Identification and Design of a Data Plane Resource Optimization Mechanism for Application-Controlled SDN

Master-Arbeit

Sascha Bleidner
PS-D-0017

TECHNISCHE UNIVERSITÄT DARMSTADT

Identification and Design of a
Data Plane Resource Optimization Mechanism for Application-Controlled SDN
Master-Arbeit
PS-D-0017

Eingereicht von Sascha Bleidner
Tag der Einreichung: 31. Mai 2015

Gutachter: Prof. Dr. David Hausheer
Betreuer: Jeremias Blendin, Dipl.-Wirtsch.-Inf.
Zweitbetreuer: Julius Rückert, M.Sc.

Technische Universität Darmstadt
Fachbereich Elektrotechnik und Informationstechnik
Institut für Datentechnik

Fachgebiet P2P Systems Engineering
Prof. Dr. David Hausheer

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Master-Arbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in dieser oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Die schriftliche Fassung stimmt mit der elektronischen Fassung überein.

Darmstadt, den 31. Mai 2015 Sascha Bleidner

Contents

List of Figures

List of Tables

1 Introduction

Software Defined Networking (SDN) is a recent innovative idea in the networking domain. Its core idea is the separation of the control and data plane within a network. The control and data plane in traditional networking devices are both integrated within the device itself, which makes it difficult to change on or the other independently. SDN separates those two components, by enabling the control plane to be implemented in a centralized fashion, which controls the data plane implementation of each individual networking device.

OpenFlow is one integral part of SDN, which implements a novel communication protocol for a centralized communication scheme between a controller and the switches within a network. The OpenFlow protocols allows the controller to configure the flow-based forwarding data plane implementation by each switch. Being able to control and configure a set of network devices by an centralized OpenFlow controller provides a number of benefits, such as easier network programming and a central view on the network topology. However, moving the control plane towards a centralized approach imposes a number of challenges and limitations, such as scalability issues.

A majority of optimization approaches exclusively focus on the challenges of the controller implementing the centralized control plane [99, 23, 113], but challenges and limitations can also be identified for the OpenFlow data plane. An comprehensive analysis of the challenges and limitations of the OpenFlow data plane does not exist yet, therefore the first contribution of this work is an analysis of the limitations and bottlenecks of the OpenFlow data plane. Those limitations and bottlenecks are derived from a structural analysis of state-of-the-art optimization approaches. The limitations and bottlenecks of the OpenFlow data plane are categorized into so called data plane resources, where each resource describes one part of the OpenFlow data plane concept, which is subject to a concrete limitation.

An overview of the collected resource optimization approaches is provided. In addition this work offers a second collection of state-of-the-art application interaction approaches, since a deeper interaction between the network and network applications can be implemented using the interfaces defined in the SDN concept. Those application interaction approaches are collected and analyzed with a focus on possible cooperation with one or multiple resource optimization approaches.

SDN already introduces novel abstractions for the networking domain, however as the analysis of bottlenecks of the OpenFlow control plane indicates, those abstractions are not yet sufficient for a flawless implementation. This work contributes towards new possible abstraction models within the SDN concept by providing a systematically classification of both resource optimization and application interaction concepts. The classification is based on the given overview of state-of-the-art resource optimization and application interaction approaches, which are analyzed regarding their similarities and common concepts. Each class within this classification is defined regarding its structure, requirements, objectives and certain tradeoffs, similar to software design patterns [110].

Instead of optimizing certain resources and interaction models for a specific SDN application, it would be desirable to be able to apply certain classes of this classification to a given SDN application. Their application should not depend on the SDN application itself, but rather on the challenges and limitations within this SDN application. In order to evaluate how general applicable those concepts would be, the concept of dynamic network service chaining is selected as an SDN application. The dynamic network service chaining system is analyzed in terms of resource limitations, where each identified resource

limitation is addressed by applying a corresponding optimization concept. The results and benefits of each resource optimization concept are evaluated with a prototype implementation. Since hardware has special characteristics compared to virtual switches running on commodity hardware, the prototype includes a NEC P5420 hardware switch, in order to evaluate the impact of adopting a system to the characteristics of a real hardware switch. The results are promising, supporting the idea of the existence of general applicable concepts onto a variety of resource limitations in the context of SDN applications.

1.1 Outline

The OpenFlow data plane is analyzed regarding its limitations in Chapter 3. Those limitations are categorized in so called data plane resources, which are used as a reference throughout this work. Those limitations are transformed into general classes of resource optimization concepts in Chapter 4, which introduces each class and discusses the combination of different resource optimization concepts. The state of the art application interaction approaches are analyzed in Chapter 5, where similarities and common concepts are gathered for defining the classes of application interaction concepts in Chapter 6. The dynamic network service chaining concept is selected for applying several concepts of both Chapter 4 and 6. The resulting system design is described in Chapter 7. A prototype of this system design is developed and described in Chapter 8 and evaluated in Chapter 9. Finally, the work is concluded in Chapter 10, which in addition gives an outlook on further research directions in the context of resource optimization and a deeper application interaction.

2 Background

This chapter gives some background insights into the relevant technologies, such as SDN, OpenFlow and dynamic network service chaining, which are integral parts of this work. OpenFlow is the basic technology, which is leveraged by the contribution of this work. Quality of Service (QoS) and Quality of Experience (QoE) are both mentioned in the scope of application interaction concepts introduced in Chapter 5 and extended in Chapter 6. The system design described in Chapter 7 introduces a novel approach of implementing a dynamic network service chaining concept by applying concepts from Chapter 4 and 6.

2.1 Software Defined Networking

OpenFlow [68, 63] defines a standardized interface and communication protocol between an OpenFlow-enabled switch and a centralized OpenFlow controller. It is a recent part of the novel Software Defined Networking (SDN) architecture approach to programmable networks. OpenFlow has become the most commonly deployed SDN technology, thus the term SDN and OpenFlow are often considered as synonyms [59]. Yet OpenFlow is only one part of the overall SDN architecture approach.

SDN describes the novel approach to decouple the control plane from the data plane of individual network components. In classical network hardware the control plane is tightly coupled with the data plane. The decision making process of today's control plane is therefore highly distributed. The communication between the different networking devices is handled by standardized distributed network protocols, which make it difficult to change individual parts of the network. This limits the possibilities for innovations inside the networking domain.

SDN is not just an approach to decouple the control plane from the data plane, it is furthermore one of the first approaches towards discovering the better abstractions in the networking domain [90]. In the past the networking domain only tried to master complexity by dividing complexity into defined network layers which highly depend upon each other. SDN rethinks this layering scheme and focuses on developing three main abstractions for distributing state information, simplify configuration and an abstraction of the forwarding model. Ultimately the core belief is that only abstractions can simplify networking programming.

However, developing abstractions is a challenging task and requires research on the requirements of all three aspects. These can finally lead to a better understanding of how a good abstraction of each of these aspects could look like. SDN technology implemented today can help to investigate on the right abstractions by setting up a basis to work on in the research community in order to exploit novel networking setups and evaluate their outcome.

2.1.1 OpenFlow

In the scope of SDN, OpenFlow defines a communication protocol between the central software controller and an OpenFlow-enabled switch. The OpenFlow architecture consists of three main components: (1) An OpenFlow-enabled switch, (2) an OpenFlow controller and (3) a well defined secure

communication protocol between the switch and the controller. OpenFlow defines a flow-based forwarding approach, in which each switch is assigned a set of flow rules stored in a flow table. This flow rule set is evaluated on every incoming packet at the switch. Each flow rule can match a flexible number of different header fields covering the header field of an Ethernet-based TCP/IP network environment. The match fields include header fields from Layer 2 to Layer 4 of the current ISO/OSI Network Layer model. In the future, customized header fields for completely new network protocol are possible to be matched with the same OpenFlow matching scheme. Traffic can be matched against fine-grained matching rules or more general rules using wildcard bits for individual header fields.

Furthermore, each flow rule is assigned with a specific action, which is performed by the switch whenever a packet matches this flow rule. Currently, actions such as forwarding packets to a specific port, dropping packets, modify header fields or encapsulate packets and transfer them to the controller are defined. The last action is primarily used if no flow rule of the switch match incoming packet. In this case the packet is encapsulated in a `packet_in` message by the switch and subsequently forwarded to the controller for further inspection. The switch delays this packet as long as it waits for the controller to answer its `packet_in` message. After inspecting the packet, the controller issues a `flow_mod` message including instructions how to handle the packet and sends it back to the switch. The switch performs the instruction included in the `flow_mod` message and installs the rule also included in this packet into its flow table, in order to be able to process subsequent packets without sending them to the controller again.

2.1.2 SDN Interfaces

The SDN approach opens up new possibilities for an enhanced interaction between network itself and applications using the network [118]. It therefore includes abstractions and interfaces for making the network more aware of applications as well as making applications more network-aware. The interface structure used within the SDN concept is depicted in Figure 2.1. Two central components of this structure leverage the feature of an extended application interaction.

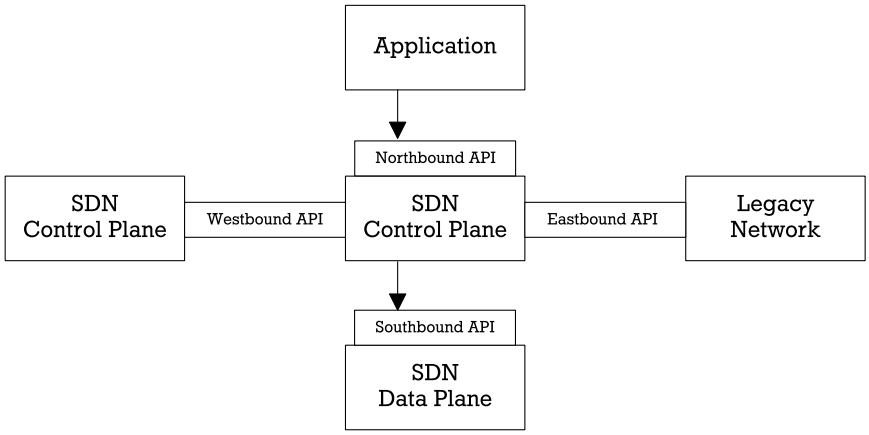

Figure 2.1: Overview of SDN Interfaces [50]

First, providing a global interface for an application requires an open and programmable control layer [36]. The SDN approach enables such an open and programmable control layer by decoupling the control plane from the data plane. This separation makes a central control plane possible instead of having the control plane distributed over each network device. This centralized SDN control plane

provides a so-called Northbound API to be accessed by an application. The Northbound API enables the exchange of information between the network and applications running on top of the network [50]. An application can use this Northbound API to provide beneficial information to the control plane, which can be used to optimize the data plane [49, 118]. For instance, a video streaming application could provide information about the current buffer utilization of a client watching a video. If the buffer of the client is close to run out of video data, the application can inform the controller via the Northbound API about this shortage of video data. The controller receiving this information might perform a path change in the network, where the related video traffic is reallocated to a different forwarding path with more available bandwidth, in order to maintain a sufficient amount of data stored in the buffer.

The second main component is the Southbound API, for which OpenFlow represents one possible implementation. It represents the interface between the data plane and the control plane. In case of the OpenFlow implementation it defines a common instruction set for manipulating the data plane forwarding hardware. The flow abstraction used by OpenFlow enables a flow-based management with novel possibilities for instance in the domain of traffic engineering [49]. Beside the forwarding behavior of the network device, OpenFlow also defines instructions for statistic gathering. Per-flow statistics can be gathered by the controller which can be transformed into a comprehensive snapshot of the current network state. Those statistics can expose congestion or under utilization in the network. Network awareness as described in Section 5.2 relies on sufficient information about the current network state.

Figure 2.1 additionally shows the Westbound API and Eastbound API, which complete the SDN interface structure. The Westbound API is used for interconnecting multiple SDN control planes responsible for different network domains. It enables the exchange of network state information for seamless flow forwarding across multiple domains [50].

The Eastbound API ensures the interoperability between a SDN control plane and a non-SDN control plane, such as a MPLS control plane. The actual implementation of this interface is dependent on the non-SDN control plane and its functionality. However, a properly adjusted implementation can enable a seamless hand over of a flow between these two control planes.

The Northbound and Southbound API in conjunction have the potential to enable an improved network management system exploiting a deeper application interaction. A global view of the distributed network hardware through the Southbound API can be combined with the power of the Northbound API, which enables network-wide operations without the need of dealing with the configuration of individual network devices [70].

2.1.3 Mice and Elephant Flows

The terminology *mice and elephant flows* recently emerged in the context of OpenFlow networks, especially in the scope of data center networks [17, 4, 18]. They are sometimes also referred to as micro and macro flows. Both terms divide the set of occurring flows within a network into two subsets. One subset is given by the mice flows, which can be characterized as follows: Mice flows carry only small amount of traffic within a network and usually only last for a short amount of time compared to their counterpart, the elephant flows. Elephant flows, in comparison, carry a majority of the traffic within the network and therefore also last longer than corresponding mice flows. Categorizing the network flows into these two subsets results in some characteristic of today's networking traffic, especially in data centers: Mice flows usually only lasts a few milliseconds [52], while carrying less than 10 Kbyte of data [3], whereas more than half of the bytes transmitted within a network are carried by elephant flows lasting longer than 25 seconds [52]. Elephant flows, in addition, have a smaller quantity, with only 1% of the flows within the network lasting 200 seconds or longer [52].

The characteristics of mice and elephant flows have some implications on the resource consumption, which will be discussed in Section 3.1.3

2.1.4 Reactive and Proactive Flow Installation

For the flow installation process in OpenFlow, two different strategies can be distinguished. First, the reactive flow installation process, which installs flow rules on demand. Whenever a flow arrives at a switch which does not have a matching flow rule in its flow table, a `packet_in` message is generated by this switch and send to the controller. The controller analyzes this message and calculates a corresponding flow rule matching this particular flow. The controller, afterwards, instructs the switch to install this flow rule, in order to be able to match subsequent packets corresponding to this particular flow. This flow installation process is referred to as reactive flow installation.

Proactive flow installation, in comparison, installs flow rules before the actual traffic matching such a flow rule arrives at the switch. Proactive flow rule therefore installs flow rules in advance, based on certain assumptions about the traffic which might be processed in the future. Proactive flow installation can avoid the delay of the first packet when reactive flow installation is used instead. However, it requires certain knowledge about the traffic characteristics, in order to predict appropriate flow rules.

2.2 Quality of Service and Quality of Experience

Discussing application awareness or network awareness requires an introduction into the terminology Quality of Service (QoS) and Quality of Experience (QoE). Both QoS and QoE are used as the fundamentals of application awareness and network awareness concepts. This section provides a short summary of both terminologies and their role in today's network management systems.

QoS [44] defines the overall performance of the network based on specific parameters measured at the network level. QoS monitors metrics such as the packet loss, throughput and delay in the network. Thresholds for those metrics can be defined as requirements. As long as the network is able to provide the specified QoS requirements, the application is assumed to provide sufficient quality to the user. However, simply relying on network-level metrics for QoS has been reported to be insufficient for a good application performance [49]. Therefore, simply optimizing a network to meet certain QoS requirements does not seem to be satisfactory for all applications.

Instead, the quality rating should be focused on the user's perception of the application. For a video streaming service a user would expect a certain loading time until the video should start playing and a subsequent playback without interrupting stalls of the video. Such expectations of a sufficient application performance cannot be directly expressed using QoS metrics, instead they are expressed using QoE metrics. QoE can be seen as an abstraction of the network quality metrics expressed by the QoS towards a description of the quality of the application perceived by the user. QoE therefore defines the targets for the application quality to be met by the underlying network. This user centric approach comes at the cost of high efforts necessary for a comprehensive QoE evaluation. In order to rate the perceived quality, a real user study is required, which is complex, time-consuming and expensive [108]. This makes QoE almost unfeasible for being evaluated in today´s networks and applications.

Since the basic idea of QoE remains important, QoE is often divided into subjective QoE and objective QoE. Whereas the subjective QoE evaluation requires real user tests, the objective QoE approach tries to evaluate the application quality by modeling media quality [8]. Objective QoE evaluations have the advantage of being evaluated by the system itself. For simplicity, the objective QoE is referred to simply as QoE in the following, acknowledging that this requires a well-evaluated QoE metric

QoE cannot be evaluated purely on the network layer, instead it usually requires an evaluation component running on the end host. This evaluation component measures the QoE achieved at the end host. For a video streaming service this could be the achieved video resolution or amount of continuous playback time.

Because of its benefits in modeling, the target requirements for a sufficient application performance, there is a shift from QoS resource management towards a stronger focus on QoE resource management in the scope of network management [103]. This observation is in line with [49], which notes that future networks need to consider and adapt to QoE demands of various applications, which can not be realized by only relying on QoS metrics.

QoS and QoE are fundamentals for Chapter 5, which discusses novel approaches of application interaction in the scope of SDN.

2.3 Network Services

This section introduces the terminology of network services in the context of a network service chaining system, which can be further extended into a dynamic network service chaining.

2.3.1 Network Service Chaining

Today's ISP networks are faced with an increasing demand for both traditional data connectivity services as well as support for advanced network services, such as firewalls, intrusion detection systems, load balancer, content caches, network address resolution and many more [51]. Such advanced network services are typically implemented using additional hardware middle boxes, deployed at fixed locations within the network or at the edges of ISP core networks [51]. Those middle boxes take traffic from the network users as an input and process the traffic depending on the service specification. Since each service typically is deployed within its own middle box in order to apply multiple services on the users traffic, the traffic needs to be forwarded through a series of middle boxes. Such a series of services is referred to as network service chain [62]. Those network service chains have mostly a static configuration and are fixed wired within the network, making them inflexible and hard to adapt to changing traffic and user requirements [6]. A static configuration and fixed interconnections between different services has scalability issues with the increasing user and traffic demands. The operation of such network service chains becomes increasingly more complex for network operators and especially ISPs, which have to address a growing number of users and even a growing number of different devices per user caused by the popularity of mobile computing devices. Network service chaining is indeed necessary for enabling valuable services in addition to the pure data connectivity in todays networks. Nevertheless the scaling problematic becomes an increasing challenge which attracts more and more interests in the research community.

2.3.2 Dynamic Network Service Chaining

In order to keep the benefits gained from the network service chaining, while addressing the scalability problems, the dynamic network services or dynamic network service chaining is subject to a number of recent publications [62, 6, 117, 51, 61, 75]. All these publications envision a network service chain, which can be easily deployed within the network, meaning that network services can be deployed on demand and chained in an easy reconfigurable order within the network. Along with the ability to

reconfigure the actual order of different network services, automated processes for maintaining and deploying network service chains are a crucial factor in order to scale with the current demand of traffic and user. One of the challenges of dynamically reconfiguring the order of chained network services is imposed in the ossification of IP over transport networks, which hinders the flexible deployment of new network layer functionality, such as novel routing protocols or security services within the network [51].

SDN and in particular OpenFlow can be one solution to overcome the limitations of the traditional network stack in terms of its inflexibility for implementing a dynamic network service chaining. The proposals in [117, 6, 61] investigate the possibilities of OpenFlow for implementing a flexible and dynamically reconfigurable network service chain.

For a more consistent description of the dynamic network service chaining process the terminology of [6] is used in the following. A service chain implements a certain functionality through the application of a series of service functions, in which a service function describes a certain network service functionality such as intrusion detection or load balancing. Each service function is realized by a certain service instance, which applies the service function as a treatment to incoming packets. A service instance can be implemented using soft- and/or hardware devices. This service instances are the building blocks out of which a service chain is crafted, by interconnecting them in a specific order.

OpenFlow can contribute in two ways to the implementation of a dynamic network service chaining. First, its ability to control the flow forwarding behavior of an OpenFlow-enabled network via a central controller can be vital for realizing the reconfigurable path between different service instances. Since the order of service instances needs to adapt to changing user and traffic requirements, the interconnection and forwarding paths between those service instances should be efficiently reconfigurable, especially without the need of changing the configuration of several devices manually.

The second contribution of OpenFlow is based on its ability to forward traffic based on fine-grained flow rules. Network service chains are user or even application specific, meaning that only a subset of the overall traffic has to be forwarded through a certain network service chain. OpenFlow's ability for fine-grained flow matching can implement those user and application-specific forwarding behavior.

In order to benefit from the flexibility OpenFlow provides with its flow-based forwarding scheme, service instances within the network need to be connected to OpenFlow-enabled switches. Since some instance might be implemented in software running on a commodity server or even inside a virtual machine, Open vSwitch can be used to connect those service instances to the rest of the network.

3 OpenFlow Data Plane Analysis

This chapter gives an overview of limitations in the OpenFlow data plane along with corresponding resource optimization approaches found in the research literature. In order to compare these approaches and find similarities, a set of network resources are defined in Table 3.1. The overall set of approaches are compared in Table 3.2

3.1 Limitations in the Data Plane

OpenFlow introduces promising abstraction in the scope of the data plane. Classical network hardware keeps the developer locked out of the inside of their implementation, on both the integrated control plane and the data plane. In the OpenFlow concept packet forwarding is modeled as a flow-based forwarding scheme, which enables the possibility of fine-grained packet processing. This opens up a wide range of applications by applying this fine-grained packet processing rules [56]. Nevertheless fine-grained packet processing needs to be carefully managed, which can be a challenging task in OpenFlow based networks, since fine-grained flow definitions can lead to an explosion in the number of flow entries [55]. The high number of flow entries compared with the ability of updating and maintaining such entries in a flow table structure is the core challenge of the OpenFlow data plane.

3.1.1 TCAM

OpenFlow flow rules can be either exact matching rules or wildcard matching rules. An exact matching rule specifies the complete bit-pattern of the matching fields of a flow (L2 up to L4 fields), whereas a wildcard rule can have wildcards for specific matching fields of a flow. Hardware switches implement wild-card rules using Ternary Content Addressable Memories (TCAM) [87], due to its low access latency and the ability to answer a search request for a specific flow rule in one clock circle.

TCAM is not dedicated to OpenFlow, since classic Ethernet switches also rely on TCAM to store the forwarding entries. But with the extended matching scheme introduced by OpenFlow, it requires potentially more TCAM space than an Ethernet forwarding scheme for two reasons:

Enlarging the TCAM storage of an OpenFlow switch seems to be the easiest solution for dealing with the higher TCAM space demand of OpenFlow. But TCAM space is a highly limited and an expensive resource, even todays high-performance network switches are equipped with a TCAM space in the size of one to two Mbit. TCAM is expensive in both energy consumption and costs. A single one Mbit sized TCAM chip is reported to cost up to 350 USD with a power consumption of 15 Watt [54]. Compared to RAM-based storage it consumes 100 times more more power per Mbit [92] and 50 times more compared to SRAM [54]. Nevertheless even with this high energy demand and costs, TCAM is very important for the forwarding performance of the data plane, since the access latency is 20 times smaller compared to SRAM based storage. Especially in high-performance network environments the latency of the packet forwarding process is crucial and requires to be as low as possible, therefore TCAM is preferable used instead of SRAM for implementing a fast lookup table for flow rules in high performance switches [54].

3.1.2 Flow Table Space

The previous section introduced TCAM as a highly expensive resource to store fast accessible flow tables. The limited storage technology in fact limits the available flow table space, which uses this technology within a switch. For this reason a switch is equipped with multiple storage technologies, such as SRAM and TCAM in conjunction. However, the difference in lookup performance of both storage technologies affects the forwarding speed achievable by the switch. Therefore, the objective of a optimization approach should be to optimize the number of flow rules used throughout the network, in order to fit within the fastest available TCAM storage.

Flow tables in a switch are stored using different technologies, which differ in both lookup-speed and costs. A flow rule can be stored within a switch on one of those storages, such as TCAM, SRAM or within a software table stored in DRAM. The different storage technologies are ordered according to their lookup speed, starting with the TCAM being the fastest technology, followed by SRAM and DRAM. [54] describes a lookup hierarchy. A flow match is first searched within the flow table stored within the TCAM. If no flow rule stored in the table matches, a lookup on the next layer based on SRAM is performed, which has a 20 time higher delay than the initial lookup at the TCAM even using a fast hash based algorithm [21]. A further miss in the SRAM table will cost further delays, which are 5 times longer, since the lookup will be performed on software tables [55]. A miss in the software table will additionally lead to invoking the control plane represented by the central controller, which introduces a delay in order of several milliseconds [18].

The hierarchical order of storing flow tables using different technologies not only affects the performance of the flow miss handling. An incoming `flow_mod` event signaled by the OpenFlow controller needs to install the corresponding flow into the right table. A comparison of different OpenFlow-enabled switches in [42] have shown, that one switch might use the software table to cache an incoming `flow_mod` event only once the hardware table is already full, whereas another switch caches incoming events arriving at a certain speed exciting the update speed of the hardware table. Thus it depends on both the capacity of the hardware table and the number of incoming `flow_mod` events if a rule can be stored directly in the hardware table or needs to be cached in a slower software table first. How fast a rule can be stored in a hardware table also differs across different algorithms used by TCAMs to insert the rules, because they may need to rewrite already stored rules entirely [42].

3.1.3 Flow Update Rate

The flow installation schemes can be categorized in two different groups: Proactive and reactive flow installations, which are introduced in 2.1.4. For a reactive flow installation scheme, the available flow update rate of a switch is crucial, defining the number of flow rules a switch is able to setup per time unit, such as seconds.

The rapid arrival time of new flows, especially in a data center environment, imposes great challenges regarding the flow update rate of today's OpenFlow hardware. A study [52] reported an median inter arrival of new flows being less than 30ms. Approximated 40 servers in a rack connected to a switch will lead to 1300 new flows arriving each second [18]. Each of these flows are only carrying a small amount of data, the median flow only carries <= 10KB [3]. Therefore, the typical scenario in a data center, where a high-performance network is necessary to reach the service requirements, needs a fast and reliable flow management which has to deal with a high flow arrival rate while processing only small amounts of data per flow. Otherwise the data plane performance will suffer from an inefficient

flow management [55]. Assuming the number of flows arriving each second to be 1300 as described above, all of these 1300 new flows could be dropped in the case where the switch runs out of flow table space to store additional flows. In fact, the OpenFlow standard defines a timeout per flow, for the switch to determine when to release an unused flow. Studies with traffic in the Internet have shown, that the average length of a flow is just 20 packets per flow [98], thus the point in time when to release flows from the table for which no packet will arrive any more is a crucial factor. Finding an efficient timeout for individual flows is a challenging task and as it turns out, the standard timeout seems to be often insufficient in terms of cleaning up the flow table space [55].

Invoking the OpenFlow controller can be seen as a three-staged process. In the first stage the packet needs to be transferred from the data-plane into the management CPU of the switch. In the second stage, the packet needs to be encapsulated and send over the secure channel as a `packet_in` message to the OpenFlow controller. The controller subsequently constructs a matching flow rule appropriate for the packet and signals this flow rule back to the switch with a `flow_mod` message. The switch receiving this `flow_mod` message installs the new rule into its flow table and then either forwards the packet or drops it according to the action associated given in the `flow_mod` message. The corresponding stages have different bandwidth limitations as described in the following section. The stage with the lowest bandwidth will limit the overall switch to controller bandwidth in the following referred to as the control bandwidth.

Data Plane Control Bandwidth:

In order to encapsulate a packet and send it to the controller, a switch requires to transfer the packet from the data plane hardware, which is usually realized as an ASIC to the management CPU. This process is highly limited by the processing power of today's internal management CPUs and the low bus speed between the management CPU and the ASICs implementing the data plane [74, 18]. The consequences and the scale of these limitations are presented for the HP 5406zl in [18]. The HP switch is advertised for being used as an enterprise-edge, small business or branch office core deployment [1]. It is equipped with an ASIC on each multi-port line card and with a single management CPU for providing OpenFlow related tasks. According to[18], the HP switch is a representative of the current generation of OpenFlow-enabled Ethernet switches, therefore limitations discovered with this hardware are representative for the limitations all current OpenFlow-enabled Ethernet switches will face.

For the HP 5406zl the bandwidth between the data plane ASIC hardware and the management CPU was measured to be only 80 Mbit/s. Compared to 300 Gbit/s processing speed of the ASIC in the same switch, this is a four-order-of-magnitude difference. Another study supports this four-order-of-magnitude difference [12] between the forwarding speed capability and the bandwidth between ASIC hardware and management CPU. Additional tests with an hardware switch running the Open vSwitch software stack measured the same bandwidth to be only 17 to 24 Mbit/s [74].

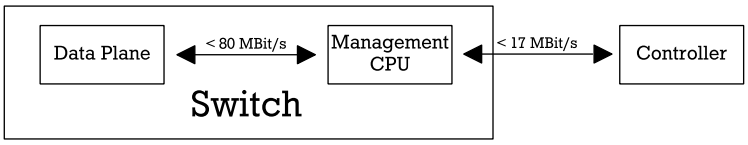

Figure 3.1: Control Bandwidth

Controller Bandwidth:

After transferring the packet from the ASIC to the management CPU, the packet is encapsulated by the relatively slow management CPU to be forwarded to the controller. Measurements with the HP5406zl have shown the available bandwidth between the switch and OpenFlow controller to be just 17 Mbit/s [18]. In addition, the tests in [74] measured the bandwidth to be only 610 kbit/s.

Both cases show that the bandwidth for transferring packets from the switch to the controller is highly limited. This bandwidth therefore limits the overall control bandwidth as it is depicted in Figure 3.1. The switch internal management CPU can be identified as being mainly responsible for the low control bandwidth. The limitations in the control bandwidth limit the overall flow setup process in OpenFlow, which is an essential part in the reactive flow installation process, because the controller needs to be invoked for the first packet of every unknown flow.

Controller Latency:

Additionally, the packet needs to travel from the switch to the controller and reverse, which introduces a delay equal to the round trip time between the switch and the controller. This additional delay plus the limitations of the control bandwidth sum up to a flow-setup latency required for each flow being reactively installed by the controller. This flow-setup latency can be too high for the requirements of certain high-performance networks [18]. Higher control bandwidth can be achieved using OpenFlow switch implementations on commodity server hardware [24], since such hardware can handle the control plane related overhead with their powerful multi-core CPU, but on the other side lacks of a fast data-plane implemented in hardware to forward packets at line speed such as the HP5406zl. Network processors are becoming increasingly popular, since they combine both the fast processing power of multi-core CPUs with forwarding specific hardware adaptions. However, they are not competitive on speed and costs yet [12].

3.2 Data Plane Resources

The previous Section 3.1 has investigated the limitations of the current available OpenFlow-enabled data plane implementations. As a next step towards an optimization approach for dealing with these limitations, this Section defines the affected resources, listed in Table 3.1, as a basis to further discuss improvements and optimizations.

Most of the research reviewed for this work focused on improving a specific resource limitations in OpenFlow environments. In the following section, these approaches found in the literature are collected and explored regarding their optimized resources. Therefore, a set of resources are defined, which are listed in Table 3.1.

3.2.1 Control Bandwidth

The control bandwidth defines the available bandwidth between an OpenFlow switch and the OpenFlow Controller. The term bandwidth in this resource is not limited to the pure bandwidth as the commonly used network metric. This data plane resource is a combination of the delay, throughput and bandwidth between an OpenFlow switch and the OpenFlow controller. The three metrics are combined to evaluate their impact on the data plane performance as a whole.

The control bandwidth is crucial when it comes to the flow setup process, since the switch needs to invoke the OpenFlow controller for setting up flow rules for unmatched packets. A smaller bandwidth

Scope	Resource Name	Description
Device	Control Bandwidth	Bandwidth between switch and controller available for control traffic
	Flow Table Size	Capacity of the device internal flow table to store flow rules
Network	Flow Rule Capacity	Specifies the overall flow rule capacity of the combined switches present in a network.
	Forwarding Capacity	Number of Flows which can be forwarded by the overall network at most

Table 3.1: Data Plane Resources

will lead to a higher delay in the flow setup process. This delay continuously affects the data plane performance, because the affected packet needs to be buffered until the controller sends the corresponding `flow_mod` message. The reactive flow setup relies on this bandwidth in two ways, first it needs to send the packet from the switch to the controller and afterwards it has to wait for the `flow_mod` message send back from the controller. If reactive flow setup is used extensively, the control bandwidth can become a bottleneck for the data plane performance. Also, the proactive flow setup and flow modifications issued by the controller are affected by the control bandwidth, but compared to the reactive scheme the control bandwidth only limits the way from the controller to the switch, since the switch does not encapsulate packets in the proactive flow setup.

3.2.2 Flow Table Size

The flow table size of each individual switch is another important data plane resource. The flow table size defines the number of flow rule entries a switch is capable of storing, regardless of the concrete storage technologies such as TCAM, SRAM or DRAM.

The flow table itself is a central element in the OpenFlow approach to implement the flow-based forwarding scheme. Each incoming packet requires to be matched by a flow rule stored in the flow table in order to be processed by the switch. The OpenFlow controller is responsible for constructing these flow rules and it pushes these flow rules onto the OpenFlow switches in the network. These flow rules can only be installed in the switch if there is flow table capacity left.

If a switch runs out of flow table capacity, it has to either drop packets for which it cannot store matching flow rules any more or it has to wait for stored flow rules to timeout in order to be deleted from the flow table. Both ways have a negative impact on the data plane performance, since dropped packets need to be retransmitted by the transport layer until they can be successfully forwarded by the switch. A switch waiting for flow rules to timeout is likely to miss timeouts of the buffered packets. These packets also need to be retransmitted by the transport layer, which lead to additional network load affecting the overall data plane performance.

3.2.3 Flow Rule Capacity

A network is a union of multiple connected switching devices. The OpenFlow abstraction commonly assumes each switch device to be equally treated by the controller. Nevertheless, the hardware capabilities

of different vendor switches differ from each other [42]. A hardware based OpenFlow switch usually has a higher forwarding speed due to its fast accessible but highly limited TCAM space, whereas a software based Open vSwitch on commodity hardware can provide moderate forwarding speed but has access to a larger but slower memory for storing flow rules. The flow rule capacity resource specifies the network wide summed up flow rule capacity of all switching devices. In order to exploit an optimal usage of the flow rule capacity, the controller needs to be aware of the different capabilities of a switch. Instead of only utilizing the full capacity of some single switches, flow rules could be distributed across multiple switches [114] or even a flow table itself can be distributed over the network [53]. This way flow rule space can be shared across the network, while sharing the traffic across multiple devices as well. If the flow rule capacity is not taken into account by the controller, some switches might run out of flow table space, whereas other switches have capacities left for storing additional flow rules. Compared to the previous two resources, which are more centric to the specific network device, the flow rule capacity addresses the overall network.

3.2.4 Forwarding Capacity

The forwarding capacity is the last resource defined as a basis for further resource optimization discussions. It shares the network wide scope with the flow rule capacity resource. Therefore, it takes the overall group of network devices into account. It focuses on the utilization of the links, which interconnect individual switches with each other and their corresponding link speeds. Optimizing only device specific resources like control bandwidth and flow table space is possible, for instance by installing wildcard rules matching the majority of the traffic proactively across the switches. Nevertheless, this approach lacks of flexible and fine-grained flow management. In addition, such small and fixed flow rule sets are likely to use only a small set of the available links inside a network. Therefore, the forwarding performance achievable with such a small and fixed flow rule set is limited by the number of links in use and their available link speeds. Instead of such an one-sided resource optimization, the forwarding capacity as a network wide resource should be considered. In an evaluation step of a concrete optimization approach, this resource can help to evaluate the impacts on the performance of such an approach taking the overall network into account. Two results would be desirable as an outcome of such an evaluation.

First, an approach might provide similar or even a higher network performance, while using less data plane resources, such as flow table space or control bandwidth, which would result in a larger forwarding capacity. The well known IP-multicast approach can be seen as an example, where the forwarding capacity resource is optimized, leading to a increased forwarding performance, without the need of additional data plane resources.

Second, an approach might increase the network performance, but with the need of additional data plane resources, meaning that additional switches or switches with higher data plane resources are necessary. In this scenario it can be evaluated how much performance gain the approach is able to generate in comparison to the additional data plane resources required. In this scenario additional data plane resources in terms of hardware should not be simply added to overcome limitations, instead limitations should be tried to resolve by optimizing individual data plane resources. Only if further optimization is either to costly in terms of computational effort or not achievable by state of the art concepts, additional hardware should be considered as necessary.

3.3 Optimizing Data Plane Resources

This section investigates an overview of different approaches studied in the literature, which address the optimization of a certain data plane resource. The different approaches are listed in the Table 3.2. The table lists data plane optimization approaches in the context of OpenFlow and the classic networking context for comparison reasons and because optimizations in the context of prior network technology can have beneficial approaches for the OpenFlow context as well. For each OpenFlow approach the table lists if the approach is fully compliant with the current OpenFlow standard. If this is not the case, the table lists if additional add-ons in software or hardware are necessary for implementing the approach. The Split SDN Data Plane approach [74] for instance adds a hardware subsystem to the switch in order to improve the control bandwidth capabilities of a switch. DevoFlow [18] is an approach which requires additional software features which enable a more flexible forwarding scheme.

Each approach was evaluated in different networking scenarios. The four common networking scenarios, data center network, university network, internet service provider and internet exchange point are distinguished and the table indicates which scenario was included during the evaluation of each approach.

The last column group lists the resources which are subject to the optimization concepts. The resources are distinguished based on the resources introduced in Section 3.2 This overview is the basis for the development of general optimization strategies for the data plane.

3.3.1 Optimizing the Control Bandwidth

Table 3.2 shows four approaches optimizing the control bandwidth, namely DevoFlow, Split SDN Data Plane, DIFANE and SwitchReduce. All of them focus on optimizing the control bandwidth in an OpenFlow context.

OpenFlow context

Dealing with a limited control bandwidth needs to be rethought in the OpenFlow context, since the control plane is moved from the individual switch to a centralized controller. One of the benefits of this approach is the reduction of complexity of a distributed control plane and it furthermore enables easier networking programmability on top of the central controller. The downside of this centralized control plane in conjunction with a reactive flow installation scheme is the additional delay caused by the process of encapsulating packets at the switch and sending them to the controller.

The three approaches: DevoFlow, DIFANE and Split SDN Data Plane address this issue in OpenFlow in two different ways. DevoFlow and DIFANE try to invoke the central controller less often to optimize the usage of the limited control bandwidth. As described in Section 3.1.3, invoking the controller in each flow setup process is expensive in terms of additional workload imposed on the switch and the delay caused by sending each unknown packet to the controller for further inspection. Without the need of a larger control bandwidth, its usage can be optimized. The controller could be used less frequently, for instance by offloading decisions back to the local switch.

Flows can be categorized as mice and elephant flows (Section 2.1.3). Each flow category introduces different challenges in the scope of the control bandwidth usage. DevoFlow and DIFANE handle these challenges in different ways: The DevoFlow approach can process the usually higher quantity of mice flows, especially in data center environments [52], locally on the switch rather than invoke the central controller for each flow setup. For the elephant flows, which carry more traffic than the mice flows,

Approach	Context		OpenFlow			Scenario				Resource			
	OpenFlow	Classic Networking	OpenFlow compliant	Software Add-on	Hardware Add-on	Data Center Network	University Network	Internet Service Provider	Internet Exchange Point	Control Bandwidth	Flow Table Space	Flow Rule Capacity	Forwarding Capacity
DevoFlow [18]	√		NO	√		√				√			√
FlowMaster [55]	√		NO		√		√				√		
Compact TCAM [54]	√		YES			√	√				√		
Split SDN Data Plane [74]	√		NO		√					√			
DIFANE [114]	√		NO	√		√	√	√		√		√	√
CacheFlow [56]	√		YES				√		√		√	√	
Labelcast [94]	√		NO		√						√		
SwitchReduce [45]	√		YES			√	√			√	√		
Hierarchical Switch Network [91]	√		YES			√					√		
SlickFlow [83]	√		YES			√				√	√		
MPLS		√	-								√		
Flow Label for ATM [76]		√	-						√		√		√
IP Multicast		√	-										√

Table 3.2: Data Plane Optimization Approaches

DevoFlow keeps the possibility to manage these flows by the central controller in order to optimize their forwarding path through the network for a better link utilization.

This local processing in DevoFlow requires additional actions like rule-cloning, multipath and re-routing. Rule-cloning describes the ability of the switch to clone wildcard flow rules usually dedicated to an elephant-flow, in order to handle a subset of the match as a mice flow, without requiring the controller to setup such a flow rule. Multipath support enables the switch to choose from several output ports to be assigned to a new cloned flow rule. The choice enables the switch the possibility to distribute traffic over multiple ports, for a better overall link utilization. The additional re-routing scheme is mainly focused on enabling a fast local fallback decision, in the case of an output port failure. The switch can decide locally which backup flow rule should be matched.

DIFANE uses another approach to reach a similar goal. It introduces authority switches as a second group of switches. The controller distributes flow rules over these authority switches. Instead of invoking the controller in case of a flow miss in a non-authority switch, the switch forwards the traffic to an authority switch. The authority switch, which has a special rule set assigned by the controller, can forward the traffic instead to the right direction and furthermore computes a rule matching this specific packet to be installed in the switch, which forwarded the traffic initially to it. This requires either additional functionality in current switches, or the authority switch can inform the controller instead, which can compute and install the cache flow rule. In both cases the delay does not affect the forwarding of the current packet since the authority switch can already forward the packet to the destination.

Both approaches have in common that they try to keep the traffic inside the faster data plane forwarding hardware and try to avoid the invocation of the central controller. Therefore, control bandwidth is preserved. Even forwarding the traffic on a longer path can be much faster and therefore more efficient than invoking the controller [114].

Another promising approach for preserving control bandwidth is SwitchReduce. SwitchReduce mainly introduces a way of exploiting capabilities of label switching known from concepts such as the widely adopted Multi-Protocol Label Switching (MPLS) in the context of OpenFlow, but also has benefits in regarding an optimized usage of the control bandwidth. For a better understanding the SwitchReduce approach can be shortly described as follows: SwitchReduce introduces a source routing approach, such that it includes the complete forwarding information for a packet into a label, which is assigned to a packet at the ingress of the network. It therefore includes a list of VLAN IDs with the size equal to the number of hops required for the packet to reach its destination. The necessary global information about the complete path for a specific packet is provided by the controller, which has a global network view to calculate such paths. The ingress switch stores an exact matching rule in its flow table associated with the VLAN IDs to be pushed to a packet matching this rule. Each intermediate switch only stores one wildcard rule for each of its outgoing link, reducing the space requirements for its flow table dramatically. The controller can assign these fixed wild card rules proactively in the absence of any previously matched traffic. On the arrival of a packet at an intermediate switch, the switch checks the first VLAN ID stored in the label, which tells the switch on which outgoing port it should forward the packet. Before forwarding the packet, the switch removes the first VLAN ID from the list of IDs. The next switch performs the same steps until the packet reaches its egress switch. The egress switch of a flow removes the last VLAN ID, and forwards the traffic according to the last VLAN ID. The labeling process is transparent for both end hosts, since all labeling information is removed from the packet at this point.

