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

An Assessment of Applications and Performance Analysis of Software Defined Networking

Michael Jarschel

Würzburger Beiträge zur
Leistungsbewertung Verteilter Systeme

Bericht 03/14

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 vom Autor.
Gesetzt in LaTeX Computer Modern 9pt.

ISSN 1432-8801

An Assessment of Applications and Performance Analysis of Software Defined Networking

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

vorgelegt von

Michael Jarschel

aus

Aschaffenburg

Würzburg 2014

Eingereicht am: 12.03.2014
bei der Fakultät für Mathematik und Informatik
1. Gutachter: Prof. Dr.-Ing. P. Tran-Gia
2. Gutachter: Prof. Dr. Paul Müller
Tag der mündlichen Prüfung: 28.05.2014

Danksagung

Auf dem Weg zur Erstellung dieser Arbeit wurde ich von vielen Menschen auf diverseste Weisen begleitet und unterstützt. Ich möchte diese Gelegenheit nutzen und mich an dieser Stelle diesen Menschen bedanken. Zu allererst ist hier selbstverständlich mein Doktorvater Prof. Phuoc Tran-Gia zu nennen, der mir die Möglichkeit eröffnet hat, eine Promotion am Lehrstuhl für Kommunikationsnetze anzustreben. Während dieser Zeit sorgte er stets dafür, dass mir und meinen Kollegen ein ausgezeichnetes Arbeitsumfeld geboten wurde, in dem es möglich war neue Forschungsideen zu entwickeln und diese im Anschluss auf internationalen Fachkonferenzen der Wissenschaftsgemeinschaft zu präsentieren. Auf diese Art und Weise war es mir möglich meine fachlichen und sozialen Kompetenzen entscheidend zu erweitern.

Insbesondere im fachlichen Bereich profitierte ich auch von den Diskussionen und gemeinsamen Veröffentlichungen mit den Forschungsgruppenleitern am Lehrstuhl: Dr. Tobias Hoßfeld, Dr. Rastin Pries und Dr. Thomas Zinner. Daher möchte ich mich auch bei ihnen für deren Gespräche und Hinweise in vielen Bereichen herzlichst bedanken.

Prof. Paul Müller möchte ich für die Erstellen des Zweitgutachtens sowie die wertvollen Kommentare zur Arbeit selbst danken. Ebenso gilt mein Dank den Prüfern in meiner Disputation Prof. Samuel Kounev sowie Prof. Reiner Kolla, der als Prüfungsvorsitzender fungierte.

In meinem Kernforschungsgebiet Software Defined Networking möchte ich mich für den wertvollen Austausch mit Prof. Wolfgang Kellerer bedanken, der in einer Journal-Veröffentlichung mündete.

Generell möchte ich mich für die Atmosphäre am Lehrstuhl und gegenseitige

Unterstützung bei allen aktuellen und ehemaligen Mitarbeitern des Lehrstuhls für Informatik III bedanken, die meine Zeit dort nicht nur zu einer angenehmen, sondern auch prägenden Erfahrung gemacht haben. Dies sind insbesondere Prof. Michael Menth, Dr. Michael Duelli, Dr. Robert Henjes, Dr. Dominik Klein, Dr. Frank Lehrieder, Dr. Simon Oechsner, Dr. Daniel Schlosser, Dr. Barbara Staehle, Dr. Dirk Staehle, Valentin Burger, Steffen Gebert, Matthias Hirth, David Hock, Christian Schwartz, Michael Seufert und Florian Wamser.

Mein Dank gilt auch allen Studenten, die ich über die Jahre hin betreute habe und die mich durch ihren Einsatz und ihr Engagement immer aufs Neue gefordert haben. Ein besonderes Dankeschön geht an meine beiden langjährigen studentischen Hilfskräfte Christopher Metter und Thomas Höhn, die mich stets tatkräftig unterstützt haben.

Zu guter Letzt möchte ich mich bei meiner Familie bedanken ohne deren bedingungslose Unterstützung diese Arbeit nicht möglich gewesen wäre.

Contents

1 Introduction

In 2008, the McKeown group out of Stanford University presented a new protocol called "OpenFlow" [11] to the scientific community. Its primary use case and feature is to enable the removal of the local network control plane from a network device to a central remote software entity.

While the concept of centralized control is far from a novel concept in communication networks and previous attempts had largely been ignored by the scientific community as well as the industry, OpenFlow soon created a hype. There were two main reasons for this development. The first reason is that at the time cloud service providers were facing enormous challenges in terms of network operation within their data centers due to the increasing popularity of cloud-based services. The way networks had been designed and operated for the previous close to 30 years was no longer flexible enough to sustain a viable operation of these new services. The second reason is the fact that OpenFlow was implemented straight away, not only in software but in prototypical hardware as well. This allowed potential adopters of this technology to experiment with it almost immediately. The most prominent example of this is Google. The network research team at Google started to design new networks based on the SDN approach right from the start and has since rolled out a global backbone network based on it [12].

However, as experimentation progressed it soon became clear that OpenFlow in its initial form would not be sufficient to meet all of the requirements put forth by the early adopters in the data center domain, let alone other networking domains. Therefore, the idea of OpenFlow was generalized in the concept of SDN.

The discussion about what actually falls under the term SDN is still ongoing and with it a large number of research questions are still unanswered. One of the key issues is the performance of an SDN system. The solution to this problem cannot be generalized. It depends on a variety of factors. First, there is the mode of operation, i.e. the question when the control plane is actually involved. Does it have to react to new traffic or is there a specific set of pre-defined rules and the control plane only needs to act in case of changes in the network, e.g., because of a failure? The second factor is the scenario in question. The main deciding factor here is whether the network topology is likely to change often, e.g., due to virtual machine migrations in a data center, or remain relatively constant as inside a WAN backbone. Finally, the desired granularity plays an important role as it determines the number of rules and the size of the network state the control plane needs to handle as well as the size of the rule tables.
A second issue is the question of applicability. Are there areas that can benefit from adopting the SDN concept and which areas are better suited to the classical fully distributed approach? While a central control entity may simplify and improve the calculation of paths in a network, it also may not be as resilient.

Another open question is whether there are new approaches beyond classical networking that have now become feasible due to the flexibility gained from an external software control plane. In particular, is the long desired vision of an application-aware network with regard to the user's experience finally within reach and if so, what has to be done to get there?
The goal of this monograph is to widen the understanding of the SDN concept and the issues that come with it. The first step towards this is to provide a thorough definition of SDN and its interfaces in the following section as well as introduce potential use cases. The second step is to provide an insight into the performance of current SDN systems and deployments on the example of OpenFlow in order to determine bottlenecks and pitfalls to avoid in the future. Beyond the performance, this monograph takes a look at the benefits that can be gained by applying the SDN approach to classical networking problem such as monitoring and what the potential trade-offs are. Finally, we show a way to enable a

QoE-aware network using SDN and highlight the necessary requirements for it to become a reality.

1.1 Definition and Use Cases of SDN

The term Software Defined Networking is prevalent in the current discussion about future communication networks. Like with any new term or paradigm, however, no consistent definition regarding this technology has formed. The fragmented view on SDN results in legacy products being passed off by equipment vendors as SDN, academics mixing up the attributes of SDN with those of network virtualization, and users not fully understanding the benefits. Therefore, we attempt to give a thorough definition of SDN and its interfaces as well as a list of its key attributes. Furthermore, a mapping of interfaces and attributes to SDN use cases is provided, highlighting the relevance of the interfaces and attributes for each scenario. This section is mainly based on and taken from [1].

1.1.1 Principles of Software Defined Networking

How networks are currently structured and operated poses a significant financial issue to internet service providers and, in fact, has become a handicap for progress in the cloud and service provider space. SDN [13] enables a programmable network control and offers a solution to a variety of use cases. The success stories of these bottom-up SDN solutions have led to a shift in the way operators and vendors perceive the network. In the following, we define four basic principles of SDN. Each of these principles is mandatory for classifying a technology as SDN.

Separation of Control- and Data Plane

The physical separation of control- and forwarding- or data plane is the best-known principle of SDN [13, 14]. It postulates the externalization of the control plane from a network device to an external control plane entity often called the "controller". In particular, this means that an internal software control plane, while it may still exist, is not enough to brand a device or technology as “Software Defined Networking”. The external controller has to have the ability to change the forwarding behavior of the network element directly. This enables several key benefits of SDN. Control- and data plane can be developed separately from each other, which lowers the entry-to-market hurdle, as a company no longer has to have expert knowledge in both areas. Moreover, the externalization of a software-based controller produces pressure on established hardware switch vendors, which are reduced to providing forwarding hardware only. This has already introduced new and disruptive start-ups to the market that have sped up innovation in the network. Even the market leader Cisco has reacted to this trend by introducing its own flavor of SDN with the Application Centric Infrastructure concept developed at the Spin-In company “Insieme” [15]. Customers are also enabled to “mix-and-match” products of different vendors and thus increase competition further. The switch vendors have reacted to that shift by forming the OpenDaylight project for an open SDN software platform. Challenges in this area are to find the appropriate control protocol for the specific scenario out of different protocols and protocol versions, and the appropriate forwarding elements which support this protocol.

Logically Centralized Control

The controller of an SDN network is a logically centralized entity, i.e. it can consist of multiple physical or virtual instances, but behaves like a single component. The global network information such a central controller possesses enables it to adapt its network policy with respect to routing and forwarding much better and faster than a system of traditional routers could. The realization of a logically centralized controller is challenging with respect to scalability depending on the specific scenario and network or virtual network size. Scalability can be achieved by implementing a centralized controller as a distributed system where the contained information has to be maintained consistently.

Open Interfaces

For SDN to reach its full potential in terms of flexibility and adaptability, it is fundamental that its interfaces are and remain open. A closed or proprietary interface limits component exchangeability and innovation. This is especially true for the interface between control- and data plane (Southbound Interface). In the absence of a standard open interface, one of the main SDN advantages – the interchangeability of network devices and control planes – would be taken away. This is also true for the remaining interfaces, which are discussed in more detail in Section 2.1. To maintain open interfaces might be challenging since vendors try to introduce proprietary interfaces or to bypass proprietary information via the open interface. This could generate additional value if entities of the same vendor are used, but also lead to deadlocks and performance bottlenecks in mixed operation.

Programmability

The fundamental paradigm shift in networking caused by SDN is represented by the introduction of network programmability. This is enabled by the external software controller and the open interfaces. The programmability principle is not limited to introducing new network features to the control plane but rather represents the ability to treat the network as a single programmable entity instead of an accumulation of devices that have to be configured individually. SDN can thus be regarded as a very suitable complement to network virtualization providing the control plane for an easy operation ('programming') of, e.g., virtual networks in network substrates or to control specific flows within a virtual network as possible applications. Here it is essential to find the appropriate abstraction level, which determines on the one hand the ease-of-use for network programmers, and on the other hand the abstraction overhead and therewith a possible performance degradation.

1.1.2 Definition and Significance of SDN Interfaces

The four key interfaces of Software Defined Networking are illustrated in Figure 1.1 for a generic, example network, consisting of three autonomous systems (AS): a conventional IP or legacy access network at the user end, an SDN-based transit-WAN, and an SDN-enabled data center network (cloud). While the definition of the South- and Northbound-API is consistent with the one put forth by the Open Networking Foundation [16], the definition of West- and Eastbound-API corresponds to the view of the author.

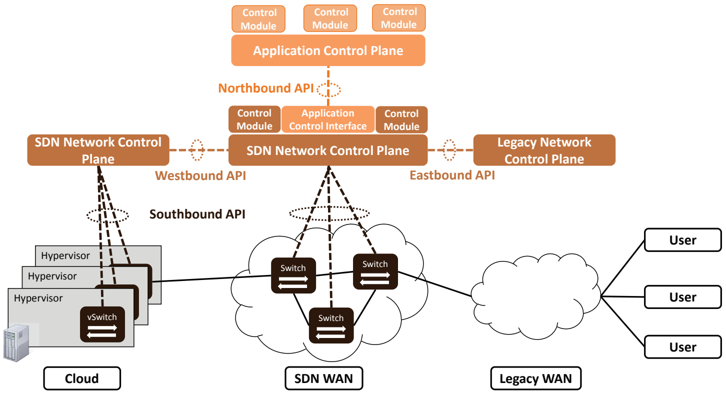


Figure 1.1: *Interfaces of a Software Defined Network*

Southbound-API

The Southbound-API represents the interface between control- and data plane. It is the enabler for the externalization of the control plane and therefore key to the corresponding SDN principle [14, 17]. Its realization is a standardized instruction set for the networking hardware. Implementation examples are the IETF ForCES Protocol [18] and most notably the OpenFlow protocol [11].

Northbound-API

SDN enables the exchange of information with applications running on top of the network. This information exchange is performed via the Northbound-API between the SDN controller and an "application control plane" [14, 17]. A universal, standardized Northboud API does not exist. Further, as the kind of information exchanged, its form and frequency depends on the targeted application and network such universal API is not useful. Standardization of this interface only makes sense for common scenarios, provided that all implementations are kept open. While the SDN controller can directly adapt the behavior of the network, the application controller adapts the behavior of the application using the network. It can be implemented as part of a single application instance to a central entity for the entire network responsible for all applications.

Westbound-API

The Westbound-API serves as an information conduit between SDN control planes of different network domains [12]. It allows the exchange of network state information to influence routing decisions of each controller but at the same time enabling the seamless setup of network flow across multiple domains. For the information exchange, standard inter-domain routing protocols like BGP could be used.

Eastbound-API

Communication with the control planes of non-SDN domains, e.g. a Multi-Protocol Label Switching (MPLS) control plane, uses the Eastbound-API [19]. The implementation of this interface depends on the technology used in the non-SDN domain. Essentially, a translation module between SDN and the legacy technology is required. This way, both domains should ideally appear to be fully compatible to each other. For example, the SDN domain should be able to use the routing protocol deployed between non-SDN domains or be able to react to Path

Computation Element Protocol (PCEP) messages requesting path setups from an MPLS domain.

1.1.3 Definition of SDN Features

The combination of these four open interfaces together with the core features we outline in the following makes SDN a very flexible and powerful tool for network control and operation. Later we show how matching of SDN's unique features to use cases can help a potential adopter of SDN to determine, whether SDN is the right technology for that use case.

Programmability

Programmability is not only a principle but also the key feature of SDN and drives most SDN use cases. This opens the control plane to innovation using conventional software development methods, in turn enabling the customization of the network according to a specific setup or scenario.

Example: Based on one or more external information resources (e.g. cloud orchestration) the routing in a network is adapted automatically to optimize the resource utilization. Google uses such a mechanism to optimize the bandwidth usage on links between the company's data centers. It achieves this by leveraging information from the traffic sources and grouping application traffic into flow groups with different priorities [12].

Protocol Independence

Protocol independence enables SDN to control or run in conjunction with a large variety of networking technologies and protocols on different network layers. This feature enables migration strategies from old to new technologies and supports the possibility to even run a different network protocol stack tailored for each application.

Example: In order to enable the migration from IPv4 to IPv6 a network operator decides to run both versions of IP in parallel. This is usually done using tunnels and encapsulation. The authors of [20] propose to use SDN-enabled forwarding elements with a centralized control plane to dynamically set up the tunnels at the end points.

Ability to Dynamically Modify Network Parameters

The ability to actively modify network parameters in a dynamic manner that is close to real time defines this SDN feature. Dynamic re-configuration is feasible in different time-scales. This covers wide area networks where only a few change operations are required per day, to data center networks where the constant instantiation or migration of virtual machines and their network connectivity has to happen in minutes or even seconds.

Example: In case of an overloaded link between two network elements with multiple routes, priority traffic is identified through application information and rerouted with minimal delay [3]. In this case the SDN controller receives information about individual flows from the application control plane to determine whether a certain application actually needs more resources and allocates a higher priority to its flows.

Granularity

Networking spans different protocol layers and also levels of data flow aggregates. SDN allows to control traffic flows with a different granularity on both the aggregate level and the protocol layers. This can range from large MPLS tunnels in core networks to a single TCP connection in a home LAN. This is a necessary feature to ensure scalability and enable the control plane to work on different levels.

Example: In [3] the SDN controller operates on the granularity of individual flows to optimize the user experience for the user of one particular session. This high granularity is feasible in networks with a low total number of flows, e.g.

home or access networks. In [19] SDN is used to interconnect a virtualized access network to a legacy MPLS core operating on tunnels only. Each tunnel can contain a multitude of flows.

Elasticity

The elasticity feature of SDN describes the ability of the SDN network control plane to increase and decrease its resource consumption based on the required capacity. As controllers run in software, they can be flexibly instantiated and synchronized using a distributed or hierarchical approach on multiple physical or virtual hosts. This enables the control plane to react to variations in traffic mix and volume.

Example: Due to a temporarily increased amount of control traffic in a data center network, the SDN controller can no longer be hosted by a single physical device and has to be distributed among several machines. However, when the situation resolves itself, the control plane can again be relocated to its original host in order to conserve resources. There are several approaches to achieve this kind of distributed SDN control plane. The Onix [21] realization is based on synchronizing the network information base, i.e. the global state of the network, across a cluster of servers. Each server directly manages a subset of network elements and exchanges information with the rest of the network controller instances via the shared network state.

1.1.4 Use-Cases for Software Defined Networking

This section introduces several use cases that we have selected to derive and to illustrate a method for classifying SDN in terms of the above features and interfaces.

Cloud Orchestration

Over the last decade, cloud services have developed at a rapid pace. However, the innovation in this field was mainly confined to server and data center technologies as well as distributed applications. This has led to networks becoming a hindrance for cloud operations. A major reason for this is the fact that networks and servers were traditionally managed separately. For cloud applications to be provisioned and operated quickly and in an automated manner, the management of both network and cloud framework needs to be integrated. SDN is a viable way to achieve this integration, as the SDN controller as well as the cloud orchestration framework is software and a (standardized) interface between both worlds is therefore easily attainable. This interface can then, for example, be used to notify the network controller of an imminent virtual machine migration or to notify the cloud orchestration that a link is overloaded and the server load should be moved to a different location. In [9], the benefits of such an interface are shown. The cloud orchestration software OpenNebula is used to orchestrate virtual servers across multiple hosts and show that a short advance notification from the cloud orchestration to the SDN controller before a virtual machine migration was sufficient to maintain the user sessions of a video streaming service during the migration.

Load Balancing

Another service required for the successful operation of online services that are hosted in data centers is load balancing. Online services, e.g., search engines and web portals, are often replicated on multiple hosts in a data center for efficiency and availability reasons. Here, a load balancer dispatches client requests to a selected service replica based on certain metrics such as server load. In general, a load balancer is typically a separately deployed function in a network that distributes the load among network and data center elements in its scope according to a certain optimization metric such as minimum average load or link cost. Today's solutions for load balancers are effective but have limited flexibility in terms of customization. Being a proprietary middlebox function, such solutions also come at a high cost. When using SDN technologies, the load balancing can be integrated within any forwarding element in the network, e.g., OpenFlow switch, avoiding the need for separate devices. Furthermore, SDN allows load balancing to operate on any flow granularity. In [9], a use case for a data center load balancer is described and a solution based on OpenFlow is proposed. Instead of using a traditional middlebox solution the functionality is realized at the OpenFlow controller and enforced by setting aggregate flow rules using wildcards in the network elements. This way the need for a dedicated balancer device is no longer existent. Current research tries to provide an abstract language which allows programmers to directly control the network and mechanisms like load balancing [22].

Routing

The API between data plane forwarding and a centralized control plane in SDN provides ample opportunities for routing protocol adaptation, which is very difficult in existing decentralized routing schemes implemented on closed box network elements. Routing services that can be realized by the SDN concept, e.g. through programming modules on OpenFlow controllers directing OpenFlow Switches, include path selection for traffic optimization, multi-homing, secure routing, path protection, and migration between protocol versions, i.e. IPv6. In [23] the authors propose a hybrid SDN/BGP control plane that on the one hand leverages the new possibilities in a simplified centralized routing approach and other hand benefits from the compatibility with legacy networks.

Monitoring and Measurement

SDN provides the network the ability to perform certain network monitoring operations and measurements without any additional equipment or overhead. The concept was introduced in [24] and is based on the fact that an SDN inherently collects information about the network to maintain a global network state at the logically centralized controller. This information can then be processed in software to obtain a subset of monitoring parameters. Furthermore, active measurements are enabled by selectively mirroring specific production traffic flows to the control plane or an external measurement device without the need of introducing artificial and potentially disruptive measurement probe traffic into the network. For example, by mirroring the traffic for a phone call at ingress and egress point of the network, the network administrator can determine the delay and quality of service for a particular call at a certain time.

Network Management

Today's network management policies are usually decided upon by the network operator and then configured once in each network element by an administrator. The larger the network, the higher the required configuration effort becomes. Hence, a once set policy is seldom modified. This leads to an often very inefficient network operation. The fact that traffic patterns continually change cannot be taken into account this way. In order to change this, the network needs to be able to adapt policies dynamically and automatically based on a range of information. This calls for a more general specification of network policies that are subsequently translated into specific rules for each device in the network using a policy engine. The logically centralized control plane of SDN offers itself as a very suitable way to enable such an approach as it has all information about the network available. For example, a high level network policy dictates the prioritization of VoIP traffic inside an Enterprise network. The SDN controller can then identify corresponding network flows and assign them to a high priority level in each device. This is dynamic on the one hand as VoIP flows are set up and terminated with each phone call and on the other hand it is automated as the devices are configured without the need for physical access and any human intervention. In fact the administrator does not have to know the topology of the network or the devices involved in order to achieve the policy's goal. Such an approach has been implemented prototypically in [25].

Application-Awareness

Using network resources efficiently and optimizing traffic flows towards high end-user Quality of Experience (Quality of Experience (QoE)) is an often cited goal for next generation networks. However, it is difficult to realize when nothing is known about the kind of applications, which are run on the network and their state. Existing approaches in this direction often rely on Deep Packet Inspection to identify the applications. This, however, is not a very accurate technique and does not take the application state or QoE into account at all [26]. With the Northbound-API of the SDN controller, the application itself can inform the network about its properties and state. This way, the network controller can direct traffic flows to complement rather than disrupt each other [3, 27]. Furthermore, a once made forwarding decision can be revised in light of changing situations in the network and a different application state. The other way around, if the network can no longer sustain a certain service level for the application due to lack of resources, it can notify the application to modify its behavior. For example, due to its architecture, SDN easily allows cross-layer optimization between applications and their demands and the network capabilities. Thus, a better use of the network resources with respect to more generic constraints like user-centrality [3] or energy-efficiency [9] is possible.

1.2 Scientific Contribution

This section is meant to provide an overview of the scientific contribution of this monograph and the studies it contains.

Figure 1.2 categorizes the referenced studies this monograph consists of according to the applied methodology on the y-axis and the area of research in the field of Software Defined Networks on the x-axis. There are three main areas of study. These are performance evaluation, SDN applications and QoE. Both practical methods like measurements as well as conceptual approaches like analytical models are applied.

The first contribution is the definition of SDN and its interfaces in the previous section [1]. It provides a framework in which to understand the remaining contributions in terms of SDN and it determines the terminology used in this monograph. The second contribution is a general performance model of an SDN system based on OpenFlow [4]. This model is based on real-world measurements and provides an excellent starting point for the investigation of SDN performance. Moving on from there [5] offers details on the performance of SDN control plane, which, as the model has shown, is the new key component essential to the performance of SDN.

Going beyond the performance of the SDN system itself [6] takes a look at how an SDN-based application for monitoring performs in relation to its classical counter parts and shows that SDN applications are indeed a viable alternative.

In [10] and [9] further SDN solutions to problems of classical networking are shown in the fields of test beds as well as data center operations, which are both areas that have spearheaded SDN adoption.

Improving on classical networking, [3] highlights a way to leverage SDN to achieve QoE-based application-aware networking by taking application information into account. However, for this to work, the significance of influence factors on the QoE for a specific application has to be known. Therefore, [2] and [7] show how this knowledge can be obtained for the particularly challenging application of cloud gaming.

1.3 Outline of This Thesis

The remainder of this theses is structured as follows. Chapter 2 investigates the performance of SDN on the example of OpenFlow using measurements, simulation, and analytic modeling to highlight key performance issues of current SDN implementations as well as make general statements about the achievable performance. In Chapter 3, SDN applications that can augment or replace classical networking methods are introduced and evaluated towards their functionality and feasibility of use. We then discuss a possible interplay between Quality of Expe-

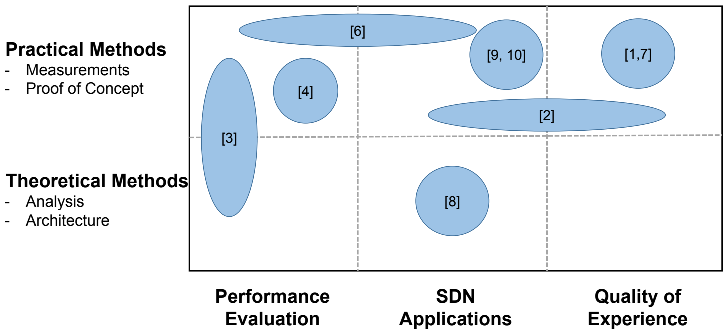


Figure 1.2: *Map of scientific contributions of the author included in this monograph in the field of SDN according to methodology and area of research. The numbers in brackets correspond to references.*

rience and SDN as we introduce an SDN-based QoE-aware networking approach as well as its requirements and ways to meet them in Chapter 4. Finally, this work is summarized and concluded in Chapter 5.

2 Performance Analysis of Software Defined Networking

The concept of Software Defined Networking represents a paradigm shift in networking, which is not yet fully understood. While SDN promises to have various benefits in terms of flexible network operation and management, critics question the performance of an SDN-based network. For SDN to become a viable alternative to conventional networking in production environments, these concerns have to be addressed and a good understanding of the factors that impact SDN performance has to be established. In this chapter, we take a step in that direction by investigating the performance of existing SDN components and create a performance model based on queueing theory.

SDN introduces significant changes not only to the control plane, but also to the data plane. Therefore, in this work we look at each of the planes individually. The only technology available that meets all our criteria for SDN put forth in the previous chapter is OpenFlow [11]. Thus, we have chosen OpenFlow to represent an SDN system in our study.

In the OpenFlow data plane a novel challenge is presented by the way packets are matched against existing rules. Contrary to conventional L2/L3 switching, packet headers are not matched against a MAC or IP address but a set of (arbitrary) packet headers defined for a specific flow. The current generation of OpenFlow switches leverages legacy switching boards to implement the SDN features. We take a look at the performance of these devices in Section 2.2.

While the data plane impacts the forwarding performance, the control plane is responsible for setting up new flows. There are several new impact factors to consider for the performance of an OpenFlow control plane compared to a conventional one. The most significant is the relocation of the control plane to an external device. Instructions sent to the data plane have to be transported via a network themselves. Furthermore, a single control plane instance can be responsible for a multitude of forwarding elements. We investigate the impact of these new elements using a self-developed controller software analysis tool in Section 2.3.

In Section 2.4, we then introduce a performance model for an OpenFlow architecture, which allows us to leverage the results from the measurements to gauge the scalability of the OpenFlow system.

