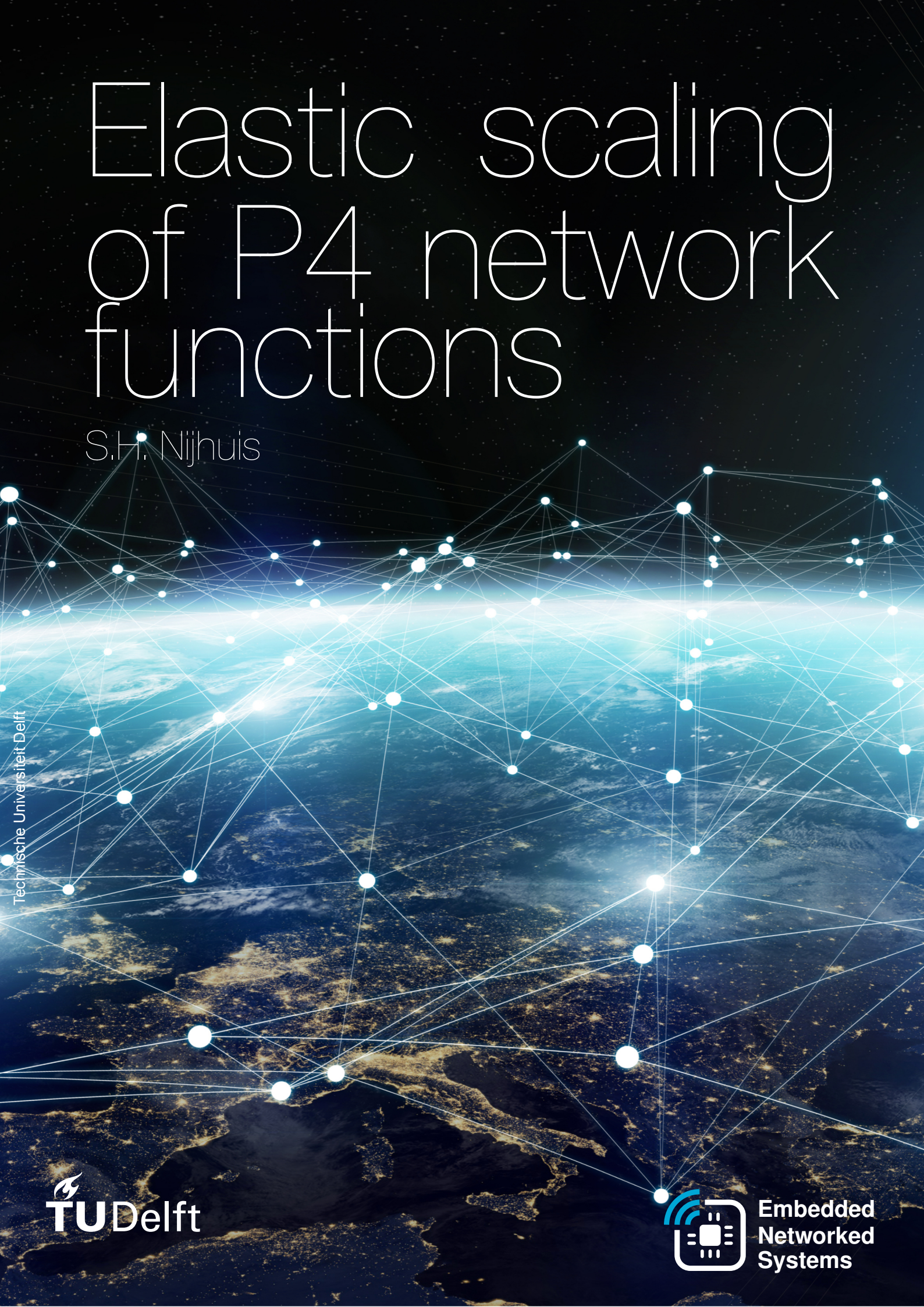
Elastic scaling of P4 network functions

S.H. Nijhuis



Technische Universiteit Delft

TU Delft

Embedded Networked Systems

Elastic scaling of P4 network functions

by

S.H. Nijhuis

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

TUD Email:	S.H.Nijhuis@student.tudelft.nl	
Email:	sjorsnijhuis@gmail.com	
Thesis date:	November 16, 2020	
Student number:	4157478	
Project duration:	June 12, 2019 – Nov 27, 2020	
Thesis committee:	Dr.Ir. Fernando A. Kuipers,	TU Delft
	Dr. Mitra Nasri,	TU Eindhoven
	Dr. Georgios Iosifidis,	TU Delft
	Belma Turkovic MSc.	TU Delft

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Embedded Networked Systems

Abstract

As recently emerged network concepts such as Network Function Virtualization (NFV) and Software-Defined Networking (SDN) promise to bring more flexibility to existing networks they also pave the road for the development of a new type of service. These novel services are known as mission-critical services which generally have tight latency and jitter restrictions. Some examples of this type of service are could gaming, cloud-connected virtual reality or remote surgery.

To avoid exceeding the tight tolerances set by these services, Network Functions (NFs) that are necessary for the network connection (e.g. firewalls), should benefit from both hardware acceleration for increased performance and scaling to reduce the hardware footprint and introduce more network flexibility. In this thesis, a novel, horizontal scaling solution for Virtualized Network Functions (VNFs) was designed and implemented in P4.

In particular, we propose a novel elastic scaling solution for hardware-accelerated switches. Our solution consists of three parts: flow migration, monitoring, and decision-making. Flow migration, the part where flows are migrated to another switch, is performed in three phases. First, the migration source migrates the state to the migration destination. Then, the migration source updates the NF states as they change on the migration source. When the controller completes its tasks, flow packets are forwarded to the migration destination and the controller can divert the traffic to the migration destination directly.

Our decision-making algorithm uses VNF, switch, and individual flow usage statistics to decide where to scale to and - if applicable - where to place a new NF instance. To reduce the monitoring overhead, only usage statistics for high-rate flows are gathered and overloads are detected within the switch. Our decision-making algorithm is able to spawn new NF instances when an overload occurs or redistribute load among already existing NF instances to optimize NF resource usage.

Results show that our algorithm reduces migration times by $0.52s$ on average while the average latency observed by the flow is reduced by approximately $5ms$ on average when compared to current state-of-the-art. Furthermore, our solution does not overflow the controller as NF states are communicated directly between switches which are equipped with links that are optimized for high data volumes.

Glossary

Control plane The part of the network that executes topology-related task (e.g. routing protocols). 3, 4, 9, 12–22, 24, 25, 27–30, 32, 33, 37, 39, 40

Data plane Packet-processing part of networks. Usually equipped with high-rate, low-latency links, which are optimized for data throughput. 2, 3, 5, 9–15, 17–22, 24, 28–30, 33–35, 37, 40, 41

Digest Short P4 message that is used to export information (e.g. header fields or metadata) from the switch to the control plane. 22

Field-Programmable Gate Array Type of reprogrammable hardware. 2

Flow A series of packets that can be distinguished based on several header fields. In most networks, flows are distinguished by source/destination IP address, TCP/UDP port, and IP protocol. iii, 3–5, 7–25, 27–35, 37, 39–41

Google Remote Procedure Call Remote Procedure Call (RPC) implementation, originally created by Google. Used in the P4 simple_switch_grpc software switch to communicate between the control- and data plane. 21, 27

High-Availability Systems in a redundant setup, so that single failure does not lead to service interruption. 12

Intrusion Detection System Network Function (NF) that monitors systems or networks for unauthorized or malicious traffic. 1

Network Address Translation Technique to change address information in network traffic. 21

Network Function Applications which add functionality to a network, such as firewalls, Intrusion Detection Systems (IDS), or network-address-translation (NAT). iii, v, 1, 7, 11, 13, 21, 27, 37, 39

Network Function Virtualization Technique to virtualize network functions, instead of using physical hardware. iii, 1, 7, 11

Programming Protocol-Independent Packet Processors High-level programming language, used to program network nodes. Commonly abbreviated to P4. iii, 2–5, 7, 9–11, 13, 17, 18, 21, 22, 24, 27, 29, 30, 39–41

Remote Procedure Calls Inter-process communication, which is used to execute code in different address spaces (usually computers connected to a network). vi, 21

Service Function Chain Sequence of network functions (NFs), applied on packets from a single flow. 4, 7, 20, 21, 39

Software-Defined Networking Novel networking technique which implements a packet-forwarding layer (data plane), controlled by a central controller (control plane). iii, 1, 7, 14, 22, 30

Switch Hardware network devices in networks. Programmable switches can execute network functions (NFs). iii, 2–5, 7–25, 27–35, 37, 39–41

Thrift Remote Procedure Call Remote Procedure Calls (RPC) implementation by Apache. Used in the P4 simple_switch software switch to communicate between the control- and data planes. 21, 27

Virtualized Network Function Network function (NF) implemented in software. iii, 1, 7, 11, 19, 27, 39

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Problem definition . . . 3

1.3 Scope . . . 4

1.4 Contributions . . . 4

1.5 Thesis outline . . . 5

2 Background **7**

2.1 Network Function Virtualization (NFV) . . . 7

2.2 Software-defined Networking (SDN) . . . 9

2.3 Programming Protocol-Independent Packet Processors (P4) . . . 9

2.3.1 P4 program states . . . 10

3 Related Work **11**

4 Architecture **13**

4.1 Scaling process . . . 13

4.2 Architecture components . . . 14

4.3 Monitoring . . . 15

4.4 Decision-making . . . 17

4.5 Flow migration . . . 17

4.5.1 State migration . . . 18

4.5.2 Initialization phase . . . 19

4.5.3 Update phase . . . 20

4.5.4 Forward phase . . . 20

5 Implementation **21**

5.1 Monitoring . . . 22

5.2 Decision-making process . . . 23

5.3 Flow & state migration . . . 24

6 Experiments **27**

6.1 Experiment setup . . . 27

6.2 Experiment topology 28
6.3 Performance metrics 28
6.4 Traffic pattern 29
6.5 Experiment scenarios 29
6.6 Experiment results 29
6.6.1 Approach without scaling algorithms 31
6.6.2 Controller-based approach 32
6.6.3 Swing state data plane approach 33
6.6.4 Data plane scaling approach 34
6.7 Comparison 34
7 Result discussion **37**
8 Conclusion **39**
8.1 Future work 40
A Appendices **43**
Bibliography **49**

1 Introduction

New concepts in networking such as Network Function Virtualization (NFV) [1] and Software-Defined Networking (SDN) [2] have emerged which, when combined, promise to introduce more flexibility, reconfigurability, and agility needed by many new services in novel network techniques, such as 5G [3].

Many of these services, e.g. cloud gaming [4], cloud-connected virtual reality [5], or tactile internet applications [3] (e.g. sending the sense of touch over the internet) have tight latency and jitter tolerances. Exceeding those tolerances can lead to motion sickness [6], a reduced experience, or unplayable games [7]. Moreover, with the introduction of 5G [8], it is expected that new mission-critical services such as autonomous vehicles [9], smart grids [10], and remote surgery [11] will become more common and make low-latency network connections vital.

This introduces many new challenges, as today's networks are not equipped for these types of new services. In order to create reliable low-latency network services, infrastructures need to be able to adapt to the tight tolerances new services require. This requires a high-performing and flexible environment, one that does not exist today.

1.1. Motivation

Traditionally, Network Functions (NFs) such as a firewall or Intrusion Detection System (IDS) are implemented in hardware without any form of virtualization (middleboxes). While this enables manufacturers to guarantee a specific performance or throughput, an NF's performance cannot be adjusted while in operation. This has the disadvantage that if the NF is not able to process all load, the network operator is left with only one option: buying new hardware with higher capacity. As network traffic is generally unpredictable (e.g. time of day or seasonal variations, or a general increase in requests for the service), hardware NFs are generally overdimensioned to prevent unnecessary hardware replacements. This is inefficient since it causes the middleboxes to waste resources.

Middleboxes generally use hardware acceleration to increase performance and reduce response latency [12]. This hardware is specifically designed or tuned for the NF, making it inefficient or even impossible to run other NFs than the hardware was designed for.

NFV promises to move network functions from dedicated hardware to commodity servers [1], offering more flexibility, elasticity, and easier management. Since Virtualized Network Functions (VNFs) are implemented in software new instances can easily be deployed when the current NF is overloaded. In

contrast to replacing a traditional middlebox, elastic scaling is executed on-demand and hardware can be used by multiple NFs.

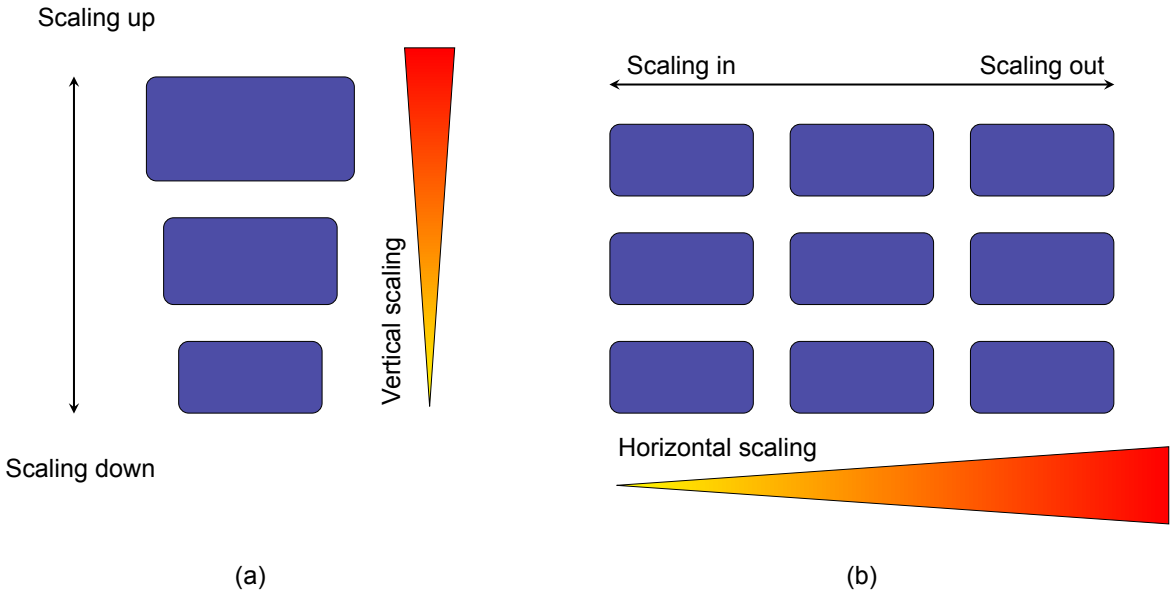


Figure 1.1: Vertical (a) and horizontal scaling (b).

Elasticity, i.e. scaling an application according to its current load, can help hardware-accelerated NFs become even more flexible. By allowing them to scale, hardware resources could be further optimized, as NFs only use hardware resources they currently need. As figure 1.1 shows, scaling can be done either vertically or horizontally. Scaling vertical involves expanding or decreasing resources (e.g. CPU, memory, or network links) of a single instance based on the load the NF experiences. When scaling horizontally, the number of instances can be increased or decreased. The total NF load is divided amongst several NF instances.

Vertical scaling is possible on only a single machine. This means that an upper scaling limit exists. When all resources on that machine are assigned to an NF (e.g. CPU, memory, or network links), an NF is unable to scale up further. With horizontal scaling, this limit does not exist. However, scaling horizontally creates several issues.

Since NFV uses commodity hardware, hardware acceleration is unavailable or requires special add-in cards tailored for usage with only one or a few NFs. Efforts have been made in order to offload parts to acceleration processors to reduce the processing time of NFV workloads [13, 14]. These solutions need to be tailored to the application since specific parts of the NF workload have to be reprogrammed to be hardware accelerated. Programming Protocol-Independent Packet Processors (P4) [15] enables many NFs to use hardware acceleration by offering a fully programmable SDN data plane that supports multiple targets that offer hardware acceleration, such as Field-Programmable Gate Arrays (FPGAs), ASICs or programmable switches [16].

P4, a high-level programming language for creating a programmable data plane, which allows network operators to implement NFs directly into the data plane. P4 supports multiple hardware-accelerated targets (e.g. FPGAs), and allows to run multiple NFs on a single device, enabling more flexible usage and clearing the way for elasticity in hardware-accelerated platforms.

Many NFs require information about previous traffic to function. These NFs are called stateful NFs. When scaling horizontally, state information (states) NFs must be migrated with the NF. Simply copying these states once is not a feasible solution since it leads to corrupt states as during state transmission the state is updated at the migration's source. Therefore, a process needs to be in place to correctly transfer and update state information during the migration process.

Usually, the link speeds of the control plane are lower and latencies are generally higher compared to data plane links, which are optimized for data transmission. Especially with modern data plane speeds reaching Tbps [16], migration traffic rates could easily overflow the control plane. Stateful NFs, (e.g. a packet counter, which counts how many packets have arrived for each flow) could update their states at line rate, as their internal state changes for each received packet. Furthermore, NFs can potentially require hundreds of states per flow to function [17]. Migrating such an amount of states requires many data to be transmitted, which would easily overflow the control plane.

On top of that, latencies between the data plane and the control plane are usually high in SDNs [18], as the control plane is not running on network devices themselves, but is separated into a single controller. Therefore, state updates have a high transmission latency between a migration's source and destination, complicating the scaling process as many states are in the process of being updated.

Multiple NF state migration solutions exist [19–27]. These solutions either use a central controller to forward state information or use high-level abstractions, such as virtual machine (VM) replication or external databases, that are not available in P4. Moreover, these implementations do not address the problem where various switches use different implementations for hash calculations, leading to corrupt states as states could be written over another flow's states. Therefore, these algorithms cannot be used when dealing with hardware-accelerated NFs.

