INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

A Testbed for Researching Conflicts in Software-Defined Networks

Rosalie Kletzander

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Bachelor's Thesis

A Testbed for Researching Conflicts in Software-Defined Networks

Rosalie Kletzander

Aufgabensteller:	PD Dr. Vitalian Danciu
Betreuer:	PD Dr. Vitalian Danciu Tobias Guggemos Cuong Ngoc Tran
Abgabetermin:	15. Mai 2017

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 15. Mai 2017

. . . . . .
(Unterschrift der Kandidatin)

Zusammenfassung

Programmierbare Netze (SDN) werden als Beginn des ,,Softwarezeitalters der Netze" gepriesen, da sie durch die Bereitstellung einer Plattform, auf der leicht zu entwickelnde Anwendungen auf einem Controller laufen, modulares und dynamisches Netzmanagement ermöglichen. SDN Anwendungen implementieren genauso wie herkömmliche Geräte in traditionellen Netzen (z.B. Router, Firewalls), verschiedene Policies und dort wo mehrere Policies koexistieren, kann es zu Konflikten zwischen ihnen kommen.

Aktuelle Forschungsarbeiten in diesem Bereich beschäftigen sich hauptsächlich mit der Erkennung und Auflösung von Konflikten zwischen Policies, aber weniger mit den Konflikten selbst. Um Konflikte besser automatisiert erkennen und lösen zu können ist jedoch ein detailliertes Verständnis derselben unumgänglich.

Diese Arbeit beschreibt den Entwurf und die Implementierung eines Versuchsaufbaus zur Untersuchung von Konflikten zwischen Anwendungen in programmierbaren Netzen und deren Auswirkungen auf das Netz. Im Besonderen werden die Umstände untersucht, die dazu führen, dass zwei an sich funktionierende Anwendungen bei zeitgleicher Ausführung unvorhersehbare Ergebnisse liefern, welche die Policies des Netzes verletzen.

Hierfür wird ein Netzmanagementszenario entworfen, welches potentiell in Konflikt stehende Policies hervorbringt. Daraufhin wird eine Parameterstudie durchgeführt, um konfliktbegünstigende Parameter zu finden, welche dann in den Versuchsaufbau einfließen.

Die Ursachen und Auswirkungen aufgetretener Konflikte werden untersucht und Ansätze zur automatisierten Erkennung und Behebung derselben diskutiert. Die Resultate zeigen, dass der Versuchsaufbau es ermöglicht, unterschiedliche Parameter, die zu Konflikten führen können, systematisch zu testen und zu analysieren.

Abstract

Software-Defined Networks (SDN) are said to have ushered in the "software era of networking", enabling modular, dynamic network management by providing a platform for easy-to-develop applications running on a controller. Different applications, just like different devices in traditional networks, implement policies and wherever several policies coexist, there is a potential for conflict.

Current research in the area of conflicts between applications primarily focuses on detection and resolution, but little research actually examines the conflicts themselves. In order to satisfy the necessary requirement of automatic detection and resolution of conflicts in SDNs, a better understanding of possible conflicts must be developed.

A testbed is designed and implemented to closely examine conflicts between SDN applications and the consequences of these conflicts. Specifically, this involves examining the circumstances that lead two applications that function perfectly well individually, to create unpredictable results that break network policies when they are run concurrently.

A network management scenario is constructed, yielding policies that are likely to conflict, and a parameter study is conducted to determine what parameters could affect conflicts between applications. The results of the parameter study are used as a basis for the design of the testbed.

The causes and effects of occuring conflicts are analyzed, and possible conflict-specific solutions are suggested. The findings show that the testbed enables systematic testing and analysis of different parameters that are likely to cause conflicts.

Contents

1 Introduction

Software-Defined Networks (SDNs) are said to have ushered in the "software era of networking" [She11]. The device diversity and thus complexity of traditional networks is replaced by homogeneous forwarding devices that simply contain forwarding tables. The vendor-specific firmware, algorithms and policies of the former middleboxes, routers and switches are replaced by software applications running on a controller. These applications install rules on the forwarding tables of the devices, and as a result, network behavior can be controlled dynamically from one logically centralized point.