This chapter is mainly based on and taken from [4] and [5].

2.1 Background and Related Work

To better understand the model for the OpenFlow performance evaluation, we first give a brief overview of OpenFlow version 1.0, which was used in our experiments. More details on OpenFlow can be found in the white paper [11] as well as in the OpenFlow specification [28].

The OpenFlow switch itself holds a flow table which stores flow entries consisting of three components. The first is a set of 12 fields with information found in a packet header that is used to match incoming packets. The second is a list of actions that dictates how to handle matched packets. The final component is a collection of statistics for the particular flow, like number of bytes, number of packets, and the time passed since the last match.

When a packet arrives at the OpenFlow switch, its header information is extracted and then matched against the header portion of the flow table entries. If checking against entries in each of the flow tables of the switch does not result in a match, the packet is forwarded to the controller, which determines how the packet should be handled. In the case of a match, the switch applies the appropriate actions to the packet and updates statistics for the flow table entry. This process is visualized in Figure 2.1.

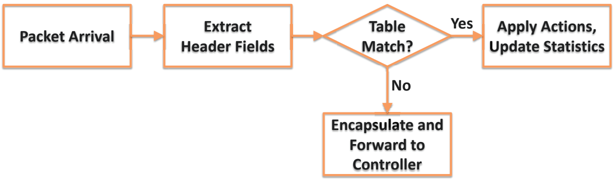

Packet Arrival
Extract Header Fields
Table Match?
Yes
Apply Actions, Update Statistics
No
Encapsulate and Forward to Controller

Figure 2.1: *Handling of incoming packets in an OpenFlow switch.*

Several papers have been published indicating possible uses for OpenFlow [29–31]. All these papers demonstrate that the concept of splitting the control plane from the data plane is useful in a variety of fields, like data center routing, energy saving, and network virtualization. However, none of these papers addresses performance issues of the OpenFlow concept.

2.1.1 Works on Data Plane Performance

Based on [32], Bianco et al. [33] give a basic performance analysis for an OpenFlow software switch. They compare OpenFlow switching, layer-2 Ethernet switching, and layer-3 IP routing performance on Linux-based PCs. As performance indicators they use forwarding throughput and packet latency. We extend these measurements by also considering various hardware-based solutions. Tanyingyong et al. [34] propose an architectural design to improve the look up performance of OpenFlow switching in Linux using a standard commodity network interface card. They show packet switching throughput increasing up to 25% compared to the throughput of regular software-based OpenFlow switching. Luo et al. [35] apply network processor based acceleration cards to perform OpenFlow switching. They show a 20% reduction on packet delay compared to conventional designs. We compare our results to their findings.

2.1.2 Works on Control Plane Performance

Current deployments of OpenFlow mostly rely on conventional switches as forwarding units. As these are usually not designed to function as flow switches, they are often performance bottlenecks and thus the research focus in terms of performance so far lay on the data plane. However, gradually the control plane performance is shifting into focus.

Curtis et al. [36] propose changes to the OpenFlow protocol as they discovered inherent performance bottlenecks with regard to high CPU load caused by control plane interaction in current OpenFlow switch implementations.

In [8], Pries et al. evaluate the usability of OpenFlow in data centers. They discovered that setting flow rules reactively already leads to an unacceptable performance when only eight switches are handled by a single controller.

With OFlops [37] a framework for performance analysis of OpenFlow switches exists. However, on the controller side only a relatively simple benchmark exists with Cbench [38]. Cbench was first developed by Robert Sherwood and has since become the standard evaluation tool for controller performance. In [39] Tootoonchian et al. use Cbench to highlight possible controller performance improvements. Still, little can be derived from the results in terms of controller behavior. Cbench is single-threaded, i.e. multiple instances have to be started to utilize multiple CPUs. It also only uses one controller connection for all emulated switches. Aggregated statistics are gathered for all switches but not for each switch individually. As a result, it is for example not possible to tell whether all controller responses are for a single switch and the others receive nothing, or whether the controller capacity is shared fairly among the switches. Our benchmark addresses these issues to obtain more granular results.

2.2 OpenFlow System Measurement

The initial implementations of OpenFlow network elements rely either on software or generic experimental hardware boards like the NetFPGA [40]. However, the technology also has potential apart from the research world, e.g., for flexible network management in data centers [41]. This has lead to an increased interest in the technology from internet service providers and subsequently hardware vendors to provide commercial grade products, which are OpenFlow-enabled. This in turn raises the question how these new OpenFlow products actually perform in general and in comparison to their experimental, but cheaper predecessors. In this section, we aim to provide a basic comparison between various OpenFlow implementations from pure software solutions to commercial switches as a performance indicator for the data plane.

2.2.1 Data Plane Performance Experimental Setup

This section outlines our experimental setup. We illustrate the composition of our testbed and the involved hardware. Furthermore, we give a brief introduction to the tested OpenFlow switches and describe the performed tests.

Testbed

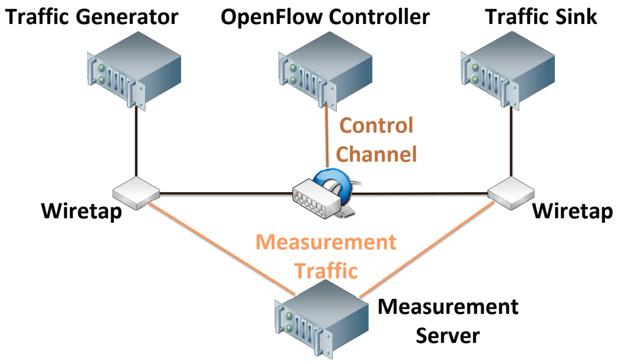


Figure 2.2: *Data plane performance measurement setup*

Figure 2.2 shows the logical view of our test bed. To operate the 1 Gbps link between sender an receiver at full capacity, we use the Raw Packet Generator pktgen. The generator is part of all current Linux kernels. In our case, a Debian 6.0 "Squeeze" is used. We run the generator on an HP Proliant DL320 using an Intel Xeon quad core processor clocked at 2.13 GHz and 10 GB of RAM. As it might be necessary at one point to use multiple interfaces or machines for traffic generation, we included an HP ProCurve 1810G-24 as aggregation switch, which bundles all incoming streams to a single outgoing interface leading to the OpenFlow switch to be tested. Figure 2.2 exemplary shows an OpenFlow switch. The switch itself is directly connected to a server, which acts as the OpenFlow controller. In our case this server is a Fujitsu Siemens Esprimo PC powered by a Pentium 4 HT clocked at 3.4 GHz and using 2 GB of RAM. The OpenFlow controller software running on top of the server is NOX 0.8 for all tests. The OpenFlow version used is 1.0. Measuring throughput and forwarding delay of the OpenFlow switch requires the test traffic to be captured once just before entering the switch and once just after leaving it. For this purpose, we use two NetOptics 1000Mbps wire taps. These taps mirror the traffic and redirect it to a capture machine. This machine is identical to the traffic generator, an HP Proliant DL320 also using an Intel Xeon quad core processor clocked at 2.13 GHz and 10 GB of RAM. However, in order to be able to capture the traffic from both wire taps and at the same time provide accurate time stamps at very fast switching times, an additional special Endace DAG 7.5 G2 capture card is needed. This card is able to capture 2x1Gbps of traffic at a time resolution of 10 micro-seconds. The last part of our testbed is the client PC at which the traffic flow terminates.
The following switches were examined in our experiments:

i) **Open vSwitch** Open vSwitch [42] is an open source software implementation of a multilayer virtual switch. OpenFlow functionality is one of its features. The Open vSwitch can either be run in a distributed fashion across several machines, inside a hypervisor or function as a control stack for switching hardware. Open vSwitch is the only software implementation tested in our experiments.

ii) NetFPGA The NetFPGA project aims to provide network researchers with a comparatively cheap programmable hardware solution. The experimental NetFPGA boards feature four 1 Gbps interfaces as well as a Field Programmable Gate Array (Xilinx Virtex-II Pro 50), which allows the implementation of basic network functions. There is a variety of sub-projects implementing network features based on this card. Among these projects is also an OpenFlow switch implementation [40] by the Stanford University. It allows the Stanford OpenFlow implementation to use the NetFPGA memory as flow tables.

iii) Pronto 3290 The Pronto 3290 is an experimental OpenFlow-only switch based on a generic Broadcom board. At the time of the test, the OpenFlow firmware was written and maintained by the University of Stanford under the name "Indigo". The switch features 48 network ports with a speed of 1 Gbps each.

Test Design

In this subsection, we briefly describe the tests, which were performed for each individual switch. All tests have been run at least six times for payload sizes of 18, 60, 200, 600, 1000, 1400, and 1472 bytes. In each run two million packets were transmitted.

a) One Fully Specified Rule: In this test, one fully specified flow rule is inserted into the switch prior to any traffic being sent. This rule matches the 1 Gbps flow of traffic sent by the traffic generator. The idea here is to obtain a baseline for the forwarding delay that can be expected from each switch.

b) Maximum Number of Fully Specified Rules: In this test, also one fully specified rule matches the generated traffic. However, here the flow table in the switch is filled with non-matching flow rules. This test is supposed to gauge the influence of the look-up time in the flow table on the overall forwarding delay of the switch.

c) **One Wildcard Rule:**Here, one rule matching the generated traffic containing at least one wildcard is inserted into the flow table. This test is designed to determine whether the type of rule (wildcard vs fully specified) has an influence on forwarding delay. Depending on the implementation, different types of rules could be handled differently, e.g. in separate hardware tables.

d) **Maximum Number of Wildcard Rules:**Once again one rule matching the generated traffic containing at least one wildcard is inserted into the flow table. However, the maximum number of wildcard rules, which do not match the flow, is inserted into the flow table. Identical to the fully specified test, this test is designed to gauge the influence of look-up delay for this type of rule.

e) **Disruptive Traffic:**This test is identical to test **a)**. However, in addition to the measurement probe, a new flow arrives at the switch every millisecond. This flow spawns the creation of a new flow rule by the controller, which is then written into the flow table of the switch. The aim of this test is to determine whether this write operation does have any influence on the look-up performance (i.e. forwarding delay) of the switch.

f) **Forward to Controller:**In this test, a rule is inserted into the flow table, which causes all incoming packets to be sent to the controller. The purpose of this test is two-fold. First, the forwarding delay is measured and second, the amount of packets a controller can handle is determined.

2.2.2 Data Plane Measurement Results

In this section, we present and describe the results of the performance tests for different OpenFlow switches.

a) One Fully Specified Rule

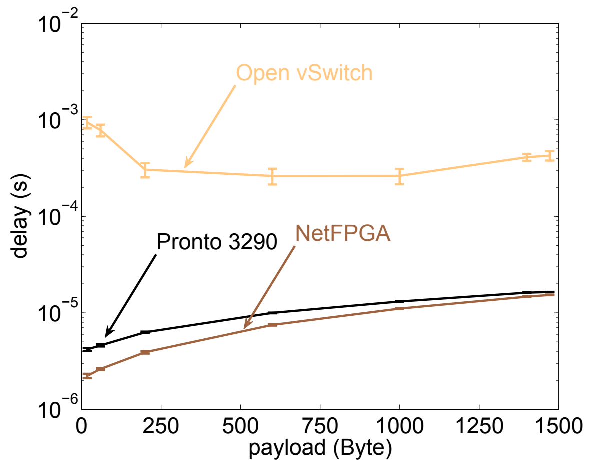


Figure 2.3: *Delay measurements with one fully specified rule*

Figure 2.3 shows the forwarding delay in relation to packet size for each of the tested switches with only one single fully specified rule, which matches the measurement probe, present in the flow table of the switch. The y-axis has a logarithmic scale to visualize all results in one graphic. A gap between the forwarding performance of the two investigated hardware solutions and the Open vSwitch is clearly visible. The software implementation is always at least one order of magnitude slower imposing a delay of between 0.2 and 1 ms on the packets. In contrast to the hardware switches, the Open vSwitch shows its highest forwarding delay when confronted with minimum sized packets then becomes faster up to 250 Byte payload. Eventually the delay increases again with the higher packet sizes up to the MTU. It also has to be noted here that the software implementation drops one third of all packets on average, while the hardware solutions lose next to nothing for the applied traffic of 1 Gbps.

The Pronto 3290 shows a delay of about 0.004 ms for minimum size packets. At MTU has increased to about 0.01 ms on average. The NetFPGA is slightly faster for minimum size packets with about 0.002 ms, but converges towards the Pronto graph for payloads near MTU size.

b) Maximum Number of Fully Specified Rules

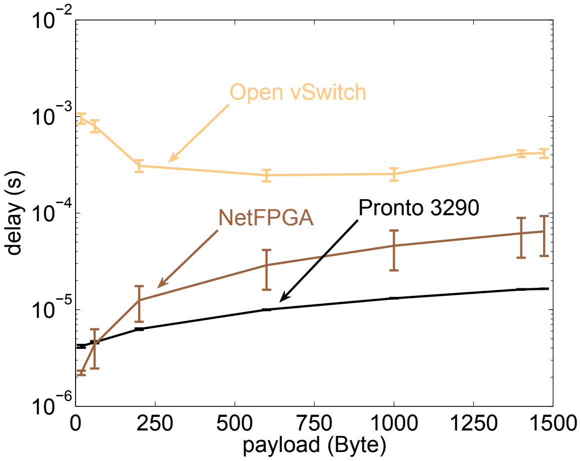


Figure 2.4: *Delay measurements with the maximum number of fully specified rules*

In Figure 2.4 the results for a full flow table with only one matching rule are given. For the Open vSwitch and the Pronto 3290 the results are identical to those shown in Figure 2.3 with only one rule in the flow table. This means that the look-up operation in the flow table was efficiently implemented for these switches and scales with the number of rules. Therefore, it does not impact the overall performance. By contrast the NetFPGA OpenFlow Switch performance is impacted by an increasing number of rules in the flow table. While it performed best for all packet sizes in 2.2.2, the forwarding delay increases steeply from about 0.002 ms at the minimum packet size to about 0.015 ms for 250 Byte packets. For larger packet sizes the NetFPGA constantly performs about 0.03 ms slower than the Pronto. This result is due to the fact that the NetFPGA OpenFlow switch is an experimental implementation with only space for 24 rules in the TCAM compared to a dedicated implementation with about 1300 rules for the Pronto.

c) One Wild Card Rule

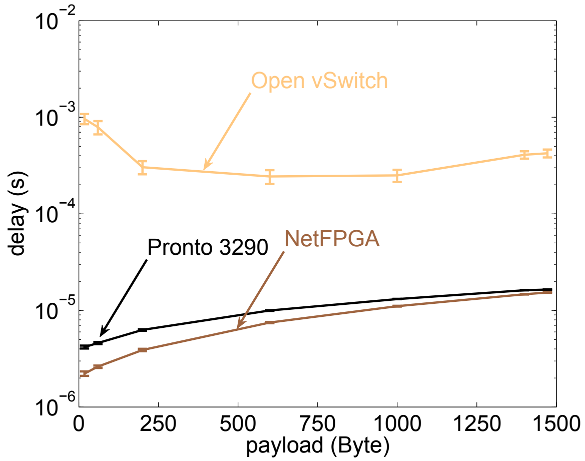


Figure 2.5: *Delay measurements with one wildcard rule*

Figure 2.5 shows the results for one matching wild card rule in the flow table. The measured delays do not differ from those in Figure 2.3 with one fully specified rule. This suggests, that rules with wild cards are not matched faster or slower than fully specified ones in all of the switches.

d) Maximum Number of Wild Card Rules

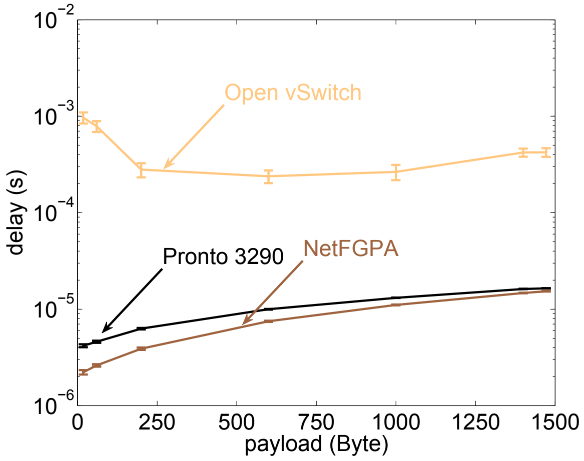


Figure 2.6: *Delay measurements with the maximum number of wild card rules*

The results for a flow table filled with wild card rules and only one matching are shown in Figure 2.6. The results barely differ from those for one wild card rule in Figure 2.5. This suggests that the flow table look-up has also been implemented to scale for wild card rules. However, this means that the NetFPGA implementation can match wild card rules more efficiently than fully specified rules. The results for a table filled with wild cards are identical to those with only one rule. However, we noted a difference for fully specified rules in the NetF-

PGA, cf. Figure 2.3 and Figure 2.4. This suggests that the look-up operation in the NetFPGA for wild card rules is more efficient than that for fully specified ones.

e) Forward to Controller

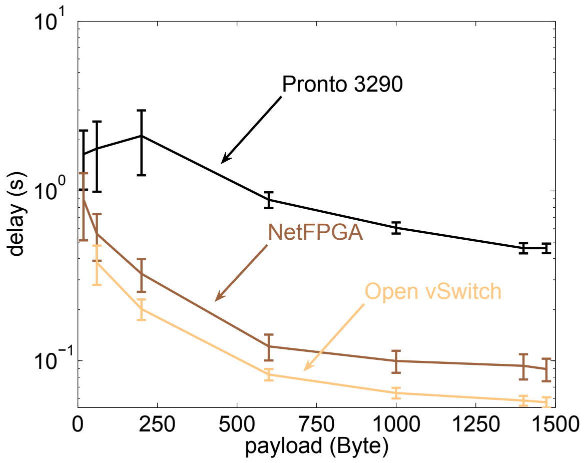


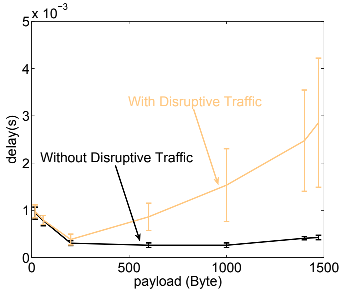
Figure 2.7: *Delay measurements with packets forwarded to the controller*

Figure 2.7 depicts the results for the forwarding delay, when all packets are forwarded to the controller. All tests are subject to massive packet loss of between 95% and 99% as the controller can not handle the necessary amount of data. The packets that are forwarded experience a delay at least one order of magnitude longer than that of the Open vSwitch in the previous tests, which is also a software implementation. All switches show similar behaviors. The forwarding performance is worst for minimum size packets at about 1 s for the NetFPGA and 1.8 s for the Pronto 3290. The Open vSwitch lost too many packets here as to establish a stable mean value. The forwarding delay decreases for all switches with larger packets with the Open vSwitch at about 5 ms, the NetFPGA at about 10 ms, and the Pronto 3290 being the slowest at about 35 ms for MTU size packets.

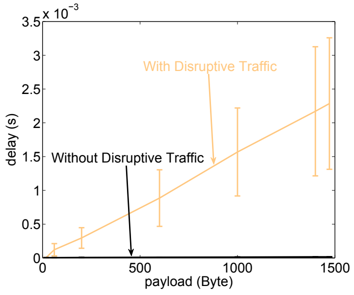
f) Disruptive Traffic

The following figures illustrate the influence on the forwarding delay caused by writing operations on the flow table created through disruptive traffic. This test is performed for the Open vSwitch and the Pronto 3290. In each case just one fully specified rule is located in the flow table and matches. The same test is repeated and disruptive traffic is introduced.

Figure 2.8a shows the results for the Open vSwitch. For small packets up to about 250 byte the introduction of disruptive traffic does not impact the forwarding delay as the switch is already under heavy load. However, for larger packets the influence of the write operations on the flow table becomes apparent. The larger the packet, the higher the mean forwarding time and also the variance is steadily increasing. Starting at values of 0.3 ms delay, the mean forwarding delay increases up to about 2.8 ms, while remaining almost constant without disruptive traffic.



(a) Open vSwitch



(b) Pronto 3290

Figure 2.8: *Delay measurements with disruptive traffic present*

The Pronto 3290 shows the greatest susceptibility to disruptive traffic as is shown in Figure 2.8b. Without disruptive traffic, the forwarding delay lies constantly below $10\mu s$ as can be seen in Figure 2.4. However, with disruptive traffic this delay increases steadily from the μs-range for minimum-sized packets to about 2ms for MTU-sized packets. Here, the variance is also greatly increased.

2.3 OpenFlow Controller Benchmark

OpenFlow controllers are the key components in the OpenFlow architecture. They are often marketed as ”networking operating system” in a software defined network. However, this designation is slightly misleading. While the OpenFlow controller certainly fills the role of an operating system, bridging the gap between physical hardware and applications, many controllers lack the stability and performance we would expect a modern operating system to have in the computing domain.

OpenFlow controllers can not be configured, but have to be programmed, which makes them more akin to operational frameworks than an actual operating system. Since the OpenFlow standard does not dictate how a controller should be implemented or even which elements it should possess beyond the OpenFlow secure channel, a variety of different implementations has been developed, each with its own behavior and performance characteristics. These differences make specific controllers better suited for certain scenarios than others. To choose an implementation over another and to analyze the system behavior for a particular deployment, these differences have to be understood.

In this section we introduce a flexible OpenFlow controller benchmark as a tool to obtain this insight. Unlike conventional benchmarks that focus on overall throughput and latency, our benchmark allows the emulation of scenarios and topologies and can evaluate the controller performance on a per-switch basis. This way, a more detailed analysis of controller performance bottlenecks as well as obscure behavior is possible. The purpose of this is to underline the necessity of tools like this for analyzing and understanding controller and, as a consequence, OpenFlow network performance.

2.3.1 Benchmark Architecture

In this section we present our OFCBenchmark tool. First, we explain the design goals that guided our development process. Then, we present the architecture of the software and describe the implementation.

Design Goals

The architecture and the implementation of OFCBenchmark are guided by the following main design goals.

- *Scalability:* The software should be designed in a way so that multiple instances can run in a coordinated way on different CPU cores, CPUs, and hosts. This achieves that the load generated to test an OpenFlow controller is not limited by a single core of the CPU or the memory of the machine that runs the software.

- *Ability to provide detailed performance statistics:* Our benchmarking software should provide performance metrics such as round trip times, sent or received packets per second, or the number of outstanding packets, in time series and on a per switch basis. This feature permits to investigate whether an OpenFlow controller treats switches differently or changes its behavior over time.

- *Modularity:* The controller development process progresses rapidly and therefore, the benchmark should be adaptable to new scenarios, which is easier in a modularized software. In addition, measurements of further performance metrics and further parameters to control the load generation should be easy to add to the benchmarking software.

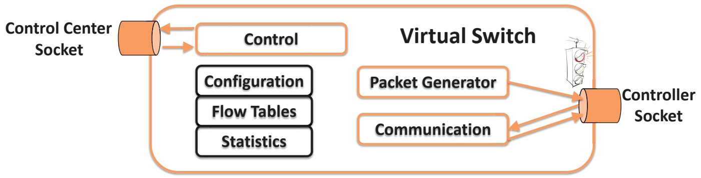


Figure 2.9: *Structure of a virtual switch.*

Architecture Design

The architecture of our OpenFlow controller benchmark consists of three main components - the OFCBenchmark Control Center (OCC), the OFCBenchmark Client (OC), and the Virtual Switch (VS). The OFCBenchmark uses a distributed approach, i.e. the benchmark can be spread over multiple hosts. Each of these hosts runs an instance of the OC. The OC itself is already a full benchmarking system. It executes the performance tests using the configured number of VS objects. However, it is limited in scale as the number of VSs is restricted by the amount of memory and computing capacity of a single host. In the distributed mode, multiple OCs connect to the OCC, which then controls the experiment.

The key component of the OFCBenchmark is the virtual switch. Figure 2.9 shows a schematic illustration of the VS structure. The virtual switch holds a simplified flow table to be able to respond to controller requests. It also has a statistics store where the benchmarking results are kept and updated. Furthermore, each virtual switch has two socket connections and three threads encapsulated in the virtual switch object. The connections serve as communication channels to the OCC and the OpenFlow Controller and are managed using the threads. This allows us to treat a VS as a true individual entity, which gives us the desired modularity, scalability, and the ability to provide detailed performance statistics.

Implementation

The OC and its VSs are written in C++ using the Boost library for thread-handling. Experiments can either be configured directly via the OCC communications channel in distributed mode or through a configuration file in standalone mode. Configuration options include number of switches, per switch packet-inter-transmission times, packet sizes, as well as the option to specify a pcap file containing OpenFlow messages for each switch to play-out. Furthermore, the OC allows the creation of a switch topology specified in a separate configuration file. This is achieved by allowing the virtual switches to seemingly forward controller generated LLDP or OFLDP packets and sending them back to the controller as Packet-In messages according to the configured topology.

At creation time the VS reads the same configuration file as the OC and connects itself to the OCC waiting for further instructions or requests. In Figure 2.9 we can see the control thread connected with the OCC through a socket. This thread executes the commands of the OCC in the VS. If the connection to the OCC is not configured in the configuration file, the switches are operated in standalone mode by the OC. The two remaining threads are using the same TCP socket and are the workers of the virtual switch.

The choice of TCP as transmission protocol reflects the OpenFlow specification. The communication-thread is responsible for handling the communication with the currently benchmarked controller using the OpenFlow protocol in version 1.0. It performs the OpenFlow handshake process and answers other controller requests. The packet-generator thread creates and sends Packet-In messages to the controller for benchmarking. The time between two sent packets can be configured. By default the time is set to zero, which results in as many packets being sent as the TCP stream allows. The Packet-In messages contain the packet header of the first packet of a new IP flow the controller has not yet encountered. Every Packet-In message is identifiable through its buffer-id. The controller responds to those packets with Packet-Out and/or FlowMod messages using the same buffer-id to identify the corresponding packet. Receiving the response, the communication thread parses the id, calculates the round trip time for this request and updates the statistics. As both threads are using the same socket, a semaphore was included to coordinate the output and avoid data corruption. This is a mutual exclusion that prevents the usage of socket output from different threads at the same time. Before sending a datagram the ”user” of the socket output has to lock the semaphore and release it after transmission. To be able to use multiple CPUs and thus to keep accurate statistics for the round trip time, the virtual switches use blocking I/O, i.e. a thread is only ”woken” by the operating system once a packet arrives. This way we avoid having to check frequently whether a packet has arrived using a single thread.

The OCC is a graphical user interface written in Delphi, so experimenters can see the current configuration at a glance and modify the test settings according to their requirements.

2.3.2 Comparison with Cbench

To verify the results of our benchmark we run a comparative test with Cbench. The test was run on a testbed consisting of two PCs directly connected through a 100 Mbps Link. Both systems share the same hardware and software configuration with a Pentium IV 3.4GHz CPU, 1 GB RAM, and Ubuntu 10.04 as operating system. One PC runs the controller – in this case Nox Classic [43] as learning switch – and the other PC runs the benchmarking tools. The benchmark is set not to introduce artificial delay between packet departures. We compare our benchmark to two versions of Cbench – the current repository version and the original version used for reference.

