing of packets, the multi-core model achieves better performance with respect to delay and jitter when load is increased for all VNs. However, the focus of this work lies on the resilience aspect of isolation mechanisms, especially in the presence of sudden bursts that congest the underlying physical queue of the multi-core router.

In [33], the behavior of virtual routers based on commodity hardware is investigated. For this, competing objectives including performance, isolation, flexibility, and fairness are taken into account. Problems of software-based traffic classification approaches that use a single physical queue are illustrated in an analysis of their isolation performance. Emulating a system with multiple physical queues via a switch that is connected upstream of multiple Network Interface Cards (NICs) and comparing it to a system that uses software classification, shows that with increasing load, the latter fails to maintain isolation. This results in unexpected behavior on all virtual queues.

The work in [34] investigates bandwidth guarantees for TCP flows in an SDN network in combinations with different switches and the Floodlight controller [2]. Two TCP traffic flows share a bottleneck connection after leaving the switch. Due to the bottleneck, congestion occurs and packets are queued at an interface

[2]`http://www.projectfloodlight.org/`

on the switch. The results show that not every switch ensures the configured bandwidth guarantees. While TCP traffic is the focus of this work, we investigate the isolation between virtual networks in case the data plane traffic is exchanged via the UDP protocol.

2.2 Method Used

The following sessions towards an understanding of the questions raised regarding isolation performance by investigating key metrics like the packet loss per virtual network using measurements in a dedicated testbed. In this section, the measurement setup, including the used hardware and software is described. Additionally, an overview of the performed experiments is provided alongside the notation for adjustable parameters and key performance indicators. Finally, the various experiment scenarios in terms of parameter values and hardware configuration are discussed.

Measurement Setup and Configuration. In order to evaluate the isolation performance of the outgoing physical queue of a switch that contains multiple virtual queues, a testbed has been set up according to Figure 4.12.

Its main components are two traffic generating hosts, h_1 and h_2, which are connected to an OpenFlow switch and send Iperf[3] UDP traffic to a third host, h_3, also connected to the switch. There are three switches with different hardware specifications as shown in Table 3.3. Each vendor has its own command for setting up queues with minimum and maximum rates on the outgoing port of the switch. For example, Pronto 3290 uses the *ovs-vsctl* command, HP 2920 applies *per-queue rate-limiting*, and NEC P5240 implements *qos-queue-list*.

For its OpenFlow functionality, the switches require a connection to an OpenFlow controller. In this work, the OpenDaylight controller[4] was chosen. In this environment, separated virtual queues are established on the switch for each

[3]`http://iperf.sourceforge.net/`
[4]`http://www.opendaylight.org/`

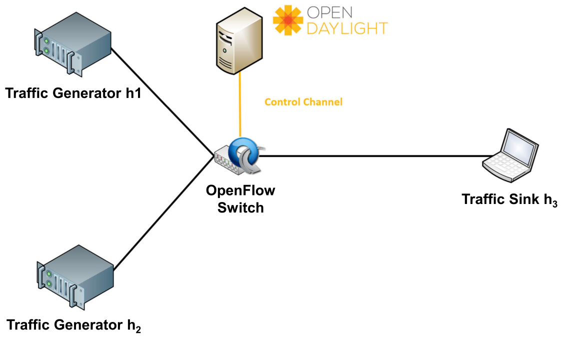


Figure 2.5: Testbed Setup

Table 2.2: Switches used in this work.

Switch	CPU	Memory	Software
Pronto 3290	MPC8541 825 MHz	512MB	PicOs 2.0.14 (Open vSwitch v1.10.0)
HP 2920	Tri Core ARM1176 825 MHz	512MB	WB.15.17.0007
NEC PF5240	PowerPC 667 MHz	1024 MB	OS-F3PA v5.0.0.1

individual sender. Thus, traffic originating from a specific source is directed to a specific virtual queue on the switch's output port via statically preconfigured OpenFlow rules.

Course of Experiments. For the actual evaluation, several influence parameters and performance indicators have been identified and were integrated into an experiment scheme. The basic scenario is depicted in Figure 2.6. It consists of traffic generator h_1 sending UDP packets with a constant rate of β_1 over the switch's first queue and the second traffic generator sending regular bursts of

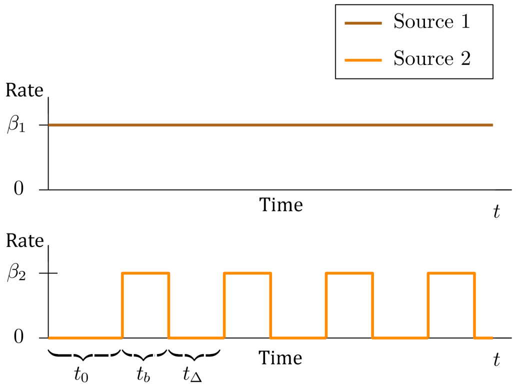


Figure 2.6: Experiment scheme for $n_b = 4$

UDP packets with a rate of β_2 over the second queue. The two queues are referred to as q_1 and q_2, respectively, and have rate limits β_1^l and β_2^l, which are obtained by applying traffic shaping. In almost experiments, the sending rate of h_1 is equal to q_2's limit, i.e., $\beta_1 = \beta_1^l$. At the beginning of each run, q_2 is idle for time t_0 in order to assure a stable system state. Then, n_b bursts of UDP traffic are produced at the second traffic generator via Iperf in regular intervals. These bursts last t_b and are interleaved with pauses of t_Δ. During the bursts, the rate limit of q_2 is exceeded, i.e., $\beta_2 > \beta_2^l$ holds.

Measured parameters include the relative and absolute packet loss on q_1, denoted as p_1 and p_1^a, respectively, as well as the received rates for both queues, β_1^r and β_2^r. These values allow statements about the influence of q_2's bursts on the performance of q_1, e.g., $p_1 > 0$ implies a violation of the isolation between queues. Furthermore, changes in the receive rate for the first flow, β_1^r, indicate a performance degradation. Hence, these parameters can be used to quantify the impact of bursts on the overall system reliability.

Table 2.3: Parameter sets used in this work

Name	$\beta_1 = \beta_1^l$	β_2	β_2^l	c	ρ	ρ_p	t_b	t_Δ	n_b
$\mathcal{P}_1$	640 Mbps	465 Mbps	100 Mbps	1 Gbps	1.1	0.75	{1s,2s,5s}	3	5
$\mathcal{P}_2$	640 Mbps	465 Mbps	206 Mbps	1 Gbps	1.1	0.85	{1s,2s,5s}	3	5
$\mathcal{P}_3$	640 Mbps	465 Mbps	255 Mbps	1 Gbps	1.1	0.90	{1s,2s,5s}	3	5
$\mathcal{P}_4$	640 Mbps	640 Mbps	100 Mbps	1 Gbps	1.28	0.75	{1s,2s,5s}	3	5
$\mathcal{P}_5$	640 Mbps	640 Mbps	206 Mbps	1 Gbps	1.28	0.85	{1s,2s,5s}	3	5
$\mathcal{P}_6$	640 Mbps	640 Mbps	255 Mbps	1 Gbps	1.28	0.90	{1s,2s,5s}	3	5

Investigated Scenarios. Given the total capacity c of the link between switch and traffic sink, the total load can be computed as $\rho = \frac{\beta_1+\beta_2}{c}$. This is a key influence factor as in the context of a low total load, the system is more likely to compensate short or less intense bursts. Furthermore, the amount of resources allocated by the network operator can have an impact on isolation performance. Hence, a parameter for the provisioned bandwidth, $\rho_p = \frac{\beta_1^l+\beta_2^l}{c}$, is introduced. Table 2.3 provides an overview of parameter combinations discussed in this work. Additionally, flow control mechanisms like IEEE 802.3X Ethernet flow control [21] can prevent overload by limiting the sending rate of traffic sources. However, this behavior can be turned off by the client in order to produce the desired amount of traffic regardless of the system state. In order to quantify the impact of flow control, experiments have been conducted for both settings.

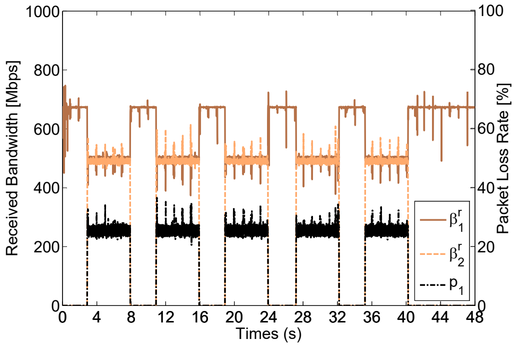
2.3 Evaluation results

This section presents results gained during the experiments described in Section 2.2. First, results obtained from a single experimental run with different switches are conducted. Then, the influence of parameters like the system load ρ, the provisioned bandwidth ρ_p, and the burst duration t_b on the resulting packet loss is investigated in the particular case of using Pronto 3290 switch as well as relationship between these parameters and the amount of time during which packet loss occurs is analyzed.

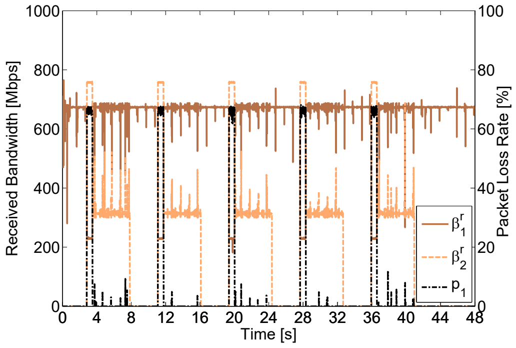
Single Experiment Run

In order to provide an illustrative example of the performed measurements, Figure 2.7 depicts a single experimental run with flow control disabled and a set of parameters as [β_1^l = 700 Mbps; β_2^l = 300 Mbps; β_1 = 670 Mbps; β_2 = 755 Mbps]. In those figures, the x-axis indicates the time while the two y-axes show the received throughput at the sink on the left hand side as well as the packet loss experienced by the traffic originating from the first traffic generator on the right hand side, which is indicated by the dash-dot line. While the solid dark brown and the dash light brown lines depict the received bandwidth from the first and the second traffic generator, respectively.

As displayed in Figure 2.7a and Figure 2.7b, in the first 3 seconds, only the traffic from the first source passes through the switch with the rate β_1. Then, at the 4th second, the second traffic starts, a congestion occurs. Due to the rate of the second network exceeds the configured maximum rate β_2^l, the bandwidth of the first network decreases, which indicates that the isolation of this network is violated. After the second source stops sending packets, from the 8th second on, the rate of the first flow is back to β_1. This phenomenon repeats several times in an experiment. The bursts from the second traffic generator initially affect the amount of data received from the first source as the total sending rate exceeds the link's capacity of 1 Gbps resulting in congestion of the output port.



(a) NEC PF5240.



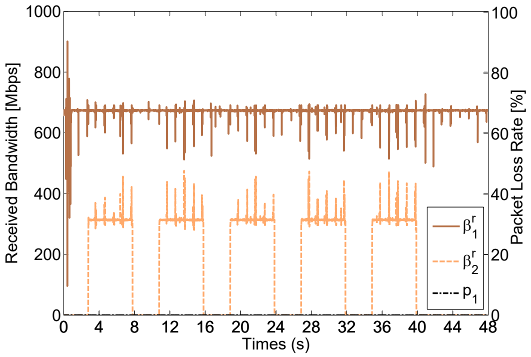
(b) Pronto 3290.

Figure 2.7: Comparison between switches in a single experiment, IEEE 802.3x flow control off [β^l_1 = 700 Mbps; β^l_2 = 300 Mbps; β_1 = 670 Mbps; β_2 = 755 Mbps].

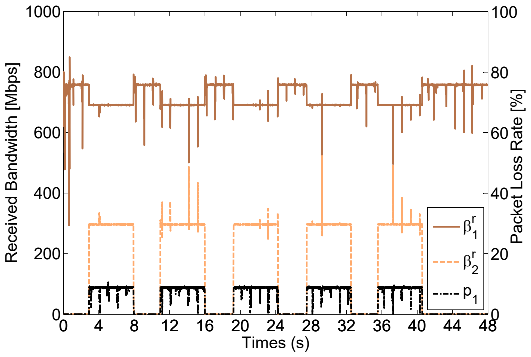
While this behavior in itself indicates a breach of isolation between VNs, additionally packets loss occurs for the traffic that passes q_1. In case of the NEC switch, Figure 2.7a, the link bandwidth is equally shared among both virtual. It makes the amount of packet loss on the first network is around 25% due to the reduction of its bandwidth and the packets are lost the whole 5s of the congestion as the dash-dot black bursts show in Figure 2.7a. A significant smaller bandwidth degradation duration is displayed in Figure 2.7b in case of the Pronto 3290, as indicated by intervals that have black boundaries. They last nearly 60ms, however, a higher packet loss rate, up to 67% is observed during this interval. After the short reduction, the received rate β_1^r is recovered as at the beginning.

While Figure 2.7 presents the result in case of NEC and Pronto switch, the data conducted from measurement of HP 2920 is illustrated in Figure 2.8. With the same parameter set as the previous figure, however, Figure 2.8a depicts a different observation, the dash-dot line and the x-axis are identical, meaning no packet loss from the first source. The congestion still arises and makes the bandwidth of the second network occupy only the rest of link capacity, approximately 300 Mbps instead of 775 Mbps, which is indicated by the dash light brown bursts. For this specific parameter set, the first network is isolated almost entirely. Nevertheless, with other parameter sets, for instance, [β_1^l = 700 Mbps; β_2^l = 300 Mbps; β_1 = 760 Mbps; β_2 = 700 Mbps], a breach of isolation is caused as demonstrated in Figure 2.8b. Although the guaranteed bandwidth β_1^l is ensured, the packets are dropped since the rate β_1 is decreased from 760 Mbps to 700 Mbps.

Figure 2.7 and 2.8 show that a violation of the isolation between VNs is possible for all switches. In order to have an insight into influence factors on the isolation, we focus on the experiments with the Pronto 3290 then present the evaluation results as follows. First of all, the impact of the IEEE 802.3x flow control is investigated. Figure 2.9a illustrates the behavior in case of flow control enabled. Similar to the previous result, the second traffic causes packet loss, however, only less than 1% for the traffic that passes q_1. Then due to flow control is enable, it kicks in fast and restricts the sending rate of the second generator, thus avoiding further congestion and therefore packet loss or bandwidth degradation for the first flow.

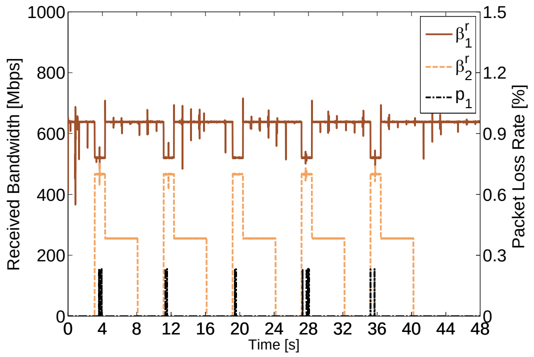


(a) Parameter set [β^l_1 = 700 Mbps; β^l_2 = 300 Mbps; β_1 = 670 Mbps; β_2 = 755 Mbps].

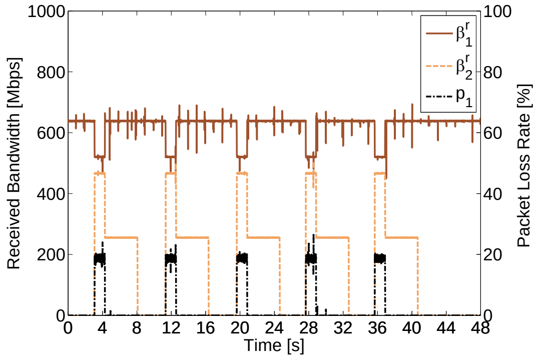


(b) Parameter set [β^l_1 = 700 Mbps; β^l_2 = 300 Mbps; β_1 = 760 Mbps; β_2 = 700 Mbps].

Figure 2.8: Comparison between parameter sets for HP switch in a single experiment, IEEE 802.3x flow control off.



(a) IEEE 802.3x flow control on.



(b) IEEE 802.3x flow control off.

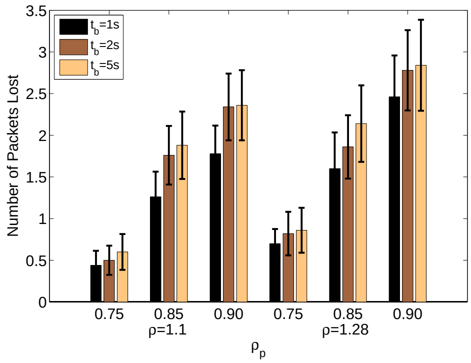
Figure 2.9: Single experiment with Pronto 3290 and parameter set $\mathcal{P}_3$ and $t_b = 5s$.

In contrast to the outlined scenario, the packet loss experienced in the absence of the flow control mechanism is orders of magnitude higher, i.e., at around 20%. Detailed measurement data is provided in Figure 2.9b. Although flow control is disabled, the receive rate β_2^r also drops shortly after the bursts start. Unlike the first case, however, this is not due to an adapted β_2, but rather due to the switch dropping packets for q_2 after experiencing an incoming rate β_2 that is higher than the provisioned rate β_2^l. The plots show that performance degradation in expectedly isolated virtual networks can occur by overload conditions in a single VN. While flow control does affect the intensity of the resulting degradation, flow control can not completely avoid it. In addition to the flow control mechanism on the sending side, the switch also behaves in a reactive fashion when incoming rates exceed the limits agreed-upon for a certain amount of time. In both cases regarding flow control configuration, a further interesting effect that is observed is that the received rate of traffic from the first source, β_1^r, is significantly reduced during the overload period while the second flow passes through at its full sending rate β_2.

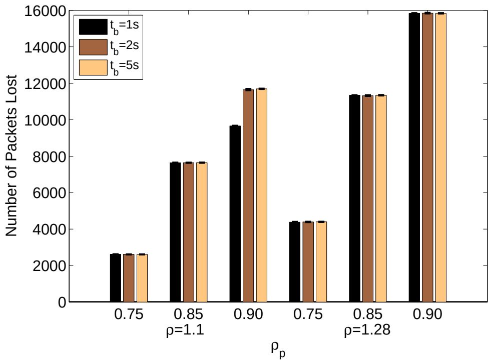
Impact of Experiment Parameters on Packet Loss

As motivated in Section 2.2, this work aims at identifying the main influence factors of the phenomena observed in the previous measurements with Pronto 3290. For this purpose, experiments with a variety of system configurations have been performed. To achieve statistical significance, each run was repeated ten times. Figure 2.10 presents the influence of the system load ρ, the provisioned bandwidth ratio ρ_p, and the burst length t_b on p_1^a, the absolute number of packets lost in a single burst event.

The first picture shows results gained from configurations in which flow control was turned on. While the ticks on the x-axis denote ρ_p, the ratio of provisioned bandwidth and link capacity, the y-axis displays p_1^a. In addition to the bars' height indicating the mean of measured values, the whiskers show 95% confidence intervals that were obtained by repeating each experiment 10 times. Bar colors represent different values for t_b. Results are presented for two different system loads ρ. The first observation is that confidence intervals for each group of scenarios with identical ρ and ρ_p overlap, i.e., different values of t_b do not have a statistically significant impact on the amount of lost packets when



(a) IEEE 802.3x flow control on



(b) IEEE 802.3x flow control off

Figure 2.10: Single experiment with parameter Pronto 3290 set $\mathcal{P}_3$ and $t_b = 5s$

the remaining parameters are fixed. The reason for this is that the flow control mechanism reacts fast enough to prevent a large packet loss, but does not affect the amount of time during which the first flow's received rate β_1^r is reduced. Thus, packet loss acts as indicator while the bandwidth impediment constitutes the actual breach in isolation. Furthermore, increasing values of ρ_p result in a higher number of lost packets p_1^a. A possible explanation for this behavior is that preconfigured rate guarantees influence the duration until the switch reacts to the overload. Thus, the time frame in which packet loss might occur becomes longer for increasing ρ_p. Hence, p_1^a, the number of lost packets increases. The applied system load ρ, however, does not seem to have a high degree of influence on the packet loss in case of enabled flow control. This can be observed by comparing pairs of groups that have identical values of ρ_p but different loads ρ. A likely explanation for this phenomenon is that while a higher value of ρ implies a higher congestion rate, it also leads to a faster detection of the burst by the flow control mechanism. The latter limits the damage done in terms of packet loss by limiting the sending rate at h_2.

Table 2.4: ρ_p and resulting t^i values for different scenarios

Parameter set	ρ_p	t^i
$\mathcal{P}_1$	0.75	0.27
$\mathcal{P}_2$	0.85	0.81
$\mathcal{P}_3$	0.90	1.2
$\mathcal{P}_4$	0.75	0.25
$\mathcal{P}_5$	0.85	0.48
$\mathcal{P}_6$	0.90	0.68

Figure 2.10b displays the measurements that were obtained with disabled Ethernet flow control. Like in the previous figure, an increased number of lost packets is observed for increasing values of ρ_p. In contrast to just a few lost packets when flow control is enabled, several thousands of packets are lost in the context of disabled flow control. Furthermore, ρ also affects the number of lost packets. However, t_b does not seem to have an influence on p_1^a even in the absence of flow control. Except in the case of the combination $\rho = 1.1$ and $\rho_p = 0.90$, the number of lost packets per burst event does not change significantly when t_b is

increased.

To shed light on this behavior, the duration in which packet loss for packets from h_1 occurs has been recorded in each experiment. This period is denoted as t^i and values for different parameter sets are shown in Table 2.4. Additionally, the table presents ρ_p which quantifies the total bandwidth provisioned. Only in the context of parameter set $\mathcal{P}_3$, t^i exceeds one second which explains the difference in the amount of packets lost observed in Figure 2.10b. As soon as $t_b > t^i$ holds, the absolute packet loss p_1^a does not change for increasing t_b.

Amount and Duration of Packet Loss

Having identified ρ_p as the main influence factor on the packet loss for the traffic from h_1 to h_3 in case of enabled flow control, Figure 2.11 provides an aggregated view on the empirical cumulative distribution function of p_1^a for different values of ρ_p.

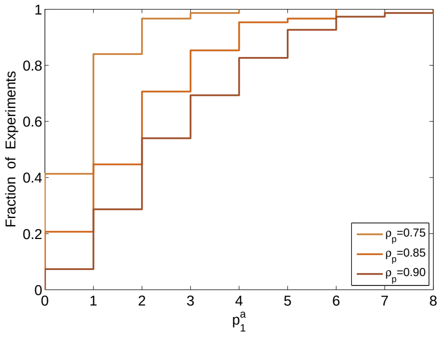


Figure 2.11: Distribution of the absolute packet loss p_1^a for different values of ρ_p (IEEE 802.3x flow control is enabled)

The x-axis denotes the number of packets lost during a single burst attempt, while the y-axis indicates the fraction of experiments for which the observed value p_1^a is less than or equal to this number. The resulting distributions are in line with the previous observations and exhibit an increasing number of lost packets for increasing values of ρ_p. For example, in the context of $\rho_p = 0.75$, p_1^a never exceeded 4 packets per burst while it did in 17% and 19% of instances for $\rho_p = 0.85$ and $\rho_p = 0.90$, respectively. In contrast to scenarios with enabled flow control, where ρ_p is the main influence factor, multiple parameters have been identified for configurations in which the flow control mechanism is disabled. Thus, a distribution of p_1^a that solely depends on ρ_p does not provide new insights and is therefore omitted.