Since these applications are written in software, modular "programming" of networks becomes possible. For future networks, this could result in network providers buying applications from specialized vendors and combining them in order to fulfill different network policies. For example, a routing application could implement the policy of routing flows along the shortest known path, and a load balancing application could implement the policy of distributing packets evenly on all relevant ports. The potential to dynamically mix and match applications depending on the network's needs without an in-depth knowledge of the specific applications is a key asset of, and argument for, Software-Defined Networks.

Naturally, new technologies harbor new challenges. Applications implementing opposing policies can interfere with one another, creating conflicts and in the worst case, breaking network policies. For example, a routing application might want a packet to be output to port A (the shortest path) and a load balancing application might want the same packet to be output to port B (for even load distribution). Given this situation, with a policy-agnostic controller and equal execution priorities, the network would be incapable of fulfilling both policies simultaneously (shortest path forwarding *and* load distribution). The repercussions of failing to uphold a network policy can range from delayed packets to the collapse of the entire network.

In itself, the problem of conflicting network policies is not a new one. In traditional networks, different types of network devices have different functions and implement different policies. Unlike in SDNs, these functions are bound to specific devices that need to be painstakingly configured by hand, resulting in a very static network setup. Once devices have been physically placed in the network or configured in such a way that no policy conflicts occur, they cannot change their configuration or position autonomously and consequently, no new conflict will be able to appear unless the network is reconfigured. As a result, the problem of solving conflicts is as static as the corresponding network. Although this is convenient for network administrators, it comes at the expense of flexibility, which is becoming an ever growing sacrifice as the demands on modern networks are becoming more and more complex, requiring exactly the type of dynamic behavior that traditional networks are incapable of providing [GB14].

Flexible network behavior is exactly what SDNs promise to provide by dynamically distributing precomputed rules onto forwarding devices. As a result, forwarding devices no longer have static functionalities, meaning swift adjustments can be made and new developments and changes can be implemented with much less effort. The downside is that possible

conflicts between policies also have to be dealt with dynamically. Resolving the conflict between the routing and the load balancing application may seem trivial, but what happens if applications are started and stopped dynamically in a matter of seconds? Or if a policy exists that gives the shortest path routing application a higher priority for specific flows? It becomes impossible for a human administrator to solve these conflicts manually and intuitively. Consequently, automatic detection and resolution of conflicts will be necessary to ensure that new network configurations and policies work together effectively.

In the past, research in the area of conflicts between applications has gone mostly in the direction of detection and resolution [PSY+12] [KZZ+13] [KCZ+13], but little research actually examines the conflicts themselves. In order to satisfy the necessary requirement of automatic detection and resolution of conflicts in SDNs, a better understanding of possible conflicts must be developed.

The goal of this thesis is to design and implement a testbed to closely examine conflicts between SDN applications and the consequences of these conflicts. Specifically, this involves examining the circumstances that lead two applications that function perfectly well individually, to create unpredictable results that break network policies when they are run concurrently.

In order to design a productive testbed for examining the circumstances that lead to conflicts and the effects thereof, a management scenario is constructed, yielding policies that are likely to conflict. With the scenario as a starting point, a parameter study is conducted to determine what parameters could affect conflicts between applications. The results of the parameter study are used as a basis for the design of the testbed. Tests are then carried out on the testbed using an appropriate subset of the parameters gleaned from the study. The results are evaluated and the occurring conflicts analyzed.

The findings show that the testbed enables systematic testing and analysis of different parameters that are likely to cause conflicts.

The thesis is structured as follows: First, background information on SDNs and management policies is given and the concept of conflicts is examined, along with a review of current research (Chapter 2). The management scenario and the parameter study are described, along with the actual implementation of the testbed (Chapter 3). The results of the tests are analyzed and the observed conflicts are discussed, along with possible methods for automatic detection and resolution (Chapter 4). In the conclusion, the findings are summarized and their implications for further research in SDNs are considered (Chapter 5).

2 Background and Related Work

Since the topic of conflicting applications is inextricably tied to Software-Defined Networks and policy conflicts, their basic concepts are necessary for understanding how conflicts occur. The following chapter explains these fundamentals, and reviews related work on conflict detection in order to provide an overview of the types of conflicts that have already been examined, and the approaches that currently exist to detect or resolve them.