Figure 2.10 shows the achieved throughput in packets per second with respect to the number of connected virtual switches. The error-bars attached to the graphs give the 95% confidence intervals, which were obtained through five repetitions of each test.

All curves increase from about 10,000 pps with just one virtual switch until they reach a stable level of saturation with about 15 connected switches. The legacy Cbench version reaches its saturation at about 14,300 pps. The current Cbench as well as our OFCBenchmark achieve a higher throughput and reach their saturation at about 16,900 pps. From about 50 connected switches we see a slight decrease to about 16,000 pps in throughput and larger confidence intervals for our benchmark. This is likely due to the larger overhead caused by managing and keeping statistics for each virtual switch independently. However, the difference to Cbench is still quite small and partially still within the confidence intervals. Therefore, we can assume that our benchmark produces comparable results to the Cbench tool.

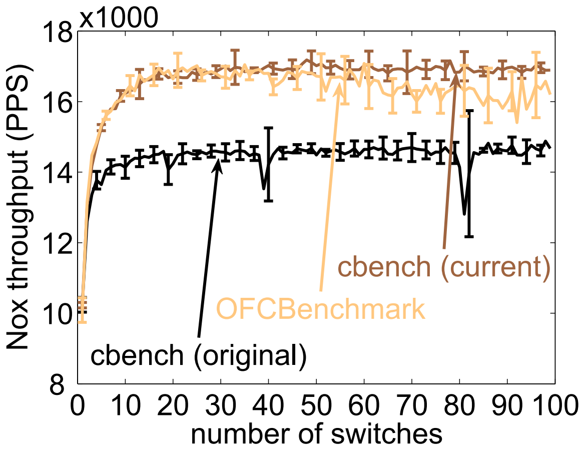


Figure 2.10: *Comparison of our benchmark with Cbench using the Nox-Classic controller.*

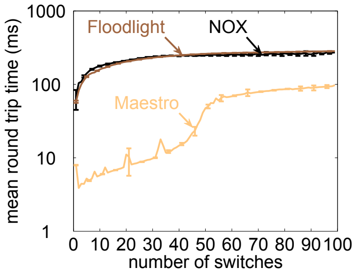
2.3.3 Controller Benchmarking Results

In this section we discuss some initial results we have obtained with our benchmark. These results are produced with a software in development. This is neither an exhaustive and/or representative comparison between the benchmarked controllers, nor can or should a general statement about the quality of the controllers be derived from this simple scenario. This is intended to showcase the features of our benchmark and a discussion of the results and their consequences. The testbed used here is identical to the one described in the previous section. The controllers measured are Nox Classic [43] (”Zaku” release), Floodlight [44] (version 0.82), and Maestro [45] (version 0.2). These controllers were chosen arbitrarily with the only requirement being that they are freely available. All controllers were set to use their respective learning switch applications. As Nox Classic has no multi-threading, the other controllers were also limited to a single thread to obtain comparable results. Nagle’s algorithm was deactivated on all test systems to avoid the influence of artificial TCP buffering. All tests were repeated 5 times to obtain the confidence intervals shown in the figures.

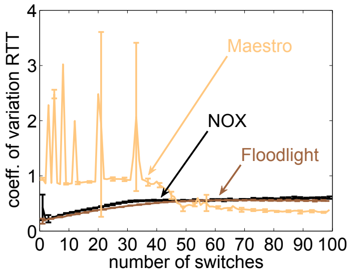
Mean Round Trip Time

The first feature test is for the round trip time (RTT), i.e. the interval from the moment a Packet-In message is dispatched from the virtual switch to the controller until the corresponding Packet-Out or FlowMod message is received by the switch. This test can also be performed with Cbench. Our benchmark extends this feature by allowing to obtain these statistics for each switch individually and as a time series as well.

Figure 2.11a illustrates the mean round trip time in milliseconds for different numbers of simultaneously connected switches to the controller. Note that the y-axis is scaled logarithmically. We observe that the Floodlight and Nox Classic controller behave in a similar way. The response time of these controllers increases rapidly from about 200 ms for one switch until the value stabilizes at about 6 s for 30 switches. Both controllers are obviously under heavy load at this point due to the relatively low performance of the hardware. However, the RTT for Floodlight continues to increase up to about 8 s for 80 switches. Maestro behaves differently. For this scenario Maestro starts at a RTT of only about 6 ms



(a) Mean



(b) Coefficient of variation

Figure 2.11: *Response time for an OpenFlow Packet-In message.*

– two orders of magnitude faster than the other controllers. With an increasing number of switches the RTT increases steadily, but far slower compared to the others, until for 100 switches a RTT of just below 1 s is reached. The largest increase in RTT can be observed between 40 and 50 connected switches. We suspect this behavior is the result of a different processing strategy for Maestro that is advantageous in this scenario as we will see indicated by the results shown in Figure 2.12.

However, the average RTT of all switches and over the whole experiment duration is neither sufficient to judge whether this value changed over time nor whether some switches experienced larger or smaller RTTs, i.e., whether some switches received a preferred treatment. To answer such questions, our tool provides time series of the RTTs on a per switch basis. As an example evaluation of this data, we calculate the average RTT over time for every switch and analyze the variability of this value among the different switches by showing the coefficient of variation (c_V) of these values, cf. Figure 2.11b.

As suggested by the small confidence intervals in Figure 2.11a, the c_V for the RTT is small for all controllers. For Nox Classic and Floodlight we see an increase from about 0.2 with one switch to a stable value of about 0.5 at 30 switches for Nox Classic and 50 switches for Floodlight. Initially Maestro displays a higher c_V at about one with outliers up to 3 and large confidence intervals. This can be explained with the much smaller mean RTT, as small derivations from small mean values have a far larger impact on the c_V than small derivations from larger ones. Mirroring the observations from Figure 2.11b, the c_V decreases between 40 and 50 connected switches to a value of about 0.3 – below that of the other two controllers.

Send- and Response Rates

Apart from determining the throughput and latency of an OpenFlow controller, it might also be interesting or even important to look at the rate it accepts packets. This can provide insights into rate control mechanisms and/or polling strategies of the controllers. Therefore, we have included this feature in our benchmark.

In Figure 2.12 the result for the number of packets per second (pps) sent from the switches to the controller through the OpenFlow secure channel is given. As our benchmark uses TCP to send the Packet-In messages to the controller instead of writing the packets raw on the wire, the send rate is determined by the TCP connection. We observe that the packet send rate for the Floodlight controller does not increase significantly with the number of switches. It starts at a rate of about 10,000 pps for one switch and increases to about 38,000 pps. For Nox Classic the increase is slightly steeper, but stalls at about 70,000 ps for 70 switches. However, for Maestro we see a far higher increase in packet send rate with the number of switches. It increases linearly from about 5000 pps to about 140,000 pps for 35 switches and then continues to increase to 150,000 pps for 50 switches. This suggests the implementation of a rate-control mechanism for Nox Classic and Floodlight, whereas Maestro accepts packets in a best effort manner.

The complement to the send rate is given in Figure 2.13 – the packet reception rate. It describes the number of responses the switches receive from the controller per second. The figure shows that the reception rate for all controllers is similar to

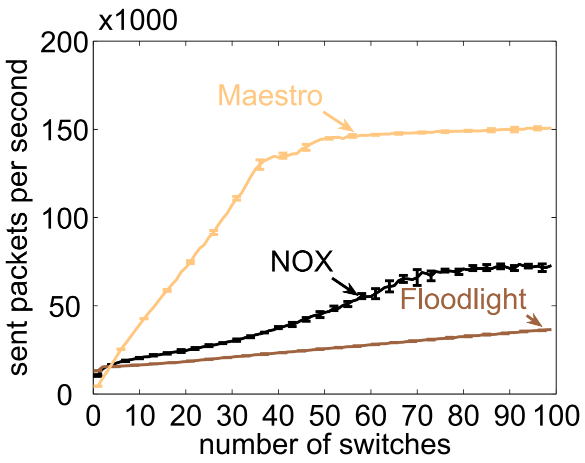


Figure 2.12: *Virtual switch Packet-In send-rate.*

the send rate of the switches shown in the previous figure. However, we do not see an increase in reception rate with an increasing number of switches for Floodlight and Nox as we observed for the rate of sent packets in Figure 2.12. The reception rate is basically stable at about 10,000 pps. The curve for Maestro displays a steep rate increase identical to the corresponding send rate. The initial rate of about 5000 pps for one switch steadily grows up to about 135,000 pps for 35 switches. As before we see the following flatter rate growth up to about 145,000 pps for 50 switches, where it remains stable. This means for Meastro there is a discrepancy of about 5000 pps between send and reception rate at this point. We call this discrepancy the ”outstanding packets”, i.e. the number of unanswered Packet-In messages by the controller. We take a look at these in the following results using the per switch analysis option of our benchmark.

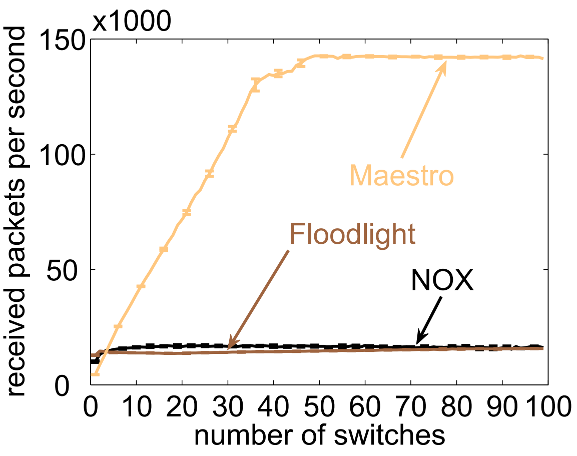


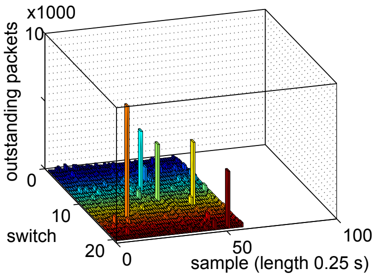
Figure 2.13: *Virtual switch Packet-Out reception-rate.*

Outstanding Packets

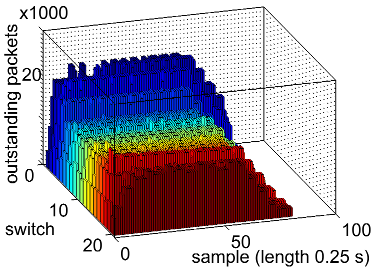
Figure 2.14 shows the evolution of the number of outstanding packets per virtual switch for a test run with 20 connected switches over time for a test run of 15 s. The values are presented for time intervals of 0.25 s and the time axis in the plots shows the interval number instead of the time value.

Figure 2.14a gives the number of outstanding packets for the Maestro controller. As we can see from the comparison of Figures 2.12 and 2.13, the number of outstanding packets is very small for 20 switches. All packets have been processed shortly after the 15 s sending period is over. However, we see some ”spikes” in the graph, i.e. samples with a large number of outstanding packets from about 2000 to 6000 packets.

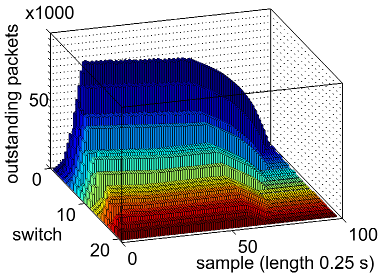
The number of outstanding packets for the Floodlight controller is given in Figure 2.14b. Floodlight shows a different behavior compared to Maestro. All switches have quite a large number of outstanding packets. Switches 1-4 show



(a) Maestro



(b) Floodlight



(c) Nox

Figure 2.14: *Response time for an OpenFlow Packet-In message.*

a particularly high number of 15,000-20,000 outstanding packets and maintain this level over the course of the experiment. The remaining switches hold a level of about 10,000 outstanding packets. This is interesting for two reasons. For one the level of outstanding packets remains constant, i.e. there is no overload in the system. This suggests the presence of a packet buffer in the Floodlight controller. Second, while most switches are treated equally, the first four switches seem to have a larger number of buffered packets. Prior to the experiment, the virtual switches are connected sequentially to the controller. Therefore, it appears that the order in which the switches are connected has an influence on the buffer size. After the end of the sending period, the buffer is gradually processed as can be seen from the decrease in outstanding packets after 60 samples. It takes an additional 5 s after the end of the 15 s experiment until all packets have been answered.

With Nox Classic the influence of the connection order on the number of outstanding packets per switch is even more significant, cf. Figure 2.14c. Apparently, Nox Classic does not treat switches equally. The first connected switch experiences a build-up of up to 100,000 outstanding packet. For each subsequent switch the average number of outstanding packets is slightly reduced. The 20th and last connected switch only experiences 10,000-15,000 outstanding packets on average. As a result, all packets of the later connected switches have finished processing only 1-2 s after the sending period ends, whereas for the first connected switch the processing takes an additional 10 s to complete. In a real network a behavior like this would lead to unfairness. Devices attached to one of the first switches would experience far larger flow setup times. Using only aggregates and mean values, we would not be able to determine the cause of the issue. While the obtained results may not be 100% accurate as the software is still in development, the fact that the results are repeatable and differ between controllers on the same test systems highlights that there are indeed notable differences between controller behaviors. This circumstance and its consequences should be investigated and this is what our approach is aimed at.

2.4 Analytical Modeling of OpenFlow

Having established a baseline for the performance of the OpenFlow system components, we are interested to determine how the performance of such a system scales with changing parameters.

Understanding the performance and limitations of the basic OpenFlow concept is a prerequisite for using it for experiments with new protocols and mechanisms. We aim to provide a performance model of an OpenFlow system. The model is based on results from queuing theory and is verified by simulations and measurement experiments with a real OpenFlow switch and controller. The advantage of this analytical model over the simulation is the fact that it can provide results in a few seconds' time whereas the simulation may require several hours to complete depending on the computing hardware. Additionally, the M/M/1-S feedback queue is already a good approximation of the actual controller performance. The model captures the delay experienced by packets that have to be processed by the controller in contrast to be processed just by the switch, as well as the probability to drop packets if the controller is under high load. Using this model, we derive conclusions about the impact of the performance of the OpenFlow controller in different realistic scenarios, and its effect on the traffic flowing through the OpenFlow-enabled switch.

2.4.1 Model Input Parameters

Based on the results from the previous section we decided to use the performance values of the Pronto 3290 as input parameters for our model as the difference between Pronto and NetFPGA is marginal and the technical specifications of the Pronto are much closer to those of commercial hardware switches than those of the NetFPGA.

The measurements which provided the data for the response times of the OpenFlow controller Nox 0.9 were performed in a different scenario. Rather than our own tool, we used Cbench [38] for the controller measurements as we were not interested in the controller behavior per se, but only in the raw throughput of the controller. The Cbench [38] tool was installed on the measurement server and the controller attached directly. The Cbench tool measures the rate with which flow requests are handled by the controller. Unfortunately, we were not able to match requests and responses to the OpenFlow controller. Hence, we need to rely on the number of answers per second the Cbench tool measured. These showed a mean value of 4175 responses per second with a standard deviation of 101.43. From this value, we calculated the mean values of the controllers sojourn times and use these in our analytical model and in the simulation.

Finally, we also require the inter-arrival times of new packets and flows. These are based on the measurements published in [46], where we analyzed the traffic of a residential wireless Internet access for over 30 days. The packet size distribution and the probability of new flows used in our performance evaluation have been extracted from this study.

2.4.2 A Simplified Model of an OpenFlow Architecture

We abstract the OpenFlow architecture as a feedback-oriented queuing system model, divided into a forward queuing system of the type $M/GI/1$ and a feedback queuing system of the delay-loss type $M/GI/1-S$. We deliberately start by assuming Markov servers for both systems, i.e. an $M/M/1$ for the forward model and an $M/M/1-S$ for the feedback model, to test the robustness of the modeling approach. The forward queue has an average service time of 9.8 microseconds. The queue size of the forward system is assumed to be infinite. In contrast, the buffer for the packets waiting on a controller response is assumed to have a finite capacity of 512, which models the queue of the feedback system. The arrival process at the switch, i.e. of the forward system, is a combination of the arrival process of packets received from the line cards with rate λ and of packets being forwarded from the switch buffer after the controller has determined the appropriate action and the corresponding entry in the flow table was created.

The OpenFlow controller is thus modeled by the feedback $M/M/1-S$

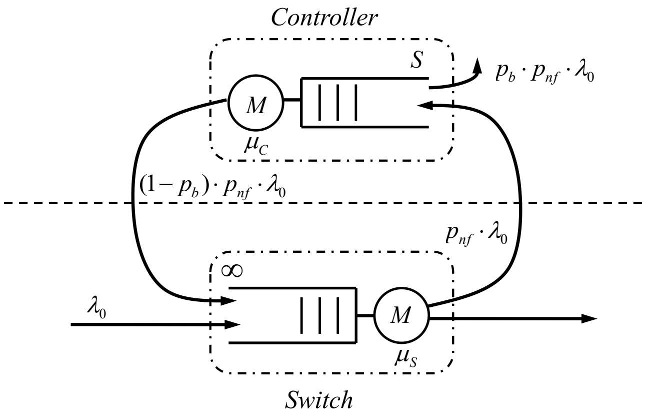

Controller
S
$p_b \cdot p_{nf} \cdot \lambda_0$
M
μ_C
$(1-p_b) \cdot p_{nf} \cdot \lambda_0$
$p_{nf} \cdot \lambda_0$
∞
λ_0
M
μ_S
Switch

Figure 2.15: *A simple model of an OpenFlow switch.*

queuing system with an exponential service time with a mean value $E[B_C] \in \{31\mu s, 240\mu s, 5.2ms\}$. The high as well as the low mean service time were chosen arbitrarily one order of magnitude larger respectively smaller than the measured mean service time of $240\mu s$. We assume that all merging traffic flows again form Poisson streams. The smallest value for $E[B_C]$ is taken from our controller benchmark, the other values have been chosen arbitrarily to reflect the impact of controller applications one or two order of magnitudes slower. The queue length S of the controller system is limited in order to model the possibility of dropped packets under high load conditions. The arrival rate in this system is a fraction of the arrival rate of packets from line cards in the switch, governed by the probability p_{nf} of a flow being seen by the switch which has no flow table entry yet.

The two main performance indicators of interest for an evaluation are the total sojourn time of a packet through the system and the probability of dropping a packet. A packet has to traverse the switch system at least once. With a probability of p_{nf}, the switch has no entry in its flow table for that packet and forwards it to the controller. A packet can be blocked at the controller with probability p_b. After the controller sojourn time, it is again queued in the switch and traverses it for a second time. The complete model of the forward and feedback queuing systems with both components and all traffic flows is shown in Figure 2.15.

It is important to note that a single packet cannot be forwarded to the controller twice, i.e., p_{nf} is only applied to the initial packet flow with rate λ_0. In our analysis, we ensure that a packet does not experience more than the sojourn time of the controller plus twice the sojourn time of the switch. Figure 2.16 illustrates the way a packet takes through the system in a more sequential manner. W_S and W_C r tively,
where

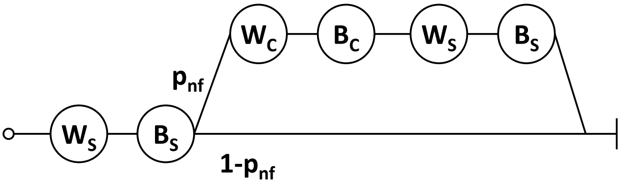


Figure 2.16: *Phase diagram of the packet sojourn time.*

Assumptions

In the forward queuing system, we take the simplifying assumption that the overall arrival process at the switch (forward), as well as the arrival process at the controller (feedback) are Poisson. This can be justified as the state processes on the forward and the feedback paths are on very different time scales, which allows the decomposition of the two queuing systems. Moreover, we queue all packet arrivals in a single queue at the switch, instead of a separate queue per line card. The feedback queue used to model the controller actually comprises out of the line-out card of the switch towards the controller, the buffer, and processing at the controller itself. However, the Nox controller we used as a reference for this model controls the traffic rate it receives from the switch in order to prevent overload. Therefore, no additional queuing happens at the controller itself, and the line-out buffer at the switch is the only place where packet loss may happen. The transmission time of packets from the switch to the controller is encapsulated in the service time of the controller.

Limitations of the Model

In its current form, the model does not capture the fact that incoming traffic at the switch is queued first per line card, i.e., one queue per port. As well, it is currently limited to a single switch per controller, whereas OpenFlow allows the same controller to be responsible for a number of different switches. Furthermore, the model assumes TCP traffic as opposed to UDP traffic. For TCP traffic only the first packet header of a new flow is sent to the controller, while for UDP all packets are relayed until a flow rule is in place. These limitations will be addressed in future work by refining the model described here. Refinements may contain replacing the forward input queue by a polling system or replacing the $M/M/1-S$ system of the controller with a more general $M/GI/1-S$ system that allows the use of a measured service time distribution as an input for the model.

Although it might be desirable to have the complete distribution for the model, we focus on the first two moments only as they provide us the most important performance indicators required by a service provider for dimensioning the network.

2.4.3 Analytical Results for the Simplified Model

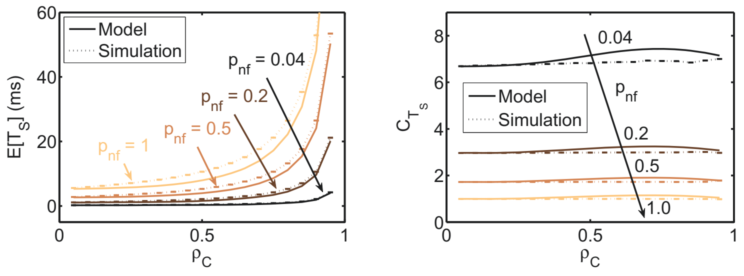
In this section, we discuss the output of our model and validate them by means of a simulation. We use a set of scenarios depicting different use-cases for an OpenFlow-enabled switch. This is mainly reflected in the forwarding probability p_{nf}. A value of $p_{nf} = 0.04$ represents a normal productive network carrying end user traffic, where [46] showed that this is the probability for new flows being observed at the switch. Values of $p_{nf} = 0.2$ or 0.5 model a network where a part of the traffic is routed via the controller, e.g., if a portion of the traffic is using a virtualized network. Finally, $p_{nf} = 1$ depicts the case where the controller handles the complete traffic going through the switch, e.g., in an experiment testing new protocols, such as described in [11].

We also vary the mean service time of the controller. The slowest value is two orders of magnitude larger than the service time of the switch, which may be the case if commodity hardware is used to run the controller. We provide results for faster controller systems as well to allow to predict the system behavior if dedicated hardware is used. The buffer size at the controller queue is set to $S = 512$ packets in all experiments, which was chosen arbitrarily as a middle ground between experimental and commercial switches.

To be able to validate the results from the analytical model, we implemented a packet based simulation in OMNeT++. The simulation model also reflects the structure of the analytical model, cf. Figure 2.15. Verification through simulation was chosen over measurements as simulations results are faster to obtain and do not require as many repetitions to stabilize. The process time of the controller in the simulation uses the same distribution as the analytical model. In contrast, we are able to study the feedback generated by packets looping back over the controller in the simulation, which we are not able to fully reflect in the analytical model. The assumptions about the arrival and departure process of the controller are also relaxed in the simulation, i.e., no Poisson process is assumed. Instead, the actual inter-departure times of packets after traversing the switch and the controller, respectively, are used.

The simulation results shown in this section are based on six simulation runs per parameter and different seeds. The 95% confidence intervals are given for the simulation results in all figures, but are only represented by dashes as they are very small compared to the scale.

Figure 2.17a illustrates the modeled as well as the simulated mean sojourn time of a packet depending on the controller load for an expected controller service time value of $E[B_C] = 5.3ms$. Graphs are shown for several p_{nf}, i.e.,



(a) Mean (b) Coefficient of variation

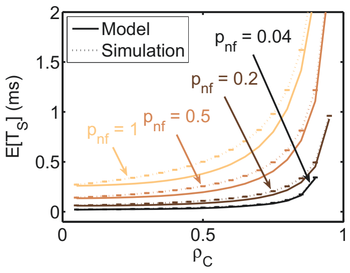
Figure 2.17: *Impact of the forwarding probability on the packet sojourn time for* $E[B_C] = 5.3ms$

the probability that a received packet at the switch represents a new flow and subsequently causes the switch to send an OpenFlow packet, which needs to be answered by the controller. Values are given for controller loads from 5% to 95% in 5% steps.

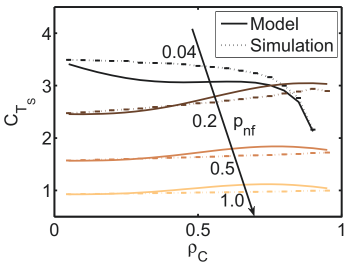
As a first general observation, we see an increase in the mean sojourn time which is caused by the influence of the controller. The more packets need to wait for a controller response, the more packets have a longer sojourn time caused by the controller service time and additional waiting time imposed by queuing. Since the controller reaches a high utilization sooner than the switch with an increasing λ_0, i.e., an increasing raw traffic rate, it contributes a much longer waiting time to the total packet sojourn time.

In case of $p_{nf} = 0.04$, the mean sojourn time of the system barely deviates from that of the switch, since only a small fraction of traffic has to be handled by the controller. Only at a controller load of 75% and above we observe the start of an exponentially rising gradient. If we increase p_{nf} to 0.2, we see this increase at a controller load of about 45%. For $p_{nf} = 0.5$ it already starts at a load of 30% and at the maximum value $p_{nf} = 1.0$ we detect the increase already at 10% controller load. In all cases, the simulation curve is located slightly above the curve for the model. For low values of p_{nf} this deviation appears to be marginal. Here, simulation and model show a nearly identical progression. With an increasing p_{nf}, we see a deviation increase except for very high controller loads. However, the deviation remains small.

In Figure 2.17b, the coefficients of variation for the sojourn times are shown also dependent on the controller load. We observe an increasing coefficient of variation with a decreasing p_{nf}. This is caused by the fact that with a smaller p_{nf} less packets are subject to the delay imposed by the controller and therefore, the deviation from the mean value for these packets is much higher. Again, we can also see a small discrepancy between simulation and model for medium controller loads. This discrepancy increases with smaller values for p_{nf} complementary to what we see in Figure 2.17a. However, simulation and model seem to be a good fit.



(a) Mean



(b) Coefficient of variation

Figure 2.18: *Impact of the forwarding probability on the packet sojourn time for* $E[B_C] = 240\mu s$

Figure 2.18a depicts the simulated as well as modeled mean sojourn times dependent on the controller load for an expected controller service time value of $E[B_C] = 240\mu s$, corresponding to Figure 2.17a. For $p_{nf} = 0.04$ and a controller load of 95% no value is given as this would violate our assumption of an offer $a \leq 1$. Overall, we observe the same effects as discussed for Figure 2.17a albeit for mean sojourn times two orders of magnitude smaller. The point where the gradient starts to increase exponentially is shifted to lower controller loads, e.g., for $p_{nf} = 0.04$ we see an increase already at 45% load as opposed to 75% in Figure 2.17a.