Following this approach, SwitchReduce preserves control bandwidth, since all intermediate switches can be assigned with fixed wild card rules in a proactive installation scheme and therefore do not require any subsequent interaction with the controller. Only ingress switches require to store exact matching rules along with the list of VLAN IDs, which should be included into the label L for matching packets.

If the controller decides to change a path of a flow, it only needs to update the exact matching rule in the ingress switch, providing a different list of VLAN IDs. This will force the packet to be forwarded on a different path, without changing a single rule in an intermediate switch. Flow statistics can still be gathered, because each flow is represented by at least one exact matching rule in an ingress switch or egress switch. The controller can monitor congested links by using the flow statistics of the wildcard rules in intermediate switches. Obtaining the specific flow which causes a congestion can be achieved by gathering the flow statistics of the exact match rules stored in the ingress switches.

A similar approach to SwitchReduce is SlickFlow presented in [83]. SlickFlow focuses on providing an efficient way to recover from link failures, which can be especially crucial in a data center environment, where application jobs usually have a relatively strict execution time frame [2]. In case of a link failure it is important for the network to provide fast recovery functionality, where traffic can get rerouted on a alternative path. Involving the controller for requesting an alternative path might introduce a delay which is not feasible for time constraint traffic. SlickFlow exploits a source based routing approach similar to the one used by SwitchReduce, where all forwarding information is carried in the header of each packet. In addition to just one forwarding path, packets in the SlickFlow approach carry an alternative path, which is utilized if a switch on the original path experience a failure, causing the path to become invalid. This additional information carried within the packet header makes an quick switch to failover path possible, while preserving control bandwidth. Advantages in preserving flow table space due to the source based routing approach are carried over from the SwitchReduce approach.

The Split SDN Data Plane described in [74] also tackles the limitations of the control bandwidth, but instead of changes in the flow rule distribution the authors focusing on increasing the control bandwidth itself. The control bandwidth was increased from as low as 324 kbit/s to 275 Mbit/s. This increase was achieved by extending a switch with an enhanced subsystem to handle the CPU intensive tasks, such as encapsulating packets for sending them to the controller and moving packets from the data plane into the software stack of the switch. The additional CPU power in the switch can also be used to implement additional features directly inside the data plane, which can enable a more flexible data plane for future developments. A downside of such a subsystem is its complexity in both hardware and software, which makes it challenging to be integrated into current OpenFlow-enabled hardware. It is therefore not a pure optimization within the current boundaries of OpenFlow hardware, but more a start of rethinking the current hardware basis.

3.3.2 Optimizing the Flow Table Space

Section 3.2.2 describes the limitation current switches face in terms of their limited flow table space capabilities. This is an important challenge to be taken into account, in particular because the flexible fine-grained forwarding capabilities of OpenFlow rely on sufficient flow table space.

Table 3.2 lists in total seven approaches which optimize the flow table space resource. Each approach is explained in more detail in the following, starting with optimization approach, which are developed even before the introduction of OpenFlow, since space requirements for storing the forwarding information can be a issue in ethernet hardware as well.

Classic Networking context

Reducing the size of the forwarding state already attracted attention of the research community, before OpenFlow was introduced. An approach for using connectionless IP networking on top of connectionless ATM switching hardware introduced a labeling scheme, which was adopted by todays MPLS [19] and approaches in the OpenFlow context. The labeling approach referred to as „Flow Label for ATM“ focused

on the ability to identify IP flows which could be forwarding using the faster ATM switching hardware, rather than relying on a more complex IP-prefix matching for determine the forwarding destination. Therefore, IP flows with a certain duration are forwarded using the connection oriented ATM switching hardware. Each switch individually requires to lookup whether an incoming packet belongs to such a flow or not. Therefore, this approach introduces labels managed across a predefined path. The labels are negotiated between adjacent switches, in a way that each switch manages its unique label space. The ingress switch starts to push a label onto the packet belonging to an IP flow. This label is known by the switch at the next hop, such that the packet can be recognized to be part of this flow which should be switched using the ATM hardware. Since the labels are not globally unique, each switch swaps the label with another label, known by the switch at the next hop. This process continues until the packet reaches the egress switch. The egress switch removes all labels from the switch to make the process transparent for the end hosts. Beside the increased forwarding speed caused by the ATM hardware, the flow labels save valuable flow table space and simplify the matching process for flows which can be forwarded by their label information rather than by their IP-prefix destination.

Such a labeling scheme used in Flow Label for ATM is also adopted in the MPLS approach, which is widely used in today's IP networks. It leverages the same benefits in terms of a simplified matching scheme and storage savings for maintaining a certain forwarding state. MPLS therefore uses labels which get pushed onto the packets to be processed by the next switch along a predefined path. Pure label switched intermediate switches only manage unique labels stored in a forwarding table along with the right outgoing port to be used for forwarding the packet. This shrinks down the space requirements for each individual entry, resulting in an overall smaller forwarding table.

SDN context

The benefits of MPLS in terms of reducing the storage requirements of the forwarding state can be transferred into the SDN context. The two approaches LabelCast and Compact TCAM have identified the potential in optimizing the space requirements of an OpenFlow rule especially because of the limitations caused by the expensive TCAM storage.

Simplify Flow Rule Matching

LabelCast proposes an additional separation of the OpenFlow data plane, into a label plane and the cast plane. While, the label plane borrows the label concept from MPLS, which simplifies and accelerates packet forwarding processing, the cast plane provides an abstraction for computation and storage resources. The label plane is designed to simplify the forwarding hardware implementation in addition to an easier packet matching only relying on a label rather than the more complex header field matching. The mapping between a concrete flow and a label known by the switching hardware is maintained by the controller. The simplified matching also reduces the storage requirements for individual flows, since the switch only stores a 3-tuple <`Label, Instruction, ServiceID`, where the label identifies the flow, the instruction field stores the actions the switch performs with the packet and the ServiceID, which is used by the cast plane to identify further computations regarding the packet. The cast plane is not implemented directly in hardware, instead it is realized using a multicore CPU and software. Overall, the abstraction introduced in LabelCast simplifies the forwarding hardware while keeping additional functionality on the cast plane using the general purpose multicore CPU.

Compact TCAM is another approach exploiting the possibility of reducing the flow table storage requirements by introducing a Flow ID comparable to the LabelCast approach. In comparison to LabelCast the Compact TCAM approach can be implemented in a OpenFlow compliant way. It also uses the central controller to maintain the mapping between a Flow ID and the corresponding flow. Therefore, it exploits

the OpenFlow handling of a table miss, where the first packet is sent to the controller for inspection. The controller assigns a unique Flow ID for this packet and stores it in a local mapping table. It then sends the assigned Flow ID back to the switch including the action to perform with the packet. The complete network view maintained by the controller is used to send a message to all switches on the path corresponding to this Flow ID. This message contains the Flow ID to be stored in the rule table by the switch. On incoming packets these switches just need to extract the Flow ID, stored in the packet by the initial ingress switch, and match it against the local stored Flow IDs. This both simplifies the matching process and preserves flow table space. The egress switch is instructed by the controller to remove this additional header storing the Flow ID from the packet to make the process transparent for the end hosts in the same way MPLS performs label switching.

SwitchReduce (Section 3.3.1) also reduces the flow table space requirements. The intermediate switches only store a highly reduced flow rule set, because they only need to store one wildcard rule for each of their outgoing links, assuming that they only implement simple forwarding actions. Additionally, the flow rule set is fixed and can be assigned proactively by the controller. The flow table space requirements for an ingress switch are higher, since it stores a high number of exact match rules for each incoming flow.

The four approaches LabelCast, Compact TCAM, SwitchReduce and SlickFlow exploit the possibility to replace complex rule matching with a simpler and flow table space preserving label switching approach similar to the label switching used in today's MPLS enabled networks. MPLS is widely adopted in the network community and the extensive studies in the deployment of MPLS makes such a labeling approach an promising approach for reducing the flow table space requirements of OpenFlow as well. This encourages the fact that label based forwarding can have a significant impact on the flow table space requirements as well.

3.3.3 Optimizing the Forwarding Capacity

The three approaches DIFANE, DevoFlow and "Flow Label for ATM" are reported to achieve optimizations of the forwarding capacity. DIFANE is able to handle a flow setup rate of 75 000 flows per second, where as, in comparison, an architecture based on the NOX controller [38] only achieved 20 000 flow setups per second. The usage of authority switches supported a linear increase of throughput with the number of authority switches.

The DevoFlow approach also reports an improved throughput by up to 32% compared to other routing algorithms such as ECMP [39]. Similar to those throughput increases, the "Flow Label for ATM" achieved approximately 4.5 more traffic than without applying a label based approach. This clearly supports the assumption, that optimizing certain resources in the network can result in an optimized forwarding capacity. Having a higher flow setup rate can ultimately results in a higher forwarding capacity, since the network is able to setup more flows per time unit, which implies it can forward traffic corresponding to those flow rules.

IP multicast is one example of an approach which optimized the forwarding capacity. Instead of an increasing usage of IP unicast for forwarding packets from one host to multiple destinations, IP multicast leverages duplication functionality of network devices on the path to each destination, in order to avoid redundant duplications of the same packet. As a result, it requires less network resources for transmitting a packet from one source to multiple destinations compared to a unicast transmission.

4 Classes of Resource Optimization Concepts

This chapter introduces classes of resource optimization concepts, which are based on the studied approaches described in Chapter 3. Similarities identified at these approaches are transformed into classes of resource optimization concepts, similar to software design patterns widely adopted in the programming community. A class of a resource optimization concepts describes a pattern, which can be applied in a general fashion optimizing a certain set of resources.

Table 6.1 gives an outlook to the four developed resource optimization concepts. Each approach from the literature analysis, which is based on one or multiple of those approaches is listed along with the marker at the corresponding resource optimization concept. Each concept is described in more detail in the following sections.

	Hierarchical Switch Topology	Relevant Flow Awareness	Label Switching	Forwarding Information Aggregation
DIFANE [114]	√			
CacheFlow [56]	√			
Hierarchical Switch Network [91]	√			
DevoFlow [49]		√		
Flow Label for ATM [76]		√	√	√
Mahout [17]		√		
MPLS			√	√
SwitchReduce [45]			√	√
Compact TCAM [54]			√	√

Table 4.1: Occurrence of Resource Optimization Concepts within the Literature

4.1 Hierarchical Switch Topology

Constraints and Objectives

The concept of an hierarchical switch topology addresses two limitations of the data plane resources described in Section 3.2. Today's OpenFlow hardware faces a limited amount of flow table space. As a consequence, a switch might run out of available flow table space while processing newly incoming

flows. Furthermore, the control bandwidth was identified in Section 3.2.1 as being a possible bottleneck in the interaction between the switch and the central controller, especially for a reactive flow setup. Therefore, the objective of this concept is to increase the usable flow table space in a scalable way to meet requirements given by the network environment. A requirement for instance could be a large flow table capacity at an ingress switch to store an extensive fine-grained rule set for access-control purposes. Having the ability to store an increased number of flow rules can additionally preserve control bandwidth, since the controller needs to be invoked less often, because it is more likely to find a matching flow rule in a larger flow rule set.

Structure

For each incoming packet a switch performs a lookup in its flow table to match the incoming packet against its stored flow rules. If no such flow rule exists, the switch is required to sent a `packet_in` message to the controller. Such delay caused by the controller interaction can be avoided if a matching rule is already installed in the switch, which is often not possible because of the limited flow table space. Having a software switch running on commodity hardware would lower this limitation, since a software switch can provide more flow table space, but this comes at a cost. A software switch provides less forwarding speed compared to a hardware based switch. Nevertheless, such diverse capabilities could have advantages if they could be usefully combined.

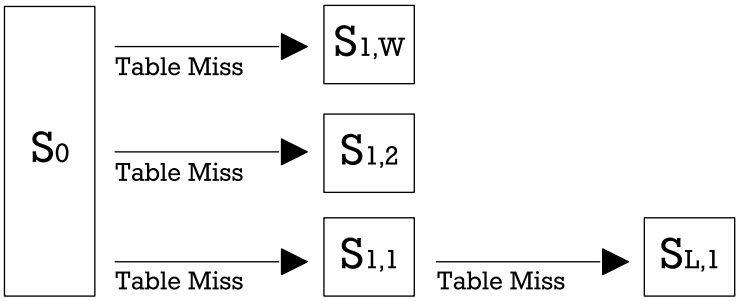

Figure 4.1: Hierarchical Switch Topology

The concept of an hierarchical switch topology aims to combine these diverse capabilities. Such a topology is depicted in Figure 4.1. The topology is called a hierarchical topology, since it is based on a sequence of switches. In this hierarchy switch S_0 represents an ingress switch. Assuming an incoming packet arriving at switch S_0, the switch would perform a lookup in its own flow table. For an unknown packet no flow rule would match the packet header and therefore a flow miss occurs. Instead of forwarding the packet to the controller, the switch forwards the packet to one or multiple switches, labeled in Figure 4.1 as switch $S_{1,1}$ to $S_{1,W}$. These switches act like a flow rule cache for the switch S_0, performing a lookup for the forwarded packet in their own flow table. Additional flow misses can be handled either by forwarding the packet to the controller or to an additional switch $S_{L,1}$, depending on the length L of the caching path. The hierarchy can have L number of switches in a row performing flow rule lookups sequentially, whereas W number of switches can perform a lookup in parallel. In case of matching a flow in any of the switches, the packet is forwarded by this switch according to the action associated with the flow rule. The controller is aware of the switch topology and can therefore assign appropriate rules to the different hierarchical layers. Delay-sensitive flows should be matched by a flow rule in the lower layers of the hierarchy, in an optimal case directly at switch S_0, whereas rules for matching flows which are less delay-sensitive can be placed in a higher layer.

Such a hierarchy has two advantages. First, it can preserve control bandwidth, since the packet is likely to match a rule in one of the switch layers, because more potential flow rules can be stored within flow tables of the multiple switches participating in the hierarchy. And second, the flow table capacities of the switches forming the topology can be combined regarding their flow table capacity and forwarding speeds. Furthermore, it is flexible regarding the throughput and flow table space requirements. The topology is defined by the two dimensions W and L. W represents the number of switches forming one layer and L represents the number of switches in a row.

Requirements

The hierarchical switch topology concept requires a comprehensive view of the network topology. A central entity requires knowledge about each switch participating in the network and all the connections between these switches. Furthermore, information regarding the concrete hardware capabilities of each individual switch are desirable. This knowledge is combined into a complete network topology map, which is used as an input parameter for a partitioning algorithm. This algorithm is necessary in order to divide a given flow rule set into subsets, which can be allocated to individual switches depending on their position in the network and their concrete hardware capabilities, such as flow table capacity. The input parameter of this algorithm are the network topology map and the flow rule set which should be installed in the network. Moreover, each flow rule contained in the flow rule set should have a weight parameter, to be evaluated by the partitioning algorithm. If the weight is associated with delay sensitivity, the algorithm should aim to maximize the total weight of the flow rules assigned to the lowest switch layer.

Applications

The concept of an hierarchical switch topology is used in DIFANE, CacheFlow and in the Hierarchical Switch Network. DIFANE uses one additional switch called an *authority switch*. The controller allocates a given rule set to the *authority switches*. The rules stored at an *authority switch* are mostly coarse-grained wildcard rules to match a high percentage of the arriving traffic. The ingress switch will forward any packets which can not be matched at the ingress switch to an *authority switch* to be matched by one of these wildcard rules. The *authority switch* can forward the packet in case of a match to the right egress switch. The concept used by DIFANE can therefore be described as an hierarchical switch topology with the dimensions $W = 1$ and $L = 1$. DIFANE was evaluated using a variety of different network topologies in comparison with the NOX controller [38]. In this comparison DIFANE achieved a small delay for the first packet of only 0.4 ms round-trip time (RTT), whereas the first packet with the NOX controller involved in the flow setup experienced 10 ms RTT. Even when looking at the delay for a flow length of 35 packets the average packet delay in DIFANE is reasonably smaller with 0.3ms compared to 0.58ms in the NOX architecture. The reduced controller interaction yield to a reduced flow setup delay.

CacheFlow is also based on a hierarchical switch topology. In this approach a single hardware switch can be connected to one or multiple additional switches. In case of a flow miss at the hardware switch the packet is forwarded to these additional switches for further matching. CacheFlow defines the topology to be elastic, therefore the dimensions W and L are variable. CacheFlow was evaluated with a synthetic access control list using the ClassBench [96] packet classification benchmark. The access control list consisted of 10 000 flow rule entries. The hardware switch in the CacheFlow approach stored 500 rule entries of the total rule entries. This subset was calculated using a special cache algorithm in order to select the most likely flow rules for matching the majority of flows. With this subset of 500 rules CacheFlow was able to perform a hit rate of around 90% for all incoming flows. With a subset of only 100 rules stored in the TCAM the approach was still able to achieve a hit rate of 87%.

Tradeoffs

In such a topology different tradeoffs have to be made with different implications regarding path length, flow delay and forwarding capacity. The controller requires an algorithm to calculate a proper flow rule partitioning of any given rule set for assigning the different flow rules to the switches participating in the topology. Rules can thereby be allocated in lower layers or even at the switch S_0 or at higher layers of the topology. This affects the path length of a flow, since a packet might have to traverse multiple switches until it matches a flow rule installed in a higher layer. Flow rules in lower layers would require less switches to be involved in the flow matching, which results in a shorter path length and less overall flow delay. Ideally, all necessary flow rules would get installed in switch S_0, but this will likely overflow its flow table space. Therefore a tradeoff in terms of flow allocation has to be made. Rules for matching delay-sensitive traffic or throughput-sensitive can be set to be more important and therefore will be installed in lower layers of the topology, whereas less important flow rules can be installed in the higher layers.

Another tradeoff is influenced by the dimensions W and L. More than one switch per layer ($W > 1$) gives the system a higher rule capacity and a smaller delay, while requiring additional ports to be occupied at switch S_0 in case of adjacent switches. More than one additional switch in a row ($L > 1$) gives the system an increased rule capacity, but flows might need to be forwarded over more switches in a row, which increases the flow delay, due to an increased path length. But compared to more switches per layer less direct connections for adjacent switches are necessary to build such a topology. Given the fact that switches at a higher layer might be implemented by a software switch, additional switches in the dimension L would limit the possible forwarding capacity, since the number of switches a packet has to pass increases with additional layers, in addition software switches usually are less powerful in terms of forwarding speed.

The topology can be build from different switches, for a switch S_0 a hardware switch using TCAM to provide a fast forwarding speed is desirable, whereas for higher layers software switches can provide a larger flow table capacity. Therefore diverse switches can be placed at a different position in such a topology to exploit their unique capabilities.

4.2 Relevant Flow Awareness

Constraints and Objectives

The flow based forwarding scheme used by OpenFlow defines a flow setup scheme with the interaction of the central controller to install appropriate flow rules in the right switches. In general, all flows are equal in this scheme and a flow miss always requires the invocation of the controller. A full flow-by-flow control in OpenFlow generates a lot of control traffic facing the controller [18]. Even before OpenFlow increasingly gained attention, a study related to ATM technology [76] argued, that flows in a network arrive too quickly to compute a route for each individual flow. Therefore, a concept is necessary to deal with the challenge of flow-by-flow control in an OpenFlow context, where flow table space to store sufficient rule sets is limited and the control bandwidth involved in the flow setup can become a bottleneck.

The concept of relevant flow awareness addresses these challenges by making the network aware of different flow characteristics. The concept is inspired by the term elephant flows, but instead of limiting this flow classification to elephant flow accountable for the majority of traffic, the flows are referred to as relevant flows. Since flows can differ in their duration, the amount of packets and the number of bytes carried by each flow. One common concept is the classification of flows into two groups referred to as mice and relevant flows (Section 2.1.3), which are treated differently in the network. The objective

of this concept is to increase the forwarding capacity of the network by allocating relevant flows to less congested paths. Therefore the controller on the one hand is required to be aware of the important relevant flows in the network to be able to allocate them to the less congested path, while on the other hand the switches should be able to handle mice flows without informing the controller for setting up each of those mice flows.

Structure

Since flows can differ in their amount of packets, their duration or their amount of data transferred per packet, it can have benefits to treat them differently in the flow handling. Flows are commonly classified as relevant flows and mice flows. In the scope of flow classification the three parameter F_D, F_P and F_B can be distinguished for each flow F in the network. The parameter F_D represents the duration of the flow. The parameter F_P represents the number of packets transmitted by the flow and the last parameter F_B represents the amount of bytes per packet transmitted by the flow. For each of these parameter a threshold value can be defined. A flow is assumed to be a mice flow as long as not all of the defined thresholds are exceeded. If all three thresholds are exceeded a flow is assumed to be an relevant flow. The parameters enable a flexible definition when a flow should be assumed to be an relevant flow by the system.

A major part of flows in a data center environment are mice flows with a small amount of relevant flows carrying the majority of the transferred data [17]. Wildcard flow rules are used to cope with the high amount of mice flows present in the network. By using wildcard rules, which match multiple mice flows, all of those mice flows are forwarded usually using the same path or at least the same link. A link which already forwards a high amount of mice flows, can now be easily overloaded when facing with an incoming relevant flow also matching the same wildcard flow rule. This problem is depicted in Figure 4.2. S_0 to S_3 represent switches in a network connected via links drawn as solid lines. Arrows represent flows forwarded over the links, a mice flow is represented by a small arrow and an relevant flow is shown as a larger arrow. In Figure 4.2 the mice flows are forwarded by S_0 to S_3 over the link containing S_1. The link is assumed to provide sufficient bandwidth for these mice flows, but is overloaded when the incoming relevant flow is forwarded by S_0 over the same path. This happens when the relevant flow matches the same wildcard flow rule at S_0 as the mice flows.

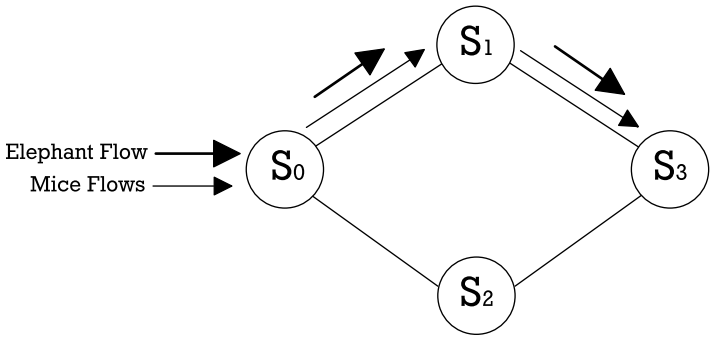

Figure 4.2: Without Relevant Flow Awareness

A possible solution for this problem is depicted in Figure 4.3, which describes the concept of relevant flow awareness. The relevant flow is detected and reported to the controller. The controller can then construct a matching flow rule for this relevant flow to be installed in S_0. During this process the controller also searches for a less congested path using a given algorithm to assign the relevant flow to.

As an outcome of the example scenario in Figure 4.3, the relevant flow is forwarded by S_0 over the less congested link containing S_2.

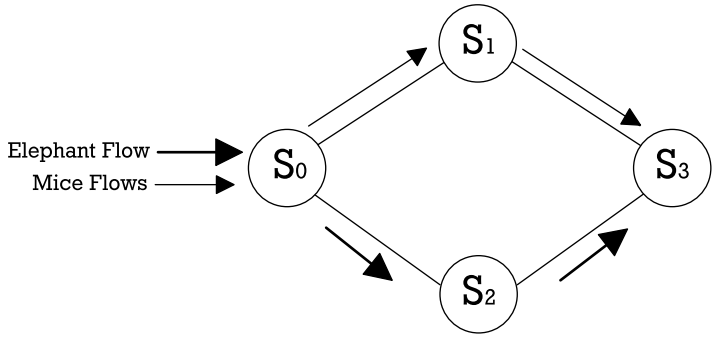

Figure 4.3: Relevant Flow Awareness

Distinguishing mice flows and relevant flows in this scenario has several benefits. The link utilization throughout the network can be increased by managing important relevant flows without having the need to manage each individual mice flow as well. This preserves both limited control bandwidth and also flow table space, since mice flows can be matched using small amount of wildcard rules, while having direct matching rules for the smaller number of relevant flows in the network.

Requirements

The relevant flow awareness concept requires a point of detection in the network. At this point of detection the traffic information about flows is evaluated and reported to the controller. The controller gathers the information from this point of detection and decides which flow can be classified as an relevant flow. The point of detection can be the switch itself processing the flows and keeping flow statistics or an end host reporting its buffer states to the network [17].

Beside the point of detection in the network, it needs to be defined, which flow is assumed to be an relevant flow and which one belongs to the category of mice flows. The categorization can be influenced using the introduced parameters F_D, F_P and F_B. The actual value set for each parameter can depend on the overall network environment and the applications running in the network.

Applications

The concept of relevant flow awareness is used in several optimization approaches, namely DevoFlow, Flow Label for ATM and Mahout. DevoFlow aims to devolve control to the switches to give a switch a more flexible forwarding scheme. This flexibility is used to forward mice flows with minimal controller interaction to preserve control bandwidth. Additionally DevoFlow identified relevant flows which receive central flow management. Since gathering flow statistics in OpenFlow might not be as sufficient as needed because of the limited resources of a switch, DevoFlow introduces new flow statistic gathering mechanisms. These mechanisms aim to provide an efficient way to gather flow statistics by the controller which is vital to identify relevant flows in the network. The authors in DevoFlow were able to improve throughput compared to an equal cost multi-path routing algorithm [39] from 32% up to 55% for different network topologies [18].

The flow label for ATM [76] approach also treats mice and relevant flows differently. Even though the terminology is different, because the flows are not referred to as relevant flows, the authors try to identify flows with a larger duration and a higher data load than the majority of flows observed in a network.

They specify an relevant flow to have a duration of at least 20 seconds ($F_D > 20$) with a transmission rate of 40 packets per flow ($F_P > 40$). Identified relevant flows in their approach were switched using the connection oriented scheme of the underlying ATM hardware, whereas the rest of flows were forwarded in a hop-by-hop scheme by the router software. Following this approach they were able to handle approximately 4.5 times more traffic than by handling all flows equally in a hop-by-hop forwarding scheme.

Tradeoffs

The concept of relevant flow awareness includes different tradeoffs influencing the performance of the concept. The boundary between mice and relevant flows can be flexible allocated using the three parameters F_D, F_P, and F_B. Each of them requires a different way of measuring to compare the current parameter of the flow versus the defined threshold. F_D implies, that a flow requires to exist for at least the time set for F_D. If F_D is set too high compared to the full duration of the flow, the relevant flow is detected to late. The same relation applies for the other two parameters F_P and F_D.

Therefore the time delay until a flow is detected as an relevant flow is also influence by the detecting location. In the DevoFlow approach a flow is detected within the network. The controller collects flow statistics from the switches to check which flow could be an relevant flow. Another approach is used by Mahout [17], which uses a shim layer installed at each end host. This layer monitors the TCP buffer of the end host connected to the network. If the buffer exceeds a certain threshold of bytes the outgoing flow is assumed to be an relevant flow. Therefore Mahout uses the parameter F_B and F_B to decide which flow can be classified as an relevant flow.

Mahout claims to provide an order of magnitude faster relevant flow detection compared to an in-network monitoring approach like used in DevoFlow. Early detection of relevant flows is a crucial component, since the detection delay needs to be smaller than the overall flow duration in order to perform optimizations on the flow placement.

4.3 Label Switching

Constraints and Objectives

The flow matching scheme of the latest OpenFlow 1.4.0 [34] defines 13 required matching fields to be implemented by OpenFlow supporting switches. With this number of fields a fine-grained flow matching can be implemented for fine-grained traffic management in the network. However, matching a high number of specific fields in one rule leads to rules with high space requirements and ultimately to a higher flow table space requirement in order to store these fine-grained rules.

The objective of the label switching concept is to reduce the space requirements for individual rules, in order to increase the number of rules which can be stored in a space limited TCAM. Reducing the size of a flow rule, by replacing a heavy fine-grained flow rule in terms of their bit-length with a shorter bit pattern can also simplify the matching complexity inside each switch.

Structure

The structure of the label switching concept is depicted in Figure 4.4. The concept describes a flexible label switching approach, where matching in a switch can be simplified by matching a label rather than complex header fields. In order to add labels to the packets in the network the defines three different switching roles:

- Ingress switch

- Intermediate switch
- Egress switch

An ingress switch stores complete OpenFlow rules in its flow table. Each of these rules get associated with a label L. For an incoming packet at an ingress switch, the switch performs a lookup for the given packet if it matches an already stored flow rule. If this is not the case, the switch forwards the packet to the central controller. The controller subsequently installs a matching rule for this packet in the ingress switch along with a label L. Such a label L can include various informations to be used in the forwarding process. Applications of the label switching concept can use the L for including path information to realize a source routing scheme or simply identify an action with a unique flow id stored in the L. The ingress switch pushes L onto the packet and forwards it according to the information provided in L included by the controller.

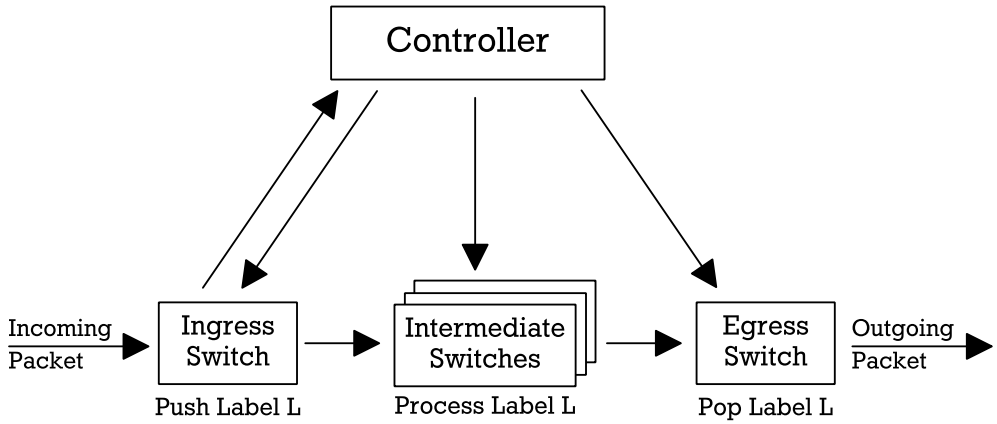

Figure 4.4: Label Switching

An intermediate switch receives packets from other ingress or intermediate switches, which already have a label L assigned to it. It therefore only processes the information contained in L to make its forwarding decision. The label can therefore include an id, which tells the switch which output port it should use to forward the packet. In this case the single unique id stored in L can replace the complex flow matching part of the rule. The switch just requires to have a table of ids associated with the action to perform on packets containing this id.

An egress switch is the last switch on a path before the packet reaches the end host. Therefore the egress switch removes the label L from the packet to make the label switching process transparent for both end hosts. A packet enters the network without a label and leaves the network without a label which was used in between to forward the packet.

Requirements

The label switching concept introduces a shifting in complexity and resource requirements from the intermediation switches to the ingress switches. The majority of rules in an approach using the label switching concept are stored in the ingress switches. The intermediate switches only require a smaller rule set. Changes in the forwarding path also mainly affect the ingress switches, because the rules need to be changed at the ingress switches. Therefore the workload for the ingress switches increases, whereas the requirements for intermediate switches in terms of flow table space and controller interaction are

lowered. The ingress switches need to be capable of coping with additional load, otherwise the label switching concept can not be applied in the network environment.

Applications

The label switching concept is applied in different approaches, in both the classical network context as well as in the OpenFlow context. The Flow Label for ATM approach used labels to identify flows which can be switched using the faster but connection oriented ATM hardware. A flow is identified by its label stored in the label L. Using this labeling scheme enabled the approach to handle approximately 4.5 times more traffic. Because the approach was not evaluated in terms of storage requirements no quantified numbers can be given here, but in general identifying a flow by a fixed length label can require less rule table space than storing matching rules for the IP-prefix.

MPLS is another example of adopting the label switching concept to forward packets in todays IP networks. MPLS uses the information in the label L to forward a packet on a predefined path through the network. The labeling information stored in L is replaced by each switch with the label information which is expected by the switch on the next hop. The process is similar to the Flow Label for ATM approach such that the predefined paths are not represented by a unique label. Instead each switch manages its own label set along with the corresponding label switched path. Today MPLS is widely deployed in many core networks [109] and has shown its capabilities for improving the forwarding efficiency of these networks. Relative performance comparison between IP and MPLS have reported MPLS to have a shorter packet forwarding delay [64], based on the simplified lookup scheme of an label switching concept. Furthermore MPLS is prominent in the domain of traffic engineering, which focuses on an increasing usage of network resources [111].

SwitchReduce adopts the label switching concept in an OpenFlow context. This approach includes the whole hop-by-hop path information in the label L. The path informations are stored as a list of VLAN IDs. The number of VLAN IDs equals the number of hops from source to destination for the packet. Intermediate switches only requires to store a single wildcard rule per outgoing port. Each rule specifies which VLAN ID is forwarded to which outgoing port. Exact matching rules only need to be stored in the ingress switches along with the predefined path. This approach has shown an average compression in terms of flow table size of more than 99% on aggregation and core switches and 49% on top of the rack switches.

Compact TCAM uses unique Flow IDs stored in the label L to reduce the flow table size of intermediate switches, since they only store a fixed length Flow ID for being matched on incoming packets, rather than a complete OpenFlow rule. The paper describing the Compact TCAM approach mainly focuses on power savings, therefore the authors evaluate the power savings achievable by applying their Compact TCAM approach in a data center environment. Nevertheless, the power gain reported at the switch fabric being about 80% compared to SDN switches supports the fact that replacing a complete OpenFlow rule matching part with a single fixed length Flow ID can greatly reduce the flow table space requirements, without introducing noteworthy drawbacks.

Tradeoffs

As mentioned in the Section Requirements of the label switching concept, the complexity and resource requirements in an labeled switch network are shifted to the ingress switches. Therefore, these switches need to be capable enough to handle this workload. Nevertheless, this resource shift enables the controller to control the forwarding paths mainly by communicating with the ingress switches, which saves both bandwidth and number of messages required to deploy a rule change in the network.

Another tradeoff is related to the amount of information stored in the label and its implication for the rest of the network. Having less information stored in the label, such as a single Flow ID in Compact TCAM, restricts the ability for proactive rule installation. Because the Flow ID gets assigned reactively by the controller, on request of the ingress switch, the paths can also be constructed only in a reactive fashion. The switches along the path can only be notified by the controller about the new Flow ID after its creation. Including additional information into the label can enable the controller to proactively install a fixed number if rules in intermediate switches. This does not harm the flexibility of the network itself, since the path selection can be easily influenced by the rule assigned at the ingress switch.

4.4 Forwarding Information Aggregation

Constraints and Objectives

Data centers with a high concentration of servers hosting multiple virtual machines (VMs) per machine pose a growing challenge on the switching fabric [72]. OpenFlows ability of forwarding traffic on a fine-grained basis can easily overrun the capabilities of current switching hardware in terms of the highly limited flow table space. Beside the higher number of flow rules necessary for fine-grained traffic engineering, rules often get installed redundantly in the network if no aggregation scheme is used.

Therefore, the main objective for the forwarding information aggregation concept is to reduce redundant information stored in the forwarding tables of the switches. Switches which connect multiple switches in the aggregation layer often face the challenge to store a high number of fine-grained flow rules. This scenario is depicted in Figure 4.5. Each of the switch $S_{1,1}$ to $S_{n,1}$ store a rule set containing m different rules. Each switch is connected with switch $S_{2,1}$ which faces the challenge of storing $n * m$ number of rules in the case it cannot aggregate some rules into a wildcard rule, which is the case of the flow depended counters should be preserved at the switch.

Requirements

The aggregation of forwarding information can be done in two scopes. Forwarding information can be aggregated purely in the local scope of each individual switch or in a wider scope including several switches, up to a scope including all switches in the network.

Aggregation is possible for both scopes, if redundant information is present in the forwarding information stored by the network devices. In order to aggregate and therefore reduce the number of forwarding entries an appropriate algorithm is necessary. The algorithm should take the forwarding information set stored by a device and output a compressed version of the forwarding information set. The algorithm should thereby ensure that the forwarding abilities are preserved, meaning that a packet which could be forwarded by the initial greater forwarding information set should also be forwarded to the same destination using the compressed forwarding information set.

Enlarging the scope to multiple network devices, the examples of current OpenFlow deployments show a high distribution of redundant flow rules across the network devices [45]. An algorithm for aggregating forwarding information in such an enlarged scope requires a comprehensive view across the forwarding information sets stored in multiple network devices. It takes this global forwarding information set as an input and calculates compressed forwarding information sets for each participating network devices. Keeping in mind that the forwarding ability needs to be preserved, other abilities such as flow statistic gatherings could be affected. As the example SwitchReduce [45] advises, this effect can be compensated to some extend to preserve flow statistic gathering abilities even if redundancy is efficiently removed across the network.

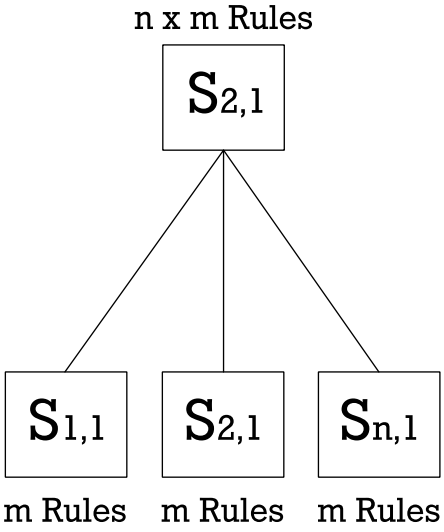

Figure 4.5: Without Forwarding Information Aggregation

Structure

In order to face the problem of the explosion of required flow rules in the higher aggregated switch layers, an efficient way of aggregate multiple flow rules into a single wildcard rule is necessary. As the authors in [14] observed, the diversity of the OpenFlow matching possibility is higher than the possible action set which can be performed at a switch. This observation is the basis for the for the concept of forwarding information aggregation. A switch such as $S_{2,1}$ should not be required to store the sum of all flow rules stored in the lower level. A function $F(s,r) = s'$ should exist, which takes an rule set s along with a requirement set r as an input and outputs a rule set s' which has an smaller or equal number of rules which does not violate any of the requirements specified in r. This function can be evaluated for each switch in the network in the network to aggregate a given rule set into a rule set with a smaller or equal number of rules. The rule set s' can have an equal number rule compared to s in the case no two or more rules in the rule set s can be aggregated into a single rule.

Application

Aggregating forwarding information to preserve table space is also a known mechanism in the classical network context. The introduction of the Classless Inter-Domain Routing [35] takes information aggregation into account as the commonly understood method to reduce forwarding state. A study on the fast growing number of the global BGP routing tables [9] also studies the scope of route aggregation possibilities. Since traditional networks heavily rely on forwarding based on IP-prefix matching, the aggregation of such IP-tables is widely studied [25, 101].

In an OpenFlow context aggregation of forwarding information is also a prominent challenge. The large diversity of possible header combinations OpenFlow rules can match enable more flexible flow management but simultaneously increases the flow storage requirements for these fine-grained rules. SwitchReduce faces this challenge in storing a limited number of static wildcard rules in a switch connecting multiple other switches. The function $F(s)$ would therefore output the a wildcard rule set with the size of the number of output ports the related switch provides. This fixed rule set is able to efficiently forward all incoming traffic in case the packets are labeled. This labeling can be done at an ingress switch of the packet in combination with the network wide view of the controller. Wildcard rules usually lack an efficient way to gather flow-by-flow statistics. SwitchReduce tackles this issue by having at least

a single exact matching rule installed for each flow in an ingress switch, which can be used to collect flow dependent statistics. SwitchReduce is reported to reduce both flow tables in an aggregation and core switch as much as 99% as well as top of the rack switches, usually used as ingress switches by 49%.

In comparison to prefix aggregation schemes mainly used in today's IP-based networks, OpenFlow rules can include wildcards at any position, making the aggregation algorithms to be used more complex. It is even assumed to be a NP-hard problem [67]. A approach of a so-called non-prefix aggregation is bit weaving presented in [69]. It relies on the simple observation, that two different forwarding rules with the same destination can be easily merged, if they only differ in one bit in their matching scheme. This bit can therefore be replaced by a wildcard. While the bit weaving can offer an average compression ratio of 23,6% using real world evaluation data [69], it struggles to perform the aggregation calculation in a reasonable calculation time [67]. This makes it valuable for offline aggregation, but imposes challenges for applying this approach for a dynamic network environment with fast changing rule sets, since the aggregation algorithm might need to be applied after each flow rule update, high computational effort is multiplied by the number of flow updates per timeframe.

SwitchReduce, Compact TCAM and flow label for ATM and MPLS all utilize forwarding information aggregation within their labeling concepts. The way they reduce the matching complexity is achieved by aggregate similar information into one label. Labels can be used to aggregate different flows, with the same path direction for the next hop.

Tradeoffs

Tradeoffs in the forwarding aggregation process are based on the amount of information being lost when forwarding information is being aggregated. A poor aggregation scheme preserves the information kept in the forwarding state for further usage. In an OpenFlow context this applies for storing fine-grained rules throughout the network to gather fine-grained flow statistic. As the SwitchReduce approach shows, fine-grained rules do not need to be stored redundantly in the network, therefore enabling a aggregation scheme without losing the ability of gather fine-grained flow statistics is possible. Nevertheless, stronger aggregation of forwarding information can affect the forwarding flexibility of the network, because the forwarding state stores less number of different paths through the network, leading to less alternative paths in case of link failure or load balancing. Additionally, aggregation of forwarding information can have side effects on the routing protocol. For instance, route aggregation of BGP routes can result in loops [93].