2.4 Proof-of-Concept Implementation of Video Control Plane in the Case of DASH

In the scheme of Dynamic Adaptive Streaming over HTTP (DASH), network resource allocation is managed in a distributed way at the end-points by each client. On the on hand, the clients are running controllers to autonomously change the video bitrate to improve the Quality of Experience (QoE). On the other hand, the resource is fairly shared with respect to the QoS parameters, but not with respect to the user's QoE. An interaction between video and network provider is a solution to overcome this problem. The exchanged information can be leveraged by a video control plane enforcing network-assisted strategies. In doing this, a network element can trigger a control mechanism such as quality adaptation or bandwidth reservation, based on the network condition and client context. We implement such a scheme using an SDN network and deploy a Network Controller, which runs on top an SDN controller, together with several network-assisted streaming strategies in a Video Control Plane (VCP).

This section, as a proof-of-concept for the approach towards a joint video and network control, describes the design and structure of VCP. Then, a comparison of performance between using VCP with several network-assisted strategies and the standard case without VCP as well as a result related to bandwidth reservation are presented.

2.4.1 Implementation of Video Control Plane

Control System Architecture. In order to enforce a video quality management policy in the scenario of a single bottleneck link between video clients and the content server, we implement a control system with the *Network Controller (NC)* as shown in Figure 2.12.

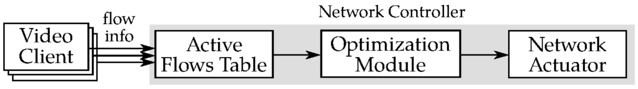
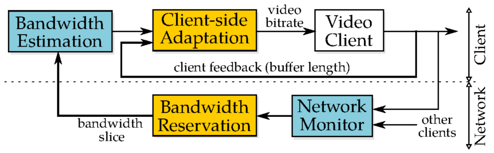


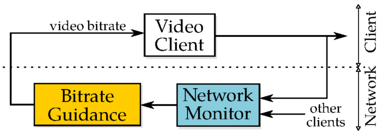
Figure 2.12: Control system block diagram.

The NC, which runs on top of an SDN controller, carries out two main functions. First, it creates and manages bandwidth slices implemented through dedicated queues on the network. Second, it handles a bidirectional communication with the video clients via a REST API. There are three components included in the NC: the *Active Flows Table*, the *Optimization Module* and the *Network Actuator*. Information of the currently active video sessions, which is provided by each video client at the beginning of the video session, is stored in the Active Flows Table. The Optimization Module then uses this information periodically to compute a new bitrate assignment every T_s. Finally, the Network Actuator enforces the computed bitrates (or bandwidth) to each active video session. Another component of the control system is the *Video Client*, which establishes/tears-down the video session by sending messages to the NC and downloads the corresponding video segments.

Network-assisted Streaming Approaches. Three categories of network-assisted approaches are investigated based on the aforementioned architecture. The *Bandwidth Reservation (BR)* assigns a bandwidth slice to a video flow through two nested control loops as depicted in Figure 2.13a. The *outer control loop* is executed in the network and configures the bandwidth slice, whereas the *inner control loop* running at the client, autonomously selects the video bitrate based on the video client feedback and bandwidth estimates. On the other hand, the *Bitrate Guidance (BG)* computes the video bitrate by a centralized algorithm running in a network element and enforced by the video client, as shown in Figure 2.13b. Finally, a combination of Bandwidth Reservation and Bitrate Guidance is taken into account as *hybird strategies*.



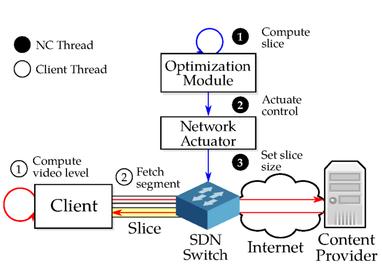
(a) Bandwidth Reservation.



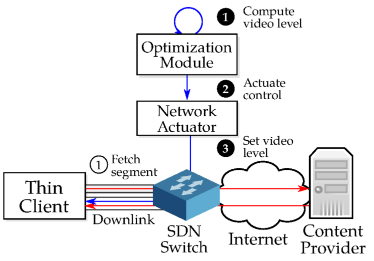
(b) Bitrate Guidance.

Figure 2.13: Network-assisted approaches for adaptive video streaming.

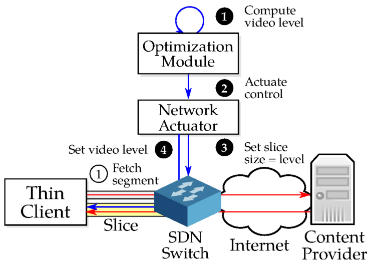
Bandwidth Reservation. The dedicated bandwidth slices are reserved to the video flows by the NC in this approach. The adaptation algorithm in the client computes the video bitrate independently without any information from the NC. Figure 2.14a visualizes the details of the implementation of the Bandwidth Reservation: ❶ the *Optimization Module* calculates the bandwidth slice assignment based on the management policy, ❷ the computed bandwidth slice is sent to the *Network Actuator*, and ❸ the dedicated slice for the flow (or a group of flows) is also created or updated. For each video segment download, the Client thread executes two actions: ① the video bitrate is selected according to its adaptation algorithm and ② the segment is retrieved from the Content Provider.



(a) Bandwidth Reservation.



(b) Bitrate Guidance.



(c) Bandwidth Reservation and Bitrate Guidance.

Figure 2.14: Network-assisted approaches for adaptive video streaming.

Bitrate Guidance. When using this approach, it is important to notice that all video flows share the same bandwidth slice. The download rate is shaped by the client in order to match the selected bitrate, thus providing service differentiation to the flow in the shared slice. The first two actions of the NC thread ❶ and ❷ in Figure 2.14b are exactly the ones executed in the BR approach. Instead of creating bandwidth slices on the network interface, the Network Actuator delivers the computed bitrate for the clients to the switch at action ❸ . Only action ① is executed in the Client in this case, it downloads the next video segment based on the video bitrate set by the NC. Such a client is labeled as a *Thin Client* since it does not send any feedback information to the NC.

Bitrate Guidance and Bandwidth Reservation. The combination of the two strategies described above is carried out. In particular, the third action of the NC thread is split into two sub-actions: 1) the bandwidth reservation in the network and 2) the bitrate guidance. Again, the client only downloads video segments.

In addition, the interactions between the client-side control loop and the network control loop are investigated with regards to three algorithms. While *Conventional* selects the video bandwidth based on bandwidth estimates [35], *PANDA* is designed to cope with the fairness issues affecting several HTTP-based Adaptive Streaming (HAS) algorithms [35] by incrementing the bitarte to probe the available bandwidth [35]. Finally, unlike those two *rate-based* algorithms, *Elastic* is a *level-based* algorithm which allows to control the playout buffer length by varying the bitrate [36]. Furthermore, *Elastic* is able to overcome fairness issues affecting the rate-based algorithms.

Video Session Management. The workflow of a video session is illustrated in Figure 2.15. A session is started at the client by retrieving the playlist from the video server. Each playlist carries the Structual SIMiarity (SSIM) values, which represent the estimated quality of videos. Then, the client sends the information needed to compute the optimal bitrate distribution in a *set-up* message to the NC: the video contend URL, the video level set and its corresponding SSIM extracted from the playlist.

As mentioned in the previous paragraph, the NC periodically executes Optimization Modudle with a sampling time T_s, the video flow cannot be served with differentiated service until the next execution. Therefore, in order to avoid a delayed start-up, the NC assigns the flow to the *Arrival Slice*, which is reserved to newly arrived video session. Afterwards, at most after T_s seconds, the Optimization Module is executed, the video flow is removed from the *Arrival Slice* and served with the differentiated service according to the adopted network-assisted approach. Finally, a *tear-down* message is sent from the client to the NC to notify that the client decides to terminate the session.

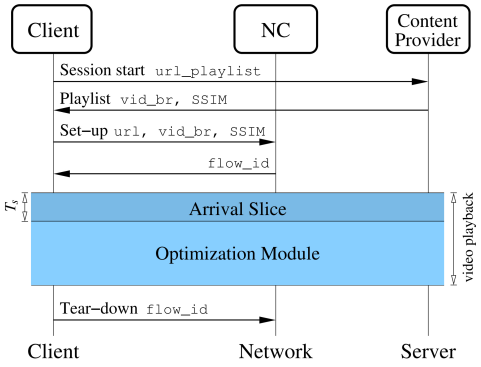


Figure 2.15: Video session flow diagram.

The Content Provider stores three videos in the catalog: Big Buck Bunny[5], Sintel[6] and Tears of Steel[7] with different resolutions 720p, 1080p, and 2160p corresponding to three classes of client devices. All videos are encoded with H.264 codec with a frame rate equal to 30 fps and the segment size is fixed to 4 seconds.

2.4.2 Performance of Network-assisted in VCP

This section provides an evaluation of the performance of the strategies mentioned in the first section. After testbed setup is introduced, the experiment scenario and measurement metrics are described. Finally, a comparison between the network-assisted strategies and result related to bandwidth reservation are presented.

[5]http://distribution.bbb3d.renderfarming.net/video/mp4/bbb_sunflower_2160p_30fps_normal.mp4
[6]https://download.blender.org/durian/movies/Sintel.2010.4k.mkv
[7]http://ftp.nluug.nl/pub/graphics/blender/demo/movies/ToS/tearsofsteel_4k.mov

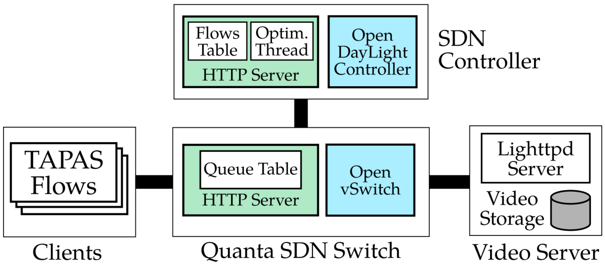


Figure 2.16: The Implementation of the VCP.

Tesbed Setup. The Video Control Plane (VCP) is implemented in the testbed as depicted in Figure 2.16. A computer [8] runs the OpenDayLight Hydrogen Release and the NC as well as an HTTP server that is responsible for exchanging information with an OpenFlow switch. By using the methodology in Section 2.2 with different switches, we choose the QUANTA T1048-LB9 as the forwarding device in our testbed due to its ability to ensure isolation between TCP flows. The switch is running PicOS [9] v2.6 and Open vSwitch [10] 2.3.0 as software switching stack. TAPAS (Tool for dApid Prototyping of Adaptive Streaming control algorithms) [37] is set up in a machine that generates a configurable number of DASH video flows. The three algorithms in the previous paragraph are implemented using TAPAS as well as the Thin Client in case of Bitrate Guidance. Another machine acting as video content server, stores video segments with different qualities (resolutions) and runs the *Lightpd* HTTP server to send those segments to the clients.

Figure 2.16 visualizes the implementation of the VCP. The NC contains two Python-based HTTP servers hosted by controller and OpenFlow switch. The HTTP Server at the controller has three main functions: first, it stores the Ac-

[8]Intel Core Duo/4GB RAM /Ubuntu 14.04
[9]http://www.pica8.com/products/picos
[10]http://openvswitch.org

tive Flow Table that has information of all flows which are currently installed in the switch. Second, it executes the optimization algorithms in the Optimization Module to set the minimum guaranteed rate for the flows assigned on the corresponding queue. Finally, the communication pipes with the switch and the clients are established via JavaScript Object Notation (JSON) APIs in order to manage the QoS queues in the switch, as well as to send the selected bitrates to the clients, respectively. On the one hand, the HTTP server running on the switch holds information of all QoS queues in a Queue Table. On the other hand, an Open vSwitch API, which is deployed to allow a flexible way to create and adjust the configuration of any queue, is also maintained in this server. *Traffic Shaping* is implemented on the Ethernet interface connected to the clients for individual queues in the total of 8 queues supported by the employed switch. However, not all of them are dedicated to video slices, as one queue implements the Arrival Slice. At each execution of the Optimization Module, the size of the *Arrival Slice* is dynamically set based on a periodically updated measure of the video traffic arrival statistics.

Experiment Scenario and Metrics. An experiment is started through a workload and lasts 900s. A workload contains the starting time generated by a Poisson arrival process with parameter λ as well as the selected video from the video catalog and the device resolution according to a discrete uniform distribution.

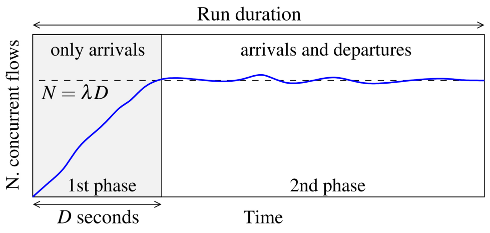


Figure 2.17: Number of concurrent video sessions over time.

Figure 2.17 displays a run which consists of two phases: the first phase lasts $D = 300$ seconds and has only flow arrivals. Thus, the number of active sessions grows with an average pace of λ. Then, during the second phase, the average arrival rate matches the average departure rate and settles to the average number of actives sessions $N = \lambda D$. As a consequence, the average bandwidth fair share for each flow is $C/(\lambda D)$ Mbps. The link load for each workload is able to set different values of λ while keeping C fixed. In our experiment, the link capacity C is set to 50 Mbps, the minimum guaranteed bandwidth b_{min} of the *Arrival Slice* equals 1000 kbps and the Optimization Module sampling T_s is set to 30s.

In order to compare the performance of the investigated strategies, we have evaluated the following metrics in each run.

RMSE. The Root Mean Squared Error is computed as the root of the average squared error between the optimal SSIM for the *n*-th user $\text{SSIM}_n^\star$, which is set by the Optimization Module, and the corresponding measured SSIM, SSIM_n

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{n=1}^{N}(\text{SSIM}_n - \text{SSIM}_n^\star)^2}.$$

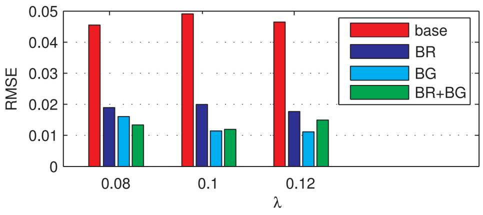
This metric measures the accuracy of the network-assisted approach in enforcing the optimal allocation according to management policy. Since we enforce a video quality fair allocation, the lower RMSE the higher the achieved fairness.

Switching Frequency. The average number of video bitrate switches in a second (measured in Hz). In [38, 39] it is shown that the switching frequency negatively affects the QoE only if it is higher than a threshold of 0.1 Hz.

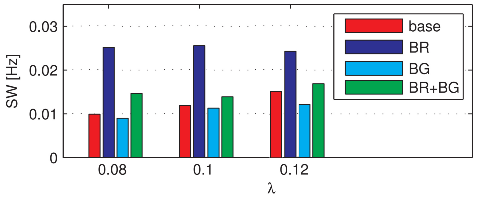
Download Rate. Describes how much data the clients download in a given time interval.

Performance of Network-assisted Approaches. The comparison of performance between different approaches, including the case in which no Video

Control Plane is involved, which is labeled as *baseline* when the client-side algorithms Elastic is employed, is presented as follows.



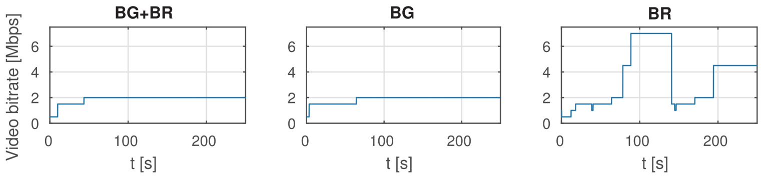
(a) RMSE.



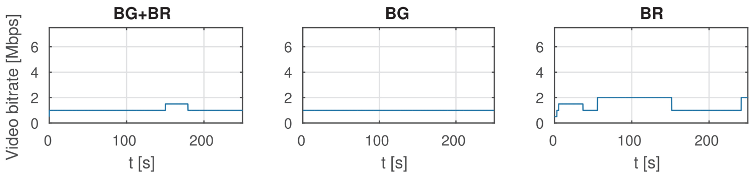
(b) Switching Frequency.

Figure 2.18: Comparison of the performance between considered network-assisted approaches as the arrival rate λ varies.

Figure 2.18a depicts the measured RMSE for several values of the arrival rate λ, as displayed on the x-axis. Three color bars represent different network-assisted approaches and one represents the case in which the VCP is disabled. In all cases, the network-assisted strategies keep the RMSE below 0.025 and achieve a lower RMSE compared to the *baseline*, which results an RMSE higher than 0.045 for all link loads. Another observation is that Bandwidth Guidance outperforms the others, however, adding bandwidth reservation to bitrate guidance (BR+BG) does not offer a clear advantage. In addition, the Switching Frequency as a function of the arrival rate λ is illustrated in Figure 2.18b. The figure shows that Bandwidth Reservation increases the Switching Frequency up to twice higher than when using BG or (BR+BG). However, it is important to note that even the highest measured frequency, i.e. 0.025 Hz does not significantly affect the perceived QoE [38].



(a) 40-th flow of the workload



(b) 45-th flow of the workload

Figure 2.19: Video bitrate dynamics of two flows of the run with $\lambda = 0.1$.

In addition to the dynamic bitrate of corresponding video sessions when using the same workload for all considered approaches, Figure 2.19 illustrates the 40-th and 45-th video session in the case of $\lambda = 0.1$. As shown in both sub-figures, while the BR strategy provoke several switches due to the client-side adaptation, the BG and (BG+BR) provoke less switches since the video bitrate is directly set by the Optimization Module. Furthermore, the isolation between flows (or virtual networks) is not violated.

2.5 Lessons Learned

In the context of Software Defined Networking, multiple virtual networks share the same physical resources, such as connectivity, memory, and storage. Hence, isolation between virtual networks or services is one of the key parameters when evaluating the performance of a system. This chapter focuses on evaluation the isolation in term of guaranteeing a granted bandwidth for a virtual network.

The combination of maximum and minimum bandwidth for an individual network is set in SDN forwarding devices (OpenFlow switches) via configuration of virtual queues on an outgoing port. SDN controller is responsible for assigning each network into a queue, therefore, traffic passing through virtual networks are separated based on features of input flow, e.g., source IP address, type of traffic (UDP/TCP), and destination application port.

The aforementioned methodology applied to several OpenFlow switches in the presence of traffic bursts helps to assess violation of the isolation between VNs as well as to determine the main influence parameters on the isolation performance. The results indicate that when congestion occurs, a virtual network might affect the bandwidth of another network in particular configuration. The influence parameters factors include characteristics of incoming traffic, configurations of virtual queues on outgoing port and flow control mechanism. In addition, vendor-specific hardware also has its own behavior, for example, HP 2920 preserves the isolation in a specified combination of burst traffic and configured rate guarantees per virtual network, however, the isolation is violated in all scenarios with Pronto P3290. In which, even in the presence of Ethernet flow control on sending devices, a violation of the isolation between VNs is possible. The overall load on the outgoing switch port also the configured rate guarantees per virtual network are defined as key influence parameters on the isolation resulting in the number of lost packets of the network that its isolation is violated.

The presented approach allows an administrator to find appropriate configurations for the involved hardware devices with respect to resource isolation before implementing the hardware into the system. It can also let the administrator to choose an applicable device for his network or special use case to guarantee bandwidth isolation.

3 Impact of the SDN Control Plane on the Data Plane Performance

The Software Defined Networking (SDN) paradigm changes several aspects regarding the operation and structure of today's networks. Instead of accessing all forwarding devices in the network individually to setup a new routing configuration, SDN allows network engineers and administrators to have a more flexible and convenient method to update this information. This is implemented by an application at an SDN controller without having to touch specific devices. The key characteristics of the resulting architecture include the separation of control and data plane as well as a logically centralized control plane. This is obtained by moving control plane functions from the network devices to a dedicated controller software running on commercial off-the-shelf (COTS) hardware. Communication between this centralized control plane and the data plane takes place via the southbound API [11], an open interface which is implemented by protocols like OpenFlow [16].

While the data plane carries out forwarding packets, the SDN control plane is responsible for exchanging signaling traffic, managing the underlying data network, and decides routing paths. To achieve these goals, the controller and OpenFlow switches exchange different types of messages during the whole time of operation such as device information, flow statistics, flow installations or removals, and port status. Such information needs to be received and processed promptly to ensure a reliable network behavior.

In addition, before moving to an SDN-based network deployment, operators need to make sure that the resulting network meets their particular requirements with respect to performance. On the one hand, when considering the data plane, these requirements might be packet forwarding algorithms, the capacity of an OpenFlow switch, which identifies the number of flows the device can handle, or the throughput that the switch can deliver. On the other hand, the control plane needs to support a high throughput at both northbound and southbound interfaces to respond quickly to changes of the network as well as a rapid response time for each message initialized from the switch [40]. However, when deploying an SDN approach, the specific requirements are use-case dependent with regards to particular scenarios. For example, in case of using OpenFlow switches in a network with low traffic, the focus is on the ability of the switch to handle small packets, rather than the size of its flow table since some devices process large packets well, but perform poorly with small packets [41]. In addition to the control plane, the performance of installing new flows might affect the operation of network devices as well as controller applications. Namely, the switch notifies the controller that it finished inserting rules, however, the rules were only installed in software and not implemented in hardware. This might lead to incorrect behavior of the controller due to inconsistent states of the network. Hence, the processing time of FlowMod messages, which indicates how fast an OpenFlow switch handles these messages to install new flows, is thoroughly investigated in this chapter concerning the control plane delay with different FlowMod message installation mechanisms. Furthermore, combinations of SDN controller and commodity switches are compared while evaluating the FlowMod processing performance.

The content of this chapter is mainly taken from [8, 10]. In the following, Section 3.1 summarizes the SDN control plane operation and related work relevant to the performance evaluation of the control plane, after focusing on different types of FlowMod-related communication schemes defined in the OpenFlow specification for adding flows in SDN networks. Then, details of the performance evaluation of these mechanisms are discussed in Section 3.2. Section 3.3 presents the results when facing the control plane delay. Finally, the lessons learned in this chapter are provided in Section 3.4.