Corresponding to Figure 2.17b, Figure 2.18b shows the coefficients of variation for the sojourn times for $E[B_C] = 240\mu s$. While the progressions of the coefficients are very similar here to those in Figure 2.17b for $p_{nf} = 0.5$ and $p_{nf} = 1.0$, they differ for $p_{nf} = 0.2$ and $p_{nf} = 0.04$. Contrary to Figure 2.17b, the model curve for these two values of p_{nf} now underestimates the simulation for small controller loads. We also note an increasing gradient for $p_{nf} = 0.2$. For $p_{nf} = 0.04$ we see an exponential decrease in the coefficient of variation at controller load above 70%. Furthermore, the model curve now underestimates the simulation for all observed controller loads. In this scenario, not only the delay imposed by the controller is relevant, but we also observe a non-zero waiting time at the switch. This is caused by the now much smaller difference between the service time of the switch and that of the controller. A high utilization of the controller also leads to a non-negligible utilization of the switch, resulting in a non-empty queue.

Finally, Figure 2.19 displays the simulated as well as modeled mean sojourn times dependent on the controller service time for $p_{nf} = 1.0$. We can observe the influence of the controller performance relative to the switch performance on the total system. As we have seen in Figure 2.17a and Figure 2.18a, the mean sojourn time increases exponentially with increasing load. However, for an $E[B_C] = 5.3ms$ the gradient already shows a much higher increase at smaller controller loads than that for $E[B_C] = 240\mu s$. With an $E[B_C] = 31\mu s$, the curve is governed by the switch delay and therefore, the mean sojourn time is barely distinguishable from the service time of the switch and does not show an increase at all.

In all cases, the observed blocking probabilities (p_B) for packets at the con-

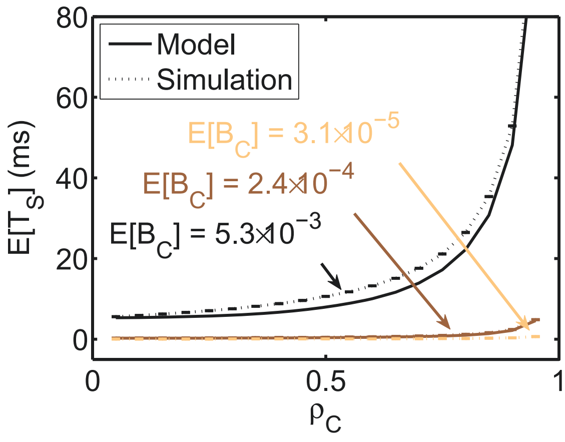


Figure 2.19: *Impact of the controller service time on the mean packet sojourn time.*

troller queue were zero for the simulation and infinitesimal small in the model. This indicates that the OpenFlow architecture is stable for our input values.

2.4.4 Generalizing the Model

The model we have introduced helps to predict the mean sojourn time and packet loss by using mean values as input to estimate the service times of both switch and controller. However, we wanted to improve the prediction for the mean service time by generalizing our model to use arbitrarily distributed service times in the controller model and arbitrarily distributed arrival times in the switch. This enables us to exploit the characteristics of measurement data as a basis for our sojourn time estimation. This is especially important since controller performance can vary greatly depending on the network application running on top of it. Another difference to our previous model is the removal of the packet loss calculation at the controller as our previous results have shown that this is only an issue when the controller is overloaded.

2.4.5 OpenFlow Controller Service Time Distribution

In this section, we describe how to obtain a realistic service time distribution for the controller and discuss the results.

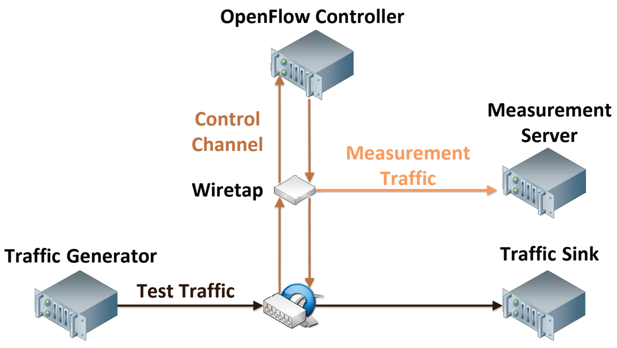


Figure 2.20: *Testbed set up to controller delay*

In the previous section, we used a measurement testbed to obtain the mean values for the service time of an OpenFlow switch and the Cbench [38] benchmark to perform the same task for the controller. This was sufficient as our initial model required only mean service times to estimate the mean sojourn time of a packet. However, for a generalized controller queue as we propose, it is necessary to have a complete service time distribution as input parameter. Therefore, we decided to adapt our existing testbed to accurately measure the delay imposed by the controller instead of relying on a software-based measurment. The testbed is depicted in Figure 2.20. For traffic generation, we use an HP Proliant DL320 server system with a 2.13 GHz quad core CPU and 10 GB of RAM. Traffic is generated by the Linux tool “pktgen” in such a way that every packet represents a new flow. The traffic generator is connected to an OpenFlow switch, in our case an NEC IP8800, via a 1 Gbps Ethernet link. However, the link is not saturated to prevent overloading the controller. When a packet reaches the OpenFlow switch, it triggers the creation of an OpenFlow "packet-in" message to the controller, which is then transmitted via a dedicated 1 Gbps Ethernet link to the controller. The traffic on the link is mirrored bidirectionally using a Net Optics Wiretap and sent to the measurement server, which is also an HP Proliant DL320 server with a 2.13 GHz quad core CPU and 12 GB of RAM. The system utilizes an Endace DAG 7.5 G2 card, which allows time-stamping of packets with a precision of less than 9 nanoseconds according to the manufacturer [47]. The controller used is NOX 0.9 running the pyswitch module on top of a standard Fujitsu Esprimo P7935 PC with an Intel Core 2 quad core CPU and 8 GB of RAM. Any other controller and application could have been used as the goal of this measurement was to obtain a sample as input for our model. However, we chose this combination as in our opinion it is the most easily reproducible. Once the OpenFlow message has been processed by the controller, a reply message is sent back to the switch. This message packet is also mirrored to the measurement server. With the message arriving at the switch, the original packet, which was stored in the buffer, is then transmitted to the traffic sink. Therefore, the delay measured is the time from the moment the OpenFlow "packet-in" message passes the Wiretap's upstream interface until the corresponding reply is recorded on the downstream interface.

Figure 2.21 shows the CDF of the controller delay for five individual measurement runs, each using a sample of 10000 packets. The mean value of all samples is 0.366 ms with a coefficient of variation (c_B) of 0.149. We observe distinct "steps" in the graph, i.e. large percentages of packets possess similar processing times. We see an increase of about 12%-15% at 0.28 ms delay, an increase of about 28%-35% at 0.31 ms delay, an increase of about 20%-30% at 0.38 ms delay, and an increase of about 28%-30% at about 0.4 ms delay. It is likely these steps are the result of differences in processing time for packets that are destined for hosts as of yet unknown to the switch resulting in flooding and those already known resulting in the creation of a flow rule. We use this input for our controller model.

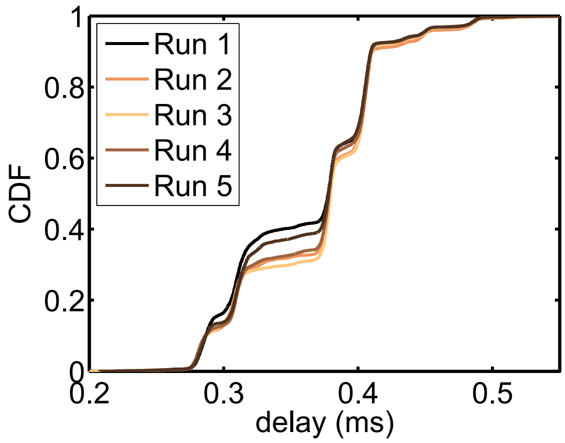


Figure 2.21: *CDF of the measured controller delays*

2.4.6 OpenFlow Architecture Model using Generalized Controller Service Times

The generalized model we introduce is an adaptation of our model presented in Section 2.4.2. Therefore, we put an emphasis on the differences between the two models in this section. The architecture model as shown in Figure 2.22 is composed out of two independent queuing systems for the switch- and control path respectively. The goal of the model is to predict the mean of the system's sojourn time $E[T_S]$, i.e., the time a packet takes on average to pass through the system, as well as its coefficient of variation c_{T_S}.

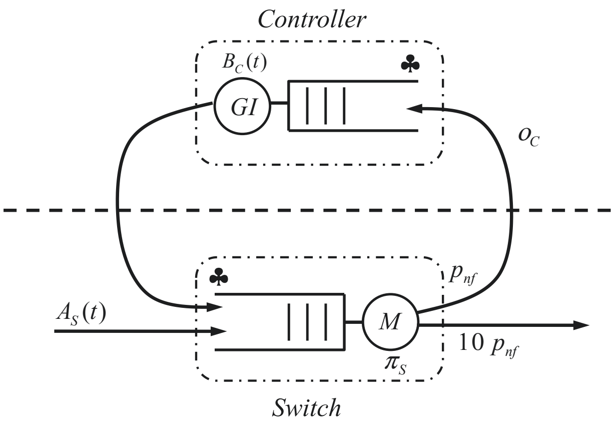


Figure 2.22: *OpenFlow system model with generalized controller service times.*

The Switch Queue

Packets arrive at the the OpenFlow switch, according to an arbitrary distribution $A_S(t)$. However, the arrival process should not result in a rate exceeding the offered service as this would produce invalid results. The switch is modeled as a GI/GI/1 system. This generalizes our previous model, which assumed exponentially distributed arrival and service times. The queue length is assumed to be infinite as the switch buffer is relatively large and should not be a bottleneck in non-overload scenarios. As a full empirical switch delay distribution is not available to us, we use an exponential distribution for the switch service time resulting de facto in a GI/M/1 system as depicted in Figure 2.22. We numerically calculate the results for the switch queue using a time-discrete analysis based on the algorithm described by Tran-Gia in [48]. It iteratively calculates the waiting time distribution by solving the general form of the Lindley equation. To gain the sojourn time distribution for the switch, the resulting waiting time distribution is then convoluted with the service time distribution. The possibility that the packet is the first of a new flow unknown to the switch is modeled through the probability p_{nf}. In this case, additional processing by the controller is required. All packets belonging to known flows leave the system after processing with the probability 1-p_{nf}.

The Controller Queue

The controller chain is modeled independently as an M/GI/1 system with the packet arrival rate λ_C. Herein lies a major difference to our previous model. The controller service time for each new flow packet is now determined by an arbitrarily chosen distribution, which can reflect actual performance measurements and thus improves the accuracy of the model. Additionally, the controller queue is now assumed to be infinite to maintain a fast computing time as packet loss did not appear to be a factor in Section 2.4.3. The mean waiting time of the controller queue is calculated using the Pollaczek-Khintchine formula and its higher moments using the Takacs formula as described in [48]. The mean sojourn time of the controller queue can then be calculated as the sum of the switch's mean waiting and service time. After controller processing is complete, the reply once again has to be examined by the switch. This is modeled as a secondary pass through the switching chain after which the packet leaves the system.

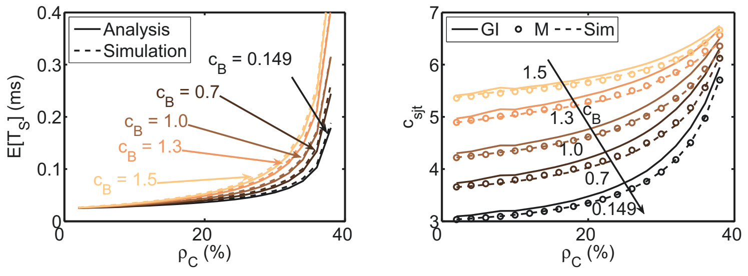
Model Assumptions and Limitations

Some of the basic assumptions and limitations of our original model still apply. The switch is assumed to have a single queue. However, as we use a single processing unit and queuing rarely occurs in the switch when the controller is not overloaded, the use of a single queue should not result in significantly different results from a model using multiple queues without prioritization. Furthermore, the model only reflects TCP traffic as for UDP flows all of their packets would be sent to the controller until a forwarding rule was in place.

2.4.7 Analytical Results for the Generic Service Model

In this section, we discuss the results of the generalized model and compare them to our original M/M/1 switch queue in conjunction with the generalized controller queue as well as to our simulation results. By using a sampled exponential distribution as arrival process for our generalized queue, we ensure comparability with our original model. The arrival rate is calculated depending on the desired controller load ρ_C. For this analysis, the probability for a new flow p_{nf} as well as the mean service time for switch μ_S and the controller $E[B_C]$ are left fixed. Reflecting findings from [46], p_{nf} is set to 4% and according to our measurements, μ_S is set to 9.8 microseconds. While the mean service time $E[B_C]$ is left fixed at 0.366 ms as determined by our measurements in Section 2.4.5, we are looking at different service time distributions at the controller with varying coefficients of variation c_B to emulate the behavior of network applications and their impact on the mean sojourn time of the system. The starting point is the empirical service time distribution yielded by our measurements, cf. Section 2.4.5, with a coefficient of variation of c_B 0.149. Additionally, we choose a log-normal distribution with a c_B of 0.7, an exponential distribution with c_B of 1.0, and two hyper-exponential distributions with a c_B 1.3, and 1.5 respectively for our analysis. A coefficient of variation of 1.0 corresponds to the behavior of our previous model with exponentially distributed service times.

Figure 2.23a shows the mean sojourn time in relation to the controller load ρ_C for the different controller service time distributions. Analytical results are displayed by a solid line, whereas simulation results are represented by a dashed line. The simulation is an adapted version of our OMNeT++ simulator. The analytical results obtained using the original switch queue are almost identical for the mean sojourn time. Hence, we omitted them for clarity in this graph. We observe a tendency for distributions with a higher coefficient of variation to cause a generally higher mean sojourn time. As expected, we also see an upturn in sojourn times with increasing controller load for all distributions due to packet queuing. What is interesting, however, is at which load we can observe the upturn and its



(a) Mean $E[T_S]$ (b) Coefficient of variation c_{T_S}

Figure 2.23: *Impact of the controller service time volatility* c_B

steepness. For our measured distribution with the lowest c_B in the comparison, it starts at a controller load of about 50% with a small gradient. With increasing c_B, this point moves left towards lower controller loads and also the gradient becomes more steep. At the highest c_B of 1.5, we see the start of the upturn already at about 20% load and the gradient is significantly higher than that for a c_B of 0.149. Another important observation is that model and simulation seem to fit quite well for all distributions, which validates our analytical model. Confidence intervals are very small and have also been omitted from the graph.

Figure 2.23b illustrates the impact of different controller service time distributions on the coefficient of variation of the sojourn time (c_{T_S}) in relation to the controller load. Analytical results obtained through our generalized model are given by a solid line, those obtained using our original M/M/1 queue are represented by circles, and simulation results are shown using a dashed line. The overall tendency here is that systems with a lower coefficient of variation at the controller service time also have a lower overall coefficient of variation. In general, the coefficient of variation increases with higher controller loads. This is due to the effects of queuing at the controller. The controller queue is inherently slower than the switch queue and with additional queuing occurring, this effect is amplified. For our measured distribution this means an increase in the coefficient of variation of the sojourn time from 3 at about 5% load to about 5.5 at 95% load, whereas the service time distribution with a c_B of 1.5 increases from 5.5 to roughly 6.5. The increase is disproportional as a high c_B at the controller already causes a significant amount of queuing without the impact of additional load. The very different processing speed between controller and switch also explains the high impact of an increased c_B at the controller on the overall coefficient of variation. We observe that an increase of just 1.351 in the c_B at the controller causes the overall system c_{T_S} to increase by about 2.4. While the analytical results using the M/M/1 queue fit the simulation quite well, the generalized queue slightly overestimates the coefficient of variation by 0.1-0.3. This is the result of numeric inaccuracies, scaling, and sampling in the computation. However, the benefit of a generalized model far outweighs the introduced error as it allows us to analyze a larger variety of scenarios by far. Furthermore, the discrepancy is quite small and as we are interested in a worst case approximation of the coefficient of variation a small overestimation is acceptable.

2.5 Lessons Learned

In this chapter, we have taken the first step to understand the performance of SDN on the example of OpenFlow. To this end we have investigated the performance of OpenFlow data plane and control plane separately.

Though all tested OpenFlow switches performed their tasks and were stable during the data plane measurements, there are huge differences in their real world applicability. The performance of the Open vSwitch as a free and flexible software implementation makes it very attractive for functional testbeds and scenarios where a relatively low OpenFlow forwarding performance is required. This is also true for the NetFPGA card. While being at least one order of magnitude faster than the Open vSwitch, this is only true in scenarios with 24 or less flow rules. Otherwise the card runs out of fast memory and also has to rely on the software path making it unsuitable for large scale scenarios. While the Open vSwitch can run distributed over several machines, the NetFPGA is also limited to its only four on board interfaces, which also limits its usability outside of a lab. The Pronto 3290 is built on a generic Broadcom network processor board. With its 48 interfaces and relatively cheap price, it is very attractive for experimental campus deployments. While the flow forwarding performance is adequate, the heavy influence of disruptive traffic casts its usability in high performance scenarios in doubt. In general, all tested switches do not appear to be ready to be used in a production environment. There appear to be too many unforeseen effects when load is applied. Of course, with the exception of the Pronto 3290, the switches are not intended for high-load situations and the Pronto appears to be a stop gap solution. Furthermore, the traversal of packets from the data plane to the OpenFlow control plane presents a significant performance challenge and should be avoided as much as possible, which limits the areas of application for these switches in an SDN deployment.

Given the importance of the OpenFlow controller for the software defined network it directs, it is key to understand the performance and behavior of this important software component for experimenters as well as for operators of productive networks. To this end we introduced our approach to a flexible and granular benchmarking and analysis system for OpenFlow controllers to gain this insight and understanding. We show that the results for conventional throughput tests are comparable to the results of the current reference benchmark Cbench and that it is possible to run the benchmark on conventional hardware. Our benchmark results, especially those in terms of outstanding packets, underline the importance of a more granular view on the system to detect performance bottlenecks and similar issues that can not be grasped from an aggregated perspective. Without this granular view and the resulting insight into the behavior of the controller, the performance of the network can not be guaranteed and in the worst case a node- or even network-wide failure may be the result.

In order to gauge the performance an scalability of the entire OpenFlow system in the planning phase, we proposed a basic model to analyze the forwarding speed and blocking probabilities of an OpenFlow architecture. Blocking can thereby only occur in the forwarding queue to the controller. The results show that the sojourn time can be greatly influenced by the processing speed of the OpenFlow controller. Our measurements have shown that the processing time of the controller lies between 220 μs and 245 μs. The impact of the controller processing time can be best seen in the variation of the sojourn time. The higher the probability of new flows arriving at the switch, the lower is the coefficient of variation, but the longer is the sojourn time.

The presented model once more underlines the importance of the controller performance for installing new flows. When using OpenFlow in high speed networks with 10 Gbps links, several controller implementations are not able to handle the huge number of new flows.

In light of this, we extended the OpenFlow performance model to better reflect the controller performance. The extension allows the use of generalized service time distributions at the controller as well as generalized arrival time distributions at the switch. The comparison of results between analytical and simulation model allows us to conclude that the assumptions made in our model are reasonable for our measured values as the difference between analytical and simulated results is negligible. This is an improvement over the previous model due to the possibility to use real measurement data. Therefore, it is the next important step towards building an accurate OpenFlow abstraction model, which allows researchers, developers, as well as engineers to appraise the performance of their network software and deployments. The results also show that a high variability in processing delay has a significant impact on the overall forwarding delay.

While current efforts to improve and quantify OpenFlow performance are rightly focused on the data path as our measurement results have shown, in our opinion, the controller performance is of the same importance and will shift more into view once the second generation of OpenFlow-enabled devices will have remedied some of the first generation's shortcomings.

The main contribution of this chapter is two-fold. First, it provides the conclusive statement that SDN does work as a concept even when used in reactive mode. Second, it shows that in spite of this there are still significant challenges ahead in order to raise the existing implementations to a level of performance that is required for production networks.

3 SDN Control Plane Applications

The main selling points of SDN beyond the potential cost reduction are an increased flexibility and simplification of existing network services and the creation of novel network applications. In this chapter, we investigate whether this is a realistic prospect.

As a representative of an existing network service, we choose the monitoring domain. We begin by highlighting some select contributions in the area of (SDN-based) network measurements in Section 3.1. We then investigate how accurately a purely SDN-based approach can measure network parameters compared to a full reference packet trace in Section 3.2. Furthermore, we highlight the potential cost and implementation difficulties of the method. We do this by performing measurements using the SDN-based approach in an OpenFlow test bed, while simultaneously mirroring and capturing the measurement and control traffic.

We proceed to implement two novel solutions to challenges in the research test bed and data center domains in Section 3.3. These new approaches use SDN to change the way current network tasks are performed and show that the operation of these networks becomes more simple and at the same time more flexible using SDN.

This chapter is mainly based on and taken from [6], [9], and [10].

3.1 Previous Works on (SDN-based) Measurements

In [49] Zseby evaluates sampling methods for passive Quality of Service (QoS) measurements in conventional networks and highlights the challenges when implementing such an approach. An SDN-based approach appears suitable to levy these challenges.

In recent years there have been several works that investigate ways on how to leverage SDN and specifically OpenFlow for network measurements and monitoring. Tootoonchian et al. [50] propose an OpenFlow-based approach for traffic matrix estimation by intelligently querying flow table counters. We evaluate the accuracy of these kind of queries for bandwidth measurements.

In [51] Jose et al. investigate the possibility to measure large traffic aggregates in commodity switches, which leads to Yu et al. [52] introducing the measurement architecture OpenSketch, which, similarly to the OpenFlow concept, separates the measurement data plane from the control plane. While this is an interesting approach, we focus on the SDN control plan itself represented by an OpenFlow controller.

Yu et al. [24] introduce the FlowSense concept, an OpenFlow-based approach to network measurements with minimal measurement costs. We extend this approach by also taking latency measurements into account, which we discuss in the following Section 3.2.1.

3.2 Accuracy of Leveraging SDN for Passive Network Measurements

In today's networks the monitoring of QoS parameters like bandwidth, packet loss, and delay is essential to ensure the smooth operation of multimedia applications as well as the control of service-level agreements and fault detection. This is often done using a measurement setup that actively sends traffic through the network. However, this approach requires expensive special-purpose measurement equipment. Furthermore, such a measurement often can only be performed in off-peak hours as an active measurement probe could disrupt critical production traffic. Thus, only limited statements are possible for the QoS experienced during business hours from inside the network. While today the application itself can actively monitor a subset of its own QoS parameters, it can not directly influence the network.

The introduction of SDN gives the network the ability to passively perform network-wide QoS measurements relying on the actual production traffic. This is possible using other techniques, but the SDN approach essentially turns network measurements into a primitive function of the network itself, eliminating the need for additional devices and allowing for a more representative view of the network state by increasing deployment flexibility. This in turn can be used as direct input for SDN network control.

3.2.1 Measurement Architecture

In addition to bandwidth measurements, our extended SDN measurement architecture based on FlowSense [24] also takes one-way delay measurements into account. Figure 3.1 illustrates the concept on the example of a connection in an intermediary SDN network from switch A to switch B. All switches in the network are each connected to the SDN controller via a control channel. This connection is used for switch control on the one hand and on the other for the polling of statistics information from the switches' flow tables.

The flow tables contain packet and byte counters for each entry. By retrieving their current values the controller can calculate the current bandwidth consumed by a traffic flow matching an individual rule. In this case, that is the bandwidth consumption of our flow on all highlighted links between switches from A to B. No additional components are required in the network to measure bandwidth consumption and the information about it can be directly used to influence the controller's policy for the network or to optimize the placement of the controller(s)

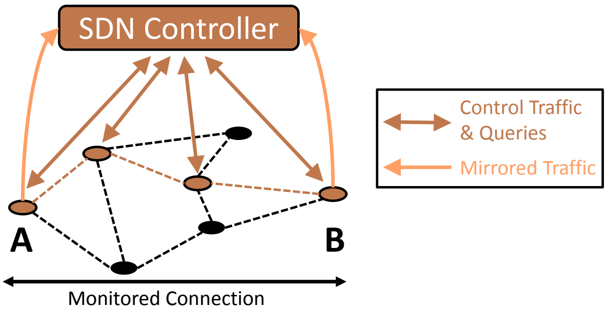


Figure 3.1: *Measurement Architecture*

within the network [53]. However, this method requires frequent queries to the individual devices in order to be accurate for a desired interval.

For the purpose of delay measurements, the methodology is different. Suppose the goal is to measure the delay on the connection from A to B in our example network. In traditional networking, a mirror port would have to be configured on the ingress and egress device that would then send all the traffic to a measurement station, which would than have to filter the data for the desired flow information and calculate the delay between the packets received from A and B. With the SDN approach, the controller can simply insert a temporary flow rule into switches A and B to send all or a sample of the packets to the controller parallel to forwarding them through the network. This way the controller again becomes the measurement device and can directly react on the network information without additional equipment.

This approach has several difficulties. We do not know how accurately the software controller can measure and calculate bandwidth and delay without the support of special-purpose hardware. Furthermore, the load on the control channel may be significant using this approach and it is in general unknown how this impacts network control. It is the goal of this chapter to answer some of the questions regarding accuracy.

As an alternative to the above described methods, a hybrid approach between conventional an purely SDN-based measurements is possible. In this approach, the flexibility of SDN is used to selectively mirror network flows to a dedicated measurement device instead of the controller. However, this method requires an additional special-purpose device and the accuracy is determined by the implementation that device itself. Therefore, the focus of this chapter lies on the purely SDN-based approach.

3.2.2 Testbed Setup

We use an OpenFlow-based testbed to evaluate the accuracy and overhead of the purely SDN-based measurement approach described in Section 3.2.1. The testbed is shown in Figure 3.2. It realizes a simplified version of the scenario in Figure 3.1. Iperf [54] is used to send a 1 Mbps UDP flow from the traffic generator to the traffic sink representing the production traffic that should be measured. The flow passes through two Pica8 Pronto 3290 switches, which represent the ingress and egress nodes of our intermediary network. Bandwidth and delay variations experienced in the network are emulated using NetEm [55] on a Linux PC. We use Floodlight [44] with a custom measurement module as OpenFlow controller running on a Dell Poweredge 860 server. The SDN-based measurements are performed using this controller. For the delay measurements, the OpenFlow switches send the traffic to the controller as well as to its destination using two OpenFlow output actions. Bandwidth is measured by regularly sending OpenFlow statistics requests to the switches. The reference measurements are performed in parallel on a separate HP Proliant DL320 server using either an Endace DAG 7.5G2 capture card or a conventional network card in conjunction with TCPdump. The traffic is mirrored to this server using two Netoptics wire taps.