2.1 State of the Art

The fundamentals of Software-Defined Networks, which include the widespread SDN communication protocol OpenFlow, are explained in this section, along with the concept of network management policies, and policy conflicts.

2.1.1 Software-Defined Networks

Software-Defined Networks are a relatively new development in the design of networks. Traditional networks contain a multitude of different devices, such as load balancers, routers, switches, firewalls. These devices may be from different vendors and each one can require meticulous, vendor- or device-specific configuration. Not only is the configuration painstaking but it is also very static. Traditional network design makes changing configuration upon network events such as the arrival of certain packets very impractical, if not impossible.

A traditional (or "legacy") network device (e.g. router, switch, firewall) can be logically split into a data plane, a control plane and a management plane (Figure 2.1). The data plane implements the physical handling of packets (e.g. forwarding on a specific interface). The rules that the data plane uses are defined by the control plane. This could be a routing algorithm (e.g. Bellman-Ford) on a router or a port learning algorithm on a switch. The management plane provides an interface for configuration and monitoring. In traditional networks, the data plane and the control plane reside on the network devices. This makes devices very autonomous, to the point of not being able to coordinate with each other.

In contrast, SDNs extract the control planes from the different devices and implement them on the controller as applications. As a result, the network devices are homogeneous forwarding devices (or "SDN devices") with an interface for communication with the controller. They simply contain flow tables, where they match incoming packets and execute certain actions (Table 2.1) on them. The applications on the controller can install or modify rules on the forwarding devices. Consequently, the controller can change the behavior of the entire network at any given moment. It also has a centralized view of the network, enabling it to make more sophisticated decisions than traditional network devices that generally have no overview of the overall network state. An overview of a few popular open source controllers is given in Table 2.2.

Since SDNs use a "match/action" approach, where header fields of different OSI layers (e.g. MAC addresses, IP addresses, TCP/UDP ports) can be matched and modified in

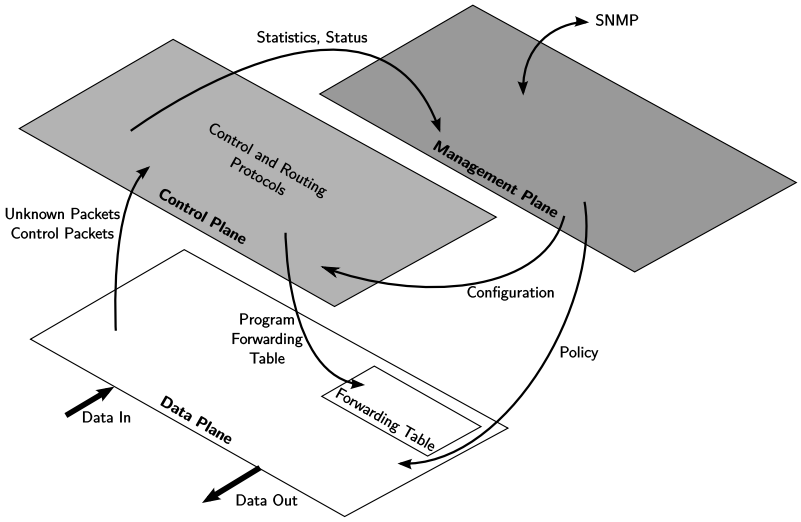

Figure 2.1: Data, control and management plane (from [GB14])

Action	Semantics
Output	forward packet to a specified port or the controller
Drop	drop packet
Group	process packet with group table (for multicast)
Modify	change specified header fields

Table 2.1: Basic packet processing actions ([Ope])

Controller	Information	Language	Current Version
NOX [GKP+08]	- developed by Nicira Networks - donated to research community in 2008 - known as the "first" SDN controller - "sibling" controller Pox (pure Python)	C++ (and Python)	"verity" (2014)
RYU [Ryu17]	- maintained by the open "Ryu Community" - framework for SDN application development	Python	4.13 (2017)
Floodlight [Pro16]	- Big Switch Networks controller - core of a commercial project by Big Switch Networks - developed by open community	Java-based	1.2 (2016)
OpenDaylight [Lin17a]	- Linux Foundation Collaborative Project - stakeholders such as Cisco, Intel	Java-based	"Boron", 5th release (2017)

Table 2.2: Popular open source controllers

a single table entry, the traditional model of the "layered network" (ISO/OSI Reference Model [Tan02]) is disregarded. This enables new possibilities for network design, for example successfully providing a single subnet containing loops and implementing shortest path forwarding on it (Section 3.3).