Swing State [28] addresses this problem. This approach uses packets of the flow to carry state information to a migration destination switch, which is then able to calculate its internal hash, to store the NF state in the correct position. However, *Swing State*'s performance heavily depends on the incoming traffic pattern. Along with other problems (see section 3), *Swing State* is not well suited for transmitting NF states within the data plane.

1.2. Problem definition

In this report, we investigate ways to introduce elastic scaling to P4 VNFs. By implementing a general scaling solution in P4, NFs could benefit from scaling and hardware acceleration simultaneously. Many NFs can be implemented in P4 to let them profit from hardware acceleration, without requiring them to run on vendor-specific hardware. As horizontal scaling does not have the single-system upper limit vertical scaling has, the solution uses horizontal scaling.

In order to implement horizontal scaling, several problems need to be solved. First, as many NFs require states to function correctly, these states have to be migrated. Using the control plane for this is infeasible, as the control plane can be easily overflowed, states can be generated faster than they can be migrated, and high latencies exist between the control plane and the control plane in SDNs.

Furthermore, when a switch is overloading, a new NF instance must be placed on another switch. As not individual NF loads, but the combined load of all NFs running on a switch causes overloads, a decision-making process will determine for which NF an action needs to take place and where a new instance will be placed. This process requires recent load information from flows, NF (instances) and switches.

Each flow needs to be processed by the NF instance that has the flow's state information. Several NFV scaling approaches [22, 23] introduce global states (see chapter 3). However, this is not feasible for P4 NFs, as many states are not incremental (meaning they cannot use local incremental updates to create a shared global state), or latencies to retrieve states from external storage would be too high, as this would congest the network. Therefore, states must be migrated with the flows, without corrupting state information. During migration, flows must continue as normal, avoiding packet loss or jitter.

This leads us to the following research question:

How can we achieve elasticity of hardware-accelerated VNFs?

In order to fully answer this question, several sub questions need to be answered:

1. *How can we efficiently and reliably detect overloads in P4 switches?*
2. *What metrics can we use to decide where to redistribute load to?*
3. *How can we migrate flows while maintaining states for stateful NFs without interrupting flows or overflowing the control plane with only a small overhead and as quickly as possible?*

This solution requires to detect overloads, decide which flows to migrate to which instance, and to migrate the flows. Flows for stateful NFs require state data, which has to be migrated with the flow. Migrating flows should not overload the control plane, and flows must continue as normal, avoiding packet loss or jitter during migration.

1.3. Scope

Topics such as VNF and Service Function Chain (SFC) placement, have been thoroughly researched [29–38]. Therefore, this research will not take initial placement of VNFs or SFCs into account but provides solutions for placing new instances in situations where a switch in a previously placed SFC experiences overloads.

Solutions that can migrate stateful NFs are also able to migrate stateless NFs. Since stateful NF migration is the more difficult of the two, this report will focus on stateful NFs. NFs that

Hardware-accelerated devices generally do not allow to assign hardware resources to specific programs, due to their design. Therefore, horizontal scaling is limited by the number of available devices. Normally, when scaling to many instances, the communication overhead on horizontal scaling can become so high that adding more instances does not improve performance. However, this limit will be much higher than what is considered in this thesis. As each instance functions independently overhead consists of installing rules and monitoring, which would already occur in situations where NFs are not scaled horizontally. Overhead due to scaling mostly consists out of traffic between the migration's source and destination. Therefore, this thesis will only consider horizontal scaling. As horizontal scaling does not have the single-system limit vertical scaling has, it is a more flexible approach.

Not all states need to be transferred. Some states depend on local information (e.g. timestamps). These states will be recovered almost instantly as they can be derived from traffic. Other states cannot be transmitted one-to-one, as they depend on network topology. These states require transfer functions, which is not considered in this thesis.

Currently, no implementation exists that can instantaneously reload P4 software without losing internal program states or installed control plane rules. Research has proven [39–43] that instantaneous, rule preserving P4 software reloads are possible, but no implementation that does not suffer from heavy performance loss is available. Therefore, this research will assume it is possible to instantaneously reload P4 software, while preserving the state. This will be modeled by adding all NFs in the switch's code. Each NF can be enabled or disabled by the control plane by installing a table rule.

1.4. Contributions

Our main contributions are as follows:

- We propose a novel elastic scaling solution for P4 NFs, which changes flow routes on scaling

events and executes a novel data plane state migration protocol, which is able to consistently migrate and update states and is resistant against out-of-order packets.

- We give an approach for efficiently detecting overloads from the data plane, which reduces overload detection latency and monitoring overhead.
- To decrease the overhead that comes with monitoring individual flows, a novel approach is presented.
- We will present metrics to decide which flows to migrate to which switches.

Our elastic scaling solution for P4 NFs will redistribute flows amongst existing instances if possible, and only scales out when necessary. A decision-making process decides if and where to place or remove instances, based on the current flow paths, network layout, and usage information from NF instances, switches, and flows. This process will decide when to scale and where to place a new NF instance. This decision-making process will also decide when to scale in, so freed hardware resources can then be used for other NFs.

1.5. Thesis outline

Chapter 2 will provide an explanation of several key concepts used for our novel elastic scaling solution. This chapter will be followed by a summary of related work (chapter 3).

In chapter 4 the solution's design will be explained. First, the components will be explained. Switch, NF, and flow information will be gathered (section 4.3) for use in a decision-making protocol (section 4.5).

The decision-making protocol is activated when overloads are detected and is able to either rebalance NF load within existing NF instances. When all existing NF instances cannot handle the extra load, a new instance is spawned.

When a solution to overcome the overload has been calculated by the decision-making protocol, the flow migration algorithm is executed. This algorithm modifies flow paths (section 4.5), and migrates NF states within the data plane (section 4.5.1) in several phases, to prevent lost or corrupted states.

Chapter 5 explains how the three components are implemented. This implementation is used to run the experiments on (chapter 6), in which our implementation will be compared to a similar, controller-based migration approach, and an approach which is not able to scale in or out. These findings will be discussed in chapter 7. Chapter 8 gives conclusions and recommendations for future research to focus on.

2

Background

In order to solve the problems mentioned in chapter 1, knowledge about several concepts is required, which includes Network Function Virtualization (NFV) (Sec. 2.1), Software-Defined Networking (SDN) (Sec. 2.2), and P4 (Sec. 2.3).

2.1. Network Function Virtualization (NFV)

Traditional middleboxes have the disadvantage that each Network Function (NF) has its own dedicated hardware box. Furthermore, if the NF should be upgraded due to an increase in NF requests, new hardware or licenses should be acquired. Since middleboxes are dedicated hardware appliances, they are unable to scale. By virtualizing NFs, the hardware can be used by multiple NFs. This allows for adding the ability to scale to NFs, assuming hardware to scale out to is available.

NFV is the technique of moving NFs (network functions), such as load balancing or caching, from traditional middleboxes to commodity services offering lower equipment costs, more flexibility, elasticity and easier management [44, 45]. This makes the network more flexible than traditional network solutions and reduces network costs [46]. Virtualized network functions will be referred to as Virtualized Network Functions (VNFs). Within networks, each flow can - depending on the network and type of flow - be processed by one or more NFs.

Very often the order in which NFs are applied on a packet is important, since most NFs operate in OSI layer 2/3 [47] and change packet information. Therefore, NFs often depend on each other and need to be executed in a specific order. This sequence of NFs will be referred to as the Service Function Chain (SFC). The SFC can consist of two or more VNFs and is defined on a per-flow basis. These NFs are not necessarily directly connected; a path between NFs could span multiple network hops, as shown by figure 2.1. If, for example, an extra link would be added between switch 2 and switch 4, the network path would change, whereas the SFC path would remain equal.

Network Function (NF) states

VNFs that don't require state information are called stateless. They only need information from the packet itself to function. This can be the case in a simple proxy VNF. The simple proxy works by forwarding incoming packets to a predefined destination and can therefore function without any data dependencies.

Most NFs require information about previous traffic to function. These NFs are called stateful NFs. For

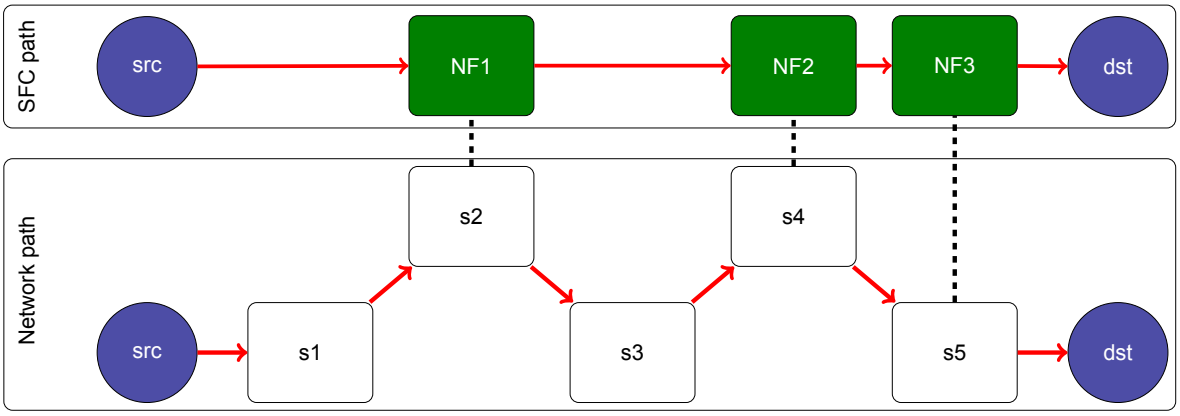


Figure 2.1: Relation between an SFC path and network path. The network path is shown below, connecting source and destination hosts. Several NFs are running on switches and process packets of the flow from *src* to *dst*. The network path consists out of switch-switch hops, whereas the SFC path consists out of NF-NF hops. A single SFC hop could span multiple network hops.

example, firewalls, where information from previously processed packets belonging to the same flow is used by the VNF to make decisions about incoming packets. NFs have global states (e.g. states used by each flow) and per-flow states.

Some of these states are incremental (e.g. packet counters). This is the case with for example the TCP protocol. TCP has multiple states (e.g. ESTABLISHED, or CLOSING), which indicate the state of a TCP connection. Firewalls can use this information to pass or block certain packets. These states are not incremental, as the numbers are a mere mapping to store connection states more efficiently.

Other NFs use states that are incremental. For example a packet counter NF, which counts how many packets (mostly for each flow) arrive at the switch. It keeps a counter with the amount of packets that arrived into the switch and updates this counter on each new packet that arrives.

State migration

Regarding migration, NF states can be distinguished into two types: *hard* and *soft* states [28]. *Hard* states are derived solely from packet information that can be recalculated by any other switch in the network, such as *source IP, destination IP*, or *destination port*.

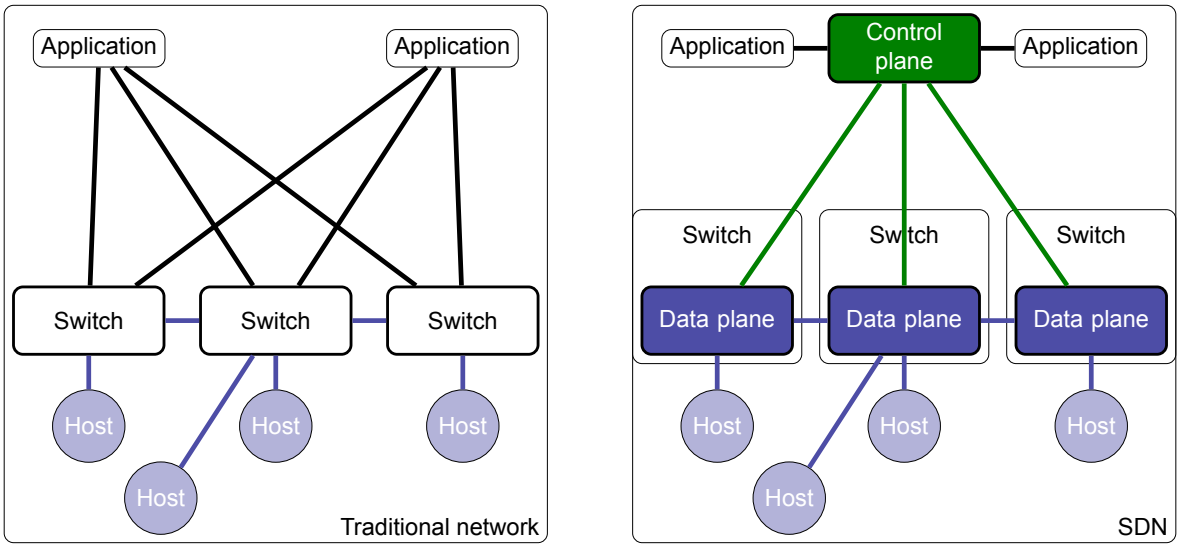
Soft states use switch-dependent information to calculate state keys, such as *ingress port* or *ingress timestamp*. State keys derived from these kind of metadata fields would not be recalculable by other switches, since it would be highly probable this metadata information differs in other switches. According to [28], *soft* states are typically used for optimization purposes and don't need to be migrated.

Some *hard* states are dependent on e.g. network topology [28]. For example a Network Access Control (NAC) solution, which only allows traffic from specific MAC addresses on specific ports. When migrated, these states would have to be migrated for the VNF to function correctly. Just migrating them would not necessarily lead to a correct function at the migrations destination, since the MAC addresses could be connected to another port. Since most of these states could be considered *soft* states, since they rely on switch-dependent information these states are ignored. As flow are migrated for stateful NFs, program states need to be merged (scaling in, where states for a flow need to be imported), or replicated (scaling out, where a flow's states need to be replicated).

2.2. Software-defined Networking (SDN)

Traditional networks consist of routers, switches, firewalls, and many more devices, where each device runs on its own hardware and has a fixed function. No single device has a clear overview of the whole network. Therefore, algorithms on the network (e.g. routing algorithms, such as Open Shortest Path First (OSPF) or Border Gateway Protocol (BGP)) are not centrally managed, which means that they can only make local choices. This behaviour is called the decentralized control plane.

SDN changes this behavior, as shown in figure 2.2. SDN centralizes the decentralized control plane into a single control plane application. Since this control plane application has a central overview of the network, this allows for more flexible and efficient behavior, for example for routing applications. Traditional protocols, such as OSPF or BGP could be integrated in the SDN, but also any other algorithm could be easily implemented into the control plane.



(a) (b)

Figure 2.2: Difference between traditional network (a) and SDN (b). The SDN distinguishes a data plane, where packets are switched, and a control plane, with global knowledge about the network. Applications that require global network information, can be connected to the control plane, whereas they would need to address each individual switch in the traditional network.

The layer where data packets are being handled is called the data plane. Data plane behaviour is determined by the control plane. In most SDN protocols, like OpenFlow [48], the data plane behavior is fixed, and only the behavior of the control plane can be programmed. The control plane requires several supporting functions, such as link-, host- and switch discovery. Furthermore, packet routing or forwarding functions are deployed to enable packets to flow efficiently over the network. SDN can use various routing principles. One is the classical approach of IPv4 networking.

2.3. Programming Protocol-Independent Packet Processors (P4)

SDN enables a fully programmable control plane. Continuing this trend, programming languages for programming the data planes [49], such as P4 [15] and Protocol-oblivious Forwarding (POF) [50] were proposed. P4 enables programmers to fully determine a switch's behaviour, since the whole packet processing pipeline is programmable.

Custom and new packet headers could be used without waiting for support from vendors. For instance VXLAN is a relatively new protocol where many devices do not yet have support for. Using P4 VXLAN's header and protocol support could be implemented very quickly.

Packet processing VNFs, such as a packet counter, firewall, or proxy application, could be implemented directly in P4 [51]. Running VNFs directly on switches gives them the advantage of hardware acceleration. Furthermore, P4 implementations are target independent. P4 can be compiled to several architectures, such as ASICs, FPGAs, Network Processor Units (NPUs), and general-purpose CPUs. These architectures are called targets. P4 programs can also be deployed to various dedicated platforms, such as Intel® Tofino™ 2 [16] switches . This makes P4 a very general solution for programmable data plane programming languages.

2.3.1. P4 program states

P4 VNFs can be stateful (section 2.1). This means that decisions the P4 programs makes depend on previous packets that have arrived earlier to the switch. A simple example of a stateful NF that can easily be implemented in P4 is a packet counter. This packet counter can be used to detect heavy-hitters: flows with a rate higher than some threshold. This information can be used to block certain flows or perform rate-limiting on heavy-hitters. If these states would be corrupted, heavy hitters would not be detected, and the NF would not be functional.

P4 stores state values in registers, which are essentially a key-value storage. Keys are usually derived from packet information by using some hash calculation on data that distinguishes packet flows from each other. Usually, IP source and destination addresses, IP source and destination ports, and the IP protocol are used for this hash calculation. Since switches can have different hash algorithm implementations, depending on the hardware capabilities, register keys for a single flow vary between switches.

3

Related Work

Network Function Virtualization (NFV) scaling and migration has been subject to many researches, such as [22, 24, 25, 52–63]. These implementations typically use virtual machine (VM) migration (e.g. migrating the current state of an Network Function (NF)). These hypervisor functions will not be available in P4 data planes, as these approaches use high-level abstractions, which are not available in high-speed data plane NFs by design.

Some [64, 65] do not migrate state information, but use load balancing to distribute traffic over instances, by assigning incoming flows to specific instances. This gives NFs the ability to elastically scale out, but only lets them scale down when an instance is fully idle (i.e. no flows are processed on that specific NF instance). Furthermore, this approach is not able to handle overloads if existing flows (i.e. flows that are already assigned to an instance) start overloading an instance. As flows are not migrated, but only assigned to instances when they first arrive, the load cannot be reduced at an overloading instance.