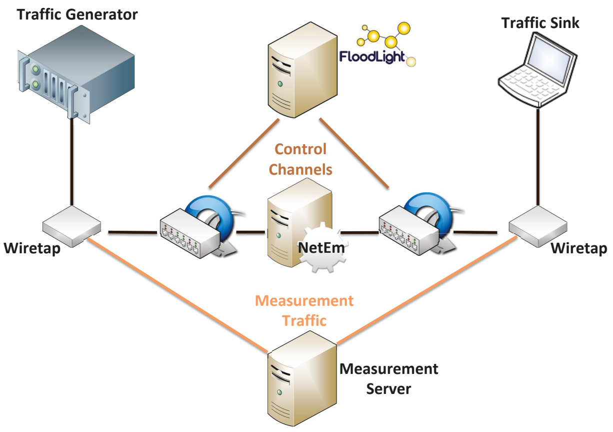


Figure 3.2: *Testbed Setup*

Technical Considerations

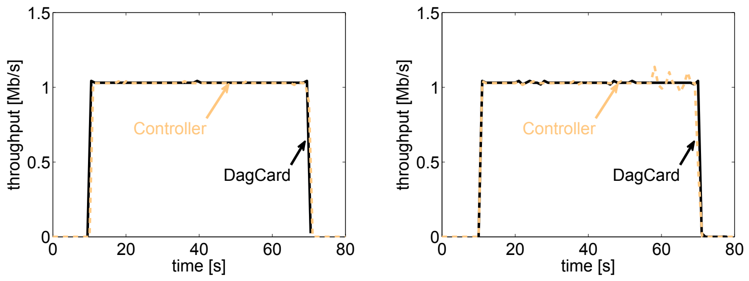
As the traffic for the SDN as well as the reference measurement is mirrored at different locations, a discrepancy in the measured delay is expected. This discrepancy is caused by the processing delay of the two OpenFlow switches. Furthermore, the measurement probe cannot exceed bandwidths of much more than 1 Mbps as the OpenFlow implementation on both OpenFlow switches handles the implementation of an OpenFlow send-to-controller action in software, i.e. on the slow path, which is limited by the relatively slow switch CPU. Future OpenFlow switch implementations will likely not be constrained by this issue. We discuss the impact of this in the following section.

Another influence factor on the accuracy of the measurements is the latency on the links of the control channels between the OpenFlow switches and the controller (cf. Figure 3.2). Before a packet arriving at the controller from one of the two switches can be timestamped, it has already experienced additional delay from processing at the switch and the transmission via the control channel. This is also true in the conventional measurement approach we have chosen as reference and can only be avoided, if the packets are timestamped at the switches and the internal clocks of the switches are precisely synchronized. Therefore, in our case an additional requirement has to be met. The term $|(\Delta t_{process_1} + \Delta t_{propagate_1}) - (\Delta t_{process_2} + \Delta t_{propagate_2})|$, where $\Delta t_{process_x}$ reflects the processing time in the switch and $\Delta t_{propagate_x}$ is the propagation delay on the control channel, has to be smaller than the desired measurement accuracy. We meet this requirement in our testbed by using identical OpenFlow switches and control channel cabling. In a real world deployment, it is also likely that identical hardware would be used and the impact of latency could be kept small by using a distributed controller and placing an instance close to the measurement point.

For the bandwidth measurements, the frequency of updates is limited to one second intervals as the OpenFlow switches only update the statistics counters in their flow tables once every second.

In an SDN deployment, it is likely that the controller would be run on a virtual machine inside the cloud. Therefore, we have performed our tests with the controller running either on the aforementioned server or in a virtual machine hosted on an identical server using the free version of the VMware ESXi 5.1 hypervisor. Particularly delay measurements require precise timekeeping in order to be accurate. As the SDN controller is run in software relying on the system hardware clock, this can not always be guaranteed. We expect this to be even more of an issue in a virtual environment with not only different processes but virtual machines competing for processing time.

The scalability of the SDN measurement approach is limited by two factors. These are the processing capacity of the controller and the control channel bandwidth. As the controller would likely be run in a cloud environment for large setups, the processing capacity can be scaled up dynamically to the required level. However, when a distributed controller approach with different switches connected to different controller instances is used to achieve this, the clocks of these instances need to be synchronized. The control channel bandwidth required for the measurements can be reduced using sampling techniques. However, the number of flows that can be monitored simultaneously will still have an upper limit.



(a) Non-Virtual Controller (b) Virtual Controller

Figure 3.3: *Used Bandwidth*

3.2.3 Measurement Results

In this section, we discuss the results of our measurements. All measurement runs were repeated at least five times in order to ensure consistency.

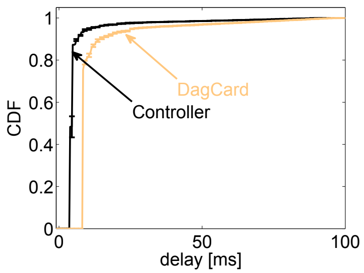
Measuring Bandwidth

As a base test for our setup we chose a bandwidth measurement using the already mentioned statistics requests. This test is very similar to those performed with FlowSense. Therefore, we use it to verify our method and setup. Figure 3.3 shows the measured bandwidth consumed by the measurement flow over the duration of a 60-second test run for both the virtual and non-virtual controller. For comparison, all packets were captured using the DAG card in the measurement server. As can be seen in the figure, in both cases the measured throughput reaches the configured 1 Mbps of the measurement probe and subsides after the traffic generator

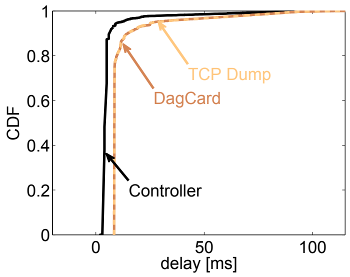
has stopped sending packets.
The SDN measurements behave nearly identical to the capture trace of the test run. While this is true for the mean of all runs performed, we can observe some inaccuracies in the particular run shown in Figure 3.3b. At around the 55 second mark, the bandwidth measurement at the virtual controller varies for several kbps around the reference value. There are two possible explanations for this behavior. It could be caused by time drift of the virtual machine clock, which is only synchronized with the server's hardware clock at specific intervals. Therefore, the controller could no longer reliably schedule its statistics requests and the query interval varies slightly leading to inaccurate bandwidth calculations.
The second explanation is that either the query or response packets for the statistics in question were delayed by either the virtual switch in the hypervisor or by the management plane of the OpenFlow switch. However, since we do not have accurate timestamps for the control channel messages, it is not possible for us to determine which is the case. Still, the variation appears marginal and thus the approach appears to be usable at least for bandwidth measurements at this frequency. A more granular resolution would require the switches to support more frequent counter updates and would involve significantly more queries to the switches' minimal control plane. This would require a much more powerful switch CPU to handle these more frequent requests and sufficient bandwidth on the control channel.

Measuring Latency

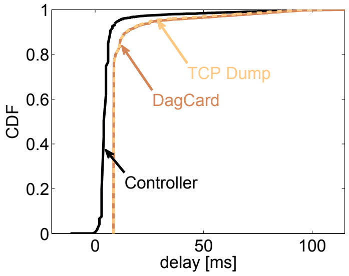
In this section, we discuss the delay measurements. Figure 3.4 shows the cumulative distribution functions (CDFs) of the delay measurements performed without the introduction of any artificial delay between the two measurement points. Figure 3.4a shows the results for the non-virtual controller and the DAG card with 95% confidence intervals for five individual runs. We observe a measured delay of 4-5 ms for about 86% of packets with the controller, whereas we see almost double that delay on the capture trace for 78% of the packets.



(a) With Confidence Intervals, Non-Virtual Controller



(b) Non-Virtual Controller



(c) Virtual Controller

Figure 3.4: *Latency Cumulative Distribution Functions (No Artificial Delay)*

As can be seen, the confidence intervals are small, indicating a good estimation of the probability for a certain delay. The exception is the ratio of packets with 4-5 ms delay as measured by the non-virtual controller. Here, the confidence interval is in a range of about 10% difference for the runs. However, this shows that our results are statistically stable. Therefore, we only use one exemplary test run for each test in the remainder of the figures in order to enhance readability.

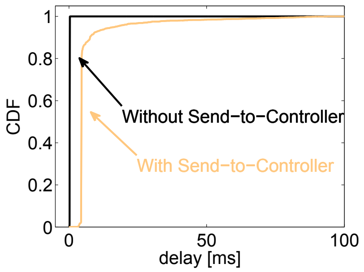


Figure 3.5: *Latency Cumulative Distribution Functions (With and without Send-to-Controller)*

As described in Section 3.2.2, our OpenFlow switches handle packets with a send-to-controller action in software, which explains this considerable delay imposed on the packets. The discrepancy between controller and capture trace is caused by the difference in measurement points in our testbed. While the measured delay at the controller is only imposed by the sending process in the first switch and the receiving process in the second switch, the capture trace sees the delay imposed by the sending and receiving processes of both switches. This means that the delay measured using the capture trace is twice as long than the delay measured using the controller. For confirmation of this circumstance, we perform measurements with the hybrid approach described in Section 3.2.1, using the OpenFlow switches to mirror traffic to the DAG card. Measuring at the same locations in the network, we can determine the impact of the send-to-controller action by running the tests with the action enabled and without. Figure 3.5 shows the cumulative distribution functions of the results. As expected, a clear discrepancy between the two curves is visible. Whereas almost all packets from the measurement without the send-to-controller action experience a delay of less than 1 ms, the packets with the send-to-controller action enabled show a delay of 6-7 ms and above. The additional latency of the value measured here to that in Figure 3.4 can be explained by the fact, that the switch now has to perform three actions in software instead of two, i.e., forward packet, forward to controller, and forward packet to the DAG card.

Figure 3.4b shows the results of a single test run for the non-virtual controller. In addition to the capture trace of the DAG card, a trace using just TCPdump is also shown. The behavior is very similar to the one observed in Figure 3.4a. We can see that with this amount of switching delay, the high time resolution of the DAG capture card does not present a significant advantage over a conventional TCPdump trace.

As expected, the results for DAG card and TCPdump do not differ greatly from these in the test using the virtual controller as shown in Figure 3.4c. However, we observe that the increase in the CDF graph for the delay measured with the virtual controller is not as steep as with the non-virtual controller. There is a visible gradient. About 84% of the packets are measured with a switching delay of 1-5 ms, which is a significantly greater value range than the 4-5 ms measured for the non-virtual controller. Furthermore, if we look at the lower end of the CDF plot, we see that the CDF does not start at 0 ms as can be seen in more detail in Figure 3.6. A small but visible percentage of packets appears to have experienced a negative delay. This cannot happen in reality and must be caused by a measurement error. This result seems to confirm our theory from Section 3.2.3 that inaccurate time keeping in the virtual machine causes irregularities in the results. If processing issues at either the virtual hypervisor switch or the OpenFlow switches were responsible, the delay would have to remain positive at all times even if it varied greatly.

In order to understand these results better, we compare two 10 seconds long time series of samples measured with both controllers and the DAG card. Fig-

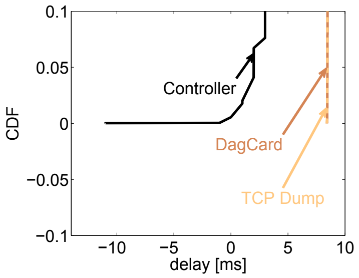


Figure 3.6: *Latency CDF Zoomed (Virtual Controller)*

ure 3.7a shows the results for the non-virtual controller. Three distinct lines of often occurring delays are clearly visible. One at about 8 ms for the DAG Card and two at 4 and respective 5 ms for the non-virtual controller. Additionally, there is a similar number of outliers for both measurement methods. For the virtual controller the samples shown in Figure 3.7b show a different behavior. While the samples of the DAG card remain similar at around 8 ms with outliers, the samples for the virtual controller show more frequent occurring delays at 1,2,3, and 6 ms. However, while the virtual controller appears to regularly measure a broader range of delays, the coefficient of variation for both controllers is next to identical at around 1.5, whereas the DAG card has a coefficient of variation of 1. The same is true for the mean and the median at 6.3 ms and 4-5 ms respectively. This tells us that even though the virtual controller appears to be more volatile and does have occasional time keeping issues, statistically those shortcomings carry no weight.

Based on these results, it appears feasible to obtain mean delay values using the purely SDN-based approach. However, for a production deployment, the switches again would have to improve their performance for applying the send-to-controller action to a packet. As our results have shown, this could be circumvented by giving the controller a secondary network interface and using a conventional output action. However, this can only serve as a temporary fix, if at all.

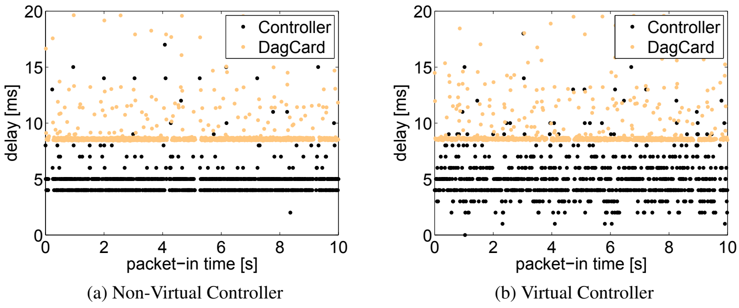
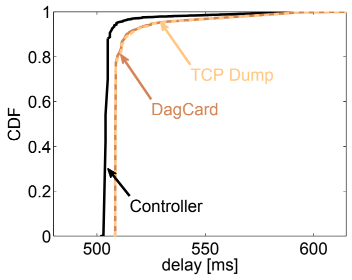


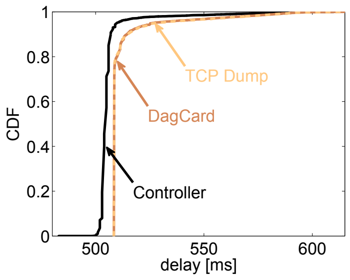
Figure 3.7: *Latency Samples*

Up to this point, we have not introduced any artificial delay into our measurements. Therefore, we set our network emulator between the two switches to impose a delay of 500 ms on the measurement probe to verify the accuracy of the measurements at a higher latency level. The CDFs of the measured delays are displayed in Figure 3.8. The results mirror those shown in Figure 3.4 for both the non-virtual and virtual controllers, albeit with an offset of the configured 500 ms delay. Therefore, we can conclude that the introduction of artificial delay has had no impact on the accuracy of the results as well as on the discrepancy between controller- and server-based measurements.

Now that we have established that the SDN-based measurement approach can indeed deliver delay measurement results statistically comparable to those of special purpose equipment in a stable environment, we take a look at what happens when the delay in the network changes. Therefore, we program a series of delay changes into our network emulator and observe whether the changes in delay are noticed on time by the controller and whether accuracy is impacted. Figure 3.9 shows a 60 seconds time series of a test run with five changes to different values



(a) Non-Virtual Controller



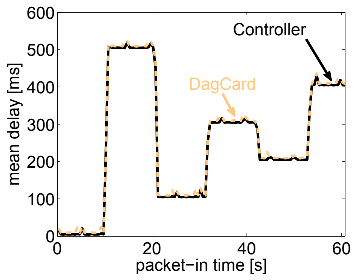
(b) Virtual Controller

Figure 3.8: *Latency Cumulative Distribution Functions (500 ms Delay)*

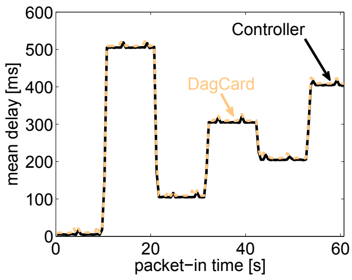
of delay. We observe that both controllers are able to closely mirror the delay present in the captured packet trace.

Sampling

As it is likely not very prudent to redirect all measurement traffic to the controller across the SDN control channel, which is also needed for network operation, the option of only redirecting a sample of packets to the controller seems viable. Therefore, we take a look at how closely the full reference delay value can be estimated using only a sample. Figure 3.10 shows the relative error for the mean delay in relation to the sampling ratio. The relative error has been obtained by repeatedly selecting random samples from the full reference measurement. It can be seen that in order to limit the relative error to 5%, the DAG card only requires about 5% of the packets, whereas virtual and non-virtual controller alike require about 10% of the sample due to the higher volatility of the measurement results. This means, that for a 95% accurate result using the SDN-based approach a sample twice the size of the e.g. the hybrid approach is required, which is a considerable overhead. This again emphasizes the importance of control channel bandwidth for the SDN-based approach.



(a) Non-Virtual Controller



(b) Virtual Controller

Figure 3.9: *Mean Latency (Variable Delays)*

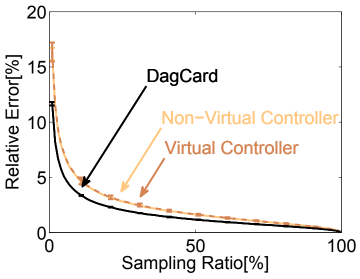


Figure 3.10: *Relative Error through Sampling*

3.3 Proof of Concept for Novel Approaches to Networking enabled by SDN

In this section, we introduce two novel SDN-based approaches to challenges in networking. The first novel SDN approach we introduce remedies the fact that the network structure of current experimental facilities is often determined by sites participating in the test bed. At each site, there are many nodes, typically connected by a simple switched network, and the connection between the sites is predefined by the topology of the connecting IP network. The advantages of this architecture is the low acquisition cost and the simple extensibility. The major drawback is that the topology is fixed and does not reflect the structures found in most large-scale networks and the Internet itself. In Section 3.3.1, we propose an *Interactive PrOxy Management* (IPOM) tool which enables us to define and emulate networks of arbitrary complexity on top of existing experimental facilities by means of OpenFlow and network emulating proxy nodes. This architecture does not require any physical changes to the experimental facility and switching between different network topology setups is performed within seconds.

The second SDN approach applies to the cloud business model of Infrastructure as a service (IaaS). IaaS is one of the prevalent business models in cloud computing and has generated much customer interest over the past few years. An IaaS provider offers the temporary deployment and maintenance of a custom virtual host and network infrastructure to its customers on which arbitrary applications can be run and/or hosted. Providers of such a service face several challenges in their data centers. One of the main issues is the inherent heterogeneity of systems and applications from different customers. As a result, a variety of different load and traffic patterns has to be handled by the same data center infrastructure. An IaaS provider has to find a good balance between the various customer application requirements and the efficient use of the available resources in the data center. The ECDC (Energy effiCient Data Center) approach we describe in Section 3.3.2 is such a smart mechanism for finding this balance. It leverages monitoring information from machines as well as network devices and environmental data to create a coherent view of the current situation in a data center.

3.3.1 Interactive Proxy Management in Future Communication Networks Using OpenFlow

In this section, we describe the IPOM tool, which allows us to leverage SDN as a means to create arbitrary logical topologies in fixed physical test beds. It is split into two parts, the topology editor for creating a network topology and the topology management tool for controlling the flows in the network.

IPOM Topology Editor

Before running experiments in a testbed environment, the physical network topology can be mapped using the IPOM topology editor. The GUI of this editor is shown in Figure 3.11. With the topology editor, nodes and switches can be added and the connection between them can be configured. When the physical topology is represented in the topology editor, it can be saved and loaded to the IPOM management tool to create virtual topologies and emulate multi-AS networks.

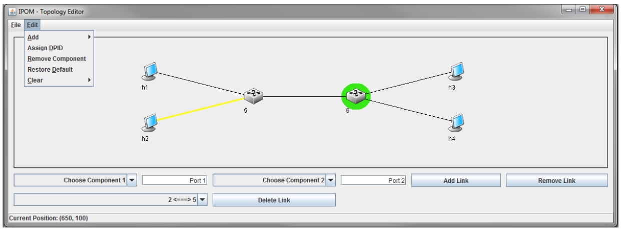
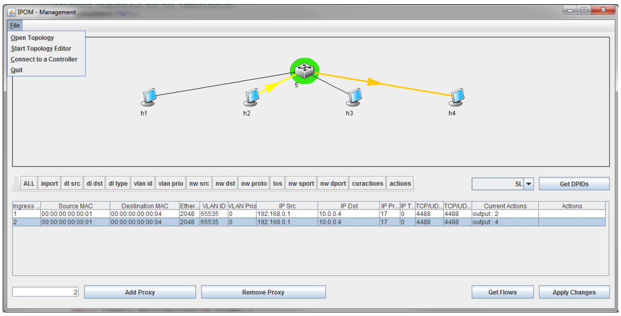


Figure 3.11: *IPOM topology editor.*

IPOM Management Tool

The management tool is the core functionality of IPOM and it provides the possibility to add and remove proxies. Figure 3.12 shows the GUI of the management tool. Besides the proxy management, arbitrary OpenFlow actions can be installed for any flows. This includes flow redirections and editing of the different fields such as MAC address and IP address. The direction of the flows can easily be identified as they are marked in the IPOM GUI. For the installation of rules on the OpenFlow switch, the BEACON controller [56] can be used together with IPOM. The communication between IPOM and the controller is thereby realized via a TCP connection. Beyond IPOM, consoles for each created virtual network node are provided either by emulating these nodes using mininet [57] or using physical hosts or KVM virtual machines provided by ToMaTo [58]. Thus, this tool provides an easy-to-use and flexible way of configuring network topologies and performing dynamic flow switching.



Ingress ...	Source MAC	Destination MAC	Ether...	VLAN ID	VLAN Prio	IP Src	IP Dst	IP Pr...	IP T...	TCP/UD...	TCP/UD...	Current Actions	Actions
1	00:00:00:00:00:01	00:00:00:00:00:04	2048	65535	0	192.168.0.1	10.0.0.4	17	0	4488	4488	output : 2	
2	00:00:00:00:00:01	00:00:00:00:00:04	2048	65535	0	192.168.0.1	10.0.0.4	17	0	4488	4488	output : 4	

Figure 3.12: *IPOM GUI with a graphical representation of the network.*

IPOM Proof of Concept

As a proof of concept for IPOM, we set up a network with four nodes and an OpenFlow switch using mininet, see Figure 3.13. This can however also be set up with a hardware OpenFlow switch. For each of the nodes, a console is shown which allows us to start arbitrary programs. A packet generator at client 1 is started which sends a single UDP packet flow to client 4. Starting with the functionality of a standard switch, the packets of this flow generated by client 1 are normally forwarded to client 4. Using IPOM, we are now able to modify the flow rules in such a way that clients 2 and 3 introduced as proxies. Thus, we are able to change the star topology into a bus topology. To verify the newly created topology, we modify the packets at client 2 and client 3. In addition, we show how to duplicate flows with IPOM in such a way that client 3 and client 4 will receive the packets generated by client 1. For instance, this option can used for network monitoring or data replication in data centers.

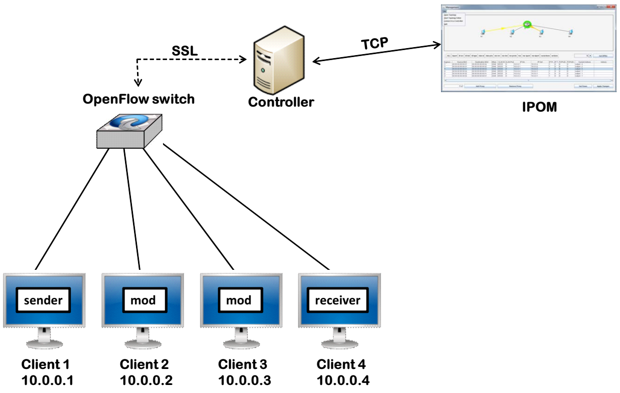


Figure 3.13: *IPOM Proof of Concept.*

3.3.2 ECDC: An OpenFlow-Based Energy-Efficient Data Center Approach

This section describes our approach to leverage SDN for energy-efficient data center operation (ECDC).

Architecture

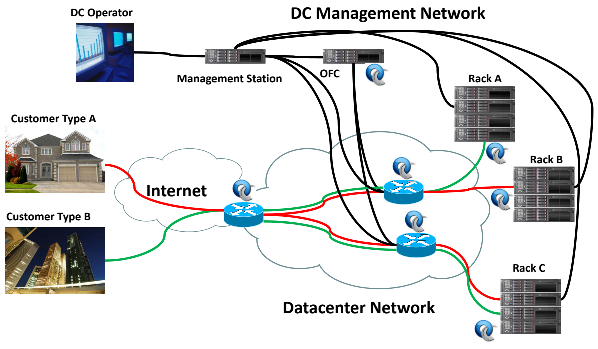


Figure 3.14: *ECDC Architecture*

Figure 3.14 shows the ECDC architecture in a simple data center scenario. On the right the servers hosting the customers' virtual infrastructures are depicted. They are organized in racks with a top of the rack switch each. The entry point into the network from the service side are the virtual switches integrated into the hypervisor of each server. On the left we see two types of customers that are connecting to the data center network from the customer side, i.e., the Internet. A type A customer is a private home user who wants to use an entertainment service, e.g., video streaming, hosted by a service provider as a virtual infrastructure in rack B and C in the data center. A type B customer is a business user who uses a business application set up by his company in the data center, e.g., a

virtual desktop infrastructure (VDI). As both types of customers have different demands and requirements, their traffic is kept in separate VLANs in the data center network to be able to manage them independently. The connectivity for type A customers is represented by red lines, for type B green lines are used. Packets are tagged at the entry switch into the data center network. The network itself is OpenFlow-enabled. Forwarding decisions for all network elements, i.e., access, edge, and hypervisor switches are handled by a central entity - the OpenFlow controller (OFC). The control connection for each network element is established via a physically isolated management network. This network also connects the controller, OpenFlow switches, physical servers, and environmental sensors to the central data center management entity - the management station.
The management station queries monitoring information on CPU-, network-, and memory-load, as well as power consumption from the connected devices via SNMP. Armed with this host of information, the management station generates the appropriate network policy for the OFC, distributes virtual machines across the servers, and powers down unused devices in order to ensure an efficient utilization of all resources while maintaining a good service quality for the customer at all times. The management station achieves this by observing a number of configured thresholds and timeouts for each service class. If a monitored parameter, e.g., CPU load, falls below or rises above a threshold for a certain amount of time as defined by a timeout, the management station will take action, e.g., by consolidating multiple virtual machines to one host or in the opposite case by spreading them over multiple hosts. Once such an action is triggered, the network, i.e., the OpenFlow controller, is immediately notified and can adapt the flow rules in the switches according to the new situation with little delay, minimizing the impact on the service. The gathered information is also presented to the data center operator through a graphical user interface as illustrated in Figure 3.15, which displays the current topology of the network and a time series of monitored values as configured by the operator. The operator is then able to facilitate changes in the operation of the data center, if this is necessary, e.g., through the introduction of a new service class.