In addition, higher compression approaches might introduce high computational demands as it is the case with bit weaving [67]. Since the aggregation process might need to be applied after each flow rule update, the computational effort increases with the flow update rate present in the network. The tradeoff can be identified between compression ratio and computational effort necessary to achieve the compression ratio. A lower compression ratio increases the flow table space requirements, but might decrease the computational effort and in addition the time delay for performing flow updates at a network device.

4.5 Combining Resource Optimization Concepts

Table 6.2 lists all of the four previously introduced concepts along with their focussed data plane resource. Each concept has its own set of resources, which can potentially be optimized applying the corresponding approach to a network. This section aims to summarize the possibility of combining different concepts in order to increase the set of optimized resources. If a combination of two approaches is possible, the two approaches are assumed to be complement. Supporting the complement status, the

requirements of one concept should match the objectives of the second combining approach. This will get more clear in the first combination of the hierarchical switch topology concept with the relevant flow awareness concept. If no real benefit of combining two concepts can be identified, the two approaches are assumed to be independent. In the case where in a combination of two concept one concept would violate the structure or basic ideas of the other concepts, this combination is assumed to be conflictive.

	Control Bandwidth	Flow Table Space	Flow Rule Capacity	Forwarding Capacity
Approach	Data Plane Resources			
Hierarchical Switch Topology	√		√	
Relevant Flow Awareness				√
Label Switching	√	√		
Forwarding Information Aggregation		√	√	

Table 4.2: Data Plane Resource Focus of Optimization Concepts

Hierarchical Switch Topology with Relevant Flow Awareness

The hierarchical switch topology concept addresses the resources control bandwidth and flow rule capacity to be optimized. The relevant flow awareness concept mainly focus on an increased forwarding capacity. Merging these resource sets would result in an increased optimized resource set. From a practical point of view, the relevant flow awareness concept can usefully support the hierarchical switch topology concept. The hierarchical switch topology concept relies on an efficient algorithm in order to decide which flow rule should be stored at witch switch level. Flow rules with higher weights or priority should be stored in the lower layers in order to be processed with less delay and shorter forwarding paths. Relevant flows can be an ideal candidate to be stored in the lower layers, if they have been identified by the point of detection of the relevant flow awareness concept. Therefore the two concepts can be assumed to be complement to each other.

Hierarchical Switch Topology with Forwarding Information Aggregation

The forwarding information aggregation concept can be usefully applied in combination with the hierarchical switch topology. Aggregation of forwarding information in order to preserve flow table space can be important for higher layers of the switch hierarchy. For the lower layers the concept tries to store exact matching rules in order to match fine-grained flows and enable statistic counter for them. For higher layers it becomes more important to find a matching flow rule in the flow table in order to be able to forward the flow without invoking the controller. Therefore, higher layers would store preferable wildcard rules, which can match a higher number of similar flows. Therefore, the two concepts can be assumed to be complement.

Label Switching with Hierarchical Switch Topology

The label switching concept already addresses the three resources control bandwidth, flow table space and flow rule capacity, whereas the hierarchical switch topology also optimizes the control bandwidth and flow rule capacity. The optimizing resource set of the combination of these two concepts would not add an additional resource. However, the tradeoff section of the label switching concept highlighted the increased requirements for ingress switches. Since the flow matching workload is mainly done by the ingress switches, the requirements for available flow table space are likely to be bigger for ingress switches than for intermediate switches, which can store shorter rules matching a specific label instead of a more complex header structure. For increasing the flow table space of ingress switches the hierarchical switch topology could be used, meaning that one ingress switch gets additional switches as a cache for flow rules. This would be the most intuitive approach to usefully combine the label switching concept with the hierarchical switch topology concept. Nevertheless, applications of the label switching concepts, such as SwitchReduce and Compact TCAM have reported a lower flow table space requirement even for ingress switches. This can mainly be traced back because a switch which acts as an ingress switch for one flow might act as an intermediate switch for another flow. This diversity results in even reduced flow table space requirements in both applications. Therefore, there is no need to apply the hierarchical switch topology concept in combination with the label switching concept, since so far there is no proven benefit of the combination of both. The two concepts are therefore assumed to be independent.

Label Switching with Relevant Flow Awareness

The label switching concept can be usefully combined with the relevant flow awareness concept within the following aspects. Having labels propagated across the network can be efficiently used to change paths by updating a flow rule at an ingress switch of this path. This makes traffic rerouting efficiently possible, which can be beneficially used by the relevant flow awareness concept in order to find optimized paths for relevant flows. Combining these two concepts can lead to an increased resource optimization, with the benefits of the label switching concepts and relevant flow awareness concept. The concepts are therefore assumed to be complement.

Label Switching with Forwarding Information Aggregation

The combination of the label switching concept with a forwarding information aggregation concept in place promisee an increased flow table space and flow rule capacity. The discussion of the possible combination of these two approaches must distinguish three different switch roles: The ingress, the intermediate and the egress switch.

Looking at the ingress switch first reveals, that the ingress switch stores occasionally fine grained matching rules along with a specific label to be assigned to matching packets. A possible aggregation algorithm would search for distinct matching rules assigning a label containing the same information. Such distinct matching rules could be aggregated without changing the forwarding behavior of the subsequent network. Nevertheless, applications like SwitchReduce rely on having at least fine-grained matching rule stored per flow in the ingress switch, in order to provide per-flow statistics at least with the help of the ingress switch. Aggregate multiple flow rules into one single rule would also aggregate the statistic counters. Since this results in a blur effect for statistic gathering, it might not be desired.

The second switch role to investigate on is the intermediate switches. They only store short flow rules matching a specific label information contained in the label initially assigned by an ingress switch. Labels could be implemented by a unique flow ID, staying consistent along the whole flow path, like it is the case in the Compact TCAM approach, or the label can be different for each hop in a network, such as it is used in MPLS and SwitchReduce. Aggregation of unique flow IDs can be assumed to be more complex

than helpful. Since the basic idea of such concepts is to identify a complete path from ingress to the egress switch by a unique ID, aggregating multiple IDs in between would harm this concept for the sake of saving flow rule space in intermediate switches.

Furthermore, flow rules matching only fixed length information such as labels or IDs are in general less space consuming, leading to decreased flow table requirements without any aggregation necessary. Even if an aggregation concept would be applied, it probably would not save much space, since the flow rules itself are already small in their size. In addition to the small saving ratio, an aggregation scheme usually requires additional computing power, in order to calculate the aggregated forwarding information. If only a small amount of flow table space is considered to be saved by an aggregation, the computing power required in order to achieve these savings makes the process likely to be inefficient.

Another way of implementing the label switching concept is using different labeling information per hop. Therefore, each adjacent switch knows which label the switch at the next hop expects for specific packet on predefined path. This information can be kept locally like in MPLS, where each incoming label is stored along with an outgoing label or the complete information can be included into the label assigned to a packet at the ingress switch in a source based routing fashion supported by the central network view of a controller.

Labels locally managed by each switch are usually assigned to a specific path, as it is the case in MPLS. Therefore, each switch manages the transitions of incoming and outgoing labels for each predefined path individually. Such information can not be aggregated locally, since this would also aggregate the path information. The information on which path a packet should be forwarded is encoded within the label information. If one switch would aggregate labels from different paths into a single label, only because they are forwarded at the same outgoing port, the next switch would lose the ability to forward those packets on different paths as it was intended. Therefore, in such a scenario label information can not be further aggregated at intermediate switches.

Instead of managing labels locally at the switch, the labels can be managed by a central entity. In this scenario, the aggregation concept can be already applied during the distribution and management of the labels. Having different labels per hop only requires a label to express on which outgoing port to forward a packet with the given label. If the label does not require to be associated with an incoming port, packets from different incoming ports, which should be forwarded over the same outgoing port can share the same label. In the domain of OpenFlow, where beside pure forwarding of packets also additional actions, such as dropping or rewriting header information it is sufficient to have a unique local label for each action possible to be performed by the local switch [45]. This already implies the objective of an aggregation scheme, which is to remove or even avoid redundant forwarding information to be stored in flow tables.

The last switch role present in the label switching concept is the egress switch role. An egress switch ensures the transparency of the concept for the end hosts. Flow rules in the egress switch match a label of incoming packets to determine the output port for those packets. Before forwarding the packet to this output port, the egress removes all label information from the packet. Therefore, the end host is not aware of any label switching concept on the transport path. For the flow rule set maintained at the egress switches an aggregation concept similar to the intermediate switches with different label per hop can be applied. Since the labels only need to express information about which action to perform with the current packet, the number of labels necessary per egress switch can be limited by the number of available actions. Therefore, during the establishment of labels for an egress switch, this upper limit can be taken into account in order to keep the number of labels stored in an egress switch limited. Further aggregation potential can not be identified.

The discussion on how to aggregate information within the label switching concept can be summarized as follows: If unique labels are used throughout the network for identifying concrete paths, an aggregation is assumed to be not efficient for aggregate label information without breaking the idea to be able to distinguish paths by their unique labels and having the same label assigned to a concrete path for each switch on this path. In this case the two concepts can be assumed to be conflictive, meaning that there is no benefit in combining these approaches.

Relevant Flow Awareness with Forwarding Information Aggregation

The basic observation of the relevant flow awareness concept is that fine-grained flow visibility throughout the network is not necessarily required. Instead, only relevant flows meeting certain predefined parameters require certain central flow scheduling. This concept has implications for an aggregation concept, which could aggregate multiple similar mice flows into a common flow rule, whereas it keeps flow rules matching relevant flows untouched. Therefore, the important relevant flows and their corresponding matching rules are preserved, keeping the visibility for a central controller to access statistic counters for those flow. Mice flows instead are matched using aggregated wildcard rules. This observation leads to the assumption that both concepts are complement and therefore can be used in combination.

5 Application Interaction Analysis

Application interaction in the networking domain is nothing completely new. Interaction already takes place, in the way an application hands over data packets to the network, which delivers them according to the header information provided by the application. This header information is focused on transport related information such as destination information. Nevertheless, the amount of information which an application provides within the header information is limited. In the same way, the information exchange between the different network layer of the OSI layer model is strictly standardized within given boundaries. This hierarchical layer model is one of the key factors a large network such as the Internet was able to evolve to such an enormous size.

However the increasing diversity of applications using the networks transport service impose different requirements for the transport quality such as latency, security bandwidth and so on. Even though a number of different approaches have been developed and implemented in the recent years extending the scope of application interaction (Section 5.2 and Section 5.3) in the direction of network awareness and application awareness, none of these approaches has lead to a wide adoption. With the increasing popularity of SDN the focus on an extended application interaction is gaining attention. Especially the Northbound API envisioned in the SDN concept can open new possibilities for an extended application interaction, in which an application can directly communicate with the central controller. The nature of information and the possibly usage of such information is discussed in Section 5.4 and ongoing.

A more extended application interaction can be used in the first place to enhance the task of network management systems, including monitoring and configuration of the network as well as security and QoS management. In order to develop a more efficient network management system, a recent study highlights the demand for an open and programmable control layer [36]. Resource management is one subtask of the network management system and is not inherently a process solely running inside the network. Instead, the resource usage highly depends on the applications accessing the network. Usually a variety of different concurrent applications compete for the available and finite network resources. The applications also have different requirements for instance regarding: reliability, bandwidth, latency and security [36]. For instance, a video streaming application transmits heavy loaded video data over the network and therefore requires a high bandwidth during the transmission. The transmission of voice data on the other hand used in a VoIP service is primarily sensitive to latency and jitter. In order to provide a sufficient VoIP service the network needs to provide a connection between the communicating end hosts with sufficient latency boundaries.

Because today's network forwarding is often done on a best effort manner [118], providing sufficient bandwidth or latency for a specific application can not be guaranteed by the network. However, these application requirements need to be considered by future networks in order to provide a sufficient application quality to an end user. Therefore, a deeper integration and interaction between applications and networks is necessary to cope with the increasing challenges in today's network management systems.

This chapter gives an overview of an abstract application interaction model. In the scope of application interaction both the concept of application awareness and network awareness are discussed. Both concepts can be identified as being a part of the overall application interaction concept. For a better understanding of each of those concepts, examples are presented including well known protocols such as RSVP [116] as well as concepts envisioned for future SDN enabled networks. Since both concepts are

represented in the SDN concept, application interaction can gain new attention using the mechanism provided by SDN implementations like OpenFlow. Therefore, the relation between SDN and application interaction is discussed in more detail at the end of this chapter.

5.1 Abstract Application Interaction Model

For a better understanding of the concept of application interaction, a possible model is depicted in Figure 5.1. The model consists of the network and a set of applications accessing the network. The network itself is divided into the control plane and the data plane. The control plane includes all forwarding decision processes like routing protocols. The data plane includes all components realizing the packet forwarding functionality. Packets are forwarded from an ingress port to an egress port based on the information from the control plane.

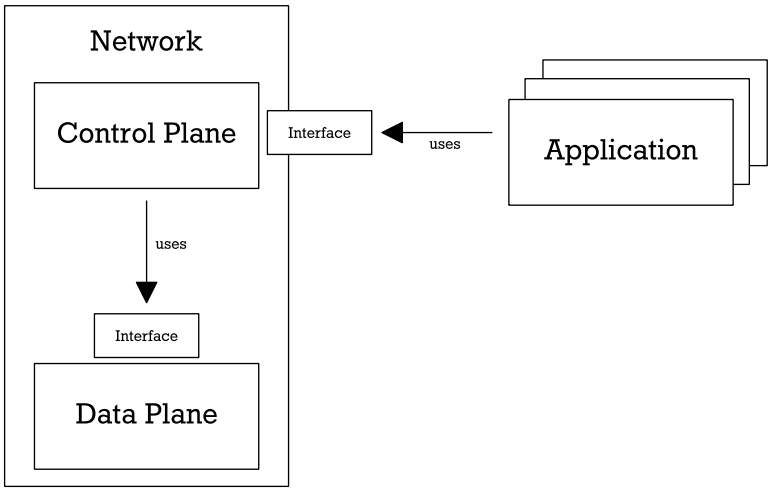

Figure 5.1: Application Interaction

The control plane should be able to gather state information of the data plane elements to take this state into account for its forwarding decisions. Therefore, the data plane interface depicted in the application interaction model is used for both, the programming of the forwarding hardware according to the control plane as well as to gather information of the data plane. Such information can be current link utilization or forwarding speed of links.

In traditional networks, packet headers carry all related information for a packet, such as source and destination. Those information is used for forwarding packets through the network. Al though there are several approaches, which include information into the packet header, which can be used to prioritize a packet on its path (Section 5.2 and Section 5.3), none of those approaches has been widely adopted for reasons with are discuss in the following sections.

Realizing a special packet handling by the network can benefit from additional application related information. Such information can make the network aware of the application using it and therefore can trigger optimizing processes inside the network. The network could provide an interface as showed in Figure 5.1 to be accessed by the application. Such an interface can be used by the application to make the network aware of the application and additionally about the specific requirements of the traffic of this application. An application could also use this interface to announce resource reservation requests, in order to reserve a certain amount of resources in advance for later usage. The extent of usage of such

an interface highly depends on how much information both the network opens up to the application and the amount of information an application shares with the network. An application might have useful information about network devices it needs to use in the future or concrete performance requirements like bandwidth or maximum allowed delay for a transmission. This information can be shared with the network to be further used to reserve appropriate resources in the data plane or to construct optimized paths at the control plane to speed up a transmission using a faster path of network devices.

Passing information from an application to the network through such an interface can make the network aware of the application running on the network. This is known in the literature as application awareness further described in the subsequent Section 5.2. Information about the current network state provided by the network and accessed by an application can be used to make the application aware of the current network state and the performance the application can expect to receive from the network. This is known in the literature as network awareness further describes in Section 5.3. Both are part of the application interaction. Concepts developed in both domains can be beneficial for concepts of an improved application interaction.

5.2 Application Awareness

The central question around application awareness is: Why should a network be aware of the applications running on top of the network? Traditionally an IP network is designed to provide data transfer services to forward data in packets from a source to a destination. Today's networks, especially the Internet has proven a strong scalability while providing the basic forwarding service to all connected end-hosts. Nevertheless, forwarding in today's networks is done primarily on best effort [118]. With the emerging number of multimedia applications like audio and video streaming the bandwidth demand for the networks is constantly growing. Network operators are forced to provide sufficient network capacity to cope with the increasing demand of the multimedia applications, while trying to reduce the operational costs of the network [103]. In order to provide sufficient network capacity, especially for high usage peaks, the supplied network hardware is usually over provisioned to meet the Quality of Service (QoS) specifications [48, 47]. Because of the over provisioned hardware base, the networks mostly operate with only a small utilization in the time frame between two traffic peaks. A better link utilization can be achieved with an improved network management, Google reports a link utilization of nearly 100% of many links and on average 70% utilization of all links in its self-operated WAN network, which is interconnecting its data centers across the globe [47]. With a better achievable link utilization less hardware will be necessary to be prepared for certain traffic peak, therefore an improved network management can help to reduce the operational costs of the network.

Beside the growing number of bandwidth intensive applications, the Internet is increasingly used for transmitting voice data. Voice over IP (VoIP) services constantly gain popularity [105] and will most likely replace the traditional telephony service in the future. VoIP data is highly delay and jitter sensitive and thus requires short delays in the network [13]. The prominent best effort transport in today's networks can not guarantee a sufficient QoS for voice data, therefore this traffic needs to be prioritized in the network in order to provide a sufficient audio quality to the end user. VoIP data can only be prioritized in the network if the network is aware of this specific application and its special QoS needs.

Different applications impose diverse requirements on the network, therefore they might need to be treated differently by the network. A video service like YouTube benefits from a high bandwidth on the path between the server hosting the video file and the user playing the video on its device [66]. Instead, of finding a path with a maximum of bandwidth between these two parties, most of today's routing protocols running in the network will find the shortest path in terms of hops between the video server

and the device of the user, even though alternative paths can be found providing a higher bandwidth than the shortest path constructed by the routing protocol [88]. Overall multimedia applications are more sensitive to bandwidth rather than the number of hops [66].

In summary, the network needs to be aware of applications currently using the network in order to provide the right QoS for the right application. Additionally, application awareness is crucial for providing visibility, billing and security in the network [81]. All of these properties are to some extent application dependent, therefore the network should be aware of the specific applications using its transport service.

Overview of Existing Application Awareness Approaches

In absence of an appropriate interface for applications to provide information to the network different approaches are used to make a network application aware. Some commonly used approaches are listed in Table 5.1. The port and protocol identification approach tries to identify an application based on its data transmitted over the network and is based on the addressing scheme of the network and transport layer. Well known ports and protocol used in combination can identify an application within the network. DNS for instance commonly uses port 53 and uses UDP as a transport protocol, HTTP traffic gets transmitted over port 80 and HTTPS over 443, both using TCP as a transport protocol. However, port and protocol-based application identification is no longer accurate, since most applications use random ports (e.g. P2P-applications) or encrypt their traffic over HTTPS [81]. Therefore, different applications transmitting over the same port can not be distinguished simply relying on the port and protocol combination used by an application.

Approach	Description	Examples
Port and Protocol Identification	Identify applications based on their used port and protocol combination	-
Deep Packet Inspection (DPI)	Identify applications based on the payload a packet carries	[84]
Machine Learning	Identify applications based on traffic patterns previously by exploiting a training model	[81, 115]
Active networks	Application can embed code into network packets which is executed on the forwarding devices itself	[97]
Labeling	Use label switching techniques to assign QoS information as a label to packets at the edge of the network	[11, 37]
Network Interface	Provide an API to directly announce application requirements to the network	[73, 49, 104, 26]

Table 5.1: Application Identification

The Deep Packet Inspection (DPI) approach inspects the complete packet, especially the payload to search for known application patterns. DPI exploits filtering techniques to match known data patterns on the inspected packets. DPI is usually more accurate than the address and port based approach [81]. However, it requires additional middle-boxes to be deployed in the network to inspect the packets.

This imposes high computational effort and pattern maintenance [81]. Additionally, DPI can not provide information about the actual application state, neither about the QoE parameters collected by the application [50]. End-to-end encrypted data traffic like HTTPS also limits the abilities of DPI [81].

The third approach exploits machine learning (ML) techniques to efficiently identify applications. ML does not require all packets to be inspected, instead it only requires specific flow level features like addresses or ports or the first N packets of a flow [57]. This information is collected to calculate a model which can be used to identify unknown packets based on comparing them to the stored model. It therefore requires less computational effort compared to DPI [107]. However, the model need to be constructed and real traffic to be trained in advanced. The identification rate of such an ML approach highly depends on the quality of training data.

As a scalable approach to enrich packets with QoS information a labeling mechanism is proposed in [11, 37]. The IETF describes an architecture for differentiated services (DiffServ). In order to simplify the mapping process of different packets to predefined traffic classes, packets get assigned with a label at the ingress of a network containing the necessary information for mapping them to a specific traffic class. The labeling information is stored in the DS-field, which is the successor of the type of service (ToS) field, which did not find wide acceptance for assigning QoS information to a specific packet. Intermediate forwarding devices in the network match against such a label and look up the desired forwarding behavior with respect to the desired QoS treatment for this packet. For latency-sensitive VoIP traffic the label could contain information that requests for a faster forwarding. A network device processing such a packet can for instance put the packet at an earlier place in its forwarding queue. Even though this approach is scalable [11] even for larger networks, the traffic classes need to be defined in advanced which makes this approach relatively static. Nevertheless, the network can be made aware of certain application traffic and its special requirements assigning them to an appropriate traffic class.

However, the practical usage ToS and DiffServ is limited to networks, where the operator controls both the network and is aware of specific applications using the network. Since the DS-field in the IP header can be in principal modified by every application or network device on the path, it is difficult to rely on a specific initiator of the encoded information, since it does not indicate which party set the information, nor does it indicate when it was set. In order to deal with this problem, the information is commonly overwritten at the edge of a network, where the data traffic is inspected and mapped onto certain traffic classes. This way, the network operator can rely on the integrity of the provided information. At the same time this prevents applications from using the DS-field for their own purpose.

A second approach leveraging the DS-field for the information exchange between applications and networks is called Integrated Services (IntServ) [7], which was defined in prior to DiffServ. Whereas DiffServ requires no additional state to be maintained by individual network devices, IntServ is based on specific resource reservations within the network. Therefore, affected network devices require the reservation of specific resources such as bandwidth for a request, which is propagated throughout the later path of a packet flow. This already names the most challenging path, connected to the challenges imposed by RSVP, where each device also has to reserve certain resources and maintain state per reservation. Such stageful mechanisms commonly have limitations regarding their achievable scalability.

Active network [97] is an innovative approach of embedding code into packets, which a network device forwarding such a packet can interpret. The code embedded in a packet may therefore directly influence the forwarding behavior of a network device. Even if this approach sounds promising and could provide a great flexibility several challenges have hindered the successful implementation of such an active network. Especially because of security concerns implied with the code execution on network devices [112].

The last approach exploits the possibilities of providing a network interface for an application to couple an application more tightly with the network. A network interface should be centrally available for applications. Since a traditional network is mainly constructed in a distributed fashion, it is difficult to provide a central entity as a host for such an interface. In this context SDN includes promising new concepts, especially for providing an application interface. The SDN northbound API is envisioned to fill the gap of a central interface. However, there is no common universal interface yet [50], nevertheless different concepts [73, 49, 104, 26] have investigated in evaluating the usage of such an interface. Each of those concepts defines its own interface to be used by a specific application to inform the network application related data. They are able to show significant improvements in their specific domain with a reasonable effort. However, because there is no universal interface yet, the interface is highly customized for a specific application. This requires that the network operator controls both the network and the application using the network. This can be the case in a data center environment where the data center owner is aware of this [47].

A recent approach exploits the capabilities of SDN to provide a more generalized interface. The PANE [29] approach defines common features which are useful for a variety of application. An application can request a network resource to be reserved, like bandwidth or access control, it can query the interface for network state information like link speeds. Additionally, an application can provide hints for the network about current or future traffic for an improved network management process. In summary network interfaces for applications can be designed to offer a richer interaction between the network and applications. Previous approaches like the DS-field might offered a communication channel for the information exchange between applications and the network, which was to narrow. The flexibility and freedom of the design of a network interface might overcome this limitation in the future.

The extended capabilities of SDN enabled networks in the domain of application interaction is further discussed in the last Section of this chapter.

5.3 Network Awareness

Network Awareness describes the concept of making an application not only using a network, but also making the application aware of specific network parameters. This network awareness requires two parts in the network. First, the network needs to be able to collect performance related information about its network components. This can include link utilization information, bandwidth provided by a link or delay information of individual routing path. In addition to the collecting of such information, there needs to be a way for the application to retrieve these information about the current network state.

Information reflecting the current network can be used by applications to adopt their network usage accordingly. In case of resource limitations on the network an application might adjust its traffic consumption to overcome those network limitations [118]. The famous voice and video application Skype is reported to adjust its transmission quality according to the network environment [41]. Although adjusting application behavior to the current network state in a network aware manner can improve the QoE of an application, it does not solve the possible reason for a network limitation, neither does it try to identify it [73]. A video stream might be allocated to a congested path with insufficient bandwidth. If the streaming application is network aware, it monitors the network bandwidth and can adjust its transmission speed in case of insufficient bandwidth for providing a fluent playback. The streaming application therefore usually decreases the video resolution or uses a different video codec with a higher compression ratio, both resulting in decreased bandwidth requirements. However, decreasing parameter such as video resolution or changing the video codec also means less quality for the transmitted video data.

In an improved solution the network might be able to forward the application traffic over a different path providing more bandwidth for the video. This way the video can be played in a better resolution leading to an improved QoE of the application. However, the network needs to be aware of the bandwidth requirements of the video streaming application. Adapting application behavior does not necessarily mean to decrease the quality of an application. Network state information can also be used to select network resources for an optimized transmission. A download client could for instance ask for the connection speeds for a set of server hosting the required file. The download client can then select the server with the best connection to download the specific file.

These examples show that sufficient network resource information valuable for improving application QoE. Beside adapting application behavior an application might try to reserve network resources in advance. Therefore, knowledge about the current resource utilization is also required to process such a reservation request.

The network needs to examine if sufficient network resources can be reserved to fulfill a given request based on the current information about the network state. The Remos Network Monitor System [66, 71] aims to provide network resource measurements across different network devices. The evaluation of such a network resource monitoring system emphasizes the importance of sufficient network resource information.

Overview of Existing Network Awareness Approaches

Beside adapting the application behavior or optimizing network resource selection described in the previous Section, Table 5.2 lists additional approaches in the network awareness domain.

The TCP protocol provides different parameters for influencing the transmission performance. Finding the right parameters to improve the transmission performance can be easier taking network state information into account, there are several approaches tuning TCP protocol parameters [106].

Network awareness can be beneficial in P2P applications, especially when in selecting overlay nodes for transmitting traffic [10]. Usually P2P applications select peers on a random basis, instead this process can be optimized using network state information.

Approach	Description	Examples
Tuning protocol parameters	Tuning protocol parameters at the end-host to improve the network performance	[106]
Overlay node selection	Select optimal overlay nodes providing an optimized path for application traffic based on network statistics	[10]
Application adjustment	Applications can adjust on the application layer to overcome network problems	[41]
Resource Reservation	Reserve network resources prior to their usage	[116, 30]

Table 5.2: Application of Network Awareness

A well-known approach deals with the reservation of network resources. The Resource Reservation Protocol (RSVP) [116] describes a protocol implementation to announce reservation request in a network. RSVP distinguishes between a sender and receiver of a data stream. Reservation requests are originated by a receiver rather than the sender, since the receiver should decide which amount of data

it wants to receive from the source. In a video streaming scenario this could be different video resolutions to be received by a TV and a smart phone. The TV might reserve more bandwidth in the network to receive a higher video resolution than the smart phone. RSVP is not the first approach towards resource reservation in a network, however earlier approaches [30] mainly focus on unicast. RSVP also includes mechanisms to reserve resources in a multicast environment, since most multicast applications like IP-TV can benefit from sufficient resource reservation. However, RSVP requires to set up individual reservations in per connection, which raised scalability concerns [5].

5.4 Application Interaction with SDN

Section 2.1.2 provided an introduction into the background of interfaces such as Northbound and Southbound part of the concept of SDN. It is worth noting that a standardized Northbound API does not exist yet and it might not be feasible to implement an appropriate interface, because it highly depends on the targeted application [50]. However, a variety of research focuses on providing an implementation for this interface. The in Section 5.2 introduced approach PANE [29] is a recent step towards the definition of a common instruction set sufficient for the interaction with a variety of different applications. Other approaches such as Procera [102] or Frenetic [32] try to introduce a high layer abstraction for application to perform network operations without dealing with too much low-level details. Multiple work has investigated on the potentials of special purpose implementations for common data center applications [104] or messaging applications such as Microsoft Lync [26]. The software-friendly network approach [112] envisions a modular system in which each application has its own plugin specialized on the requirements of the application to be part of an overall interface. Whether such a modular system, or the right abstraction of the Northbound API will emerge, remains an open question. Nevertheless, each of these works shows the potential of a deeper interaction between applications and the network. Traffic engineering and resource optimization benefit from such an application interaction, keeping this part of SDN a vital part for further research. Beside the promising outlook of the SDN concept in an application interaction domain, each implementation has to deal with limitations mainly imposed by the current OpenFlow hardware as discussed in Section 3.1. Especially limitations affecting the gathering of per-flow statistics in conjunction with the limits in the control bandwidth must be taken into account in designing application interaction. The limitations in the amount of per-flow statistics the controller can pull from the network devices [18] have not negligible impact how accurate the current network state can be measured to be taken into account by further traffic engineering and network management processes. In summary, the SDN concept has great potential in the domain of application interaction, especially because of its novel abstraction and interface concepts. Nevertheless, for a complete integration of application information into the network management task, further research is necessary to overcome current challenges regarding limitations in the data plane.

6 Classes of Application Interaction Concepts

Based on the approaches discussed in Chapter 5, this chapter introduces general concepts for using application interaction in a network. A general concept can be identified, if multiple approaches to realize application interaction have a common underlying concept. Therefore, the approaches for application awareness (Section 5.2) and for network awareness (Section 5.3) have been analyzed for investigating a common underlying concepts. Each concept is described in detail following the structure used in Chapter 4. Each concept is therefore described regarding its constraints and objectives, structure, requirements, applications and tradeoff.

	Resource Reservation	Application State Report	Application Topology	Application Adjustment
RSVP [116]	√			
PANE [29]	√	√	√	
SFNET [112]	√	√		√
Application Aware YouTube Video Streaming [49]		√		
QoE-aware Video Streaming using SDN [73]		√		
Software Defined Multicast [86]			√	
OpenFlow Lync Applications-optimized Network [26]			√	
Skype [41]				√

Table 6.1: Occurrence of Application Interaction Concepts within the Literature

Table 6.1 gives an overview of the four application interaction concepts developed in this work, along with the approaches found in the literature. For each of those approaches the underlying identified concept is marked, which is further described for each concept in the following sections.

6.1 Resource Reservation

Constraints and Objectives

With the emergence of new bandwidth intensive applications like video streaming, the best-effort service model of the Internet does not deliver an appropriate QoS for these emerging applications anymore [116]. Not only bandwidth intensive applications confront the Internet with challenges regarding

sufficient forwarding capacity, also delay-sensitive applications like VoIP applications call for a proper QoS level for transmitting their application data. It is worth noting that in order to provide the right QoS for a particular application the network can be highly benefit from an extended application interaction covering the diversity of application requirements. One possible solution was already identified in the early years of the Internet with the Stream Protocol proposal [31], providing a way to reserve resources in the network. This concept was adopted by later concept proposals like RSVP [116], as well as by recent proposals in the SDN domain [112, 29]. Therefore, the first general concept for application interaction is introduced as Resource Reservation. An application might have deeper knowledge of its resource requirements during its execution. For not burden the network with the complex investigation of resource requirements for each individual application, an application should be able to inform the network in prior about its required resources. In order to process such an resource reservation request the network might check the availability of the particular resources. If enough resources are available to satisfy the request the network can initiate necessary steps to reserve the requested share of resources and subsequently inform the application about the taken reservation. In any other case the reservation is declined, leaving open space for further steps performed by the application.

Structure

An example for illustrating the concept of resource reservation is depicted in Figure 6.1. The example shows a network consisting of four switches ($S_0 - S_3$). The black edges in between two switches represents a connection, a bolder line represents a connection with 100 Mbps bandwidth, the thinner lines represent a connection with lower bandwidth (10 Mbps). The end host $H1$ is connected to the switch S_0 and a second end host is connected to switch S_3. The resource reservation concept works as follows:

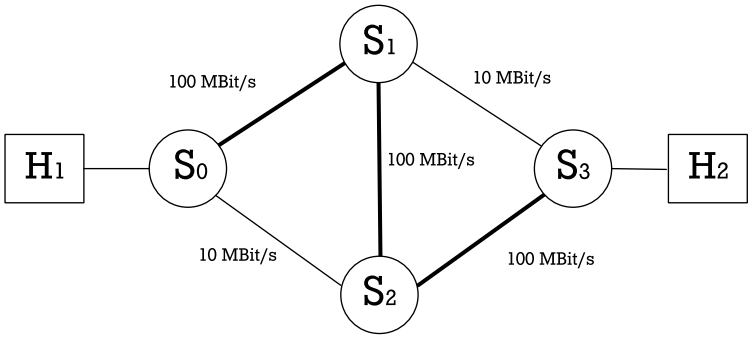

Figure 6.1: Example Network for Resource Reservation (Without Reservation)

Assuming a video streaming application with a server hosting the video file and a client requesting a specific file from the video server. H_1 is assumed to be the client in this example and H_2 is assumed to be the server hosting the video file. The video streaming application knows about the bandwidth requirements for streaming a requested video file with a specific resolution. In this example the client requests a video file requiring more than 10 Mbps bandwidth to be played with a sufficient quality, meaning now stalling or resolution decrease will occur.

Usually multiple possible paths from a source to a destination exists. Out of these possible paths a routing protocol selects one for forwarding traffic from the source to the destination according to a predefined parameters. In most cases a simply shortest path is selected by a routing protocol. In the example of Figure 6.1 the shortest path from H_2 to H_1 would be either $\{S_3; S_1; S_0\}$ or $\{S_3; S_2; S_0\}$. Since

both paths include an edge an edge with only 10 Mbps of available bandwidth, the requirements of the video streaming application can not be fulfilled.

At this point the resource reservation concept can provide a possible solution. Since the video hosting server H_2 knows about the bandwidth requirements, it tries to reserve sufficient resources on a path to H_1. A resource reservation C can be a set of parameters required by an application for a network connection. For instance $C = \{B, D\}$ would describe a resource reservation consisting of a bandwidth B and delay D. Additional parameters can be added to the resource reservation C depending on the needs of an application and the ability of the network to measure each parameter per connection. In other words, the network needs to be able to measure a parameter T for a specific connection in order to be able to process a reservation for the resource specified by T.

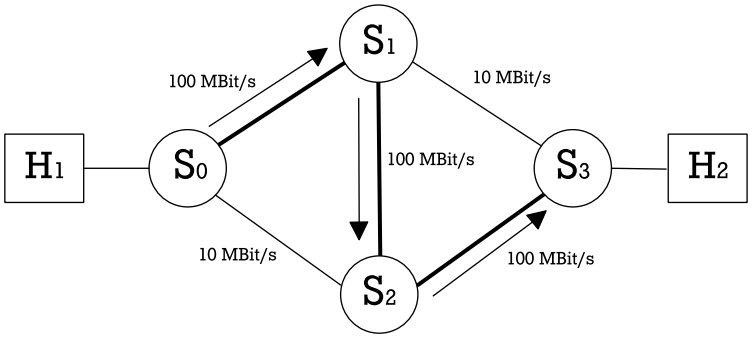

Figure 6.2: Example Network for Resource Reservation (With Reservation)

Subsequently to receiving a reservation request C, the network is required to determine if the C can be achieved by one of the available paths between H_2 and H_1. None of the two shortest path available can satisfy the bandwidth requirements. A longer path $\{S_3; S_2; S_1; S_0\}$ instead, can satisfy the bandwidth requirements of the video streaming application with an available end-to-end bandwidth of 100 Mbps. Figure 6.2 shows the selected path. Therefore, the reservation in advance can force the network to select a path satisfying the application requirements if available. If there is no such path the initiator of the request should be informed about the unavailability of sufficient network resources. Additionally, the answer should specify the amount of resources available for reservation. This way the application can adapt to the decreased resource availability. The video streaming application could reserve less bandwidth instead while lowering the bandwidth requirements for the video stream with a codec providing a higher compression ratio.

Requirements

Enabling resource reservation in a network requires several aspects thoughtfully designed and established. First of all, the network itself needs to be able to determine sufficient information about its current state. More precise it needs to measure all possible parameters which should be reservable by a reservation request C. Additionally, the resource reservation concept benefits from a comprehensive knowledge of the network topology.

A central OpenFlow controller can provide such a comprehensive view of the network topology as well as flexible per-flow statistics which can be used for selecting an appropriate path for a given reservation request.

Without a central controller resource reservation has to be implemented in a distributed fashion, meaning a reservation request is forwarded through individual network devices. Each of these network devices need to understand such an request an will individually check if the request can be satisfied.

Another requirement for a resource reservation concept is fairness. Each connection only has a finite amount of resources available. Assuming multiple clients reach reserving a specific amount of the overall available resources, the sum of all reservation requests can easily outrun the available resources. In this scenario the network requires an efficient conflict resolver. The conflict resolver has to balance incoming requests across all requesting clients. Furthermore, a resource reservation is likely to be abused by malicious clients requesting all available resources for their own. Therefore, trust and security should also be taken into account when implementing a resource reservation concept.

Applications

The resource reservation concept is applied across multiple approaches discussed in the literature. The Stream Protocol (ST) proposal [31] was already developed in the early days of the Internet because of the need for way to provide appropriate QoS for certain applications. This pioneering approach was especially developed to support voice conferencing over the network.

Eighteen years later RSVP [116] was introduced adopting the idea of the resource reservation concept. At the time the ST was designed, no sophisticated multicast routing protocol existed. Therefore, ST had to rely on unicast [116]. With the emerge of multicast routing protocols RSVP was developed with the benefits of multicast routing in mind. Especially because most multicast applications can also make use of a resource reservation approach to ensure a sufficient application quality.

The previous example applications applied the resource reservation concept in the classical network domain, where the distributed nature of the network is one of the greatest challenge. OpenFlow instead provides a central controller with a central network view, which can simplify resource reservation. Therefore, recent approaches take the resource reservation concept into account to apply it for application interaction in the OpenFlow domain.

PANE [29] is an approach for implementing a comprehensive application API on top of OpenFlow. One functionality of PANE enables applications to request a guaranteed minimum bandwidth for a connection between two hosts. The functionality is specialized for bandwidth reservation, thus it does not provide additional connection parameter such as delay to be specified by the application. The PANE implementation was evaluated in use with the coordination service ZooKeeper [43] used for distributed systems and with Hadoop, which is open source implementation of MapReduce [20] data framework. For the evaluation with ZooKeeper additional TCP flows were generated to simulate a congestion scenario, Nevertheless,with the bandwidth reservation functionality provide by PANE the system was able to perform similar to the scenario without additional congestion [29]. For the Hadoop related evaluation PANE was able to decrease the job's completion time of high priority jobs by in average 19%, when the bandwidth for these jobs was reserved [29].

Another approach in the OpenFlow domain is SFNET [112]. Similar to PANE it enables different functionality to be accessed by an application. One of this functionality provides bandwidth reservation. The reservation process implemented by SFNET works as described in the Structure paragraph. The approach was evaluated using a TCP connection between two hosts. Additional UDP flows were generated to simulate a congestion scenario. Without any bandwidth reservation the TCP connection achieved an average transmission rate of 280 kbps, whereas an average transmission rate of 2.24 Mbps was possible using bandwidth reservation in advance.

Tradeoffs

A central tradeoff related to the resource reservation concept is the additional delay caused by the reservation process. The delay varies depending if a distributed approach is used such as RSVP or a SDN enabled approach like PANE. The distributed approach is likely to take more time until each network device on the path has processed the request compared to a central approach where each network device maintains a controller connection, thus can be directly notified by the controller. The delay should be kept as small as possible, otherwise it might not be suitable for short lasting connections to reserve resources, instead only longer lasting connection would use the resource reservation.

Another tradeoff can be identified within the number of clients which can reserve resources in the network. Especially a centralized approach such as PANE or SFNET need to consider scalability as a limitation if a high number of active clients will access the central interface to perform reservation requests. In some scenarios only a subset of all hosts in a network might need to access the interface in order to perform reservation requests. For the video streaming example it might be sufficient to only provide the reservation interface for the streaming server instead of including the client as well. Therefore, scalability concerns can be tackled in two ways. First, the central controller might be extended with additional controller being responsible for a subset of the overall network. In this case open challenges, such as keeping a consistent central network state with multiple controllers, remain. Second, the number of hosts able to access the interface should be limited to a reasonable subset of hosts, without restricting applications from the interface.

This also includes trust and security thoughts, because the clients accessing such a reservation interface should be trustworthy in order to preserve functionality and fairness. Fairness needs to be considered as an additional tradeoff. It should not be possible to reserve 100% of a resource for a certain connection in order to provide best-effort forwarding service for hosts not reserving bandwidth for their data transmission. Resource reservation therefore should be an optional service for applications rather than mandatory. Some connection might last to short for a reservation to be useful or other applications might not be able to forecast their resource requirements.

6.2 Application State Report

Constraints and Objectives

Application awareness approaches like DPI and the Addressing approach mentioned in Section 5.2 try to identify which packet belongs to which application. Both approaches do not take the actual application state into account [50], which might be useful for the network. Different application state could be interpreted by the network to optimize the data transmission. For instance could an application inform the network, that the current transmission speed does not meet its current QoS or QoE requirements, therefore the network can try to find a better path for this particular flow. The objective of the application state report concept is to enable a interaction between an application and the network, in which the application reports its current state to the network. This implies that the network is aware of different application states and has knowledge of how to react at a given application state.

Structure

The application state report concept is based on the observation, that detailed information about the current application state can be useful for the network to optimize resource usage. Furthermore, state information from applications can result in a better QoS and QoE for applications because the network can better adapt to specific application requirements. The structure of the application state report concept is depicted in Figure 6.3.