Others do consider both local and shared states. FreeFlow [54] temporarily suspends flows while updating global states. This will lead to high flow latencies for flows with many shared states, as the flow needs to be paused while transmitting states. Moreover, hardware accelerated devices cannot temporarily pause flows. With data rates reaching Tbps [16], buffers must be impossibly large to accommodate this temporary buffering.

LODGE [23] proposes a global state model for distributed NFs, which use local states and receive global updates from other Virtualized Network Function (VNF) instances, essentially creating distributed states on multiple switches. This approach works by leveraging multicast groups to propagate individual state updates to create a distributed state, a state which exists on each instance. While this could be used to migrate state information from a transmitter to a receiver, LODGE is only able to create global incremental states (states that incrementally increase/decrease each packet). As many NFs require states that are not incremental (e.g. TCP protocol states, which a firewall uses to decide whether to block certain packets), LODGE is unsuitable for our application, as general NFs should be able to use and migrate non incremental states.

StatelessNF [22] only considers global states, which are migrated to an external state data storage. This abstraction can not be made in P4 data planes. P4 is not able to queue packets awaiting the switch to retrieve any non-local states. Therefore, retrieving external state information would impose a significant performance impact on NFs, which would make retrieving remote or distributed states infeasible for P4 VNFs.

A multitude of controller driven migration schemes exist [25–27]. For example Gember et al. [24] pro-

posed a state migration scheme using the control plane. Controller based approaches are generally infeasible as the control plane is usually much slower than the data plane. Since states can be updated at line rate, and NFs may keep hundreds of states per flow, this would easily flood the control plane. Furthermore, various implementations might use different hash implementations. Simply copying information and putting it in the other device's database at the same index will not directly lead to a correct state interpretation: it could even corrupt states, as it is possible that states on the receiver overwrite another flow's states already present in the register array. Furthermore, as the switch processes NF packets, the state will not be used, as it is located at the wrong memory position.

Non-controller driven state updates are widely researched, mostly for High-Availability (HA) purposes [19–21]. Many NFs require some form of HA, where a backup device could take over a failing device without losing state information. This information is usually propagated via a direct link, without the use of any controllers. Pico Replication (PR) [20] implements this by monitoring middleboxes for state updates and transmits them if changed, do not use the control plane for state transmission. However, PR does not address the problem of incorrect states, as it was meant as a HA state synchronisation solution and therefore it could be assumed that similar hardware was used in a similar network configuration.

Luo et al. [28] addressed this problem and proposed Swing State, a data plane state migration algorithm. The main novelty of Swing State is that data packets are used to migrate state information within the data plane, in order to let migration destinations calculate their own internal hashes. To do so, Swing State appends a new migration header to the packet, which will carry updated state values to the migration destination. While Swing State addresses differences in hash calculations in various switch implementations and avoids using the control plane for migration, it heavily depends on the pattern of incoming traffic. States will not be migrated until traffic arrives.

As both the state receiver and transmitter need to be in certain states, a protocol state mismatch at the migration source and destination could lead to incorrect output behaviour (either forwarding a packet twice to its destination or no forwarding at all). Using full data packets for state transmission causes Swing State to have a large overhead, which could be reduced by using smaller update packets. Lastly, Swing State cannot handle out-of-order packets or packet loss, as states could be overwritten by older information.

4

Architecture

Our novel approach enables Network Functions (NFs) to scale horizontally over several switches. Two processes are considered, namely scaling in and scaling out. Our novel elastic scaling solution for P4 NFs activates when a switch overload is detected. Our overload detection exploits the programmability of P4 switches to enable fast detection of overloads (while the packets are processed on the switch), triggering a scaling-in/out and state transfer directly in the data plane. This makes our approach ideally suitable for programmable data planes operating at Tbps.

In order to use hardware more efficiently, an algorithm periodically determines whether NFs could be scaled in to eliminate unnecessary, almost idle NF instances. These freed hardware resources can be reused by other NFs, for example when another switch detects an overload or when a new NF is deployed to the network.

A custom decision-making process has been implemented to detect appropriate mitigations for scaling in/out. This process uses monitoring information from switches, NFs, and individual flow to base decisions on. Two situations can occur when overloads are detected:

1. Only migrate flows (to an already running instance)
2. Deploy an NF and migrate flows to this new instance

The process first checks if NF instances exist which could take over a portion of the load (1) that has to be migrated away from the overloading switch. This has the advantage that hardware resources (where NFs are running on) will be used more efficiently, as less hardware resources are used for NFs. Scaling (2) only occurs if all NF instances cannot be migrated to existing NFs. In that case, the only option to overcome these overloads without interrupting flows is allocating new resources to the NF.

4.1. Scaling process

The decision-making process generates an action to mitigate overloads or removing NF instances to free hardware resources. After this, the flow migration process is started. Flows are migrated in multiple steps, to ensure that no packets or states are lost or corrupted by the migration. Phase one (initialization phase), consists of transferring NF states to the migration destination switch, after which phase two (update phase) is started. In phase two, state updates are sent to the migration destination if state changes occur at the migration source switch, called *stateUpdates*. Meanwhile, the control plane

starts installing new traffic rules, without disturbing the current packet flow. After the initial state has been received and the path installation has been completed, the process enters phase 3: the forward phase.

In this phase, the migration source only forwards packets to the migration destinations, which processes them and forwards them to their destination. The control plane changes the flow's path in such a way that traffic is sent directly to the migration destination. Any leftover packets will still be forwarded to the migration destination.

4.2. Architecture components

This chapter outlines the general design of our solutions and its components. Our elastic scaling solution exists of three main components:

1. A monitoring component (Sec. 4.3). The novelty of this component is that only relevant (high-rate) flow statistics are gathered, to reduce monitoring overhead, while all switch and NF statistics are retrieved.
2. A decision-making component (Sec. 4.4), which generates responses on overload situations and is able to free hardware resources when the situation allows to reduce the number of NF instances.
3. Flow migration (Sec. 4.5), which transfers NF states within the data plane.

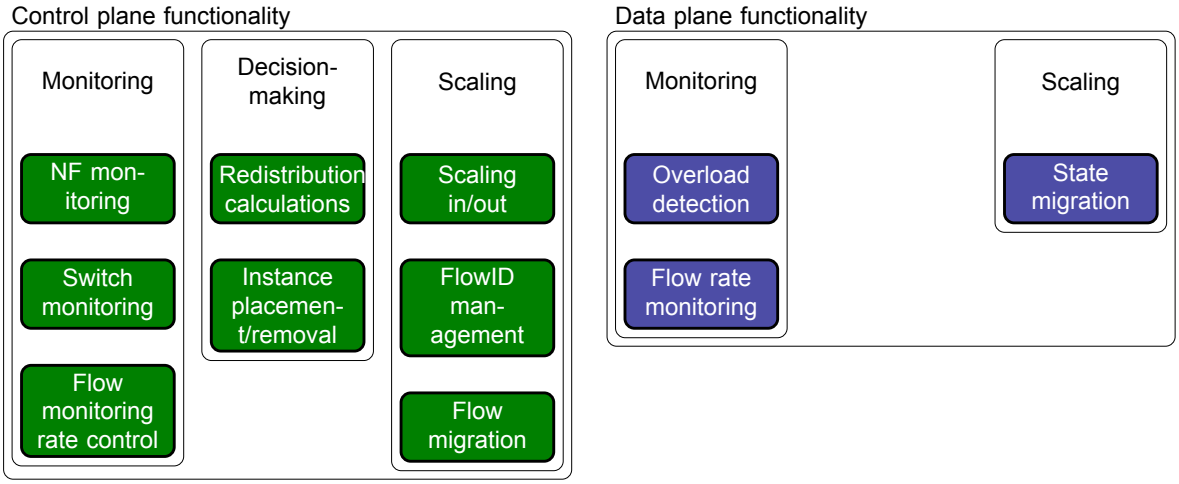


Figure 4.1: Control plane (green) and data plane (blue) architecture components, separated in three categories. Components that perform monitoring tasks send information to the control plane. This information is used by the decision-making algorithm, which calculates mitigations for overloads and optimizes hardware usage by reducing the number of NF instances if possible. Several scaling components initiate and control the flow migration procedure.

Figure 4.1 shows which functionality is implemented in the control- or data plane. The control plane is well suited for tasks that require a global network overview (e.g. links, switches, usage statistics). Typical Software-Defined Networking (SDN) frameworks already deploy several tasks, such as link or switch detection and routing algorithms. Our novel elastic scaling solution extends this functionality with monitoring (switches, NFs, or flows), flow management, NF, or instance placement tasks.

Tasks that require low latency and large bandwidth are executed in the data plane. To reduce monitoring overhead and overload detection latencies, overload detection and flow rate monitoring is performed on the switch, instead of using a polling-based procedure initiated from the control plane, where the

control plane regularly sends a message to the switch to retrieve information. To prevent overloading the control plane on NF state migration, state migration is executed in the data plane. Since the data plane is equipped with high-rate and low-latency links, optimized for data transmission, the data plane is an excellent network to communicate NF states quickly and reliably.

4.3. Monitoring

Our solution needs several monitoring data. Usage statistics from switches, NFs, and individual flows are considered by the decision-making process to detect valid scaling options. However, monitoring individual flows comes with several challenges. First, the sample interval needs to be decided upon. As internet traffic is generally unpredictable, short traffic bursts might trigger unnecessary flow migrations when sampling too often. High sampling frequencies lead to high overheads, as many data is transferred.

Sampling with a too low rate might delay overload detection, as the overload detection latency (the latency between occurrence of the overload until activation of the decision-making algorithm) depends on the sample frequency and the latency between the control plane and the data plane. This can be a problem, since a switch has limited buffers, which can fill up completely during this period, forcing the switch to start dropping packets.

Therefore, overloads must be detected quickly. To reduce both the overload detection latency and the polling overhead, our solution detects overloads on the switches. Doing so has two main advantages:

1. By eliminating the polling interval from the overload detection latency, overloads are detected more quickly, as packets are processed at the switch.
2. Polling rates can be reduced, as overloads are detected independently from the polling-based monitoring mechanism and no high-rate NF and switch usage is required.

Switches detect overload themselves by monitoring egress buffers and notify the control plane when an overload occurs. Switches report overloads when queue sizes are longer than a threshold. The threshold should be chosen low enough to allow the control plane to scale out or redistribute load before buffers are completely full and packets need to be dropped. Furthermore, the threshold should be high enough to allow some slight, occasional buffering due to short traffic bursts, without immediately triggering overloads, leading to unnecessary migration actions.

In order to gather Switch and NF instance loads, switches will be regularly queried (polled) to reply with current usage statistics. As overloads are detected by a separate process, our solution will poll switches at a low rate. This will provide the decision-making algorithm with statistics about switch, and NF instance usage, without inducing a significant load on the switches. To further reduce the overhead, switch and NF instance loads are appended to packets belonging to the link discovery process.

Another challenge is per-flow monitoring. In order to reduce a switch's load, without overloading the migration destination, our solution is able to execute per-flow migration. This requires knowledge about individual flow rates, which can be very inefficient to poll.

Two options for polling individual flow rates exist. Either each flow rate is polled individually, or one really large packet must be sent, containing all flow statistics. Polling each flow rate individually will lead to a high overhead, as each flow rate is reported individually, which imposes a significant load on the switch and control plane. Retrieving all individual flow rates in one request will require increased processing at the switch to accommodate such packets[1]. Furthermore, as registers can have millions

[1] A 10Gbit/s link can carry 14.880.952 IP packets/sec, which leads to a packet size of 15KB. For 100Gbit/s and 1Tbit/s lines the packet sizes at maximum packet rates will be 17,5KB and 19,3KB, respectively.

of rows for storing data, retrieving this data will be quite ineffective: in most cases these registers are sparsely populated, meaning many irrelevant data is transmitted. Therefore, a novel data-plane approach was chosen to report individual flow statistics.

Updating all active flows on the switch with a low rate can still easily overflow the switch. Since the decision-making process only depends on flow rates for high-rate flows, our solution only monitors high-rate flows.

Algorithm 1 Pseudo code for individual flow rate transmission.

```
n, Flow rate threshold (per-flow value)
N, Global rate threshold (per-switch value)
p, Number of received packets for flow
if p >= n then                          ▯ Update is sent every n packets
    if n >= N then                      ▯ Update is sent for rates larger than N
        Send flow update
    end if
    p = 0
end if
```

Pseudo code 1 shows how individual flow rates are reported. Each switch counts packet quantities, and reports them to the control plane each n packets. The rate is controlled by the control plane, by increasing or decreasing n, on a per-flow basis. The control plane aims to receive rate updates for flows (*flowUpdates*) approximately each second, to prevent bursts from having a too large influence on these rates, while having quite precise and recent updates without posing a significant load on the switch. Increasing the interval (e.g. to $10s$) would lead to too inaccurate flow rates, whereas decreasing the interval (e.g. to $0.1s$) leads to high overhead.

Flow rate updates are only sent if the rate is higher than N. Therefore, decreasing this value will lead to more flow rate updates and increasing this value will lead to less flow rate updates. A process in the control plane monitors if a sufficient flow volume can be migrated from the switch in case an overload of a preset percentage occurs (e.g. 10%). While the monitored flow volume is smaller than the worst-case overload volume, the value for N is decreased until the monitored flow volume is larger than the worst-case overload volume. To reduce the individual flow monitoring overload, N is increased when the monitored flow volume is larger than 1.3 times the worst-case overload, to prevent the monitored flow volume from becoming smaller than the worst-case overload volume, while still limiting the overhead.

Using this mechanism, small increases or decreases in flow rate will only slightly influence the rate of flow rate updates. However, this algorithm will not account for flows that either stop or change from high- to low rates. Instead of relying on old, and presumably incorrect information (since no flow rate update was received, one can safely assume the packet rate is lower than its last update, as a *flowUpdate* would have been received), an algorithm is designed that calculates the worst-case flow rates for flows that did not receive updates recently. As flow rates are updated approximately each second, the worst-case packet rate is calculated for flows that are not recently updated. This worst-case flow rate is equal to $\frac{n}{t_{lastupdate}}$, since the maximum amount of packets received by a switch for a flow is equal or less than *n* packets since the last flow rate update. To reduce overestimation of the worst-case packet rates, this algorithm is only used for flows that were not updated in the previous $1.5*$ *interval* seconds, as dividing by small $t_{lastupdate}$ can lead to incorrect packet rate estimations. For flows which have been updated more recently than this period, the latest received flow rate is returned.

Switches transmit *flowUpdates* for high-rate flows. By comparing a preset worst-case overload volume (e.g. 10% of the switch's capacity) to the total monitored flow volume, the control plane can increase or decrease the rate for which the switch sends *flowUpdates*.

4.4. Decision-making

A decision-making algorithm is deployed in the control plane and uses network layout and monitoring information to calculate actions (e.g. redistribution or scaling out) for overload situations, and manages flow migrations. It periodically checks whether NF instances can be removed, to optimize hardware usage and free resources for use in potential overload situations. As P4 switches lack timer-interrupt procedures, the control plane rate-controls individual flow monitoring. Furthermore, low-rate switch and NF rate monitoring is performed from the control plane. The algorithm sets the following constraints on finding a valid set of flows that it can migrate:

1. The flow volume is large enough to stop the source switch from overloading
2. The destination switch will not be overloaded by the flow volume
3. Any new switches in the flow's path will not be overloaded
4. For each flow, the flow path length will not increase more than N hops

Condition 1 ensures a sufficient solution to overcome the overhead is found by the decision-making algorithm. Conditions 2 and 3 prevent unnecessary overloads, as immediately overloading another switch while migrating a switch would be inefficient. Condition 4 prevents network congestion. Especially in large or distributed networks it can be quite inefficient to migrate an NF to another location, as bandwith is generally limited. Furthermore, each switch experiences load by only forwarding traffic. As path lengths increase, more switches need to forward packets, hence increasing loads on multiple switches. N is set to limit this and to prevent network congestion. Depending on the network topology, N can be set to a higher or lower value.

When an overload is detected, a decision-making process checks which actions need to occur in order to reduce the load for the overloading switch. First, it considers rebalancing traffic, to prevent unnecessary scaling when spare capacity is available on NF instances that are already running. If it can find an instance that meets the above conditions, flows can be migrated to an existing NF instance, eliminating the need for scaling. When no solution arises, the algorithm is run again, but now considering switches that do not currently run the NF. It is vital to keep flow paths as short as possible, as networks could get congested easily if many flow paths increase too much in an inefficient manner.

4.5. Flow migration

The flow migration algorithm is responsible for migrating flows from one NF instance to another and is only active when scaling in/out or rebalancing load. For most NFs, state data has to be migrated with the flow to prevent incorrect NF behaviour at the migration destination. Our algorithm is able to migrate flows belonging to stateless NFs as well, since they are more simple and do not require state migration. Thus, in this subsection the more complex of the two, flow migration for stateful NFs will be explained.

We developed a new algorithm able to migrate flows from stateful NFs, as current algorithms use the control plane for state migration, which is easily overflowed by the amount of packets that such migrations could generate. Furthermore, the performance of current state-of-the-art data plane state migrations depend on the pattern of incoming traffic. It is able to migrate NF states from one NF to another consistently, directly in the data-plane and changes a flow's path when states are migrated. Furthermore, our algorithm, in contrast to the state-of-the-art data plane state migration algorithms, is able to successfully migrate states and flows, even in situations where out-of-order packets and packet loss occur.