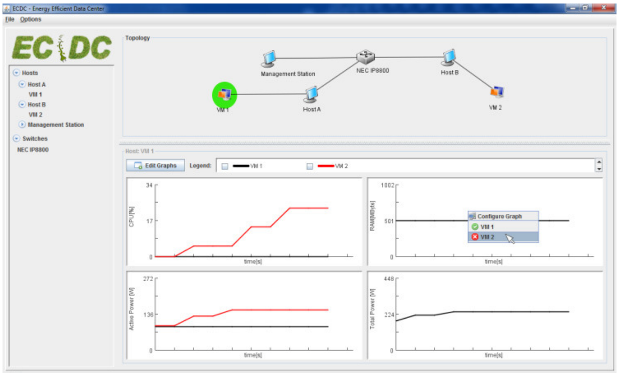


Figure 3.15: *ECDC Operator GUI*

Proof of Concept

The proof-of-concept testbed is hosted on the German-Lab [59] (G-Lab) facility in Wuerzburg, Germany. Four rack servers are used as computing nodes running OpenNebula [60] and KVM [61] as hypervisor using the Open vSwitch [42] as virtual switch. The management station is hosted on a fifth server running the OpenNebula management software as well as our Java-based data center management software. The management network is a legacy IP network realized by the Cisco top of rack switch of the G-Lab facility. As OpenFlow controller, we use BigSwitch's Floodlight hosted in a G-Lab virtual machine. The OpenFlow data center network is represented by a Pronto 3290 OpenFlow switch [62]. The proof of concept implementation shows the operation of the ECDC-enabled data center over the course of a business day. Using our own traffic generator, we emulate the behavior of the two types of users introduced in the previous section. We condensed the emulated "day" to a short cycle. During the progression of this cycle, we show the changes in the system as an operator would perceive them using our ECDC GUI as our software adapts the resource allocation according to the demand. In the topology section of the GUI (cf. Figure 3.15) topology changes caused by the migration of virtual machines as well as the powering up and down of physical hosts will be displayed. In the monitoring section the collected information reflecting load changes in servers is illustrated by line graphs. If an OpenFlow switch is selected in the topology display, the monitoring section changes to show the switch's flow table entries. By selecting an entry, the path of the corresponding flow through the network is displayed in the topology section.

3.4 Lessons Learned

In this chapter, we have compared the accuracy of purely SDN-based network measurements to that of a full reference packet capture trace using special purpose hardware. The results show that, while the accuracy for an individual delay does not reach the 7.5 ns accuracy of the reference measurement, the mean delay and especially mean bandwidth are on par with the capture card in the range of 1 ms. Therefore, if the mean value of the packet delay within this level of accuracy is sufficient input for the operation of a particular network, SDN-based measurements appear to be a viable and cost-efficient alternative. The inherent flexibility of SDN to mirror and redirect traffic on a per-flow basis greatly simplifies the measurement setup and in combination with virtualization can enable rapid deployments and tear downs on-demand. However, this is only possible if the SDN-enabled hardware is further developed to support this kind of function, sufficient control channel bandwidth is available, and the latency imposed by switch processing and control channel transmission is sufficiently even for both measurement points. Our results show that otherwise the measurements would become inaccurate and, more significantly, could disrupt the operation of the network. Our experience with the used OpenFlow 1.0 switches suggests that the improvement of SDN hardware is still a challenge. However, our results also indicate that the described SDN hybrid approach may serve as a working stepping stone towards pure SDN measurements.

Further investigating possible applications for SDN, we introduced two tools for network management that leverage SDN in two different scenarios. The first tool is IPOM, a topology configurator, that leverages the feature of OpenFlow switches to arbitrarily redirect flow at run time. Using this tool network researchers can easily create desired network topologies and to dynamically reroute traffic flows via proxies , e.g., to enable the modification of packets. An otherwise fixed network substrate in a test bed becomes a lot more useful by providing it with SDN functionality. Experiments can be conducted using a range of configurations while still running in a controlled environment yielding repeatable results. The usefulness of IPOM can be further enhanced by using it in conjunction with provisioning tools for virtual test environments such as ToMaTo.

The second tool we introduced is our smart data center management software ECDC. It allows for an integrated adaption of computing and network resources according to the required capacity to ensure a smooth operation of services in an IaaS scenario. At the same time the software aims to minimize the carbon footprint of the data center in question by consolidating capacities and powering down those not needed. To achieve this, we leverage the SDN approach implemented in the Open vSwitch in conjunction with the Floodlight controller as well as the proven open-source cloud management software OpenNebula. The automated approach we implemented as a proof of concept by providing a notification interface between the controllers shows that such an autonomous data center operation is possible. Together, the data center and network controllers can react to changing situations faster than they could individually and much faster than a human operator could. However, human interaction is still required in setting the appropriate thresholds that trigger an action either to save energy by shutting off devices and consolidating virtual machines or doing the opposite to improve the performance.

The main contribution of this chapter lies in showing how three common networking problems from different domains can be solved using SDN in a much simpler way than in conventional networking and that these solutions, especially in the case of network monitoringg, are indeed feasible in real-world scenarios.

4 Leveraging the SDN Northbound-API for QoE-based Application-Aware Networking

With the introduction of fast and reliable core networks and wide-spread availability of broadband internet access, a trend towards moving more and more services away from the end devices to remote data centers has established itself. This is the well known concept of Cloud Computing. While initially only services with few requirements towards the delivery network, e.g., email, were moved into the cloud, today a wide variety of much more complex applications and services is available to users remotely. This has resulted in significantly increased requirements on network QoS as users expect the same service standard from the remote service as they would have with a local setup. In many cases simple over-provisioning is no longer cost effective. Furthermore, the performance of a specific application cannot be determined by simply relying on QoS metrics [63]. Instead, a good application quality, e.g., the video quality or short waiting times, is the metric by which a user quantifies his or her Quality of Experience. Therefore, a major challenge for future networks is to dynamically adapt to QoE demands of the applications in the network. This is especially true for networks with limited resources, like today's access networks. Application-Aware Networking is a way to provide a good application quality to users of these networks.

The introduction of Software Defined Networking opens a path towards the realization of this approach. By introducing an external and programmable network control plane, SDN creates a flexible, adaptable, and open interface to the network, the "Northbound-API". It enables the exchange of application information with the network. This in turn can be leveraged to augment the network management to improve the user QoE. The challenge here is to determine which kind of information should be how often exchanged.

In this chapter, we investigate how we can obtain and leverage application information in the context of the SDN Northbound-API. We begin by analyzing the subjective user quality of a resource-intensive cloud application, i.e. cloud gaming, in Section 4.2. We abstract performance indicators, which can be used as input for an SDN controller. We then proceed by investigating how effectively an SDN controller can use different types of information to improve or maintain the QoE of an application in a multi-path scenario in Section 4.3.
This chapter is mainly based on and taken from [2], [7], and [3].

4.1 Background and Related Work

Currently, the prevalent idea in networking for improving the quality of a service for the end-user is to differentiate traffic flows using QoS levels. For this purpose, different QoS classes are defined according to the expected type of traffic in the network and applications with similar needs are assigned to them. These classes ensure a minimum reserved traffic rate according to the QoS parameters of the application type.

However, a QoS-based provisioning alone is often not sufficient to provide an acceptable application quality. This is especially the case for applications with time-dynamic QoS requirements. For example, according to video encoding, download patterns, or user behavior an application may not have a fixed demand for bandwidth. Instead, bandwidth is required depending on the application state. SDN provides an interface to convey this application state to the network. This allows the network control plane to optimize the flow of traffic according to the information available.

4.1.1 Background and Works on Application-Aware SDN

In an SDN-enabled world, new open interfaces exist between the application, the data-plane, and the control-plane. The interface between data- and control-plane is called the ”Southbound-API”. It enables the externalization of the control plane from the forwarding device to a logically-centralized network control plane, often simply called ”controller”. As a software entity, the controller can be freely programmed and adapted to the network according to the operator’s requirements. Currently, the most popular realization of this interface is OpenFlow [11], which we use for our experiments.

While the Southbound-API is an important component of SDN, from our point of view, the significant additional value of SDN lies within the ”Northbound-API” interface between the network control plane and what we call ”application control plane”, i.e. applications running on top of or interacting with the network itself. This enables the exchange of information about the application and network state, respectively. Curtis et al. [64] suggest an optimized data center flow scheduling by notifying an OpenFlow-like controller about elephant flows detected at the hosts’ socket buffers. In [65] Das et al. demonstrate how SDN-based aggregate routing can be adapted with the QoS parameters of applications in mind. The author [9] shows how a pre-notification of the network control plane in case of a virtual machine migration can serve to maintain service. We are going one step further by also taking the actual application quality and state over time into account to maintain a good service quality for customers.

Technical Details on YouTube Streaming

YouTube, one of the most important VoD platforms, provides mainly small to medium sized video clips in different qualities. The default video compression format is H.264/MPEG-4 Advanced Video Coding (AVC). To watch a video, the user opens the YouTube web page where an HTML-5 or Adobe Flash player is embedded for video playback. The video player requests the video data from a YouTube streaming server in the Internet using the HTTP protocol. YouTube uses progressive video streaming which means that the video is already played out, while the client downloads the content into a buffer or a temporary file in the background. If the buffer is sufficiently filled, a smooth video playback can be guaranteed. If the buffer is empty, the video playback is interrupted and stalling occurs. According to [66, 67], stalling is the dominating factor of the QoE for online video streaming, clearly exceeding the significance of video resolution. Hence, a simple mapping of a QoS parameter such as throughput to YouTube QoE is difficult, as the QoE depends on the buffer level and video encoding. This complexity makes YouTube streaming a good candidate for the Application-Aware SDN approach.

4.1.2 Works on QoE in Inter-active Video Applications

Nave et al. [68] describe an architecture for cloud gaming as developed in the European FP6 Integrated Project Games@Large. Pigora et al. [69] discuss the benefits of applying a cloud gaming approach to training and education by introducing their solution called 'Nexus Web' as an example. They describe implementation challenges and give a rough estimate of the QoS. Quality of Experience, however, is not mentioned. We overcame the implementation challenges by procuring special purpose hardware. Chan [70] simulates the impact of a wireless environment on cloud gaming using Opnet and draws conclusions regarding the QoS and its scalability. Additionally, Chan found that a moving user will experience a significant drop in the QoS. However, he also does not discuss user-based QoE. Chang et al. [71] propose a methodology for quantifying the performance of several VDI solutions in a gaming scenario. To this end they use a classic 2d game and capture the graphics output at server and client for a comparison of quality. However, they do not incorporate actual user feedback and current 3d video games.
Szigeti et al. [72] recommended guidelines to setting QoS parameters for interactive video or video conferencing traffic. As interactive video is related to cloud gaming, we have taken their recommendations into account when designing our

tests. However, cloud gaming has different QoE characteristics from interactive video and therefore these values do not exactly apply.
Three classes of games with different behavior towards QoE are identified by Claypool et al. [73]. These are “Omnipresent” (e.g. real-time strategy games), “Third-Person Avatar” (e.g. role-play games), and “First Person Avatar” (e.g. First Person Shooters). We adopt these classes and chose one game from each for the purpose of our tests in order to account for the effects of varying content. Additionally, Claypool et al. [73] also give a latency range in which each type of game performs well. We have based the choice of latency values for our tests based on these results.

4.2 Obtaining Key Performance Indicators on the Example of Cloud Gaming

Recently, a new type of cloud service has been introduced, which combines internet video and online gaming and may have the most stringent demands on network QoS to date: cloud gaming. This new service has been subject of a case study by Ojala et al. [74] underlining its potential from a business point of view. Yet, business is not the only field from which cloud gaming has received attention. As early as 2009, Ross [75] identified gaming as the ”Killer-App” for cloud computing and Chang [76] even believes that ”gaming will save us all”.
The service essentially moves the processing power required to render a game away from the user into a data center and streams the entire game experience to the user as a high definition video. Traditionally, only multi-player games required network connectivity. For each player a game client is connected remotely to a server, which hosts and controls the game environment, receives input commands and sends out status updates. The amount of data exchanged is usually quite small as the user experience is created at the client device. However, in cloud gaming the entire user experience is generated on the server and has to be delivered through the network. This is where cloud gaming is significantly differ-

ent from conventional online gaming in terms of network QoE. While in conventional online gaming the user experience is generated at the client and therefore the network does not have any influence on the presentation, it may greatly affect the quality in Cloud Gaming.
From a network point of view there are several challenges to overcome to operate such a service in the quality expected by the users. Unlike conventional video streaming or web applications Cloud Gaming does not require either a relatively high constant down-link bandwidth or low latency, but both. We determine these parameters based on actual user perceptions to identify key influence factors for QoE in cloud gaming, which in turn determine which information has to be exchanged between an SDN controller and the application to ensure a good service quality. To achieve this goal, subjective user surveys are required. Therefore, we have designed a local testbed at the University of Würzburg that emulates a cloud gaming service. This testbed is used to provide a test person with a game experience similar to that of a cloud service. We have developed a series of tests to gauge a user's reactions to varying settings of propagation delay and packet loss. Based on this setup we performed a survey with test persons and derive general conclusions on the impact of certain QoS parameters on QoE and identify influences of content and perception from the results.

4.2.1 Survey Parameters and Design

In Subsection 4.2.1 we select the range of the QoS parameters loss and delay whose influence on QoE is tested in our user tests. In Subsection 4.2.2 we discuss the attributes of the test group our survey is based on. Our testbed is explained in subsection 4.2.1. Finally, we characterize the actual survey process in Subsection 4.2.1.

QoS Parameters of the Survey

An IP network connection maybe influenced by numerous factors: delay, jitter, packet loss, packet re-ordering or packet duplication to mention only a few. How-

ever, to Cloud Gaming in its current form only two parameters are relevant for the QoE - packet delay and loss. Delay affects the time a user's action is executed and the results are perceived. In Cloud Gaming this would be the time from the pressing of a controller button to the intended action. All other influence factors result in the application not being able to display a video frame or execute an input command in time. These effects are handled by the network encoding or treated by the application identical to packet loss. To meet the real-time constraint the software cannot wait for one packet to be delivered for an arbitrary amount of time or in an arbitrary order. As a consequence the program will have no choice, but to drop the data resulting in loss. From the user's point of view, lost or late packets lead to the same quality degradation independent from the underlying cause e.g. network congestion or jitter. Therefore, all of these effects can be investigated by just examining the influence of packet loss.
Pantel et al. propose in [77] that a delay greater than 100 ms should be avoided based on study of two racing games. We take this value as a starting point for designing our own initial subjective tests. The next QoS parameter we consider in our tests is loss. Since there is no reference value for loss in Cloud Gaming, we take a look at [72] by Szigeti et al., which gives guidelines for the related field of video conferencing. It states that loss should be no more than 1 percent, one-way latency should be no more than 150 ms, and jitter should be no more than 30 ms. In [78] by Henderson et al. the authors describe the effect that degraded QoS can dissuade players from joining a networked game, but those already connected to a server are more tolerant towards bad QoS. We consider this effect in relation to Cloud Gaming, but it affects only the usage of the service, i.e. users might quit the service or not subscribe to it. In this chapter, we focus on influences occurring while using the service.

As mentioned before, we investigate the three classes of games defined by Claypool et al. [73]. Table 4.1 gives an overview of the specific scenarios we define. The table gives the scenario id as well as the specific settings for delay and loss. Finally, it also gives the direction - client to server or server to client - to which the parameters are applied. The first scenario (B) we introduce is the

Scenario ID	Delay	Packet Loss	Direction
B	0 ms	0.0%	both
D1	80 ms	0.0%	both
D2	200 ms	0.0%	both
D3	300 ms	0.0%	both
L1	0 ms	0.3%	both
L2	0 ms	1.0%	both
M1	40 ms	1.5%	both
M2	180 ms	0.3%	both
A1	120 ms	1.0%	client to server
A2	120 ms	1.0%	server to client

Table 4.1: *Test Scenarios and Applied Parameters*

baseline, which is essentially a setting in which all parameters are set to zero. We do so in order to check for the placebo effect, i.e. some of the test subjects could imagine a distortion where there is actually none, simply because they find themselves in a test situation. Additionally, we define three delay-only scenarios (D1-3). These are our subjective perception threshold for delay at 160 ms round-trip time (RTT), a noticeable disturbance of play at 400 ms RTT and 600 ms RTT where players should no longer be able to play. Here the delay is identical on up- and down-link. This results in the input commands being received late by the game service and the feedback video being delayed also.
Having considered delay, we then introduce two scenarios with symmetric packet loss of 0.3 and 1 percent per link (scenarios L1,L2) being the only source of disturbance. The effect of packet loss on the down-link are a notable fragmentation of the video as well as lost keystrokes on the up-link. After looking into delay and packet loss individually we are interested in the question, which parameter is dominant and has a larger influence on the QoE. To determine this, we create two mixed scenarios combining delay and packet loss (scenarios M1,M2). Finally, we introduce two scenarios with asymmetric settings to investigate whether applying the same parameters on either the up or the down-link changes the outcome of

the QoE perception (scenarios A1,A2).

Emulation of Cloud Gaming

Figure 4.1 depicts our testbed setup from a logical point of view. The idea of this setup is to replicate the basic infrastructure of OnLive and its competitors intend to use to deliver the game experience to their customers. Hence, three individual components have to be reproduced. The hardware shown on the right hand side of the Figure replaces the data centers. To replace the servers which would usually render the game we use a conventional PlayStation 3 gaming console. This device is optimized for gaming and the games running on it are optimized for its hardware. Therefore, the risk of false results caused by erratic behavior of the rendering hardware is minimal. The images created by the Playstation are then streamed to the client via a special purpose hardware, called Spawn Box. The Spawn HD-720 is capable of streaming the output produced by many modern consoles over an IP network to its client software (Spawn Player). This software is a modified version of the well-known VLC media player. It displays the video and transmits the client input to the Spawn box, which in turn relays it to the game console. The Spawn Player is configured for smooth replay at the best possible quality i.e. a video resolution at three quarters of 720p and a video codec bit-rate at 3 MBit/s. The box uses HaiVision's MAKO-HD hardware, which was originally designed for the purpose of high definition video conferencing and hence uses progressive H.264 video encoding. Both video and user input are transmitted through the network via a RTP/UDP connection.

In the center of Figure 4.1 the component emulating an IP WAN, e.g. the Internet, is represented by a cloud. In fact this is a computer running the Linux-based network emulator NetEM on Debian Lenny. The NetEM software is capable of producing a variety of effects a wide area network could have on a packet stream. However, we only use it to introduce fixed delay as well as random loss as explained in 4.2.1. A client is represented by an Intel Pentium IV personal computer in our experiment running the Spawn Player software on Windows XP as seen on the left of the Figure.

For the purposes of conducting the survey, we introduce a fourth component. A web-server that controls the simulation by remotely configuring the WAN-simulator and displaying the front-end of the QoE poll as well as storing its results.

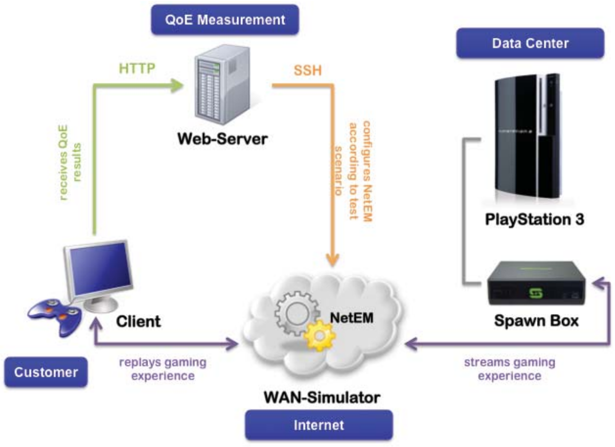


Figure 4.1: *Logical View of the Testbed Setup*

Survey Process

The test participant is asked to use the client PC. The client PC is equipped with two monitors, that serve two different purposes. While on the first monitor the researcher conducts the opinion poll and could control the test, the subject is to play the game on the second display. First we create a unique identifier for each participant and store his age. Next the player can pick one of our three games according to the three classes defined in [73]. We chose Pro Evolution Soccer for the omnipresent perspective (slow-pace game-play), Final Fantasy XIII for the 3rd person perspective (medium-paced game-play) and Gran Turismo HD Concept for the 1st person perspective (fast-paced game-play) (cf. 4.2.1). The participants are allowed to repeat the test using another game. Subsequently we interview the test person on whether or not they favor games of that particular genre in order to determine if the test participant is potentially biased by their preference. We then ask the participant to estimate his/her skill in gaming as explained in Subsection 4.2.2.

Following these initial questions the subject is allowed to explore the game and its controls in 10 minutes of free play time. During this period the game is intentionally not affected by any distortions, so that the player can use this experience as a reference point (perfect experience) to the scenarios introduced in the testing phase. Every test subject is supposed to experience every scenario we introduce exactly once during the test. To avoid biased results caused by a specific sequence of scenarios, we decided this sequence to be randomly generated with the exception of always starting with the baseline.
Each scenario lasts for about 1 minute. At the end of a scenario the researcher asks the participant for his current game experience, i.e. the quality of experience perceived by the player. This rating is expressed by the so called Mean Opinion Score (MOS) [79] for perceived quality of experience. Each experience was mapped to a value ranging from 1 to 5 with increasing values implying increasing quality ranging from bad to excellent. We left it to our participants to decide, which aspect of their experience image quality or responsiveness they weighted the most in their rating, since we intend to express the entire game experience by this value. With all ten scenarios being completed we then ask the test participant whether or not they are willing to pay a monthly fee for the overall experience they just made on the understanding that they can play any game they wanted to. We do this in order to get an overall impression of how the tests are perceived. Finally, we informally interview our participants on their general attitude towards the idea of Cloud Gaming and the potential they attribute to the concept.

4.2.2 Rater Reliability

In this section we have a look at the demographics of our survey participants and subsequently determine the reliability of their ratings.

Demographics of the Test User Group

A study performed for Electronic Arts [80] in 2005 polled 3000 people in Germany aged 14 and above for the purposes of in-game advertising. It argues that only 5% of all gamers actually play often and are so called "intense gamers". By contrast the major percentage encompasses two groups: 24% are what Electronic Arts calls "casual gamers" and 54% of the interviewees are considered to be "leisure gamers". The study implies that most gamers and therefore most potential users of cloud gaming in Germany play on an occasional basis. Hence, the sample we took was aimed at getting a representative share of the target population defined by playing on a regular or occasional rather than an intense basis. Our sample is made up of 58 participants. Participants were often unsure whether they played on a regular or occasional basis. Therefore, we changed the question and asked the participants how they perceived their skill at gaming, which seems to be a less vague indicator. 15.2% of the participants consider themselves to be skilled gamers, while 44.6% think that their gaming skill is "medium", and 39.2% even judge themselves as "low". These percentages can be mapped to the groups of "casual gamers" and "leisure gamers". We can conclude that most of our test subjects do not play on an intense basis and thus our sample should lie within the target audience of Cloud Gaming.

Rater Reliability and Diversity

In order to determine the reliability of our rater, we use two measures - intra- and inter-rater reliability as described by Hoßfeld et al. [67]. Intra-rater reliability determines the consistency of ratings made by one single individual. We use the Spearman rank correlation coefficient to quantify both measures. This coefficient determines whether the relationship between two variables can be described with a monotone function. Here this would be the ranking given by the person and the value of the network parameter in question. Ideally, the ranking should change proportionally with the value of the network parameter resulting in different results for each setting. No repetition of values in the ranking would result in a Spearman rank correlation coefficient with an absolute value of one. In Figure 4.2 a CDF for the intra-rater reliability of our users is given. We consider users with a Spearman rank correlation of greater than an absolute value of 0.60 to be reliable and thus consistent in their ratings. The Figure shows that roughly 80% of our users fall into this category.

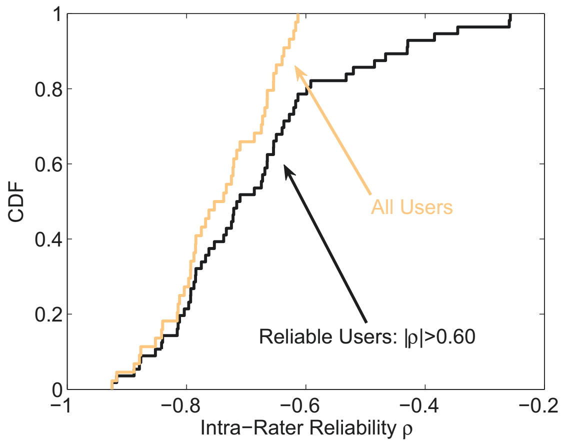


Figure 4.2: *Intra-Rater Reliability*

Inter-rater reliability on the other hand describes the degree of agreement between multiple users given the same test. Figure 4.3 gives the inter-rater reliability of our users in different scenarios. For Scenarios in which only packet loss is applied, we see a high absolute value of the Spearman rank correlation between users between about 0.75 and 0.9, which clearly indicates loss as an important factor. However, in delay-only scenarios the picture is not so clear. Especially for the fast-paced game the ratings are very diverse. Based on the parameter weights in Table 4.2.4 the overall inter-rater reliability lies just around our cut-off point of 0.6. This indicates a significant difference in perception dependent on the user. This is underlined by the standard deviation of the opinion score (SOS) for reliable and overall users as shown in Figure 4.4 and a relatively high SOS parameter a of about 0.3-0.35. This shows that the assessment of QoE in cloud computing is not trivial as even with a larger number of test participants this deviation will not be significantly lower.

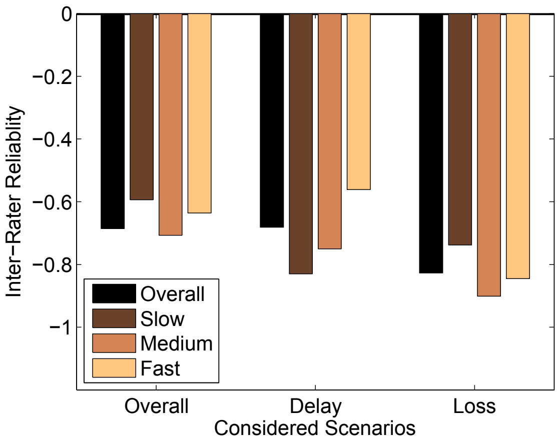


Figure 4.3: *Inter-Rater Reliability*

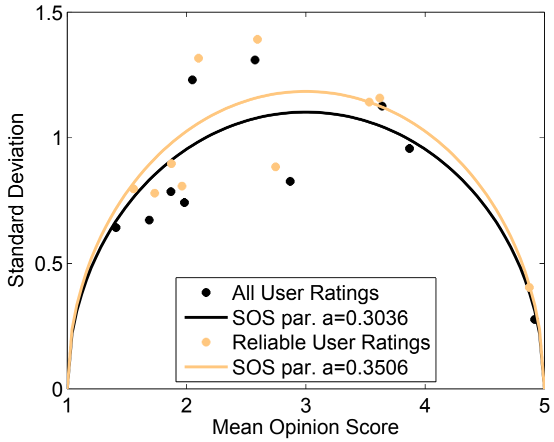


Figure 4.4: *SOS*

4.2.3 Identification of Key Influence Factors for Cloud Gaming QoE

Figure 4.5 illustrates the surveyed MOS value for each game in each of our scenarios. The plot is based on the data of 79 test runs, respectively 790 user votes. The y-axis indicates the MOS for a particular scenario, denoted by its scenario ID on the x-axis. At first glance it is apparent that the MOS values of each scenario differ from game to game. This variation is most remarkable in the bi-directional delay scenarios (D1-3). It seems the slower the game-play gets the better the ratings become. For instance, scenario D2 is rated at 1.2143 MOS (bad) in combination with the racing simulation (fast), while it is rated at a value of 2.2308 MOS (poor) using the role play game (medium) and with the soccer simulation (slow) even scores a MOS value of 2.96 (fair). We therefore suspect that faster games are more delay-sensitive than slower ones. This agrees with the classification of Claypool et al.. It is reasonable that the influence of delay on Cloud

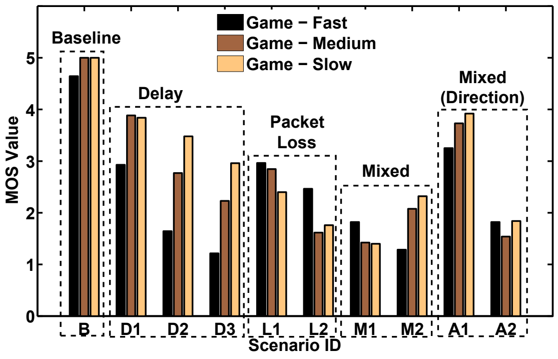


Figure 4.5: *MOS Ratings per Scenario/Game*

Gaming is similar to its influence on conventional games.