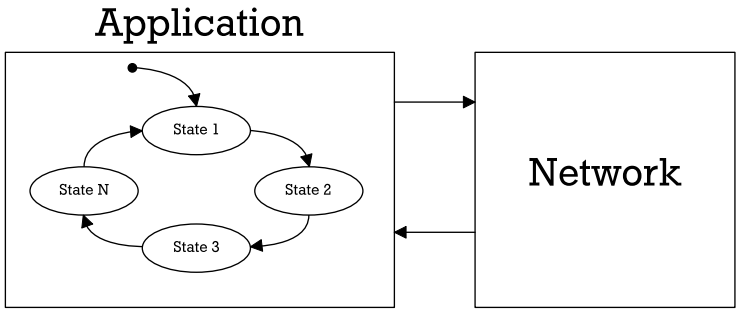

Figure 6.3: Structure of the Application State Report Concept

An application can be assumed to have different distinguished states, during its runtime. The number of distinguished states can vary depending on how fine-grained the application states are maintained. These states can be modeled as a state machine, with an initial $State1$. The transitions from one state to the next state also vary depending the actual application logic. Figure 6.3 illustrates just one example state machine with N states with each state having a transition to the next state in a circular fashion. Each state represents a point during the execution of an application where it can be useful for the network to be informed about this particular state. For instance could an application report to the network, that it is operating in normal mode, meaning all QoS or QoE requirements are satisfied by the current network connection. Another state could indicate an undersupplied scenario, where the application wants to report to the network, that it needs an improved network connection, because individual QoS parameters are not satisfied. Having this state information available in the network can enable the network to optimize forwarding path to improve the QoS for the particular application flow, from which an undersupplied state was reported. An application might need additional network resources only for a specific amount of time, after this it could report being back in normal state to the network. This has two advantages. First, the network can reallocate network resources after a application returned into normal mode. Second, it can operate without any adaptions until an application reports a new state to the network, after which the network can react again on the changed state of the application.

Requirements

For implementing the application state report concept a network requires to provide a way for applications to provide the necessary information. In a non-SDN domain the only possibility is to encapsulate such information within the packet itself. The IP-header provides a so-called differentiated service field (DS Field) which is 8-bit long. For this field an encoding for different service identifiers already exists [77], but it could be reused in order to encode an application state. This field usually is set at the ingress of the network, instead the application itself could set the DS Field according to the application state. However, it imposes some challenges, because each network device has to keep a certain state for individual application states, in addition providing an optimized path, meaning a path with more bandwidth or less delay is difficult with only a local view of a single network device. A SDN controller might fill this gap with its network wide view, making it easier to find certain optimized path. Furthermore, the SDN Northbound API can serve as a central interface for applications to report their current state. The controller can subsequently exploit this information for making network wide path adaptions and path reconfigurations in order to optimize forwarding paths for applications reporting a undersupplied

state. Therefore, the application state report can be seen as a reactive interaction scheme, compared to the previous discussed resource reservation concept, which is a proactive scheme.

As an additional requirement the application and network need a common language for specifying an application state. The network has to be able to understand application state information and requires knowledge about how to deal with those information. Because different applications have different QoS requirements such as bandwidth or delay, along with the state information the application should provide information about itself for the network to identify the application. This helps to provide appropriate treatment for the application data in certain application states.

Applications

Besides an resource reservation feature, PANE [29] provides a feature called hints, which is similar to the application state report concept. With a hint an application can provides useful information to the network such as desired flow-completion deadline, flow's size in bytes or predictability of future traffic. These information could be mapped onto different application states, to be triggered in case a transition to this state is taken. The flow-completion deadline could be a teardown state, which indicates the network that a certain application will stop transmitting data in the near future.

Two studies explore the capabilities of reporting application state information to an controller in a SDN [73, 49]. Both approaches deal with video streaming as an example application and provide similar implementations of the application state report concept. The approach in [73] measures QoE metric at the clients video player and reports these metrics to the SDN controller. If a QoE parameter, such as quality of the video drops under a certain threshold, the application is assumed to run in an undersupplied state, therefore the SDN controller tries to identify a potential bottleneck at the path between the server hosting the video and the clients host. It subsequently tries to avoid the bottleneck by selecting a different path around the bottleneck using MPLS. The approach was evaluated streaming a 1080p video file multiple times with application state information and without providing application state information to the network. In the streaming scenario without available application state information, the client was able to see receive 1080P in 69,08% of the total of 100 minutes streaming length, the rest of the time the player at the client had to decrease the resolution to cope with the network conditions. The application state information used for changing the network configuration resulted in 77,16% of the total of 100 minutes streaming of 1080p content. 480p content was only received for 6,58% of the time compared to 11,18% of the time if no application state was reported.

In [49] the frame buffer of the video player was monitored by a plugin. In case the frame buffer dropped under a certain threshold the plugin reported this buffer underrun to the SDN controller. The testbed setup consisted of two switches connected with five individual lines, which should simulate a multi path setup. Both switch were connected to an OpenFlow controller. One switch was connected with the Internet for receiving the video content from YouTube and the second switch connected the client playing the video to the network. Within their evaluation different approaches of allocating one of the five lines between the switches to a specific flow were tested. DPI was also tested and performed well in identifying the video stream to be switched to a dedicated link. However, DPI was not able to detect when a buffer underrun might have got resolved. Therefore, the video stream was kept exclusive on the dedicated link, resulting in a inefficient network resource usage.

With the help of the application state information, in this case the frame buffer usage, the OpenFlow controller was able to detect when the video stream returned to a normal operational mode, meaning that the frame buffer was filled with sufficient buffered frames. In this case the controller could use the link for carrying additional traffic beside the video streaming traffic, as long as the client does not report

any further buffer underrun. Additionally, the video was played without any stalling events using the application state information.

Tradeoffs

Usually a concept for improving efficiency comes at a cost implied in a tradeoff. In the case of the application state report concept the overhead implied with the concept can be identified as a tradeoff. Even though an application only reports its actual state, in case the state never changes no additional communication is necessary, a network might want to have more fine-grained application related information. Therefore, more states would be defined for providing a more detailed application information base to the network. More states usually result in more state transitions taken by the application during execution, which leads to a increasing communication overhead beside the actual data traffic transmitted by the application. The amount of information provided by the application and the data transmission rate should be balanced in order to limit the overhead caused by the application state information exchange.

Furthermore, an application might try to abuse the concept for its own benefit. Therefore, it could continuously signal an undersupplied state to the network, causing the network to increase the computational power for finding an alternative path with better transmission parameters. Therefore, a conflict resolver like mentioned in the tradeoff paragraph of Section 6.1 might be necessary. Another possible solution is discussed within PANE [29], where a hint provided by an application is only optional and does not imply any action performed by the network.

6.3 Application Topology

Constraints and Objectives

Constraints related to the application topology concept can be illustrated with the history of IP-multicast. Even after several years of research and beside its promising advantages, IP-multicast still struggles to be successfully deployed because of several reasons [22]. Because multicast capabilities are highly valuable for a number of different applications, such as video streaming or P2P systems, approaches for implementing multicast on the application layer have gained popularity [40]. But most of these approaches come at a cost, that the underlying network infrastructure is unaware of the application implementing multicast on top of this infrastructure. This unawareness usually results in an inefficient network resource usage. The application topology concept aims to counter this issue by providing a way to make the network aware of the topology maintained by an application. Exploiting this knowledge can lead to a more efficient usage of network resources to realize certain application topologies.

Structure

A high level example for the current prominent network situation from an application point of view is depicted on the left side in Figure 6.4. Host H_0 is assumed to run an application which sends data to each of the hosts H_1 - H_N. In absence of any IP-multicast functionality inside the depicted network, H_0 has to transmit the data N - times via a unicast stream. From an application point of view H_0 would desire a tree based structure depicted on the right side in Figure 6.4, where the data only requires to be transmitted by H_0 once and duplicated if necessary by the network. In order to achieve such a tree based forwarding structure inside the network, the application running on H_0 could share the desired topology with the network in prior to the data transmission.

Even though informing the network about such a topology introduces communication overhead, it can result in significant performance and efficiency gain for applications using a relatively static multipoint

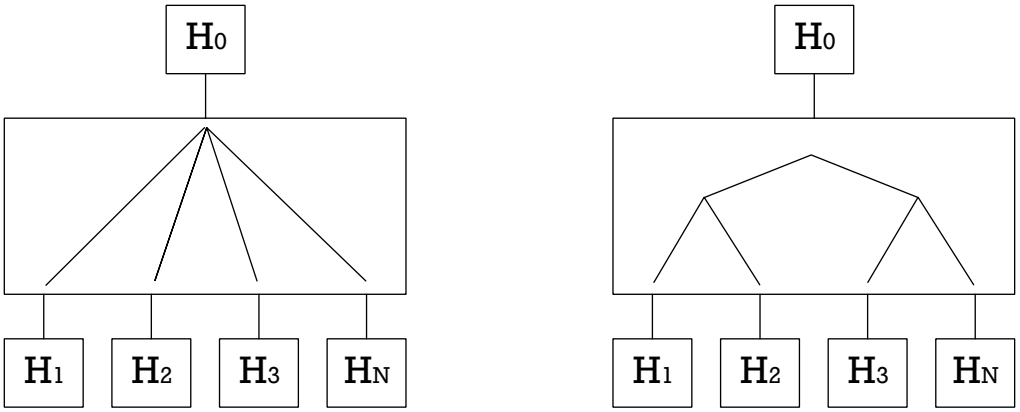

Figure 6.4: Network unaware of the topology (left) compared to topology awareness (right)

communication topology. The concept does not limit the topology to be fixed to multicast traffic. Instead any topology should be realizable by the network depending of the needs of the application.

Requirements

The requirements for the application topology are similar to all other application interaction concepts. The network requires an appropriate interface for the application to exchange the necessary topology information. Besides that, the more crucial requirement lies within the ability of reconfigure a complete set of network devices at runtime of the network. In traditional networks there is no centralized control plane, which makes it complex and time communing to reconfigure a lot of network devices at once. The SDN paradigm with its centralized control plane instead can provide the necessary ability to configure network devices from one central point of control. A consistent and comprehensive view of the overall network is beneficial for implementing network wide topology changes.

For an efficient interaction between an application and the network in order to provide topology information, there is a need for an appropriate description language. The language should be designed to efficiently express a topology and details for communicating special needs of an application.

Applications

Application layer multicast is one example where more application control over the network topology can result in an improved network performance. The authors of [86] discuss a novel approach named Network Layer Software Defined Multicast, which exploits the knowledge of a overlay tree topology to build a highly efficient network layer delivery tree. An OpenFlow-enabled network with a central controller in charge is used for evaluating this approach. The P2P system can therefore interact and provide information to the controller about the connection topology of the overlay network. By adapting the networks configuration towards a more overlay friendly topology, the approach is able to reduce traffic volume compared to a random overlay network topology by 40%, which is similar of what to expect from an pure IP-multicast approach. Making the network more aware of the application and especially its overlay topology can therefore result in a better network resource utilization as well as increased performance.

PANE [29] includes a feature which can be identified as being similar to the application topology concept. The so-called path control enables an application to specify waypoints. A waypoint forces the network to find a path, which includes the specified waypoint. This can be used to force certain traffic to be forwarded through a firewall. The counter part of the waypoint feature is the avoid command. The

avoid command can be used to force the network to avoid certain network devices for application data. In the firewall context this command could be used to enable certain traffic to bypass the firewall, which could be important for a research department for instance.

A feasibility study of an extended Lync messenger implementation was realized in a cooperation between Microsoft and HP [26]. The Lync application was modified to report the addresses and port information of individual communication partners to an OpenFlow-enabled network. This information was subsequently used by the network to be aware of proper QoS treatment for those flows between the reported communication partners. The Lync messenger provides both video and audio communication, which both depend on a certain real-time transmission ensured by the network. Making the network aware of the specific requirements of the application data and its topology has several advantages compared to a system which relies on DPI to identify the application data within the network. It avoids the computational costs of a DPI solution and furthermore enables QoS tagging of encrypted traffic, which would be not detected using a packet inspection mechanism.

The application topology concept can be applied whenever an application relies on a relatively static and priorly known topology for exchanging data between different end hosts. Such applications can be found in data center environments, where a huge amount of data gets processed using the MapReduce approach [20]. According to this approach a huge data set gets split apart and distributed over several processing units, to be processed as individual subsets. After processing each individual subset, the data set is gathered from all processing units to be merged into the final result. The split and merge process required multiple fast connections between the right hosts at one point in time. Since optical circuits become increasingly available in data centers, it is crucial to set up an optical circuit between the root of the data set and the processing units to efficiently distribute the subsets across the processing units. The work [104] discusses an approach for a MapReduce application to report traffic demands and the structure of job distribution to the network. This enables the network to set up optical circuits between the right communication partner, which the study reports to result in an increased job completion time and less reconfigurations of the optical circuits. Applications handling huge data amounts usually also have a central management entity [104], which controls and monitors the execution of the data processing. All useful information about the application and its state is concentrated within such an management entity. Therefore, it is an ideal candidate for being prepared to interact with a central controller of a SDN.

Tradeoffs

Fairness keeps being a major tradeoff implied in each application interaction concept. Making control plane functionalities externally available imposes a careful control over changes requested by applications. This is especially the case in the application topology concept, where an application should be able to influence the internal network topology for its own purpose. Abuse of the functionality has to be considered as a central point of interest when designing and implementing this concept, otherwise network resources could be occupied by an application resulting in an unfairness situation for other applications using the same network.

6.4 Application Adjustments

Constraints and Objectives

Each of the previous introduced application interaction concept includes changes at the network layer initiated by informations provided by an application. Instead, the application adjustments concept focus on using information gathered from the network to be interpreted by an application. The application further bases its decisions related to the network on these informations. Making the application aware

of the current network condition can improve the decision making process of the application logic. A download client could select a server hosting the requested file based on the available bandwidth reported by the network. Another application could be a data intensive process, such as backups which could be rescheduled depending on the current network load.

Therefore, the objective of the application adjustments concept is the realization of an improved network awareness of applications.

Structure

The structure of the application adjustments concept is depicted in Figure 6.5. The application wants to transmit data over the network, but it has different options to choose from, in this case option *A* and option *B*. For a better understanding, these options could represent different codecs for encoding a video file resulting in a bigger and smaller file with a better and less quality. Another example can be to choose from a list of download server providing each providing the same file. In order to download this file as fast as possible the application is interested in the bandwidth and delay currently available for each connection to one download server.

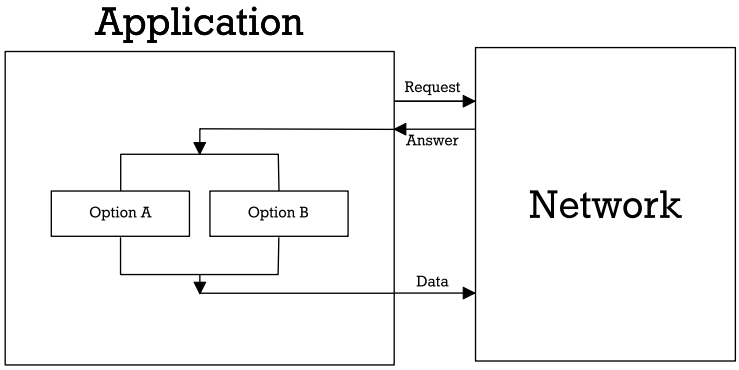

Figure 6.5: Structure of the Application Adjustment Concept

From an application point of view such questions are hard to evaluate without any additional information about the current network state. The interface of traditional networks is limited to provide a socket based interface for the application to transmit its data. There is network resource related information provided to the application while transferring data. For congestion avoidance applications exploit end-to-end measurements, such as implemented in the TCP protocol. Nevertheless,these information are only an indication of the actual network condition, since TCP packets could be forwarded by the network using different path. End-to-end measurements, especially measurements of the available bandwidth on a network path remain a challenging topic [46].

Instead of relying on the application point of view, the application adjustments concept allows an application to request a certain information *C* about the network condition on a specified path. The actual requested information *C* can vary, it could be the available bandwidth, the congestion state or the current delay of the specified path between. The interface should be expressive enough to provide application with enough network state information to be used as input parameters for its decision making process. The application can then select the appropriate option out of its possible option space to use the available network resources in a highly efficient way. If there is only limited network resources available the application can continuously adapt its data consumption. Having multiple applications adapting to the network state can result in a more balanced network resource utilization without having an application directly overloading a network resource and causing a congestion state.

Requirements

As it is the case for all application interaction concepts, the application adjustments concept also requires an interface provided by the network in order to query for the current network state. Beside a common language to specify resource requests and path information, the network requires comprehensive monitoring capabilities. In the traditional network domain there is no inherited functionality to monitor QoS network metrics. Therefore, systems like the Remos monitoring system were developed [65, 66] to fulfill the task of monitoring the current network state, as well as enabling applications to query for those information.

OpenFlow with its implementation of the in SDN envisioned Southbound API can further simplify the collecting of network state information. Nevertheless,collecting sufficient flow statistics across the network imposes challenges. Section 3.1 discussed limitations in the data plane, also affecting the efficient collecting of flow statistics. To cope with those limitations approaches in the OpenFlow domain specifically address challenges related to the flow statistic collection. Both, OpenTM [100] and DevoFlow [18] are two examples for approaches which take these challenges into account.

Beside flow statistics, which can be used to rate the congestion state of a link and its available bandwidth, latency becomes an increasingly important metric for paths inside a network. Especially the QoS and QoE of VoIP applications are sensitive to latency and therefore rely on sufficient information about end-to-end latency. OpenFlow can also be used to gather sufficient latency information within the network [79].

Applications

Skype is a prominent VoIP application widely used all over the world to communicate over the Internet. In order to cope with network limitations, such as bandwidth limitations and higher delays in mobile networks, Skype uses different approaches to adapt its data consumption to the current network condition [41]. If the application measures an increasing packet loss over the network connection it starts sending replications of its data in order to keep a sufficient QoE for the end user. For dealing with bandwidth limitations in the network Skype can adapt its voice codec to achieve higher data compression and decrease bandwidth consumption. It even uses application layer overlay structures to reroute traffic dynamically over third-party machines if the direct end-to-end connection is poor.

An approach applying the application adjustments concept in the SDN domain is called SFNET [112]. Beside its resource reservation capabilities mentioned earlier, SFNET also offers a network congestion status to applications. A congestion status can be beneficial for a variety of applications, such as backup applications which want to transfer huge amounts of data over the network. An application can implement a back-off strategy for its data transmission in case the network is currently congested. The evaluation of the mechanism implemented in SFNET showed the performance improvements gained through verifying the network's congestion status prior to the data transmission. Therefore, a file of 10 MB was first transferred without any congestion. While adding additional data streams to the connection, which should simulate a congestion scenario, the data transfer took in average the double amount of time as the first transfer without congestion. In the last experiment the application first checked the congestion state of the network and only transferred the file if the network shows no congestion. With this approach, the data transfer took similar amount of time as in the network without congestions.

These are only a small subset of possible application adjustments implemented in today's application to deal with changing network conditions. However, both show the potential a comprehensive knowledge of the current network state and appropriate adaption to this state can offer.

Tradeoffs

Adjusting the application resource consumption offers flexible ways to cope with a variety of network scenarios. Beside concepts to increase the application performance, for instance by selecting the best available connection to download a file from a server, an application might be forced to decrease quality parameter such as video resolution or data compression in order to keep the application running in an environment with limited network resources. Therefore, a tradeoff in the adaption process can be identified, meaning that the application might not be able to keep a constant QoS or QoE. Furthermore, adaption does not solve the root of a resource limitation within the network [73], it only helps to deal with the effects of a resource limitation. If a network limitation is theoretically resolvable a better solution might be to exploit the information about the limitation for trying to resolve the limitation, rather than work around it. This ultimately increases the network performance available for the application, therefore the application might be able to resign complex adjustments.

6.5 Integration of Application Interaction

Chapter 4 investigated on introducing classes of resource optimization concepts. Towards the design of a data plane resource optimization mechanism for application-controlled SDN, one of the resource optimization concepts will be elected to be applied on a given network structure. Each of these optimization concepts has certain requirements and furthermore introduces tradeoffs affecting the network. Therefore, this section further investigates on the information the introduced application interaction concepts can deliver to the network and ultimately to the optimization concept. On the one hand the kind of information a certain application interaction concept can delivery should meet requirements of a certain optimization concepts, on the other hand the information should be useful for lowering the effects of the imposed tradeoffs of an optimization concept.

This Section aims to investigate which application interaction concept can provide vital information for which of the resource optimization concepts. This combination is part of the design of an network API, as it is depicted in Figure 6.6. The left box represents the network view and the right box represents the application view. In between these two layers the network API is responsible for the information exchange.

Even though the Figure 6.6 lists all introduced concepts on both the network view and the application view, in reality only a single or a combination of concepts would be applied to the network. For the application view a concept could be imagined realizing even all application interaction concepts, as it was shown in PANE, which realizes the resource reservation, application state report and application topology concept. For the design of the network API in between the concepts two main questions should be considered:

1. Which kind of information is useful for which kind of resource optimization concept.
2. Which application interaction concept can provide this kind of information.

Beginning with the first question, the resource optimization concepts first need to be analyzed regarding their information requirements. Table 6.2 lists all resource optimization concepts along with their possible information requirements. These requirements are extracted from the corresponding requirements and tradeoff paragraph of their introduction in Chapter 4. For each Information requirement the possible information source is listed in terms of the application interaction concept which could provide the necessary information to the resource optimization concept.

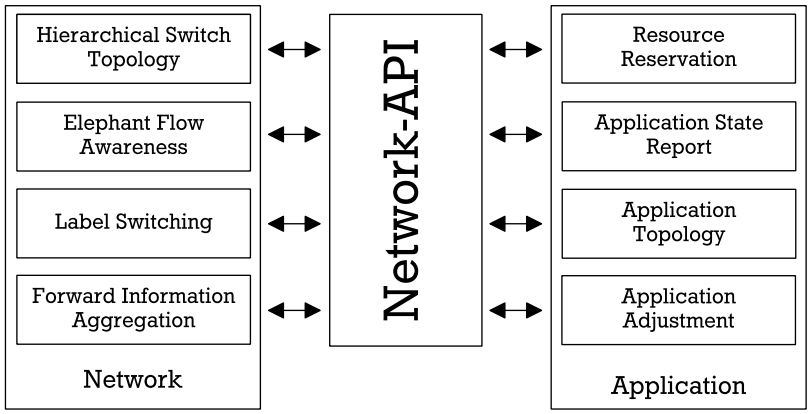

Figure 6.6: Network-API

Concept	Information Requirements	Information Source
Hierarchical Switch Topology	Flow Rule Weight	Application State Report Resource Reservation
Relevant Flow Awareness	Flow Characteristics	Application State Report Resource Reservation
Label Switching	Topology Information	Application Topology Application State Report
Forwarding Information Aggregation	-	-

Table 6.2: Requirements of Resource Optimization Concepts

Hierarchical Switch Topology

The hierarchical switch topology concept can benefit from informations regarding the weight of flow rules, meaning that some flows have specific requirements for instance regarding their delay or bandwidth. Since the partitioning algorithm of the hierarchical switch topology tries to maximize the total weights of all flows assigned to the lowest layer of the hierarchy, the weight of a flow rule is vital for its treatment within the network and ultimately the performance a flow achieves inside the network. In case of a VoIP application, the flow data is sensitive to delay and jitter, therefore the hierarchical switch topology should be aware of those flows, in order to allocate them to lower switching layers in the L dimension defined in the concept structure. In order to be aware of those flows without relying on identification mechanisms such as DPI or machine learning approaches, a VoIP application could specifically announce its flows to the network using the network API. This information is then passed to the controller managing the hierarchical switch topology to initiate or change the flow allocation within the network in order to ensure that the flow rule is installed in the right layer of the switch topology.

The two application interaction concepts which can provide such information are the application state report and the resource reservation concept. Both concepts can provide flow specific information from which the weight could be derived. The resource reservation concepts follows a proactive approach, where applications announce their requirements regarding their flows in advance to the following application data transmission. This can avoid flow setup delays, since the controller might install the flow rules appropriate for the reservation request in advance in the right switch layer. The application state report follows a more reactive approach, in which applications would report state changes to the network. In the VoIP example mentioned earlier, the application state report might only report if the flow requirements are not yet fulfilled by the network, resulting in a bad voice quality and jitter in the con-

nection. In this scenario the network would require to change the flow rule location on the fly during the connection, which imposes high challenges, since the flow rule needs to be migrated on the fly including the risk of affecting the ongoing connection. Such on the fly migration could be avoided with additional state definitions for the application state report concept. A state representing the connection setup could be imagined, reporting the connecting parties during the setup of a VoIP call to the network, which could then configure the path for the subsequent ongoing connection in advance.

Relevant Flow Awareness

The relevant flow awareness concept has a similar requirement as the previously discussed hierarchical switch topology concept. Since the concept needs to distinguish flows in two categories, the concept can benefit from information regarding the concrete flow characteristics. The concept distinguishes flows using three parameters F_D, F_P and F_B, with F_D specifying the duration, F_P specifying the number of packets and F_B specifying the number of bytes per packet. For some applications these parameters might be known in advance by the application itself. A video streaming application for instance is aware of the amount of data it requires to transmit as well as the duration estimated by the playback time of a video. Since the relevant flow awareness concept relies on a early point of detection, the application itself would be the best information source to be aware of upcoming relevant flows.

The resource reservation concept as well as the application state report concept can act as a source for information regarding upcoming relevant flows. Both concepts could be implemented using the parameter structure given by the relevant flow awareness concept. Potential relevant flows could be derived from the resource reservations requested by an application, giving a central controller knowledge about the flows before they actually become present in the network. This can enable proactive flow rule setup for preserving control bandwidth.

The application state report concept can also provide functionality to be integrated with the relevant flow awareness concept. During execution of the application the application state might change to a state where a specific flow of the application might need to be visible for the central controller to perform investigate on possibilities to reallocate the flow to a different path in order to increase the bandwidth. Since in the relevant flow awareness concept the controller is mainly focused on relevant flows and assumes that mice flows get handled without special intervention of the controller, the application requires to make the controller aware of its flow. The application could announce its flow during the state transition as being undersupplied forcing the controller to perform certain optimization actions on this particular flow.

Label Switching

The label switching concept enables the possibility to proactively install flow rules across the intermediate and egress switches in a network. Following the concept proposed in [45], a central controller can calculate the complete path for an newly incoming flow and make the ingress switch inserting this path information completely into the label assigned to the packets corresponding to this flow.

Therefore, one optimization potential can be identified in the path determination of the central controller and the subsequent flow install process in an ingress switch. For proactive flow rule installation in an ingress switch, the controller requires information about the source and destination of a specific flow. Such topology related information could be delivered by the application topology concept. Following this concept an application with a specific topology, such as a multicast tree or a topology connecting reduce nodes for a MapReduce job, announces the communication endpoints to the network in advance. A central controller could leverage this topology information in order to calculate spanning-trees from each source to destination. Each tree would be implemented installing the corresponding flow rules into the

ingress switch of each root of a spanning-tree. This proactive flow installation avoids delay potentially affecting the flow forwarding performance.

Similar information could be provided by the resource reservation concept, where the topology can be derived from the reservation request. Beside the topology information the reservation request also includes vital information about the required network resources, such as bandwidth. This information can be taken into account by the label switching concept when constructing the path for specific application data.

Another key factor of the label switching concept is the ability to efficiently reroute traffic onto a different forwarding path, leveraging the fact that the path is completely defined at the ingress switch. Therefore, the controller would need to change the flow rule, especially the action part, at an ingress switch in order to forward traffic on a different path. This ability can be vital for reacting for instance on reported congestions on specific path. The application state report concepts can benefit from this ability, because it relies on the network's ability to change the current forwarding situation in case of reported undersupplied states. A video streaming application might be allocated to a path where the traffic experiences a congestion within a certain section of this path. Following the application state report concept, the application would report this issue during its transaction from a normal state to a potentially undersupplied state. As a reaction the central controller can investigate on the congestion on this path and change the forwarding path for the application data around the congested section, without having to change flow rules in each switch along the path. Instead, the label information assigned to the application data at the ingress switch requires to be modified to express the new forwarding path.

Forwarding Information Aggregation

The forwarding information aggregation concept has special implications, since there can be no real information requirements identified, which would be useful for a more efficient aggregation concept. Therefore, none of the discussed application interaction concepts provide any benefit to be exploited in adding one of the concepts to the forwarding information aggregation concept.

	Resource Reservation	Application State Report	Application Topology	Application Adjustment
Resource Optimization Concept	Application Interaction			
Hierarchical Switch Topology	√	√		
Relevant Flow Awareness	√	√		
Label Switching	√	√	√	
Forwarding Information Aggregation	√	√		

Table 6.3: Integration of Application Interaction

7 System Design

This chapter introduces a system design for a dynamic network service chaining system (Section 2.3.2). The service chaining system is selected as a system for applying optimizations, since constraints (Section 7.1.1) regarding the network resources of such a system can be identified for having potential for being further optimized. Those optimizations are applied using the three classes of optimization concepts introduces in Chapter 4.

The first applied optimization concept is the label switching concept, which aims to reduce the flow table space requirements of the core switches, which interconnect edge and several service node switches with each other within a service chaining system. One tradeoff of the label switching concept is its focussed workload, where the majority of flow operations are necessary at the edge switches of the network. Therefore, the hierarchical switch topology concept is applied in conjunction, in order to help distribution the workload imposed as flow operations over some switches within the network, which have additional processing power and flow table space available, such as the core switches. The third optimization concept is the relevant flow awareness concept, which aims to optimize the path selection within the system, in order to select the forwarding path for certain flows based on certain requirements, such as delay sensitivity or required bandwidth.

Furthermore, Section 7.5 describes possible application interaction concepts, which can support the corresponding resource optimization concepts, by offering application related information. It is worth noting that the majority of optimizations in the scope of OpenFlow proposed in the literature start their optimizations from a control plane perspective, where the data plane requirements are mostly considered later in the design process [95, 102, 23], which can be referred to as an top-down approach. Instead, this system design approach follows a bottom up-approach, which starts from the data plane perspective, considering data plane requirements and especially the data plane resources defined in Section 3.2 at the beginning of the design process.

7.1 Scenario

The proposed system design, following a bottom-up approach with a string focus on optimizing resource efficiency, is arranged in the scenario of network service chaining [62]. The concept of dynamic network service chaining and the contribution of OpenFlow within this topic is discussed in Section 2.3.2. Even though, OpenFlow addresses some of the challenges of implementing a dynamic network service chaining challenges regarding the scalability remain for further investigation. These challenges are mainly connected to the requirement for maintaining large number of fine-grained flow rules within the network, in order to cope with the requirements given by the dynamic network service chaining. Along with the high number of fine-grained flow rules to be maintained, another challenge arises, which is discussed in the following in more detail.

7.1.1 Constraints

Since network service chains are specifically deployed and maintained for user specific traffic, the complexity of maintaining fine-grained flow control even with OpenFlow increases with a growing number

of users. Furthermore, a network service chain might require traffic distinction based on certain applications and users. For instance web related traffic of a user accessing the websites through a mobile device is subject to certain proxy services, whereas video content for the same user requires content caching services. Therefore, the number of different fine-grained flows to be controlled by the system even more increases with the number of users and number of diverse applications. In summary, at least one flow rule per user is necessary in order to provide the appropriate network service chaining treatment to the users traffic.

With respect to the limitations discussed in Section 3.1, implementing dynamic network service chaining with OpenFlow is likely to introduce challenges regarding the available flow table space for maintaining an increasing number of fine-grained flows [117]. Those fine-grained flows are necessary for the system to maintain a control of the overall traffic in a fine granularity, because user specific or even application specific traffic requires its own network service chain to be forwarded through. Therefore, the scalability of such a system highly depends on the systems ability to provide sufficient flow table space in the OpenFlow-enabled switches.

In addition to the space requirements, maintaining such large flow tables throughout multiple switches in the network is likely to suffer from an insufficient control bandwidth, which is limiting the possible flow setup and modification rate. A higher number of flows to be maintained by the switch results in a higher usage of the control bandwidth, since more potential flows need to be setup initially and afterwards require potential modifications initiated by the central controller. Modifications can be assumed to happen frequently caused by changing requirements in terms of changing network services assigned to a users network service change, as well as completely newly deployed network service chains for newly joined network users.

7.1.2 Objectives

Therefore, a system which will have to deal with large flow tables and frequent flow rule modifications, requires special thoughts to put into a concept to preserve control bandwidth. In summary, the proposed system design should fulfill the following five objectives:

1. Sufficient flow table space: Providing sufficient flow table space for storing fine-grained flow rules.
2. Avoid redundant flow rules: Avoid redundant storage of fine-grained rule within the network.
3. Fine-grained flow rules: Optimize the system for efficiently maintaining a large number of fine-grained flow rules.
4. Flexible forwarding behavior: Efficient configuration and modification of the forwarding behavior throughout the network.

The first up to the third objective is related to the data plane resource flow table size introduced in Section 3.2.2. The fourth objective is mainly controlled by the efficient usage of the data plane resource control bandwidth, which is introduced in Section 3.2.1. The fifth objective can be addressed using the knowledge summarized in Chapter 6.

7.2 Bottom-up System Design

The system design is developed following a bottom-up design approach, where in the first step the data plane resource constraints are identified. In the second step, the data plane resource optimization concepts are selected, which can address those identified limitations. These concepts are adapted to the

requirements of the service chaining system, such as supporting the path construction including a set of service instances.

7.2.1 Identifiying Data Plane Resources

As a first step towards a resource efficient system design, the resources which mainly contribute to the current limitations of implementing a dynamic network service chaining system based on OpenFlow need to be identified. Table 7.1 lists the data plane resources along with a description why these resources need to be taken into account during the system design. The last column describes the consequences caused by the limited data plane resources.

Resource Name	Description	Consequence
Control Bandwidth	Bandwidth available for control traffic to setup and modify flow rules within a switch.	Limits the flow setup and flow modification rate per switch and ultimately for the complete network.
Flow Table Size	Capacity of the OpenFlow-enabled switches for storing flow rules.	Limits the overall scalability of the system, which is necessary to support a growing number of user and application requirements.

Table 7.1: Limited Data Plane Resources

In order to flexibly configure and modify the forwarding configuration of all OpenFlow switches within the system, two operations occur frequently during the execution of the system:

First, new network service chains are setup for new users and applications, therefore the appropriate forwarding flow rules need to be setup in each switch along the predefined path of the traffic. The traffic needs to be forwarded to the first service instance and after passing each following service instance, the switch in charge of the outgoing link of each service instance need to know where to forward the traffic next. This can be either another service instance or a normal network such as the Internet, in case the traffic reached the last service instance for a specific network service chain.

The second operation reconfiguring a network service chain. This operation takes place whenever the requirements for applying specific services to user or application specific traffic are changing. This also includes the failure of network services described in Section 9.1.3. Therefore, a specific service instance might need to be removed from a network service chain, an additional service instance might need to be added to a service chain or the order in which the service instances are applied to the traffic might change. In any case the flow rules within the switches interconnecting the service instances of a specific network service chain need to be modified by the controller. In addition, the flow rules forwarding traffic towards the network service chain might be also subject to modifications, in case the first service instance changes or is removed from the chain. To sum this up, joining and leaving of users and applications, as well as changing requirements of the network service chains require frequent flow rule modification in all switches along a path starting at the users device up to the end of the service chain, where the traffic is usually forwarded to the Internet.

Whereas the discussion was mainly concerned about the available control bandwidth necessary for executing the necessary flow rule setups and modifications by the controller, the flow table space is the second data plane resource, which plays a key role for a scalable dynamic network service chaining implementation. Network service chains are deployed specifically for individual users and/or even specific

applications executed at the user side. Implementing such fine granularity in OpenFlow requires a high amount of fine-grained flow rules matching the number of unique user and application combinations. Since both, the number of users, as well as the number of diverse applications are constantly growing, the scalability of the system depends on its ability to store a sufficient number of fine-grained flow rules.

7.2.2 Identifying Resource Optimization Concept

After identifying the affected data plane resources, the next step towards a resource efficient system design, is based on selecting an appropriate resource optimization concept from the different classes defined in Chapter 4. Since control bandwidth and flow table space are the main resources, which affect the overall limitation of the system, the label switching concept is a promising optimization approach, since it addresses both resources. The label switching concept therefore concentrates the main workload in terms of flow rule density and the maintenance of those flow rules to the ingress switch of a network. While this can save both control bandwidth and flow table space at the intermediate switches, it puts a higher workload on the ingress switch, which might overload the capabilities of an ingress switch in a scenario such as the dynamic network service chaining.

Optimization Concept	Optimizing Resource
Label Switching	Control Bandwidth; Flow Table Space
Hierarchical Switch Topology	Control Bandwidth; Flow Rule Capacity
Relevant Flow Awareness	Forwarding Capacity

Table 7.2: Selected Resource Optimization Concepts

For this reason, a second resource optimization concept is used in combination with the label switching concept, to rebalance the workload requirements for the ingress switches of the network. In order to share both the workload in terms of flow rule related operations, such as flow installation and modification and the flow table space requirements, the hierarchical switch topology is a possible candidate to do so. Its ability to increase the flow rule capacity, by partitioning the flow table space between multiple switches, can help to reduce the workload the ingress switches. Therefore, the system should be able to dynamically transfer flow rules from ingress switches towards a next hop switch, in order to use free flow table space in those intermediate switches. This way the usable flow table space with a combination of ingress and intermediate switch can be considerable increased compared to the available flow table space of just a single ingress switch. Along with the hierarchical switch topology, the relevant flow awareness concept can beneficially support the selection of flow rules, which should be considered to move to a second switch layer first.

Table 7.2 lists all three selected resource optimization concepts, along with the resource subject to the optimization efforts. More details to the application of both resource optimization concepts are given within the next section, which describes the architecture of the system.

7.3 Architecture

This section gives an overview of the architecture of an example dynamic network service chaining. The architecture consists of several switches, which have different requirements, regarding their capabilities such as forwarding speed or flow table space (Section 7.3.3).

7.3.1 Overview

A simplified overview of the dynamic network service chaining concept applied in a mobile network scenario is depicted in Figure 7.1. The topology is based on the topology proposed by other approaches in the area of dynamic service chaining systems [117, 6]. In this example topology, there are two mobile hosts H1 and H2 connected via a mobile network such as a cellular network. This network is connected to the core network layer via the OpenFlow-enabled switch $\texttt{Edge\_Switch}_1$. The two OpenFlow-enabled core switches $\texttt{Core\_Switch}_1$ and $\texttt{Core\_Switch}_2$ are interconnected and connect several service instances. The range of service instances includes legacy service functions, such as DPI and firewalls, several virtual machines (VM) hosts, running multiple Service Instances label as S_N within separate VMs. Each host runs an virtual software switch VS_N which interconnects all running VMs on this host with each other an with the core network.

In addition, routing service functions are running inside a VM, which is also connected through a virtual software switch $\texttt{VS}_4$ to the core network layer. A routing service function adds decision making capabilities to the network service chaining concept. Instead, of having statically ordered interconnected service instances for a specific user, a service instance might be ignored for some parts of the users traffic, therefore a routing service function can decide based on the traffic where to forward the traffic next.

Each service instance has one ingress and one egress link, with which they are connected to an virtual software switch. Service instance implementing routing service functions have multiple links, in order to implement their decision making process. The destination hosts H3 and H4 are somehow connected to the Internet, which is connected with an edge OpenFlow-enabled switch $\texttt{Edge\_Switch}_0$ to the core network.

Since service functions process header information and may change an arbitrary header field on their purpose, the forwarding functionality of the system should not rely on a specific header field. Instead, it must be able to identify user traffic across multiple service functions. However, introducing a special network service header, which carry the required information for identifying the user and the path is a recent approach for this challenge [82].

7.3.2 Motivating Example

Dynamic network service chaining is implemented within this example network the following. Assuming host H1 wants to communicate with host H3, H1 will have to authenticate itself as a valid user for using the mobile network connection. At the same time a network service chain is associated with H1. For this example the network service chain for H1 consists for simplification reasons of S1. The resulting path is depicted Figure 7.2. For explaining all relevant concepts, this minimal path includes enough components to point out every major part of the system design.

The network has to ensure ensure that traffic originated from H1 is always forwarded through each service instance in the correct order, as specified for this network service chain. Response traffic from H3 must take the same path back, therefore the network must ensure that the network service chain assigned to H1 is also traversed by the responding traffic coming from H3 to H1.