Our flow migration protocol is performed in three stages:

1. Initialization phase. NF states are migrated to the destination NF within the data plane and flow paths are changed to accommodate transmitting packets from the destination NF to the original destination of the flow.
2. Update phase, where states are updated as packets arrive at the migration's source. These updates are propagated in-data plane to the migration destination.
3. Forward phase. The migration's source only forwards packets to the migration destination, which processes them and forwards them to the original destination of the flow.

Migrations are carried out on a per-flow basis. This has multiple advantages. First, if multiple flows have to be migrated from a switch, each flow can be migrated to a different NF instance. Second, migration only has to be performed for flows that are migrated. Information for other flows, that are already active on the migration destination, is not overwritten, as states can be merged easily. This enables both easy load rebalancing and scaling out, since states from different NF instances can easily be merged, without collisions.

During the update phase, traffic rates for flows that are migrated on the migration's source could double when a worst-case NF (which changes states for each packets, triggering a *stateUpdate*) states are migrated. This causes an even higher load on the switch, and therefore increases the rate at which the buffers grow. This makes it important to keep this phase as short as possible, since a migration protocol—designed to mitigate overloads—should not force the switch to overload even more and start dropping packets.

In contrast to *Swing State*, our algorithm only requires packet cloning in the update phase. As our protocol switches to the forward phase, packets are only forwarded to the migration destination, whereas *Swing State* keeps cloning packets until the traffic is diverted. This has the main advantage that the overloading switch after completion of the update phase experiences a lower flow rate than

In order to perform in-data plane state migration, switches need to be able to exchange information directly. Therefore, the control plane programs each switch with information to reach all other switches present in the network, without requiring direct links between switches. This information is programmed when switches are first detected by the control plane, to prevent unnecessary latency during state migration, which is executed during switch overloads and therefore time critical. This makes flow updates more flexible, since state migration algorithm packets are routed independently from NF traffic, and flow route updates don't disturb switch-to-switch communication.

Since different P4 switch hardware vendors can use different hash implementations, hashes of flows do not require to be the same. This can become problematic, as it creates the possibility that states are overwritten by another flow's states, corrupting states for both flows. Our state migration algorithm addresses this by requiring each switch to calculate its own internal hash, by providing network information used for these calculations in all migration packets.

4.5.1. State migration

Our state migration process is explained in table 4.1. Flow migration is performed in three stages, namely the initialization, update and forward phase. Table 4.1 explains the state migration protocol and figure 4.2 shows the flow path updates.

Table 4.1: The state migration process. The set of states for a specific flow $\mathcal{F}$ is migrated from migration source switch $\mathcal{S}$ to migration destination switch $\mathcal{D}$. The flow's source is *srcHost*, its destination *dstHost*

Packet number	Source		Destination	Information
	Initialization phase			
1	Control plane	→	$\mathcal{S}$	Information for hash calculation. $\mathcal{S}$ calculates hash and appends state information to the packet. Control plane will start installing flow rules on $\mathcal{D}$. Control plane installs new flow path [$\mathcal{D}$, *dstHost*].
2	$\mathcal{S}$	→	$\mathcal{D}$	$\mathcal{D}$ will write the state information to its internal registers.
3	$\mathcal{D}$	→	Control plane	Acknowledge NF state reception.
	Update phase			
..	$\mathcal{S}$	→	$\mathcal{D}$	State update values until pkt 4 is received. $\mathcal{S}$ handles Virtualized Network Function (VNF) traffic.
	Forward phase			
4	Control plane	→	$\mathcal{D}$	$\mathcal{D}$ executes NF function for $\mathcal{F}$, $\mathcal{S}$ only forward packets to $\mathcal{D}$.
5	$\mathcal{D}$	→	Control plane	$\mathcal{D}$ acknowledges packet 4 to control plane Control plane changes flow path from [*srcHost*, $\mathcal{S}$] to [*srcHost*, $\mathcal{D}$]
..	$\mathcal{S}$	→	$\mathcal{D}$	Any packets for flow.
6	Control plane	→	Switches	Reroute flow $\mathcal{F}$

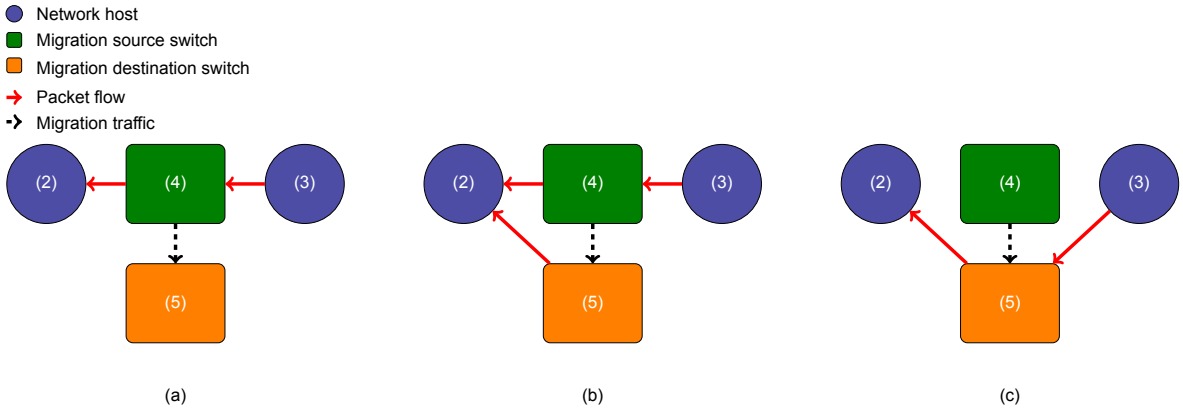


Figure 4.2: Data plane overview of the flow migration procedure. An overload is detected and a flow between (3) and (2) has to be migrated.Switch (4) performs an NF on the traffic. In the initialization phase (step a), switch (4) migrates state to switch (5). After initial state migration, an acknowledgement is be sent to the control plane. Meanwhile, switch (4) communicates state updates to switch (5), and the control plane installs a new path [5,2]. When the acknowledgement is received by the control plane, and the state reception has been acknowledged to the control plane, the process enters the forward phase (step b). The control plane indicates switch (4) to forward packets to switch (5), which processes the NF. Switch (4) acknowledges this indication to the control plane, after which the control plane installs path [(3),(5)]. Finally, (step c), the control plane removes paths [(3),(4)] and [(2),(4)], and the flow migration is be completed.

4.5.2. Initialization phase

A packet is sent by the control plane to the migration source NF $\mathcal{S}$. This packet contains all necessary information for the NF to calculate the hashes for state values that need to be migrated (e.g.

source/destination IP and port). With this information, $\mathcal{S}$ calculates hashes for flow $\mathcal{F}$ and appends its state values to the packet. NF $\mathcal{S}$ then forwards this packet to the migration destination NF $\mathcal{D}$. NF $\mathcal{S}$ enters the forward phase: all subsequent state updates are forwarded to NF $\mathcal{D}$. $\mathcal{D}$ processes packets received from $\mathcal{S}$ and informs the control plane it has received state information for flow $\mathcal{F}$. NF $\mathcal{D}$ enters the forward phase as well and starts accepting state updates for $\mathcal{F}$.

Furthermore, after sending the packet to source NF $\mathcal{S}$, the control plane starts installing a path from $\mathcal{D}$ to the next Service Function Chain (SFC) (either *dstHost* or another NF) path hop after $\mathcal{S}$. Figure 4.2 shows this in (b), where path [(5),(2)] is installed by the control plane.

4.5.3. Update phase

After completing the initialization phase, flow $\mathcal{F}$ still follows its original path and packets are still processed by NF $\mathcal{S}$, which is sending state updates to $\mathcal{D}$. In contrast to the state-of-the-art solutions, such as *Swing State*, our solution applies sequence numbers to update packets. $\mathcal{D}$ compares this sequenceID to the sequenceID embedded in the state update it previously received. If the sequenceID of the previous state update is higher than the sequenceID embedded in the current update, a newer state value is already stored, and thus this update can be discarded. This solves the problem with out-of-order packets state-of-the-art solutions have. Adding a sequenceID to packets has the advantage that NF $\mathcal{D}$ will only overwrite state with newer information, even if older updates arrive later at $\mathcal{D}$ than newer ones. Since no incremental states are transmitted, any migration traffic packet loss is overwritten when newer packets arrive at $\mathcal{D}$. When the control plane has received the NF state reception acknowledgement from $\mathcal{D}$, the control plane will signal the start of the forward phase.

If for some reason this acknowledgement is lost (e.g. packet loss or corruption), it will be resent. Switches do this by checking sequenceIDs. Each N state updates, it will resend the state reception acknowledgement. The value for N is network-specific and depends on the delay between the control plane and data plane and the flow rate; acknowledgements shouldn't be sent too quickly, but lost acknowledgements shouldn't prevent a successful migration.

4.5.4. Forward phase

If signalled by the control plane, NF $\mathcal{S}$ will stop processing packets for flow $\mathcal{F}$ and only forward these packets to $\mathcal{D}$, which will then process packets from $\mathcal{F}$. Extra header information informs $\mathcal{D}$ that $\mathcal{S}$ did not process a packet, and $\mathcal{D}$ should do so. Figure 4.2 shows this as stage (c). The path for packets from flow $\mathcal{F}$ is [(3),(4),(5),5(2)], where (5) performs the NF on packets from $\mathcal{F}$.

When the control plane has signalled $\mathcal{S}$ to only forward packets to $\mathcal{D}$, the control plane changes path [(3),(4)] to [(3),(5)], to let traffic from flow $\mathcal{F}$ directly arrive at $\mathcal{D}$. Any packets on route to $\mathcal{S}$ will still be forwarded to $\mathcal{D}$. Meanwhile, the control plane changes the flow's path so that packets arrive directly at $\mathcal{D}$. When this is completed, the control plane will remove old network rules that are installed.

5

Implementation

Basic control plane and data plane Network Functions (NFs) from the architecture, as explained in chapter 4, were implemented to compare the proposed algorithms to the current state-of-the-art. *Simple_switch_grpc* P4 targets were deployed in a Mininet environment which enables fast prototyping for network applications. These targets use the more modern and standardized *P4Runtime* [66] API, in contrast to the Thrift Remote Procedure Call (Thrift RPC) implementation which the *simple_switch* uses. These switch targets are based on the *v1model* P4 architecture. Programs written for this switch architecture are easily portable with little to no changes to other P4 architectures such as the *psa_model*, which the Intel® Tofino™ 2 [16] uses. The switches were equipped with Thrift RPC as well, to remedy limitations (see section 5.3) of the Google Remote Procedure Call (gRPC) Remote Procedure Calls (RPC) infrastructure. The control application was implemented in Python.

Mininet is an environment to run a virtualized network on a single computer. Many parameters such as switch types, network layout, and link capacity/delay can be programmed. In Mininet, several P4 switches were deployed to form a network of programmable switches.

Since most P4 NFs (e.g. Network Address Translation (NAT)) operate on OSI layer 2/3 [47], these NFs will change packet headers, rendering the packet after NF processing unrelatable to the packet before NF processing. This makes it challenging to determine a flow's full path, especially for Service Function Chains (SFCs), where a flow is sequentially processed by multiple NFs. However, the control plane should be able to only change a flow's path partially[1], since rerouting a flow's full path would lead to many flow migrations. This would frustrate NF migrations, as switch and NF load statistics would be more and more unreliable as more flows are migrated, increasing the possibility of overloading more switches.

To overcome this, it should be possible to migrate only a single NF. Especially in SFCs where multiple NFs are executed sequentially it can be efficient to only change a single NF's location, instead of migrating the whole SFC. Therefore, flows at the in- and output of NFs should be relatable to make partial route changes possible. As many NFs change a packet's network information, packets should be distinguished based on another attribute, which is not changed by NFs. Therefore, each flow is assigned a flowID by the control plane. The first switch in the flow's network path will append this flowID, the last switch in the path will remove the flowID. This will ensure the flowID is consistent on each switch and single NFs can be changed in the SFC without the necessity to migrate other NFs in the SFC.

As P4 switches cannot instantaneously reload firmware without losing its internal state, scaling in or out

[1] Figure 4.2(a) demonstrates this. Flow (3)(4) needs to be related to (4)(2) for successful flow migration.

is implemented using table rules, which can enable or disable NFs. This simulates live, instantaneous and rule-preserving firmware reloads.

5.1. Monitoring

A novel algorithm that performs overload detection in the data plane (Sec. 4.3) was implemented, since current state-of-the-art algorithms impose high loads on switches and the control plane when used for individual flow monitoring. If the output queue length is rising, and the output queue length is longer than a threshold, a message is sent to the control plane. As small traffic bursts can cause temporary queuing, the threshold should be set high enough so that these bursts will not lead to (unnecessary) migrations. Overload notifications are sent as a digest, a short gRPC message sent by the switch to the control plane. Since the used P4 targets do not support transmitting digests at the end of packet processing, information, as in table 5.1, is stored in a special register, whose transmission will be triggered by the next packet arriving at the switch. While it is possible that no packets arrive after a digest has been prepared—hence delaying overload detection—one could safely assume that in the case of overloads new packets arrive, as these digests are only sent when the switch is overloading. While this could delay overload detection, it is considered insignificant as it is a shortcoming of the used P4 target, not of P4 itself, and real-world hardware will support sending egress digests.

Table 5.1: Information transmitted in digests

Timestamp	timestamp when the packet entered the queue
egress port	egress port for which the queue is rising
enqueue length	length of the queue when the packet was put into the queue
dequeue length	length of the queue when the packet was taken out of the queue
queue timedelta	time the packet spent in the queue

Switch and NF instance usage data is polled by the control plane. Packets that are used to discover active Software-Defined Networking (SDN) links are modified to retrieve switch and NF usage data. For each packet that is processed by the switch, a register value is increased. The same yields for each NF: each packet processed increases an internal counter. On the arrival of link discovery packets, usage information for the switch itself and its NFs is appended to these packets and forwarded to the control plane.

Individual flow rate updates can impose a significant load on the switch, similarly to high-rate polling. Whilst P4 supports meters, which are objects within the switch that measure data rate by assigning each packet with green, yellow, or red (indicating e.g. idle, busy or failing) values, meter implementations are architecture-specific and some vendors might not even implement them. Furthermore, having only three thresholds would be too coarse for individual flow rates. Therefore, individual flow rates are reported by the data plane. Switches will report flow rates for every 10 packets by default. This default value can be easily increased, as real-world hardware is generally much faster than our virtual environment. The threshold value should be chosen such that rates for new flows will quickly be available, as flow rates are generally higher than the threshold.

If the following conditions are met, the switch will send a digest message to the control plane with the amount of received packets and a timestamp. These digests can be sent directly, as they are generated in the ingress pipeline, instead of the overload signaling digests, which are generated from the egress pipeline.

5.2. Decision-making process

The decision-making process determines an appropriate solution to mitigate overloads on switches. First, the algorithm (shown in appendix A) tries, for each NF on the overloading switch, to find a set of flows it can migrate to another NF instance. If such a set can be found, no scaling is necessary, and the set of flows can be migrated to other instances. If no solution is found, the algorithm is executed again, but now considering switches where an NF instance can be spawned, instead of switches where an instance is already running. A solution consists of a switch, NF, and a set of flows.

Since load statistics for switches and NFs are low-rate and coarse-grain, a 10% margin is taken into account. This margin is introduced to ensure first that the decrease in flow volume on the overloading switch is large enough to stop the switch from overloading, and second, to make sure that extra flow volume introduced on the other switch will not immediately overload the migration destination. If no solution arises through neither rebalancing flows nor spawning new instances, these margins will have to be decreased. Furthermore, other solutions could be considered by increasing the allowed extra path length. In extreme cases, admission control should be in place to allow access to prioritize NFs or traffic.

Chapter 4 proposes four constraints. First, the flow volume should be large enough to stop the switch from overloading. Furthermore, any switch which will experience an increased load (by processing an NF or forwarding packets) must not be overloaded immediately. This is verified by calculating a solution and comparing if the sum of flow rates would (1) mitigate the overload and (2) will not overflow any new switches in each flow path.

The last constraint, which limits additional path length, is verified using the SFC path. Figure 5.1 shows an SFC path, where a flow is processed by three NFs. The decision algorithm checks if flows from NF2 could be migrated to NF2'. Since packets need to be processed by NF1, the route section [NF1, NF2, NF3] is compared to the path section [NF1, NF2', NF3].

If the path section length increases by more than N hops, the flow is not considered for migration. A large increase in path length could lead to very inefficient paths, which could easily congest the network and add lots of latency to the flow, especially in large or distributed networks. Therefore, N is chosen to be a relatively low integer: $N = 3$.

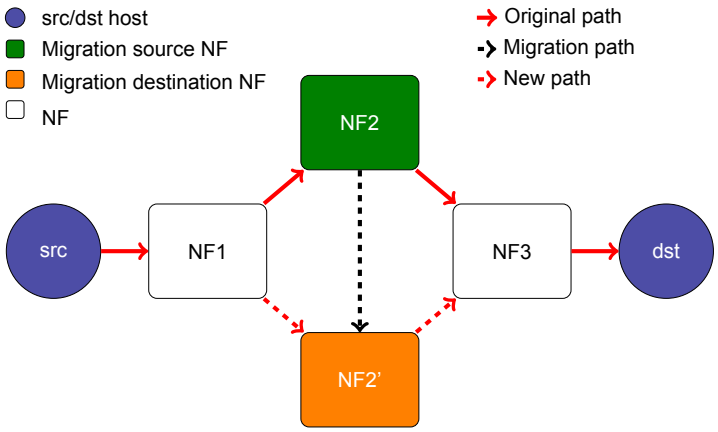


Figure 5.1: The SFC path for a flow from *src* to *dst* is shown. The flow is processed by three NFs, NF1, NF2, and NF3. If the switch where NF2 runs is overloaded, NF2' is evaluated as a migration option. The extra path length is calculated by subtracting the path [NF1, NF2, NF3] from the path [NF1, NF2', NF3], where path including NF2' does not include the switch where NF2 currently resides.