3.1 Background and Related Work

In Section 3.1.1 an overview of the operation of the control plane in SDN networks is provided. Afterwards, two mechanisms used to proactively install flows in an OpenFlow switch as *asynchronous* and *synchronous* method are outlined in Section 3.1.2. Then, related work with regards to performance evaluation of the SDN control and data plane is discussed in Section 3.1.3.

3.1.1 Operation of the SDN Control Plane

As mentioned in Section 2.1.1, the SDN control plane is the middle layer of the SDN architecture and is responsible for managing the features of an SDN network. In this section, a particular look on this layer is presented. It includes fundamental modules, interfaces and methods to install OpenFlow rules to switches in order to implement networking features.

SDN Controller Core Modules. Figure 3.1 illustrates a general structure of an SDN controller, which insist of two communication interfaces and some basic modules, as well as applications running on top.

To implement routing decisions, manage network devices and flows, and to provide essential information like flow statistics or flow updates to applications via a northbound API, every controller needs to perform several core features. First, both end-user devices such as desktops, printers, laptops, or mobile devices and network devices (OpenFlow switches) need to be discovered. Second, the information regarding the interconnection between these devices has to be maintained and updated promptly if there is any change. Finally, a database of flows in the network is stored and synchronized timely with the network devices to ensure a reliable behavior of the network. As displayed in the middle of Figure 3.1, several modules performing these functions. For example, devices and topology discovery, flow management, device management, and statistics tracking. These modules are usually designed as internal elements in the controllers. Nevertheless, to obtain an efficient operation of the network, additional

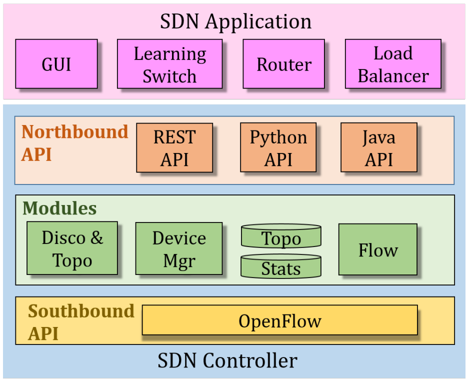


Figure 3.1: SDN controller anatomym (adapted from [42]).

modules are also deployed, e.g., routing module or traffic engineering module.

SDN Controller Northbound Interface (NBI). Programmability, which is one of the key principles of SDN, is expressed through the ability to program the behavior of a network in a flexible way and allows third-party applications to run across multi-vendor commodity hardware. NBI is known as the main driver to implement this principle. It supports APIs for accessing the network device as well as directly provides information about the status of network resources to upper applications. Network functions like path computation, security, bandwidth allocation, and routing are realized by exchanging knowledge of the physical underlying network between the applications and the SDN controller via NBI. Furthermore, NBI also supports orchestration systems such as OpenStack Neutron [1] or VMWare vCloud Director [2] to manage network services in a cloud [43].

Figure 3.2 presents how the controller interacts with applications. The NBI

[1]https://github.com/openstack/neutron
[2]https://www.vmware.com/support/pubs/vcd_pubs.html/

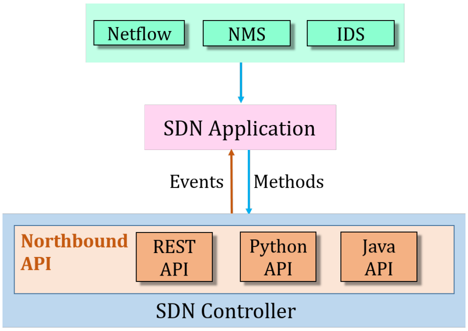


Figure 3.2: SDN controller northbound API [42].

is the intermediate component between them and is responsible for forwarding *events* that are of interest by the applications. However, not every event is necessary to collect, depending on the function of the application. Depending on the purpose of an application, it can select concrete events to query. For instance, a forwarding application needs to know information related to the network topology such as link up/down or device on/off, rather than traffic statistics. Then, the applications perform different methods to apply their algorithms or actions to respond to the received events. Moreover, external inputs from a network monitoring system or a Network Management System (NSM) can also trigger an SDN application to control the network. Nevertheless, presently, there is no common standard for NBI [44], due to the design of the applications varies depending on the requirements, each service might use a specific API to exchange essential information with the controller. Therefore, many types of NBIs are able to coexist in the same controller instance, e.g., REST API, Python API, and Java API as shown in Figure 3.2.

SDN Controller Southbound Interface (SBI). The southbound interface of SDN networks provide the communication between the controller and network devices using protocols like OpenFlow, NETCONF [45], or XMPP [46]. Among them, OpenFlow is the most popularly developed protocol for the SBI. A network device supporting OpenFlow forwards packets based on the information in a *flow table*, which consists of multiple flow table entries. Each of which contains match fields, actions, and counter as illustrated in Figure 3.3.

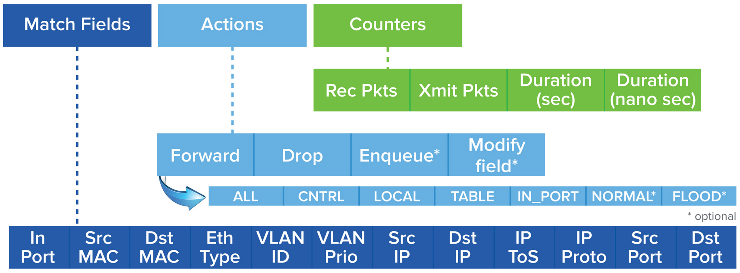


Figure 3.3: OpenFlow 1.0 Table Entry [47].

First, the *match fields* describe conditions that a packet has to meet so that a specific rule applies. These fields cover different protocols, e.g., Ethernet, IP, or layer 4 application ports. The fields can either be specific values or a wildcard match. Second, the *action* determines the way in which the packets should be processed. OpenFlow 1.0 supports a set of actions such as forwarding the packet to a given port, dropping the packet or sending it to the controller via a secure channel, etc. Third, the *counter* is responsible for collecting statistics about flows to keep track of the number of packets and bytes as well as the installation duration of each flow.

Figure 3.4 outlines the scheme of processing a packet when it arrives at an OpenFlow switch. After the packet header is parsed and compared with every flow entry in the table, if no entry matches the packet header, the header will be encapsulated and sent to the controller in a PACKET-IN message to request an instruction with the packet. Then, the controller based on its information about the network topology instructs the switch by an appropriate forwarding path

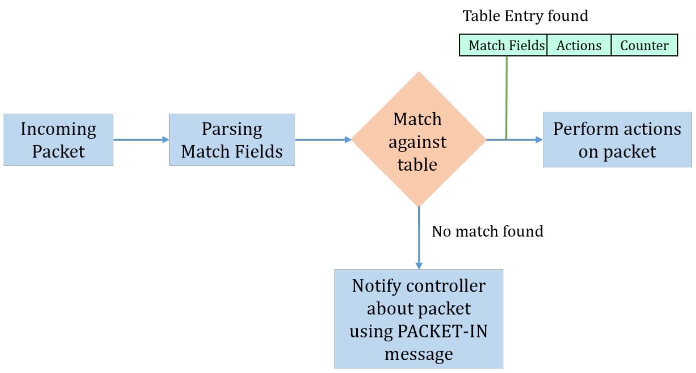


Figure 3.4: Packet Processing in an OpenFlow Switch [42].

for the packet, which is installed as a new entry. The next packets that have the same characteristics will follow this rule as well. Otherwise, if the packet matches one or several flow entries, the switch implements the action of the entry that has the highest priority, e.g., forwards the packet to an output port. Meanwhile, the counter of this entry is updated to record the statistics of the flow and inform the controller once a statistic request is inquired.

Flow Installation Approaches. To install a new rule in an SDN switch, a controller either acts according to a proactive method, or a reactive approach. On the one hand, the *reactive* method is precisely the process mentioned in the previous paragraph, which requires the controller to react to each new incoming packet, or in other words, the rules are installed on demand during run time. On the other hand, in the *proactive* approach, the flow tables in all switches are pre-populated before traffic arrives. The switches are proactively programmed with flow paths or followed a policy defined by the controller. Therefore, the traffic is forwarded without controller involvement. Both approaches have several pros and cons as listed in Table 3.1.

Table 3.1: Proactive vc. Reactive Flow Installation [48–50]

Criteria	**Proactive**	**Reactive**
Method	Controller pre-installs table entries for all possible traffic patterns	First packet of each flow triggers rule insertion by the controller
Setup time	⊕ Zero flow setup time for each flow	⊖ Each flow incurs flow setup time
Flow Table	⊖ Requires large flow table	⊕ Efficient use of flow table
Connection lost	⊕ Loss of connectivity does not disturb traffic in data plane	⊖ Loss of connectivity between controller and switch limits utility of the switch
Apply for	⊕ Good for stateless forwarding: L3 routing, static firewall	⊕ Good for stateful forwarding: L2 switching, dynamic firewall
Others	⊖ Requires aggregate rules, foreknowledge of traffic patterns	⊖ Controller might be a bottleneck when having so many request

Nevertheless, the reactive method might lead to a bottleneck at the controller due to an enormous number of PACKET-IN messages in the control plane. Therefore, in this chapter, the proactive flow installation is focused on, in which the controller sends FlowMod messages in advance to set up new rules in the switch as presented in the next Section.

3.1.2 Methods of Sending FlowMod Messages

Information exchange between systems can be performed by means of one of two paradigms: *synchronous* and *asynchronous* messaging. Asynchronous messages are passed between two entities: the sender emits multiple messages and does not wait for a response before continuing to send the next messages. In contrast the synchronous messaging, where the sender does not send a new message until it receives an acknowledgment of the previous one. In the scenario of sending FlowMod messages, the OpenFlow specification [51] defines the optional Barrier messages which can be used to perform both kinds of messaging. The desired behavior can be achieved by using Barrier messages either after each FlowMod message or after multiple FlowMod messages. When installing rules on an OpenFlow switch, an SDN controller offers implementations for those methods as described in detail in Figure 3.5.

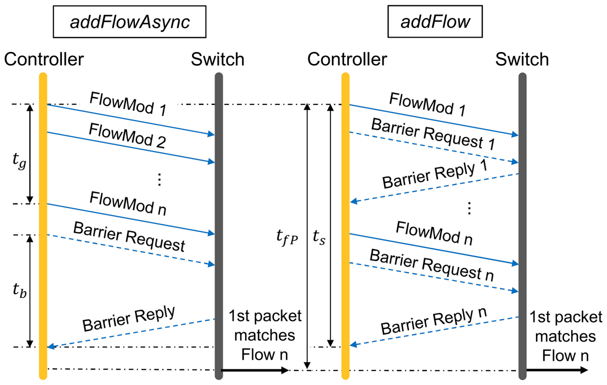


Figure 3.5: Asynchronous and synchronous methods for adding flows to an OpenFlow switch.

The time sequence diagram when applying the asynchronous method to send FlowMod messages is presented on the left hand side of Figure 3.5. The controller sets the rules by sending a batch of FlowMods and terminates the process with a Barrier Request. Whenever the switch finishes installing all the rules, it confirms this to the controller with a Barrier Reply. In this chapter, this method is referred to as *addFlowAsync* and is distinguished from *addFlow*. In the *addFlow* method, the controller always sends messages in a sequence in which a FlowMod message is followed by a Barrier Request message and waits for a Barrier Reply, before dispatching the next FlowMod. By doing this, the controller ensures that each requested rule is actually installed in the table of the switch.

Furthermore, Figure 3.5 illustrates the different components of the FlowMod processing time that are investigated in this chapter. On the most left, t_g represents the time that the controller needs to generate n FlowMod messages in the case of *addFlowAsync* and t_b is the duration between BarrierRequest and BarrierReply. The time between the first FlowMod and the last BarrierReply indicates how long it takes the switch to finish setting up n rules and is denoted as t_s in both cases. Finally, t_{fP} denotes the time difference between the first FlowMod message and the first data plane packet that is forwarded by the switch according to the last FlowMod it received as depicted on the right side of Figure 3.5. This verifies that the corresponding flow entry is actually installed in the data plane of the switch.

These parameters are measured and analyzed in detail in Section 3.2 to define the impact of different FlowMod installation mechanisms regarding their processing time in the SDN control plane. The measurements are implemented by using several switches with these mechanisms characterized by different degrees of accuracy, cost, complexity, and the capability of performing measurements at run time.

3.1.3 Performance Evaluation of SDN Components

This section covers two main areas of related work. On the one hand, approaches for evaluating the performance of different aspects and components of an SDN architecture are presented. On the other hand, an overview of mechanisms for identifying and addressing the heterogeneity of switches in the SDN data plane is provided.

Possibilities for testing the network as a whole in the context of SDN are discussed in [52]. While integrated tests are a long term goal, it is necessary to understand the behavior of the individual network elements, i.e., controllers and switches in case of SDN. In an effort to provide means to test the switch behavior with respect to compliance with the OpenFlow protocol specification, the authors of [53] present the OFTest suite. In contrast to this work, those studies focus on function tests rather than performance tests.

The study conducted in [54] features a dedicated hardware traffic generator in order to test the data plane performance of Linux-based OpenFlow switching. In a similar setup, the authors of [55] investigate the characteristics of virtual switches and underlying virtualization techniques. In both works, the main interest lies in the data plane performance of the different switch implementations. This differ to this work, on the other hand, investigates different measurement mechanisms for the control plane performance of OpenFlow-enabled switches and provides a first step towards classifying these mechanisms according to criteria like accuracy and complexity.

In [56] presents comparisons the control plane performance between three of the most common SDN controllers: Opendaylight Beryllium[3], ONOS Falcon[4], and Floodlight 1.2[5]. The tests setup 100k flows in Mininet[6] environment, then restart the network and wait until all flows are confirmed being re-programmed. Instead of using an emulation tool (Mininet) to emulate OpenFlow switches, our work implements testbeds with physical devices from different vendors, to probe whether vendors variation has an impact on the investigated times. Furthermore, not only open-source controllers but also a commercial controller are

[3]`https://www.opendaylight.org/what-we-do/current-release/beryllium`

[4]`https://wiki.onosproject.org/display/ONOS/Release+notes+-+Falcon+1.5.0`

[5]`https://floodlight.atlassian.net/wiki/spaces/floodlightcontroller/pages/24805419/Floodlight+v1.2`

[6]`http://mininet.org/`

considered in our work.

Kuzniar *et al.* [57] investigated different hardware switches to identify the characteristics of the interaction between the SDN control and data plane, e.g., rule installation latency or the accuracy of installed rules confirmation. The work discovers that the switches might send the confirmation of flows installation, even before the flows have really been applied in the data plane. Similar interest regarding the flow updating rate, OFLOPS [58] is a software framework for testing OpenFlow switch performance in the data plane as well as in the control plane. Its extension, OFLOPS-Turbo [59] is capable of 10 GbE traffic generation and utilizes the open-source NetFPGA-based OSNT [60] traffic generator and capture system. Focusing on the processing of FlowMod messages, our work aims at comparing different mechanisms for switch performance evaluation and identifying their trade-offs.

Furthermore, analytical approaches like [61] and [62] investigate the influence of various network parameters on the performance of an OpenFlow architecture. Since such models are often based on measurements, the accuracy of these measurements also positively affects the quality of the resulting models. Therefore, one key aspect of our analyses is the accuracy of the available measurement mechanisms. A methodology for assessing the accuracy of measurements in the SDN context is presented in [63]. In addition to measurements performed by an SDN controller module, wiretaps installed at both ends of a communication channel serve as a means of providing the ground truth. This technique is also applied in the experiments that are conducted during the course of this work.

Several previous works highlight the heterogeneity of SDN switch hardware in terms of functionality, performance, and OpenFlow protocol compliance [64, 65]. Unexpected or unreliable behavior such as additional delays and inconsistency between control and data plane pose several risks with respect to security as well as correct forwarding behavior. Hence, this heterogeneity needs to be taken into account for proper planning and design of real world deployments.

Some aspects of the heterogeneity, e.g., OpenFlow protocol compliance, are addressed by approaches such as TableVisor [66] and FlowConvertor [67] that introduce abstraction layers to translate given OpenFlow messages to device specific directives that take into account the behavior of individual switch hardware. While their focus is on maintaining functional homogeneity, we address the performance aspect. Finally, methods for data plane verification and consistency checks between data and control plane are proposed in [68]. However, rather than focusing on the identification of faulty switches, we are interested in performance prediction.

3.2 Proactive Flow Installation Evaluation

This section focuses on the performance evaluation of the FlowMod processing time when installing OpenFlow rules following *proactive method* as mentioned in Section 3.1.2. Beginning with an overview of testbed setup including the hardware used, topology, and the course of the experiment is described. Afterwards, the evaluation results in the scenario without any additional delay in the SDN control plane are presented. First, a comparison of the behavior of different switches is presented by analyzing various components of the FlowMod processing time that are recorded at the wiretaps. Second, an in-depth evaluation of the accuracy levels achieved by the different measurement methods demonstrates their feasibility in the tested scenarios. Finally, recommendations regarding the choice of approaches are derived from the aggregated measurement results.

3.2.1 Experimental Setup

There are several options for assessing the FlowMod performance of an OpenFlow switch in terms of the aforementioned components of the processing time. Three such approaches are discussed as follows.

First, a purely software-based solution using the OpenDaylight controller is analyzed. In this context, the Hydrogen release[7] of the OpenDaylight controller is installed on a PC[8] which uses the Ubuntu 12.04 operating system. While the controller is attached to the OpenFlow switch under test and generates control plane traffic, i.e., FlowMod and Barrier messages, the Iperf[9] software running on an additional server is used to send UDP traffic to the switch's data plane. Finally, the traffic sink waits for the arrival of matched packets. In the context of the OpenDaylight controller, there are two options for obtaining the desired components of the FlowMod processing time. On the one hand, traffic captures recorded by the *tcpdump* tool running on the machine that hosts the controller can be used to calculate t_g, t_b, and t_s. On the other hand, it is possible to implement a Java module for the controller which intercepts the timestamps of events like sending FlowMods or receiving Barrier messages in order to derive the aforementioned times. Due to the use of concurrency within the implementation of these methods, however, not all timestamp-based measurements are as accurate as the ones obtained via tcpdump.

The second approach utilizes a Spirent C1[10], an Ethernet testing platform which allows generating traffic according to different protocols including OpenFlow. This device has four 1GbE interfaces, three of which emulate the controller, traffic source, and traffic sink, respectively. FlowMod messages are sent to the management port of the switch. Simultaneously, the traffic source keeps sending traffic to the traffic sink. The Spirent Test Center software (STC) provides means to measure traffic characteristics like packet delay, including values corresponding to t_g, t_b, t_s, and t_{fP}. In addition to the result database generated by the STC, the Spirent C1 also allows directly recording packet traces at each individual port. Using these captures, it is possible to determine the components of the FlowMod processing time as well.

[7] `https://www.opendaylight.org/software/downloads/hydrogen-base-10`

[8] Intel(R) Core(TM)2 Duo CPU E8500/4G RAM

[9] `https://iperf.fr/`

[10] `http://www.spirent.com/Test-solutions_datasheets/Broadband/PAB/Spirent_TestCenter/STC_C1-Appliance_Datasheet`

As mentioned in Section 3.1.3, the OFLOPS framework provides means for measuring FlowMod processing times as well. The framework is capable of measuring FlowMod processing times according to [58] which are displayed in Table 3.2, together with the two mentioned tools. Additionally, two wiretap devices are used to accurately capture the processing times which are used as ground truth.

Table 3.2: Comparison of measurement mechanisms.

Time	**addFlowAsync**			**addFlow**		
	STC	ODLM	OFLOPS	STC	ODLM	OFLOPS
t_g	✓	✓	✓	N.A.		
t_b	✓	✓	✓			
t_s	✓	✓	✓	✓	✓	✓
t_{fP}	✓	✗	✓	✓	✗	✓

In order to evaluate the performance of an OpenFlow switch with respect to the processing time of FlowMod messages, the following scenario is implemented in a testbed. The testbed is set up according to recommendations for testing OpenFlow performance published by Spirent[11] and is shown in Figure 4.12.

Starting with an empty OpenFlow table in the switch, the controller sends a FlodMod message that installs a rule with the lowest priority to drop all packets in order to avoid forwarding unmatched packets to the controller. Afterwards, the controller starts sending a batch of FlowMods to install these rules according to one of the two methods that were presented in Section 3.1.2. Simultaneously, the traffic source generates data plane traffic that matches the last rule. Finally, the results of the experiment are extracted from the reports that are generated by the OpenDaylight controller or Spirent Test Center, allowing to evaluate how fast an OpenFlow switch processes FlowMod messages. The results are validated

[11]`http://www.spirent.com/~/media/White%20Papers/Broadband/PAB/OpenFlow_Performance_Testing_WhitePaper.pdf`

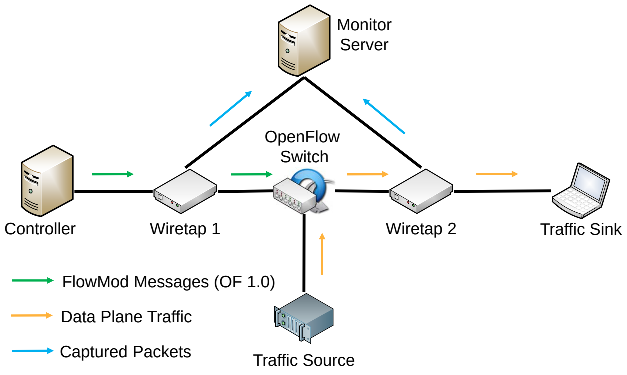


Figure 3.6: Logical testbed setup.

by using tap devices to capture transmitted packets and accurately calculate the latency between the first FlowMod and the arrival of the first packet which matched the last rule.

Two Net Optics tap devices[12] are installed before and after the OpenFlow switch in order to mirror both control plane traffic to and from the switch as well as data plane traffic to the traffic sink. The monitor server HP Proliant DL32 uses two Endace DAG 7.5G2 capture cards to capture every incoming packet. Based on their time stamps, it is possible to calculate all components of the FlowMod processing time as well as t_{fP} for validating the installation of OpenFlow rules. On the data plane, the traffic source sends UDP traffic with its IP as source IP address and the destination IP address corresponding to the traffic sink. This traffic can not reach the traffic sink until the last FlowMod, which has the matching fields regarding source and destination IP addresses, is installed. The experiments are implemented using three OpenFlow switches from different vendors, focusing on an evaluation of the installation speed of OpenFlow rules. The spec-

[12]`http://www.ixiacom.com/products/ixia-gig-zero-delay-tap/`

ifications of the switches under test are summarized in Table 3.3.

Table 3.3: Switches used in this work.

Switch	CPU	Memory	Software
Pronto 3290	MPC8541 825 MHz	512MB	PicOs 2.0.14 (Open vSwitch v1.10.0)
Quanta T1048	MPC8541 825 MHz	1024MB	PicOs 2.6 (Open vSwitch v2.3.0)
NEC PF5240	PowerPC 667 MHz	1024 MB	OS-F3PA v5.0.0.1

The flow table of the NEC PF5240 has a limit of 2816 entries and the Quanta switch often exhibits unexpected behavior in the context of generating more than 2000 flow table entries. Hence, the number of FlowMod messages used in the experiments ranges from 2 to 1800 messages. Each run is repeated 10 times in order to obtain 90% confidence intervals.