For simplification reasons the numbered arrows only indicate the traffic direction from H1 to H3. The traffic on the reverse traverse each switch and service instance in the opposite direction. In an OpenFlow context each arrow requires one flow rule specifically matching the traffic from H1 in each switch to connect an incoming arrow with an outgoing arrow. Taking the response traffic into account one additional flow rule for each forwarding flow rule is required. This results in two required flow rule

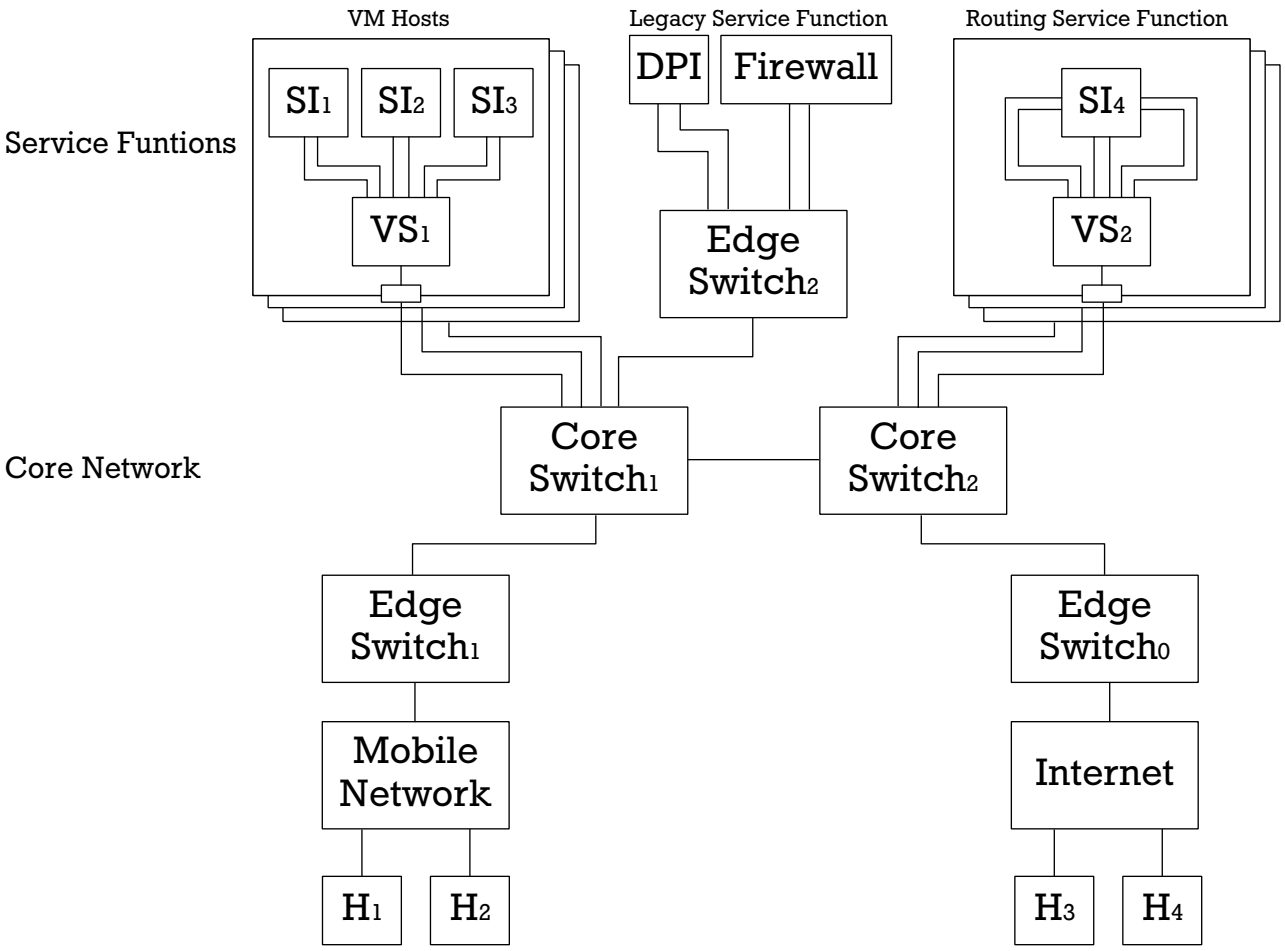

Figure 7.1: Dynamic Network Service Chaining

in $\texttt{Edge\_Switch}_1$, which forwards traffic from `H1` to $\texttt{Core\_Switch}_1$ and the reverse direction. The $\texttt{Core\_Switch}_1$ switch now has to forward the traffic to $\texttt{VS}_1$, which will forward the traffic through the service instance $\texttt{S}_1$. After the traffic went through $\texttt{S}_1$, $\texttt{VS}_1$ forwards it to $\texttt{Core\_Switch}_1$, which forwards it subsequently to $\texttt{Core\_Switch}_2$. The core switch simply forwards it to $\texttt{Edge\_Switch}_0$, which is connected to the Internet.

In total five forwarding steps between $\texttt{Edge\_Switch}_1$ and $\texttt{Edge\_Switch}_0$ are necessary to forward traffic according to the specific order of service instances in this example topology. Even though the flexible flow-based forwarding scheme of OpenFlow-enabled such forwarding behavior including loop backs to $\texttt{Core\_Switch}_1$, it requires an amount of fine-grained flow rules installed on each switch in the topology in order to forwarding specifically the traffic of `H1` through this example network service chain. Table 7.3 lists the number of flow rules required for deploying the network service chain as discussed in the example for both forwarding traffic and reverse traffic.

From the numbers listed in Table 7.3 the following assumptions regarding the flow table space requirements of such a dynamic network service chaining concept can be derived.

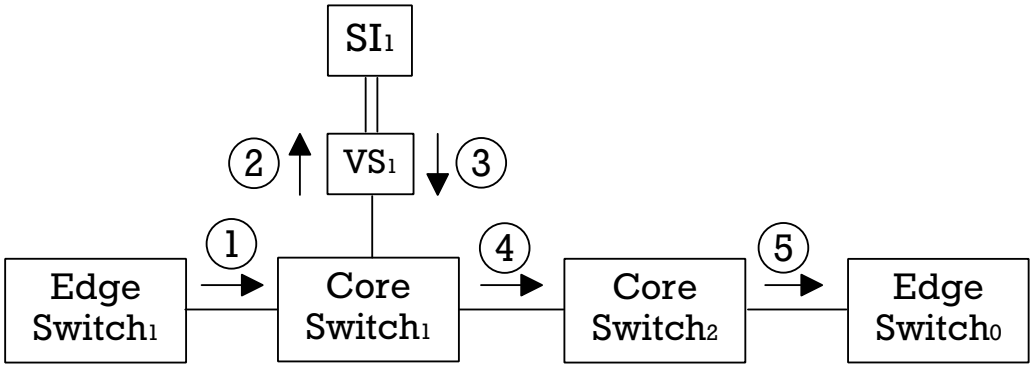

Figure 7.2: Network Service Chaining for Host H1

Switch	Number of Flow Rules
$\texttt{Edge\_Switch}_1$	2
$\texttt{Edge\_Switch}_0$	2
$\texttt{Core\_Switch}_1$	4
$\texttt{Core\_Switch}_2$	2
$\texttt{VS}_1$	2

Table 7.3: Flow Rule Requirements

The flow table space requirements grow linearly with the amount of users on the edge switches, which connect the user to the core network, such as $\texttt{Edge\_Switch}_1$.

The number of flow rules required on an edge switch connecting multiple service instances such as $\texttt{VS}_1$, grows linearly with the number of service instances attached to it. Each service adds two flow rules to the switch for each user having the service instance as a part of its network service chain. Having a high density of service instances within a VM therefore imposes high flow table space requirements on the virtual software switch connecting interconnecting all service instances.

The number of flow rules on a core switch grow with the number of users and the number of service instances within a network service chain of the users. Each flow rule stored on a switch connected to a core switch is required to be stored on the core switch as well. Therefore, the core switches have high requirements regarding the flow table space.

A high number of flow rule entries within a table often implies a high amount of possible flow modification messages send from the controller to a switch in order to maintain a flow rule set covering the forwarding requirements of multiple users and their assigned network service chain. In the example depicted in Figure 7.2, it can be assumed that the DPI and Firewall service instance should be removed from the network service chain of H1. Therefore, the traffic does not need to be forwarded to $\texttt{Edge\_Switch}_2$ and can be Instead, forwarded directly forwarded to $\texttt{Core\_Switch}_2$ after passing $\texttt{S}_1$ and $\texttt{VS}_1$. The controller would require to modify the flow rule stored in $\texttt{Core\_Switch}_1$. The flow rules previously forwarding traffic to $\texttt{Edge\_Switch}_2$ should be removed accordingly, in order to save important flow table space. The same applies for the flow rules related to traffic from H1 stored in $\texttt{Edge\_Switch}_2$. As potentially new user frequently join and leave the system, the network switches, especially the switches in the core network are constantly faced with frequent flow modification demands.

7.3.3 Switch Roles

In the network topology for dynamic network service chaining depicted Figure 7.1 and with respect to the example network service chain depicted in Figure 7.2 two different switch roles can be identified.

First, the role of edge switches either located on the user side or at the Internet connection side. The virtual software switch as well as the edge switch connecting the legacy service functions are also part of the edge switch role. The second switch role includes the core switches. In the following, the requirements and the implications for further optimization potential is discussed for both edge and core switches.

Edge Switch

Edge switches require a high amount of flow table space in order to store the flow rules per user. The same applies for the edge switches interconnecting the service instances, since each service instance requires two flow rules per user and service instance. In order to cope with the high amounts of flow table space virtual software switches are running inside each VM host to interconnect the service instances, since they can offer reasonable flow table space [42]. For the edge switches sufficient flow table space is required, since an edge switch is imposed with a high amount of flow rules. For this purpose virtual software switches can be used or hardware switches, which are equipped with sufficient memory in order to store the flow rule entries. The drawback of having software based switches running on commodity hardware, such as an Open vSwitch, is the lower achievable throughput in comparison to an OpenFlow-enabled hardware switch. Therefore, the edge switches interconnecting the legacy service functions, as well as the mobile network and the Internet to the core network layer should be implemented using OpenFlow-enabled hardware switches providing reasonable flow table space. At the same time they should provide sufficient performance to implement necessary OpenFlow actions, such as adding VLAN-IDs to a packet. The amount of actions a switch can perform in hardware depends on the specific model and switch vendor. Some switches can implement the majority of OpenFlows action functionalities on layer 3 and upwards in software only, which comes at the cost of a slower forwarding performance.

Core Switch

Core switches on the other side should be selected with a high throughput capability in mind, since they interconnect all edge and service node switches and therefore have to deal with a high number of flows. OpenFlow-enabled hardware switches with high throughput performance are available [42], but they show limitations within they available flow table space and control bandwidth. Since Table 7.3 showed an linear increase of flow table space requirements for core switches, a central optimization goal for the system design must be to reduce the flow table space requirements for core switches, in order to make hardware switches with their resource constraints applicable for the core switch role. [42] lists a HP ProCurve J9451A hardware switch, with a limited flow table space of 1500 flow table entries with a flow setup rate of only 40 flows/s, which has a switching fabric speed of 96 Gbps. As this specific example hardware outlines, a tradeoff for having reasonable forwarding performance within the core layer, can be limited flow table space and flow setup rate available at recent generation of OpenFlow-enabled hardware. Therefore, finding a balance in order to exploit the different capabilities of both hardware and software switches is one additional objective for a system design.

7.4 Resource Optimization Concepts

Section 7.2.2 identified possible resource optimization concepts to be applied within the system design in order to enhance and optimize the dynamic network service chaining concept. In the following, the two resource optimization concepts are discussed regarding their actual implementation within the system design.

7.4.1 Label Switching Concept

The label switching concept discussed in Section 4.3 has two benefits for the dynamic network chaining topology. First, it can reduce the flow table space requirements for core switches and second it can reduce the amount of flow modifications required for reconfigure a path, since the path is predefined at an edge switch. In addition the label switching concept.

The label switching concept, inspired by the SwitchReduce [45] concept, is applied to the network topology depicted in Figure 7.1 as follows: Each edge switch in the network topology is also an edge switch for the label switching concept structure 4.3. A label switched path always has an ingress switch and an egress switch. In case of the dynamic network service chaining topology there a several label switched path between the users edge switch and the edge switch connected to the Internet. A label switch path describes the network path between the ingress, intermediate and egress switch of the label switching concept.

The first path consists of the users edge switch label $\texttt{Edge\_Switch}_1$, which acts as an ingress switch, the core switch $\texttt{Core\_Switch}_1$ represents an intermediate switch and the egress switch for this path is represented by the virtual software switch $\texttt{VS}_1$. Therefore, a labeled switch path is constructed between each user and the first service instance. Since the service instances should not be aware of the labeling information included in the packet header, a label switched path is not spanned across multiple service instances. Instead, between each two service instance, which are adjacent service instances within a network service chain, a label switched path is constructed.

The last label switched path is constructed between the edge switch connecting the last service instance in the network service chain and the edge switch connected to the Internet, in the example depicted in Figure 7.2, this is $\texttt{Edge\_Switch}_2$ and $\texttt{Edge\_Switch}_0$. The switch $\texttt{Core\_Switch}_2$ is the intermediate switch of this label switched path.

Label Information

VLAN-IDs are used to identify the action which should be performed by a switch in a hop by hop fashion. Therefore, the ingress switch requests the central controller for calculating a path to the egress switch connected to the destination of a flow. The controller calculates this path with the help of its central view. The resulting path must include all service instances required for the users traffic. The complete path is the divided into subpaths, following the structure of labelled switched path described previously.

The path information, such as which output a packet takes at each switch at each hop is decoded as a VLAN-ID stack. Each VLAN-ID represents one action to be performed by a switch on the path. Each switch on the path therefore takes the top of the stack VLAN-ID and searches for a flow rule in its flow rule table matching the specified VLAN-ID. After determining which action to perform for the matched VLAN-ID, the packet is forwarded to the output port assigned to the action in this specific flow rule. Before the packet is forwarded, the top of the VLAN-ID stack is removed.

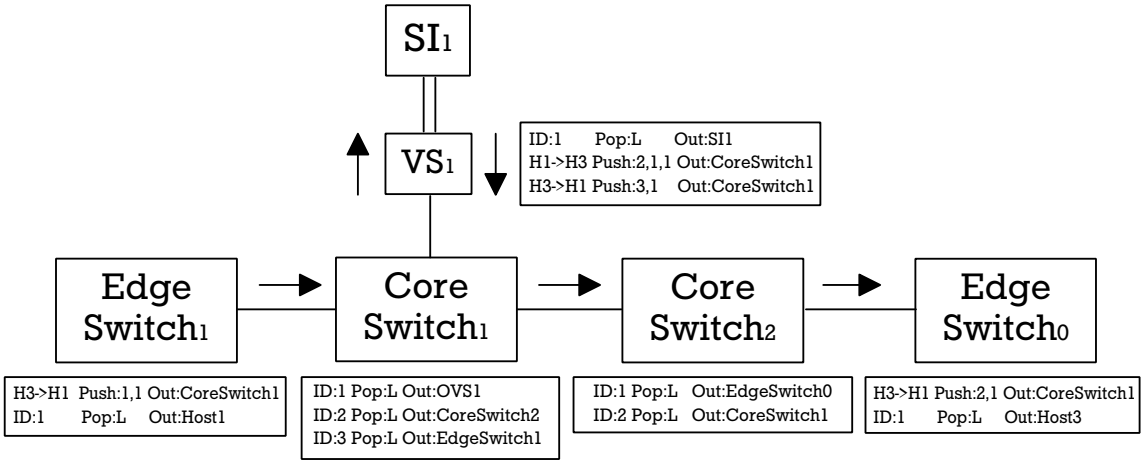

Figure 7.3: Network Service Chaining for Host H1

Example of VLAN-ID stacks

This process and the resulting flow tables are depicted in the example in Figure 7.3. The flow rule syntax is simplified for an easier understanding of the label switching concept, but can be expressed using OpenFlow in a real implementation. In this example a flow rule either matches on source and destination addresses, H1 → H3 for traffic coming from H1 and originated to H3 or an ID, such as a VLAN-ID 1. An action of each rule can be either pop the top the VLAN-ID stack, which is indicated with Pop:L or to push a number of VLAN-IDs to the packet, such as Push:1,1. Each rule additionally is associated with an out action, which outputs matching packet to a specific port, in this example the ports are labeled with the connected switch.

$\texttt{Edge\_Switch}_1$ maintains a flow rule matching incoming traffic from H1. This flow rule adds the VLAN-ID stack 1,1 to all incoming packets from H1 and forwards them to $\texttt{Core\_Switch}_1$. The core switch $\texttt{Core\_Switch}_1$ maintains a set of just three fixed flow rules, which cover all possible output ports of the switch. Therefore, the matching space is bounded by the forwarding action space. The switch pops the top element from the VLAN-ID stack and searches for a matching flow rule. For the VLAN-ID 1 its table says to forward the packet to $\texttt{VS}_1$. Since $\texttt{VS}_1$ is the egress switch for the flow coming from $\texttt{VS}_1$, it also removes the top and also removes the last remaining element of the VLAN-ID stack, which is again 1. This VLAN-ID is matched against its internal flow table, where the associated action with the matching rule with ID 1 is to forward the packet to service instance $\texttt{SI}_1$, which is directly connected to $\texttt{VS}_1$.

Since the service instances should not be necessarily aware of the VLAN-ID stacking system, all VLAN-IDs are removed from the packet at this point. After the packet is processed by $\texttt{SI}_1$, it forwards it back to $\texttt{VS}_1$. Since all path relevant label information was removed before the service instance, the switch now needs to add the path information again to the packet. It therefore maintains a rule matching packets coming from $\texttt{SI}_1$ to add a VLAN-ID stack to them, which will guide them through the remaining network.

Resource Optimizations

Applying the label switching concept has a major impact on the core switch roles. Each core switch can be proactively equipped with a fixed rule set. This is due to the fact that the number of flow rules of intermediate switches can be bound by the actions such a switch requires to perform. Limiting the action set to the forwarding action, this results in bounding the number of required rules for core switches

to the number of adjacent switches. As long as the topology of the switches and the number of links between the intermediate switches remain unchanged, the flow rule set requires no changes as well.

In addition, since the flow rule sets are static, the only flow modifications required have to happen at the edge switches. Therefore, the core switches have less load on their control bandwidth.This is especially vital, since the core switch role is usually represented by hardware switches, which have higher constraints in their control bandwidth, since their are more optimized for providing fast forwarding performance.

Since all path information is stored within the flow tables of the edge switch roles, the most workload in terms of flow modifications and flow table space requirements is located at the edge switches. Especially the extensive packet modification, such as add the VLAN-ID stack carrying the path information adds a high amount of workload to the edge switches. But this tradeoff is necessary in order to free up flow table space at the core network, as well as control bandwidth usage. But the resources of edge switches in terms of flow table space and control bandwidth are also not infinite, therefore the workload concentration on the edge switches can lead to a bottleneck.

7.4.2 Hierarchical Switch Topology Concept

Since the label switching concept introduces a major tradeoff, such that the workload is concentrated at the edge switches of each path, a possible solution could be to support the edge switches by adaptively move flow rules in a controlled way to an adjacent switch. The edge switches should be supported by additional switches, in order to share the workload imposed by the label switching concept. Since optimizing approaches should not require additional switches, it would be beneficial to reuse unused resources from elsewhere in the network.

With the hierarchical switch topology concept such a workload sharing across multiple switches can be implemented. Therefore, each edge switch becomes part of a hierarchical switch topology, where unused resources from the adjacent core switch can be used to cope with the imposed workload.

Since core switches have only a small static flow rule set assigned, their is likely to be free flow table space left. Therefore, such a core switch can be part of the second switch layer of a hierarchical switch topology, as it is discussed in Section 4.1. The central controller, which is in charge of installing and modifying all flow rules throughout the network, can adaptively move flow rules from am edge switch and to a core switch. Figure 7.4 shows an example with the two switches $\texttt{Edge\_Switch}_1$ and $\texttt{Core\_Switch}_1$ along with their flow tables. $\texttt{Edge\_Switch}_1$ maintains one flow rules for each unique host connection. Depending on the number of hosts on both sides this can sum up to a high amount of required flow table space. $\texttt{Core\_Switch}_1$ only has its fixed flow rule set consisting the number of flow rules equal the number of output ports in use, in this concrete example this results in four flow rules.

In order to keep a consistent forwarding configuration, each edge switch requires a default flow rule, which matches all traffic, which can be implemented in OpenFlow using wild cards for each matching field. This default flow rule is assigned with the lowest priority, since it should only match traffic which is not matched by any more specific flow rule within the switches table. This flow rule will match all traffic, which could not be matched by another flow rule at the edge switch and subsequently forward it to the next layer core switch.

The central questions for moving a flow rule from an edge switch onto a core switch is, which flow rule should be moved and when should it be moved. Different strategies are possible in the scope of moving flow rules. It can be beneficial to keep a certain amount of free flow table space within the edge switch, therefore flow rules should be moved onto a core switch even before a critical usage level at the edge

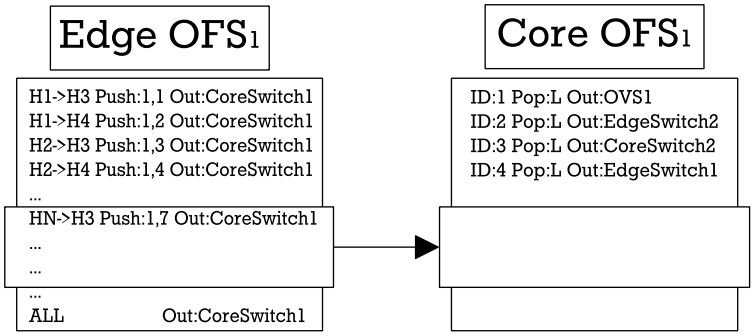

Figure 7.4: Move Flow Rule from $\texttt{Edge\_Switch}_1$ to $\texttt{Core\_Switch}_1$

switch is reached. This way, the edge switch keeps a certain amount of free flow table space in order be prepared for high demand peaks, in case a high amount of users join the network at the same time.

Different parameters can influence the decision making process, of when to move a certain flow rule to a core switch. The time delay imposed during operations such as flow installations, modifications and removals are different depending on the actual hardware or software switch. Switches with a fast forwarding performance, usually have a higher flow installation time delay, resulting in a smaller flow setup and modification rate [42]. This difference in time delay should be taken into account when evaluating the number of flow rule which can be moved to a core switch.

Due to this observation, flow rules which can be assumed to change less often should be identified for a possible movement to a core switch. This will be subject of the relevant flow awareness concept application to the system design, which is described in the next Section 7.4.3.

Not every flow rule can be moved to another switch, since it might have dependent parameters, which require them to be stored on a specific switch. Therefore, two categories of flow rules can be distinguished. One category is represented by flow rules, which can be moved to another switch referred to as moveable flow rule, such as the core switch and the second category are represented by non-movable flow rules, which are limited to one switch.

If a flow rules is movable depends on its match configuration and the action assigned to it. Furthermore, even if a flow rules can be moved to another switch, this might results in the need for additional hops for the traffic, which should be avoided. Flow rules which match a specific user, independent of switch specific parameters, such as the in-port, can be moved to another switch. This applies for flow rules initially stored at the edge switches, which passing the label information to a packet. Those label information can also be assigned by the core switch.

Flow rules, which connect two service instances running on the same VM host in comparison, should be kept on the corresponding service node switch, in order to avoid additional hops for those packets. Flow rules at the edge switches, which remove the label information and forward the packet to the end host, are also non-moveable, since the label information should be removed at the last switch.

7.4.3 Relevant Flow Awareness Concept

Chapter 4 introduced the relevant flow awareness concept as a concept, where the amount of flows which require central management is limited to relevant flows, which carry the majority of the data throughout all flows in the network. Instead, of focusing on the amount of data a flow carries, within this system design the duration a flow is essential. This is the case, since as longer a flow lasts within the network,

meaning that a specific user is sending and receiving data over the same path and ultimately through the same service chain, the longer the flow duration will be. This also results in a more consistent flow rule associated with the flow at an edge switch.

Since the flow setup rate of a core switch can be assumed to be less than those of an edge switch, the flows rules which are moved from an edge to a core switch should be selected according to their consistency. In other words, only flow rules which can be assumed to stay unchanged for a specific amount of time should be moved to a core switch. Therefore, Instead, of calling it relevant flow awareness, the concept is slightly changed to be termed relevant flow awareness concept in this system. A relevant flow in this case is characterized by its duration. Since the flow itself does not directly specify its duration, the system requires a point of detection, where a estimated duration for a certain flow can be calculated. This could be implemented in a similar fashion than the approach introduced in [17], where relevant flows are reported utilizing the information of the TCP transmission buffer. It could also be possible to directly modify certain applications, offering an interface where they can report required information regarding their estimated session duration.

7.5 Application Interaction Concepts

The design of the resource efficient dynamic network service chaining system was introduced from a point of view focusing on the resource optimization concepts applied within the system design. As described in Section 6.5, application interaction can be a vital part of resource optimization concepts. Application information can be used within the resource optimization concepts in order to support the decision making processes within certain optimization concepts.

Table 7.4 lists possible application information, which can be useful for the dynamic network service chaining system. Along with the specific application information the optimizing concept is listed, for which the listed information can be important.

Application Information	Description	Optimization Concept	Mandatory
User Identity	User Identity to select proper network service chain	Label Switching	YES
User Session Duration	Estimated length of a user session	Relevant Flow Awareness	NO
User Application Identity	Type of application a user is running	Relevant Flow Awareness	NO
User Application State	Current state of an application, such as insufficient bandwidth or normal operation	Label Switching	NO

Table 7.4: Overview of Application Information useful for the Dynamic Network Service Chaining

The dynamic network service chaining system design proposed in this work can provide network service chains per user and therefore maintains per-user path through the network. In order to setup these path and assign the appropriate network service chain to a user, the system needs to be aware of the specific user. Therefore, knowledge about the user identity is mandatory for the system. Upon connecting a users device to the system, for instance through the mobile network, the user identity is

investigated by the system and matched against a database, which stores user identities along with the network service chain related information. Such a database can be maintained by the ISP of a mobile cellular network, where costumers with different classes of subscriptions could be assigned to different network service chains. Knowledge about a the user identity and its network service chain to be assigned to its traffic is subsequently used by the label switching concept in order to setup the right paths through the network including the necessary service instances.

The user session duration is another important information, which is not yet included in a dynamic network service chaining system. Nevertheless, in this system, the estimated length of a specific user session is important for the decision making process, which decides which flow rule should be moved from an edge to a core switch. Moreover, the application identity, such as the type of the users application can support the estimation process of the session duration, such that some applications can be assumed to have a longer data transfer time than others. Whereas a website request will only take a very short amount of time, a video streaming session or a virtual private network session will take a reasonable longer amount of time. Both information sources can be combined for being used by the relevant flow awareness concept in order to select optimal flow rules to be moved from an edge to a core switch.

In addition, information regarding the current application state as described in Section 6.2 can be vital for the label switching concept. Especially in scenarios where a network service chain is unable to provide sufficient network resources, such as bandwidth, the application state can indicate an upcoming bottleneck in advanced. The controller can access this information and try to calculate an alternative path for the traffic through the network. But since the traffic has to traverse certain service instances, the controller could configure the network to let the traffic affected by an upcoming bottleneck bypass certain service instance, which are mainly responsibly for a certain bottleneck. This can be for instances resource intensive service instances such as DPIs or firewalls.

8 Prototype

This chapter gives an overview of the prototype implementation of the system design introduced in Chapter 7. The prototype is focused on providing the core functionality of the dynamic service chaining concept, which enables the network to identify traffic associated to a specific user to be forced to traverse a certain order of service instances within the network. In addition to this core functionality, the prototype implements two resource optimization concepts, in order to provide the dynamic service chaining functionality with a higher efficiency with respect to the network resources introduced in Chapter 3. The system design includes the three resource optimization concepts: label switching, hierarchical switch topology and relevant flow awareness. The prototype implementation focuses on the first two concepts, due to the absence of publicly available real-world data usable as an input for the relevant flow awareness. Nevertheless, the Chapter 9 discusses how the presented results also impact the application of the relevant flow awareness concept in future works.

8.1 Components

An example topology depicting the components of a dynamic service chaining topology is depicted in Figure 7.1. For the prototype only a subset of those components is selected, with respect to implementing and evaluating only the core functionality, while preserving necessary characteristics of certain components, in order to achieve valid results. The subset of selected components is depicted in Figure 8.1. This subset of components is sufficient for implementing the core functionality, such as the label switching concept with an edge switch which adds the label information and a core switch, which forwards traffic according to those label information. The service instances are implemented on two different service nodes, which can be in the following used to simulate a failure scenario (Section 9.1.3). The hierarchical switch topology concept requires at least two switches, in order to be properly implemented. In the depicted topology, flow rules can be moved from an edge or Open vSwitch to the core switch. The topology is further described in more detail in the following paragraph:

The topology consists of seven components: OpenFlow controller, edge switch, core switch, two hosts `H1` and `H2` and two VM hosts. `H1` and `H2` are both connected to the same edge switch, which is in this case acts as both ingress and egress switch. Therefore, `H1` is acting as a source host and `H2` is acting as a destination. Furthermore, the edge switch is represented by an Open vSwitch, running on a dedicated host. The core switch Instead, is represented by a hardware switch. The available hardware switch for this prototype is a NEC PF5240 [1]. The usage of an OpenFlow-enabled hardware switch has some additional implications for the prototype, which are covered by Section 8.1.2. The VM hosts both run one Open vSwitch each, which has one port connected to the physical ethernet link of the VM host. This link is wired to a port at the core switch. Each VM hosts runs several service instances, which have to ports each connected to the Open vSwitch of the service host. The reason for two dedicated VM host is based on the use case described in Section 9.1.3. In this use case one VM hosts experience a failure and is forced to shutdown its operation. The prototype in this use case is able to handover user sessions from the failed VM host to the second VM host.

[1] http://www.necam.com/SDN/DataSheets/doc.cfm?t=DataSheetPF5240 [Last accessed 02.05.2015]

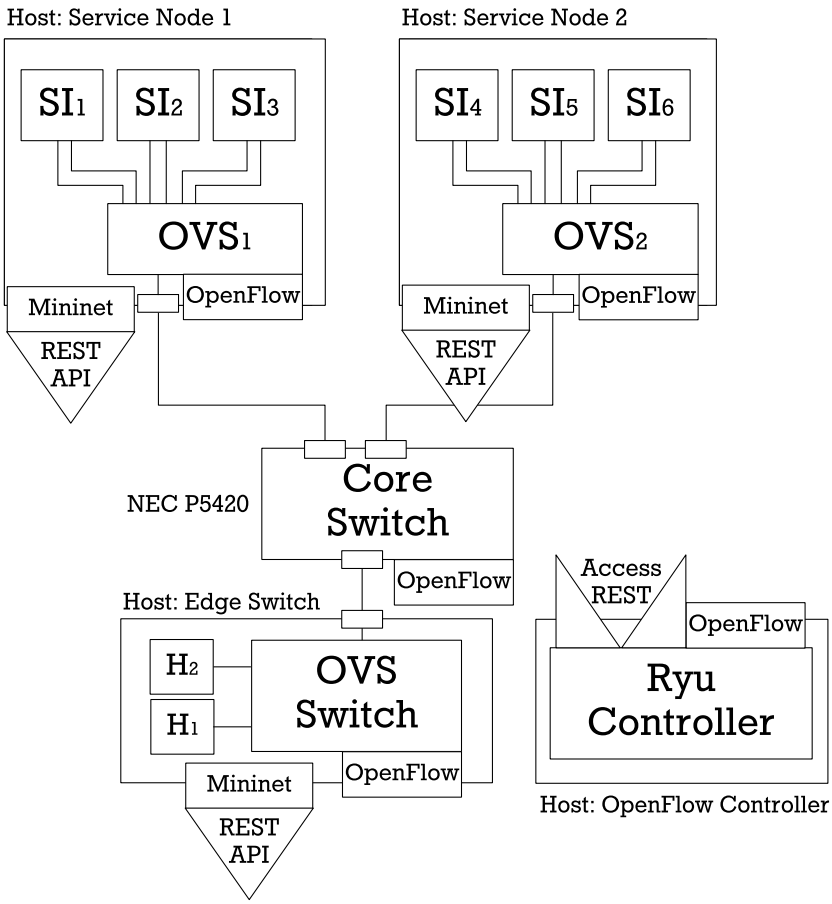

Figure 8.1: Topology of the Prototype Implementation

The following sections will further describe each of the components of the prototype implementation in more detail. This covers both the functionality of each component as well as the software tools and frameworks which are used to implement this functionality.

8.1.1 OpenFlow controller

The OpenFlow controller based on the Ryu SDN Framework [2] in version 3.19. The Ryu SDN Framework is written in Python, which makes it an ideal candidate for a rapid prototype development. In its core Ryu is an event driven application, which fits the event and message based communication pattern used by OpenFlow. Ryu currently supports the OpenFlow versions 1.0 up to 1.4. For the prototype OpenFlow version 1.3.1 is used, as it is the highest version of OpenFlow, which is supported by the NEC hardware switch.

The Ryu controller handles all OpenFlow related communication between the switches within the prototype. Each switch in within the prototype implementation is OpenFlow-enabled, therefore the complete forwarding scheme within the network is centrally controlled by the Ryu controller. The Ryu framework makes use of the Eventlet networking library [3]. The library adds concurrent networking functionality, enabling Ryu the ability to handle multiple incoming OpenFlow messages.

[2] http://osrg.github.io/ryu/ [Last accessed 29.04.2015]
[3] http://eventlet.net [Last accessed 29.04.2015]

A Ryu program can be executed without additional thread handling. Within the program a method can be set to listen for specific events, to be called whenever those events occur. Each OpenFlow message in the OpenFlow specification[34], such as `OFPT_FEATURES_REQUEST` is modeled in Ryu as an event. A Ryu program essentially consists of multiple methods describing the steps to be executed upon the arrival of certain OpenFlow messages. This event driven architecture is used within the prototype, in order to measure certain time periods. The controller, for instance, sends out certain `OFPT_FLOW_MOD` messages, followed by an `OFPT_BARRIER_REQUEST` message. The switch will then execute all flow modifications requests and upon completion answer with a `OFPT_BARRIER_REPLY` message, before executing additional OpenFlow messages, received after the `OFPT_BARRIER_REQUEST` message. Such a `OFPT_BARRIER_REPLY` message triggers each method within the Ryu program, which listens to those events. Listening to such OpenFlow messages is models in Ryu using Python's decorator functionality [4].

In addition to the OpenFlow processing, the Ryu controller is extended to access certain customary developed REST-APIs in order to control other hosts remotely. The exact functionalities offered by those REST-APIs varies depending on which host is requested and therefore is further described in the following sections covering the additional components of the prototypes. The REST-APIs are accessed by the controller via HTTP requests. Instead, of the Python's own implementation, Requests [5] is used to implement the HTTP requests within the Ryu controller. Requests provide an easier and more readable way to integrate HTTP requests within a Python program.

It is worth noting that the controller encapsulate all processing logic for the prototype including the logic to perform certain experiments for the evaluation. Since the controller requires comprehensive knowledge about the complete underlying network, it makes the controller an ideal candidate for unifying the processing logic in conjunction with the OpenFlow processing, which is directly connected to the processing logic.

8.1.2 Hardware Switch

Although the required functionality of the prototype could be implemented relying on software switches, such as Open vSwitches, a hardware switch has unique characteristics, which sets them apart of the characteristics experienced by a software switch. The hardware switch used for this prototype is the NEC P5420, which support OpenFlow in the versions 1.0 up to 1.3.1. The most noticeable difference is the lower flow installation performance, which limits the number of flows which can be stored in the forwarding table of a hardware switch. One aspect of the hierarchical switch topology addresses such differences between switches within one network and aims to balance the workload across switches offering different performance characteristics. Therefore, it is vital for the prototype to integrate with an hardware switch, in order to evaluate the impact of balancing workload across switches with different performance characteristics.

Beside the noticeable difference in terms of flow modification performance, which is investigated in more detail in the following evaluation, the NEC switch has some more unique characteristics which has to be paid attention to when using the OpenFlow functionality of this switch. The following section only discusses a subset aspects are only a subset of the unique characteristics, which are encountered during the work on this prototype.

[4] https://wiki.python.org/moin/PythonDecorators [Last accessed 25.04.2015]
[5] http://docs.python-requests.org/en/latest/ [Last accessed 02.05.2015]

Forwarding Table

The NEC switch does not have one single forwarding table, Instead, it offers five dedicated forwarding tables for entries to be stored in. Table 8.1 gives an overview of the five different forwarding tables available at the NEC switch. Each table can be addressed according to the OpenFlow specification by the listed table id. The different tables are sorted according to their search priority within the switch, from the highest priority table `mpls1` to the lowest priority `software` table.

Table ID	Table Name	Table Size	Search Order	Search Speed	Forwarding based on
#50	mpls1	512	1	High	Not applicable
#51	mpls2	16382	2		Hard- or Software
#0	normal1	5632	3		
#1	expanded	262144	4		
#20	normal2	4096	5		
#99	software	-	6	Low	Software

Table 8.1: Forwarding Tables Features of NEC P5420

The table size column gives an overview of the available flow table space of each table. Each of those values specifies the maximum amount of storable flow entries. The actual available flow table space depends on three parameters, flow detection mode, OpenFlow table resource mode and flow match field used within a complete match specification.

Flow Detection Mode

The flow detection mode specifies how the switch detects a series of frames, it can detect frames for instance in the Ethernet V2 or the mode specified in IEEE 802.3 SNAP format [80]. The prototype uses the ethernet detection mode. Beside the specified flow detection mode, the large difference between the usable flow table space also varies due to different memory technologies used. For instance, the expanded flow table is probably not implemented using TCAM, since the size of the TCAM required to store up to 262144 flow entries is simply not available in todays hardware switches. Instead, for this table a CAM might be used. A CAM is still faster than normal RAM, since it can be entirely searched in one operation. But CAM does not support the search for any value, which a TCAM in comparison does.

OpenFlow Table Resource Mode

The OpenFlow table resource mode indicates which flow match fields the OpenFlow controller is intended to use. Based on this table resource mode selection, only a subset of the complete list of flow match fields in OpenFlow 1.3.1 is available to the controller. For instance in table resource mode 9 the flow match field `ETH_DST` will be unavailable. It is assumed, that the specification of a table resource mode gives the switch the ability to configure its internal memory storage system to optimize memory usage to a specific subset of flow match fields. If the OpenFlow controller does not require a specific set of flow match fields, the switch might be able to optimize the available storage based on the remaining flow match fields, which are required by the OpenFlow controller.

Flow Match Field Implications

In addition, the usable flow table space can also vary depending on the OpenFlow flow match fields used to match certain flows in the network. Even though the table resource mode specified allows certain flow match fields, the usable flow table space can noticeable decrease if certain flow match fields of this subset are actually used. But these restrictions only apply to the expanded flow table of the NEC switch. For instance if the `ETH_TYPE` match field is specified to match IPv4 (`0x0800`) and IPv6 (`0x86DD`), the expanded flow table only support 24576 flow entries Instead, of initially 262144, which is only 9,3% of the initial flow table space.

Software-based Forwarding

The NEC switch can implement the forwarding of flow entries based on its fast path ASIC hardware or by utilizing the order of magnitude slower internal CPU. The fast path ASIC hardware within the switch does not support all available actions in OpenFlow 1.3.1. If such an action is specified by the flow entry, the switch is forced to execute the action using its internal CPU, which results in a noticeable slower forwarding speed. The switch lists those entries in its flow table with the key word `software-based`. For instance, if the `OFPOutput` action is set to output matching packets to the `OFPP_IN_PORT`, the NEC switch is forced to process those packets by its internal CPU in software. Naturally this action might not be widely used, since it is likely to help implementing loops within the network. However it is necessary for certain functionalities of the prototype, which are described in the Section 8.3

Beside the slower forwarding, which would only affect this particular flow, the software based forwarding of certain flow entries can lead to a higher utilization of the internal CPU, which is required to process incoming flow modification messages originated by the OpenFlow controller. Which may ultimately result in a decreased flow modification performance.

OpenFlow Interface

All though the NEC P5420 switch supports OpenFlow up to version 1.3.1, there are some actions which are not available to be specified by the controller in its flow modification messages, when Ryu is used as an OpenFlow controller. Most importantly, the actions for editing VLAN tags of packets, required to implement the label switching concept, as it is described in Chapter 7, are not available in conjunction with the Ryu controller. The development community of the Ryu SDN Framework are aware of those special cases and therefore started a Ryu certification website [6]. The Ryu certification consists of a variety of different OpenFlow actions, which are tested with each of the listed switches. Even though the NEC P5420 is not listed at this side, the comparable NEC PF5220 is listed along with its list of supported OpenFlow features given a Ryu controller.

Missing VLAN Tag Action

Both the `PUSH_VLAN` and `POP_VLAN` operations are tested with a negative result, meaning that the Ryu controller is not able to install certain flow rules including those actions to such a NEC switch. Since editing VLAN tags are a core functionality of the initial system design, an alternative had to be developed for the prototype, in order to implement a working label switching concept. Instead, of matching specific VLAN tags of packets, in order to identify their predefined path, the destination MAC address is used to

[6] http://osrg.github.io/ryu-certification/switch/NEC_PF5220 [Last accessed 03.05.2015]

decode this information. More detailed information about how this information is represented by the destination MAC address is presented in the Section 8.3.

8.1.3 Edge Switch Host

The edge switch host runs three components of the prototype implementation. First, an Open vSwitch instances acting as an edge switch and two connected hosts `H1` and `H2`. In the complete dynamic service chaining topology in Figure 7.1, there are dedicated switches acting as an ingress and egress switch, with the source host connected to one switch and the destination host connected to the second switch. However, for the prototype implementation these two switches are combined into one single switch, hosted on the edge switch host. Instead, the traffic from `H1` to `H2` and vice versa is forwarded through the complete topology as described in Section 8.3.

Since each edge switch is imposed with a high demand of available flow table space in the system design, this switch is implemented as an Open vSwitch within the prototype. For the prototype implementation and especially for the evaluation, flow table space is more crucial than forwarding performance, since the traffic for putting the prototype under a certain workload only consists of small packets, which do not have high bandwidth demands. The traffic characteristics and its generation is described in more detail in Section 8.2

The topology consisting of the two hosts and the Open vSwitch are emulated on the edge switch host using Mininet [7] [58]. Mininet can create a variable number of software switch instances, where each individual software switch can be controlled by Mininet, in terms of there interconnections or additional commands available on the software switch. For this prototype an Open vSwitch is used within mininet to emulate the edge switch topology. Open vSwitch in its version 2.3.1 also fully supports OpenFlow 1.3.1. In addition Mininet is capable of virtualizing a custom number of Linux hosts, by assigning each host its own virtualized network stack. Those virtualized network stacks are implemented based on network namespaces offered by the underlying linux operating system. Each host can have a custom number of virtual ethernet links, which has typically two endpoints. One end point is located in the local namespace and the other end point is located in the global namespace. In the case of the edge switch both `H1` and `H2` have a single virtual ethernet link, which interconnects them to the Open vSwitch.

Mininet is also written in Python, which eases the integration with the Ryu controller. Mininet emulates multiple topologies across the hosts included in the prototype. In order to control those different topologies, each Mininet instance on each host can be controlled using a custom build REST-API.

REST-API

As described in Section 8.1.1, the Ryu controller encapsulate all processing logic. Therefore, the controller requires to interact with different interfaces provided by the different hosts in order to control their current state. Therefore, a custom REST-API is developed for each individual host using the Bottle Python Web Framework [8]. The REST-API enables the controller to basically control the Mininet topology. Table 8.2 lists the available commands of the edge switch host. Each request to the REST-API is a blocking request, in other words, the called host is executing the requests method and returns an HTTP status code after execution finished. This can be indented, as for instance for the setup topology command or unintended for the start ping command. In the first case the HTTP status code 200 is only returned after the complete topology in Mininet is set up. The start ping command in comparison sets up a new

[7] http://mininet.org [Last accessed 02.04.2015]
[8] http://bottlepy.org/docs/dev/index.html [Last accesses 24.04.2015]

background process on the host, since otherwise the HTTP request would not receive any answer, until the ping process is stopped at the host. The column Blocking lists which API call is implemented as a blocking- and which one is a non-blocking call.