Another effect of the "match/action" approach is that SDNs, unlike traditional networks, do not operate on packets, but instead on flows, which are sets of packets defined by specific criteria (the fields that can be matched) between a source and a destination. By operating on flows, SDNs automatically enable communication-oriented service, as each packet of a flow can be treated equally, and different flows can be handled differently, according to their properties. For example, latency-critical flows (e.g. Voice-over-IP) can be favored over undemanding flows that do not have to arrive in real-time (e.g. email). As a result, Quality of Service (QoS) is dramatically less complicated to implement in SDNs compared to traditional networks. [KRV+14] [GB14]

2.1.2 OpenFlow

Since SDNs are a relatively new development, no standardization exists yet. Nonetheless, most SDNs utilize the widespread OpenFlow protocol for controller-device communication [KRV+14].

OpenFlow [Ope] devices (also known as OpenFlow Switches) contain one or more flow tables and a group table. The controller can install flow entries in these tables, each of which contains match fields, counters and a set of instructions.

When a packet arrives on a forwarding device, it is first processed by the first table. The packet header is compared to the flow table entries in the order of their priority, which can be specified by the responsible application. If it matches an entry, corresponding actions (Table 2.1) are executed. If the packet does not match any regular entry, it is passed on to the table miss entry. Common actions executed here are sending the packet to the controller, or passing it on to the next table. The packet is processed in this fashion in each table it is passed to until it is either forwarded to an output port, or ends up at the last table, where, lacking a match, it will be dropped. This pipeline is shown in Figure 2.2.

OpenFlow gives programmers a lot of freedom in respect to what kind of rules are installed where. As long as the specifications of the protocol are followed, programmers can configure the forwarding devices in any way they want. As a consequence, they must pay attention to programming their applications in such a way that rules do not conflict with one another, possibly breaking network policies.

2.1.3 Management Policies

Generally speaking, "a policy is a high-level overall plan embracing the general goals and acceptable procedures [...] of a governmental body"[1]. Policies are declarative, defining *what* a desired result is instead of *how* this result will be achieved. Management policies can exist on all levels of an organization, from overall goals down to policies in IT management and within that, policies in network management.

Sloman and Moffet [MS94] present a model for describing management policies in distributed systems. In this model, policies can provide either positive guidance or constraints for policy subjects (i.e. people whose responsibility it is to carry out actions to achieve the

[1] "policy." *Merriam-Webster.com*. Merriam-Webster, 2017. 28 April 2017

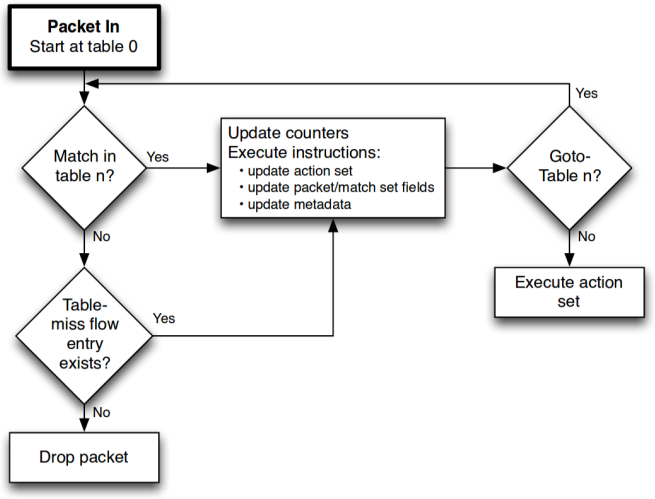

Figure 2.2: Flowchart of the OpenFlow pipeline (from [Ope])

policy goal). In a large organization there often is a policy hierarchy with high-level policies defining organizational goals and lower level policies specifying actions and goals required to fulfill the high-level policies. Policy subjects execute actions on target objects, which can be any object (e.g. configuration item of a device, header field of a packet) the policy subject influences to carry out policy goals.