Especially in large networks, processing many NFs on many switches for many flows could be a processor-intensive task. Therefore, only switches that could handle the extra flow volume are consid-

ered, as other switches will never be considered.

5.3. Flow & state migration

The previously discussed flow- and state migration algorithms (Sec. 4.5) were implemented. To enable reliable, efficient state migration, that is resistant to out-of-order packets and packet loss, two new headers were defined. The first header consists out of protocol state information:

- flowhdr,
- flowID,
- pktType,
- migStatus,
- sequenceID,
- VNFID,
- migSessionID.

This header, which is used for all migration packets, eliminates the need to keep protocol states synchronized between the migration source and destination, as all information for transmitting migration packets, and protocol information is embedded in the packet. The *flowhdr* is a static value that indicates the parser (Appendix A) that the following packet is a migration packet. The *flowID* is the *switchFlowID* of either the migration source (traffic from the control plane to the migration source) or the migration destination (traffic from the migration source to the migration destination). The *pktType* and *migStatus* mark the packet type (e.g. initial sync, state update, flow packet forward) and the protocol status (e.g. initialization phase, update phase, or forward phase). The sequenceID is an incremental value. When state updates are received by the migration destination, it first checks the sequenceID. If the sequenceID it has stored in its registers is lower than the sequenceID in the packet, the state update is processed and the new sequenceID is stored. If the received sequenceID is lower than the sequenceID stored in its registers, an out-of-order packet has been received. As more recent information has already been stored, the packet will be ignored. The *VNFID* indicates which NF the state data is from, since each NF has its own data header type. This allows the switch to append the correct value header.

The *migSessionID* primarily indicates the *mirror_session*, a session which clones packets to specific egress pipelines. Shortest paths between switches are calculated when a switch is first connected to the network and *mirror_sessions*[2] are installed on each switch, to enable direct switch-switch communication. This approach enables in-data plane state migration without requiring switches to have direct links to each other, which could be very inefficient in large networks. By combining switchIDs in *migSessionIDs*, source and destination switches can be easily calculated, as equation 5.1 shows.

$$\begin{aligned} \text{source switchID} &= \text{migSessionIDs} \gg 4 \\ \text{destination switchID} &= \text{migSessionIDs} \wedge \text{0x0f} \\ \text{migSessionIDs} &= \text{source switchID} \ll 4 \wedge \text{destination switchID} \end{aligned} \tag{5.1}$$

The source switchID is derived by bitshifting the *migSessionID* four bits to the right. The destination switchID is derived by an *AND* operation with *0x0f*. On the other hand, the *migSessionID* can be derived from the source and destination switchIDs by the inverse operation (bitshifting the source switchID four bits to the left and then add the destination switchID).

[2] This functionality is currently not supported by gRPC. Therefore, Thrift RPC was enabled on the switches targets to add support for *mirror_sessions* on the P4 targets.

The second header contains network information which the migration source and destination use for register index calculations, and are similar to information from flow packets. It is important that this information is the same header information NFs use for index calculation, otherwise state information could still be migrated to a wrong register index at the destination. The *networkInfo* header contains:

- IPv4 source/destination address,
- TCP/UDP source/destination port (for protocols that don't use ports, a default value is used),
- IPv4 protocol.

Protocols that do not use ports (e.g. ICMP or IGMP) are assigned a default value of 143. To differentiate flows between the same source and destination the IPv4 protocol number is added. Both the migration source and destination use this information to calculate internal hashes, since different vendors use conflicting hash implementations, causing state information to be migrated to faulty register indexes and state information for other flows to be overwritten.

When combined, these headers are 28 byte. This ensures migrations do not congest links by sending unnecessary data, while still transmitting all necessary data.

Several data are required for a successful state migration. First, the migration source needs information about the migration destination, flow, and NF. This information is provided by the control plane during the initialization phase. The migration source switch stores the migration status, *VNFID* and *migSessionID* in its registers, at an index calculated from the networkInfo header, that is embedded to the packet.

When flow packets arrive at the migration source, the packet is cloned. The first packet is processed and forwarded a usually, and the clone will be sent to the migration destination. Further information for the migration header is retrieved from internal registers. Network information is copied from original packet headers, after which all unnecessary headers (e.g. Ethernet or IPv4) are removed to reduce the size of the packet. Flow rules are installed to insert the flowID value based on the migration destination (calculated from the *migSessionID*).

The forward phase is initialized by the control plane, after finishing installing the new path (see section 4.5). Any subsequent packets will not be processed by the NF on the migration's source, but will be forwarded to the migration destination, which will process and forward them to the flow's destination. To avoid a desynchronization between a migration's source, this phase transition is only communicated to the migration destination when the first packet is forwarded.

6

Experiments

This chapter will validate the design as described in chapter 5 by comparing it to scenarios without elastic scaling capabilities, a controller-based scaling approach, and *Swing State*, currently state-of-the-art. To evaluate our solution, we used the experiment setup, explained in section 6.1. Several test scenarios were run (section 6.5) on an experiment topology (section 6.2) while performance metrics explained in section 6.3 were collected. The results of these experiments will be explained in section 6.6. Section 6.7 will give a comparison between the used scenarios.

6.1. Experiment setup

A Mininet environment with P4 *simple_switch* switches and a control application, written in P4, were deployed on a virtual system equipped with 6 CPU cores and 6GB of RAM. The *simple_switch* targets were modified to support both the Google Remote Procedure Call (gRPC) and Thrift Remote Procedure Call (Thrift RPC) interfaces, which allows the switch to mimic egress cloning. This is not supported by *simple_switch* targets, which is required by our implementation in order to let switches forward flow packets and send state updates simultaneously to the migration destination switch.

The Mininet environment is embedded in prebuilt Docker [67] containers[1], which has the benefit that switching between various *simple_switch* target versions (e.g. log level, or specific versions) could be done in seconds. A control plane applications capable of performing P4 elastic scaling is implemented in a Python 2.7 environment [69]. Despite the fact that Python 2.7 is deprecated, many source libraries are not available for Python 3.x.

As the scaling in procedure is similar to the scaling out procedure, the focus of the experiments is scaling out since this procedure is more time critical: scaling in is performed to optimize and free resources, while scaling out is performed to prevent high latencies and packet loss for flows that are processed by Network Functions (NFs) running on overloading hardware.

Currently, live-reloading of P4 software is not possible. To model instantaneous, rule preserving reloading of P4 code, the Virtualized Network Function (VNF)'s code is embedded in the code at setup time and is activated by installing table rules. Therefore, migration times for both the control plane migration algorithm and our implementation will have a similar offset. Since this is a relative comparison, only redistribution times are shown.

[1] The used docker containers are available on DockerHub [68].

6.2. Experiment topology

To evaluate our solution the topology shown in figure 6.1, consisting of 4 switches, is used. This topology is a simple topology that allows only *switch 2 (s2)* and *s3* to run and scale NFs, as no paths are possible that do not include *s1* or *s4*. Furthermore, *s2* and *s3* are not directly connected, which requires either *s1* or *s4* to forward state information when states are communicated within the data plane.

Several performance tests concluded that packets could be generated with a rate up to roughly 750 packets/sec. Therefore, we configured the maximum packet rate on switches *s1* and *s4* to 500 packets, and on switches *s2* and *s3*, which will run NFs, to 250 packets. This configuration ensures that other factors (e.g. system bottlenecks) are not significant in the experiments. The queuing threshold (the threshold that triggers an overload) is therefore set rather low: if the switch's queue is rising and is longer than 20, an overload is triggered.

In normal networks a delay exists between the control plane and data plane. Since our topology is running on a single system this delay does not exist. To emulate a physical distance between the control plane and data plane, a constant delay of $500ms$ is implemented. By setting this value high, all scaling scenarios require several state updates to be transmitted.

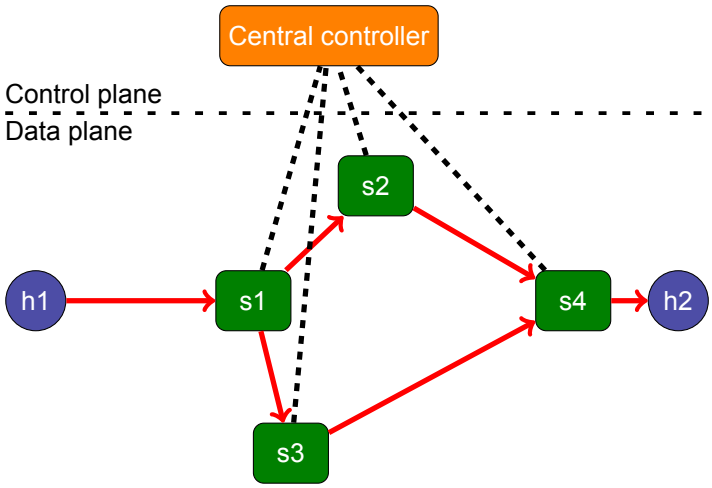


Figure 6.1: The network used for experiments. Four switches (s1-4) and two hosts (h1-2) are spawned in a Mininet environment. A Python control plane application was written, which is able to perform elastic scaling.

Each flow is processed by a packet counter NF, which increments a counter for each packet the NF receives. This packet counter is chosen since it has the most dynamic NF states (it changes its state each packet), making it the worst-case single variable NF. Therefore, if a solution is able to migrate states belonging to a packet counter NF, it is able to handle less dynamic ones (e.g. a firewall, where TCP states do not change for each packet). When the network starts, all NFs are handled by *s3*. When *s3* overloads one or more flows are migrated to *s2*. The default path for flows is therefore *(h1, s1, s3, s4, h2)*.

6.3. Performance metrics

The goal of the experiments is to validate the design and qualitatively measure improvement in migration time, overhead, and the quantity of packets exchanged between the control- and data plane. Therefore, queue lengths, packet delays, initial synchronization, and redistribution times are measured during the experiments.

Therefore, queue lengths, initial synchronization and redistribution times give an indication about how

well the migration protocol works. The time between overload detection in the switch and the initial migration of the NF's states to the migration destination switch is the initial synchronization time. As traffic rates could double during state synchronisation when using a worst-case NF (which triggers a *stateUpdate* for each flow packet), it is vital to keep this phase as short as possible. The total synchronization time is the time between overload detection in the switch and the moment the flow's path has been changed to directly send traffic to *s2*. This indicates that new packets are sent directly to the migration destination, which relieves the overloading switch.

In order to measure queue lengths and protocol times the state migration P4 code is modified to embed a timestamp in packets sent to the control plane. Furthermore, the overload detection, embedded in the P4 code, was modified to report queue lengths (on adding a packet to the queue and removing it from the queue respectively) and the packet timestamp for each packet belonging to the experiments. Since the timestamps start at the boot time of each switch, timestamp offsets are adjusted for.

Packet delays are measured from the transmitting host (*h1*) to the receiving host (*h2*). *H1* sends a timestamp in the packet data, which is subtracted by *h2* from the reception time.

6.4. Traffic pattern

Traffic with an increasing amount of packets was sent from *h1* to *h2*. Initially, 9 flows, each with a rate of 25 packets/sec, form an NF load of 225 packets/sec, which will not overload *s3*. A new flow of 25 packets/sec is added to the total flow volume after each 10-second interval. This pattern is chosen to test multiple iterating flow migrations. Since the packet rate is increasing with 25 packets/sec each 10-second interval, switch *s3* will experience a 10% overload at t≈ 20, which is considered to be the worst-case overload for our situation.

6.5. Experiment scenarios

Several scenarios are tested with the experiment topology. The implementation of our novel data plane scaling algorithm is modified in such a way that state updates are communicated using the control plane, and four situations are compared:

1. no scaling,
2. scaling, using a controller-based approach,
3. scaling, using current state-of-the-art in-data plane scaling (*Swing State*), and
4. scaling, using a data plane-based approach.

All scenarios are run on the network depicted in figure 6.1. The first scenario is a single NF, which is unable to scale and migrate. The second scenario, which is the current state-of-the-art is able to scale using a controller-based state migration approach, which is similar to our implementation, except that NF state migrations and updates are communicated through the controller. *Swing State*, the third approach communicates states within the data plane. The fourth scenario will communicate these states directly from the migration source switch (*s3*) to the migration destination switch (*s2*).

6.6. Experiment results

The scenarios, as explained in section 6.5 were executed on the experiment topology. Figures 6.2, 6.3, 6.4, and 6.5 show the traffic rate, packet delay, and queue length of load distribution, for the

scenario where no scaling algorithm, a controller based state migration approach, the current state-of-the-art data plane approach (*Swing State*), or a data plane based approach were active. To improve the visibility of the graphs, only the first $35s$ of the experiment is shown. These measurements were repeated multiple times, which showed similar results. Other traffic patterns (more lower-rate flows, and less higher-rate flow) were tested as well, which led to similar observations.

Our solution seems to detect overloads earlier. However, as all scenarios (the controller-based, *Swing State*, and our data plane flow migration approach) use the same algorithm for overload detection this is likely caused by virtualization overhead or other programmatic inefficiencies in VMWare, Docker, the *simple_switch* P4 target or Mininet. Therefore, only relative times are compared.

The upper graph in figures 6.2, 6.3, 6.4, and 6.5 shows the packet delay measured during the experiment for each scenario. Visible in each graph is a peak in latency of roughly $500ms$, caused by the installation of new flows in the network at $t = 10s$, $t = 20s$, and $t = 30s$. Especially at the start of the experiment, where 9 new flows arrive at the control plane almost simultaneously. As in any Software-Defined Networking (SDN), packets from unknown flows are first sent to the controller, which then installs the flow path on the network. Due to the additional latency between the control plane and the data plane, the first few packets of each flow experience more delay than packets from flows which already have been installed on the network.

All test scenarios showed a slight increased latency in a small period at approximately $25s$. This could be due migration algorithms, which can delay certain packets. However, as the scenario without any scaling scenario also shows this increase, it is more likely that some tasks are performed regularly in the *simple_switch*, Mininet or Python, which could slightly delay a small amount of packets.

After migrations have been performed, some flows are processed by a different switch. These flow latencies will vary per-flow. Flows that are processed by the (previously) overloading switch will have a higher latency than flows processed by the migration destination. As the migration destination switch does not have queue buildup it does not add queuing delays to flows it processes. Therefore, flows processed by the migration destination experience less latency during their transmission in the network.

This behaviour is shown in the four figures. Both the controller-based algorithm, *Swing State* and our data plane scaling algorithm show alternating flow latencies which are observed by several flows after one or more flows have been migrated. While the queue is reducing on the migration's source, the maximum latencies decline.

The middle graph in in the four figures depicts the traffic statistics during the experiment for each scenario. Shown are the switch capacity (250 packets per second), the total amount of NF packets (combination of packets handled by the migration source switch and migration destination switch), individual switch NF packet statistics, and migration overhead packets. Except from the scenario without scaling, all four implementations show that overhead traffic is still transmitted between the control plane and the data plane when the flow migration is finished. This is due to the fact that we consider a migration to be ready when the flow path has been changed. Any packets that have accumulated in the switch's queue (as happens in overloading switches) still need to be transmitted. This causes migration overhead to be visible, even after our migration period ended.

The bottom graph in the four figures shows the queue lengths and the migration period during the experiment for each scenario. Queue lengths for packets entering the switch's queue (enqueue length) and for packets leaving the queue (dequeue length) are shown. If a packet's dequeue length is higher than its enqueue length it means the queue is growing since the queue increased its length while the packet was in the queue.

Grey areas show a migration of a single flow. Since one overload could lead to several flow migrations, different shadings indicate multiple flow migrations. Darker areas indicate several flow migrations.

6.6.1. Approach without scaling algorithms

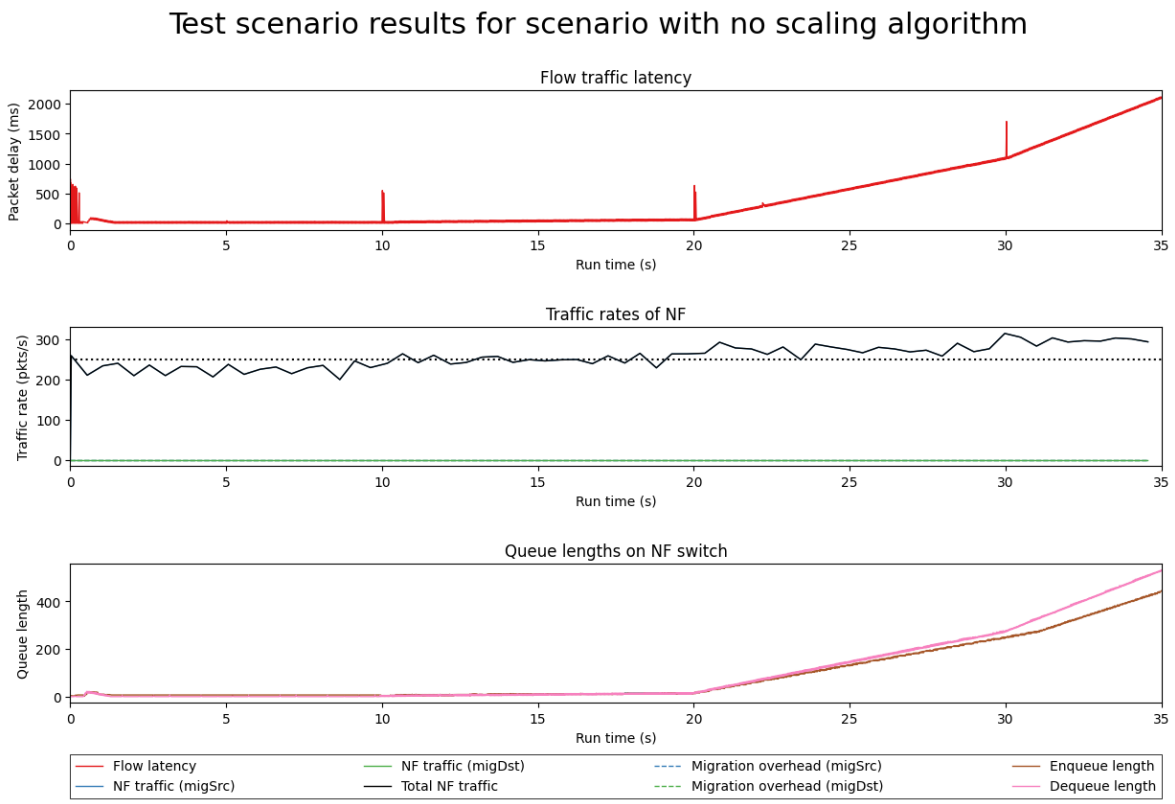


Figure 6.2: Baseline measurement for the traffic profile sent through the experiment topology without any overload mitigation algorithms. The upper graph shows the latency experienced by the flow, the middle graph shows the NF traffic processed by the switch and the lower graph shows the queue lengths on the switch.