3.2.2 Comparison of Switch Behavior

Figure 3.7 shows different components of the FlowMod processing time based on measurements performed at the wiretaps while using different numbers of FlowMod messages and different switches. In particular, the time between the first FlowMod and the last Barrier Reply, t_s, and the time until receiving the first packet on the data plane, t_{fP}, are presented. On the x-axis, the number of flows is displayed. According to the limitations of the different switches under test, this number is varied between 10 and 1800. Differently colored and shaped curves denote different switch models and time components, respectively. The y-axis represents the processing time of the corresponding parameter combination. Additionally, error bars provide 90 % confidence intervals obtained by repeating each experiment 10 times. For the presented measurements, FlowMod messages were generated with the OpenDaylight controller using the *addFlowAsync* method.

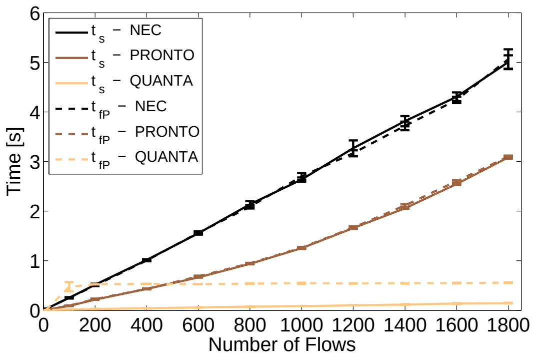


Figure 3.7: Different switch behavior measured at the wiretaps.

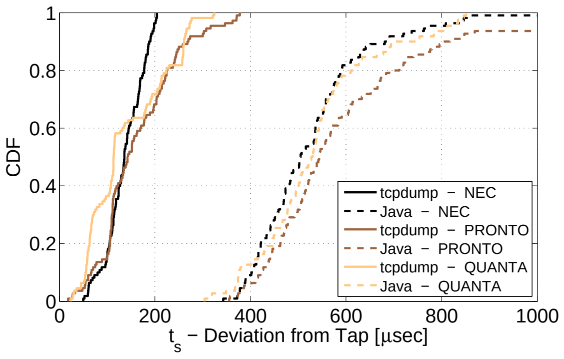
While the NEC switch exhibits the longest processing times of up to almost 5 seconds in case of installing 1800 flows, it is the only device for which the relationship $t_s > t_{fP}$ holds throughout all experiments. Thus, when the controller receives the Barrier Reply message, the switch's flow table update is already finished, resulting in a consistent state between the switch and the controller. For the Pronto and Quanta switches, on the other hand, this relationship is reversed: after receiving the Barrier Reply message, the controller expects the flow installation to be completed, although it is still being processed in the switches. Such an inconsistency could potentially cause unexpected behavior, e.g., when the controller uses its northbound API to communicate the seemingly finished update to an application that starts its transmission immediately. While the extent of this deviation is in the order of magnitude of 50 ms in case of Pronto, it is increased to roughly 400 ms in the case of the Quanta switch. On the other hand, the Quanta switch consistently outperforms the other two models in terms of t_s and t_{fP} by a significant margin as soon as more than 600 flows are installed.

These results demonstrate that each switch model might have its own characteristic behavior. Hence, network operators need to evaluate the performance of hardware devices before deploying them in the network in order to ensure a reliable network behavior. In the following section, the accuracy of different approaches for measuring switch characteristics is evaluated.

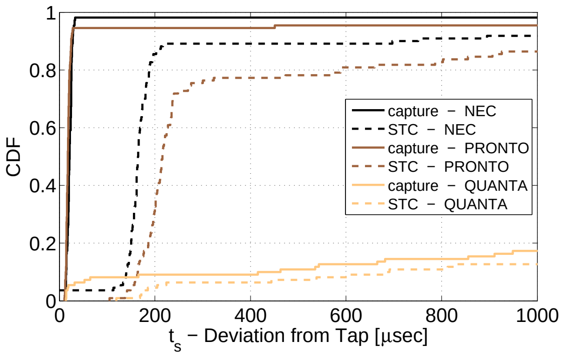
3.2.3 Accuracy Assessment of Measurement Mechanisms

The setup time t_s is an important performance indicator for OpenFlow switches since it usually constitutes the majority of the FlowMod processing time. Additionally, in case of the OpenDaylight Java module, it is possible to measure the value of this parameter during runtime and to integrate it into the controller's feedback loop. Therefore, Figure 3.8 displays cumulative distributions of the accuracy of the t_s measurements for the two measurement tools OpenDaylight and Spirent C1. In this context, the accuracy refers to the difference between the value recorded by the measurement tool and the corresponding wiretap which is considered to be the ground truth. There is also a distinction between the results achieved from different measurement probes, i.e., Java and tcpdump in case of OpenDaylight and Spirent Test Center and port-based captures in case of the Spirent C1.

Figure 3.8a shows the accuracy of the measurement probes available for the OpenDaylight controller. While the accuracy in microseconds is displayed on the x-axis, the y-axis denotes the fraction of measurements that are below a particular accuracy threshold. Different line colors represent different switches and the line shapes correspond to the measurement probes. In case of OpenDaylight, curves corresponding to different probes form two groups of which their values differ only marginally from switch to switch. The first group represents the tcpdump measurements and features values that are mainly in the range between 100 and 300 μsec. The values reported by the Java module inside the controller are higher, with values ranging from 400 μsec to 1 ms. This behavior is consistent with the measurement setup: since the Barrier Reply message from the switch first passes the controller's network interface before arriving in the user space Java application. Thus, the time measured by the latter is higher than



(a) OpenDaylight module.



(b) Spirent C1.

Figure 3.8: Distribution of the accuracy of the t_s measurement when using different measurement tools and probes.

in case of tcpdump.

Like the previous figure, the curves in Figure 3.8 form groups according to the measurement probe. While the port-based capture provides measurements that are accurate up to an order of magnitude of roughly 30 μsec in most cases, the reports generated via the Spirent Test Center show a difference of around 200 μsec as well as significantly more outliers in terms of accuracy. When using the Spirent C1 in conjunction with the Quanta switch, irregular behavior is observed regardless of the number of installed flows. Since the Spirent C1 is a proprietary closed-source system, this phenomenon can not be investigated in detail.

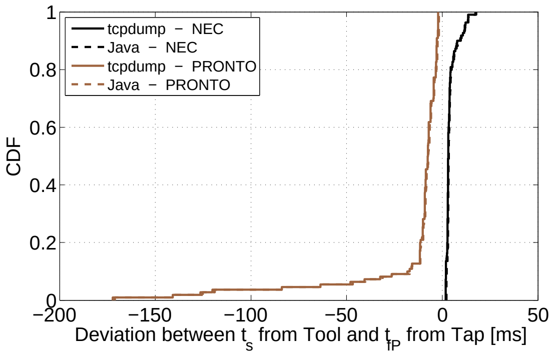


Figure 3.9: Distribution of $\delta = t_s^{tool} - t_{fP}^{tap}$.

While Figure 3.7 shows that t_s and t_{fP} are almost identical for the majority of switch models, Figure 3.9 presents a quantitative view on the observed deviation between tool-based measurements of the setup time t_s^{tool} and the time until the first packet t_{fP}^{tap} as reported by the wiretap. Values on the x-axis represent the difference $\delta = t_s^{tool} - t_{fP}^{tap}$ and the y-axis indicates the fraction of measured values below each threshold. Due to the irregularities discussed in the previous paragraphs, measurements regarding the Quanta switch are omitted for the sake of readability. As reported before, the tcpdump and Java probes provide very similar data, with tcpdump being slightly more accurate. Additionally, the figure allows deriving guidelines for determining the time until a set of Flow-

Mod messages are actually processed in the data plane of a particular switch model. For example, $\delta > 0$ in case of the NEC switch implies that the Barrier Reply is always sent after the rule is installed in the switch. Hence, the controller receives information that corresponds to the switch's data plane. In contrast, the minimum value of $\delta = -170$ ms in case of the Pronto switch means that up to 170 ms may pass before the controller and switch are in a synchronized state. Information regarding these delays can be obtained by performing such measurements before actually deploying the switches in a network, hence allowing the operator to incorporate them into the northbound and southbound APIs of the SDN controller and increase the reliability of the information exchange in the network.

3.2.4 Correlation Analysis of Measurement Mechanisms

In order to provide an aggregated overview of the results, Table 3.4 shows the correlations between the tool-based measurements and the ground truth according to the wiretap devices. Previous results indicate that in case of the Spirent C1, the port-based captures provide a higher level of accuracy. In the context of the OpenDaylight controller, the Java module is more relevant in a practical context due to its capability to perform measurements during/at runtime. Hence, only these probes are included in the table. The correlation is determined according to Spearman's rank correlation coefficient [69].

In addition to the correlation between measurements of the same component of the FlowMod processing time, the table also provides information on the relationship between t_s measured via the tools and t_{fP} measured via the wiretap. A high degree of correlation in this context implies that it is possible to reliably predict the time until the requested FlowMod messages are installed in the switch's data plane when the t_s measurements are given.

For the NEC and Pronto switches, all correlations are above 99 %. This behavior is in line with previous observations which show deviations in the order of magnitude of less than 1 ms for values that are as high as multiple seconds. In

Table 3.4: Correlations between measurements from different tools and the wiretap-based ground truth.

Mode	addFlowAsync				addFlow	
Tool	ODL/Java		STC/Capture		ODL/Java	STC/Capture
Pair / Switch	(t_s^{tap}, t_s^{tool})	$(t_{fP}^{tap}, t_s^{tool})$	(t_b^{tap}, t_b^{tool})	(t_s^{tap}, t_s^{tool})	$(t_{fP}^{tap}, t_s^{tool})$	$(t_{fP}^{tap}, t_s^{tool})$
NEC	1.0000	0.9999	0.9983	1.0000	0.9999	0.9988
Pronto	0.9999	0.9980	0.9958	0.9992	0.9994	0.9989
Quanta	0.9999	0.8146	0.8790	0.9984	0.9396	0.3727

case of the Quanta switch, however, there is no significant correlation between t_s and t_{fP}. As observed in Figure 3.7, the time until the first data plane packet is matched in the switch is nearly constant while the setup time increases when the number of installed flows is increased.

3.3 Influence of Control Plane Delay on Proactive Flow Installation

In this section, we present the results of the experiments that is similar to the setup in Section 3.2.1, however, an additional computer with two 1 Gbps Network Interface Cards (NICs) running Ubuntu 16.04 is inserted between SDN controller and OpenFlow switch in order to generate delays in the SDN control plane. It emulates the transmission delay in both directions, i.e., from the switch to the controller and vice versa. The current testbed is setup as detailed in Figure 3.10. The red lines indicate links with delay, which is set via the tc command[13]. One additional controller is investigated in this work - the latest version

[13]`sudo tc qdisc add dev [interface] root netem delay [delayValue]`

of the Python-based Ryu controller[14] v4.18 is used in conjunction with an additional module that allows the generation of FlowMod messages according to the two methods mentioned in Section 3.1.2.

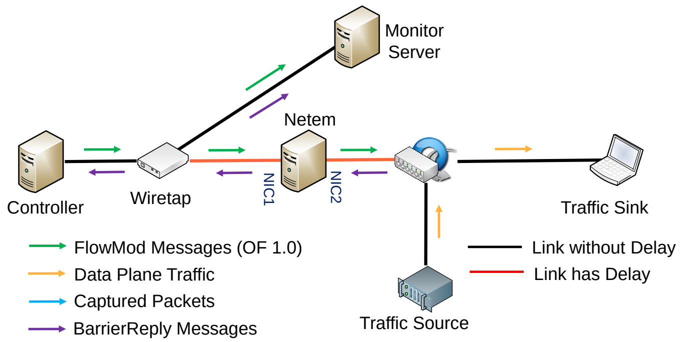


Figure 3.10: Logical testbed setup when adding the control plane delay.

In the following, first, we demonstrate the heterogeneity of the switch hardware. This is achieved by comparing the FlowMod processing times of different switches when installing different numbers of flows and applying different amounts of control plane delay. Afterwards, we show that using prior information on the hardware specific characteristics and controller-based delay measurements, it is possible to achieve a high degree of correlation between the flow setup time, t_s, and the time until flow rules are active in the data plane, t_{fP}. This outcome highlights that reliable estimations of t_{fP} are possible at run time. Finally, we present results regarding the impact of the controller implementation on the FlowMod processing time.

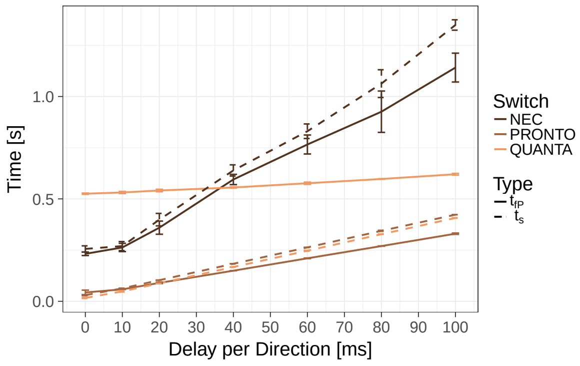
[14]http:https://osrg.github.io/ryu/

3.3.1 Sensitivity of Switches towards Control Plane Delay

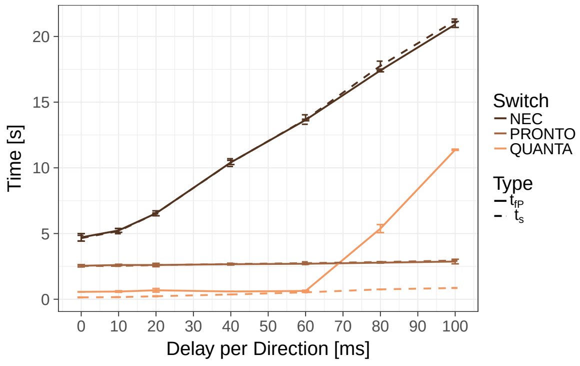
The two graphs of Figure 3.11 highlight the individual behavior of the three switches that are used in this work with respect to their sensitivity to parameters such as the amount of control plane delay and the number of flows that are installed. Their x-axes represent the control plane delay that is set in each direction between switch and controller, i.e., a value of 10 ms corresponds to a round trip time of 20 ms. The y-axes denote the flow setup times t_s and t_{fP} that are recorded by means of the wiretap devices and are represented by dashed and solid lines, respectively. Finally, differently colored curves correspond to different switches. For each parameter combination, five experiment runs are performed in order to construct 95 % confidence intervals that are indicated by the error bars. The results in the figures are based on measurements with the OpenDaylight controller. Experiments with the other two controllers yield qualitatively similar results and are discussed in Section 3.3.2.

Figure 3.11a displays results from experiments in which a total of 100 FlowMod messages are sent to the switch via the *addFlowAsync* mechanism, i.e., 100 FlowMods are followed by one pair of BarrierRequest and BarrierReply messages. Three observations can be made. First, the three switches operate at different time scales. With processing times that are lower than 500 ms for all delay values, the Pronto switch consistently outperforms the other two switches in this scenario. Second, the sensitivity of the switches towards the control plane delay varies significantly, as indicated by the different slopes of the individual curves. Consequently, the NEC switch achieves lower values of t_{fP} than the Quanta in scenarios with a low delay, whereas the Quanta switch is least affected by the increasing delay and gives better results for delays that are larger than 40 ms. Third, while the NEC and Pronto switch send their barrier replies after having installed all flow rules into the data plane, i.e., $t_s > t_{fP}$, the Quanta switch sends out the confirmation before the rules are active. Hence, a window of inconsistency of up to half a second can occur if the controller is unaware of this behavior.

Increasing the number of installed flows to 1800 exposes additional dif-



(a) 100 flows.



(b) 1800 flows.

Figure 3.11: Influence of the control plane delay on the FlowMod processing time when using different switches and different numbers of flows. Scenario details: OpenDaylight controller and addFlowAsync mechanism.

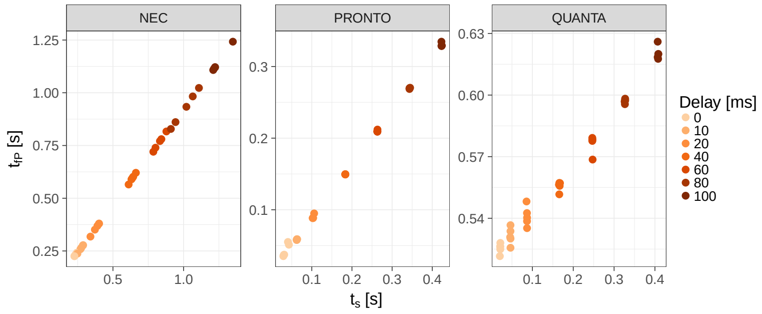
ferences between the switches. The corresponding results are shown in Figure 3.11b. For all switches, the increased number of FlowMod messages that need to be processed results in larger setup times. Furthermore, the high delay sensitivity of the NEC switch is even more pronounced in this scenario, with setup times ranging from 5 to over 20 seconds. In the case of the Quanta switch, a significant increase of the installation time is observed for delay values larger than 60 ms. Combined with the premature barrier reply message, this can be a major threat to the state of consistency. Only the Pronto switch is able to maintain nearly constant t_s and t_{fP} values for all delay settings.

While the wiretap-based measurements that are presented in the previous figures demonstrate the differences between the hardware switches, there is a high pairwise similarity between t_s and t_{fP} values. We use this relationship to derive a mechanism that can be used to infer t_{fP} from information regarding the particular switch model that is in use and measurements at the controller. These measurements include tcpdump on the controller machine to obtain t_s and a simple round trip time measurement like ping to determine the control plane delay.

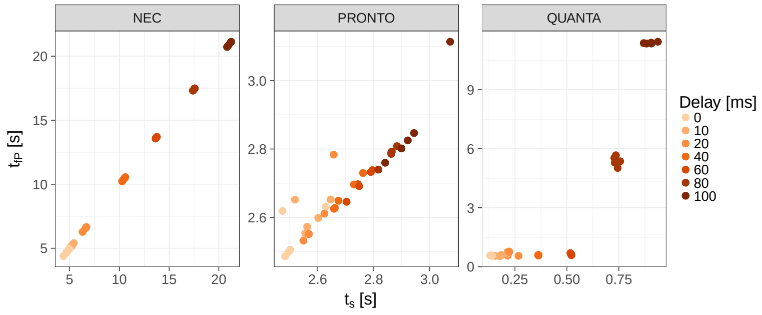
For each of the three switches, the graphs in Figure 3.12 show the measurement of the flow setup time t_s at the controller on the x-axis and the actual time until the first data plane packet t_{fP} at the wiretap on the y-axis. Differently colored dots denote different control plane delays.

In Figure 3.12a, results that are obtained when installing 100 flow rules are displayed. Although the times that are recorded for the three switches have significantly different ranges, a high linear correlation between t_s and t_{fP} can be observed. Hence, using switch-specific information regarding its sensitivity towards control plane delay in conjunction with round trip time and t_s measurements is sufficient for an accurate estimation of the flow installation time in the data plane.

Similar results are obtained in case of the installation of 1800 flow rules, as presented in Figure 3.12b. While the NEC switch has the highest setup and processing times, it also has the most consistent behavior and an almost perfect linear correlation. Except for few outliers, the Pronto switch also shows a high degree of correlation, even with the increased number of flows. Finally, the Quanta



(a) 100 flows.



(b) 1800 flows.

Figure 3.12: Flow setup time t_s recorded at the controller and time to first data plane packet t_{fP} recorded via the wiretap for the three different switches. Scenario details: OpenDaylight controller and addFlowAsync mechanism.

switch produces outliers for high control plane delays. Nevertheless, this behavior is observed consistently - qualitatively as well as quantitatively - in multiple repetitions of our experiments, as indicated by clusters of dots in the scatter plot. Therefore, this switch-specific characteristic can also be taken into account by the controller when making predictions regarding the data plane state.

Summarizing, our findings show that using simple controller-based measurements in combination with switch properties, which can be determined prior to deployment, can be used for performing an accurate prediction of the FlowMod installation time in the data plane of OpenFlow switches.

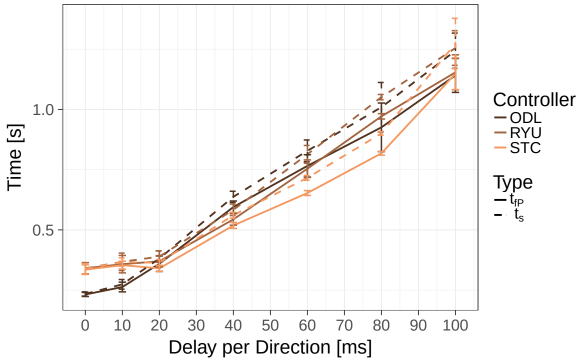
3.3.2 Impact of Controller Implementation

While the previously shown results focus on the peculiarities of different data plane hardware, this section is devoted to the influence of the controller implementation on the performance. To this end, experiments with the NEC switch are conducted with three different controllers. These include the Java-based OpenDaylight controller, the Python-based Ryu controller, as well as the controller implementation that is provided by the OpenFlow Testing Package of the Spirent C1. In case of the OpenDaylight and Ryu controller, the same host machine is used to ensure that the results are not affected by a heterogeneity of the underlying hardware. The graphics in Figure 3.13 show t_s and t_{fP} values for different numbers of flows on the y-axis and the control plane delay on the x-axis. Differently colored lines correspond to the three controllers.

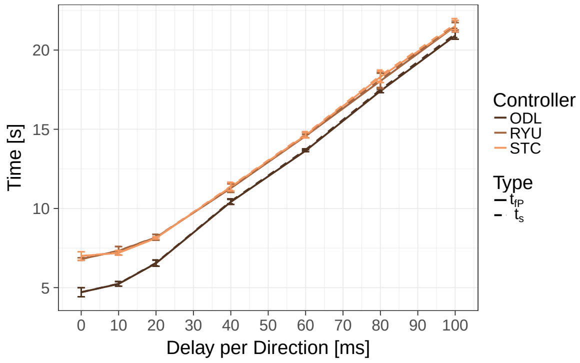
When 100 flows are installed, the majority of confidence intervals in Figure 3.13a overlap. This indicates that for control plane delays that are larger than 10 ms, no statistically significant difference between the three controllers can be identified. In the case of control plane delays that are lower than 20 ms, using the OpenDaylight controller leads to setup times of roughly 0.23 seconds as opposed to setup times of roughly 0.34 seconds that are observed for Ryu and the Spirent-based controller.