Within an organization, there might be a policy that states that the organization must have an internet presence by establishing and maintaining a website. This is a high-level policy that will subsequently be split into lower-level policies that define actions that can be executed directly (Figure 2.3). For example, the organization's IT management would define a set of QoS requirements for the website and its resident network: the website must have high availability, and traffic must have average loss, delay and throughput. Network management might decide to fulfill these QoS requirements by overprovisioning, which could mean installing several servers and several paths, so that the likelihood of a server or a path being overloaded is lessened. In addition, if one of the servers or paths fails, another path or server can be used. Low-level network policies derived from the established requirements would then be to balance the load evenly on all paths and on all servers, maintaining the same server per flow so that stateful communication is made possible, and detecting failure of a component so that packets can be redirected.

Management of networks is a very complex task. Network management policies are abstractions which simplify this process. If the policies are clearly defined, planning actions to reach goals becomes more straightforward and evaluating whether or not they are being fulfilled becomes simplified. If human error is taken out of the equation, the only reason for policies to fail is if they do not harmonize. If this is the case, policy conflicts arise which can block the achievement of policy goals. [MS94]

2.1.4 Policy Conflicts

Since a main objective of this thesis is to examine network policy conflicts in SDNs, the question arises as to what a conflict actually is. Merriam Webster defines "conflict" as "a

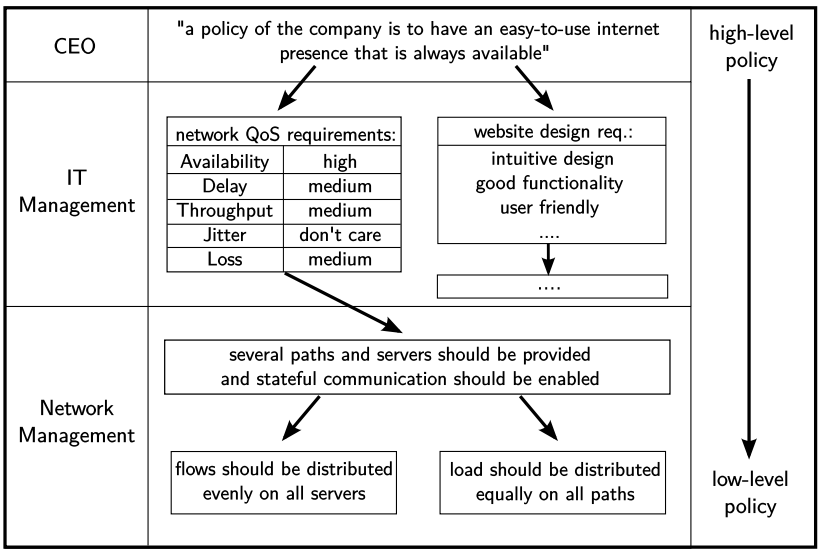

Figure 2.3: Policy hierarchy in an organization

competitive or opposing action of incompatibles"[2]. In the context of networks and network policies, policies can be "incompatible". To be incompatible or opposing, network policies must first have something in common, where the conflict originates. As Sloman and Moffett [MS94] show, policy conflicts can only occur when policies overlap (i.e. the intersection of the set of policy subjects or target objects is not empty).

A possible incompatibility of policies is exemplified in the following scenario of access control in a campus network. One network policy (A) states that students may not access certain areas of the network. Whereas, another policy (B) allows employees to access these same areas. Since some employees (e.g. teaching assistants) are also students, their traffic falls under both categories and as a result, the target objects of the two policies overlap (Figure 2.4 a). This creates a conflict, as following one policy would lead to dropping the same traffic that the other policy would allow.

Despite this potential for conflict, just because there is an overlap between polices does not mean that there will *always* be a conflict. Specifically, in the above example there might be another policy (C) that anonymously monitors student traffic. The target object of this policy (traffic generated by students) is the same as the target object of policy A mentioned above that blocks certain areas for students (Figure 2.4 b). In this case they do not conflict, since monitoring traffic does not modify it and thus, the first policy is not inhibited.