Impact of Symmetric Delay and Loss on QoE

Figure 4.6 illustrates the measured MOS values for the bi-directional delay scenarios. The delay values are shown on the x-axis and the y-axis gives the corresponding MOS values. The values for the x-ticks are taken from scenarios B and D1-3. The results for each game are plotted as two graphs - one for all raters and one for reliable raters only. Confidence intervals are given for each MOS value. The intervals are small, hence we can conclude, that the MOS values are stable and enough ratings were collected. The difference between all users and the reliable group appears to be marginal. We observe that all graphs decrease with increasing delay. As suspected, there is a decline of MOS values with increasing delay. Furthermore, the plot confirms that the racing simulation appears to be most delay-sensitive for its graph runs below the others. Up to a delay of 80 ms the user experience has the same quality for role play game (medium) and soccer simulation (slow). The delay value of 80 ms was chosen to lie in the area

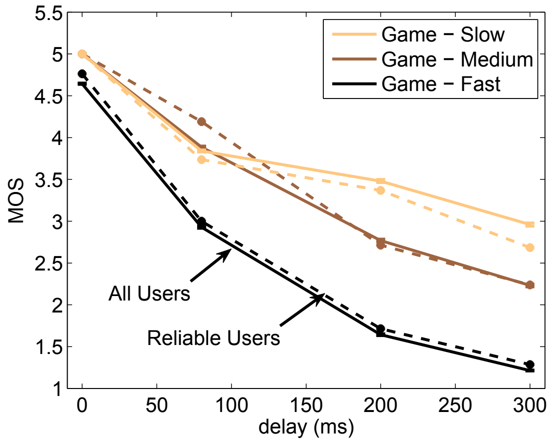


Figure 4.6: *MOS for Scenarios with Applied Delay*

of threshold where players start to notice the delay. While the delay is recognized in the racing simulation and rated with a MOS value of 3, only some people detected it in the role play game and the soccer game resulting in a MOS value of 4 for both. At a delay of 200 ms, however, the graph of the soccer game is clearly above the role play game graph which allows us to draw the conclusion that indeed the slower the game is, the less delay influences the user rating.

Figure 4.7 visualizes the surveyed MOS values for packet loss. MOS values are shown on the y-axis, while the values for packet loss are on the x-axis. We used the packet loss values of scenarios B,L1 and L2 for the x-ticks. Again, the results for each game are plotted as two graphs - one for all raters and one for reliable raters only. Also, the confidence intervals are for each MOS value are again small, indicated a stable value. Same as before, the gap between all users and the reliable group is very small. It becomes obvious that all graphs drop with increasing packet loss. Consequently, we also conclude that there is a decay in

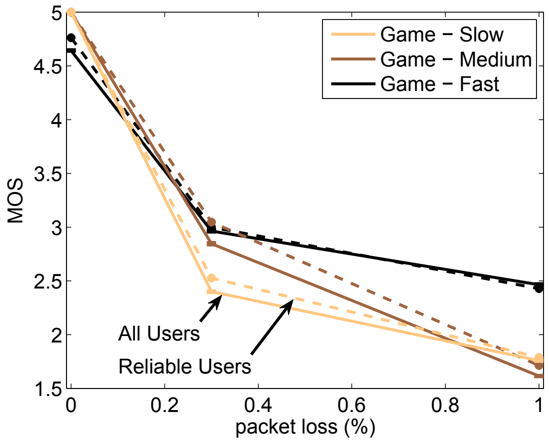


Figure 4.7: *MOS for Scenarios with Applied Packet Loss*

MOS with increasing packet loss. We assume this is due to the fact, that with increasing packet loss the video quality degrades more and more. We note that in essence the racing simulation, the most upper graph, appears to be most resilient towards packet loss. This might be a result of the circumstance that in fast paced games the player never really focuses on his environment as it is changing rapidly and thus degraded video quality becomes less important. Furthermore, fast paced games have a much higher command input rate than slower games. Here a lost keystroke is often subconsciously repeated. These facts seem to confirm our assumption.

User Perception of Delay vs Loss

In Figure 4.8 we used a two-dimensional surface-plot to identify a user tendency on what is perceived worse for each game: packet loss or delay. On the x-axis the reader can observe the MOS values of scenario M1, while on the y-axis the MOS

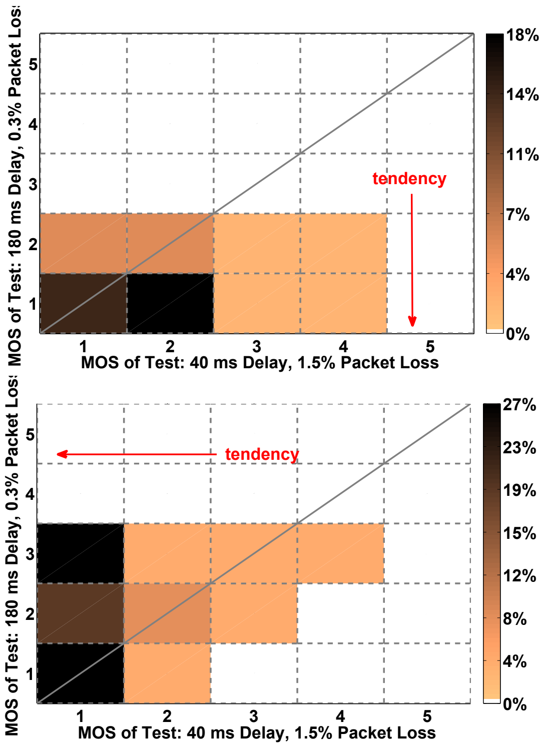


Figure 4.8: *Mixed Scenarios*

values of scenario M2 are denoted. Each point displayed as a square represents the rating for both scenarios. The z-axis, i.e. the color of a square indicates the frequency of a rating combination. The darker a square is, the more participants voted for this combination of MOS scores. For instance, the black square in the upper left plot (game - fast) at the coordinates (2,1) implies that 36% of all users that judged the racing simulation rated scenario M1 with a MOS value of 2 and scenario M2 with a MOS value of 1.
Additionally we delineated the angle bisector in each plot. Squares that are located left or above this line indicate a preference towards scenario M2, while squares that are located right or below the bisector indicate a favor for scenario M1. Squares that lie exactly on the angle bisector express neutrality, i.e. the MOS value for scenario M1 equals that given to scenario M2. In Figure 4.8 we observe that about 50% of all people that rated the racing simulation considered scenario M1 and M2 equally bad. The remaining 50%, however, show a clear tendency towards scenario M1. This further reinforces the assumption made when looking at loss only, that fast games seem to be more tolerant towards loss than others. Furthermore, we see that the delay-intensive test is perceived worse. This fits with our results so far. Delay appears to be the decisive factor in fast paced games. Players of fast games would rather accept higher packet loss rates than they would tolerate high delays, for the game-play and the players' success in the game very strong depend on their ability to react swiftly.
The plot for the medium-paced game (rpg) shows quite an opposite trend. Here most of the participants lean towards scenario M2. In the role play game over 50% of the users prefer the delay-intensive scenario over the loss-intensive, while about 40% remain neutral. This game therefore appears delay-resilient, but loss-intolerant. Players of medium-paced games would prefer high delay over high packet loss rates, since they are more interested in what they see (i.e. video quality) than in responsiveness. The reason for this is two-fold. On the one hand responsiveness is not that decisive for the game-play and the players' success in the game. On the other hand the ability to immerse in the simulated world is far more important in games like this. For the slow-paced game we could not derive

any clear tendency. We observe a content-dependency and as we have seen, the question which parameter is perceived worse cannot be answered globally.

Evaluation of Asymmetric Network Conditions on QoE

Finally we have a look at the results of the asymmetric scenarios A1 (client to server connection disturbed) and A2 (server to client connection disturbed). The results for each of these scenarios contrast each other, although they use the same parameters albeit in different directions.

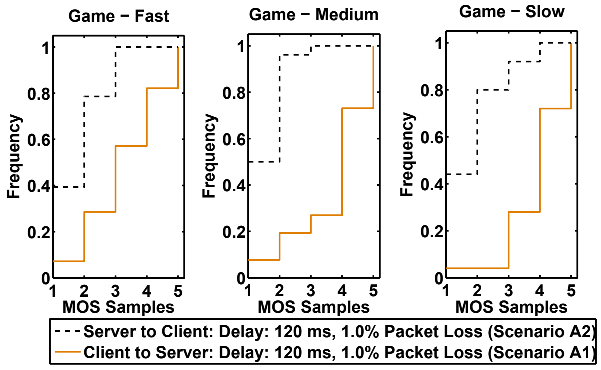


Figure 4.9: *MOS for Scenarios with Asymmetric Disruptions*

The MOS value of scenario A1 was more than twice as high as the MOS value given to scenario A2. Figure 4.9 shows the cumulative distribution functions (CDF) using the MOS value as random variable. Since MOS values are discrete, we see a stair-plot. Each plot displays one game. The first observation we make is that in all games the graph of scenario A2 slopes upwards much faster than the graph of scenario A1. This indicates that generally more test participants disliked the distortion of the server to client connection. For instance, while in the fast-paced game less than 30% of the test participants rated scenario A1 with

a MOS value 2, almost 80% rated scenario A2 the same. Hardly anybody rated scenario A2 better than a MOS value of 3, except for the slow-paced game where less than 10% gave a MOS value of 4. The explanation for this tendency is quite obvious: Server-to-client loss of 1% results in massive video distortions, while client-to-server loss of 1% remains virtually 'invisible'. Very few of the test persons ever knowingly complained about control inputs being dropped. However, a packet loss of 1% can very well compromise more than 20% of the picture in the video stream. The graph of the role play game increases the fastest. Over 90% rated it with $MOS \leq 2$. Again, this is linked to the way people experience the game. Role play gamers want to immerse into the world of game, therefore video distortions of this magnitude can hardly be tolerated as they greatly decrease the visual experience.
Comparing the client-to-server graph of each game, we observe that it continuously bottoms out the slower the game gets. This means the less participants rate client-to-server distortions as bad the slower the game becomes. Although not all people consciously recognized the dropping of control inputs, it had an impact on their rating. If a soccer player will not pass the ball immediately, the test subject will simply press the button again as these games often do have an inherent delay to a players action. If a vehicle in a racing game will not turn immediately, however, it might be too late and the player might crash into a wall. We come to understand that server-to-client packet loss due to video distortion is far more critical for many Cloud Gaming applications than client-to-server packet loss, which might not even be knowingly perceived in a great deal of cases. Client-to-server packet loss only becomes grave, if a missed input potentially results in the player using the game. The delay of 120 ms was hardly recognized, no matter in which direction.

4.2.4 Towards a Key Quality Indicator

So far we have derived several qualitative influences that different parameters have on the QoE of a cloud gaming application. However, for a service provider

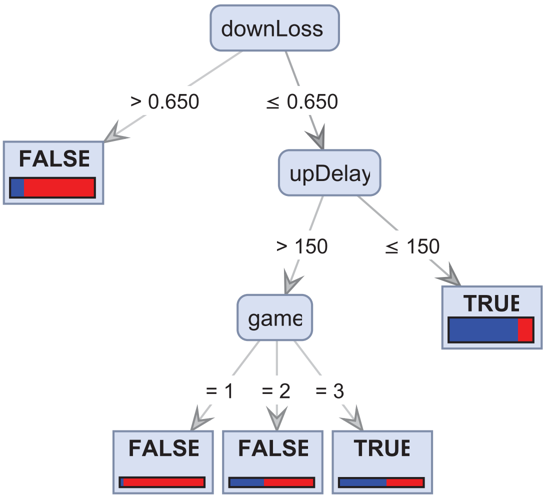


Figure 4.10: *Decision Tree of QoE Impact Factors*

it is also important to know, how significant the influence of a certain parameter is compared to others. This way the service provider can structure the service in such a way as to ensure a minimum level of QoE at all times. We have used the standard data mining and statistics tool Rapidminer [81] to derive the importance of the parameters in our survey.

Table 4.2.4 lists the parameters and their assigned weights based on the information gain calculated by the tool for samples yielding a fair quality, i.e. a MOS value of three and above. It identifies downstream packet loss as the most important parameter for QoE in cloud gaming in our survey with a maximum weight of 1, followed by downstream delay, which is already significantly less important with weight of 0.583. This shows, that the downstream transferring the video has a statistically higher impact on QoE than the upstream with the upstream packet loss and delay at weights of 0.370 and 0.212 respectively. However, both upstream parameters still have a significant weight, while it appears that the influence of game type, player skill, etc. is negligible.

Additionally we used the WEKA [82] implementation of the REPTree algorithm in RapidMiner to construct a decision tree. This method tries to construct a subset of specific decision rules from a general rule covering the entire data set, i.e. the test results, by recursively splitting it based on information gain. The rule at the root of this tree, i.e. the first split, signifies the most important parameter for the decision. The decision we want to make here is whether the game quality is acceptable, i.e. true, or bad, i.e. false. The resulting tree based on our test results is illustrated in Figure 4.10. Our tree has the downstream packet loss as the most significant parameter at its root. A loss value of greater than 65% will result in a bad experience. If this is not the case, the upstream delay becomes the next significant influence factor. Here a delay of less than 150 ms will result in an at least acceptable experience. However, if the delay is higher, the game type becomes the next decisive factor. Under these circumstances only the slow game can yield acceptable results. This again suggests a difference in perception for up- and downstream impact factors as seen in the previous section.

Parameter	Weight
Downstream Packet Loss	1.0
Downstream Delay	0.583
Upstream Packet Loss	0.370
Upstream Delay	0.212
Type of Game	0.067
Player Skill	0.006
Player Attitude Towards Game	0.006
Player Age	0.0

Table 4.2: *Weight of Parameters Based on Information Gain*

4.3 SDN-based Application-Aware Networking

After having presented a way to determine the impact of QoS parameters on the application QoE for cloud gaming, we proceed by examining how these application quality parameters as well as different kinds of other information, such as per-flow parameters, or application signatures, can support a more effective network management in an SDN-enabled network. Application information and related QoS levels offer greater flexibility in terms of supporting QoE than hard QoS parameters. However, using them may require an overhead of signaling effort compared to management at the network level. Therefore, we take a look at the trade-off between the QoE improvement due to more detailed application information and corresponding signaling overhead. Unfortunately, this level of detailed application information was not available to us for our use case of cloud gaming. Therefore, all approaches are emulated in an SDN-enabled testbed for the application of YouTube streaming as the same type of information is not easily obtainable for the mostly proprietary cloud gaming systems. We use the YouTube quality monitoring tool YoMo [83], which monitors the buffer filling level and the occurrence of playback stalling to quantify the impact each approach has on the YouTube QoE.

4.3.1 Scenario and Testbed Setup

Our Application-Aware SDN testbed emulates a path selection scenario for an access network provider. The access network provider, e.g. a mobile network operator, transmits the data of its customers over multiple leased lines to the Internet. The goal of the provider is to use these lines as efficiently as possible, i.e., as few lines as possible should be rented as long as the QoE of the user does not suffer.

The provider has chosen an OpenFlow-enabled device as termination point for several access connections to its customers. While the provider does have exclusive last mile-access to the customers, the upstream connectivity belongs to a different ISP. Therefore, the OpenFlow device is connected via leased virtual channels across the WAN to a second OpenFlow-enabled device in the provider's Internet backbone. A customer of the provider is watching a YouTube video, while other customers run file downloads or surf the web. The provider is interested in providing a good quality of experience to the YouTube user, while at the same time not overextending its leased resources.

Testbed Setup

Figure 4.11 shows our testbed setup for the reference case. As OpenFlow-enabled devices at the access and provider edge, we use two Pronto 3290 switches [62] running PicOS 1.6.1. Both are configured for out-band management and are connected via their management interfaces to an HP ProCurve 1810-24 switch, forming the management network. A Dell PowerEdge 860 server is used as controller host and is also connected to the HP switch. As controller software, we are using the Floodlight controller [44] from BigSwitch running our own modules. The ”virtual” provider connections are represented by five links between the two switches using Cat-5 cabling. The physical ports on the switches for these five links are set to 10 Mbps link speed. The ”provider switch” is connected to a Cisco router, which serves as Internet gateway for our testbed. The YouTube user is represented by a standard PC running Ubuntu Linux. The browser used to access YouTube is Mozilla Firefox running our YoMo plugin. When the browser is directed to play a YouTube video, YoMo, among other things, is able to identify the TCP-flows used for the transmission as well as track the buffered and current playtime in the YouTube player. Since the QoE of YouTube depends on stalling [66, 67], monitoring the buffered playtime gives us an indication whether the current performance offered by the network leads to a QoE degradation for

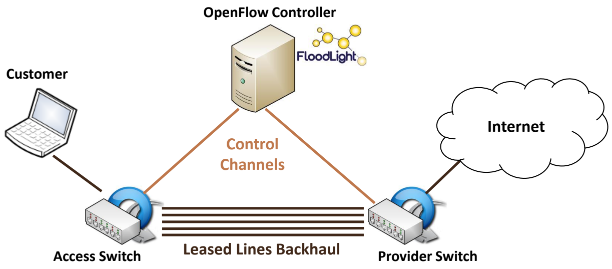


Figure 4.11: *Application-Aware SDN Testbed Setup*

Experiments

In the following, we discuss the differences between each of the experiments as well as their purpose. The start of the YouTube video coincides with the start of each experiment. The video is played out with a resolution of 480p by default.

i) **Reference Experiment:** In this experiment only YouTube traffic to a single client is transmitted. The controller chooses one of the five available links at random to transfer the flow. The YouTube traffic can use the full 10 Mbps available on that link. This experiment gives us a baseline in terms of the available buffered playtime we can expect under optimal conditions.

ii) **Reference Experiment with Interfering Traffic:** For this experiment two additional PCs are connected to the testbed. One is connected to the access switch, the other to the provider switch. We use Iperf [54] to generate traffic between those two machines in order to emulate other users on the network. In addition to the YouTube traffic, 20 TCP flows are started sequentially after 60 seconds with a flow inter-arrival time of one second. The controller directs all traffic via only one of the five links, which gives us a worst case approximation.

iii) **Round-Robin Path Selection:** The testbed setup for this experiment remains the same as in the previous case. Once again 20 TCP flows are generated. However, this time the controller can use more than one link. It does so by directing each new flow to a different link in a round-robin fashion. This experiment represents our naive load-balancing approach.

iv) **Bandwidth-Based Path Selection:** The same traffic and testbed as in the previous experiments is also used here. The controller is still able to use all links. Links are selected by their currently used bandwidth. When a new flow arrives, the controller determines the least loaded link and directs the flow to it. At the same time the controller checks the bandwidth required by each of the flows every second via the switches' flow table counters. If there is a link with free capacity available, the controller will then redirect the largest flow in terms of bandwidth consumption from a loaded link to the free link. In order to avoid constant redirection, this can only happen once every ten seconds for a specific flow.

v) **Deep Packet Inspection:** We extend the testbed by a machine performing Deep Packet Inspection (DPI). The machine is connected to the management network and can be contacted by the controller. The experiment parameters are the same otherwise. In this experiment, the controller directs all traffic via one link. The first ten packets of each flow, are mirrored to the controller, which then sends them to the DPI machine running a combination of TShark, the console version of Wireshark [84], and several filter rules based on regular expressions. The DPI informs the controller about the nature of the flow. If a particular flow is a YouTube video, the controller will redirect the flow to another less congested link.

vi) Application-Aware Path Selection: Finally, in this ex-periment, we leverage the information YoMo provides us with as input for the controller. The experiment is identical to the previous one, except that 50 TCP flows are generated to create a high load scenario and the machine used for DPI in the previous experiment is now used to receive application information containing the current YouTube buffer level and flow information. We call this machine the ”application station”. When the buffer level gets below a certain threshold, the application station informs the controller that an action is required for a particular flow in order to maintain the QoE for the user.

4.3.2 Measurement Results

In this section, we discuss the measurement results of our experimental investigation. All experiments were repeated five times. However, for the sake of visualization, only one representative run is depicted. Each experiment has a duration of 420 seconds. The used video[1] has a mean data rate of 2.6 Mbps with standard deviation of 250 kbps.

Reference Experiment

The upper curve in Figure 4.12 shows the pre-buffered playtime of the YouTube player at the client in seconds over the duration without interfering traffic. It can be seen that a pre-buffered playtime of about 55 seconds is reached within the first 10 seconds. Thus, 55 seconds playback can be achieved without further data. While the video is played out, the buffer decreases but is constantly refilled so that the buffer maintains a stable level. This is as expected for the reference

[1] YouTube Video: "Waterfall" 90mins "Sleep Video" Bull Creek; http://www.youtube.com/watch?v=WZtn2n51Xrw

experiment. The buffer level never drops significantly and, most importantly, it never reaches zero, which would cause the video to stall.

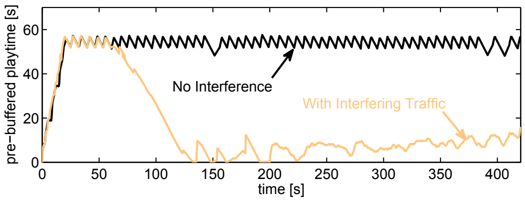


Figure 4.12: *Buffered Playtime (1 Link), No Interference and 20 Additional TCP Flows*

Reference Experiment with Interfering Traffic

The repetition of the reference experiment with interfering traffic yields a different result as is illustrated in Figure 4.12, which again shows the pre-buffered playtime in seconds over time. Up to the point when the interfering traffic starts, the behavior is exactly the same as in the reference experiment. However, with the reduced bandwidth available, the buffer level continuously falls as the video is played out until it is empty and the video stalls at about the 140s mark. At about 200s the YouTube player automatically reduces the default resolution of 480p to 360p, and therewith the video bit rate. This enables the video to be played out again with less stalling albeit in a lower quality. The buffer, however, does not recover and stays at a low level of about 10s pre-buffered playtime.

Round-Robin Path Selection

The approach of balancing the flows across multiple links in a round robin fashion should naively improve the situation for the YouTube user compared to the one link scenario. However, this is not necessarily the case with heterogeneous traffic as is shown in Figure 4.13. After the initial undisturbed phase, the YouTube buffer once again can not maintain its high level. However, this time it is not immediately emptied and the video keeps playing. As the controller assigns all flows in a round robin-fashion but can not tell, which flow is an ”elephant” and which is a mouse, the bandwidth distribution can be very uneven. This eventually also comes to haunt our YouTube video at about 270s after the start of experiment when it has to share its link with multiple high bandwidth flows. Subsequently, the buffer is drained and the video stalls yet again.

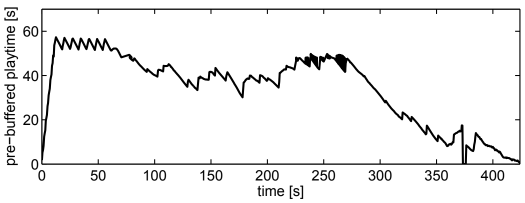


Figure 4.13: *Buffered Playtime (5 Links, 20 Additional TCP Flows, Round-Robin Path-Selection)*

Bandwidth-Based Path Selection

Taking into account the used bandwidth per flow is the next logical step from our round-robin approach that suffered from uneven bandwidth distribution. However, it fares little better in terms of YouTube streaming performance as can be seen in Figure 4.14. This is due to fact that all flows, except the YouTube flow, are ”elephants” and try to use the maximum bandwidth available. When the interfering traffic starts at 60 seconds, the buffer begins to decrease. While this is not as swift as in the one-link scenario, it steadily decreases and eventually reaches a stalling event. At this point YouTube switches again to a lower resolution and the video is able to recover.

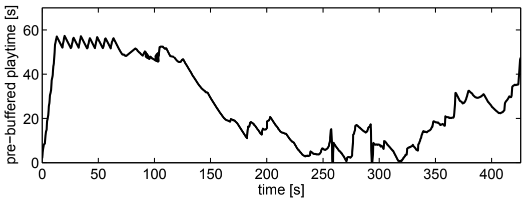


Figure 4.14: *Buffered Playtime (5 Links, 20 Additional TCP Flows, Bandwidth-Based Path-Selection)*

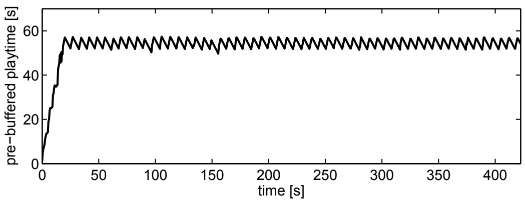


Figure 4.15: *Buffered Playtime (5 Links, 20 Additional TCP Flows, DPI-Based Path-Selection)*

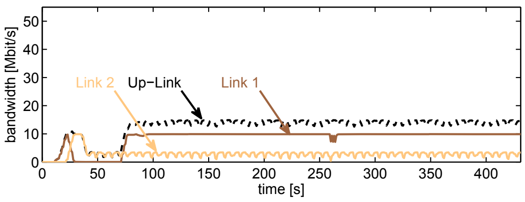


Figure 4.16: *Used Bandwidth (5 Links, 20 Additional TCP Flows, DPI-Based Path-Selection)*

Deep Packet Inspection

The previous experiments show that network information alone is not sufficient to provide the YouTube user with a good performance. However, using deep packet inspection, we can identify the YouTube traffic in the network and prioritize it. This approach yields the desired success as is shown in Figure 4.15. The YouTube pre-buffered playtime behaves the same way as in our reference scenario without any interfering traffic. However, the overall usage of the available network resources is reduced as depicted in Figure 4.16. Here, the used bandwidth in Mbps over time is shown. Initially, there is a spike up to the maximum utilization of 10 Mbps on Link1. This is due to the YouTube buffer ramping up. After about 20 seconds the deep packet inspection has identified the traffic as YouTube video and notified the controller, which redirects the video to its own dedicated Link2. When the interfering traffic starts at 60s, it remains on Link1, not able to influence the YouTube stream. Since the DPI can not provide application state information all links are reserved for YouTube streams, which in this case results in a network resource utilization of just about 15% on average.