These phenomena are even more pronounced in the case of 1800 flows. Figure 3.13b shows that using the OpenDaylight controller leads to consistently faster flow setup times than Ryu and Spirent. Differences between 1 and 2 seconds are observed for setup times that range between 4.7 and 21.6 seconds.

The aforementioned results indicate that the controller is not merely a generator of FlowMod messages but can also affect the performance. In-depth analyses



(a) 100 flows.



(b) 1800 flows.

Figure 3.13: Impact of the controller choice on flow setup times for different numbers of installed flows. Configuration details: NEC switch and addFlowAsync mechanism.

of the corresponding packet dumps show that the sending behavior of the OpenDaylight controller and the corresponding packetization of OpenFlow messages differs from the other two controllers. Hence, controller developers should be aware of such mechanisms in order to adapt to switch capabilities and opportunities to improve the overall performance.

3.4 Lessons Learned

In this chapter the impact of the SDN control plane on the performance of SDN switches and the operation of SDN controllers is investigated with different combinations of SDN controllers and switches. The indicator parameters are several components of the processing time of FlowMod message when installing rules proactively in OpenFlow switches with different approaches. Also, the accuracy referring to the difference between the value recorded by the measurement approaches and the corresponding the ground truth obtained by Wiretap devices [63] is analyzed. Furthermore, delay in the SDN control plane is also taken into account and evaluated to infer the reliably of FlowMod processing times.

To evaluate the performance of OpenFlow switches with respect to the processing time FlowMod messages, two approaches are presented. These approaches are characterized by different degrees of accuracy, cost, complexity and the capability of performing measurement at run time. The first method is a software-based approach featuring a module for an open-source controller such as OpenDaylight implemented in Java or Ryu programmed in Python. The module generates a number of FlowMod messages and sends these to the switches by performing two mechanisms, i.e., while the *asynchronous* mechanism establishes a batch of FlowMod messages which is followed by a single Barrier-Request to ask the switches to confirm flow installation, the *synchronous* mechanism continuously generates pairs of FlowMod and Barrier-Request. The processing times are calculated by a Java/Python function itself as well as extracting the relevant data from capture files recorded inside the controller and from another external monitoring server. In addition to the benefits regarding costs and ease of use, the software-based controller module provides the capability to run during normal operation. This enables features like switch performance monitoring at low costs in terms of resource overhead.

The second approach is based on the commercial Spirent C1 dedicated testing platform. Similar to the Java-based approach, OpenFlow rules are installed proactively in a similar manner as aforementioned mechanisms. The Spirent Test Center application records multiple parameters related to the performance of a switch to a database such as the number of active flows, flow setup times, or barrier reply/response times. Furthermore, the application provides several OpenFlow testing for both OpenFlow controller and OpenFlow switch, e.g., table capacity testing, pipeline processing performance, or packet in/out performance. However, in this chapter, the focus lies on the FlowMod performance and its implementation. The results show that this approach has a smaller distinction between results from this approach and the measurement probe (port-based captures) than case above (for example Java module and tcpdump).

The two approaches are evaluated with respect to the accuracy of their measurements of several components of the FlowMod processing time. Moreover, they represent different classes of mechanisms that are available to network operators who need to make sure that a planned SDN deployment meets the requirements of a particular use case. The aspect of switch performance evaluation is especially relevant since the measurements show significant differences between different switch models. Results of the experiments with three switches from three different vendors show that both approaches achieve an accuracy in the sub-millisecond range when compared to measurements performed with dedicated capture cards at wiretaps. Except for one switch model - the Quanta T1048 - with unexpected behavior, the mechanisms achieve similar accuracy levels independent of the device under test. Furthermore, high correlations between measurements at the tools and the wiretaps indicate that measured values can be used to derive performance metrics even more accurately. Especially for the NEC and Pronto switches, all the correlations are above 99%. Additionally, during flow installation time, the NEC switch always inform the controller after the rules are applied, and packets on the data plane are forwarded. The Pronto switch, however, sends the confirmation up to 170 ms before the rules are activated.

Finally, varying control plane delays are considered by inserting a network emulation to add delays. This includes not only varying processing times but also different degrees of sensitivity towards the control plane delay between the controller and the switch. In addition, switch-specific characteristics that can be extracted prior to deployment can be used in conjunction with simple measurements at the controller in order to accurately predict the data plane state and performance of switches. Such a prediction mechanism can significantly reduce the window of inconsistency between an SDN controller and the switches it manages. Consequently, the implementation details of the SDN controller can also have an impact on the overall FlowMod processing performance due to sender-side behavior. This leads to optimization potential that can be taken into account by both, controller developers who want to improve the general performance of their controller as well as network operators who want to maximize compatibility and reliability of the components in their network.

4 Flow Monitoring Approaches in SDN Networks

Network monitoring refers to activities that overseeing the operation of a computer network as well as examining the status of the network. Namely, it can verify whether network components such as servers, switches, and routers have any problem or not, as well as observing the performance of networks like bandwidth usage or network traffic. Doing this provides useful information to operators in several aspects. First, as the knowledge of the system is regularly collected, troubleshooting can be performed quickly to prevent the system from unpredictable problems. The time needed to diagnose problems might be reduced significantly because of a better understanding what is going on in the network as well as timely issue alerts. Second, the security and privacy of data and system can be improved by detecting abnormal behavior based on information about file access, user activities, or network intrusion logs. Finally, monitoring tools collect network performance information like traffic on specific devices, utilization of individual servers, or response times of services. Such metrics [70] are exemplified as in Figure 4.1, they can be leveraged to analyze network state, make forecasts of the environment, or become the inputs of a traffic engineering tool.

Within a legacy network, traffic monitoring is utilized either by *active* or *passive* methods or a combination of them. While *active monitoring* [71] injects traffic and tracks these packets, which might impact the performance of the network, *passive monitoring* [72] requires additional hardware like TAP (Test Access Point) device or a specific Switch Port Analyzer (SPAN) port on the de-

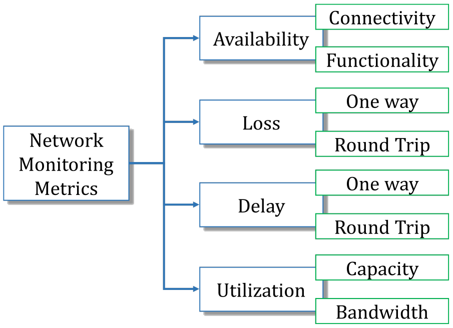


Figure 4.1: Network Monitoring metrics. [70]

vice. In these cases, data plane traffic is forwarded to another machine, whole data when using TAP or a part of the traffic in case of SPAN. On the one hand, TAP requires dedicated hardware added into the network. On the other hand, the performance of the switch might be affected due to the exceeded utilization of the link connected to the SPAN port. In contrast, in an SDN network, the traffic in a network is monitored without mirroring data plane traffic or tracking probe packets. Instead, an SDN controller collects the information of the traffic by utilizing communication with SDN switches via OpenFlow protocol. In this way, the monitoring traffic does not adversely impact the data passing through underlying hardware as well as does not need any additional sophisticated device.

Among the aforementioned parameters, the focus in this chapter is traffic monitoring at flow-level. A flow in the context of SDN is a series of packets which are defined based on common characteristics of those packets, e.g., layer 2 addresses, protocols, or VLAN tags. The traffic in SDN is an aggregation of multiple flows, hence, keeping track of the flows provides the state of exchanged data between devices. Furthermore, flow monitoring allows ensuring Quality of Service (QoS) for individual applications and services due to its capacity to provide the details of counters for ports, tables, and queues of the flows in all OpenFlow switches. This also helps to resolve intermittent network performance problems regarding bandwidth utilization or traffic congestion.

Monitoring flows in SDN mainly relies on the operation of *OpenFlow statistics* messages, an SDN controller gathers information of every flow by exchanging FlowStasRequest/FlowStasReply messages with devices in the network. Per-flow statistics are regularly requested according to a predefined polling interval. Doing that ensures reliability and timely monitoring information that is relevant to management and configuration of an SDN-based network. However, the accuracy of flow monitoring depends on the frequency of requests from the controller and also may impact on the performance of an SDN controller. Since the more the requests, the more the tasks per second that the controller needs to process. Each SDN controller has a default interval to request the statistic of all flows from all devices. When applying this approach to a large scale network which has an enormous number of switches and its associated flows, it may lead to performance degradation issues at the collection time. In fact, the interval is possible to adjust to a larger number before running an instance of the controllers in order to overcome the performance issue. Nevertheless, the accuracy of monitoring might be reduced in case of decreasing query frequency. Therefore, a trade-off between the performance of controller and the accuracy of flow monitoring has to be taken into account. This chapter discusses the trade-off alongside the evaluation of two monitoring approaches applied in the ONOS controller.

The work in this chapter is taken from [6, 8]. After presenting relevant background and related works in Section 4.1, the evaluation of *Adaptive Flow Monitoring* approach in the ONOS controller to monitor flows in SDN networks is describes in Section 4.2. Then, Section 4.3 introduces another approach called *Selective Flow Monitoring* which is developed for more effective and more flexible flow monitoring. Finally, lessons learned from our studies are provided in Section 4.4.

4.1 Background and Related Work

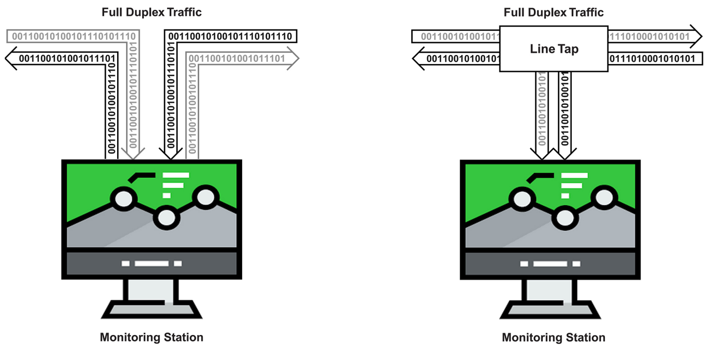
In the following, approaches regarding network monitoring are introduced in 4.1.1. Then, Section 4.1.2 outlines different ways to monitor SDN networks. After that, Section 4.1.3 highlights the related research done in the field of monitoring SDN networks with a focus on flow statistics.

4.1.1 Network Monitoring Approaches

In the past, computer network administrators might only monitor a few devices within a small scale of the network. The speed of connections was smaller than 100 Mbps. However, nowadays with the development of communication technology, internet infrastructure, and hardware configuration, the administrators have to deal with higher speed and large-scale networks. The links between devices support rates up to Gigabit per second and the number of devices reach hundreds to thousand. Network monitoring becomes more and more critical because of the need to maintain the stability of the system as well as quickly detect and resolve problems. Whereby, sophisticated monitoring tools and methods are researched and applied in two main directions: active or passive monitoring as shown in Figure 4.2.

Active Network Monitoring. This technique is also called "intrusive" monitoring and operates by inserting packets into an existing network [71] as illustrated in Figure 4.2a. The probe traffic is actively sent and tracked over the network, then it is analyzed and processed to have relevant metrics. An exemplary active monitoring tool is the *ping tool*, which measures loss of packets and round trip time between two points in the network based on ICMP Echo Request/Reply messages. *Iperf* [1] is another common example of an active monitoring solution, that allows evaluating TCP/UDP bandwidth performance. On the one hand, such methods allow flexible control of monitoring packets such

[1] `https://iperf.fr/`



(a) Active Monitoring. *(b) Passive Monitoring.*

Figure 4.2: Network Monitoring Approaches. [73]

as the type of packet or the injection time. On the other hand, sending extra packets might impact the performance of a small and low-speed network.

Passive Network Monitoring. The passive network monitoring approach or "non-intrusive" [72] is implemented by using devices to capturing traffic passing through them. There are several implementations of this approach. First, it can be done in specialized network sniffer hardware, for example, the Net Optics wiretap device[2]. The packets cross the device and are mirrored to a monitoring station as displayed in Figure 4.2b. Here, they are captured and analyzed to obtain a sophisticated investigation, e.g., network topology, applications generating the packets, or used protocols, etc. The second implementation is a on-device software, which has the capability to capture packets, and runs as applications such as *Wireshark* [3] or *tcpdump* [4]. Finally, it can be a function assembled into network devices like switches, routers or hosts. Examples of such built-in tech-

[2]http://www.ixiacom.com/products/ixia-gig-zero-delay-tap/
[3]https://www.wireshark.org/
[4]https://www.tcpdump.org/

niques contain Simple Network Monitoring Protocol (SNMP), Remote Monitoring (RMON), and Netflow [74] capable devices. Performing this approach does not interfere with the traffic on the network, however, it might require an additional hardware besides a monitoring station. In addition, since it might gather details of the packets, privacy and security issues also need to be taken into account.

Table 4.1: Comparison between active and passive monitoring [75]

Active Network Monitoring	Passive Network Monitoring
Relies on injecting test packets into network environment	Does not inject any artificial traffic into existing network
Monitor test traffic and bases results on real-time data	Monitor historic traffic and bases results on long-term data
Measures traffic with dedicated monitoring software only	Requires specialized device to measure traffic
Increase load on networking hardware	Adds very little overhead to networking hardware
Generates data on particular aspects of network performance	Provides a holistic overview of network performance
Collects smaller amounts of data specific to the problem at hand	Collects larger volumes of data that can be mined for a wide variety of information
Measures traffic both inside and outside network environment	Limited to measuring traffic on network devices is connected to
Works as a "controlled experiment", making it ideal for measuring a specific metric	Works like an "observational study", making it useful for analysis of large data volume

Hybrid Network Monitoring. The two methods above have various differences which are described in Table 4.1. Each of them has their own pros and cons. *Passive monitoring* might be a good choice over *active monitoring* regarding the overload of traffic on the network. Nevertheless, post-processing the captured packet can take a large time. Hence, the combination of two methods is utilized to leverage the best aspects of both approaches. For example, the schedule of passive monitoring can be conducted after running the active measurements to get a specific condition (bandwidth, delay) [76]. Zangrilli et. al introduce Watching Resources from the Edge of the Network (WREN) in [77] in order to combine active and passive monitoring techniques. Depending on the utilization of the network, the equivalent method is used. For example, passive monitoring is used for measuring the load of the network; once the traffic is low, WREN actively introduces packets to estimate other parameters.

The approaches, as mentioned earlier, are widely used in both legacy and SDN networks. For example, while in Chapter 3, Section 3.2.1, *passive monitoring* is applied to calculate flow installation times, in this chapter, the monitoring approach, which is specially used for SDN network, is introduced. Similar to the active method, an SDN controller sends packets to the network of OpenFlow switches. However, instead of tracking those packets, the controller waits for the respond of those packets in flow statistics reply messages. Those messages contain statistic information of flows in forwarding devices. Details of this approach is outlined in the next section.

4.1.2 Monitoring an SDN Network

As discussed in Section 3.1.1, SDN networks use the OpenFlow protocol to exchange information between an SDN controller and data plane devices. Implementing such protocol allows the controller to retrieve the number of packets which are processed in the switches. The key element for monitoring traffic in the data plane is using flow statistic request and reply messages, which is described in detail as follows.

OpenFlow Counter. It is one of the main components of an OpenFlow table entry as displayed in Figure 4.3. While the work in the last chapter is performed with OpenFlow version 1.0, this chapter focuses on version 1.3.0 to have an up-to-date research. With version 1.3.0 more components in the flow entry are added in comparison with the 1.0 version such as *Cookies* or *Timeouts*. They are identifiers specified by the SDN controller and durations that inform when the entry is deleted, respectively.

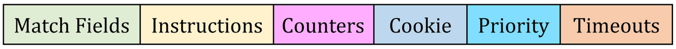
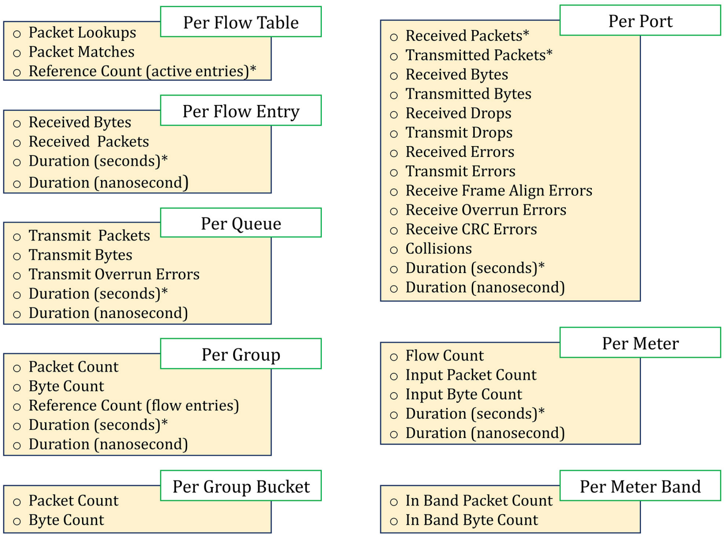


Figure 4.3: OpenFlow v1.3.0 Flow Entry [78].


Per Flow Table

- Packet Lookups
- Packet Matches
- Reference Count (active entries)*

Per Flow Entry

- Received Bytes
- Received Packets
- Duration (seconds)*
- Duration (nanosecond)

Per Queue

- Transmit Packets
- Transmit Bytes
- Transmit Overrun Errors
- Duration (seconds)*
- Duration (nanosecond)

Per Group

- Packet Count
- Byte Count
- Reference Count (flow entries)
- Duration (seconds)*
- Duration (nanosecond)

Per Group Bucket

- Packet Count
- Byte Count

Per Port

- Received Packets*
- Transmitted Packets*
- Received Bytes
- Transmitted Bytes
- Received Drops
- Transmit Drops
- Received Errors
- Transmit Errors
- Receive Frame Align Errors
- Receive Overrun Errors
- Receive CRC Errors
- Collisions
- Duration (seconds)*
- Duration (nanosecond)

Per Meter

- Flow Count
- Input Packet Count
- Input Byte Count
- Duration (seconds)*
- Duration (nanosecond)

Per Meter Band

- In Band Packet Count
- In Band Byte Count

Figure 4.4: OpenFlow v1.3.0 Counters [78].

OpenFlow 1.3 supports IPv6 that enhances the scalability of the network and declares the expectation to apply SDN in Internet of Thing (IoT) environments. Moreover, not only the number of matching fields is increased, but also multiple flow tables are implemented which allows more flexible matching of packet headers. This leads to many parameters that the *Counter* provides since they are maintained for each flow entry, flow table, as well as each port, queue, group, meter, and meter band [78]. To have an overview of the details of those parameters, Figure 4.4 describes all the counters with the mark * for required counters.

Depending on vendors, an OpenFlow switch might support all the counters above. Otherwise, it can consist of some required counters, e.g., duration of a flow entry, received and transmitted packets per port, or the number of active entries in a table. This information is regularly queried from the controller to monitor the state of a network by using OpenFlow statistics messages.

OpenFlow Statistics Messages. Figure 4.5 illustrates a session of an exemplary communication between an SDN controller and a switch. At the

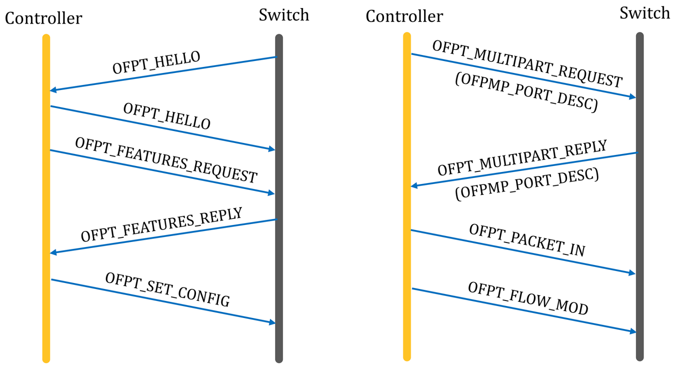


Figure 4.5: OpenFlow v1.3.0 Messages Exchange [79].

beginning, after a TCP connection is initiated from the switch, it sends an OFPT_HELLO message to the controller with the information of supported OpenFlow version. If no error occurs, the controller sends back another hello message to finish establishing a session. Then, an OFPT_FEATURE_REQUEST message is dispatched to ask the switch to provide its datapath ID as well as its capabilities. Next, the controller sends the OFPT_SET_CONFIG to set the fragmentation properties of later packets or the maximum bytes that data path should transmit to the controller. The OFPT_PACKET_IN and OFPT_FLOW_MOD use to inform and install a new flow, respectively.

The focuses in this section are OFPT_MULTIPART_REQUEST and OFPT_MULTIPART_REPLY which are used for requesting and providing the state of the data plane, respectively. The messages provide various statistics as mentioned in Table 4.4: flow, table, port, queue, meter, etc. Figure 4.6 shows the detailed structure of OFPT_MULTIPART_REQUEST message. The field *Type* in the header of this packet defined which concrete parameter is queried. For example, a value of "0x0001" corresponding to an individual flow statistics, "0x0004" describes a query port statistics, or "0x0005" represents a request for a queue.

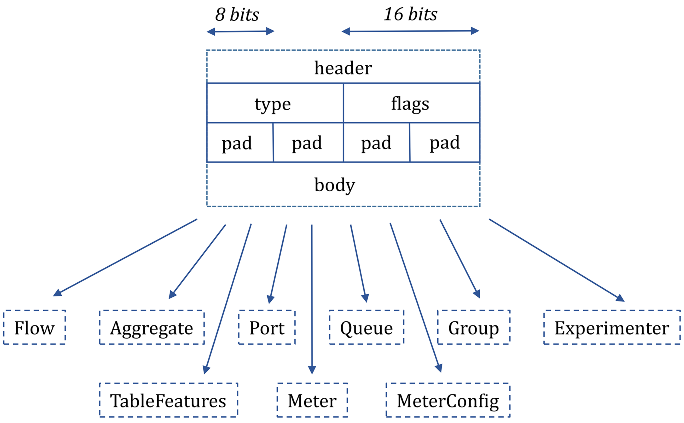


Figure 4.6: OpenFlow v1.3.0 Multipart Request [79].

OpenFlow version 1.3 aggregates information regarding these statistics into *multipart messages*, which combines many types of counters, in order to reduce the amount of packets that is exchanged during query duration. The switch response the corresponding message by an OFPT_MULTIPART_REPLY which contain all information that the controller wants to know. The request can be sent directly from an application running at the controller or via RESTful API with the PUT command.

4.1.3 Related Work

This section outlines several studies related to monitoring an SDN network. NetFlow [80] is a well-known tool used to collect network traffic information and also supports flow-level monitoring. With the purpose to reuse existing NetFlow analyzing application in OpenFlow networks, some proposals discuss a combination of OpenFlow and NetFlow in the SDN environment [81, 82]. On the one hand, OF2NF [81] introduces an application built for the Ryu controller, which allows gathering flow statistics using OpenFlow protocol capabilities and exporting this information from the controller over NetFlow protocol to a collector. On the other hand, the authors in [82] proposed two sampling methods implemented within the OpenDaylight controller towards a NetFlow implementation for SDN networks. While the first method based on the sampling rate defined by the number of bits checked for source and destination IP suffixes, the other one uses a hash function on fields of a packet header to decide the polling interval. SBAR [83] exports flow-level monitoring reports to an external analytics tool such as the way NetFlow operates. Besides, it classifies traffic by application protocols in particular for web and encrypted traffic to identify the source application (e.g., Netflix, Facebook) of the traffic.