Clearly, policy conflicts may occur whenever more than one policy is active, and detecting and solving policy conflicts has been an issue since the advent of policies in network management (and even prior to that when "policies" were not yet labeled as such) [Ver02]. In contrast to how policy conflicts are dealt with in traditional networks, SDNs offer a different approach to conflict detection and solution.

[2] "conflict." *Merriam-Webster.com*. Merriam-Webster, 2017. 28 April 2017

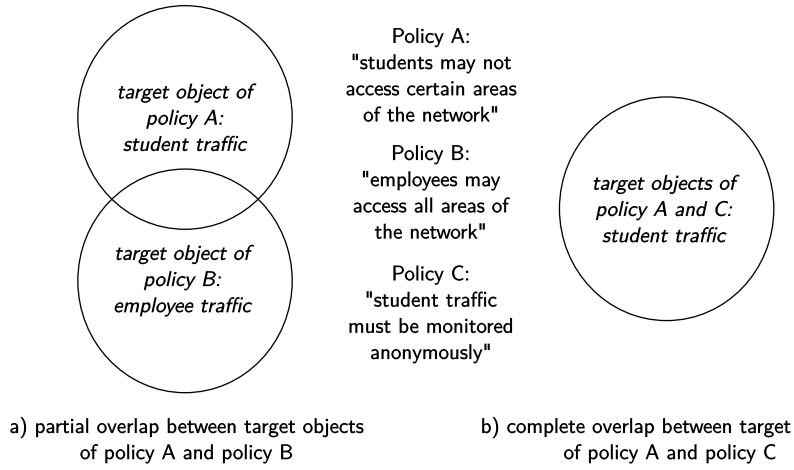

Figure 2.4: Overlapping policies in a campus network access control scenario

Policy Conflicts in Traditional Networks

In traditional networks, conflict detection and resolution is often carried out in an ad-hoc, manual way. In the example above, the conflict resolution is relatively straightforward: the policy allowing employees to access certain areas needs to overrule the policy blocking all students, so that teaching assistants can also access necessary areas. Although the conflict can be solved intuitively in this case, the actual implementation of the solution can be more complex. Different devices (e.g. firewalls or servers) need to be manually configured to carry out this decision, which requires in-depth knowledge of available devices and technologies [BAM09]. In addition, as soon as there is a change in policy (e.g. only *graduate* teaching assistants may access said areas), the whole process of reconfiguration starts from the beginning.

Solving policy conflicts in traditional networks by applying priorities can also be accomplished through the physical placement of devices. For example, by placing firewalls on the network edge, incoming traffic will always be processed by them first, before it is potentially forwarded on to a router or other network device.

However, as rearranging physical network componenents is not only time-consuming and impractical, but manually reconfiguring a large number of devices is also painstaking and error-prone, traditional networks suffer from being very static, as well as complex. On the flip side, because the implementation of policies is static, solving policy conflicts does not have to be dynamic either. In other words, once a conflict has been solved and the network reconfigured accordingly, no new conflicts will occur unless the network is reconfigured again.

Policy Conflicts in Software-Defined Networks

The situation of static configuration and policy drastically changes in SDN, where dynamic changes are a key feature and homogeneous forwarding devices can execute any rule that has been installed.

Apart from the kinds of network-wide conflicts that SDNs as well as traditional networks must deal with, another level of complexity is introduced in SDNs since devices are no

longer bound to specific functionalities. Specifically, any kind of rule can reside on any device in the network and thus, conflicts between different rules on a single device must also be resolved. These "device-local" conflicts can be dependent on the type of protocol used, which is generally some version of OpenFlow.

Device-Local Conflicts A device-local conflict occurs when a single SDN device is sufficient for breaking network policies. This can be caused in several ways.

As described in Section 2.1.2, an OpenFlow device contains several tables. If the controller provides no mechanism to explicitly allocate a specific table for a specific application, it is possible that several applications will conflict by writing their rules to the same table. The result is that a packet will match the flow rule with the highest priority and omit all the remaining rules installed by other applications.