Command	URI	Blocking
Setup topology	*/start/<num_of_user>/<waiting_time>*	Yes
Stop topology	*/stop*	Yes
Start ping from `H1` to `H2`	*/start_ping*	No
Stop ping	*/stop_ping*	Yes
Start a new TCPDump capturing	*/start_tcpdump/<filename>*	No
Stops the TCPDump capturing	*/stop_tcpdump*	Yes

Table 8.2: Edge Switch Host API overview

The controller can setup the topology, which in this case sets up the Mininet topology with one Open vSwitch switch acting as a combined ingress and egress switch and the two Hosts `H1` and `H2`. The parameter <`num_of_user`> and <`waiting_time`> are used for the traffic generation described in Section 8.2. The topology can also be stopped remotely, which is useful in between two experiments, in order to start over with a clean state. The start ping command prompts `H1` to start pinging `H2`. The stop command intentionally stops the ping. The same applies for the TCPDump commands. The creation of a packet trace can be initiated remotely, by calling the appropriate URL with the <`file_name`> parameter, which specifies the later filename for the stored packet trace. The purpose of being able to capture packet traces is further described in Section 8.2.2

8.1.4 Service Node Host

Beside the edge switch host and the Ryu controller, the prototype topology consists of two additional dedicated hosts. The service node hosts each run a Mininet topology emulating a service node as described in Chapter 7. The service node hosts runs an Open vSwitch initiated and controlled by Mininet. This software switch interconnects all running service instances SI_n with the physical ethernet link of the service node host itself. Each service instance is implemented as a dedicated host in Mininet. Beside the processing related to the IP forwarding of each packet, the service instances do not further process the packet or simulate any real service instance behavior. Since one integral part of the evaluation is focused on time measurements, the round trip time of a packet should not depend on the processing power of the service node, with respect to the packet processing in each individual service instance.

A service instance has two separate virtual ethernet links connected to the Open vSwitch. One link acts as an in-port and the other one acts as an out-port for the traversing packets. The Mininet topology can emulate a custom amount of service instances, which can be controlled remotely through a REST-API. For the prototype implementation, two dedicated service node hosts are used in order to simulate the failure of one of them. Both hosts run the same the same mininet script, which makes the prototype scalable in terms of the possibility to add additional service nodes.

REST-API

The service node hosts REST-API offers the functionality listed in Table 8.3. The topology can be initiated remotely using the setup command. The parameter <`num_of_service_instances`> specifies

the number of service instances, which are initiated by Mininet and connected to the Open vSwitch. The <index> parameter is used to distinguish two service node hosts from each other, by assigning the index as a datapath id to the Open vSwitch.

Command	URI	Blocking
Setup topology	*/start/<index>/<num_of_service_instances>*	Yes
Stop topology	*/stop*	Yes
Simulate failure	*/fail*	No

Table 8.3: Service Node Host API overview

The stop topology command tears down the created topology in order to start over with a new topology setup. Section 9.1.3 described the use case of a failing service host. In order to implement this use case, the simulate failure command can be remotely used, which removes the physical ethernet link of the host from the Open vSwitch. This causes incoming traffic to be dropped at the ethernet link, rather than being forwarded though the service instances.

8.2 Traffic Simulation

Being able to generate and capture traffic is important for the prototype for two reasons: First, sending traffic from H1 to H2 is used to verify the correctness of the routing scheme of the prototype described in Section 8.3. For this purpose simple ping requests are sufficient, since they can be tracked to verify they traversed each switch and service instances in the network. The prototype does not implement a continuous verification by tracking the path of each packet, Instead, the ping was used during the development and testing of the prototype implementation. The verification of the path of each packet was done by comparing the OpenFlow per-flow counter maintained by each OpenFlow switch.

8.2.1 Traffic Generation

Beside the traffic usage for verification purposes, generating traffic is important in order to put the prototype under a certain workload. Ping requests again are selected for simulating workload for several reasons: First, their generation is simple. The ping requests are sent from H1 to H2. Even though, H1 only has one assigned IP-address, ping requests with the source IP address set to different IP addresses can be generated and sent by a single host. For this purpose hping [9] in its version 3 is used, which offers the ability to craft customized ping request packets and send them to H2. The crafting logic is specified using an htcl-file, which is generated upon setting up the Mininet topology at the edge switch host. For each user, which is specified by the the parameters <num_of_user> an IP address is generated starting from the initial address 10.0.0.1. The third octet of each users IP address is calculated as the following: 3. $octet = userid/255$. Whereas the fourth octet of the IP address is generated as: 4. $octet = userid \mod 255$. This results in the following IP address for different user as listed in Table 8.4

For each user a ping request is generated with its assigned IP address as the source IP address. For the source MAC address field, each ping request uses the source MAC address of H1, therefore the ping packets are identified throughout the network using their source IP address, rather than the MAC source address, which is identical for all ping request packets. In addition, each ping request can specify a

[9] http://www.hping.org/hping3.html [Last accessed 04.05.2015]

User ID	Third Octet	Fourth Octet	IP Address
1	$1/255 = 0$	$1 \mod 255 = 1$	10.0.0.1
2	$1/255 = 0$	$2 \mod 255 = 2$	10.0.0.2
300	$300/255 = 1$	$1 \mod 255 = 45$	10.0.1.45

Table 8.4: User ID to IP Address Mapping

sequence number, which is included by the destination host in its ping reply message. The sequence number provides a consistent way to map a ping request to it's reply, which is important for the evaluation.

Ping requests are generated indefinitely, until the creation is stopped remotely from the controller by calling the stop ping command. Nevertheless, a `<waiting_time>` parameter can be set, which defines the time in ms hping waits in between each loop of sending one ping request for each user. This parameter can be set to 0, however, this imposes a high load on both the sending host as well as the rest of the network, since the ping requests are send to fast. Finding an appropriate value for this parameter is subject to the evaluation process described in Chapter 9.

8.2.2 Traffic Capturing

The generated ping requests are used for both verifying the integrity and correctness of the routing scheme of the prototype, as well as for analyzing the flow modification performance of a switch. For this purpose the ping requests and their corresponding ping reply send by the destination host `H2` are captured at the edge switch host. TCPDump [10] is used as a capturing tool, which is a command line tool utilizing the Libpcap library for capturing packets on a linux host. Traffic is captured at the virtual ethernet link connecting `H1` with the Open vSwitch running on the edge switch host. The internal routing scheme is completely transparent to the end host therefore the host sends and receives packets without any knowledge of the destination MAC rewriting done within the network.

Capturing traffic can be initiated remotely by the controller using the REST-API provided by the edge switch host. During the development and testing of the prototype some issues occurred while capturing traffic with TCPDump. Although the 42 byte sized ping packets only accommodate for a small amount of bandwidth, their packet per second rate throughout the evaluation reached values up to 53k packets/s. Giving this number, TCPDump should difficulties to keep up capturing all ping requests and replies. In such a case specifying a greater buffer size available to TCPDump solves these difficulties.

8.2.3 Traffic Evaluation

Packet traces, which are captured during the execution of the prototype, are stored for later processing. They are not processed during the execution, since parsing the resulting packet traces would result in additional CPU usage, which would impact the other processes running on the edge switch host such as the Open vSwitch. Instead, of live processing the stored packed traces, the traces are processed later on by a custom developed Python script. The Python library DPKT [11] provides extensive functionalities for parsing packet traces and extracting per packet information, such as timing informations and

[10] http://www.tcpdump.org [Last accessed 20.03.2015]
[11] https://pypi.python.org/pypi/dpkt [Last accessed 26.04.2015]

header information. The parsing script is used in the evaluation to determine a variety of performance characteristics, such as the duration a switch requires to install a certain set of flow rules.

8.3 Implementation

The prototype includes the implementation of the label switching and the hierarchical switch topology concept. This section provides a detailed description of the implementation of both concepts within the prototype topology depicted in Figure 8.1.

8.3.1 Label Switching Concept

Section 8.1.2 highlighted the difficulties of the attempt to implement the label switching concept based on VLAN tags. The initial concept of using VLAN tags for encoding the path information could not be implemented using the Ryu controller and the NEC switch. The NEC hardware switch does support the required VLAN functionality, but it is not accessible via the Ryu controller, which is confirmed by the developers of Ryu (Section 8.1.2) As a alternative to VLAN tags, the prototype uses a different header field within the MAC header, where the necessary path information is encoded. Figure 8.2 illustrates the encoding scheme used by the prototype. The first byte of the mac remains unused. The user id is similar to the scheme used for generating corresponding source IP addresses. Even though the user is matched by its IP address throughout the network, the user id is stored within two bytes of the destination MAC address. Although the matching of the user id within the MAC address is not currently used, it could be vital to additionally identify the user with these two bytes, for instance if a service instances rewrites the IP header information. The most important value for the label switching concept implementation is stored in the second last byte of the MAC address. This byte stores the ID, which encodes a specific action executed by the core switch. The last byte stores the last hop information, which is used at the egress switch to map the packet to the corresponding destination host.

In order to be transparent, the manipulated destination MAC address is only used in the internal network and stays transparent to every end host. This transparency is achieved by rewriting the destination MAC address back to the MAC address of the destination host, before the edge switch forwards the packet to the destination host.

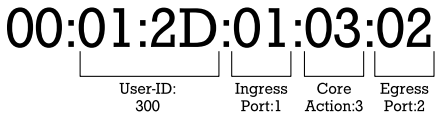

Figure 8.2: Destination MAC Path Information Encoding

For implementing and evaluating the prototypes functionality, the destination MAC encoding is sufficient, since it has the same flow rule saving potential on a core switch. However, the destination MAC address only has a limited amount of bits for storing path information, therefore only short paths can be encoded. For the prototype, where the longest path is just three hops, this is not an issue. Nevertheless, in a larger network the VLAN tags provide a larger encoding space supporting longer paths.

8.3.2 Hierarchical Switch Topology

The hierarchical switch topology is implemented in a proactive scheme. Since the controller is able to calculate all necessary flow rules for a correct forwarding scheme in advance, the rules are not moved from one switch to another during the execution. Instead, the rules are either stationary or moved. Stationary flow rules are installed on the initial switch, whereas moved rules directly are installed at the core switch.

Service Instance Routing

The service instances are emulated as routers, which forward traffic between their in-port and out-port. The IP header information of each packet remains unchanged, whereas the source MAC address is rewritten by the service instance to the MAC address of the corresponding out-port. This information is important, since forwarding rules which match traffic based on the in-port at the service node, in order to identify the originated service instance, could not be moved to the core switch. Because the core switch would be unable to identify the service instance, the packet came from, Instead, it would only know about the service node, which forwarded the traffic to the core switch. Therefore, the source MAC address of each packet forwarded by a service instance can be used as a match field even at the core switch to consistently identify packets in the network.

9 Evaluation

The system presented in Chapter 7 is designed with four objectives in mind (Section 7.1.2):

1. Sufficient flow table space: Providing sufficient flow table space for storing fine-grained flow rules.
2. Avoid redundant flow rules: Avoid redundant storage of fine-grained rule within the network.
3. Fine-grained flow rules: Optimize the system for efficiently maintaining a large number of fine-grained flow rules.
4. Flexible forwarding behavior: Efficient configuration and modification of the forwarding behavior throughout the network.

Those objectives address the optimized usage of the flow table space within the network, as well as the control bandwidth provided by each switch. Those network resources are essential for the dynamic service chaining system. The focus of this evaluation is set to investigate on the impact of the two applied resource optimization concepts, the label switching concept and the hierarchical switch topology concept, which are both implemented in the prototype.

In order to evaluate their impact on the efficient resource usage, a failure scenario is described in Section 9.1.3. For this scenario the failure time, defined as the time required for the installation and modification of all flow rules necessary to implement all paths for each user is evaluated. Since the failure time is mainly dependent on the time each switch requires to perform certain flow operations, such as installing or modifying flow rules, the efficient usage of available control bandwidth can results in a time reduction. The flow table space in addition can be measured by simply counting the number of installed flow rules per switch, which is comparable throughout different approaches. The control bandwidth is also dependent on the flow table space used per switch, such as more flow installations also benefit of a larger control bandwidth. Therefore, the failure time can be utilized to evaluate the effect of applying both resource optimization concepts on the dynamic service chaining system.

The evaluation is focused on the resource efficiency and therefore does not include the explicit evaluation of a possible application interaction concept. Application interaction is assumed to be a necessary part of the dynamic service chaining system, for instance in order to identify the user. The relevant flow awareness concept is not yet implemented in the prototype and therefore not part of this evaluation. However, it can be added in a future enhanced prototype, where such application information is taken into account.

9.1 Scenario

Evaluating the characteristics of a dynamic network service chaining system, requires certain assumption about the configuration of the system. Even though not much details about the usage of such network service chaining systems in todays mobile networks is publicly available, the configuration settings and performance aspects can be approximated according to certain assumptions. One parameter which has to be approximated is the number of actual user, which can by served by a service chaining system implemented by the prototype. In addition, the parameter specifying the number of service instances, which portably can be hosted by a VM host is important for evaluation the system assuming operating under a real workload.

9.1.1 Number of User

For approximating the number of user a variety of different components have influence on this parameter. Starting at the edge switch, which connects all user to the service chaining system, following the path along the service nodes, which run service instances per user. In order to outline a practical usage scenario, the traffic per user is important, in order to approximate how many user an edge switch can securely handle within a service chaining system.

The Global Mobile Traffic Forecast [16] from 2015 published by Cisco predicts an average mobile connection speed for smartphones reaching 8.4 Mbps in the year 2016. Although this average speed is not used by each customer in every minute of operation, it can be used to define an lower bound for the amount of user a service chaining system is capable of, if each user would ask for its average connection speed of 8.4 Mbps. Assuming an edge switch with 10 Gbps per port, with an amount of 80% usable I/O performance leads to Equation 9.1

$$\frac{8\ Gbps}{8.4\ Mbps} = 975\ User. \tag{9.1}$$

Assuming each user using its full mobile connection speed at any time is not a practical approximation, more realistic results can be derived from the monthly data usage per user compared to the usage time. In [15] the average amount of data usage for smartphones in the year 2014 is given by 2000 MB per month, which is predicted to increase to up to 5458 MB by the year 2019. The authors in [60] report a daily smartphone usage of 207 minutes per day, which results in an average data rate per user calculated in Equation 9.2.

$$\frac{5458\ Mbyte}{207\ minutes \times 60\frac{seconds}{minute} \times 30 days} = 14 kbps. \tag{9.2}$$

14 kbps would be the minimum data rate, in order to achieve the monthly amount of data per user. Given this data rate, the number of user within the service chaining system can be calculated as given by Equation 9.3.

$$\frac{8\ Gbps}{14\ kbps} = 600000\ User. \tag{9.3}$$

This amount of user represents an upper bound for the maximum number of user, which would be feasible for a 10Gbps interface on an edge switch. However, the flow table space required for 600 000 user, is likely to overload even an Open vSwitch with higher flow table space characteristics. The authors in [28] mention a flow table space of just 14 000 flows, which can be kept in the L3 cache of the processor executing the Open vSwitch. 14 000 is not the maximum amount of storable flow rules for an Open vSwitch, Instead, flow rules stored within the L3 cache of the processor result in an increased forwarding speed. Therefore, larger flow tables can still be suitable, if maximum forwarding speed with minimal delay is not absolutely necessary.

The authors in [42] report 65 000 flow rules for being stored by an Open vSwitch. On an edge switch each user requires two flow rules, resulting in 20 000 flow rules per edge switch. The ingress and egress switch in the prototype are being implemented using the same Open vSwitch instance, which results in 40 000 potential flow rules. A range of 1000 user up to 10 000 user are assumed to be practical for being evaluated with the prototype implementation.

9.1.2 Number of Service Instances

The number of service instances depends on two central aspects: First, the number of service instances, which are installed in a row at a user's path, in other words, the number of different service instances assigned to a user's traffic. The second aspect deals with the amount of service instances each VM host is able to operate.

The first aspect, can be defined as the service chain length. In today's mobile networks a variety of different mostly static configured and fixed located service instances exists. The list includes, for instance web proxy services, video optimizer, firewalls, and deep packet inspection services. On the one hand the users traffic should not be forced to traverse to many service instances in a row, since, the traffic is delayed by each service instance, depending on the amount of processing required at each instance. On the other hand, the dynamic service chaining concept should be able to support a certain service chain length, in order to prove extensibility for future development, which might add the need for additional service instances.

The service chain length additionally depends on the second aspect mentioned: The performance of the VM host. Memory speed can be one limiting factor for the VM host, since virtual switching using an Open vSwitch requires a lot if memory interaction for each virtual interface forwarding packets within a singe physical host. In line with the observations in [27], a memory to CPU bandwidth of 200 Gbps is assumed for a server hosting multiple service instances. Half of those 200 Gbits/s can be assumed to be available for the Open vSwitch forwarding. Each service instance requires additional memory bandwidth according to Equation 9.4. The equation calculates the number of times a packet traversing SI service instances is processed by an interface (virtual or physical). Each packet is at least processed two times by the physical interface a the service node, and in addition is processed by two virtual interfaces per service instance SI

$$MemoryBandwidth_{Host}(SI) = ((SI - 1) * 2 + 4) * IS_{SI} \tag{9.4}$$

where:

SI = Number of Service Instances

IS_{SI} = Interface Speed of a Service Instance

Following this equation with an interface speed for each service instance of 10 Gbps, the 100 Gbps available memory speed result in four service instances per user on each service node.

9.1.3 Failure of a Service Node

The main objective of this evaluation is set to a failure scenario, where a service node experience a failure, which affects all service instances hosted by this service node. At the time such a VM host failure occurs, the system requires to reconfigure its forwarding configuration, such that traffic which previously traversed the failed service node is forwarded to a second service node instead. In addition, the service instances hosted on the failed service node require to be restored at the second service node within the system. Along with the service instances, the necessary flow rules connecting the service instances as a service chain are also required to be installed on the second service node.

The time it takes to perform all those described operations is referred to as failure time. The system design includes two resource optimization concepts, which are applied to the prototype implementation. Evaluating the impact of those resource optimization concepts on the failure time achieved by the prototype is the major objective.

9.2 Time Measurement Methodology

The failover time can be measured using different methods, which might result in different results depending on the accuracy and precision of each measurement method. This section provides insights on the variety of measurement method used throughout the evaluation.

9.2.1 Metric

The main metric used throughout the evaluation is time and in particular time durations. The smallest time durations measured within the evaluation are longer than 100ms, which is in line with the possible resolution of the timestamps, the linux operating system can deliver. The majority of timestamps are gathered using the `time.time()` method of python, which returns an float value with a resolution of 100 nanoseconds. In addition to the timestamps gathered using python, Wireshark is used to determine the arrival time of certain packets. The frame arrival time stored by Wireshark has a resolution of 1000 nanoseconds, which is also sufficient for determine time durations greater than 100ms. Beside resolution, accuracy and precision are important for a resilient time measurement, which can be assumed to be sufficient as well in the test environment using linux hosts.

Time measurements in distributed systems can be challenging, due to the different system times on each device. Especially for measuring durations with a resolution of less than 100ms would require a careful time synchronization between the individual devices. In an OpenFlow-enabled network, the OpenFlow controller can act as a single instance for accurate time measurements. The following sections investigate on the different methods for measuring the installation and modification time a switch requires for a given set of flow rules.

9.2.2 Barrier Command Time

The barrier command is specified in the OpenFlow protocol [34], as a command to notify the controller as soon as all previous OpenFlow operations are executed on the switch. In order to receive such a notification from a switch, the OpenFlow controller has to send a barrier request message in advance. Upon the arrival of this barrier request, the switch has to finish all previously received OpenFlow messages and subsequently answer the barrier reply with an barrier request, before it starts processing OpenFlow messages, which arrived after the barrier request message.

The barrier command can be used to determine the end time for a failover scenario in the following way. An example sequence is depicted in Figure 9.1. The OpenFlow controller starts sending flow modification messages directly after setting its `Start_Time`.

The OpenFlow controller continuous to send all flow modification messages, with the last message being the barrier request message. The switch will install all flow rules included in the flow modification messages and will answer with a barrier reply message, notifying the OpenFlow controller about the completion of all requested operations. The `End_Time` for the flow modification process can be set equal to the arrival time of the barrier reply send by the switch, since all requested flow modifications can be

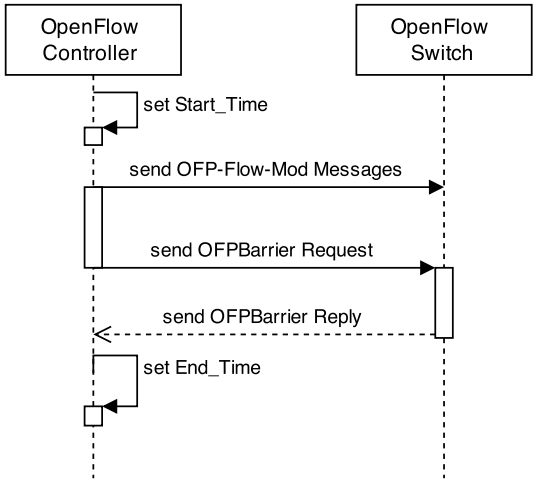

Figure 9.1: Barrier Command

assumed to be executed by the switch at this point in time. The accuracy of this measurement can be negatively affected by the connection conditions of the secure channel between OpenFlow controller and the switch for two reasons: First, the barrier request might take longer to reach the switch, leading to a gap between the last flow modification message and the barrier request. And second, the barrier reply might get delayed on its way through the network. However, upon receiving the barrier reply message, the controller can be certain about the completion state of its requested modifications.

9.2.3 Ping Request Time

The barrier command could be used for determine the overall failover duration, if it delivers accurate results. A novel evaluation scheme in order to investigate the accuracy of the barrier command is developed, since no suitable information could be found in the research community.

Since no information, regarding the accuracy of the barrier command, could be found in the research community, a novel evaluation scheme is developed. Therefore, a second measurable information is required, in order to compare the results of both measurements. The completion time of certain flow modification operations at a switch can also be measured using active traffic send to the switch. A certain flow modification can be assumed to be completed at the switch, if traffic matching this flow rule is forwarded by the switch according to the assigned action. The assumption of this underlying experiment is that the if the barrier command can provide accurate timing information, if both measured times show an comparable trend.

Figure 9.2 depicts both measurements performed with the NEC switch used in the prototype implementation. The results are extracted from the experiment described in Section 9.3.1. The x-axis represents an increasing number of flow rules, which are installed per experiment in a row by the OpenFlow controller. The y-axis represents the total time required for the NEC switch to install the number of flow rules on the x-axis. Each trend is illustrated using three lines, with two dashed lines and one solid line. The lower dashed line illustrates the 25%-quantile, where 75% of the values are above this line and 25% are below. The solid line represents the median value, which is calculated by the 50% quantile. The upper dashed line represents the 75%-quantile, where 75% of the values per experiment are below this line. The same scheme is used throughout the evaluation for each figure using lines. It is also worth noting that the resolution of the y-axis throughout each figure in this evaluation is adapted to the range of y-values for

a better visualization. Therefore, trends can be misleading due to the difference in the y-axis resolution. The green line, which is almost completely hidden by the red line, shows the measured time using the `End_Time` determined by the received barrier reply. The red plot illustrates the corresponding `End_Time` determined by the traffic send to the switch.

The traffic measurement is performed by processing packet traces, which are captured using the traffic capture functionality implemented in the prototype (Section 8.2.2). Therefore, two hosts `H1`, `H2` are connected to the NEC switch, with `H1` sending ping requests (Section 8.2.1) and `H2` sending ping replies for received ping requests. The number of distinct source IP addresses generated by `H1` has to be equal to the number of distinct flow rules installed by the controller, in order to have one matching flow rule per IP address. The `End_Time` for the traffic measurement is calculated as follows:

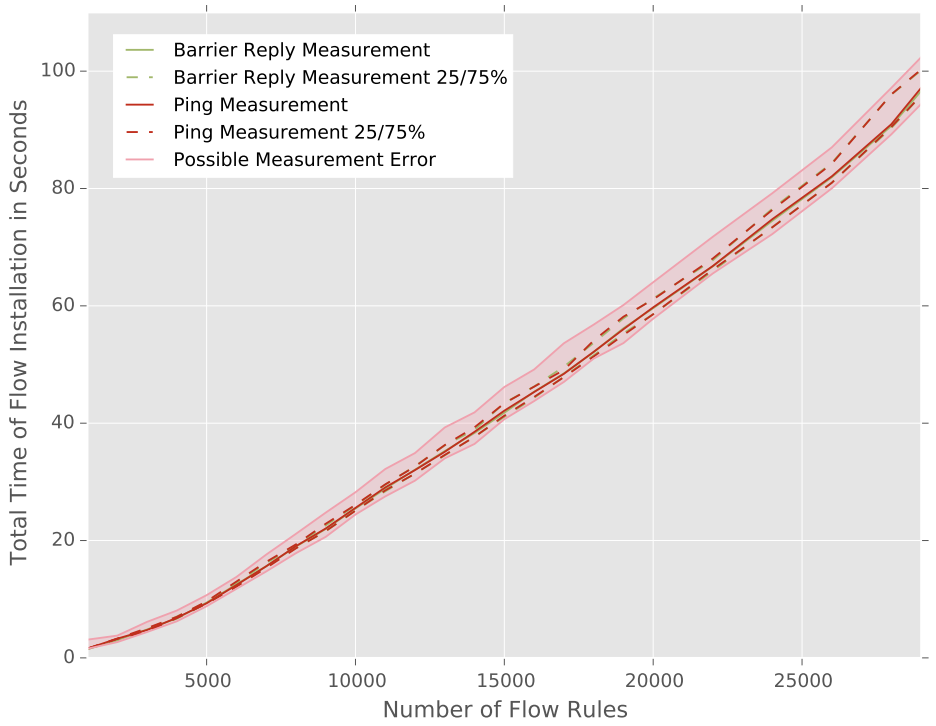

Figure 9.2: Flow Rule Installation Time NEC P5420

Since the controller start the execution of ping continuous ping requests remotely, it has no certain knowledge about when the ping actually started. Therefore, the controller requests `H1`, before its starts sending the first flow modification message. After a short waiting interval, the controller starts sending flow modification messages to the NEC switch. This point in time is assumed within the packet trace, when the first ping request matching the first installed flow rule receives an ping reply by `H2`. During the installation of flow rules requested by the controller, more and more ping requests are forwarded by the switch, since the matching flow rule was installed. The `End_Time` is assumed to be the earliest point in time, where all send ping requests are answered by an ping reply. At this point in time the switch has installed all flow rules completely and is able to forward all ping requests without any flow table misses. Figure 9.2 illustrates this time as the red line, which closely follows the underlying green line given by the installation time measured using the barrier command.

Since the ping requests are not sent continuously, but rather have a gap of 500ms in between each full interval of ping requests, the measured time could be shorter than the actual flow install process takes. This is the case, if the waiting interval of 500ms is at the same point in time, when the switch starts install flow rules. The first successful ping requests can only be detected after those 500ms, therefore the

measured time can be at most 500ms to short. The transparent red area illustrates this possible error, with its higher bound, whereas the lower bound of the area illustrates the minimal measured time.

However, the close match between both measured times across the whole spectrum of different flow rules proves the accuracy of the barrier command for this experiment.

Figure 9.3 supports these results by illustrating the cumulative probability for four different number of total flow rules. Instead, of the flow installation time, the lines show the flow installation performance per second, calculated by dividing the number of flow rules installed in each round by the required time. The cumulative probability visualizes the distribution of the measured flow performance values within multiple experiments with the same input parameter.

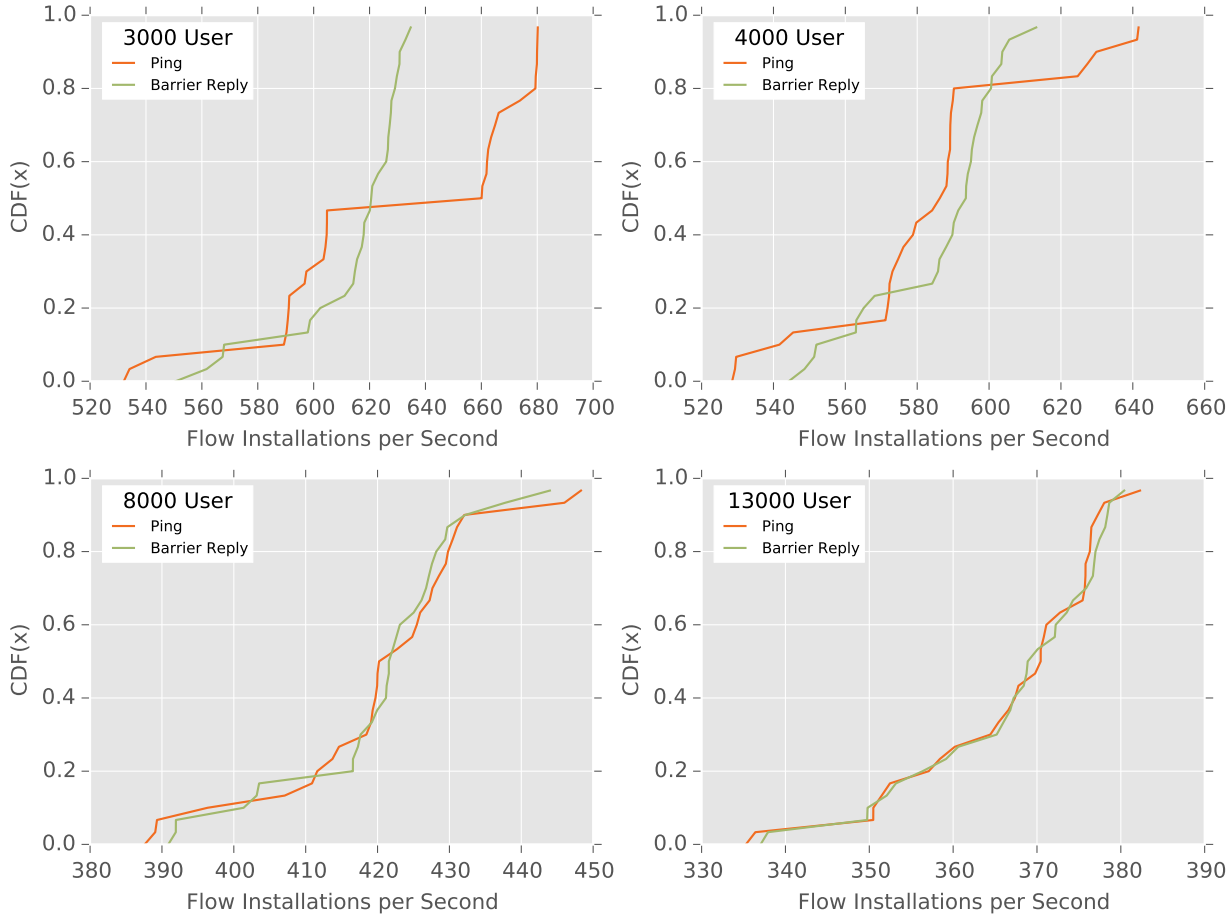

Figure 9.3: Distribution of Flow Rule Installation Performance NEC P5420

The green line is based on the time measurement conducted using the barrier command and the orange line is based on the time measurements calculated from the packet traces. With an increasing number of flow rules, both lines converge until they almost map each others trend for 13000 flow rules. This effect can be explained by taking the possible measurement error into account, which is constant based on the waiting time interval in between a full set of ping requests. Since this value is set to constantly 500ms, its influence decreases, since the overall installation process takes several seconds for higher number of flow rules. Therefore, the accuracy of the time measured using the captured traffic increases for higher number of flow rules.

In summary the barrier command can provide accurate timing informations, which can be used to validate the completion of flow modification messages for individual switches. Combining such information can be used to calculate the complete failover time by relying on the barrier reply messages from

the individual switches. The failover time would be determined by the last barrier reply received by the controller.

9.2.4 Barrier Receive Time

Measuring the time based on the barrier reply throughout the evaluation revealed some conspicuities. One of those conspicuous effects is illustrated in Figure 9.4. Whereas the previous experiment only covered the interaction between the Ryu controller and a single switch, the controller interacts with multiple switches at the same time in the prototype topology. From the complete topology consisting of four switches in total, Figure 9.4 depicts the core switch represented by the NEC switch and the ingress switch located at the edge switch host. The observed effect is most noticeable for the core switch. The orange line visualizes the time difference between the barrier reply received and processed by the Ryu controller and the time of arrival at the host. The arrival of the host is tracked using Wireshark capturing incoming OpenFlow packets at the controller interface. The plot clearly outlines an increasing time delay of the barrier reply message to be processed by Ryu, with a maximum difference in time of almost four seconds for 10000 user. The time difference is not only noticeable for the hardware switch, but also for the Open vSwitch implementing the ingress switch. The time difference for the ingress switch is smaller in size and has a smaller gradient, but shows a much greater spread with an increasing number of users and ultimately a higher number of installed flow rules.

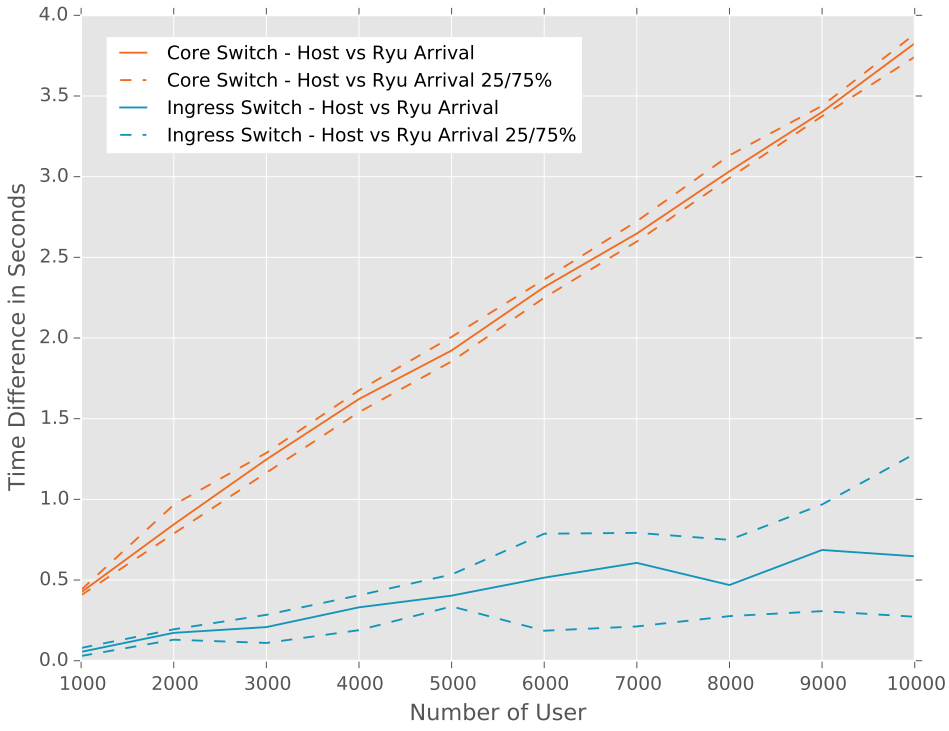

Figure 9.4: Time Difference between Barrier Reply Arrival Times

Such a time difference between the processing of the barrier reply event by the Ryu controller and its earlier arrival at the host itself could be explained by performance bottlenecks within the Ryu SDN Framework. It seems that Ryu has trouble to process the barrier reply shortly after the host received the packet, when it concurrently sends other flow modification messages to other switches. This assumption is based on the greater time difference observed at the core switch, which in this case only installs five

flow rules regardless of the number of user. Therefore, it is always the first switch to finish the installation process. Upon receiving the barrier reply message, the Ryu controller still sends further flow modification message to the ingress, and both service node switches.

This also gives hints why this effect is not an issue in the first experiment, since Ryu only interacted with one single switch and therefore did not have any performance issues while processing the barrier reply message. A study of the performance of different OpenFlow controller revealed similar performance limitations in the Ryu SDN Framework [89].

9.3 Flow Operation Performance

The flow operation performance is defined as the performance achieved by a switch for installing or modifying flow rules in their flow table. The flow operation performance is measured in flow operations per seconds and is mainly influenced by the control bandwidth network resource defined in Section 3.2.1. The flow operation performance is conducted for both the NEC hardware switch and an Open vSwitch, which are both used in the service chaining prototype.

Measuring the achievable flow operation performance for both switches gives a basic idea of what can be expected by each switch in the later service chaining topology, since the failure time is based on who rapidly the failover path information can be installed within each switch in the network.

The flow installation performance is measured for a switch with an empty flow table, therefore each flow rule requires to be installed by the switch. The flow modification performance in addition is measured by sending flow modification messages inducing the identical flow match, but having different action parts defined for the flow rule. Therefore, the switch has to find the matching flow rule for which it will modify the action part according to the action defined in the flow modification message.

Parameter

The following parameters are changed throughout the experiment, which can have an impact on the switch performance. First, the number of rules to be installed or modified by the switch. The switch is tested with a number of flow rules ranging from 1000 to 30000 flow rules. Second, the switch is evaluated once without traffic arriving at the switch to be processed and once without traffic, in order to evaluate the effect of additional forwarding processing on the switch performance. Third, the priorities defined in a flow rules are once set to a fixed identical value and once set randomly for a number between 1 and 20.

Metric

The flow operation performance is divided into two distinct performance metrics: First, the number of flow rules, which can be installed by a switch per seconds is referred to as flow installation performance. The second metric is the flow modification performance, which indicates how much flow rules a switch is able to modify per second.

Topology

The topology for performing the switch performance experience is simplified, in order to avoid additional components, which could effect the switch performance. The topology therefore only consists of two

hosts connected to the switch, which is subject to the performance test. One hosts can be triggered by the Ryu controller, in order to generate ping requests, addressing the second host.

Before investigating the failover time achievable based on the application of both the label switching concept and the hierarchal switch topology, as a first step, similar to the experiment conducted in the Section 9.2, the flow operation performance in the context of OpenFlow is evaluated in the following sections. The OpenFlow performance is evaluated for two scenarios, one in which a specific number of flow rules are installed in a switch, which has no entry in its flow table yet and second a flow modification performance test, in which each stored flow rule is modified by adding a different action to it. Each of these two test scenarios are conducted using both the NEC hardware switch and an Open vSwitch. Each switch is tested in range of 1000 to 30 000 flow rules, both for installing newly added flows and modifying exciting flow rules.

9.3.1 NEC P5420

In the following the measured flow operation performance for the NEC P5420 switch results are presented.

Flow Installation Performance

Figure 9.5 illustrated the results of the flow installation performance experiment for the NEC switch. Based on the figure three characteristics of the switch performance of the NEC can be derived. First, at best the switch is able to perform around 820 flow installations per second, illustrated by the orange line. This performance is stable across the whole range of number of flow rules represented at the x-axis. Second, the flow installation performance with traffic traversing the switch is always less than without traffic, which can be seen by comparing the purple (with traffic) line and the orange (without traffic) line and furthermore the green (with traffic) and blue (without traffic) line. The processing power required for the forwarding of packets, even if it is hardware based forwarding, impacts the overall flow installation performance.

Third, the performance of the flow installation is influenced by the priorities assigned to the flow rule. Both the green and blue line show a decreasing flow installation performance, starting at initially 800 and 650 flow installations per second, both lines follow a decreasing trend towards 320 flow installations per second for 30000 flow rules. Whereas the performance with identical priorities remains stable, the decreasing trend is obvious for different priorities.

In addition, the red line illustrates the performance measured based on the processed packet traces. The transparent area marks the possible error, nevertheless the actual results show a strong overlay with the performance measured using the barrier command, which is supported by the cumulative probability visualized in 9.3.

Flow Modification Performance

The same set of parameters are used for the second experiment, which investigates the flow modification performance. Figure 9.6 illustrates the results of this experiment, which show similar characteristics compared to the flow installation performance.

Most noticeable, the overall flow modification performance is higher, with the lowest value being 860 flow modification per seconds up to 1100 flow modifications per second. Having the switch modify its

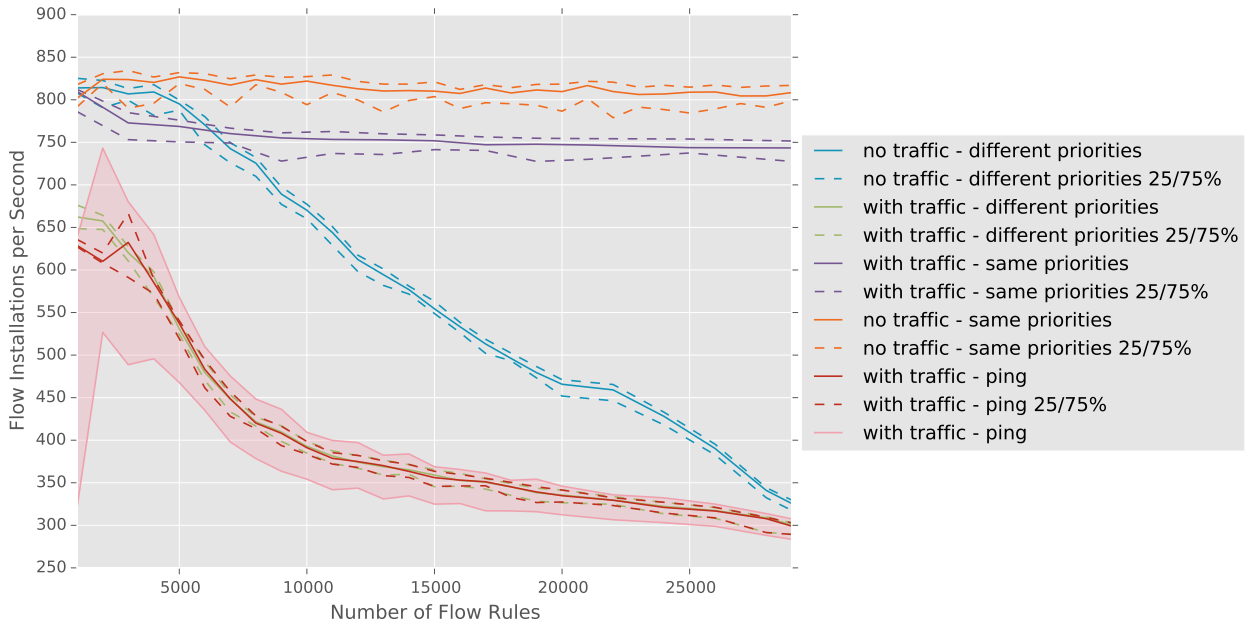

Figure 9.5: Flow Rule Installation Performance of NEC P5420

flow table space, while processing incoming network traffic results in a lower flow modification performance. Where the NEC switch showed a decreasing installation performance, when imposed with different flow rule priorities, it only shows a slight performance decrease (green line) for both traffic and different priorities. Interestingly, if no traffic is present, the switch shows equal flow modification performance, independent from the priority parameter. In comparison, the switch shows different flow modification performance, when traffic is present, where different priorities lead to a smaller flow modification performance.