PayLess [84] introduces a network monitoring framework for SDN-based networks. A network application can create a *MonitoringRequest* object in JSON format. Then, the object is registered and through RESTful API Payless interacts with the Floodlight controller to collect the data based on this request, e.g., throughput and packet-drop for a particular user. In an effort to balance the accuracy of statistics and the overhead at the controller and switches, the authors propose an approach that dynamically adapts the frequency of the flow statistics collection for all flows. A scheduler is built and adjusts the monitoring frequency according to network load. Also, monitoring duration to each flow are added, hence, flows that have significant link utilization are assigned high polling frequencies. OpenNetMon [85] implements a POX Controller module that monitors not only the throughput but also per-flow QoS metrics like path delays on the network and application layer. Based on these measurements, a fine-grained Traffic Engineering can be performed. Furthermore, a timer is used to increase the polling interval for new flows and flows with high fluctuations w.r.t. their statistics, while stable flows are queried less frequently. The study conducted in [86] features a monitoring scheme that minimizes the communication overhead by aggregating the request and reply messages. This approach is based on the assumption that querying only a small number of switches is sufficient to obtain statistics of the majority of flows in the network. The polling switches selection is modeled as a weighted set cover when considering the amount of exchanged statistics data as communication cost. The results show that FlowCover can improve the utilization of communication cost by up to 50%. However, those works have not measured the resource consumption at the controller. Instead, our work provides insights into the trade-offs regarding this aspect of adaptive and selective flow monitoring by analyzing the CPU utilization at the ONOS controller.

SDN-Mon [87] focuses on another aspect when proposing a framework with its components reside in both controller and switches. Monitoring modules are installed in controller-side and switch-side. Those modules communicate with each other by implementing their messages based on the OpenFlow protocol. Therefore, the monitoring function is separated from the general OpenFlow operation. It facilitates the controller to support more flexible monitoring. However, this framework requires the capability to install an external module in switches to exchange SDN-Mon messages with an application at the controller. That might require the switches are open-source software switches like Open vSwitch [5] or Lagopus [6], but not proprietary closed-source hardware.

On the direction of developing an independent monitoring application running on top of an SDN controller, MonSamp is presented in [88] including monitoring and flow sampling. The Northbound API is used for communication between the application and the controller. A similar approach is SOFTmon [89], the application is programmed in Java and has a GUI that displays graphs for metrics as flow count per switch, received/transmitted bytes per port, or the number of packets/bytes per flow.

FlowSence [90] relies on OpenFlow *PacketIn* and *FlowRemoved* messages to estimate network utilization on each link. Since *FlowRemoved* contains the duration of an entry and the amount of traffic matched, this can be used to infer the utilization contributed by the flow. The benefit of this mechanism is that it takes the state of traffic in the network rather than query it. Nevertheless, according to the authors, there are two limitations of this method, which the authors also mentioned. First, the estimated utilization might be computed only at discrete points in time with checkpoints determined by arrivals of FlowRemoved. Second, if a flow lasts for a long time, it delays the computation of the network utilization.

The work in [91] presents OpenTM that is capable of choosing switches based on the routing information to periodically poll flow statistics. For example, NOX controller running OpenTM can query the statistics of switches along a flow path. The query is based on different strategies: following a distribution of random variables like non-uniform distribution that queries the switch closer to the destination host, or specifing a switch based on its load to request the statistics. ProgME [92] collects flow information based on the proposed *flowset*, an arbitrary set of flows that depends on the requirements of an application or a particular traffic condition.

[5] `https://www.openvswitch.org`
[6] `http://www.lagopus.org/`

4.2 Performance of Adaptive Flow Monitoring in the ONOS Controller

The ONOS controller inquiries the statistics of all flows in the network with a default setting frequency is 5 seconds. However, this method faces to the fact that the controller might overload due to an enormous number of requests. In [93] a flow monitoring approach is introduced with the goal to minimize the overhead of an SDN controller when polling OpenFlow statistics. This section aims to evaluate the approach in terms of resource usage at the controller when applying *Adaptive Flow Monitoring* (AFM). First, general information of this mechanism is described in Section 4.2.1. Next, Section 4.2.2 highlights detailed evaluation results.

4.2.1 Adaptive Flow Monitoring Algorithm

As shown in Figure 4.4, every flow entry in an OpenFlow table contains the duration which is calculated from the time it is inserted until the time this entry is requested. It is called "Flow Lifetime" in this work. The flows are classified in 3 groups of FlowLiveType based on their lifetime, e.g., LONG flows are the flows last more than 15 seconds at the query point, this value in case of MID flows is 10 seconds. Figure 4.7 visualizes the detail of this classification.

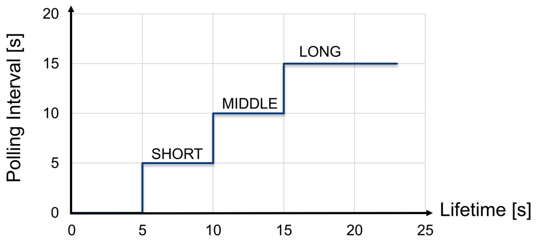


Figure 4.7: Classification of FlowLiveType based on their lifetime and their corresponding polling intervals.

The AFM algorithm relies on this lifetime to decide appropriate polling intervals for particular flows. Figure 4.8 illustrates the algorithm of this approach.

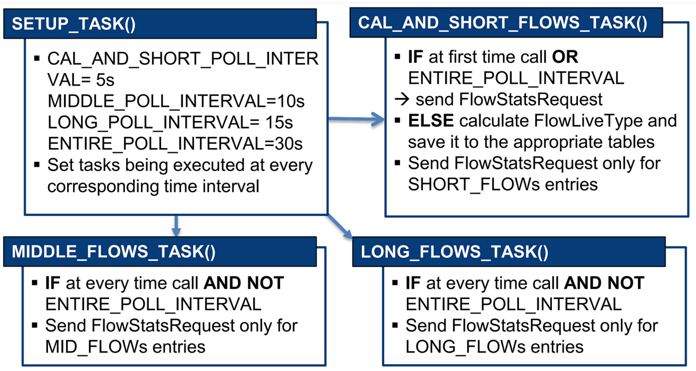

SETUP_TASK()
▪ CAL_AND_SHORT_POLL_INTERVAL= 5s
MIDDLE_POLL_INTERVAL=10s
LONG_POLL_INTERVAL= 15s
ENTIRE_POLL_INTERVAL=30s
▪ Set tasks being executed at every corresponding time interval
CAL_AND_SHORT_FLOWS_TASK()
▪ IF at first time call OR ENTIRE_POLL_INTERVAL
→ send FlowStatsRequest
▪ ELSE calculate FlowLiveType and save it to the appropriate tables
▪ Send FlowStatsRequest only for SHORT_FLOWs entries
MIDDLE_FLOWS_TASK()
▪ IF at every time call AND NOT ENTIRE_POLL_INTERVAL
▪ Send FlowStatsRequest only for MID_FLOWs entries
LONG_FLOWS_TASK()
▪ IF at every time call AND NOT ENTIRE_POLL_INTERVAL
▪ Send FlowStatsRequest only for LONG_FLOWs entries

Figure 4.8: Adaptive Flow Monitoring Algorithm.

Flows in a group follow corresponding task which is applied to this group. For example, the LONG_FLOWS_TASK() is executed for the flows in the LONG group only. Meaning, the controller generates FlowStasRequest messages for *individual flows* in this group every 15 seconds. Flow groups are updated every 5 seconds by implementing CAL_AND_SHORT_FLOWS_TASK(). This task calculates and updates lifetimes of current flows as well as re-arranges the flows in each group. An important notice is that the FlowStasRequest message in each task is assigned for a particular flow. Hence, the number of FlowStasRequest messages is exactly the number of flows in the network. This is different from *Standard Flow Monitoring* (STD) where the controller request the statistic of all flows by one-single-FlowStasRequest. However, STD supports a fixed polling interval, if this interval so small, more accuracy can be obtained, but the utilization of the controller also increases. AFM prevents high query frequency by using these task above as well as after an ENTIRE_POLL_INTERVAL (twice a LONG polling interval), an exact FlowStasRequest like in case of STD is performed.

4.2.2 Evaluation Results

Experimental Setup. The Adaptive Flow Monitoring is evaluated in a testbed as depicted in Figure 4.9. A computer [7] runs ONOS Hummingbird (version 1.7.0) as an SDN controller. Mininet version 2.2.1 is installed in a virtual machine [8] in an OpenStack Cloud [9] to simulate a network. In the first scenario, a tree topology consisting of 63 OpenFlow switches and 64 hosts is created. Once the topology is completely loaded, the *pingall* command in the Mininet environment is executed and the controller installs around 37,000 in the switches. Doing this ensures connections between hosts in the network. Furthermore, another configuration at the controller, which prevents these flows from being removed during the time of the experiment, is set before running a test. In this case there is no traffic on the data plane. As second scenario, a ring topology was chosen with 4 switches,

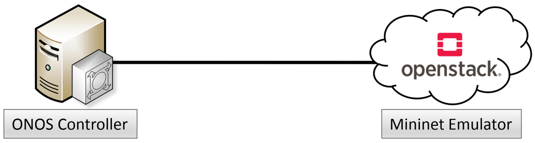


Figure 4.9: Experimental Setup.

each switch connected to 4 hosts. Data plane traffic is replayed from a PCAP file [10] which is recorded from the real traffic of a New Zealand ISP. The file is modified to get suitable IP addresses in the simulated network. Pidstat [11] is used to monitor the ONOS process, and details regarding the average memory usage and CPU consumption are exported regularly during its runtime. Every experiment is repeated 10 times to get 95% confidence intervals for both cases, Standard Flow Monitoring (STD) and Adaptive Flow Monitoring (AFM).

[7]Intel(R) Core(TM) i7-3770 CPU @3.40GHz / 12GB RAM / Ubuntu 14.04 LTS
[8]Intel(R) Core(TM) 2 Duo/ 8GB RAM / Ubuntu 14.04 LTS
[9]https://www.openstack.org
[10]`https://wand.net.nz/wits/ispdsl/2/`
[11]`http://sebastien.godard.pagesperso-orange.fr/man_pidstat.html`

CPU Utilization in a Single Run. Figure 4.10 shows the CPU consumption when AFM is enabled and disabled in the first scenario mentioned above. While the runtime of an experiment in seconds is displayed on the x-axis, the y-axis denotes CPU utilization. Different colors correspond to the two approaches.

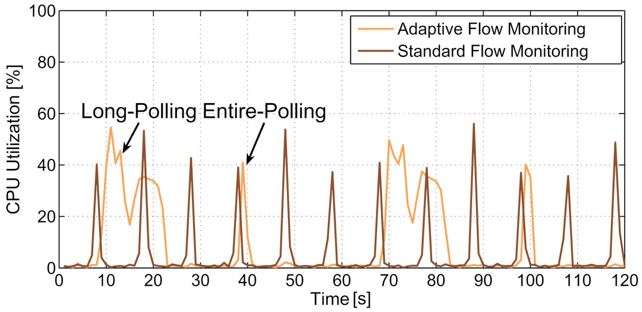


Figure 4.10: CPU Utilization in a single experiment.

In this case, the polling intervals of AFM are 10s, 20s, 30s for SHORT, MIDDLE, LONG flows, respectively. The ENTIRE_POLL_INTERVAL is set to 60s. Meanwhile, STD requests statistics of all the flows every 10 seconds, which is illustrated by separated peaks 10 seconds apart. Almost values are around 40%, and the highest is 58%. The duration of the peaks is nearly 2 seconds. As a result, the time that the controller takes to finish a round of requesting and processing requests is quite fast. In contrast to a short processing time when STD is enabled, a delay more than 10 seconds is needed in the context of AFM. The query frequency is less than STD, however, it takes a longer time to finish a poll. An explanation is that the number of FlowStasRequest messages sent by AFM is much larger than STD. While STD generates one messages per switch to query all flows in the switch, AFM needs to send each flow a request. Consequently the number of request in case of STD equals the number of switches, it is 64. This value when AFM is enabled in around 37,000 messages. In addition, this represents the scenario of AFM when every flow is of type LONG. Hence, only

Long-Polling is carried out. The *Entire-Polling* every 30 seconds after the *Long-Polling* is exactly the polling in case of STD, which is shown by similar shapes of two lines around 40th second and 100th second.

Comparison of CPU Utilization. In the last evaluation, the controller does not remove any flows during the experiment and there is no data plane traffic. Besides, the AFM is considered in the case that all flows are LONG flows. In order to evaluate AFM in a more realistic scenario, a captured file is used to generate traffic where the lifetime of flows varies randomly. This file is replayed by using Tcpreplay [12] from a host. That means that the number of flows is changed from time to time and the controller removes the flows that do not contain matched packets. The average CPU utilization in this scenario is provided in Figure 4.11 with different polling intervals, which are on the x-axis. The ∞ symbol implies a large value that is longer than runtime. In this case, there is no FlowStasRequest message requested and it is labeled as "Without Monitoring". Three color bars represent different scenarios, and the whiskers show 95% confidence intervals.

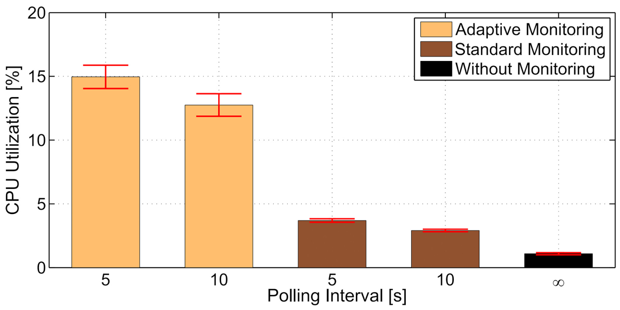


Figure 4.11: Comparison between approaches in term of CPU Utilization.

As the black bar indicates, the CPU load is around 1% without any polling

[12] `https://tcpreplay.appneta.com/`

activity, since the controller still communicates with the OpenFlow switches and handles other messages such as LLDP, GetConfig, or FeatureReq messages. Another observation is that the smaller the polling interval is, the higher the CPU usage becomes. In both cases of STD and AFM, 10s polling results in less CPU load than 5s. However, with the same approach, different values of polling intervals do not have a significant impact on CPU load as shown as narrow gaps between corresponding bars. In addition, the performance of AFM does not outperform the standard. Its results for polling intervals of 5s and 10s are 13% and 15%, respectively. These values are more than three times as high in comparison with STD. The reason is due to large number of FlowStasReq messages during AFM polling intervals. Unlike this behavior, STD generates only 4 messages (corresponding to 4 switches) to query all flows.

Moreover, when generating *FlowStasReq*, AFM requires exact matching fields to distinguish a flow, hence, the complexity of this message is significantly higher than the one generated by STD (only a wildcard for retrieving all flows in a switch).

The results illustrate the drawback of Adaptive Flow Monitoring with respect to CPU consumption. Based on the idea of classifying flows into different groups and querying each group with a particular polling interval, we deployed another monitoring approach, called *Selective Flow Monitoring* (SFM). When applying AFM, all flows are requested to send their information to the controller. On the contrary, SFM method only inquires the flow, which is specifically inquired by the controller, reply the statistics message. Therefore, SFM reduces CPU usage compared with STD. The details of the SFM mechanism and evaluation are discussed in the next section.

4.3 Selective Flow Monitoring

In the following, after presenting a testbed featuring the Open Network Operating System (ONOS) controller, the SFM mechanism is introduced. Then, a comparison between the performance of the SFM mechanism and the default monitoring scheme is discussed. In this context, the CPU utilization of the controller is used as performance indicator. After identifying relevant influence factors like the number of flows and switches in the network, we investigate the viability of the approaches in different scenarios. Finally, we provide guidelines regarding their choice.

4.3.1 Measurement Setup

To investigate the performance of the SFM approach, a testbed as shown in Figure 4.12 is set up. The Ibis release[13] of the ONOS controller is installed on a PC[14] which uses the Ubuntu 16.04 operating system. Another PC with the same specification runs Mininet[15] and is used to create a network of OpenFlow switches.

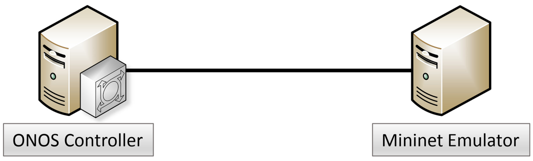


Figure 4.12: Logical Testbed Setup.

At the beginning of the measurement, after the controller is started, a predefined number of flows is installed in the switches via the REST API by using cURL[16]. The term *flow* represents a *flow entry* in an OpenFlow Table of the switch. Each flow has a timeout of 5,000 seconds to ensure that it lasts the whole 5 minutes of the experiment. Depending on the scenario, up to 408,000 flows are installed. Since this work focuses on the performance of the controller when dealing with control plane information, there is no data plane traffic on the network. Each run is repeated 5 times in order to obtain 95% confidence intervals.

[13] `http://onosproject.org/`, version 1.8.0
[14] Intel(R) Core(TM) i7-4770 CPU @3.40GHz / 16GB RAM
[15] `http://mininet.org/`, version 2.2.1
[16] `https://curl.haxx.se/`

Pidstat[17] is used to monitor the average CPU consumption of the ONOS controller as in the last Section.

In the first scenario, one OpenVswitch[18] is created in Mininet, and the flows in the network are partitioned into three groups, i.e., all flows in a group have the same destination IP address (IP_DST) but different source IP addresses. All IP addresses are declared with network mask 255.255.225.255, which makes all flows unique and avoids the aggregation of rules in the flow table of the OpenFlow switch. Once SFM is enabled, the flows are queried for their statistics according to three polling intervals: SHORT (5s), MIDDLE (10s), and LONG (15s). At each polling time, the controller asks the switch for information of the flows that have the same combination of output port and destination IP, i.e., (OUTPUT; IP_DST). The switch replies to this *StatsRequest* by sending *StatsReply* messages only for the flows that match this condition and aggregates them in a *MultipartReplyMessage*. When using the standard method, the ONOS controller inquires statistics of all flows every 5 seconds and does not require any match condition. The portions of the number of flows from each group are varied in order to investigate the impact of these factors on the controller's CPU utilization.

In the second set of experiments, we evaluate the performance impact of scenarios where more than one statistics message is required to fetch information about each class of flows. In the following, we refer to this number of messages as the number of sub-groups since each class of flows is partitioned based on a set of flow rules. In the i-th group, the tuple for matching its statistics becomes (OUTPUT; IP_DST$_i$). Additionally, testbeds with multiple switches based on a tree topology are considered.

4.3.2 Selective Flow Monitoring Mechanism

In this section, we conduct equations calculating the number of messages that needs to process by the ONOS controller in both Standard Flow Monitor-

[17]`http://sebastien.godard.pagesperso-orange.fr/man_pidstat.html`
[18]`http://openvswitch.org/`

ing (STD) and Selective Flow Monitoring (SFM) methods. Besides, the operation of SFM as well as the differences between these approaches are described.

Standard Flow Monitoring in the ONOS Controller. As mentioned above, the ONOS controller regularly sends *FlowStatsRequest* messages every 5 seconds to get the information of all flows in the network. After receiving a *FlowStatsReply* message, the controller processes it and gets the details of all flows.

The number of messages which ONOS needs to handle at every polling point in case of Standard Flow Monitoring (STD) is denoted as N_p^{STD} and can be calculated as follows:

$$N_p^{STD} = N_{(sent)}^{STD} + N_{(recd)}^{STD}.$$

In this equation, $N_{(sent)}^{STD}$ and $N_{(recd)}^{STD}$ represent the amount of messages that are generated and sent as well as received and processed by the controller, respectively. The network consists of N_{SW} switches and the total number of flows is N_F. In case of STD, the controller only generates one single $FlowStatsRequest$ message corresponding to a switch. Thus, the equation above becomes:

$$N_p^{STD} = N_{SW} + N_F.$$

Considering an experiment duration of T_{exp} with regular polling interval t_{poll}, the total amount of messages:

$$N^{STD} = \frac{T_{exp}}{t_{poll}} \cdot N_p^{STD} = \frac{T_{exp}}{t_{poll}} \cdot (N_{SW} + N_F). \tag{4.1}$$

This equation illustrates that the smaller the polling interval, the higher the number of messages. Hence, increasing polling interval results in a decrease of CPU consumption since fewer messages need to handle. However, the information of some flows that have lifetimes less than the interval might be missed. To overcome this issue, SFM approach is introduced and described as follows.

Selective Flow Monitoring Approach. The SFM approach allows administrators to increase the monitoring accuracy for particular flows that are of special interest, e.g., those with strict application-specific QoS demands or SLAs. Furthermore, SFM introduces several configurable types of polling intervals, which can be set by the operator actively, depending on how often the flows are to be monitored.

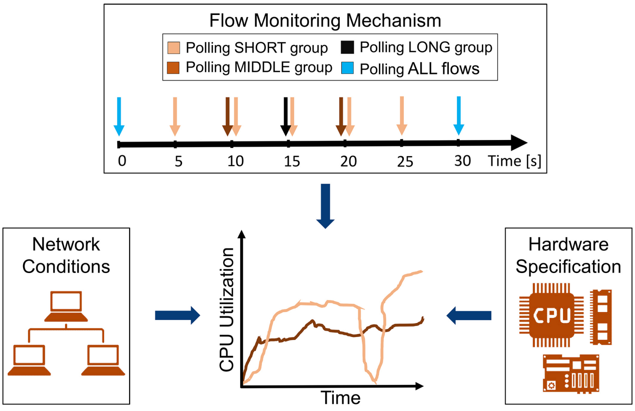


Figure 4.13: Influence factors on CPU utilization of the SDN controller when monitoring flow statistics.

Figure 4.13 illustrates several components that affect the CPU utilization of an SDN controller which are investigated in this work. First, the hardware configuration of the controller such as its RAM capacity and the particular CPU model affects message processing time as well as the amount of flow rules that can be managed. Second, network conditions like the number of connected switches or the composition of traffic flows dictate the amount and the rate of messages that are exchanged between the switches and the controller. Finally, the used flow monitoring mechanism, i.e., SFM or STD, has an effect on the message rate, too.