Application-Aware Path Selection

While the deep packet inspection approach has already enabled us to provide a good experience to the user, we also wasted a lot of bandwidth as the dedicated link for the YouTube video is only slightly used after the initial ramp-up of the buffer and the other dedicated links remain empty. It would be beneficial for the network only to use the extra resources when there actually is a problem and return to normal operation when it no longer persists. This is where the benefits of the SDN northbound interface come into play. By leveraging this interface, we can implement application-state awareness in the network. The benefits can be seen in Figure 4.17. With all traffic on one link, the buffer level of the YouTube video starts to decrease, like we have seen in 4.3.2. When it reaches a threshold of 20s pre-buffered playtime, the controller is triggered and it redirects the YouTube traffic to a less-loaded link. However, this time this is not a dedicated link. It can be used by other traffic. Therefore, once the YouTube video has reached a buffered playtime of 35 seconds, the controller can use more capacity on the link for other traffic until the video again reaches its lower playtime threshold. As we can see in Figure 4.18, all links in this experiment are fully loaded. Despite of this, the YouTube user still experiences a good quality using the Application-Aware SDN approach.

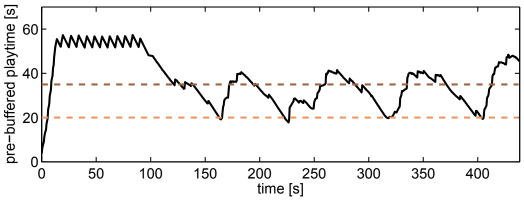


Figure 4.17: *Buffered Playtime (5 Links, 50 Additional TCP Flows, Applicaton-Aware Path-Selection)*

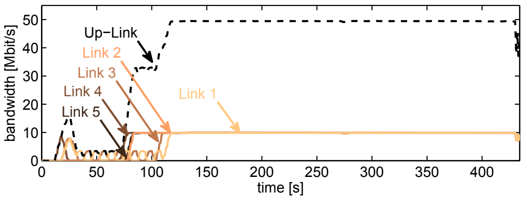


Figure 4.18: *Used Bandwidth (5 Links, 50 Additional TCP Flows, Application-Aware Path-Selection)*

Resource-Overhead

All in all, two OpenFlow switches are required for these approaches in addition to the controller. For deep packet inspection and application-aware networking another machine is required to gather information about the running applications. Furthermore, all of the described methods to improve the performance for the YouTube user cause a certain overhead on the control plane. In the following, we describe said overhead for each approach and determine its efficiency in terms of resource consumption. As a gauge for the efficiency ρ of resource utilization, we use the ratio of bandwidth used on average once the interfering traffic has started,

$$\rho = \frac{Mean(UsedBandwidth)}{AvailableBandwidth}.$$

a) **Round-Robin Path Selection:** The simplest solution with round-robin flow scheduling also has the least overhead. Here, only the OpenFlow Packet-In, Packet-Out and Flow Mod messages have to be transmitted via the control channel. No additional traffic and components are necessary. For this approach all resources are used, therefore $\rho \approx 1$.

b) **Bandwidth-Based Path Selection:** For the band-width-based approach, more overhead in terms of control channel traffic is required compared to the round-robin approach. The controller needs to periodically query the flow table counters in the switches to determine the current bandwidth utilization of each flow and link. Additionally, the balancing of bandwidth usage causes more flow redirection operations, which increases the number of sent Flow Mod packets and the CPU load on the switches. Again all available resources are used with $\rho \approx 1$.

c) **Deep Packet Inspection:** The deep packet inspection approach requires an additional, potentially heavy loaded, computing resource in the control plane to perform the packet analysis. Furthermore, the first ten packets of each flow have to be mirrored. We use the control channel for this. The bandwidth utilization on the control channel in the DPI case shows an increase of bandwidth utilization to just under 1 Mbps when packet mirroring occurs at the beginning of the experiment. This is double the required bandwidth used by normal reactive flow setups, which peak at about 0.5 Mbps. However, compared to the bandwidth-based approach, no real-time querying of the switches is necessary. A general problem of DPI is also that the application signatures have to be constantly updated, which requires additional expenses. As there is no access to the application state in this approach, all links except one have to be dedicated lines for potential YouTube flows. As in our scenario only one YouTube flow is actually transmitted a lot of bandwidth is left unused leading to a ρ of just 0.15. Using only one line for YouTube flows would increase the bandwidth usage to $\rho \approx 0.85$ which is still less compared to the other scenarios.

d) Application-Aware Path Selection: Like DPI, the application-state-aware approach also requires an additional computing instance to receive and filter the application information for the controller. While this method puts no significant overhead on the control channel, it requires the exchange of information of the application station instance and the served clients. This may cause a significant amount of traffic, if the application state changes constantly. Furthermore, an additional software component is required at the client to monitor the application. This can be a part of the application itself or, as is the case with us, be a plug-in for the software that should be monitored. With the benefit of application state information, all resources can once again be used leading to a $\rho \approx 1$.

Quantifying the Results

Figure 4.19 shows the cumulative distribution functions of the pre-buffered playtime for five experiment runs of our approaches once interfering traffic has started. For the sake of readability the confidence intervals are only drawn for the experiment without flow management. As can be seen, they are very small and this is also true for the other approaches. The approach without any flow management performs the worst in terms of playtime, having about 70% of the time a pre-playtime below 20 seconds. The Bandwidth-based and Round Robin approaches fare slightly better, but at the cost of a reduced video quality. The Round-Robin approach seems to outperform the Bandwidth-based approach, but this is because the video stalls earlier and so the Round Robin approach benefits much sooner from the reduced video size. Deep Packet Inspection and the Application-Aware SDN approach show by far the best performance with a pre-buffered playtime of 50 seconds and above for about 90 and 80% of the time, respectively. However, taking the conserved bandwidth and smaller resource overhead into account, the Application-Aware SDN approach appears to be the most viable.

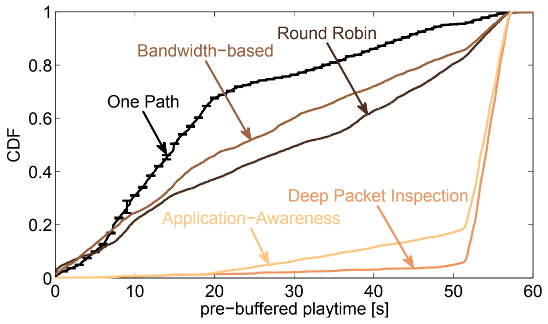


Figure 4.19: *Cumulative Distribution Function for the Pre-Buffered Playtime (20 Additional TCP Flows)*

4.4 Lessons Learned

In this chapter, we have presented our findings on how to obtain and leverage QoS and QoE information in conjunction with the SDN Northbound-API to improve the quality for users and make the network management more resource-efficient.

We described a test setup to perform a subjective survey on the topic of cloud gaming as a resource-intensive example application and evaluated the results. We determined that the QoS parameter that influences the perceived game experience most is downstream packet loss with a weight of 1.0 based on information gain.

While this is very similar to QoE in conventional gaming, in cloud gaming, it is far more important for players in which direction packet loss occurs as upstream packet loss only has a weight of 0.37. This is caused by the fact that, in general, the perceived quality of the video plays an important role and is severely impacted by even slight amounts of packet loss. This is especially true for games that rely on impressive visuals. The second most important impact factor is downstream delay with a weight of 0.583. This smaller value can be explained by the fact that humans can compensate a certain amount of constant delay so that it does not greatly lessen the experience.

However, the QoE is not only dependent on the QoS parameters of delay and packet loss, but also has to be put into context with the content as we have seen that different types of games may result in disruptions to be perceived differently by the user. Therefore, it is obvious that solely with measurable QoS parameters, the true user-perceived quality can not be estimated. Additional application information and, ideally, application state information is required.

We continued by illustrating the benefits of combining application-state information with SDN network control for network management for the example of YouTube streaming as the application state could not be determined from our proprietary cloud gaming setup. We saw that users can benefit profoundly from this approach compared to purely QoS-based methods. Using application state information and deep packet inspection as input to manage the network, the users' video stream can be prevented from stalling. In both cases, the pre-buffered playtime is above 50 seconds in 80% of the time. In the same scenario, the conventional approaches fail to prevent stalling. However, the improved performance comes at a cost of a resource overhead caused by the necessary signaling traffic and computation. Generally, more signaling and computing is required to profit from the advantages in QoE. Currently, to achieve this, a reactive flow setup with OpenFlow is required, which limits the applicability of this approach to smaller networks such as the scenario described. Still, with the ongoing trends towards Network Functions Virtualization and the increasing use of network processors, this may change in the future.

The main contribution of this chapter is on the one hand to quantify the impact of QoS parameters on the QoE of cloud gaming and on the other hand to show that the SDN control plane can be used to aggregate and leverage QoS and QoE information provided by the network devices as well as the applications running in the network to efficiently manage the use of the available network resources.

5 Conclusion

Software Defined Networking has emerged as a game-changing approach to networking, which is evident by the growing rate of adoption and number of available products on the market. The inherent flexibility it provides compared to traditional networking has become its key incentive beyond the possibility of CAPEX reduction. Consequently, the concept is applied to more and more areas in networking.

However, not all of these areas are suited equally to this new approach. Through the performance investigations discussed in this monograph, it became apparent that SDN in its current form is best suited for two types of networks. The first type are networks whose traffic patterns are well-known or can be controlled. In this type of networks, the forwarding rules can be pre-determined by the SDN controller. Thus, it only has to become involved when either a configuration change is required or a failure occurs. In both cases the flexibility of realizing the control plane in software with open interfaces enables the controller to swiftly take the correct action. In particular, we have shown that the advance notification of an SDN controller of an imminent topology change can facilitate the maintenance of service quality.

The second type of networks are those with a relatively small number of flows but a high need for management, e.g., due to limited resources. Typically, networks of this kind are access networks. We have shown that with the input of application information and the on-the-fly adjustment of forwarding rules through an SDN controller leveraging this information, the Quality of Experience of an application in such a network can be improved significantly. This is especially true when classical Quality of Service parameters are not sufficient, e.g., due to traffic asymmetry, as we illustrated in the case of cloud gaming.

Beyond using SDN as a means for network control, we have shown that it can be applied to augment the network and its management. We were able to determine that SDN can be applied in the monitoring domain to passively monitor the network without a significant loss of accuracy but with a significant gain in flexibility and reduction of overhead. In fact, network monitoring recently has emerged as a possible ”killer-app” for SDN as attested by recent product announcements like ”BigTap” from BigSwitch Networks.

Despite of these promising results, we have also determined that current SDN implementations have to be significantly improved to become relevant beyond test beds and insular deployments. This starts with the available switching hardware, which does not support all of the required capabilities on the data path. This takes away a lot of the flexibility as performance is key and features that do not work swiftly cannot be used in production deployments. However, the software controllers are also an issue. In our studies, we have shown that controller performance has a huge impact on the performance of the overall system. At the same time we determined that current controller realizations have shortcomings in the way they treat devices and incoming traffic so that service disruption is a distinct possibility.

In general, however, our studies have demonstrated that SDN can indeed work as a concept. The investigations we have conducted were based on early prototypes and many of the shortcomings we found are being addressed. The novel approaches to leverage SDN we have shown are gaining more traction within the networking community and may soon be realized in products for specific domains. Despite of this, SDN is not an omni-tool that should be applied to each and every use-case without question. It is prudent to weigh the benefits against drawbacks for each scenario in which one might consider using SDN.

Bibliography and References

Bibliography of the Author

— Journals and Book Chapters —

[1] T. Zinner, M. Jarschel, T. Hoßfeld, P. Tran-Gia, and W. Kellerer, "A Compass Through SDN Networks," 2014, to be published in IEEE Communications Magazine.

[2] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hoßfeld, "Gaming in the clouds: QoE and the users' perspective," *Mathematical and Computer Modelling*, vol. 57, no. 11-12, pp. 2883–2894, June 2013.

— Conference Papers —

[3] M. Jarschel, F. Wamser, T. Höhn, T. Zinner, and P. Tran-Gia, "SDN-based Application-Aware Networking on the Example of YouTube Video Streaming," in *Proceedings of the 2nd European Workshop on Software Defined Networks (EWSDN 2013)*, Berlin, Germany, October 2013, pp. 87–92.

[4] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia, "Modeling and Performance Evaluation of an OpenFlow Architecture," in *Proceedings of the 23rd International Teletraffic Congress (ITC 2011)*, San Francisco, CA, USA, September 2011, pp. 1–7.

[5] M. Jarschel, F. Lehrieder, Z. Magyari, and R. Pries, "A Flexible OpenFlow-Controller Benchmark," in *Proceedings of the 1st European Workshop on*

Software Defined Networks (EWSDN 2012), Darmstadt, Germany, October 2012, pp. 48–53.

[6] M. Jarschel, T. Zinner, T. Höhn, and P. Tran-Gia, "On the Accuracy of Leveraging SDN for Passive Network Measurements," in *Proceedings of the Australasian Telecommunication Networks & Applications Conference (ATNAC 2013)*, Christchurch, New Zealand, November 2013, pp. 41–46.

[7] M. Jarschel, D. Schlosser, S. Scheuring, and T. Hoßfeld, "An Evaluation of QoE in Cloud Gaming Based on Subjective Tests," in *Proceedings of the Workshop on Future Internet and Next Generation Networks (FINGNet 2011)*, Seoul, Korea, June 2011, pp. 330–335.

[8] R. Pries, M. Jarschel, and S. Goll, "On the Usability of OpenFlow in Data Center Environments," in *Proceedings of the Workshop on Clouds, Networks and Data Centers collocated with the IEEE International Conference on Communications (ICC 2012)*, Ottawa, Canada, June 2012, pp. 5533–5537.

— Demonstrations —

[9] M. Jarschel and R. Pries, "An OpenFlow-Based Energy-Efficient Data Center Approach," in *Proceedings of the ACM SIGCOMM 2012 conference*, Helsinki, Finland, August 2012, pp. 87–88.

[10] M. Jarschel, D. Schlosser, M. Duelli, and R. Pries, "IPOM: Interactive Proxy Management Tool for Future Communication Networks Using OpenFlow," Kaiserslautern, Germany, pp. 1–2, June 2011.

General References

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in

Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, April 2008.

[12] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu *et al.*, "B4: Experience with a globally-deployed software defined wan," in *Proceedings of the ACM SIGCOMM 2013 conference*, Hong Kong, China, August 2013, pp. 3–14.

[13] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks.* O'Reilly Media, 2013.

[14] O. M. E. Committee" *et al.*, "Software-defined networking: The new norm for networks," *Open Networking Foundation White Papers*, 2012.

[15] J. Duffy, "Cisco takes fight to SDNs with bold Insieme launch," http://www.networkworld.com/news/2013/110613-cisco-insieme-275666.html, November 2013, last accessed on 2014.03.06.

[16] "Open networking foundation," https://www.opennetworking.org/, last accessed 2014.03.12.

[17] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 36–43, July 2013.

[18] A. Doria, R. Gopal, H. Khosravi, L. Dong, J. Salim, and W. Wang, "Forwarding and control element separation (forces) protocol specification," 2010.

[19] A. Devlic, W. John, and P. Sköldström, "Carrier-grade network management extensions to the sdn framework," in *Proceedings of the 8th Swedish National Computer Networking Workshop (SNCNW 2012)*, Stockholm, Sweden, June 2012.

[20] G. Hampel, M. Steiner, and T. Bu, “Applying software-defined networking to the telecom domain,” in *Proceedings of 16th IEEE International Global Internet Symposium (GI 2013) collocated with IEEE INFOCOM 2013*, Turin, Italy, April 2013, pp. 133–138.

[21] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, “Onix: A distributed control platform for large-scale production networks,” in *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, Canada, October 2010, pp. 351–364.

[22] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, “Composing software defined networks,” in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL, USA, April 2013, pp. 1–13.

[23] C. E. Rothenberg, M. R. Nascimento, M. R. Salvador, C. N. A. Corrêa, S. Cunha de Lucena, and R. Raszuk, “Revisiting routing control platforms with the eyes and muscles of software-defined networking,” in *Proceedings of the first workshop on Hot topics in software defined networks (HotSDN 12)*, Helsinki, Finland, August 2012, pp. 13–18.

[24] C. Yu, C. Lumezanu, Y. Zhang, V. Singh, G. Jiang, and H. V. Madhyastha, “Flowsense: Monitoring network utilization with zero measurement cost,” in *Proceedings of the Passive and Active Measurement Conference (PAM 2013)*, Hong Kong, China, January 2013, pp. 31–41.

[25] H. Kim and N. Feamster, “Improving network management with software defined networking,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, February 2013.

[26] T. Hoßfeld, R. Schatz, M. Varela, and C. Timmerer, “Challenges of QoE Management for Cloud Applications,” *IEEE Communications Magazine*, vol. 50, no. 4, pp. 28–36, April 2012.

[27] G. Wang, T. Ng, and A. Shaikh, “Programming your network at run-time for big data applications,” in *Proceedings of the first workshop on Hot topics in software defined networks (HotSDN 12)*, Helsinki, Finland, August 2012, pp. 103–108.

[28] “OpenFlow Switch Specification, Version 1.4.0,” https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf, October 2013, last accessed on 2014.03.06.

[29] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, “Elastic Tree: Saving Energy in Data Center Networks,” in *Proceedings of the 7th USENIX Symposium on Networked System Design and Implementation (NSDI 10)*, San Jose, CA, USA, April 2010, pp. 249–264.

[30] S. Das, G. Parulkar, P. Singh, D. Getachew, L. Ong, and N. McKeown, “Packet and Circuit Network Convergence with OpenFlow,” in *Proceedings of the Optical Fiber Conference (OFC/NFOEC’10)*, San Diego, CA, USA, March 2010.

[31] R. Braga, E. S. Mota, and A. Passito, “Lightweight DDoS Flooding Attack Detection Using NOX/OpenFlow,” in *Proceedings of the 35th Annual IEEE Conference on Local Computer Networks*, Denver, CO, USA, October 2010, pp. 416–423.

[32] M. P. Mateo, “Openflow switching performance,” Master’s thesis, Politecnico Di Torino, Torino, Italy, 2009.

[33] A. Bianco, R. Birke, L. Giraudo, and M. Palacin, “Openflow switching: Data plane performance,” in *Proceedings of the IEEE International Conference on Communications (ICC2010)*, Cape Town, South Africa, May 2010, pp. 1–5.

[34] V. Tanyingyong, M. Hidell, and P. Sjödin, "Improving PC-Based OpenFlow Switching Performance," in *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '10)*, New York, NY, USA, October 2010.

[35] Y. Luo, P. Cascon, E. Murray, and J. Ortega, "Accelerating OpenFlow Switching With Network Processors," in *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS' 09)*, New York, NY, USA, October 2009, pp. 70–71.

[36] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee, "Devoflow: scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM 2011 conference*, Toronto, ON, Canada, August 2011, pp. 254–265.

[37] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An Open Framework for OpenFlow Switch Evaluation," in *Proceedings of the Passive and Active Measurement Conference (PAM 2012)*, Vienna, Austria, March 2012.

[38] R. Sherwood and K.-K. Yap, "Cbench Controller Benchmarker," http://www.openflowswitch.org/wk/index.php/Oflops, 2011, last accessed on 2014.03.09.

[39] A. Tootoonchian, S. Gorbunov, Y. Ganjali, M. Casado, and R. Sherwood, "On controller performance in software-defined networks," in *Proceedings of the 2nd USENIX conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE '12)*, San Jose, CA, USA, April 2012.

[40] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the netfpga platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, San Jose, CA, USA, November 2008, pp. 1–9.

[41] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker, “Applying nox to the datacenter,” in *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, New York, NY, USA, October 2009.

[42] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, “Extending networking into the virtualization layer.” in *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, New York, NY, USA, October 2009.

[43] “Nox Classic,” https://github.com/noxrepo/nox-classic, last accessed on 2014.03.09.

[44] “Floodlight,” http://www.projectfloodlight.org/floodlight/, last accessed on 2014.03.09.

[45] “Maestro,” http://code.google.com/p/maestro-platform/, last accessed on 2014.03.09.

[46] F. Wamser, R. Pries, D. Staehle, K. Heck, and P. Tran-Gia, “Traffic characterization of a residential wireless internet access,” *Special Issue of the Telecommunication Systems (TS) Journal*, vol. 48, no. 1-2, pp. 5–17, 2011.

[47] Endace, “DAG 7.5G2 Datasheet,” http://www.emulex.com/artifacts/b4469f7d-ecee-4022-8232-295390c7c036/end_ds_all_dag_7.5g2.pdf, last accessed on 2014.03.09.

[48] P. Tran-Gia, *Einführung in die Leistungsbewertung und Verkehrstheorie.* Munich, Germany: Oldenbourg, July 2006.

[49] T. Zseby, “Deployment of sampling methods for sla validation with non-intrusive measurements,” in *Proceedings of Passive and Active Measurement Workshop (PAM 2002)*, Fort Collins, CO, USA, March 2002, pp. 25–26.

[50] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, “Opentm: traffic matrix estimator for openflow networks,” in *Proceedings of the Passive and Active*

Measurement Conference (PAM 2010), Zurich, Switzerland, April 2010, pp. 201–210.

[51] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches," in *Proceedings of the USENIX workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE '11)*, Boston, MA, USA, March 2011.

[52] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch," in *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL, USA, April 2013, pp. 29–42.

[53] D. Hock, M. Hartmann, S. Gebert, M. Jarschel, T. Zinner, and P. Tran-Gia, "Pareto-Optimal Resilient Controller Placement in SDN-based Core Networks," in *Proceedings of the 25th International Teletraffic Congress (ITC 2013)*, Shanghai, China, September 2013.

[54] "Iperf: The tcp/udp bandwidth measurement tool," https://github.com/esnet/iperf, last accessed on 2014.03.09.

[55] S. Hemminger *et al.*, "Network emulation with netem," in *Linux Conf Au*. Citeseer, 2005, pp. 18–23.

[56] "Beacon," http://www.beaconcontroller.net/, last accessed on 2014.03.09.

[57] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets-IX)*, no. 19, Monterey, CA, USA, October 2010.

[58] D. Schwerdel, "ToMaTo - Next Generation Testbed Software," ICSY Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, Tech. Rep. ICSY-2010-3, 2010.

[59] D. Schwerdel, D. Günther, R. Henjes, B. Reuther, and P. Müller, "German-lab experimental facility," Berlin, Germany, September 2010, pp. 1–10.

[60] J. Fontán, T. Vázquez, L. Gonzalez, R. Montero, and I. Llorente, "Open-nebula: The open source virtual machine manager for cluster computing," in *Proceedings of the Open Source Grid and Cluster Software Conference 2008*, Oakland, CA, USA, May 2008.

[61] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the linux virtual machine monitor," in *Proceedings of the Linux Symposium*, Ottawa, ON, Canada, June 2007, pp. 225–230.

[62] Pica8, "Pronto 3290 OpenFlow Switch," http://www.pica8.org/products/p3290.php, last accessed on 2014.03.11.

[63] M. Fiedler, K. Kilkki, and P. Reichl, "From quality of service to quality of experience," in *Dagstuhl Seminar Proceedings*, no. 09192, Dagstuhl, Germany, May 2009.

[64] A. R. Curtis, W. Kim, and P. Yalagandula, "Mahout: Low-overhead data-center traffic management using end-host-based elephant detection," in *Proceedings of the 30th IEEE International Conference on Computer Communications (IEEE INFOCOM 2011*, Shanghai, China, April 2011, pp. 1629–1637.

[65] S. Das, Y. Yiakoumis, G. Parulkar, N. McKeown, P. Singh, D. Getachew, and P. D. Desai, "Application-aware aggregation and traffic engineering in a converged packet-circuit network," in *Proceedings of the Optical Fiber Conference (OFC/NFOEC'11)*, Los Angeles, CA, USA, March 2011.

[66] F. Dobrian, A. Awan, D. Joseph, A. Ganjam, J. Zhan, V. Sekar, I. Stoica, and H. Zhang, "Understanding the impact of video quality on user engagement," in *Proceedings of the ACM SIGCOMM 2011 conference*, Toronto, ON, Canada, August 2011, pp. 362–373.

[67] T. Hoßfeld, R. Schatz, M. Seufert, M. Hirth, T. Zinner, and P. Tran-Gia, "Quantification of YouTube QoE via Crowdsourcing," in *Proceedings of the IEEE International Workshop on Multimedia Quality of Experience - Modeling, Evaluation, and Directions (MQoE 2011)*, Dana Point, CA, USA, December 2011.

[68] I. Nave, H. David, A. Shani, Y. Tzruya, A. Laikari, P. Eisert, and P. Fechteler, "Games large graphics streaming architecture," in *Proceedings of the IEEE International Symposium on Consumer Electronics (ISCE 2008)*, Vilamoura, Portugal, April 2008, pp. 1–4.

[69] M. Pigora and S. Waldron, "The 3d world in your browser: A server rendering approach," in *Proceedings of the Interservice/Industry Training, Simulation & Education Conference (I/ITSEC 2010)*, Orlando, FL, USA, November 2010.

[70] D. Chan, "On the feasibility of video gaming on demand in wireless lan/wimax," last accessed on 2014.03.11.

[71] Y.-C. Chang, P.-H. Tseng, K.-T. Chen, and C.-L. Lei, "Understanding the performance of thin-client gaming," in *Proceedings of the IEEE International Communications Quality and Reliability Workshop (CQR 2011)*, Naples, FL, USA, May 2011, pp. 1–6.

[72] T. Szigeti and C. Hattingh, *End-to-End QoS Network Design: Quality of Service in LANs, WANs, and VPNs (Networking Technology)*. Cisco Press, 2004.

[73] M. Claypool and K. Claypool, "Latency and player actions in online games," *Communications of the ACM*, vol. 49, no. 11, p. 45, November 2006.

[74] A. Ojala and P. Tyrvainen, "Developing cloud business models: A case study on cloud gaming," *IEEE software*, vol. 28, no. 4, pp. 42–47, July 2011.

[75] P. Ross, "Cloud computing's killer app: Gaming," *IEEE Spectrum*, vol. 46, no. 3, pp. 14–14, March 2009.

[76] T. Chang, "Gaming will save us all," *Communications of the ACM*, vol. 53, no. 3, pp. 22–24, March 2010.

[77] L. Pantel and L. C. Wolf, "On the impact of delay on real-time multiplayer games," in *Proceedings of the 12th International workshop on Network and operating systems support for digital audio and video (NOSSDAV 2002)*, Miami Beach, FL, USA, May 2002, pp. 23–29.

[78] T. Henderson and S. Bhatti, "Networked games: a qos-sensitive application for qos-insensitive users?" in *Proceedings of the ACM SIGCOMM workshop on Revisiting IP QoS: What have we learned, why do we care? (RIPQOS)*, Karlsruhe, Germany, August 2003, pp. 141–147.

[79] *P.800 : Methods for subjective determination of transmission quality*, ITU-T Std.

[80] P. Kabel, F. Hermann, and M. Hengstenberg, "Spielplatz deutschland," 2006.

[81] "RapidMiner," http://rapidminer.com, last accessed on 2014.03.11.

[82] "WEKA," http://sourceforge.net/projects/weka/, last accessed on 2014.03.11.

[83] B. Staehle, M. Hirth, R. Pries, F. Wamser, and D. Staehle, "YoMo: A YouTube Application Comfort Monitoring Tool," in *Proceedings of the Workshop on Quality of Experience for Multimedia Content Sharing (QoEMCS 2010)*, Tampere, Finland, June 2010.

[84] "Wireshark," http://www.wireshark.org/, last accessed on 2014.03.11.

ISSN 1432-8801