9.3.2 Open vSwitch

In addition to the hardware switch, an Open vSwitch instances is evaluated regarding its flow operation performance, which is later compared to the results presented for the NEC P5420 hardware switch.

Architecture

Before presenting the results of the performance evaluation, it is worth exploring the fundamental architecture design of the Open vSwitch software. Figure 9.7 depicts the twofold architecture design based on the description given in [78]. The Open vSwitch (OVS) in its core consists of two integral components, which implement the packet forwarding functionality. The ovs-vswitchd is a userspace daemon, which is responsible for the communication between the Open vSwitch and the OpenFlow controller. This especially includes processing and storing the flow rules given by the controller. The other major component is represented by the datapath module, called ovs kernel module, which is executed directly inside the kernel in order to make use of the faster performance of kernel modules compared to software executed in the user space. For this reason, the ovs kernel module is kept as simple as possible and often also specifically written for a certain operating platform [78].

The two major components work together at the packet processing as follows: Upon the arrival of a new packets, the ovs-kernel module receives the packet from the ethernet interface and checks if it has a

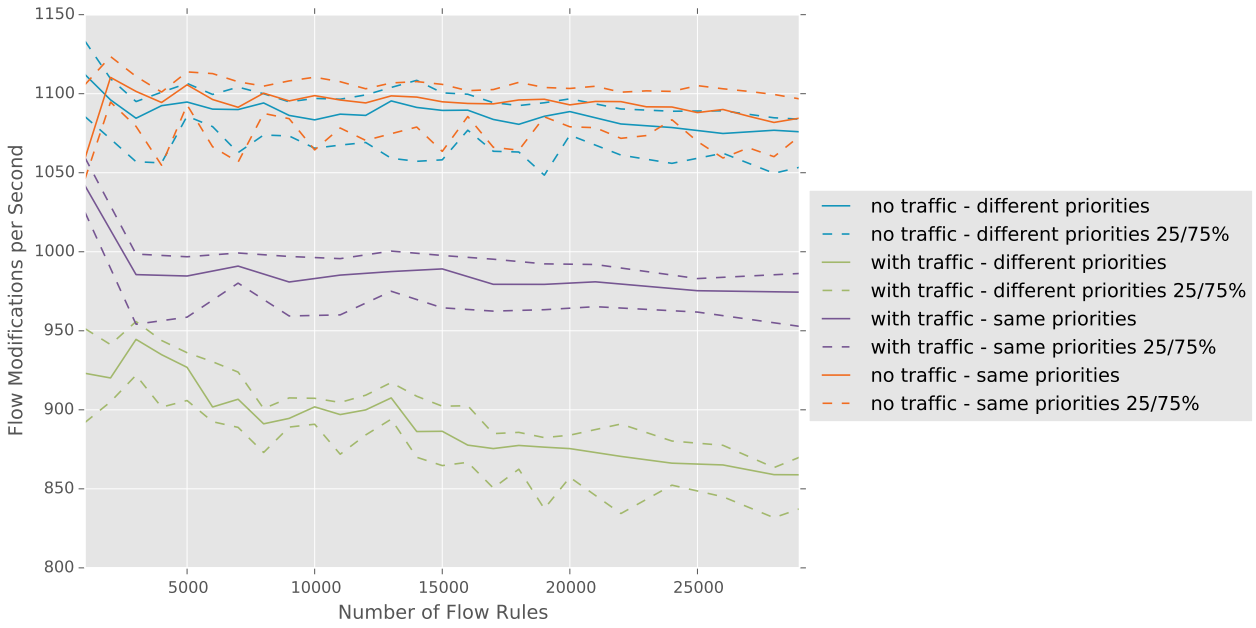

Figure 9.6: Flow Rule Modification Performance of NEC P5420

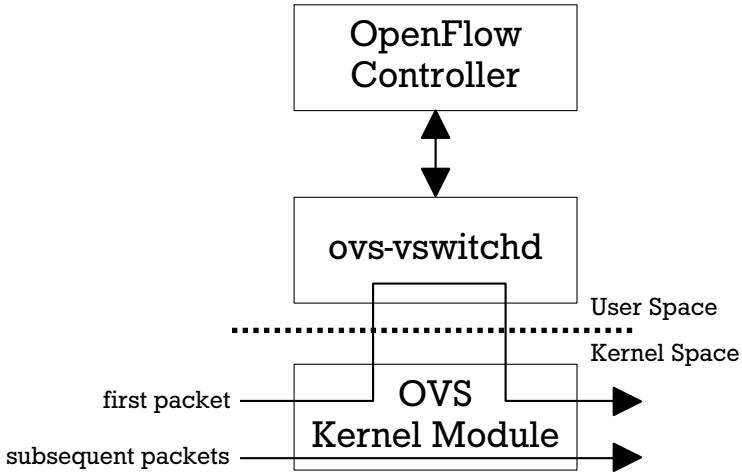

Figure 9.7: Open vSwitch architecture based on [78]

stored flow rule, which can be associated with the packet. The ovs kernel module does not directly store OpenFlow rules, but rather transform them into its own representation, keeping the same functionality in terms of matching and action space. If there is no matching stored flow rule within the ovs kernel module, the packet is redirected to the ovs-vswitchd module. Assuming a matching rule within the user space, the ovs-vswitchd redirects the packet back to the ovs kernel module along with the actions such as packet header modifications and output port information, which are subsequently applied by the ovs kernel module. The ovs-vswitchd can also instruct the ovs kernel module to cache the given action information in order to process subsequent packets without the interaction of the user space, which speeds up the forwarding process for those subsequent packets.

Dividing the forwarding processing into those two major components has two benefits: First, the kernel module can be implemented in a simple fashion, optimized for certain platforms with respect to speed and reliability. Extended functionality, such as supporting new OpenFlow versions, can be implemented within the ovs-vswitch daemon, without the need for extensive changes within the ovs kernel module.

Second, the ovs-vswitch daemon located in the user space has access to an extensive amount of memory in order to store flow rules instructed by the controller. Frequently used flow rules can be cached in the ovs kernel module for combining fast processing with a large flow table space based on the ovs-vswitch daemon.

The following results of the flow operation performance experiments support the benefits from this two stage implementation.

Flow Installation Performance

The Open vSwitch is evaluated with a similar parameter set. Figure 9.8 illustrates the results of the performance experiment. The achievable flow installation performance varies from up to 16 000 to less than 2000 flow installations per second. The reason for this widespread interval is mainly founded on the possible error of the time measurement based on the captured packet trace. This time is drawn with the red line, which shows significant peaks between 1000 and 17 000 installed flow rules. Such a large difference in the time measurement between the red and green line, could be due to the short oval installation time measured for the Open vSwitch. Installing up to 10 000 flow rules only takes less than one second, which increases the probability of failure regarding the precision of the time measurement.

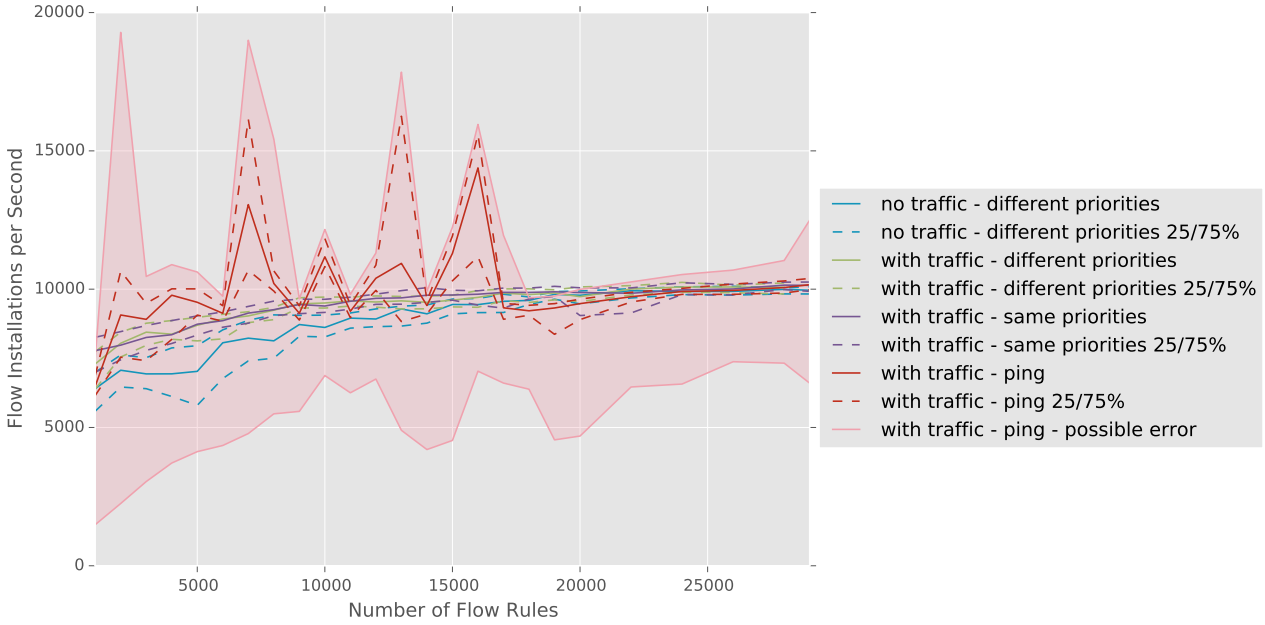

Figure 9.8: Flow Rule Installation Performance of Open vSwitch

However, for flow installations of more than 10 000 flow rules, the Open vSwitch shows an converging flow installation of 10 000 flow rules per seconds, for all parameter settings. In addition, the Open vSwitch does not seem to be affected by different flow rule priorities (green line) compared to identical priorities (purple line).

It is worth noting, that the flow installation performance without traffic (blue line) at least for an amount of flow rules below 15 000 is less than the performance achieved while processing additional traffic. A similar observation can be made for the flow modification performance.

Flow Modification Performance

Figure 9.9 depicts the results of the flow modification performance measurement, which shows similarities with the flow installation performance. The overall flow modification performance converges with an increasing number of flow rule modification towards 10 000 flow rule modifications per second. Modifications with less than 15 000 flow rules to process show a lower modification performance, with minimal less than 6 000 flow modifications per second.

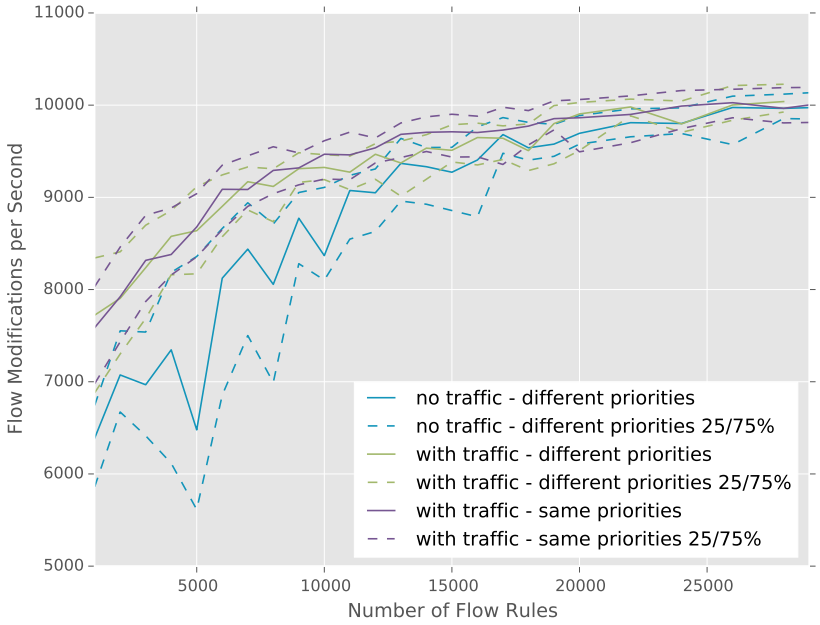

Figure 9.9: Flow Rule Modification Performance of Open vSwitch

This supports the observation made for the flow installation performance, where the Open vSwitch also showed a lower performance for smaller number of flow rules. Again the experiment performed without traffic shows an overall smaller flow modification performance, with the greater differences at the lower number of flow rules, even though the switch should be require less processing power if no traffic needs to be processed. These results in combination with the results depicted in Figure 9.8 indicate a optimization of the Open vSwitch performance for the scenarios, where the Open vSwitch is under workload.

Performance Differences between NEC P5420 and Open vSwitch

Since both evaluated switches are part of the prototype topology (Figure 8.1), it is worth comparing the results from the performance experiments. The service chaining system design (Chapter 7) does not require different priorities assigned to the flow rules, since each flow match is distinct without overlapping with other flow match specifications. Therefore, the performance achieved by both switches with same priorities within the flow rule set and with additional traffic to be forward is crucial for the performance, which can be assumed to be achieved within the service chaining prototype.

The NEC switch delivers a stable flow installation performance of close to 800 flow rules per second, even for larger amount of flow rules. The Open vSwitch delivers up to 10 000 flow installations per second in the same scenario, which is a remarkable difference of more than the order of magnitude of 10. A similar relation can be shown for the flow modification performance, where the Open vSwitch

also delivers 10 000 flow modification per second, where as the NEC switch is able to perform 990 flow modifications per second. The difference remains an order of magnitude of greater than 10.

This large difference in both performance indications is especially affective for the later evaluation of the hierarchical switch topology concept, where specific flow rules are moved from the Open vSwitch to the NEC hardware switch. Such a large difference is surprising, since a previous Open vSwitch version 1.7.0 running on an Intel Xeon 3210 was reported [42] to be capable of a flow setup rate of just 408 flows/s.

9.4 Service Chaining Topology

Parameter

The service chaining topology is evaluated with focusing on the failure of a VM host use case described in Section 9.1.3. The parameter set for this evaluation consists of two basic parameter: First, the number of user within the service chaining system. Since each user is assigned with a specific path for its network traffic, the number of user proportional to the number of flow rules required to configure those user specific paths in the network. The number of user are evaluated for both the hierarchical switch topology and the label switching concept in a range of 1000 to 10 000 user.

The second parameter is dependent on the hierarchical switch topology concept, where a subset of flow rules is moved by the controller from one switch to another switch, in order to achieve a more equal distribution of flow rules and workload. Therefore, different percentages of flow rules, which are moved by the controller are evaluated with respect to their impact on the overall system behavior and especially their impact on the failover time.

Metric

Time is the main metric, which is evaluated in each experiment with a different parameter sets. The main objective of the system design (Chapter 7) is to reduce the time required for restoring a valid forwarding behavior after a failure of a VM host. Therefore, the time required for changing the flow table configuration is measured for each switch within the topology. The time measurement is based on the observations described in Section 9.2. Therefore, the arrival time of the barrier reply at the ethernet interface of the controller is captured, since the controller has to deal with multiple switches concurrently, which showed time shiftings in the processing of barrier replies at the controller.

Topology

Figure 9.10 depicts the topology used within each service chaining experiment. The functionalities and characteristics of each of the five hosts are described in the Section 8.1. The service nodes host different amount of service instances, with $\texttt{Service\_Node}_1$ hosting four service instances and $\texttt{Service\_Node}_2$ hosting eight service instances in total. Four of those eight service instances are unused, since they are reserved as backup service instances in case of the simulation of a failure of $\texttt{Service\_Node}_1$

Table 9.1 gives an overview over the hardware information of the components used in the service chaining experiment, along with the software installed and used on each host.

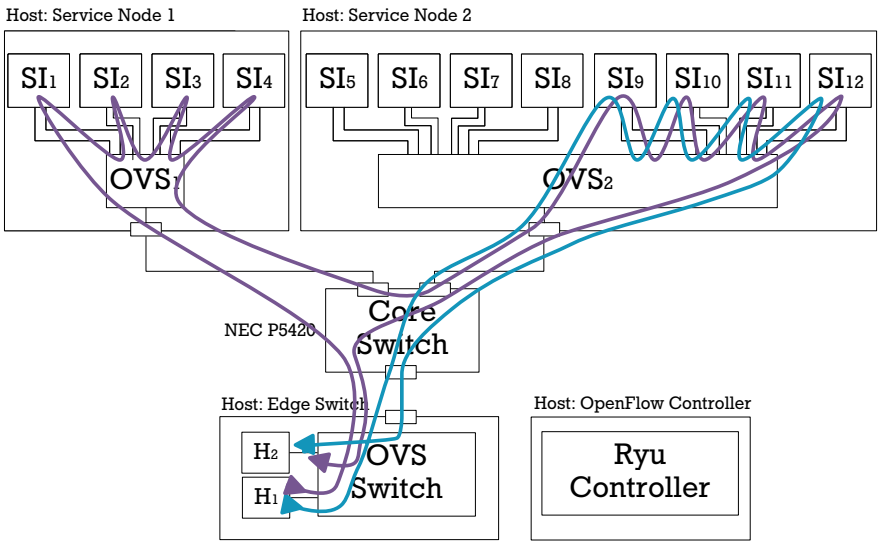

Figure 9.10: Testbed Topology

Host / Switch	Hardware Information	Software Information
Edge Switch	CPU: Intel(R) Pentium(R) CPU G640 @ 2.80GHz RAM: 8GB Network Card: Intel 82579LM Gigabit	hping3: Version 3.0.0-alpha-2 tcpdump: Version 4.2.1 Mininet: Version 2.2.1rc1 Open vSwitch: Version 2.3.1
Service Node	CPU: Intel(R) Pentium(R) CPU G640 @ 2.80GHz RAM: 8GB Network Card: Intel 82579LM Gigabit	Mininet: Version 2.2.1rc1 Open vSwitch: Version 2.3.1
Controller	Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz RAM: 32GB Network Card: Intel 82579LM Gigabit	Ryu: Version 3.19
NEC P5420	-	Firmware: OS-F3PA Ver. V5.1.1.0

Table 9.1: Hardware Information of the Prototype

Forwarding Configuration

The purple and blue line indicate the two different paths which are configured for the evaluation. The purple path is always used by the first half of the number of user and the blue path, which only includes service instances located at the $\texttt{Service\_Node}_2$, is used for the second half of the users. For 1000 initial user, this results in 500 user assigned to the purple path and 500 assigned to the blue path. This splitting is implemented, since the $\texttt{Service\_Node}_1$ is performing the failure simulation, which forces the system to allocate all user, which are affected by the failure to a new service node. In this case, the users assigned to the purple path are reallocated to a new path, where the service instances 5-6 are used as new service instances Instead, of the service instances 1-4. During a failure the blue path remains the same. The user assigned to the blue path represent a certain workload, which has to be imposed to the $\texttt{Service\_Node}_2$, since in reality there would not be just a service node without any traffic already assigned to it ready for acting as a backup service node. Instead, the user with failed service instances would be distributed among all other service nodes.

In case of the failure of Service_Node$_1$, a new path has to be configured for each user assigned to the purple path. This includes redirecting the traffic of those user to Service_Node$_2$, in order to avoid Service_Node$_1$ and configuring the forwarding for the new service instances 5-6 at Service_Node$_2$. Independent of the actual forwarding scheme, flow rules are required to be installed on Service_Node$_2$. The redirection of the traffic to Service_Node$_2$ can be either configured at the Edge_Switch (label switching concept) or must be configured at the Core_Switch (without usage of the label switching concept). The results in terms of required time to perform both forwarding reconfigurations are presented in the following sections.

9.4.1 Ryu Controller Performance

Section 9.2.4 has discussed a possible performance bottleneck within the Ryu SDN Framework. In order to measure accurate time durations in the following experiments, the barrier receive time at the interface of the controller is measured. In order to avoid additional time drifts, caused by potential performance bottlenecks, all required flow rules are calculated for each experiment in advance. Since, all forwarding path for each user are relatively static and know in advance, the required flow rules for both the initial setup and the failure of a VM host can be calculated before starting the experiment.

9.4.2 Naive Approach

The naive approach implements the forwarding configuration described in 9.4 by relying on per user path flow rules installed in the Core_Switch. For implementing a path shown with the purple line in Figure 9.10, six flow rules are necessary at the core switch, since the path traverses the core switch three times for one direction and additional three times for an answer following the reverse path. In case of the failover use case, the flow rules per user at the Core_Switch and at the Service_Node$_2$ need to be modified or newly installed.

The left side of Figure 9.11 illustrates the time (y-axis) required for installing all necessary flow rules in each of the four switches four an increasing number of user on the x-axis. The overall installation time is completely defined by the *Core_Switch*, since the corresponding red line is higher than each other time line over the complete range of user. The difference between the *Core_Switch* and the time spent at Service_Node$_2$ even increases with a higher number of user.

The right side of Figure 9.11 illustrates the time per switch required to install and modify the flow rules in case of a failure of Service_Node$_1$. This time is referred to as the failover time in the following paragraph. The failover time is also completely defined by the Core_Switch, with an overall trend similar to the increasing time difference between both lines for a higher user number. One possible way of reducing the overall time required for reconfiguring the network in case of a VM host failure would be to reduce the amount of time required by the Core_Switch, since it is always slower than the Service_Node$_2$.

9.4.3 Label Switching Concept

As a first resource optimization approach the label switching concept is applied to the service chaining prototype. The main optimization affects the Core_Switch, since it only requires a fixed set of flow rules regardless of the number of user. Instead, the number of flow rule required for the Core_Switch is given by the number of required actions. In the service chaining topology five actions can implement the forwarding functionality required for both possible user path.

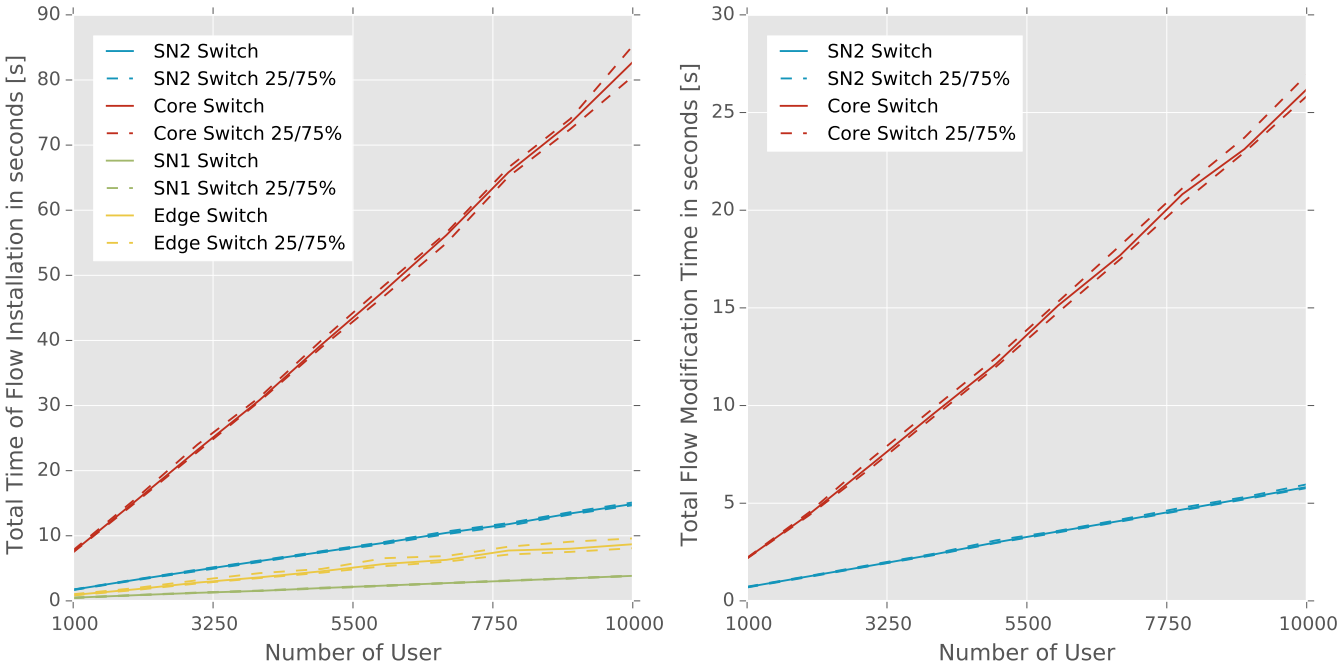

Figure 9.11: Flow Installation and Failover Time without Label Switching Concept

The left side of Figure 9.12 illustrates the timings of each switch with the applied label switching concept. The `Core_Switch` requires the smallest amount of time, since it only installs five flow rules. All other three switches are each implemented using an Open vSwitch, therefore the performance of each switch should be similar. The difference in the required installation time is based on the different number of flow rules required per switch. The $\texttt{Service\_Node}_2$ switch requires the most amount of flow rules, therefore the overall installation time depends on this switch.

The right side of Figure 9.12 illustrates the timings for the failure scenario. Instead, of modifying flow rules at the `Core_Switch`, the path can be completely configured by modifying the flow rules at the edge switch. In addition to the new path information the flow rules connecting the service instances hosted by $\texttt{Service\_Node}_2$ are required to be installed. Therefore, the $\texttt{Service\_Node}_2$ switch is still responsible for the overall failover time.

The reduction of the failover time is depicted on the left side of Figure 9.13. The application of the label switching concept in this particular topology is able to reduce the failover time from 26 seconds (purple line) to just 6 seconds (green line), which equals an reduction of 77 % in terms or required time. It is worth noting, that the network implements the identical per user traffic forwarding behavior in both scenarios. The reduction is based on optimizing the flow table space requirements and the control bandwidth as two of the defined data plane resources in Section 3.2.

The left side of Figure 9.13 shows the number of flow rules modified at the `Core_Switch` (purple) and at the $\texttt{Service\_Node}_2$ switch (green). The application of the label switching concept clearly reduced the number of flow rules installed and modified in the failure scenario at the `Core_Switch` from being increasing with increasing number of user to a fixed number of just five flow rules. The application of the label switching concept does not affect the number of flow rules installed and modified at the $\texttt{Service\_Node}_2$ switch. However, the Open vSwitch is able to deal with more than twice the number of flow rules compared to the `Core_Switch`, but only requires 23 % of the time modifying those flow rules. This superior performance is mainly based on the larger available control bandwidth of the Open vSwitch.

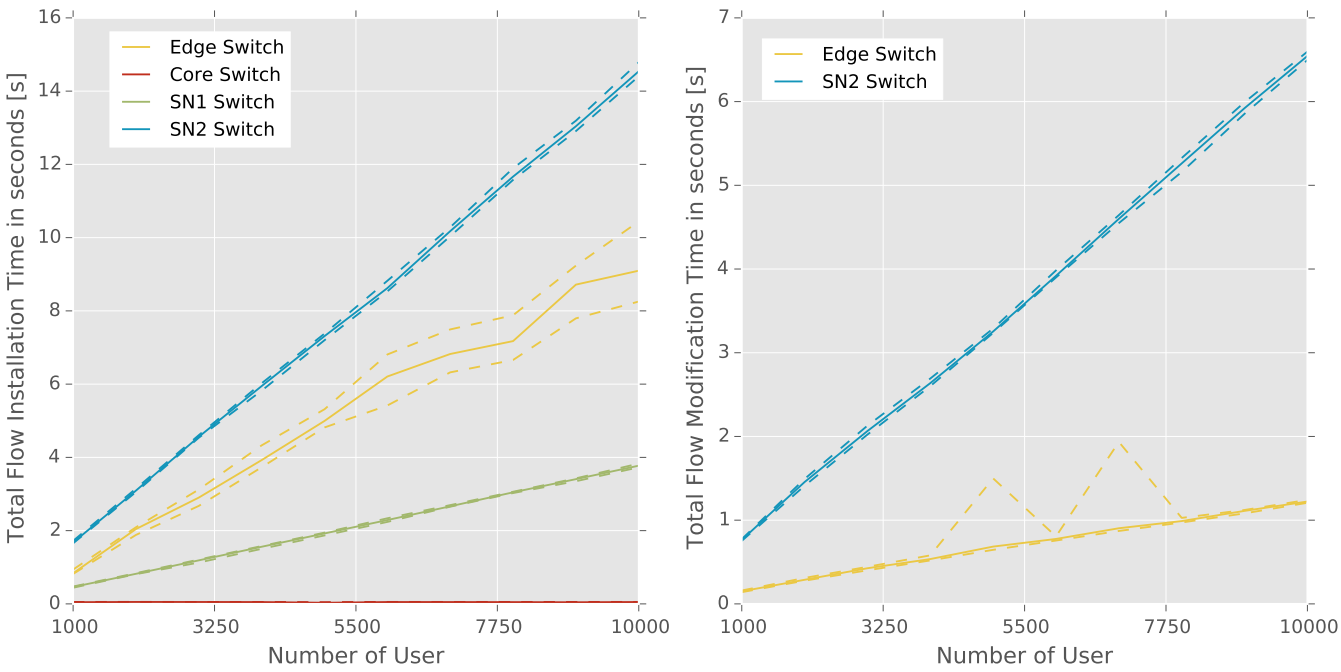

Figure 9.12: Flow Installation and Failover Time with Label Switching Concept

The label switching concept could prove its reduction in required control bandwidth and flow table space especially for the `Core_Switch`. However, its application requires changes within the network, mainly in terms of modifications somewhere in the packet header. The path information require bytes within those packet header to encode those information. For the prototype the bytes of the destination MAC address header field are used, requiring the label switching concept to be implemented in between two compatible end stations. Packets modified by an ingress switch require a second modification, before they can be delivered to the end host, which is not aware of the label switching concept and especially expects its own MAC address to be defined in the destination MAC address header field. Without additional customizable bytes within packet headers or even additional packet header, the label switching concept would always override certain information for its own path information. Those information need to be restorable at the egress switch.

9.4.4 Hierarchical Switch Topology Concept

The application of the label switching concept has shown a reduced failover time by 77 %, by cutting the number of flow rules stored at the `Core_Switch` to as low as just five required flow rules, which is equal to the number of ports in use for the prototype. This flow rule set list static, since it does not require any modification, because it matches label information within the destination MAC address. Instead, all modifications and installations are moved to the $\texttt{Service\_Node}_2$ switch and the `Edge_Node` switch. As a possible next step, the released resources in terms of flow table space and control bandwidth of the `Core_Switch` could potentially be used by assigning the switch to a hierarchical switch topology in conjunction with a second switch from the service chaining topology. Within this hierarchical switch topology, rules initially targeted for one switch can be installed at the `Core_Switch` instead, without changing the path length or forwarding behavior of any user.

It is worth noting, that flow rules always are moved to a switch which is one hop closer to the destination of a flow. Therefore, the traffic does not experience additional hops or delay on its path. The

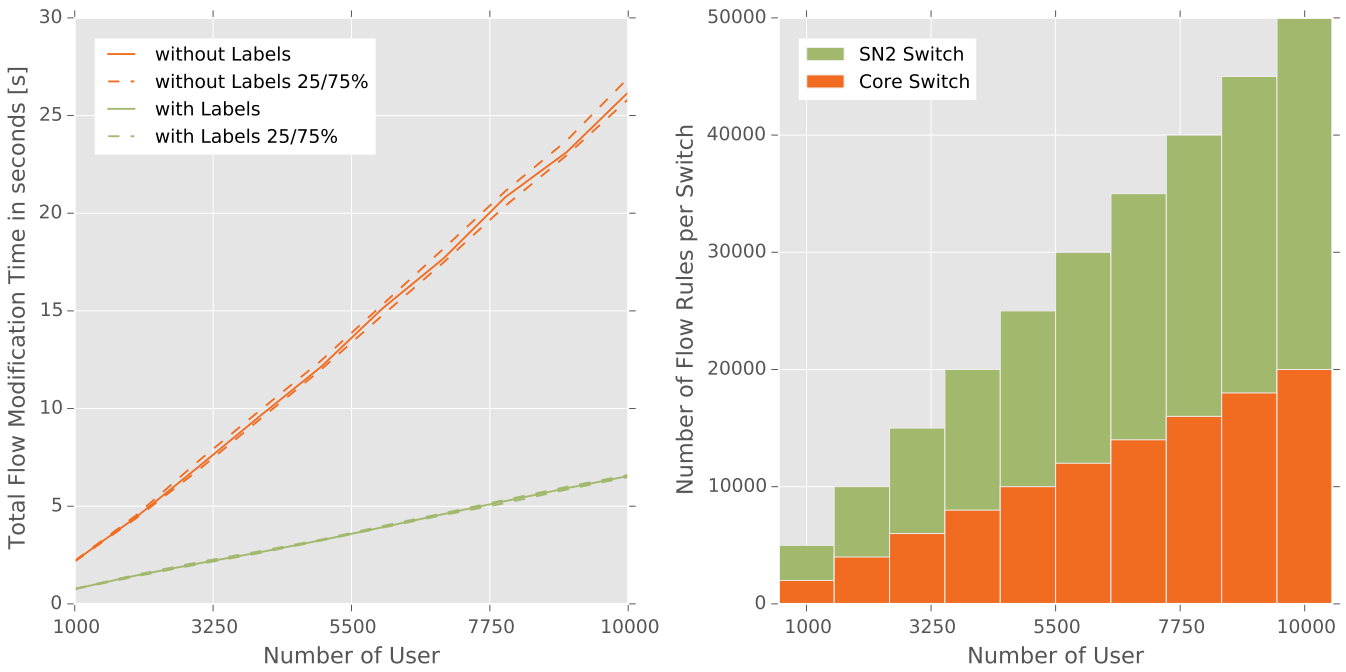

Figure 9.13: Total Failover Time with and without Label Usage

necessary labeling information are assigned at the `Core_Switch`, for moved flow rules, rather than being assigned at the $\texttt{Service\_Node}_2$ switch. This is crucial, since in a more complex scenario with multiple core switches flow rules could also be moved to a switch, which is not initially on the path of a flow, which would increase both the path length and the potential delay.

Figure 9.12 illustrated the failover time with the applied label switching concept. As it turned out, the failover time is completely dependent on the $\texttt{Service\_Node}_2$ switch. Therefore, this switch is a possible candidate for being part of a hierarchical switch topology in conjunction with the `Core_Switch`. Section 7.4.2 investigated on the question which flow rules can be moved to another switch and which one can not, in the context of the service chaining topology. For a service node, only one kind of flow rules are available for being moved to another switch. Applied on the topology described in Section 9.4, only two flow rules per user can be installed at the `Core_Switch` Instead, of the $\texttt{Service\_Node}_2$ switch. Those flow rules encode the path information for directing traffic from the service node to the destination node and for the reverse traffic from the service node back to the source host.

The left side of Figure 9.14 illustrates the number of flow rules modified and installed at the $\texttt{Service\_Node}_2$ switch. The topology is setup initially with 6000 user, but only 3000 user are affected by the failure scenario and only the flow rules of those 3000 user are available for being moved to the `Core_Switch`. The x-axis shows the different percentage of moved flow rules, ranging from 0% up to 50%, where 50% specifies that for half of the 3000 user, two flow rules are moved from the $\texttt{Service\_Node}_2$ switch to the `Core_Switch`. In comparison to the 30 000 flow rules stored at the $\texttt{Service\_Node}_2$ switch, only a small subset is actually moved to the `Core_Switch`, resulting in a small overall decrease of stored flow rules with a maximum of 10% of the flow rules moved to the `Core_Switch` at a moved percentage of 50%.

The right side of Figure 9.14 illustrates the time required for installing and modifying the decreasing number of flow rules, with an increasing percentage of moved flow rules. Moving 10% of the flow rules to the `Core_Switch` results in a overall time decrease of 0,4 seconds, since the switch has linear flow installation and modification performance as illustrated in Figure 9.9 in the range of 25 000 to 30 000

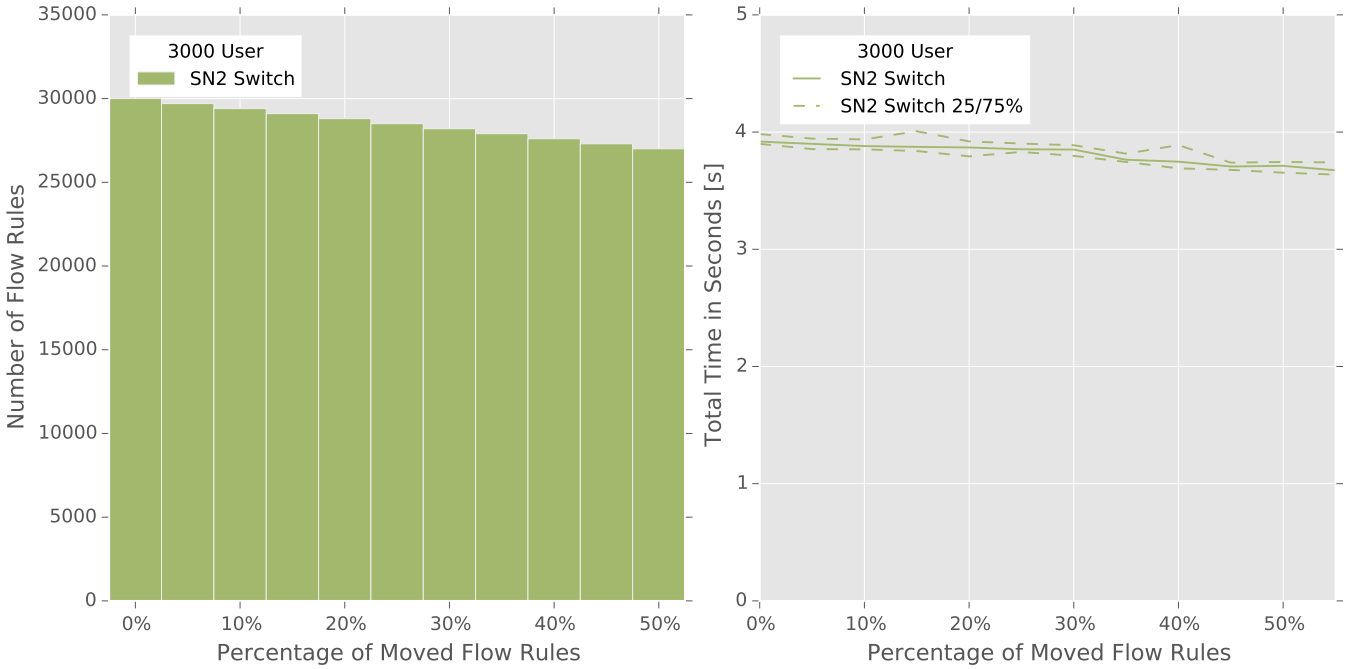

Figure 9.14: Affects of Flow Movement on the Failover Time for 3000 User at the $\texttt{Service\_Node}_2$ switch

flow rules. This time reduction could be increased by moving more than just 10% of the flow rules to the `Core_Switch`, but Figure 9.15 illustrates the limitation occurring at the `Core_Switch`.

The left side of Figure 9.15 illustrates the increasing number of flow rules moved to the `Core_Switch`, corresponding to the decreasing number depicted in Figure 9.14. The right side visualizes the time required to install those flow rules, which shows a rapid growing time, which already surpasses four seconds for more than 1250 flow rules, which are the flow rules corresponding to 20% of the 3000 user. Passing the installation of four seconds at this context results in a increase of the overall failover time, since the $\texttt{Service\_Node}_2$ switch only requires less than four seconds four its flow installation and modification.

It is worth noting that the minimal time reduction achieved for $\texttt{Service\_Node}_2$ switch comes at the cost of the fast increasing time required by the `Core_Switch`. Figure 9.16 illustrates the connection of these two times for four different amount of user. The blue line of each Figure illustrates the time required by the $\texttt{Service\_Node}_2$ switch and the red line illustrates the time required by the `Core_Switch`.

The crossing point between these two lines marks the point at which the overall failover time starts to be dependent on the `Core_Switch`, rather than being dependent on the $\texttt{Service\_Node}_2$ switch for the interval smaller than this crossing point. Even though it seemed practical to make use of the released flow table space and control bandwidth of the `Core_Switch`, the evaluation results lead to the following observation.

For none of the different user parameter a significant reduction of the failover time could be achieved. Even though the failure time does increase with an increasing amount of moved flow rules, the increase is such small, that it almost remains within the distribution of the values. In other words, even though the median of the measured time values decreases with a small amount, the values above the 75% quantile for the higher percentage of flow movement are equal to the median of the experiments without any moved flows. Therefore, no significant reduction can be identified.