Figure 4.13 also gives an example of how SFM works. The controller classifies flows into three different groups based on a specified condition and each group's statistics are queried with the respective polling interval. At the beginning, all flows are polled. After that, every 5 seconds the controller asks the state of the flows that are marked as SHORT which is indicated by the light brown arrows. Then, every 10 seconds, the dark brown arrow shows that flows in the MIDDLE group are queried, and every 15 seconds, *FlowStatReply* messages of flows in the LONG group are sent to the controller. Finally, an ENTIRE polling happens each 30 seconds to ensure that all flows are updated. It is worth noting that when an ENTIRE polling is performed, no other polling takes place. In contrast to this mechanism, the standard approach in the ONOS controller frequently inquires information of all flows every 5 seconds. The differences between SFM and STD are the variation of the number of polling intervals as well as the classification of flows into groups. Hence, only the flows of a particular group are queried rather than all flows.

Assume that the amounts of flows that are requested with the short, middle, and long polling intervals are N_s, N_m, and N_l, respectively. Hence, in Equation 4.1, the total number of flows corresponds to $N_F = N_s + N_m + N_l$. When SFM is enabled, four types of requests can happen when polling:

a) Only SHORT flows are queried:

$$N_1^{SFM} = N_{1(sent)}^{SFM} + N_{1(recd)}^{SFM} = N_{SW} \cdot n_s^g + N_s.$$

b) Flows that are either SHORT or MIDDLE are queried:

$$N_2^{SFM} = N_{SW} \cdot (n_s^g + n_m^g) + (N_s + N_m).$$

c) Flows that are either SHORT or LONG are queried:

$$N_3^{SFM} = N_{SW} \cdot (n_s^g + n_l^g) + (N_s + N_l).$$

d) All flows are queried (ENTIRE-polling):

$$N_4^{SFM} = N_{SW} + N_F.$$

In this context, n_*^g denotes the number of sub-groups within each group and N_* refers to the total number of flows within each group.

During an experiment with duration T_{exp}, t_i is the polling interval that corresponds to short, middle, long, and entire interval. Hence, the total number of messages during an experiment can be calculated as follows:

$$N^{SFM} = T_{exp} \cdot \sum_{i=1}^{4} \frac{1}{t_i} \cdot N_i^{SFM}. \tag{4.2}$$

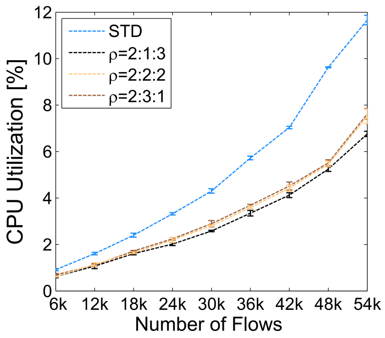
Suppose that the polling intervals for the MIDDLE and LONG groups, t_m and t_l, are multiples of the polling interval for the SHORT group, t_s, i.e., $t_m = \alpha \cdot t_s$, $t_l = \beta \cdot t_s$. In an analogous fashion, the numbers of flows in each group are $N_m = m \cdot N_s$ and $N_l = n \cdot N_s$. Hence, from Eq. 4.2, the number of messages in case of SFM, N^{SFM}, is calculated as follows:

$$\begin{aligned} N^{SFM} = \frac{T_{exp}}{t_s} \cdot \Bigg[N_{SW} \cdot \Big[2n_s^g + \frac{2(n_s^g + n_m^g)}{\alpha} + \frac{1 + 2 * (n_s^g + n_l^g)}{\beta} \Big] \\ + \Big(2N_{SW} + \frac{2 + 2m}{\alpha} + \frac{3 + m + 3n}{\beta} \Big) \cdot N_s \Bigg]. \end{aligned} \tag{4.3}$$

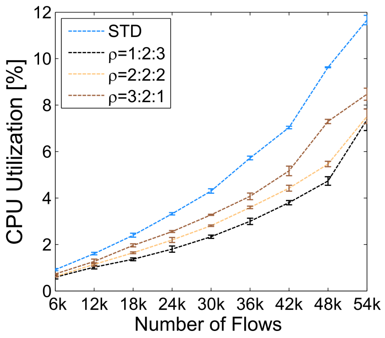
Equations 4.1 and 4.3 present several influence factors which impact the number of flow statistics messages, such as the number of sub-groups (n_s^g, n_m^g, n_l^g), the relative size of each group ($\rho = N_s : N_m : N_l$), the ratios between the corresponding polling intervals ($\gamma = t_s : t_m : t_l$), as well as the number of switches in the network N_{SW}. Details regarding the investigation of those parameters are presented in Section 4.3.3.

4.3.3 Performance Evaluation of SFM

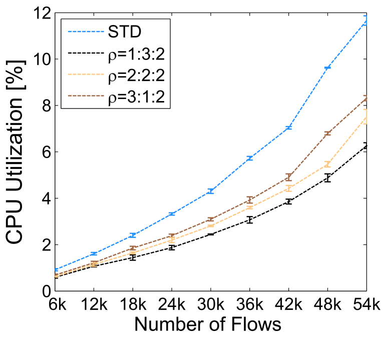
This section provides a comparison between the SFM and STD approaches in terms of the controller's CPU utilization alongside the impact of the size ratios of flow groups. Then, the impact of changing the number of sub-groups and the number of switches in the network is presented.



(a) Fraction of SHORT flows is constant.



(b) Fraction of MIDDLE flows is constant.



(c) Fraction of LONG flows is constant.

Figure 4.14: Influence of relative group size on CPU utilization.

CPU Utilization in Case of a Single Switch. Figure 4.14 displays the impact of different portions of the number of flows in each group. For a given total number of flows in the network on the x-axis, the y-axis shows the corresponding mean CPU usage during the controller's run time. In this experiment, the maximum memory that is assigned to the JVM equals 512 MB which is the default setting of the ONOS controller. As a result, there is a limitation of 54,000 flows that the controller can handle without throwing an "overhead limit exceed" exception. The whiskers show 95% confidence intervals which are obtained after 5 experiment repetitions. The blue curve represents the results in case of Standard Flow Monitoring (STD), while the other colors indicate the measurement data when enabling SFM with different ratios of each flow type. The three subfigures highlight the CPU utilization that results from setting the ratio of one flow class to a constant while varying the ratios of the two other classes.

For all scenarios, with the same number of flows, STD consumes more CPU resources than SFM. The gap between the blue curve and the others becomes wider when increasing the number of flows and achieves a maximum value of nearly 4%. In case of SFM, the ratio of the SHORT group has the largest impact on the controller's CPU usage, which is displayed in Figure 4.14b and Figure 4.14c. When keeping the portions of MIDDLE and LONG groups constant, the highest CPU utilization is reached if the SHORT group has the highest ratio (brown curves). A reasonable explanation is that the flows in the SHORT group are queried most frequently among all groups. Therefore, a higher number of flows in this group results in more messages and tasks per second that the CPU needs to carry out and leads to a high CPU load. However, in this worst case of SFM, the resource consumption at the controller is still less than in the case of STD.

The portion of SHORT flows is defined as a main influence factor on the CPU usage of the controller according to the above discussion. In order to perform a deeper investigation, measurements with different numbers of sub-groups in the SHORT group are carried out and the results are presented in Figure 4.15. There are six dashed lines in different shades of brown that correspond to different numbers of sub-groups within the SHORT group. These numbers directly affect the number of destination IP addresses in the flow table, and range from one to 50 groups. The number of flows in each group (SHORT, MIDDLE, LONG) is unchanged and follows the ratio $\rho = 1:2:3$. For example, at the first point on the x-axis, the total number of flows equals 6,000, of which 3,000 flows are classified as LONG, 2,000 flows as MIDDLE, and 1,000 flows as SHORT. In these 1,000 flows, the number of destination IP addresses changes from 1 to 50, which makes the controller generate different numbers of *FlowStatsRequest* messages every 5 seconds, coressponding to the number of IP addresses (also known as the number of sub-groups). However, when the controller request the statistic of all those SHORT flows, the quantity of *FlowStatsReply* messages is constant and equals 1,000 - the number of SHORT flows that are installed.

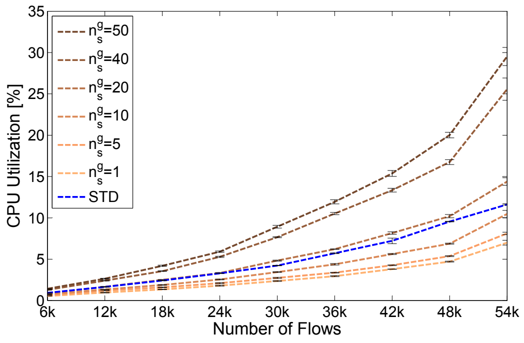


Figure 4.15: Impact of the number of sub-groups in the SHORT group.

The blue line represents the data that is obtained when STD is enabled. A trade-off between using the two mechanisms is illustrated. When the number of sub-groups is smaller than 20, SFM shows a better result, i.e., less CPU usage. At 20 sub-groups, there is no significant difference between STD and SFM unless 54,000 flows are installed, as illustrated by overlapping confidence intervals between those two cases. However, if more sub-groups exist, more resources are required to adapt to the large number of messages that arrive at the controller. This observation implies that in some cases, SFM results in a higher CPU usage than the standard method. However, due to its ability to query the information of particular flows, SFM provides significant resource savings when the number of sub-groups is low.

CPU Utilization in Case of Multiple Switches. Equation 4.3 shows that the number of switches in the network N_{SW} is also a factor that impacts CPU utilization. In order to highlight this effect, Figure 4.16 displays cumulative distributions of the controller's CPU usage in both cases, SFM and STD with different values of N_{SW}.

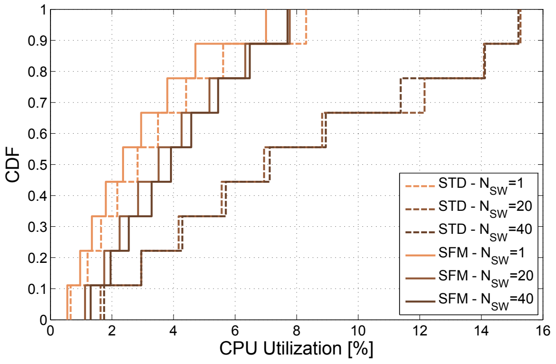


Figure 4.16: Distribution of the CPU utilization for different values of N_{SW} when using STD and SFM.

In this context, the total number of switches varies between 1, 20, and 40 switches, represented by the line colors, respectively. The resulting distributions in case of SFM are shown as solid lines, and the dashed lines present measured data when using STD. For this evaluation, we aggregate measurements from scenarios that feature between 6,000 and 54,000 flows in increments of

6,000 flows. While the CPU load is displayed on the x-axis, the y-axis indicates the fraction of measurements in which the CPU utilization is below the corresponding threshold. Again, the memory limit for the JVM equals 512 MB, and the value $\rho = 1:2:3$ is considered for the portions of groups. Since the number of *FlowStatsRequest* messages is equal to the amount of OpenFlow devices in the network, more switches in the network lead to more communication from the controller at each polling event. Consequently, an extra amount of work needs to be handled by the CPU and its load rises substantially in comparison to the single switch case. Especially in case of STD with a single switch, the CPU load never exceeds 9%. In contrast, when using multiple switches, the CPU utilization exceeds 10% in more than 30% of cases.

Additionally, the maximum observed CPU utilization when using SFM is below 8%, which equals only half the value in the context of the standard method. This show that the SFM method outperform the standard with respect to controller's CPU consumption. In both cases, increasing N_{SW} leads to higher CPU utilization. However, the growth gradually converges when the network contains more than 20 switches, as exhibited by the overlapping lines that correspond to $N_{SW} = 20$ and $N_{SW} = 40$. The largest gap between those configurations and the one featuring a single switch is significantly smaller in the case of SFM than when using STD, with the largest difference being 2% compared to nearly 7%.

Impact of Java Virtual Machine Memory. Due to the fact that the ONOS controller runs as a Java program, it is necessary to take into account the memory space for the Java Virtual Machine (JVM) when investigating the performance of the controller. The amount of memory that is dedicated to the ONOS controller can be controlled by means of an environment variable[19].

Figure 4.17 displays the maximum number of flows that the controller can handle before throwing exceptions regarding memory issues. While the x-axis

[19]To allocate *minValue* of heap memory at the start and up to a maximum of *maxValue*, the following command can be used: `export JAVA_OPTS="-server -Xms[minValue] -Xmx[maxValue]"`

denotes the number of flows which ranges between 24,000 and 408,000 in steps of 24,000, the y-axis represents the mean CPU utilization. Differently colored bars correspond to two configurations regarding the maximum assigned memory for the JVM, i.e., 2 GB and 4 GB, respectively. The whiskers provide confidence intervals that are obtained from 5 runs. The same combination of the flow class ratio ρ=1:2:3 ($N_s : N_m : N_l$) and polling time ratio γ=1:2:3 ($t_s : t_m : t_l$) as in previous experiments is used for the SFM scenario.

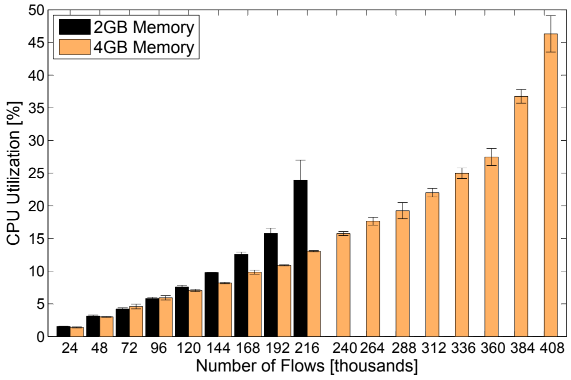


Figure 4.17: Influence of JVM Memory.

The first observation is that doubling the memory for the JVM does not mean that the controller is able to process two times more flows. In case of using 2 GB, the maximum number of flows in the network is 216,000. The value when increasing the memory limit to 4 GB equals 408,000 flows and is therefore 1.89 times higher. Additionally, when the controller has more capability for storing processed information, the CPU consumption decreases. While the reduction is not significant in the context of less than 120,000 flows, it is equal to up to 45% when 216,000 flows are involved. Since the controller reserves the set amount of memory regardless of actual usage and number of flows, it is important to determine a value that balances the trade-off between potentially unused memory resources and performance benefits. Otherwise, the memory might become a performance bottleneck of the controller. This is not only true for the ONOS controller but also for other Java-based SDN controllers like OpenDaylight, Floodlight, and Beacon. Given the network state in terms of the number of active flows, an administrator can derive viable memory limits.

4.4 Lessons Learned

In this chapter we focus on the control plane with applications that allow monitoring a network at flow-level. Several algorithms regarding flow monitoring in SDN networks are introduced. First, Standard Flow Monitoring (STD) operates based on typical OpenFlow functionality, i.e., information exchange schemes and counters for ports, tables, queues, and meters. STD regularly requests the information in pre-defined intervals. The accuracy of this method depends on the frequency of FlowStasRequest. More accurate monitoring can be obtained when reducing this interval, however, it might also lead to a high resource consumption. In an effort to balance these parameters, the second approach is studied - Adaptive Flow Monitoring (AFM). In this context, flows are classified according to their lifetime and the polling intervals are adapted accordingly, resulting in a high polling frequency for short flows and a low polling frequency for long flows. Finally, Selective Flow Monitoring (SFM) aims at reducing the overhead at an SDN controller. This method also allows to proactively and flexibly choose specific flows that network administrators desire to monitor, instead of querying all flows in the network.

Next, comparisons between three approaches in terms of the controller's CPU utilization and memory usages are implemented. On the one hand, the AFM mechanism can lead to an increase in control plane traffic since the controller queries flows from each category individually rather than requesting statistics of all flow entries in a switch as the STD does. The intensity of this effect is proportional to the number of flows in the network, which in turn is proportional to the number of FlowStatsReply messages that are processed by the controller. In addition, although the query frequency of AFM is less than STD, though, its duration for processing a polling point is longer than with STD. Both reasons result in a roughly 3 times higher CPU utilization when comparing AFM and STD. On the other hand, SFM performs significantly better in terms of CPU usage than the standard method in most of the experimental cases with the maximum gap between SFM and STD in term of CPU load is around 4% in the scenario of a single switch. Once the number of switches increases to more than 20, CPU usage consumed by SFM equals only half the value of STD. Moreover, SFM has the ability to select a particular flow to query its statistic. Therefore, a detailed investigation of SFM is conducted by first analyzing the total number of control plane messages that are exchanged between the switches and the controller in each case. Then, the resulting equations are used to identify the main influence factors on the performance of the SFM approaches. The results show that the group that has the highest polling frequency is defined as a main influence factor on the CPU usage. Accordingly, multiple groups (e.g., from 10 to 40) with the same such interval lead to a higher CPU utilization than the standard method. Furthermore, investigations regarding the amount of memory that is allocated to the Java Virtual Machine indicate that the ONOS controller can not handle more than 216,000 flows in request/reply statistics information if the dedicated memory is 2GB. With the configuration of 4GB, the maximum number of flows is 408,000 flows.

To summarize, we see that flow monitoring in SDN networks mainly based on OpenFlow flow statistics messages. Several influence parameters on flow monitoring are determined, such as polling interval, the number of flows and switches, as well as the size ratios of flow classes. In addition, not only the hardware regarding the CPU is relevant, but also the amount of memory also needs to be taken into account. Within presented approaches, AFM shows drawback in term of controller CPU utilization as long as the information of each flow has to be gathered by a single message. In this context, the proposed approach SFM is better than AFM and for specific scenario with particular combination of mentioned parameters, SFM also outperform the standard flow monitoring method in the ONOS controller. Given the results presented in this chapter, an operator can identify appropriate parameter combinations based on the composition of the network and flow characteristics.

5 Conclusion

The development of information technology in many fields results in new requirements for underlying network infrastructures in terms of availability, connectivity, and flexibility. Besides, applications are deployed and updated in a short time. They create a diversity of traffic patterns and an enormous number of exchanged data. Therefore, networks need the ability to adaptively reconfigure devices and to control traffic flows based on the condition of the network and requirements from applications. Likewise, saving time and less complexity when expending the scale of the network are also necessary. In that context, the advent of Software Defined Networking (SDN), which was initiated in 2008, is a promising solution for high demand on resources, unpredictable traffic patterns, rapid and automatic network reconfiguration.

In this monograph, we aim to determine performance factors for the operation of SDN-enabled networks, which covers all components of its layered architecture. Different methods including software-based approaches and using commercial dedicated hardware are applied to analyze and evaluate relevant metrics. Besides, corresponding evaluation results are provided alongside experimental guidelines which allows administrators to assess SDN elements before applying the SDN paradigm. Both appropriate hardware devices and instances of software controllers are considered.

To bring to customers a good quality of services, the requirements of network performance for each application/service such as delay, packet loss, or bandwidth should be met. Also, the isolation between services when sharing the same physical resources needs to be guaranteed. When applying new network infrastructure like SDN, it is important that the previous outlined conditions are obtained.

Chapter 2 describes local testbeds to investigate how good the isolation between virtual networks is implemented on SDN forwarding devices. The violation of the isolation is represented by lost packets in a network when congestion occurs. Main influence parameters on the isolation performance are identified and investigated with switches of several vendors. It is shown that besides the overall load on the outgoing switch port also the configured rate guarantees per virtual network have a significant impact on the number of lost packets. The measurements further show a correlation between the degree of congestion on the outgoing port, the configured rate limits, and the delay until the switch reacts to the violation of the resource isolation. In particular, among the experimented switches, the HP 2029 shows that with a specific configuration and condition, virtual networks do not impact the others when congestion occurs. Based on these investigations an SDN switch providing proper isolation was selected for a video streaming scenario. Consequently, the presented method in this chapter can be used to find appropriate configurations for the involved hardware devices with respect to resource isolation.

Another aspect is the impact of specific control mechanisms of the control plane on the data plane performance. SDN controllers are responsible for management of data plane devices and decision-making for packet forwarding in the network. To instruct switches to forward traffic flows, SDN controller uses specific messages called FlowMod to install flow table entries in the switches. Since such operations are handled in the slow path of the switches, the corresponding processing times constitute an important performance indicator for switches.

Chapter 3 presents and compares two mechanisms for evaluating the performance of OpenFlow switches in terms of processing FlowMod messages. On the one hand, we use a software-based approach featuring a module for the OpenDaylight and Ryu controllers. On the other hand, the Spirent C1 dedicated testing platform is utilized. Additionally, we use wiretap devices in order to obtain highly precise measurements. Result of experiments show that both mechanisms achieve an accuracy in the sub-millisecond range when compared to measurements performed with dedicated capture cards at wiretaps. In addition to the benefits regarding costs and ease of use, the software-based controller module also has the capability to run during normal operation. While different switches have their own behavior, implementation details of the SDN controller also have an impact on the overall FlowMod processing time performance due to sender-side behavior. In particular, a tradeoff between fast installation and precision needs to be taken into account in case of QUANTA switch. In this case, the confirmation of installation is sent to the controller before the rules are actually installed in the switches which might lead to an incorrect action at the controller. We also confirm that based on the sensitivity of switches with regard to delays in conjunction with round trip time (RTT) and setup time at the controller is sufficient for an accurate estimation of installation time in the switch. The presented results provide an overview of such delays for a developer when building an application related to flow installation time. Besides, it can be referred by a network operator when selecting suitable data plane hardware for his network considering flow installation time as a key criterion.

SDN controllers also manage activated traffic flows exchanging in SDN networks. The information of such flows like the number of packets or their lifetime is collected by using the request-response method. This kind of information provides the current status of flows in the network and can be leveraged as the input of algorithms for traffic engineering, e.g., load balancing or multipath routing based on the bandwidth of each flow. SDN controller regularly requests network devices to send statistics of all flows in their OpenFlow tables. The duration between requests called *polling interval* has a fixed value and pre-defined before running a controller instance. However, using the same polling interval for all flows does not take into account the heterogeneity of real world traffic and thus results in an imbalance between monitoring accuracy and control plane overhead. In particular, high frequent querying results in a high resource consumption at the controller in large networks due to an enormous number of requests/responses per second.

We proposed a monitoring method that aims to balance flow statistics' accuracy and performance of SDN controllers in Chapter 4. Especially, network administrators can use this method to classify flows according to their individual requirements in terms of monitoring frequency, e.g., less frequent polling of elephant flows and frequent polling of QoS sensitive Voice over IP (VoIP) connections. This provides more flexible and more proactive flow monitoring than the default setting in all SDN controllers, which is based on regular and flow-independent polling. Then, the developed method is compared with others default methods in the ONOS controller in terms of controller CPU utilization and memory consumption. The measurement results show that our proposal significantly improves resources utilization as compared with the built-in algorithms in most cases. Additionally, investigations regarding the amount of memory that is allocated to the Java Virtual Machine indicate that not only the CPU is relevant but that the memory also becomes an important criterion when choosing an SDN controller. Based on these results and the composition of network and flows characteristics, network operators can identify appropriate memory configuration for his controller as well as a suitable method for flow monitoring.