Figure 6.2 shows the traffic statistics, flow latency and queue lengths for traffic sent through *s3* without any algorithms in place to rebalance flows or scale an NF to another switch. As the incoming packet rate increases, the queue length increases, since the input packet rate is higher than the output packet rate and no algorithms are in place to reduce loads on the system. It is clearly shown that the queue size and latency packets increase faster as the incoming packet rate increases; each of the four intervals (0:10, 10:20, 20:30, 30:35) shows a faster increasing queue and latency, due to the increase of the number of flows that transmit packets. The maximum observed queue size in the shown interval is roughly 3527.

The middle graph shows that only *s3* is processing packets. Both the latency and queue size are increasing faster as the packet rate increases. Furthermore, as no flows can be migrated, no migration overhead traffic is visible.

In order to model a maximum capacity of an NF, a limit was set on the transmission rate on each switch. However, queuing only occurs after the NF has been processed. This results in the fact that the NF seems to be performing above its capacity. However, this is not the case, as packets are buffered after NF execution.

The lower graph shows an increasing queue. Due to the absence of scaling algorithms, loads cannot be reduced. This results in an ever-increasing queue, until the maximum queue length is reached. Then, the overloading switch will start dropping packets as there is no memory to temporarily store the packet in.

6.6.2. Controller-based approach

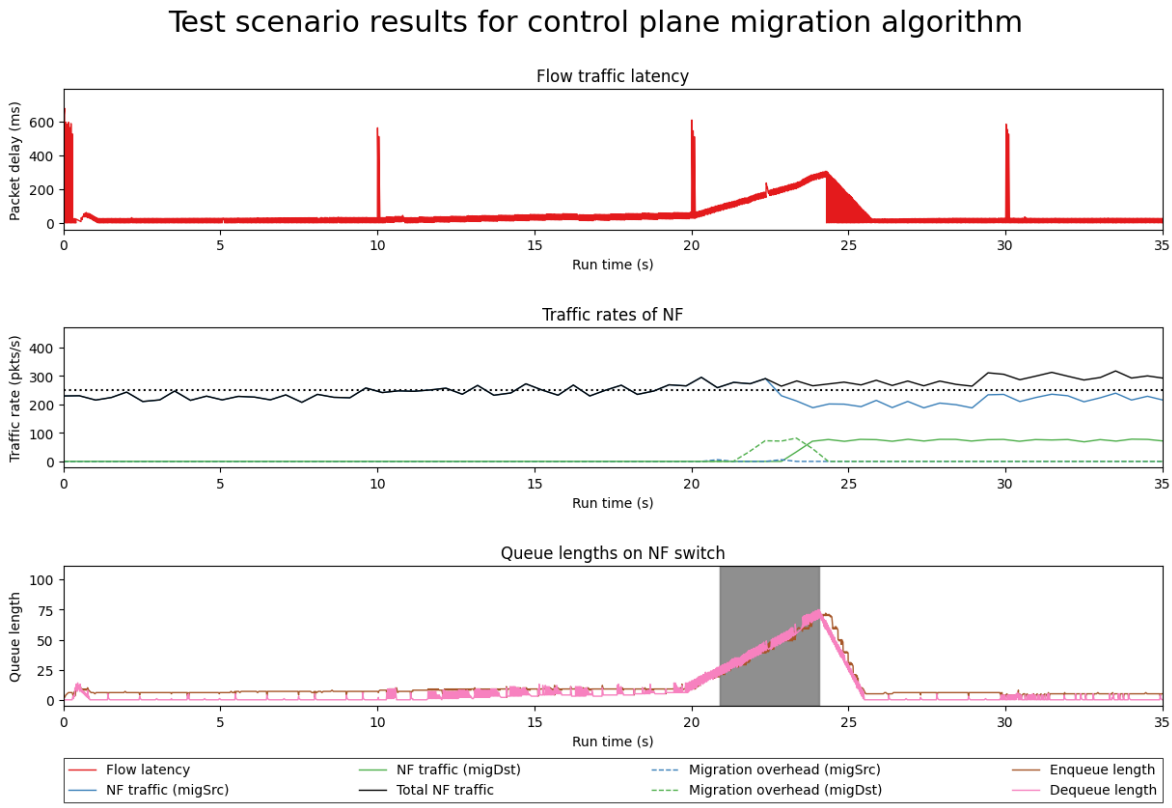


Figure 6.3: Migrations performed using control plane based state migration. The upper graph shows the latency experienced by the flow, the middle graph shows the NF traffic processed by the switch and the lower graph shows the queue lengths on the switch.

When introducing scaling techniques to the setup, several flow migrations are observed (figure 6.3). Figure 6.3 shows the traffic statistics, flow latency and queue lengths for traffic sent through the network with a control plane based scaling algorithm, which uses the control plane to communicate state updates from *s2* to *s3*.

Figure 6.3 shows that after approximately 21 seconds an overload is detected. Three flows are migrated. During the migration period, the migration destination experiences migration overhead, which is approximately equal to the total flow volume it will take over from the overloading switch. Since a worst-case NF is used, state updates are generated for each flow packet. The migration overhead at the migration's source is consisting of a few packets.

Due to increasing queue sizes, flows experience higher latencies. Maximum observed latency during the migration period is approximately $300ms$. The maximum observed queue size during migration was roughly 75. After $3.16s$ the migrations are completed. The NF instance load reduces, queues at the previously overloading switch decline, and maximum flow latencies decrease.

6.6.3. Swing state data plane approach

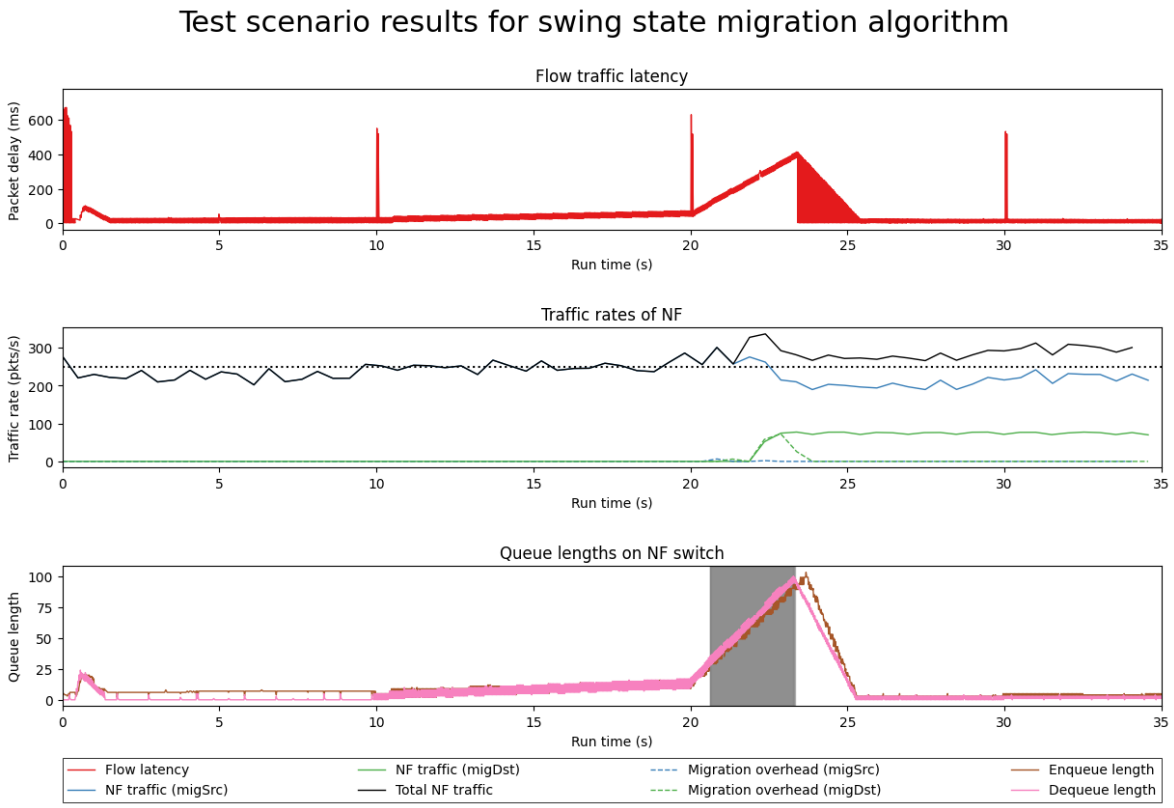


Figure 6.4: Migrations performed using current state-of-the-art (*Swing State*) data plane based state migration. The upper graph shows the latency experienced by the flow, the middle graph shows the NF traffic processed by the switch and the lower graph shows the queue lengths on the switch.

Figure 6.4 shows flow latency, traffic statistics, and queue lengths for traffic sent through the network with our data plane based scaling algorithm. This algorithm communicates states directly in the data plane between the migration's source (*s3*) and destination (*s2*).

As *Swing State* clones traffic to the migration destination and lets this switch process the NF to update internal states (after which the packet is dropped), NF traffic is increasing at the migration's destination. Then, both the migration's source and destination are processing NF packets, but the migration's destination drops them after processing. Only when the path is changed by the control plane, traffic starts to decline at the migration's source.

Due to increasing queue sizes, flows experience higher latencies. During this period, the maximum observed latency (excluding the peaks at each 10-second interval and at the beginning of the measurements) is approximately $400ms$. The maximum observed queue size during migration was roughly 100. After $2.61s$ the migrations are completed. The NF instance load reduces, queues at the previously overloading switch decline and maximum flow latencies decrease.

6.6.4. Data plane scaling approach

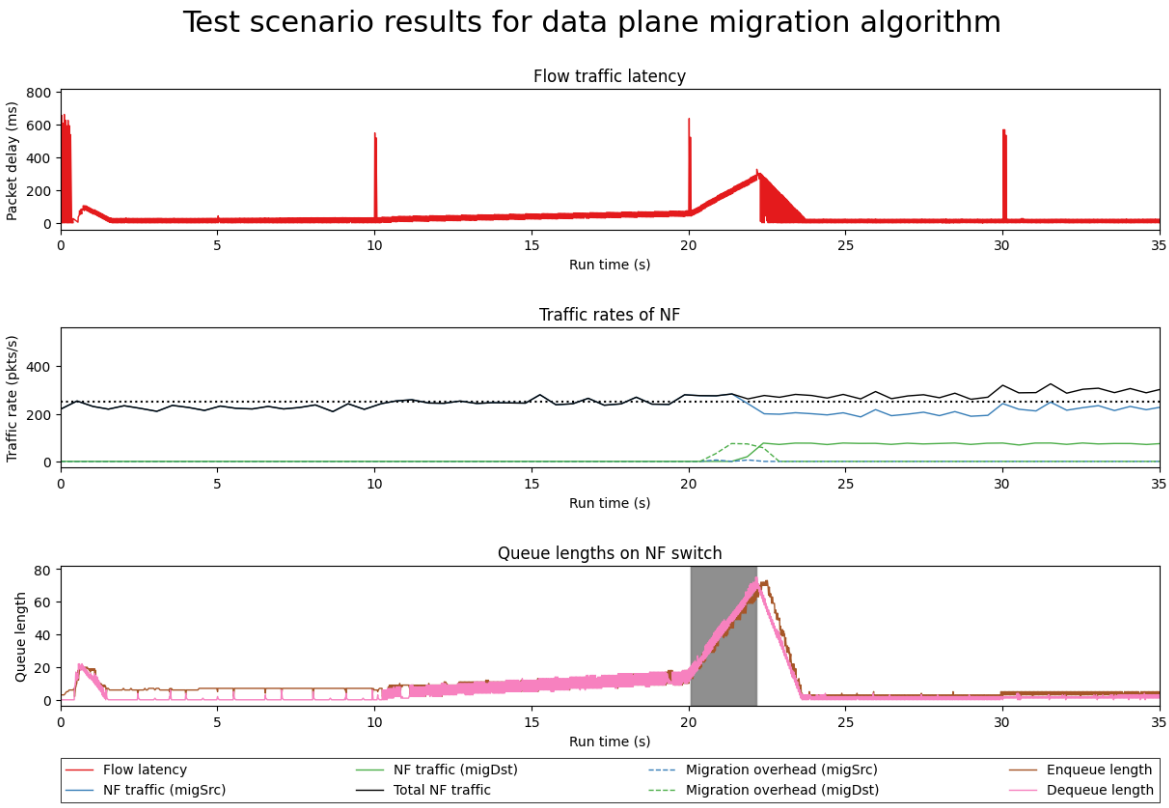


Figure 6.5: Migrations performed using data plane based state migration. The upper graph shows the latency experienced by the flow, the middle graph shows the NF traffic processed by the switch and the lower graph shows the queue lengths on the switch. Each single flow migration is indicated as a gray area; more simultaneous migrations are darker areas.

Figure 6.5 shows flow latency, the traffic statistics, and queue lengths for traffic sent through the network with our data plane based scaling algorithm. This algorithm communicates states directly in the data plane between the migration's source (*s3*) and destination (*s2*).

Figure 6.5 shows that after approximately 20 seconds an overload is detected. Three flows are migrated. During this period, the migration destination experiences migration overhead, which is approximately equal to the total flow volume that is migrated. Since a worst-case NF is used, state updates are generated for each flow packet. After $2.16s$ the migrations are completed. The NF instance load reduces, queues at the previously overloading switch decline and maximum flow latencies decrease. During the migration period, latencies did not exceed $290ms$. The maximum queue size was roughly 76.

6.7. Comparison

The four scenarios are compared in terms of latency experienced by the flows, queue lengths, and observed traffic. The above measurements are repeated multiple times, each showing similar results. Table 6.1 summarizes the main differences between the scenarios.

[2]Flow latencies are based on 1-second averages, to reduce the impact of rule installations on the latency, while still considering differences between various scenarios.

Table 6.1: Comparison of no scaling implementation, a elastic scaling solution using a controller based state migration approach, *Swing State*, and our novel data plane based state migration. Measurements are obtained in several measurements where in total 50 flow migrations were performed for each scaling scenario.

	Initial state sync time (s)			Total migration time (s)			flow latency[2] (ms)			Max queue length
	min	avg	max	min	avg	max	min	avg	max	
No elastic scaling	-	-	-	-	-	-	2.43	9832.53	28096.50	7048
Elastic scaling										
Control plane	2.06	2.11	2.19	3.11	3.38	4.23	2.59	56.07	276.97	128
Swing State	1.03	1.07	1.13	2.60	2.65	2.72	3.68	47.60	226.47	106
Data plane	1.04	1.09	1.14	2.06	2.13	2.20	3.51	43.50	177.75	80

Overall, the scenario without a scaling solution performed the worst. When any flow migration algorithm is enabled on the network, flow latencies and maximum queue lengths drastically improve. The controller-based design improves this behaviour by implementing scaling. This is further improved by *Swing State* and our implementation, which use the data plane for transferring state.

Experiments show that *Swing State* and our solution show similar overhead rates. However, due to the fact that *Swing State* uses full packets to transfer state and our solution is using small packets of 28 byte, our solution requires lower migration bandwidth.

Compared to the controller-based design, *Swing State* is able to reduce migration latencies with $0.73s$, and is able to reduce initial state synchronisation by $1.07s$ on average. *Swing State* performs initial state migration $20ms$ quicker than our solution on average, while our implementation performs full migration $0.52s$ quicker than *Swing State* does. Furthermore, our algorithm shows improvements in observed queue sizes and latency experienced by flows compared to the other scenarios.

Swing State clones packets in all protocol phases. This leads to higher packet rates on the switch that is already overloading. Our algorithm improves this, by only requiring packet cloning in the update phase. Compared to *Swing State*, our algorithm reduces maximum queue sizes during migration by 25%. As the controller-based approach takes longer to migrate flows, queue sizes grow even longer.

7

Result discussion

Experiments showed that our solution is able to migrate flows reliably and efficiently, by directly communicating Network Function (NF) states without using the control plane. Measurements have proven that our solution is able to migrate flows on average in $2.13s$, where a comparable control plane based migration solution takes $3.38s$ to migrate flows. *Swing State*, the current state-of-the-art approach on migrating NF states, takes $1.07s$ to perform initial state migration, and migrates the flow in $2.65s$ on average. Our algorithm achieves initial state migration in a comparable time ($1.09s$), and improves flow migration times by $0.52s$.