The reasons for the hierarchical switch topology having no significant time reducing effect in this scenario are twofold. First, the amount of flow rules, which are moved from the $\texttt{Service\_Node}_2$ switch

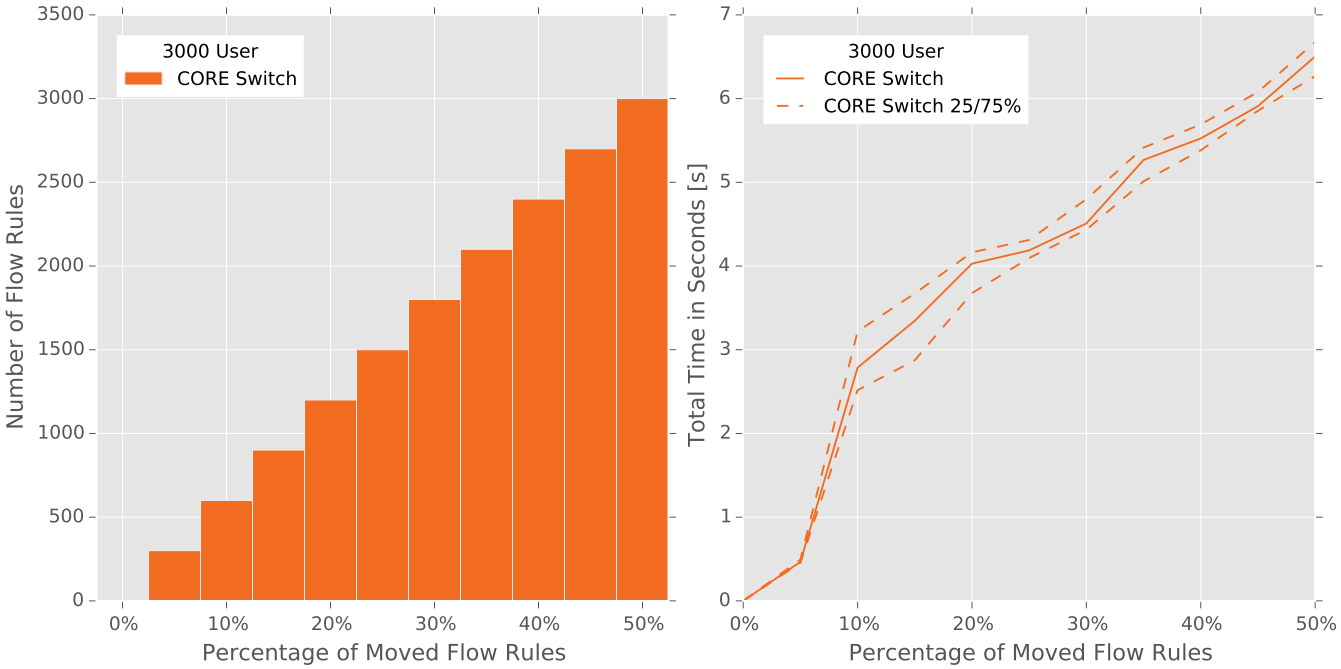

Figure 9.15: Affects of Flow Movement on the Failover Time for 3000 User at the `Core_Switch` switch

to the `Core_Switch` is too small for realizing a significant time difference for the time required by the $\texttt{Service\_Node}_2$ switch. Second, the difference in terms of the flow operation performance as discussed in Section 9.3 are to large. A significant time reduction would require two switches involved in the hierarchical switch topology, with more equal performance characteristics.

9.5 Flow Operation Performance Analysis

Moving flow rules from the $\texttt{Service\_Node}_2$ switch to the `Core_Switch` has shown no significant time reduction in the failover time based on the results presented in Section 9.4.4. One reason for these results are given by the large difference in terms of flow operation performance between the `Core_Switch` represented by the NEC hardware switch and the $\texttt{Service\_Node}_2$ switch, which is based on an Open vSwitch.

$$T_{Si}(N_{Si}) = \frac{N_{Si}}{P_{Si}} \tag{9.5}$$

where:

S_i = Switch with the Index i
T_{Si} = Total Flow Operation Time
N_{Si} = Number of Flow Operations
P_{Si} = Flow Operation Performance

This section investigates the relations between the flow operation performance and the achievable time reduction. The time required for a switch S_i to perform operations, such as installation or modification,

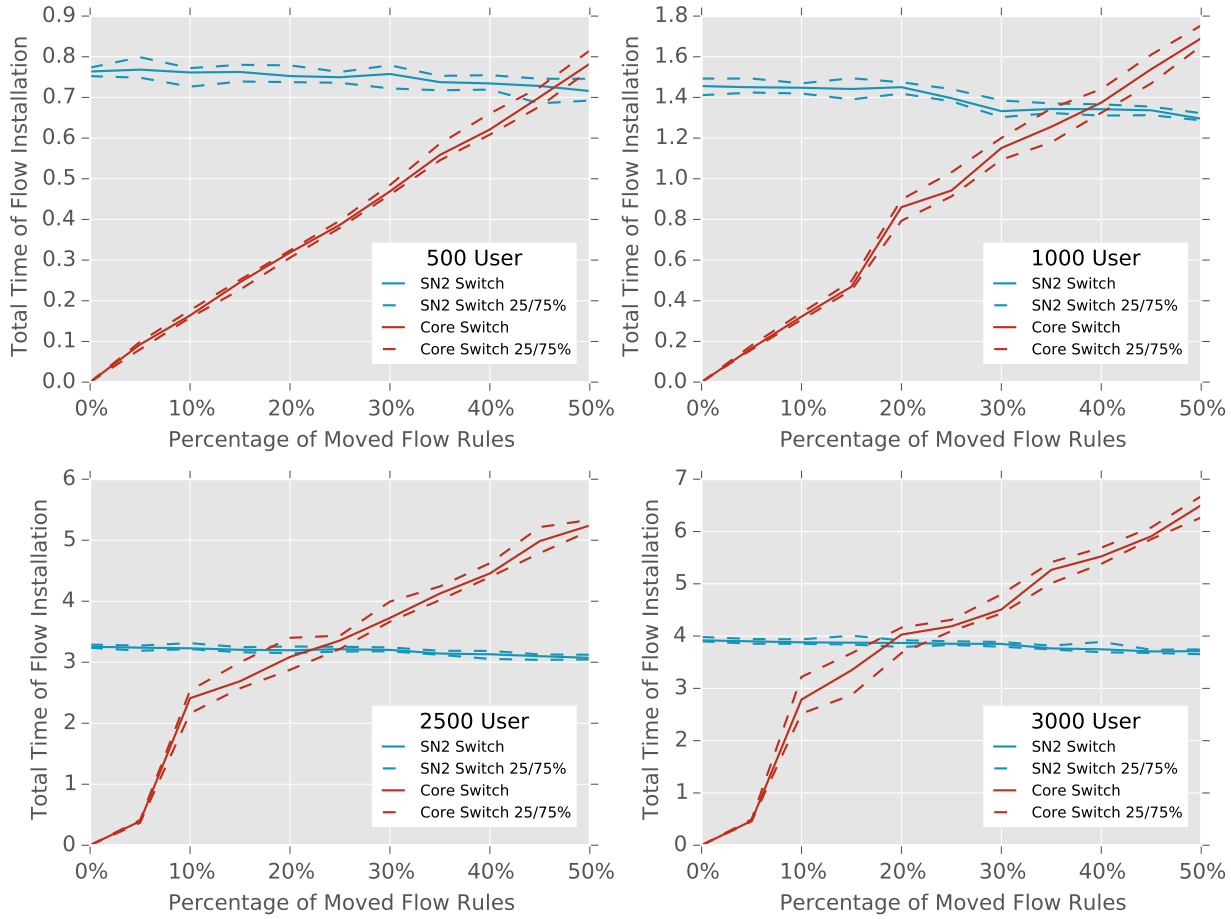

Figure 9.16: Failover time with the Hierarchical Switch Topology Concept

for N_{Si} number of flow operations is given by the Equation 9.5. The number of flow operations can simply be divided by the flow operation performance factor, which indicates how many flow operations a switch can perform per second. Given the results from Section 9.3, from a certain number of flow operations and ongoing, both the NEC and the Open vSwitch showed their flow operation performance being constant, at least assuming flow rules with identical priorities for the NEC switch. Therefore, the total flow operation time T_{Si} can be approximated using a fixed flow operation performance P_{Si}. It is worth noting, that other research reports exponential growth for the time required to add or modify entries within the flow table for an Open vSwitch and other hardware switches [85]. However, they only evaluated the time requirements in a range of 1 to 1 000 flow operations, where the evaluation of the flow operation performance in this work is based on a range of 1 000 up to 10 000 flow rules.

Moving a certain number of flow rules to a switch $S2$, which has potentially less flow operation performance, is limited by the time required for a switch $S1$ to perform the remaining non moved flow operations. Therefore, the maximum time for a switch $S2$ can be set to equal the time required for the switch $S1$ shown in Equation 9.6. $S2$ could be faster than this time , which does not affect the overall flow operation time, but should not be slower than $S1$, since it would then increase the total flow operations time. This is the case for the `Core_Switch` for a rate of moved flow rules of more than 20% shown in Section 9.4.4.

$$T_{S1}(N_{S1}) \geq T_{S2}(N_{S2}) \tag{9.6}$$

Including Equation 9.5 into Equation 9.6 results in the Equation 9.7, which indicates a first relation between the performance of $S1$ compared to the performance of $S2$, given a certain distribution of flow rule N_{S1} operations for $S1$ and N_{S2}, which are the number of flow operations for $S2$.

$$P_{S1}(N_{S1}, N_{S2}, P_{S2}) = \frac{N_{S2}}{N_{S1}} \times P_{S2} \tag{9.7}$$

In addition N_{S1} can be simply substituted by the relation between the total number of flow operations being the sum of N_{S1} and N_{S2}, given by Equation 9.8.

$$N_{S1} = N - N_{S2} \tag{9.8}$$

where:

$N =$ Number of Total Flow Operations

As a result of the substitution of N_{S1}, the flow operation performance relation between $S1$ and $S2$ is given by Equation 9.9. The relation between both flow operation performance is independent of the total number of flow operations N and only depends on the percentage of moved flow operations M to $S2$, since the flow operation performance was shown to stay equal after a certain number of flow operations. For a better understanding the Equation 9.10 lists some example values, for a percentage of moved flow operations from $S1$ to $S2$ of 10%, 20%, 40%, 50%.

$$\frac{P_{S2}}{P_{S1}}(M) = \frac{1}{\frac{1}{M} - 1} \tag{9.9}$$

where:

$M =$ Percentage of the moved flow operations

The results given in Equation 9.10, can directly be translated into time reduction of the total flow operation, since Equation 9.5 shows a linear dependency between the flow operation time and the number of flow operations. Therefore, a percentage of 20% of moved flow operations to a switch $S2$ reduces the flow operation time of $S1$ by 20%. In order to achieve those 20% time reduction, a switch $S2$ requires at least 25% of the flow operation performance of switch $S1$. For a reduction of 40% it requires at least 60% and for cutting the time in half it would require the same flow operation performance than the switch $S1$.

$$\frac{P_{S2}}{P_{S1}}(0,1) = 0,1\overline{1} \qquad \frac{P_{S2}}{P_{S1}}(0,2) = 0,25 \qquad \frac{P_{S2}}{P_{S1}}(0,4) = 0,6\overline{6} \qquad \frac{P_{S2}}{P_{S1}}(0,5) = 1 \tag{9.10}$$

Figure 9.17 shows the trend of Equation 9.9. The lower left dot indicates the performance relation between the Open vSwitch and the NEC P5420, where the NEC switch achieves less than 10% of the

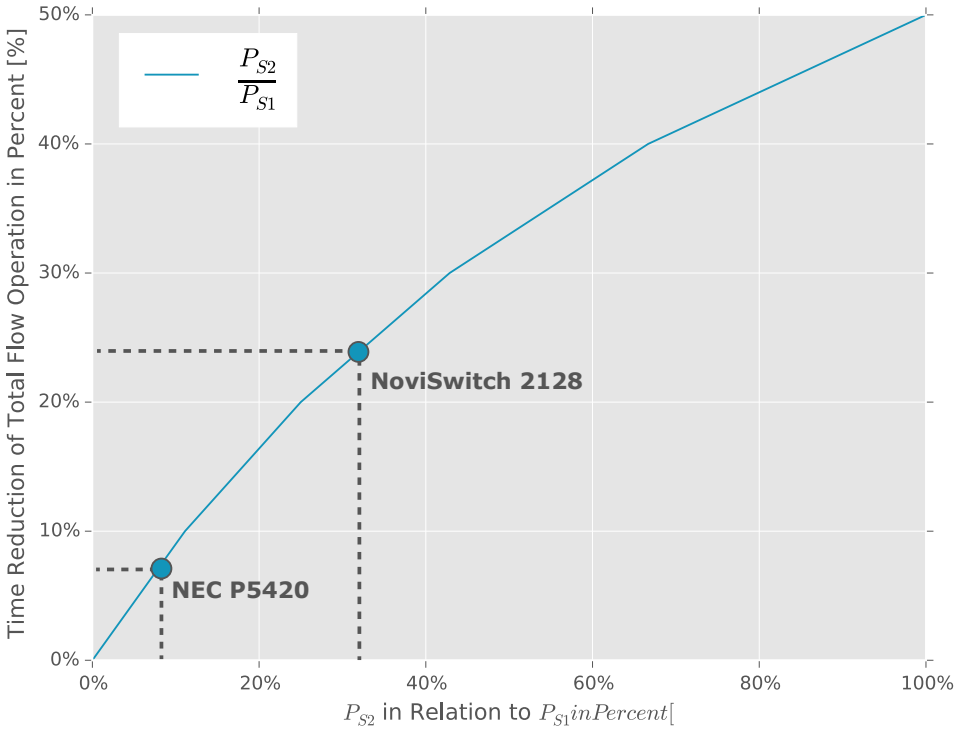

Figure 9.17: Illustration of Equation 9.9

flow operation performance of the Open vSwitch, which is visualized by the x-axis. Therefore, it can only reduce the total flow operation time by less than 7% (y-axis). Instead, of using the NEC P5420 as a core switch, another switch offering a better flow operation performance can result in a more significant time reduction. The second dot depicted in Figure 9.17 shows the approximated time reduction achievable with the NoviSwitch 2128 [1], which according to its data sheet offers a flow operation performance of 3200 flow modifications per second. Offering 32% of the flow operation performance of the Open vSwitch could reduce the failure time by 24%. It becomes clear, that higher flow operation performance at the core switch is beneficial for reducing the failure time, however for a higher flow operation performance the slope of the time reduction decreases, decreasing the effect of the hierarchical switch topology concept.

9.6 Comparison with other Service Chaining Concepts

After presenting the results achieved by applying the label switching concept and the hierarchical switch topology concept in conjunction to a dynamic service chaining system, it is worth noting how the achievements can be compared to other concepts developed in the context of service chaining systems. One of those recent developments is StEERING [117]. The concept developed for this approach also identified the flow table space as a major limitation for implementing a dynamic service chaining system, since the requirements of flow table space can dramatically increase with the product of applications and user of the system.

StEERING describes a specially developed approach for service chaining systems. It addresses the challenge of high demands of flow table space in dynamic service chaining concepts by utilizing the multi table feature added in OpenFlow 1.1 [33]. Instead, of storing individual exact matching flow rules

[1] http://noviflow.com/products/noviswitch/ [Last accessed: 21.05.2015]

per user or per application, StEERING tries to provide flow matches distributed over multiple tables. In order to match a flow match entry on a higher table, information about the matched entry in a previous table is carried via the metadata field supported by OpenFlow.

The requirements of flow table space for implementing the StEERING concept is the main metric for comparing this concepts with the prototype implemented for this work. StEERING achieves a linear growth of flow rules per switch, with an increasing number of user. This linear growth is an important improvement over having a higher demand of flow table space, if flow rules are required to be stored on a per user and per application basis. However, StEERING requires a similar amount of flow rules for each switch within the overall service chaining topology, which includes potential hardware switches for interconnecting different service nodes. StEERING does not address flow table resource constraints of those switches. These varying hardware characteristics are subject to the optimization of the label switching concept implemented in the prototype, which reduced the flow rules installed in the `Core_Switch` significantly to the number of ports used at the switch. The amount of flow rules only grows for the edge and service node switches. In the current implementation each user and application would require at least one exact matching rule at the edge switch and multiple flow rules on the service node depending on the number of service instances hosted by the service node. However, in a future extension the prototype could easily adopt the multi table feature added in OpenFlow 1.1, in order to avoid an extensive increase of flow rules.

Even though the hierarchical switch topology showed no significant benefit in the prototype, Section 9.5 discussed improved results for a smaller performance difference between an edge and a core switch. Such a distribution of the workload imposed for flow operations is not supported by the StEERING concept. In addition the hierarchical switch topology concept supports distribution of workload and flow rule operations across two switches, where one of those switches is an `Core_Switch`

Comparing the results achieved for reducing the failover time is challenging, since the evaluation presented for StEERING has focused on other metrics, which are not directly comparable with the results presented in this evaluation. However, the further reduction in terms of flow table space requirements at the core switch layer demonstrate, how a general resource optimization concept, such as the label switching approach can be adapted with some simple changes to a more specific challenge given by a dynamic service chaining system.

9.7 Conclusion

The results of the presented evaluation can be sorted into two categories: Fundamental results as being observed for the difficulties regarding a precise time measurement (Section 9.2) as well as the results presented for the flow operation performance (Section 9.3). And more specific results in the scope of the service chaining topology (Section 9.4)

The results of the presented evaluation can be divided into three kinds: First, the difficulties regarding a precise time measurement utilizing the Ryu SDN Framework could be highlighted along with an appropriate set of methodologies for tackle those difficulties.The most reliable method with respect to the effort required for performing the time measurement has been the barrier request time measured at the packet arrival time of the barrier reply.

The second outcome of this evaluation has given some insights into the flow operation performance characteristic of both the NEC P5420 hardware switch as well as a recent Open vSwitch implementation. The Open vSwitch demonstrated it higher capability of an order of magnitude of 10 higher performance in both disciplines. However, the experiments were focused on the capabilities defined by the control bandwidth, rather than including forwarding speed as an additional experiment. Hardware switches in

general are more capable of forwarding traffic with a higher speed due to their specialized hardware. The Open vSwitch in comparison showed how capable its flow installation and modification performance can be while running on commodity hardware.

The third outcome showed the achievements possible by applying two of the introduced general resource optimization concepts, to a dynamic service chaining system. The application of the label switching concept showed a promising reduction of the failover time by as much as 77%. Without reducing any functionality in the forwarding behavior of the overall network. The number of flow rules in the `Core_Switch` is significantly reduced by the label switching concept, by encoding path information at the edge of the network, rather than having those information redundantly stored in multiple flow rules across the network. Applying the label switching concept to the service chaining topology was implemented by using the destination MAC address header field, however since there is no dedicated customizable header field available in todays packet header, the label switching concept always overrides certain information which need to be restored at the egress switch. Beside this information override, the label switching concept did not introduce additional costs for the system.

The application of the hierarchical switch topology concept has shown less effective results in further reducing the failover time. The initial idea of making use of the released resources of the `Core_Switch` could be implemented, but without leading to a significant reduction in the failover time. The large performance difference between the hardware switch and the Open vSwitch are identified as a main reason for the results. A theoretical analysis was performed, which showed the potential effectiveness of the hierarchical switch topology concept. The time reduction mainly depends on the flow operation performance of each switch. A higher flow operation performance at the core switch could therefore significantly increase the benefit of the hierarchical switch topology.

10 Conclusion and Future Work

The contribution of this work is twofold: First, a structural analysis of bottlenecks and limitations of current OpenFlow data plane implementation is provided. Those bottlenecks are categorized in data plane resources, which are used to compare the optimization goal and results of state-of-the-art optimization approaches.

Second, a categorization of resource optimization and application interaction concept is developed. For each context four distinct concepts could be defined, showing the similarities between different specific approaches found in the literature. A long-term goal of such a categorization can be to utilize the defined classes of resource optimization and application interaction concepts to be applied automatically to a given SDN application, rather than being required to develop concepts specifically for each given application. The information provided by this categorization can further be used to develop novel abstraction in the area of SDN, where an abstraction can result in an optimized usage of a certain data plane resource.

In order to show the usefulness of the categorization, a combination of two resource optimization concepts was applied to a recent approach known as dynamic network service chaining. In order to assess this application of two of the defined concepts, a prototype was developed implementing the basic characteristics of a dynamic network service chaining system, plus the resource optimizations defined by the defined concepts. Since some major limitations in the context of OpenFlow are specific to OpenFlow-enabled hardware switches, a NEC hardware switch was used to implement the prototype, in order to show the impact of the resource optimization on a hardware switch, rather than relying completely on the emulation of network devices.

The evaluation of this prototype has shown a significant optimization potential, which could be achieved by applying the label switching concept to the dynamic network service chaining system. The failover time evaluated in case of a failure of a VM host could be reduced by 77% by the usage of labels. StEERING, another concept for dynamic network service chaining in comparison achieves a linear growth for flow rules per user for each switch within the network. Different switch capabilities in terms of flow table space and flow operation performance are not considers by StEERING.

The hierarchical switch topology concept was applied in a next step for distributing the workload across multiple switches. However, there was no significant optimization achievable in this scenario in addition to the optimization achieved by the label switching concept. The expected results of the hierarchical switch topology concept could not be detected due to the large difference in terms of flow operation performance of the NEC switch compared to the Open vSwitch. Such a large difference was not expected given the numbers reported in the literature, which reported 408 flow installations per second for an earlier Open vSwitch version. Instead, the latest Open vSwitch version 2.3.1 was able to achieve up to 10 000 flow operations per second.

In addition, the evaluation has investigated some metrics of the achievable switch performance in terms of flow installations and modifications per second for both an Open vSwitch and the NEC hardware switch. In summary, the flow installation and modification performance achieved by the NEC switch is around 10 times less than the same performance metric achieved by the Open vSwitch. The focus on evaluating the switch performance was set to the installation and modification rate, since these metrics

were important for the later prototype evaluation. Metric such as forwarding speed were not considered for this evaluation.

Overall, this work has given insights in the fields of optimizing OpenFlow resource consumption with a strong focus on the data plane implementation of today's OpenFlow-enabled hardware. It therefore contributes steps towards a novel concept of constructing and developing resource optimized SDN applications.

Future Work

The performance evaluation of both the Open vSwitch and the NEC hardware switch can be further extended, in order to complete the picture of the potential usable switching performance of both devices. The flow operation performance was evaluated using similar flow rules. They were similar in both the match and the action part. In a future evaluation, the flow rules could match a traffic based on a network traffic trace, gathered from real production networks.

Such a traffic trace can also be vital for a future evaluation of the dynamic network service chaining implementation. Instead of only relying on ping requests and replies as a workload traffic, such a network traffic trace can lead to results further matching the behavior of a real world implementation. In addition to a more realistic workload, the relation between processing power required for the Open vSwitch and the service instances running on the same VM host could be worth investigating. For the evaluation in this work, the processing power accessible by the Open vSwitch was not limited, however for being implemented in a data center environment, the processing power might be focused on the service instances, rather than on the Open vSwitch.

The resource optimization concepts presented in Chapter 4 give a first overview of possible classes of resource optimization concepts. There are further concepts possible, which can be developed similar to software design patterns. Those general concepts can result in a better understanding of how to optimize certain resources within a system. However, optimization should not be the final answer to certain limitations, instead limitations should be ultimately addressed at the root. In the context of OpenFlow and especially OpenFlow-enabled hardware, limitations such as the control bandwidth and flow table space are likely to be improved in future products, mainly because they are identified as some of the strongest limitations in the current hardware generations. The same applies for the introduced classes of application interaction concepts, which are a first set of possible concepts, which could be extended in the future.

The future development of networking hardware will include products specifically designed with the needs of OpenFlow in mind, which will overcome some of the resource limitations discussed in this work. However, limitations and challenges are likely to become smaller, but will always be present in the complexity of today's networking domain. Therefore, having common optimization patters and concepts can improve both the understanding of systems as well as their future optimization at the core development of switching hardware itself.

The resource optimization concepts introduced within this work could be applied to other challenging networking implementations, similar to the application at the dynamic network service chaining implementation. This can further help to understand if those general concepts can be applied like a toolbox, where a certain general tool can be used to deal with a specific issue. Tools can also be combined, such as it was evaluated for combining the label switching concept with the hierarchical switch topology concept.

The main focus of the evaluation was put on the resource optimization concept, whereas the application interaction concepts were only theoretically discussed. Nevertheless, application interaction

concepts have a strong potential in leveraging the new paradigms supported by SDN. A deeper interaction between network application and the network itself have not been implemented in a widespread scenario, due to some of the challenges discussed in this work, such as fairness. However, application interaction should be something to further investigate on, since some of the challenges facing today's networks can be solved by integrating the application more tightly within the network.

Bibliography

[1] HP 5400zl Switch Series. URL http://www8.hp.com/us/en/products/networking-switches/product-detail.html?oid=1827663#!tab=features [Last Accessed 15.05.2015].

[2] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DTCP). ACM SIGCOMM Computer Communication Review (CCR), 41 (4):63–74, 2011.

[3] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In ACM SIGCOMM Conference on Internet Measurement, 2010.

[4] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine grained Traffic Engineering for Data Centers. In ACM Conference on emerging Networking Experiments and Technologies (CONEXT), 2011.

[5] Y. Bernet. The Complementary Roles of RSVP and Differentiated Services in the Full-service QoS Network. IEEE Communications Magazine, 38(2):154–162, 2000.

[6] J. Blendin, J. Rückert, N. Leymann, G. Schyguda, and D. Hausheer. Position Paper: Software-Defined Network Service Chaining. In EWSDN Workshop, 2014.

[7] R. Braden, D. Clark, S. Shenker, et al. Integrated Services in the Internet Architecture: an Overview. RFC 1633, Internet Engineering Task Force, 1994.

[8] P. Brooks and B. Hestnes. User Measures of Quality of Experience: Why being objective and quantitative is important. IEEE Network, 24(2):8–13, 2010.

[9] T. Bu, L. Gao, and D. Towsley. On Characterizing BGP Routing Table Growth. Computer Networks, 45(1):45–54, 2004.

[10] E. Burger. Application-Layer Traffic Optimization (ALTO) Problem Statement. RFC 5693, Internet Engineering Task Force, 2009.

[11] M. Carlson, W. Weiss, S. Blake, Z. Wang, D. Black, and E. Davies. An Architecture for Differentiated Services. RFC 2475, Internet Engineering Task Force, 1998.

[12] M. Casado, T. Koponen, D. Moon, and S. Shenker. Rethinking Packet Forwarding Hardware. In HotNets, 2008.

[13] Y. Chen, T. Farley, and N. Ye. QoS Requirements of Network Applications on the Internet. INFORMATION KNOWLEDGE SYSTEMS Management, 4(1):55–76, 2004.

[14] Y. Chiba, Y. Shinohara, and H. Shimonishi. Source Flow: Handling Millions of Flows on Flow-based Nodes. ACM SIGCOMM Computer Communication Review (CCR), 41(4):465–466, 2011.

[15] Cisco. Cisco Visual Networking Index (VNI) Global and North America (NA) Mobile Data Traffic Forecast Update 2013-2018. Presentation, 2014.

[16] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2014–2019. White Paper, 2015.

[17] A. R. Curtis, W. Kim, and P. Yalagandula. Mahout: Low-overhead Datacenter Traffic Management using End-host-based Elephant Detection. In IEEE INFOCOM, 2011.

[18] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-performance Networks. 41(4):254–265, 2011.

[19] B. Davie and Y. Rekhter. MPLS: Technology and Applications. Morgan Kaufmann Publishers Inc., 2000.

[20] J. Dean and S. Ghemawat. MapReduce: simplified Data Processing on large Clusters. Communications of the ACM, 51(1):107–113, 2008.

[21] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching using Bloom Filters. In Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, 2003.

[22] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment Issues for the IP Multicast Service and Architecture. IEEE Network, 14(1):78–88, 2000.

[23] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella. Towards an elastic distributed SDN Controller. 43(4):7–12, 2013.

[24] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to scale Software Routers. In ACM SIGOPS on Operating systems principles, 2009.

[25] R. P. Draves, C. King, S. Venkatachary, and B. D. Zill. Constructing Optimal IP Routing Tables. In INFOCOM. Conference of the IEEE Computer and Communications Societies, 1999.

[26] P. Edholm. HP and Microsoft Demo OpenFlow Lync Applications-optimized Network, 2013. URL http://www.nojitter.com/post/240153039/hp-and-microsoft-demo-openflowlync-applicationsoptimized-network?cid=nl_nojitter_2013-04-18_html&elq=8800d9d402d145278e2d80370ebca183 [Last Accessed 15.05.2015].

[27] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Performance Characteristics of Virtual Switching. In Cloud Networking (CloudNet), 2014.

[28] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Assessing Soft-and Hardware Bottlenecks in PC-based Packet Forwarding Systems. ICN 2015, page 90, 2015.

[29] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory Networking: An API for Application Control of SDNs. 43(4):327–338, 2013.

[30] D. Ferrari, A. Banerjea, and H. Zhang. Network Support for Multimedia a Discussion of the Tenet Approach. Computer Networks and ISDN Systems, 26(10):1267–1280, 1994.

[31] J. Forgie. ST-A proposed Internet Stream Protocol. IEN 119, 1979.

[32] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In ACM SIGPLAN International Conference on Functional Programming, number 9, 2011.

[33] O. N. Foundation. OpenFlow Switch Specification v1.1.0. 2011.

[34] O. N. Foundation. OpenFlow Switch Specification v1.3.1. 2012.

[35] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan. RFC 4632, Internet Engineering Taskforce, 2006.

[36] A. Georgi, R. Budich, Y. Meeres, R. Sperber, and H. Hérenger. An Integrated SDN Architecture for Application Driven Networking. International Journal On Advances in Systems and Measurements, 7(1 and 2):103–114, 2014.

[37] D. Grossman. New Terminology and Clarifications for Diffserv. 2002.

[38] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. ACM SIGCOMM Computer Communication Review (CCR), 38(3):105–110, 2008.

[39] C. E. Hopps. Analysis of an equal-cost multi-path algorithm. 2000.

[40] M. Hosseini, D. T. Ahmed, S. Shirmohammadi, and N. D. Georganas. A Survey of Application-layer Multicast Protocols. IEEE Communications Surveys & Tutorials, 9(3):58–74, 2007.

[41] T. Hoßfeld and A. Binzenhöfer. Analysis of Skype VoIP traffic in UMTS: End-to-end QoS and QoE measurements. Computer Networks, 52(3):650–666, 2008.

[42] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity Switch Models for Software-defined Network Emulation. In ACM SIGCOMM workshop on Hot Topics in Software Defined Networking, 2013.

[43] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In USENIX Annual Technical Conference, 2010.

[44] Itu-T. Definition of terms related to Quality of Service. Telecommunication Standardization Sector of Itu, pages 1–30, 2008.

[45] A. S. Iyer, V. Mann, and N. R. Samineni. Switchreduce: Reducing Switch State and Controller Involvement in Openflow Networks. In IFIP Networking Conference, 2013, 2013.

[46] M. Jain and C. Dovrolis. End-to-end available Bandwidth: Measurement Methodology, Dynamics, and Relation with TCP Throughput. ACM SIGCOMM Computer Communication Review (CCR), 32(4):295–308, 2002.

[47] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed Software Defined WAN. ACM SIGCOMM Computer Communication Review (CCR), 43(4):3–14, 2013.

[48] M. Jarschel and R. Pries. An OpenFlow-based energy-efficient Data Center Approach. In ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication, 2012.

[49] M. Jarschel, F. Wamser, T. Hohn, T. Zinner, and P. Tran-Gia. SDN-based application-aware Networking on the Example of Youtube Video Streaming. In IEEE Software Defined Networks EWSDN Workshop, 2013.

[50] M. Jarschel, T. Zinner, T. Hoßfeld, P. Tran-Gia, and W. Kellerer. Interfaces, Attributes, and Use cases: A Compass for SDN. IEEE Communications Magazine, 52(6):210–217, 2014.

[51] W. John, K. Pentikousis, G. Agapiou, E. Jacob, M. Kind, A. Manzalini, F. Risso, D. Staessens, R. Steinert, and C. Meirosu. Research Directions in Network Service Chaining. In IEEE Future Networks and Services (SDN4FNS), 2013.

[52] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In ACM SIGCOMM conference on Internet Measurement Conference, 2009.

[53] Y. Kanizo, D. Hay, and I. Keslassy. Palette: Distributing Tables in Software-defined Networks. In IEEE INFOCOM, 2013.

[54] K. Kannan and S. Banerjee. Compact TCAM: Flow Entry Compaction in TCAM for Power aware SDN. In Distributed Computing and Networking, pages 439–444. 2013.

[55] K. Kannan and S. Banerjee. Flowmaster: Early Eviction of dead Flow on SDN Switches. In Distributed Computing and Networking, pages 484–498. 2014.

[56] N. Katta, O. Alipourfard, J. Rexford, and D. Walker. Infinite Cacheflow in Software-Defined Networks. In ACM Hot Topics in Software Defined Networking, 2014.

[57] H. Kim, K. C. Claffy, M. Fomenkov, D. Barman, M. Faloutsos, and K. Lee. Internet Traffic Classification demystified: Myths, Caveats, and the best Practices. In ACM Conference on emerging Networking Experiments and Technologies (CONEXT), 2008.

[58] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In ACM SIGCOMM Workshop on Hot Topics in Networks, 2010.

[59] A. Lara, A. Kolasani, and B. Ramamurthy. Network Innovation using Openflow: A Survey. IEEE Communications Surveys & Tutorials, 16(1):493–512, 2014.

[60] U. Lee, J. Lee, M. Ko, C. Lee, Y. Kim, S. Yang, K. Yatani, G. Gweon, K.-M. Chung, and J. Song. Hooked on Smartphones: An Exploratory Study on Smartphone overuse among College Students. In ACM Conference on Human Factors in Computing Systems (CHI), 2014.

[61] W. Lee, Y.-H. Choi, and N. Kim. Study on Virtual Service Chain for Secure Software-Defined Networking. Advanced Science and Technology Letters (ASTL), 29:177–180, 2013.

[62] N. Leymann and D. T. AG. Flexible Service Chaining. Requirements and Architectures. EWSDN. Presentation, 2013.

[63] T. A. Limoncelli. Openflow: A Radical new Idea in Networking. ACM Queue, 10(6):40, 2012.

[64] G. Liu and X. Lin. MPLS Performance Evaluation in Backbone Network. IEEE International Conference on Communications (ICC), 2:1179–1183, 2002.

[65] B. Lowekamp, N. Miller, T. Gross, P. Steenkiste, J. Subhlok, and D. Sutherland. A Resource Query Interface for Network-aware Applications. Cluster Computing, 2(2):139–151, 1999.

[66] B. Lowekamp, N. Miller, R. Karrer, T. Gross, and P. Steenkiste. Design, Implementation, and Evaluation of the REMOS Network Monitoring System. Journal of Grid Computing, 1(1):75–93, 2003.

[67] S. Luo, H. Yu, and L. M. Li. Fast Incremental Flow Table Aggregation in SDN. In IEEE Computer Communication and Networks (ICCCN), 2014.

[68] 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 (CCR), 38(2):69–74, 2008.

[69] C. R. Meiners, A. X. Liu, and E. Torng. Bit weaving: A non-prefix Approach to Compressing Packet Classifiers in TCAMs. IEEE/ACM Transactions on Networking (ToN), 20(2):488–500, 2012.

[70] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman. Application-aware Data Plane Processing in SDN. In ACM Hot Topics in Software Defined Networking, 2014.

[71] N. Miller and P. Steenkiste. Collecting Network Status Information for Network-aware Applications. 2:641–650, 2000.

[72] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. 9:39–50, 2009.

[73] H. Nam, K.-H. Kim, J. Y. Kim, and H. Schulzrinne. Towards QoE-aware Video Streaming using SDN. In IEEE Global Communications Conference (GLOBECOM), 2014.

[74] R. Narayanan, S. Kotha, G. Lin, A. Khan, S. Rizvi, W. Javed, H. Khan, and S. A. Khayam. Macroflows and Microflows: Enabling Rapid Network Innovation through a Split SDN Data Plane. In EWSDN Workshop, 2012.

[75] R. Narisetty and D. Gurkan. Identification of Network Measurement Challenges in OpenFlow-based Service Chaining. In IEEE Local Computer Networks Workshops (LCN Workshops), 2014.

[76] P. Newman, T. Lyon, and G. Minshall. Flow labelled IP: A connectionless Approach to ATM. 3: 1251–1260, 1996.

[77] K. Nichols, D. L. Black, S. Blake, and F. Baker. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, Internet Engineering Task Force, 1998.

[78] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, et al. The Design and Implementation of Open vSwitch. In USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2015.

[79] K. Phemius and M. Bouet. Monitoring Latency with Openflow. In Conference on Network and Service Management (CNSM), 2013.

[80] J. Postel and J. K. Reynolds. Standard for the Transmission of IP Datagrams over IEEE 802 Networks. RFC 1042, Internet Engineering Task Force, 1988.

[81] Z. A. Qazi, J. Lee, T. Jin, G. Bellala, M. Arndt, and G. Noubir. Application-awareness in SDN. 43 (4):487–488, 2013.

[82] P. Quinn and U. Elzur. Network service header, 2015. URL https://tools.ietf.org/html/draft-ietf-sfc-nsh-00 [Last Accessed 15.05.2015].

[83] R. M. Ramos, M. Martinello, and C. Esteve Rothenberg. SlickFlow: Resilient Source Routing in Data Center Networks unlocked by OpenFlow. In IEEE Local Computer Networks (LCN), 2013.

[84] P. Renals and G. A. Jacoby. Blocking Skype through Deep Packet Inspection. In IEEE System Sciences, HICSS, 2009.

[85] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In Passive and Active Measurement, 2012.

[86] J. Rückert, J. Blendin, and D. Hausheer. Software-Defined Multicast for Over-the-Top and Overlay-based Live Streaming in ISP Networks. Journal of Network and Systems Management, pages 1–29, 2014.

[87] B. Salisbury. TCAMs and OpenFlow: What every SDN Practitioner must know. URL https://www.sdncentral.com/technology/sdn-openflow-tcam-need-to-know/2012/07/ [Last Accessed 15.05.2015].

[88] S. Savage, A. Collins, E. Hoffman, J. Snell, and T. Anderson. The End-to-End Effects of Internet Path Selection. In ACM SIGCOMM Computer Communication Review (CCR), volume 29, 1999.

[89] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky. Advanced Study of SDN/OpenFlow Controllers. In ACM Central & Eastern European Software Engineering Conference in Russia, 2013.

[90] S. Shenker, M. Casado, T. Koponen, N. McKeown, et al. The Future of Networking, and the Past of Protocols. Open Networking Summit, 2011.

[91] H. Shimonishi, H. Ochiai, N. Enomoto, and A. Iwata. Building Hierarchical Switch Network using OpenFlow. In Intelligent Networking and Collaborative Systems (INCOS), 2009.

[92] E. Spitznagel, D. Taylor, and J. Turner. Packet Classification using cxtended TCAMs. In IEEE Network Protocols, 2003.

[93] J. Stretch. BGP route aggregation. URL http://packetlife.net/blog/2008/sep/19/bgp-route-aggregation-part-1/ [Last Accessed 15.05.2015].

[94] J. Su, G. Lv, Z. Sun, Y. Chen, and T. Li. Labelcast: A Novel Data Plane Abstraction for SDN.

[95] A. Tavakoli, M. Casado, T. Koponen, and S. Shenker. Applying NOX to the Datacenter. In HotNets, 2009.

[96] D. E. Taylor and J. S. Turner. Classbench: A Packet Classification Benchmark. In IEEE INFOCOM 2005. Conference of the IEEE Computer and Communications Societies, volume 3, 2005.

[97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. IEEE Communications Magazine, 35(1):80–86, 1997.

[98] K. Thompson, G. J. Miller, and R. Wilder. Wide-area Internet Traffic Patterns and Characteristics. IEEE Network, 11(6):10–23, 1997.

[99] A. Tootoonchian and Y. Ganjali. HyperFlow: A distributed control plane for OpenFlow. In Internet Network Management Conference on Research on Enterprise Networking (INM/WREN), 2010.

[100] A. Tootoonchian, M. Ghobadi, and Y. Ganjali. OpenTM: Traffic Matrix Estimator for OpenFlow Networks. In Passive and Active Measurement, 2010.

[101] Z. A. Uzmi, M. Nebel, A. Tariq, S. Jawad, R. Chen, A. Shaikh, J. Wang, and P. Francis. SMALTA: Practical and near-optimal FIB Aggregation. In ACM Conference on emerging Networking Experiments and Technologies (CONEXT), 2011.

[102] A. Voellmy, H. Kim, and N. Feamster. Procera: A Language for High-level Reactive Network Control. In ACM Hot topics in Software Defined Networks, 2012.

[103] F. Wamser, D. Hock, M. Seufert, B. Staehle, R. Pries, and P. Tran-Gia. Using buffered Playtime for QoE-oriented Resource Management of YouTube Video Streaming. Transactions on Emerging Telecommunications Technologies, 24(3):288–302, 2013.

[104] G. Wang, T. Ng, and A. Shaikh. Programming your Network at Run-time for Big Data Applications. In ACM Hot topics in Software Defined Networks, 2012.

[105] B. Warf. Geographies of Global Telephony in the Age of the Internet. Geoforum, 45:219–229, 2013.

[106] E. Weigle and W.-c. Feng. A Comparison of TCP automatic Tuning Techniques for Distributed Computing. In IEEE High Performance Distributed Computing, 2002.

[107] N. Williams and S. Zander. Real Time Traffic Classification and Prioritisation on a Home Router using DIFFUSE. Technical report, CAIA Technical Report, 2011.

[108] S. Winkler. Digital Video Quality: Vision Models and Metrics. John Wiley & Sons, 2005.

[109] R. Winter. The coming of Age of MPLS. IEEE Communications Magazine, 49(4):78–81, 2011.

[110] P. Wolfgang. Design Patterns for object-oriented Software Development. Reading, Mass.: Addison-Wesley, 1994.

[111] X. Xiao, A. Hannan, B. Bailey, and L. M. Ni. Traffic Engineering with MPLS in the Internet. IEEE Network, 14(2):28–33, 2000.

[112] K.-K. Yap, T.-Y. Huang, B. Dodson, M. S. Lam, and N. McKeown. Towards software-friendly Networks. In ACM Asia-Pacific Worksho Systems (APSYS), 2010.

[113] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On Scalability of Software-defined Networking. IEEE Communications Magazine, 51(2):136–141, 2013.

[114] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-based Networking with DIFANE. ACM SIGCOMM Computer Communication Review (CCR), 41(4):351–362, 2011.

[115] S. Zander, T. Nguyen, and G. Armitage. Automated Traffic Classification and Application Identification using Machine Learning. In IEEE Local Computer Networks, 2005.

[116] L. Zhang, S. Deering, D. Estrin, S. Shenker, and D. Zappala. RSVP: A new Resource Reservation Protocol. IEEE Network, 7(5):8–18, 1993.

[117] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patney, M. Shirazipour, R. Subrahmaniam, C. Truchan, et al. StEERING: A software-defined networking for inline service chaining. In ICNP, 2013.

[118] T. Zinner, M. Jarschel, A. Blenk, F. Wamser, and W. Kellerer. Dynamic application-aware Resource Management using Software-Defined Networking: Implementation Prospects and Challenges. In IEEE Network Operations and Management Symposium (NOMS), 2014.