The work presented in this monograph can be considered as technical guidelines for understanding the operation of SDN networks and evaluating the performance of SDN components. Namely, influence factors on the performance of the control and data planes are identified, software-based performance measurement with similar accuracy as dedicated commercial hardware is described, or methods to choose appropriate hardware and suitable controller instance for specific performance criteria are presented. In addition, based on the results and proposed algorithms, the optimization potential of an SDN network can be taken into account by both, controller developers who aim to improve the general performance of their controller as well as network operators who want to maximize compatibility and reliability of the components in their particular networks. Applying introduced methods and mechanisms facilitates better network management and effective operation to enhance network performance and monitoring. This work can be adapted to investigate others implementations of SDN networks, e.g., distributed architecture with several instances of controllers. In this context, the mechanism to exchanged information between controllers and switches might be different, however, it is still relying on the operation of OpenFlow protocols, which is thoroughly examined in this monograph. Besides, the evaluation related to data plane devices can be applied to inspect packet processing time of programmable switches. Finally, further research can extend this work with new versions/instances of controllers and switches to gain a better insight into the operation of SDN networks.

Acronyms

API	Application Programming Interface.
CLI	Command Line Interface.
COTS	Commercial Off-The-Shelf.
DASH	Dynamic Adaptive Streaming over HTTP.
FIB	Forwarding Information Base.
HAS	HTTP-based Adaptive Streaming.
IoT	Internet of Thing.
ISP	Internet Service Provider.
JSON	JavaScript Object Notation.
JVM	Java Virtual Machine.
NBI	Northbound Interface.
NFV	Network Functions Virtualization.
NICs	Network Interface Cards.

NSM Network Management System.

NV Network Virtualization.

ONF Open Networking Foundation.

OSPF Open Shortest Path First.

OTT Over The Top.

QoE Quality of Experience.

QoS Quality of Service.

REST Representational State Transfer.

RIP Routing Information Protocol.

RMON Remote Monitoring.

SBI Southbound Interface.

SDN Software Defined Networking.

SLA Service Level Agreement.

SMS Short Message Service.

SNMP Simple Network Management Protocol.

SSH Secure Shell.

SSIM Structual SIMiarity.

Telnet Terminal Network.

VCP Video Control Plane.

VLAN Virtual LAN.

VoIP Voice over IP.

VPN Virtual Private Network.

Bibliography and References

Bibliography of the Author

Journal Papers

[1] G. Cofano, L. De Cicco, T. Zinner, A. Nguyen-Ngoc, P. Tran-Gia, and S. Mascolo, "Design and performance evaluation of network-assisted control strategies for http adaptive streaming", *ACM Transactions on Multimedia Computing, Communications, and Applications*, 2017.

Conference Papers

[2] A. Nguyen-Ngoc, S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, and M. Jarschel, "Investigating isolation between virtual networks in case of congestion for a pronto 3290 switch", in *Workshop on Software-Defined Networking and Network Function Virtualization for Flexible Network Management (SDNFlex 2015)*, Cottbus, Germany, Mar. 2015.

[3] S. Lange, A. Nguyen-Ngoc, S. Gebert, T. Zinner, M. Jarschel, A. Koepsel, M. Sune, D. Raumer, S. Gallenmüller, G. Carle, and P. Tran-Gia, "Performance benchmarking of a software-based lte sgw", in *2nd International Workshop on Management of SDN and NFV Systems*, Barcelona, Spain, Nov. 2015.

[4] G. Cofano, L. De Cicco, T. Zinner, A. Nguyen-Ngoc, P. Tran-Gia, and S. Mascolo, "Design and experimental evaluation of network-assisted strategies for http adaptive video streaming", in *Best Student Paper Award, ACM Multimedia Systems Conference (MMSys)*, Klagenfurt, Austria, May 2016.

[5] A. Nguyen-Ngoc, S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, and M. Jarschel, "Performance evaluation mechanisms for flowmod message processing in openflow switches", in *IEEE Sixth International Conference on Communications and Electronics*, Ha Long City, Vietnam, Jul. 2016.

[6] A. Nguyen-Ngoc, B. Sphend, S. Gebert, T. Zinner, and P. Tran-Gia, "Evaluation of adaptive flow monitoring in onos", in *ONOS Build 2016, Poster Session*, Paris, France, Nov. 2016.

[7] S. Gebert, S. Geissler, T. Zinner, A. Nguyen-Ngoc, S. Lange, and P. Tran-Gia, "Zoom: Lightweight sdn-based elephant detection", in *First International Workshop on Programmability for Cloud Networks and Applications (PROCON)*, Sep. 2016.

[8] A. Nguyen-Ngoc, S. Lange, T. Zinner, M. Seufert, P. Tran-Gia, N. Aerts, and D. Hock, "Performance evaluation of selective flow monitoring in the onos controller", in *4th International Workshop on Management of SDN and NFV Systems (ManSDN/NFV)*, Tokio, Japan, Nov. 2017.

[9] A. Nguyen-Ngoc, S. Lange, G. Stefan, T. Zinner, and P. Tran-Gia, "Estimating the flow rule installation time of sdn switches when facing control plane delay", in *19th International GI/ITG Conference on "Measurement, Modelling and Evaluation of Computing Systems" (MMB)*, Erlangen, Germany, Feb. 2018.

[10] A. Nguyen-Ngoc, R. Simon, S. Lange, G. Stefan, T. Zinner, and P. Tran-Gia, "Benchmarking the onos controller with ofcprobe", in *7th International Conference on Communications and Electronics (ICCE)*, Hue, Vietnam, Jul. 2018.

General References

[11] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer, “Interfaces, attributes, and use cases: A compass for sdn”, *IEEE Communications Magazine*, vol. 52, pp. 210–217, Jun. 2014.

[12] *Traditional vs software defined networking*, IPknowledge, 2014. [Online]. Available: `http://www.ipknowledge.net/wp-content/uploads/2014/12/SDN.pdf`.

[13] M. Jammal, T. Singh, A. Shami, R. Asal, and Y. Li, “Software-defined networking: State of the art and research challenges”, *Computer Networks*, vol. 72, pp. 74–98, 2014.

[14] H. Wen, P. K. Tiwary, and T. Le-Ngoc, “Network virtualization: Overview”, in *Wireless Virtualization*. Cham: Springer International Publishing, 2013, pp. 5–10, ISBN: 978-3-319-01291-9. DOI: `10.1007/978-3-319-01291-9_2`. [Online]. Available: `https://doi.org/10.1007/978-3-319-01291-9_2`.

[15] G. Ferro, *What is openflow*, [Online]. Available at http://content.ipspace.net/get/what%20Is%20OpenFlow.pdf, May 2012.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: Enabling innovation in campus networks”, *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008, ISSN: 0146-4833. DOI: `10.1145/1355734.1355746`. [Online]. Available: `http://doi.acm.org/10.1145/1355734.1355746`.

[17] Open Networking Foundation, *Software-defined networking (sdn) definition*, [Online]. Available at https://www.opennetworking.org/sdn-definition/, 2017.

[18] D. A, S. J.H, H. R, K. H, W. W, D. L, G. R, and H. J, *Forwarding and control element separation (forces) protocol specification*, [Online]. Available at https://tools.ietf.org/html/rfc5810, Mar. 2010.

[19] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015, ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2371999.

[20] C. T. Support, *Comparing traffic policing and traffic shaping for bandwidth limiting*, [Online]. Available at https://www.cisco.com/c/en/us/support/docs/quality-of-service-qos/qos-policing/19645-policevsshape.html, May 2014.

[21] IEEE, *IEEE802.3x Specification for 802.3 Full Duplex Operation*, 1998.

[22] Y. C. Hoong, *Flow control on gigabit ethernet interfaces*, [Online]. Available at http://www.itcertnotes.com/2011/06/flow-control-on-gigabit-ethernet.html, Jun. 2011.

[23] S. Bhatia, M. Motiwala, W. Muhlbauer, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford, "Hosting virtual networks on commodity hardware", vol. GT-CS-07-10, 2008/// 2008.

[24] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In vini veritas: Realistic and controlled network experimentation", *SIGCOMM Comput. Commun. Rev.*, vol. 36, no. 4, pp. 3–14, Aug. 2006, ISSN: 0146-4833. DOI: 10.1145/1151659.1159916. [Online]. Available: http://doi.acm.org/10.1145/1151659.1159916.

[25] S. Ahn, S. Lee, S. Yoo, D. Park, D. Kim, and C. Yoo, "Isolation schemes of virtual network platform for cloud computing", *KSII Transactions on Internet and Information Systems*, vol. 6, no. 11, pp. 2764–2783, Nov. 2012, ISSN: 1976-7277.

[26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization”, *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 164–177, Oct. 2003, ISSN: 0163-5980. DOI: 10.1145/1165389.945462. [Online]. Available: http://doi.acm.org/10.1145/1165389.945462.

[27] N. Fernandes, M. D. D. Moreira, I. Moraes, L. H. Ferraz, R. Couto, H. E. T. Carvalho, M. Campista, L. Costa, and O. C. M. B. Duarte, “Virtual networks: Isolation, performance, and trends”, vol. 66, pp. 339–355, Jun. 2011.

[28] O. Hohlfeld, “Impact of buffering on quality of experience”, PhD thesis, Technische Universität Berlin, 2013.

[29] M. J. Karol, M. G. Hluchyj, and S. P. Morgan, “Input versus output queueing on a space-division packet switch”, *Communications, IEEE Transactions on*, vol. 35, no. 12, pp. 1347–1356, 1987.

[30] N. McKeown, A. Mekkittikul, V. Anantharam, and J. Walrand, “Achieving 100% throughput in an input-queued switch”, *Communications, IEEE Transactions on*, vol. 47, no. 8, pp. 1260–1267, 1999.

[31] C. Clos, “A study of non-blocking switching networks”, *The Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, Mar. 1953, ISSN: 0005-8580. DOI: 10.1002/j.1538-7305.1953.tb01433.x.

[32] T. Hoßfeld, K. Leibnitz, and A. Nakao, “Modeling of Modern Router Architectures Supporting Network Virtualization”, in *2nd International Workshop on the Network of the Future (FutureNet II) in conjunction with IEEE GLOBECOM*, 2009.

[33] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy, “Fairness issues in software virtual routers”, in *Proceedings of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2008.

[34] R. Durner, A. Blenk, and W. Kellerer, "Performance study of dynamic qos management for openflow-enabled sdn switches", in *2015 IEEE 23rd International Symposium on Quality of Service (IWQoS)*, Jun. 2015, pp. 177–182. DOI: 10.1109/IWQoS.2015.7404730.

[35] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran, "Probe and adapt: Rate adaptation for http video streaming at scale", *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 4, pp. 719–733, Apr. 2014, ISSN: 0733-8716. DOI: 10.1109/JSAC.2014.140405.

[36] L. D. Cicco, V. Caldaralo, V. Palmisano, and S. Mascolo, "Elastic: A client-side controller for dynamic adaptive streaming over http (dash)", in *2013 20th International Packet Video Workshop*, Dec. 2013, pp. 1–8. DOI: 10.1109/PV.2013.6691442.

[37] L. De Cicco, V. Caldaralo, V. Palmisano, and S. Mascolo, "Tapas: A tool for rapid prototyping of adaptive streaming algorithms", in *Proceedings of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming*, ser. VideoNext '14, Sydney, Australia: ACM, 2014, pp. 1–6, ISBN: 978-1-4503-3281-1. DOI: 10.1145/2676652.2676654. [Online]. Available: http://doi.acm.org/10.1145/2676652.2676654.

[38] P. Ni, R. Eg, A. Eichhorn, C. Griwodz, and P. Halvorsen, "Flicker effects in adaptive video streaming to handheld devices", in *Proceedings of the 19th ACM International Conference on Multimedia*, ser. MM '11, Scottsdale, Arizona, USA: ACM, 2011, pp. 463–472, ISBN: 978-1-4503-0616-4. DOI: 10.1145/2072298.2072359. [Online]. Available: http://doi.acm.org/10.1145/2072298.2072359.

[39] T. Hoßfeld, M. Seufert, C. Sieber, and T. Zinner, "Assessing effect sizes of influence factors towards a qoe model for http adaptive streaming", in *2014 Sixth International Workshop on Quality of Multimedia Experience (QoMEX)*, Sep. 2014, pp. 111–116. DOI: 10.1109/QoMEX.2014.6982305.

[40] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, “On controller performance in software-defined networks”, in *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, San Jose, CA: USENIX, 2012. [Online]. Available: https://www.usenix.org/conference/hot-ice12-0/controller-performance-software-defined-networks.

[41] Pica8, *Sdn system performance*, [Online]. Available at http://www.pica8.com/pica8-deep-dive/sdn-system-performance/.

[42] P. Goransson, C. Black, and T. Culver, *Software Defined Networks, Second Edition: A Comprehensive Approach*, 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016, ISBN: 0128045558, 9780128045558.

[43] L. Doyle, *The role of northbound apis in an sdn environment*, [Online]. Available at http://searchsdn.techtarget.com/answer/The-role-of-northbound-APIs-in-an-SDN-environment, 2013.

[44] J. H. Cox, J. Chung, S. Donovan, J. Ivey, R. J. Clark, G. Riley, and H. L. Owen, “Advancing software-defined networks: A survey”, *IEEE Access*, vol. 5, pp. 25 487–25 526, 2017. DOI: 10.1109/ACCESS.2017.2762291.

[45] M. Bjorklund, J. Schoenwaelder, and A. Bierman, *Network configuration pro-
tocol (netconf)*, [Online]. Available at https://tools.ietf.org/html/rfc6241, Jun. 2011.

[46] P. Saint-Andre, *Extensible messaging and presence protocol (xmpp): Core*, [Online]. Available at https://tools.ietf.org/html/rfc6120, Mar. 2011.

[47] Spirent, *Openflow performance testing white paper*, [Online]. Available at https://www.spirent.com/~media/White%20Papers/Broadband/PAB/OpenFlow-Performance-Testing_WhitePaper.pdf, Mar. 2015.

[48] J. W. King, *Software-defined networking introduction to openflow*, [Online]. Available at https://www.slideshare.net/joelwking/introduction-to-openflow-41257742, Nov. 2014.

[49] A. Capone, *From dumb to smarter switches in software defined networks: An*
overview of data plane evolution, [Online]. Available at http://www.beba-project.eu/presentations/SDN-tutorial-CAMAD-v2_full.pdf, Dec. 2014.

[50] C. S. Hong, *Manageability of future internet*, [Online]. Available at http://slideplayer.com/slide/9802157/, Apr. 2015.

[51] *OpenFlow Switch Specification v1.5.0*, [Online]. Available at https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf, Open Networking Foundation, 2014.

[52] M. Kuzniar, M. Canini, and D. Kostic, “Often testing openflow networks”, in *2012 European Workshop on Software Defined Networking*, Oct. 2012, pp. 54–60. DOI: 10.1109/EWSDN.2012.21.

[53] Projectfloodlight, [Online]. Available at http://www.projectfloodlight.org/oftest/.

[54] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, “Openflow switching: Data plane performance”, in *2010 IEEE International Conference on Communications*, May 2010, pp. 1–5. DOI: 10.1109/ICC.2010.5502016.

[55] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, “Performance characteristics of virtual switching”, in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, Oct. 2014, pp. 120–125. DOI: 10.1109/CloudNet.2014.6968979.

[56] OpenDayLight, *Openflow:testing*, [Online]. Available at https://wiki.opendaylight.org/view/Openflow Testing#Tools, Aug. 2016.

[57] M. Kuzniar, P. Peresini, and D. Kostic, “What you need to know about sdn control and data planes”, in *EPFL Technical Report EPFL-REPORT-199497*, 2014.

[58] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, “Oflops: An open framework for openflow switch evaluation”, in *Proceedings of the 13th International Conference on Passive and Active Measurement*, ser. PAM’12, Vienna, Austria: Springer-Verlag, 2012, pp. 85–95, ISBN: 978-3-642-28536-3. DOI: 10.1007/978-3-642-28537-0_9. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-28537-0_9.

[59] C. Rotsos, G. Antichi, M. Bruyere, P. Owezarski, and A. W. Moore, “Oflops-turbo: Testing the next-generation openflow switch”, in *2015 IEEE International Conference on Communications (ICC)*, Jun. 2015, pp. 5571–5576. DOI: 10.1109/ICC.2015.7249210.

[60] M. Shahbaz, G. Antichi, Y. Geng, N. Zilberman, A. Covington, M. Bruyere, N. Feamster, N. McKeown, B. Felderman, M. Blott, A. W. Moore, and P. Owezarski, “Architecture for an open source network tester”, in *Architectures for Networking and Communications Systems*, Oct. 2013, pp. 123–124. DOI: 10.1109/ANCS.2013.6665194.

[61] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, “Modeling and performance evaluation of an openflow architecture”, in *2011 23rd International Teletraffic Congress (ITC)*, Sep. 2011, pp. 1–7.

[62] 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)*, Dec. 2013, pp. 1397–1402. DOI: 10.1109/GLOCOM.2013.6831269.

[63] M. Jarschel, T. Zinner, T. Höhn, and P. Tran-Gia, “On the accuracy of leveraging sdn for passive network measurements”, in *2013 Australasian*

Telecommunication Networks and Applications Conference (ATNAC), Nov. 2013, pp. 41–46. DOI: `10.1109/ATNAC.2013.6705354`.

[64] 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: `https://doi.org/10.1007/978-3-319-15509-8_26`.

[65] D. Y. Huang, K. Yocum, and A. C. Snoeren, "High-fidelity switch models for software-defined network emulation", in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

[66] S. Geissler, S. Herrnleben, R. Bauer, S. Gebert, T. Zinner, and M. Jarschel, "Tablevisor 2.0: Towards full-featured, scalable and hardware-independent multi table processing", in *Network Softwarization (NetSoft), 2017 IEEE Conference on*, 2017.

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

[68] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency in software defined networks", in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016.

[69] L. Statistics, *Spearman's rank-order correlation*, [Online]. Available at https://statistics.laerd.com/statistical-guides/spearmans-rank-order-correlation-statistical-guide.php, Mar. 2013.

[70] J. W.-K. Hong, *Software defined networking: Traffic monitoring and analysis*, [Online]. Available at http://slideplayer.com/slide/7395148/, Jan. 2015.

[71] B. Landfeldt, P. Sookavatana, and A. Seneviratne, “The case for a hybrid passive/active network monitoring scheme in the wireless internet”, in *Proceedings IEEE International Conference on Networks 2000 (ICON 2000). Networking Trends and Challenges in the New Millennium*, 2000, pp. 139–143.

[72] A. Ciuffoletti and M. Polychronakis, “Architecture of a network monitoring element”, in *Euro-Par 2006: Parallel Processing*, W. Lehner, N. Meyer, A. Streit, and C. Stewart, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 5–14, ISBN: 978-3-540-72337-0.

[73] I. ImageStream Internet Solutions, *Network monitoring white paper*, [Online]. Available at http://www.telogic.com.sg/PDF/Monitoring_White_Paper.pdf, Apr. 2003.

[74] Cisco, *Netflow docwiki*, [Online]. Available at http://docwiki.cisco.com/wiki/NetFlow, May 2013.

[75] J. du Toit, *Active vs. passive network monitoring: An infographic*, [Online]. Available at https://www.irisns.com/active-vs-passive-network-monitoring-an-infographic/, Apr. 2016.

[76] L. Cottrell, *Passive vs. active monitoring*, [Online]. Available at https://www.slac.stanford.edu/comp/net/wan-mon/passive-vs-active.html, Mar. 2001.

[77] M. Zangrilli and B. B. Lowekamp, “Using passive traces of application traffic in a network monitoring system”, in *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, ser. HPDC ’04, Washington, DC, USA: IEEE Computer Society, 2004, pp. 77–86, ISBN: 0-7803-2175-4. DOI: 10.1109/HPDC.2004.38. [Online]. Available: http://dx.doi.org/10.1109/HPDC.2004.38.

[78] O. N. Foundation, *Openflow switch specification v1.5.0*, [Online]. Available at https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.0.pdf, 2012.

[79] F. Tean, *Openflow message layer*, [Online]. Available at http://flowgrammable.org/sdn/openflow/message-layer/statsrequest/, 2014.

[80] Cisco, *Introduction to cisco ios netflow - a technical overview*, [Online]. Available at https://www.cisco.com/c/en/us/products/collateral/ios-nx-os-software/ios-netflow/, May 2012.

[81] D. Pajin and P. V. Vuletić, "Of2nf: Flow monitoring in openflow environment using netflow/ipfix", in *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, Apr. 2015, pp. 1–5. DOI: 10.1109/NETSOFT.2015.7116138.

[82] J. Suárez-Varela and P. Barlet-Ros, "Towards a netflow implementation for openflow software-defined networks", in *2017 29th International Teletraffic Congress (ITC 29)*, Sep. 2017.

[83] J. Suárez-Varela and P. Barlet-Ros, "Sbar: Sdn flow-based monitoring and application recognition", in *2018 Symposium on SDN Research*, Mar. 2018.

[84] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba, "PayLess: A Low Cost Netowrk Monitoring Framework for Software Defined Networks", in *14th IEEE/IFIP Network Operations and Management Symposium (NOMS 2014)*, 2014. [Online]. Available: http://www.bibsonomy.org/bibtex/26900dadb678b1428ee1078e98fe3cf89/chesteve.

[85] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, "OpenNetMon: Network Monitoring in OpenFLow Software-Defined Networks", in *Network Operations and Management Symposium (NOMS)*, 2014.

[86] Z. Su, T. Wang, Y. Xia, and M. Hamdi, “Flowcover: Low-cost flow monitoring scheme in software defined networks”, in *2014 IEEE Global Communications Conference*, Dec. 2014.

[87] X. Thien Phan and K. Fukuda, “Sdn-mon: Fine-grained traffic monitoring framework in software-defined networks”, vol. 25, pp. 182–190, Feb. 2017.

[88] D. Raumer, L. Schwaighofer, and G. Carle, “Monsamp: A distributed sdn application for qos monitoring”, in *2014 Federated Conference on Computer Science and Information Systems*, Sep. 2014, pp. 961–968. DOI: 10.15439/2014F175.

[89] M. Hartung and M. Körner, “Softmon - traffic monitoring for sdn”, vol. 110, pp. 516–523, Dec. 2017.

[90] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, “Flowsense: Monitoring network utilization with zero measurement cost”, in *Passive and Active Measurement*, M. Roughan and R. Chang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 31–41, ISBN: 978-3-642-36516-4.

[91] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, “Opentm: Traffic matrix estimator for openflow networks”, in *PAM*, 2010.

[92] L. Yuan, C. N. Chuah, and P. Mohapatra, “Progme: Towards programmable network measurement”, *IEEE/ACM Transactions on Networking*, vol. 19, no. 1, pp. 115–128, Feb. 2011, ISSN: 1063-6692. DOI: 10.1109/TNET.2010.2066987.

[93] T. Choi, *Adaptive flow monitoring & selective dpi for onos*, [Online]. Available at https://wiki.onosproject.org/display/ONOS/OPEN-TAM%3ATrafficAnalysisandMonitoring, Mar. 2016.

ISSN 1432-8801