In contrast to expectations, *Swing State* was able to perform initial *flow* migrations approximately $20ms$ quicker than our implementation. As *Swing State* has to wait for packets when the protocol is activated, our implementation directly forwards state information to the migration's destination. This could be explained by the fact that we did not use hardware accelerated devices, but a virtual software_switch. Since we run different programs, one program could be asking more resources than another program, slightly delaying the switch. This could also be caused by inefficiencies in our system. As the environment was virtualized, load on other servers could influence the measurements.

As our control plane latency is $500ms$, the initial state migration time is heavily impacted by this. As *Swing State* and our algorithm require only two messages between the control plane and the data plane: one to detect the overflow, and one to instruct the overloading switch to transfer states, it transfers state more quickly. The controller-based algorithm requires four communications: detecting the overflow, instructing the overloading switch, sending the state from the overloading switch to the control plane, and sending the state from the control plane to the migration destination.

If we subtract the transmission delays from these migration times, state migration via the data plane is performed in $90ms$. The controller-based approach takes $70ms$ to initially migrate a state. This difference is most likely due extra processing delays in the controller, which needs to determine what to do with the packet, causing an additional delay of approximately $20ms$ on our system.

Short initial state migration times are important for our algorithm since the largest overhead is observed during the update phase of the flow migrations, as each state update requires an additional packet on top of the overload that the switch already experiences. When the state reception acknowledgement arrives, the control plane can initiate the forward phase, where the overloading switch only forwards packets to the migration destination instead of forwarding flow packets and sending state updates.

The experiments are performed in a very low-rate environment. As rates are higher, the control plane would experience overloads, which would changed the outcome of the experiments. However, as we could only use a single system, system overloads did occur when running all of the network, switches,

controller, and monitoring scripts all at once, delaying not only the controller application, but also other tasks, such as packet generation.

Even in this environment, our algorithm shows several benefits over both the controller-based design and *Swing State*. First it has a much lower migration overhead during state migration as our algorithm uses 28 byte packets to which the state values are appended, instead of using the whole packet to append state information. Since our solution uses smaller packets and finishes quicker (an improvement of 8.6% on average), our solution causes less data to be transferred. Second, our algorithm imposes less load on an overloading switch. *Swing State* requires packet cloning throughout all migration protocol phases, whereas our algorithm only duplicates packets during the update phase, which takes $1.09s$ on average. This shows in maximum observed queue lengths. *Swing State* reaches queue lengths of 106, whereas our algorithm is able to perform migration with maximum observed queue lengths of 80.

8

Conclusion

In this thesis we proposed a new solution to add elasticity to hardware-accelerated switches. A novel, horizontal scaling solution for P4 Network Functions (NFs) was designed and implemented. We investigated ways to introduce elastic scaling to P4 Virtualized Network Functions (VNFs). By implementing a general scaling solution in P4, NFs could benefit from both scaling and hardware acceleration simultaneously, as many NFs can be implemented in P4 to let them profit from hardware acceleration, without requiring them to run on vendor-specific hardware.

Our solution is able to migrate flows by offloading a certain part of the switch's load to another switch already present in the network by using horizontal scaling. Flow migration is performed in three stages, namely the initialization phase, update phase, and the forward phase. In these phases, a state is migrated to the migration destination and updated if states change at the migration source. When this is all completed, the flow path is changed in such a way that the load is reduced from the overloading switch.

A decision-making algorithm decides what flows to migrate to which switch, given an already existing Service Function Chain (SFC). It is able to scale elastically, based on flow rate information from switches, NFs, and individual flows. To reduce overload detection latency, switch overloads are detected by the switches. Flow rate information is gathered by a novel process, which only monitors high-rate flows, instead of all flows, in order to reduce the monitoring overhead.

Our solution has several advantages over the current state-of-the-art: *Swing State*. First, it has a much lower migration overhead during state migration as our algorithm uses 28 byte packets to which the state values are appended, instead of using the whole packet to append state information. Since our protocol finishes migrations quicker, and the observed migration overhead packet rate is similar, our solution will transmit less data to transfer NF states.

Second, our algorithm imposes less load on an overloading switch. *Swing State* requires packet cloning throughout all phases on the migration's source, whereas our algorithm only duplicates packets during the update phase, which takes $1.09s$ on average. Third, our solution is able to deal with packet loss, as the NF states are plainly overwritten, instead of sending incremental values. Our protocol is able to deal with out-of-order packets, since it does not overwrite NF states with older information, due to the incremental *sequenceID* appended to state updates.

In order to reduce the overhead of detecting overloads, our solution detects overloads on the switches, which monitor their own queues. If the queue size is longer than a certain amount of positions and is growing, the control plane is notified of an overload. This eliminates the need for high-rate polling, as the overload detection latency does not depend on a sample frequency. Moreover, this reduces the

overall overload detection latency, as overloads are detected as quickly as the best-case in sampling scenarios[1].

In order to find a location to spawn new instances, several data can be used. In this thesis, we propose to use overall switch loads which, together with a decision-making algorithm that considers NF, and individual flow rate statistics, decides to migrate which flows to which NF instance. These values are sampled approximately every second. Switch and NF rate statistics are polled using packets of the link discovery process. Flow statistics are sent by the switches. To increase efficiency, rate statistics are only gathered for high-rate flows.

This decision-making process ensures that solutions proposed by the process are valid and do not overload other switches by checking if the total flow rate is within limits of the switch and path lengths do not increase above a certain threshold. To ensure the solution is valid a safety margin of 10% on the switch capacity and the flow rate that is migrated is taken into account to allow some variability in flows that cross the switch.

As states are communicated within the data plane, the control plane only has a coordinating role. Therefore, the control plane will not be overloaded by performing flow migrations.

The load on the control plane is reduced, as the control plane only indicates protocol phases, instead of actively relaying NF states. Packets indicate the protocol phase, which is coordinated by the migration source. This ensures that protocol phases are only programmed on one switch which, in turn, prevents desynchronization issues.

Migrations are only performed when necessary (i.e. when a switch overloads) and only migrates flows to switches which are not overloaded by the additional load. A mechanism to optimize NF instance usage is in place, that periodically checks whether NF instances can be removed without overloading switches.

Our decision-making protocol takes network topology into account, by preventing path length to increase more than three network hops. Our P4 solution is able to forward state updates between switches, eliminating the need for direct links between the migration source and migration destination.

In order to decide which flows to migrate to which switch, our solution uses a low-overhead to provide the decision-making algorithm with switch, NF, and individual flow information. Since only high-rate flows are considered for migration, low-rate flows are not monitored to reduce the monitoring overhead.

These findings provide a potential mechanism for elastic scaling in hardware accelerated VNFs. Compared to a similar control plane based state migration approach, our data plane based migration protocol reduces initial state synchronization times by $1.02s$ and improves flow migration times by $1.26s$, a reduction of 37%. As this decreases queue buildup on the overloading switch, the average latency experienced by flows reduced by $12.57ms$.

8.1. Future work

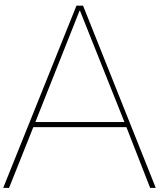
Future studies should further develop and confirm these initial findings by running experiments on real-world hardware accelerated environment, instead of a virtual Mininet environment. The assumption that stateful, instantaneous firmware reloads on P4 hardware is currently unavailable might be addressed in future studies, as technology advances. In future work, investigating how to mitigate overloads by migrating multiple NFs to multiple switches might prove important for a more efficient network and more flexible deployment of hardware. Especially large, dense networks might benefit from this.

[1] The best-case overload detection latency in a monitoring scheme which uses sampling to retrieve usage statistics is when an overload occurs the exact moment before indicators are sampled.

Furthermore, future work could focus on migrating NF states that are network-dependent (e.g. in- or output port). More research should be conducted into the individual flow monitoring. As currently a fixed flow volume, depending on the overload volume and the capacity of the switch is monitored, monitoring behaviour could be further optimized by only monitoring flows when the load is over a certain percentage.

Research could be done in order to achieve high-availability P4 NFs, by transmitting state updates even when switches are not overloading. If a separate phase of the protocol could be implemented, flows could automatically be routed to backup NF in the case the primary NF fails. Our solution could have uses in line congestion prevention, by considering link loads, instead of switch loads.

This work could be used as a way to achieve elastic network slicing. If slices could be placed dynamically using our in-data plane state migration algorithm, network slicing would become more dynamic and could use hardware more efficiently; ultimately leading to lower costs for customers.



Appendices

Decision-making process for flow migration

Algorithm 2 Pseudo code for decision-making process $SwitchPool is either the pool of switches that does not run the NF yet (scaling), or the pool of switches that does (rebalancing flows volumes).

```
switch                                          ▯ Switch that is experiencing overload
migrateVol                                      ▯ volume to reduce load on switch, with 10% margin

for each NF on switch do
    Find all current NF instances on switch
    for each sw $SwitchPool do
        freeVolunused capacity on sw                                        ▯ 10% margin
        if freeVol > migrateVol then
            for each flow on switch do
                if newPathLength < 3 + curPathLength then    ▯ Only allow an added path length of 3
                    totalFlowVol += currentFlowVol
                end if
                if totalFlowVol > migrateVol && totalFlowVal < freeVol then
                    return Solution                                         ▯ Return NF, switch, Flows
                end if
            end for
        end if
    end for
end for
return noSolution
```

P4 parser tree

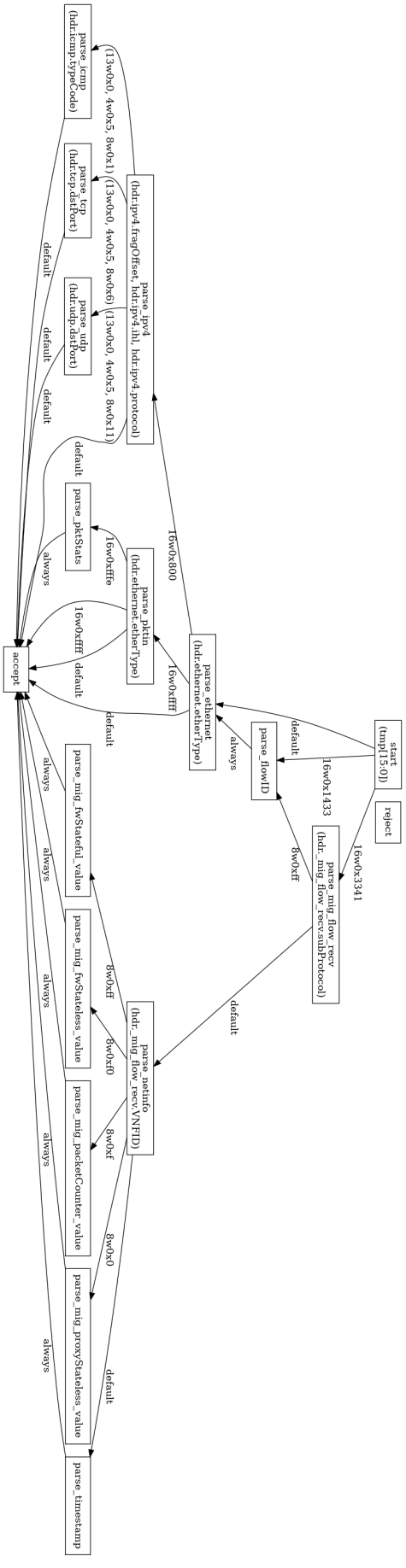


Figure A.1: Parser representation of P4 parser. Parsing starts at start, for different fields it can parse different headers.

P4 parser tree

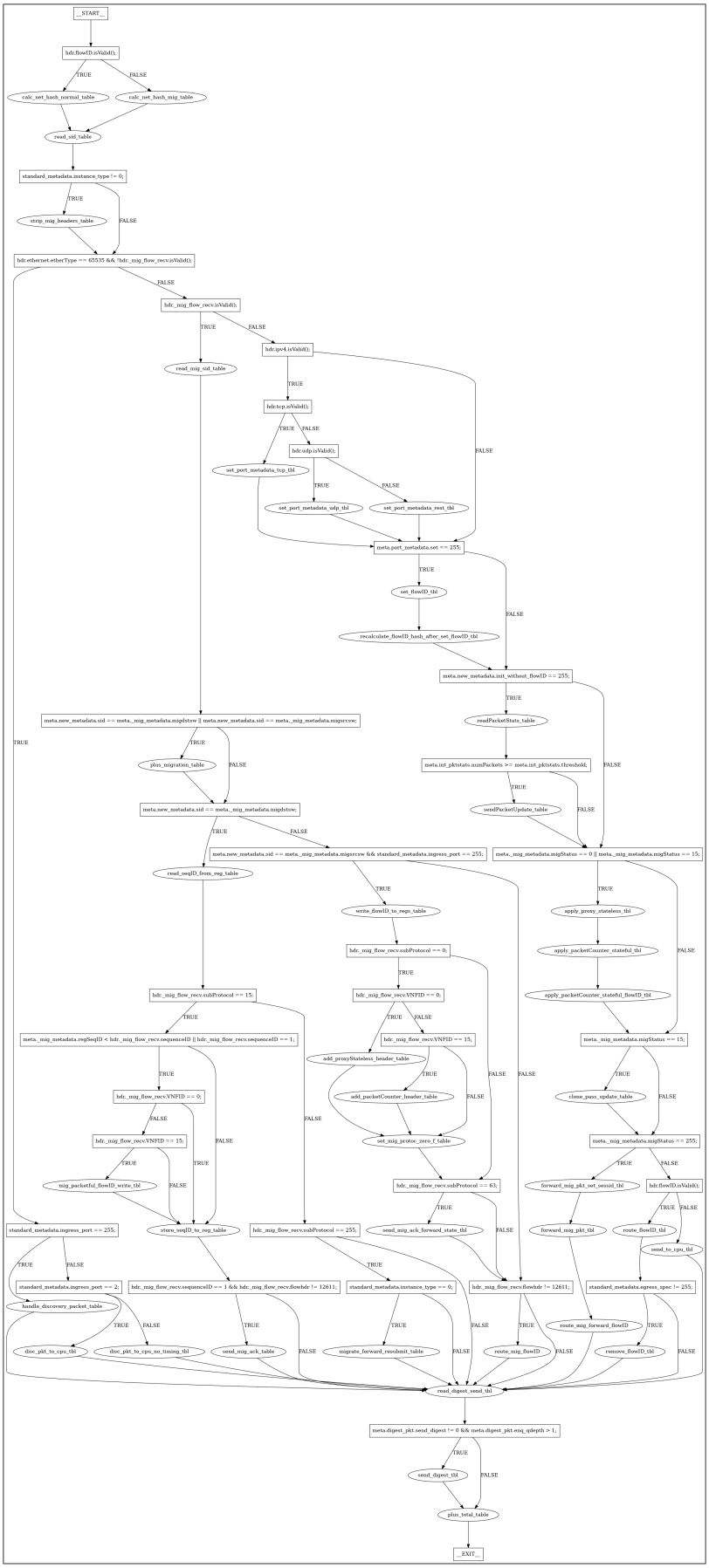


Figure A.2: Flow diagram representation the ingress pipeline in the P4 program

P4 egress logic tree

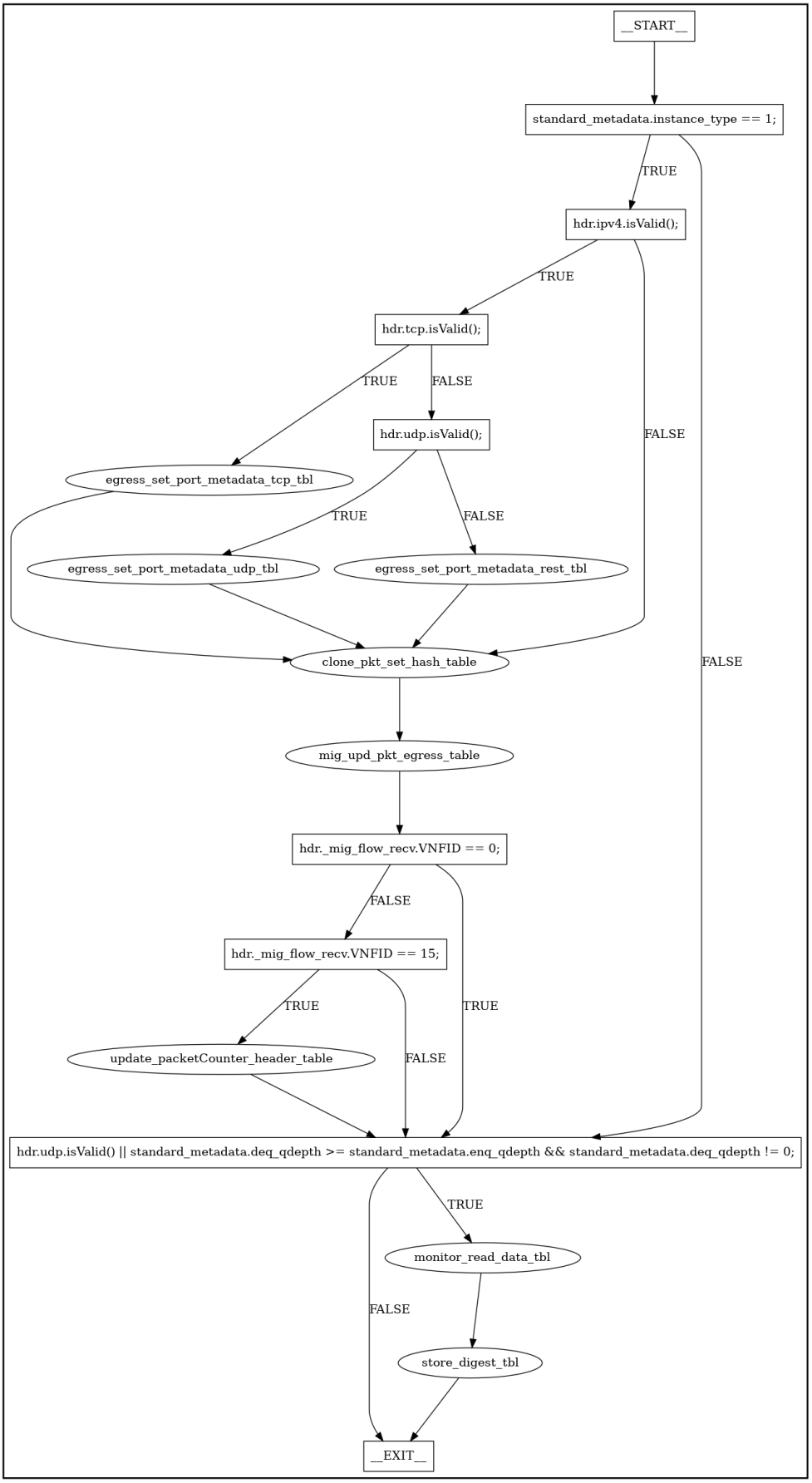


Figure A.3: Flow diagram representation the egress pipeline in the P4 program

Bibliography

[1] Margaret BT Chiosi, Don Clarke, Peter Willis, Andy Reid CenturyLink, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Javier Benitez, Uwe Michel, Herbert Damker KDDI, Kenichi Ogaki, Tetsuro Matsuzaki NTT, Masaki Fukui, Katsuhiro Shimano, Dominique Delisle, Quentin Loudier, Christos Kolias, Ivano Guardini, Elena Demaria, Roberto Minerva, Antonio Manzalini, Diego López, Francisco Javier Ramón Salguero, Frank Ruhl, and Prodip Sen. Network Functions Virtualisation. Technical report, ETSI, 2012. URL `http://portal.etsi.org/NFV/NFV_White_Paper.pdf`.

[2] Diego Kreutz, Fernando M.V. Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 2015. ISSN 15582256. doi: 10.1109/JPROC.2014.2371999.

[3] Meryem Simsek, Adnan Aijaz, Mischa Dohler, Joachim Sachs, and Gerhard Fettweis. 5G-Enabled Tactile Internet. *IEEE Journal on Selected Areas in Communications*, 2016. ISSN 07338716. doi: 10.1109/JSAC.2016.2525398.

[4] Wei Cai, Ryan Shea, Chun Ying Huang, Kuan Ta Chen, Jiangchuan Liu, Victor C.M. Leung, and Cheng Hsin Hsu. A survey on cloud gaming: Future of computer games, 2016. ISSN 21693536.

[5] Ejder Bastug, Mehdi Bennis, Muriel Medard, and Merouane Debbah. Toward Interconnected Virtual Reality: Opportunities, Challenges, and Enablers. *IEEE Communications Magazine*, 55 (6):110–117, 2017. ISSN 01636804. doi: 10.1109/MCOM.2017.1601089.

[6] Kjetil Raaen. Measuring latency in virtual reality systems. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2015. doi: 10.1007/978-3-319-24589-8{_}40.

[7] Ryan Shea, Jiangchuan Liu, Edith Ngai, and Yong Cui. Cloud gaming: Architecture and performance. *IEEE Network*, 2013. ISSN 08908044. doi: 10.1109/MNET.2013.6574660.

[8] Arun Agarwal, Gourav Misra, and Kabita Agarwal. The 5th Generation Mobile Wireless Networks-Key Concepts, Network Architecture and Challenges. *American Journal of Electrical and Electronic Engineering*, 2015. doi: 10.12691/AJEEE-3-2-1.

[9] Sreekrishna Pandit, Frank H.P. Fitzek, and Simone Redana. Demonstration of 5G connected cars. In *2017 14th IEEE Annual Consumer Communications and Networking Conference, CCNC 2017*, 2017. ISBN 9781509061969. doi: 10.1109/CCNC.2017.7983187.

[10] S. Sofana Reka, Tomislav Dragičević, Pierluigi Siano, and S. R. Sahaya Prabaharan. Future generation 5G wireless networks for smart grid: A comprehensive review, 2019. ISSN 19961073.

[11] Tamás Haidegger, József Sándor, and Zoltán Benyó. Surgery in space: The future of robotic telesurgery. *Surgical Endoscopy*, 2011. ISSN 14322218. doi: 10.1007/s00464-010-1243-3.

[12] Xiaoyao Li, Xiuxiu Wang, Fangming Liu, and Hong Xu. DHL: Enabling flexible software network functions with FPGA acceleration. In *Proceedings - International Conference on Distributed Computing Systems*, volume 2018-July, pages 1–11. Institute of Electrical and Electronics Engineers Inc., 7 2018. ISBN 9781538668719. doi: 10.1109/ICDCS.2018.00011.

[13] Leonhard Nobach and David Hausheer. Open, elastic provisioning of hardware acceleration in NFV environments. In *Proceedings - International Conference on Networked Systems, NetSys 2015*, 2015. ISBN 9781479958047. doi: 10.1109/NetSys.2015.7089057.

[14] Zvika Bronstein, Evelyne Roch, Jinwei Xia, and Adi Molkho. Uniform handling and abstraction of NFV hardware accelerators. *IEEE Network*, 2015. ISSN 08908044. doi: 10.1109/MNET.2015.7113221.

[15] Pat Bosshart, George Varghese, David Walker, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, and Amin Vahdat. P4. *ACM SIGCOMM Computer Communication Review*, 2014. ISSN 01464833. doi: 10.1145/2656877.2656890.

[16] Intel® Tofino™ 2: P4-programmable Ethernet switch ASIC that delivers better performance at lower power. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, 2020. [Online; accessed 11-November-2020].

[17] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the way for NFV: Simplifying middlebox modifications using StateAlyzr. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2016*, 2016. ISBN 9781931971294.

[18] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. Measuring control plane latency in SDN-enabled switches. In *Symposium on Software Defined Networking (SDN) Research, SOSR 2015*, 2015. ISBN 9781450334518. doi: 10.1145/2774993.2775069.

[19] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 2002. ISSN 03600300. doi: 10.1145/568522.568525.

[20] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC 2013*, 2013. ISBN 9781450324281. doi: 10.1145/2523616.2523635.

[21] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Maneshy, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenke. Rollback-recovery for middleboxes. In *SIGCOMM 2015 - Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 227–240, 2015. ISBN 9781450335423. doi: 10.1145/2785956.2787501.

[22] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. *NSDI*, 2017.

[23] German Sviridov, Marco Bonola, Angelo Tulumello, Paolo Giaccone, Andrea Bianco, and Giuseppe Bianchi. LODGE: LOcal Decisions on Global statEs in prograna naable data planes. In *2018 4th IEEE Conference on Network Softwarization and Workshops, NetSoft 2018*, pages 28–36, 2018. ISBN 9781538646335. doi: 10.1109/NETSOFT.2018.8460115.

[24] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. *Sigcomm '14*, 2014. ISSN 0146-4833. doi: 10.1145/2619239.2626313.

[25] Jingpu Duan, Xiaodong Yi, Shixiong Zhao, Chuan Wu, Heming Cui, and Franck Le. NFVactor: A Resilient NFV System Using the Distributed Actor Model. *IEEE Journal on Selected Areas in Communications*, 2019. ISSN 15580008. doi: 10.1109/JSAC.2019.2894287.

[26] Libin Liu, Hong Xu, Zhixiong Niu, Peng Wang, and Dongsu Han. U-HAUL: Efficient state migration in NFV. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2016*, 2016. ISBN 9781450342650. doi: 10.1145/2967360.2967363.

[27] Wenxin Wang, Yujie Liu, Yong Li, Haoyu Song, Yue Wang, and Jian Yuan. Consistent State Updates for Virtualized Network Function Migration. *IEEE Transactions on Services Computing*, 2017. ISSN 19391374. doi: 10.1109/TSC.2017.2765636.

[28] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. Swing state: Consistent updates for stateful and programmable data planes. In *SOSR 2017 - Proceedings of the 2017 Symposium on SDN Research*, 2017. ISBN 9781450349475. doi: 10.1145/3050220.3050233.

[29] Bo Yi, Xingwei Wang, Keqin Li, Sajal k. Das, and Min Huang. A comprehensive survey of Network Function Virtualization, 2018. ISSN 13891286.

[30] Manuel Peuster, Holger Karl, and Steven Van Rossem. MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments. In *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks, NFV-SDN 2016*, 2017. ISBN 9781509009336. doi: 10.1109/NFV-SDN.2016.7919490.

[31] Bernardetta Addis, Dallal Belabed, Mathieu Bouet, and Stefano Secci. Virtual network functions placement and routing optimization. In *2015 IEEE 4th International Conference on Cloud Networking, CloudNet 2015*, 2015. ISBN 9781467395014. doi: 10.1109/CloudNet.2015.7335301.

[32] Hendrik Moens and Filip De Turck. VNF-P: A model for efficient placement of virtualized network functions. In *Proceedings of the 10th International Conference on Network and Service Management, CNSM 2014*, 2014. ISBN 9783901882661. doi: 10.1109/CNSM.2014.7014205.

[33] Md Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, and Raouf Boutaba. On orchestrating virtual network functions. In *Proceedings of the 11th International Conference on Network and Service Management, CNSM 2015*, 2015. ISBN 9783901882777. doi: 10.1109/CNSM.2015.7367338.

[34] Mark Shifrin, Erez Biton, and Omer Gurewitz. Optimal control of VNF deployment and scheduling. In *2016 IEEE International Conference on the Science of Electrical Engineering, ICSEE 2016*, 2017. ISBN 9781509021529. doi: 10.1109/ICSEE.2016.7806110.

[35] Rami Cohen, Liane Lewin-Eytan, Joseph Seffi Naor, and Danny Raz. Near optimal placement of virtual network functions. In *Proceedings - IEEE INFOCOM*, 2015. ISBN 9781479983810. doi: 10.1109/INFOCOM.2015.7218511.

[36] Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol, Marinho Pilla Barcellos, and Luciano Paschoal Gaspary. Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions. In *Proceedings of the 2015 IFIP/IEEE International Symposium on Integrated Network Management, IM 2015*, 2015. ISBN 9783901882760. doi: 10.1109/INM.2015.7140281.

[37] R. Guerzoni, R. Trivisonno, I. Vaishnavi, Z. Despotovic, A. Hecker, S. Beker, and D. Soldani. A novel approach to virtual networks embedding for SDN management and orchestration. In *IEEE/IFIP NOMS 2014 - IEEE/IFIP Network Operations and Management Symposium: Management in a Software Defined World*, 2014. ISBN 9781479909131. doi: 10.1109/NOMS.2014.6838244.

[38] Sanghyeok Kim, Sungyoung Park, Youngjae Kim, Siri Kim, and Kwonyong Lee. VNF-EQ: dynamic placement of virtual network functions for energy efficiency and QoS guarantee in NFV. *Cluster Computing*, 2017. ISSN 15737543. doi: 10.1007/s10586-017-1004-3.

[39] Cheng Zhang, Jun Bi, Yu Zhou, Abdul Basit Dogar, and Jianping Wu. HyperV: A high performance hypervisor for virtualization of the programmable data plane. In *2017 26th International Conference on Computer Communications and Networks, ICCCN 2017*, 2017. ISBN 9781509029914. doi: 10.1109/ICCCN.2017.8038396.

[40] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. HyperVDP: High-Performance Virtualization of the Programmable Data Plane. In *IEEE Journal on Selected Areas in Communications*, 2019. doi: 10.1109/JSAC.2019.2894308.

[41] David Hancock and Jacobus Van Der Merwe. HyPer4: Using P4 to virtualize the programmable data plane. In *CoNEXT 2016 - Proceedings of the 12th International Conference on Emerging Networking EXperiments and Technologies*, 2016. ISBN 9781450342926. doi: 10.1145/2999572.2999607.

[42] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4Visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *CoNEXT 2018 - Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018. ISBN 9781450360807. doi: 10.1145/3281411.3281436.

[43] Mateus Saquetti, Guilherme Bueno, Weverton Cordeiro, and Jose Rodrigo Azambuja. VirtP4: An Architecture for P4 Virtualization. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019. ISBN 978-1-7281-3510-6. doi: 10.1109/IPDPSW.2019.00021.

[44] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*, 2014. ISBN 9781931971096.

[45] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *SOSP'09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, 2009. ISBN 9781605587523. doi: 10.1145/1629575.1629578.

[46] M. D. Ananth and Rinki Sharma. Cloud Management Using Network Function Virtualization to Reduce CAPEX and OPEX. In *Proceedings - 2016 8th International Conference on Computational Intelligence and Communication Networks, CICN 2016*, 2017. ISBN 9781509011445. doi: 10.1109/CICN.2016.17.

[47] John D. Day and Hubert Zimmermann. The OSI Reference Model. *Proceedings of the IEEE*, 1983. ISSN 15582256. doi: 10.1109/PROC.1983.12775.

[48] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 2008. ISSN 0146-4833. doi: 10.1145/1355734.1355746.

[49] Weverton Luis da Costa Cordeiro, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. Data Plane Programmability Beyond OpenFlow: Opportunities and Challenges for Network and Service Operations and Management. *Journal of Network and Systems Management*, 25(4), 2017. ISSN 10647570. doi: 10.1007/s10922-017-9423-2.

[50] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane. In *HotSDN 2013 - Proceedings of the 2013 ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013. doi: 10.1145/2491185.2491190.

[51] Yun Sen Xiao Lin, Jun Bi, Yu Zhou, Cheng Zhang, Jian Ping Wu, Zheng Zheng Liu, and Yi Ran Zhang. Research and Applications of Programmable Data Plane Based on P4. *Jisuanji Xuebao/Chinese Journal of Computers*, 42(11), 2019. ISSN 02544164. doi: 10.11897/SP.J.1016.2019.02539.

[52] Eric Keller, Soudeh Ghorbani, Matt Caesar, and Jennifer Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks, HotNets-11*, 2012. ISBN 9781450317764. doi: 10.1145/2390231.2390250.

[53] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 2015. ISSN 19324537. doi: 10.1109/TNSM.2015.2401568.

[54] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System Support for Elastic Execution in Virtual Middleboxes. *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, 2013. ISSN 00010782. doi: 10.1145/1327452.1327492.

[55] Wenyu Shen, Masahiro Yoshida, Taichi Kawabata, Kenji Minato, and Wataru Imajuku. VConductor: An NFV management solution for realizing end-to-end virtual network services. In *APNOMS 2014 - 16th Asia-Pacific Network Operations and Management Symposium*, 2014. ISBN 9784885522888. doi: 10.1109/APNOMS.2014.6996522.

[56] M He, A Basta, A Blenk, N Deric, and W Kellerer. P4NFV: An NFV Architecture with Flexible Data Plane Reconfiguration. In *2018 14th International Conference on Network and Service Management (CNSM)*, 2018. ISBN 2165-963X VO -.

[57] A. Lombardo, A. Manzalini, G. Schembra, G. Faraci, C. Rametta, and V. Riccobene. An open framework to enable NetFATE (Network Functions at the edge). In *1st IEEE Conference on Network Softwarization: Software-Defined Infrastructures for Networks, Clouds, IoT and Services, NETSOFT 2015*, 2015. ISBN 9781479978991. doi: 10.1109/NETSOFT.2015.7116179.

[58] Sameer G. Kulkarni, Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K. K. Ramakrishnan, Timothy Wood, Mayutan Arumaithurai, and Xiaoming Fu. Nfvnice: Dynamic backpressure and scheduling for NFV service chains. In *SIGCOMM 2017 - Proceedings of the 2017 Conference of the ACM Special Interest Group on Data Communication*, 2017. ISBN 9781450346535. doi: 10.1145/3098822.3098828.

[59] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C. Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using FlowTags. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014*, 2014. ISBN 9781931971096.

[60] Aaron Gember, R Grandl, and a Anand. Stratos: Virtual middleboxes as first-class entities. *ONS summit; TR*, 2013.

[61] Zafar Ayyub Qazi, Cheng Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. SIMPLE-fying middlebox policy enforcement using SDN. In *SIGCOMM 2013 - Proceedings of the ACM SIGCOMM 2013 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, 2013. ISBN 9781450320566. doi: 10.1145/2486001.2486022.

[62] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and implementation of a consolidated middlebox architecture. In *Proceedings of NSDI 2012: 9th USENIX Symposium on Networked Systems Design and Implementation*, 2012.

[63] Franco Callegati, Walter Cerroni, Chiara Contoli, and Giuliano Santandrea. Dynamic chaining of Virtual Network Functions in cloud-based edge networks. In *1st IEEE Conference on Network Softwarization: Software-Defined Infrastructures for Networks, Clouds, IoT and Services, NETSOFT 2015*, 2015. ISBN 9781479978991. doi: 10.1109/NETSOFT.2015.7116127.

[64] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. E2: A framework for NFV applications. In *SOSP 2015 - Proceedings of the 25th ACM Symposium on Operating Systems Principles*, 2015. ISBN 9781450338349. doi: 10.1145/2815400.2815423.

[65] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 1–15, 2018. ISBN 978-1-931971-43-0. URL https://www.usenix.org/conference/nsdi18/presentation/woo.

[66] The P4.org API Working Group. The P4.org API Working Group. https://p4.org/p4runtime/spec/v1.0.0/P4Runtime-Spec.pdf, 2016. [Online; accessed 22-January-2020].

[67] Charles Anderson. Docker, 2015. ISSN 07407459.

[68] Nijhuis, S.H. Simple_switch_grpc docker build, 2020. URL https://hub.docker.com/r/zjorsie/p4mn-docker.

[69] Nijhuis, S.H. Github repository - p4dpscaling, 2020. URL https://github.com/zjorsie/p4dpscaling.