Another type of device-local conflict stems from the order of the tables. In contrast to traditional networks, which implement the order of network functions by the physical position of the respective devices, in SDNs the order of the *tables* is also a determining factor for the order of application execution. For example, if a routing application and a firewall application install their rules on the same device in the first and second table, respectively, every packet matched by the routing application's rules in the first table will be forwarded directly to a port without ever being passed to the firewall application.

Although this is a simple conflict to solve for a human administrator, the manual resolution of conflicts is not really a satisfying solution for SDNs because it greatly inhibits the true flexibility that they are meant to provide.

Conflicts in Virtualized SDNs SDNs enable not only dynamic modifications of network behavior, but also network virtualization by slicing a physical network into several virtual ones [KRV+14]. This introduces another level of complexity and possible conflicts, as rules and policies of different slices can also conflict with one another. FlowVisor [SGY+10] introduces an architecture for slicing the network, while Natarajan et al. [NHW12] propose algorithms for detecting conflicts in such networks. Although these kinds of conflicts are relevant for some forms of SDNs, they are outside of the scope of this thesis and will not be discussed further.

Further Conflicts Conflicts in SDNs are as varied as the policies they originate from. As a result, it is difficult, if not impossible to find an exhaustive taxonomy to describe conflicts. However, the real potential of SDNs in reference to policy conflicts lies in the centralized role of the contoller. While policies in traditional networks have to be managed more or less by hand and device configuration is a distributed action, difficult or even impossible to manage in a consistent way, SDNs manage their devices homogeneously from a central point by design. As a result, policies are also implemented at a central point, which transforms the challenge of automatic detection and resolution of conflicts from a near impossible feat to a conceivable, albeit complex one.

2.2 Related Work on Detecting and Solving Conflicts in SDNs

Given this new possibility for dealing with conflicts in SDNs, many different approaches have been developed for detecting and handling conflicts. Most of these approaches are reactive, examining the state of the network after a new rule is installed in order to recognize whether a conflict has occured. The other approach is to detect conflicts proactively, analyzing new rules in the scope of existing ones before installing them.

2.2.1 Reactive Conflict Handling

Detecting conflicts reactively by analyzing the state of the network is a daunting task, as networks can be very complex and checking different invariants for the entire state can be time-consuming. However, there are approaches that make reactive conflict detection possible in real time, which is important if policies must be evaluated dynamically in a running network.

VeriFlow VeriFlow [KZZ+13] is a tool that resides between the controller and the network devices and verifies network-wide invariants in real time. It monitors all network update events as they occur (forwarding rules, application events, controller messages) and consequently can keep track of the current network state.

Since latency needs to be kept as low as possible to be able to work in real time, VeriFlow limits verification to the parts of the network that may be influenced by an update. VeriFlow makes use of the knowledge that policies must overlap to create a conflict and only verifies affected rules and packets.

To do this, it uses *equivalence classes*, sets of packets that will receive the same forwarding actions through the network. Using the information that rules need to overlap in some way for conflicts to happen, it checks overlap of the new rule and the old rules with the equivalence classes, hereby filtering out any rules or equivalence classes that are not relevant.

Once the filtering is complete, a graph is created, describing how each equivalence class would be forwarded through the network with the new rule in place. VeriFlow then checks pre-defined invariants, for example basic reachability, isolation of vlans, or more specific invariants such as whether packets always traverse a single point (for access control, etc).

When a problem is detected, VeriFlow can either block the new rule, or send out a warning that can then be handled by a human administrator. If the load becomes so large that VeriFlow is no longer capable of verifying in real time, it can no longer block rules as they are installed, but it can still send out warning messages.

NetPlumber NetPlumber [KCZ+13] is a real-time policy checking tool that verifies rule changes as they occur. It creates a "plumbing graph" that describes all possible paths of flows through the network. Nodes of the plumbing graph are the forwarding rules which can be linked with one another (rule A modifies header H and passes it to rule B). Whenever a rule is inserted or modified, the plumbing graph is updated accordingly.

"Probe Nodes" are nodes in the plumbing graph that check policy or invariants by monitoring flows received on a set of ports. To filter the flows that should be checked and thus reduce latency, the specially designed regular expression language "Flowexp" is used. The policies that can be checked are predefined by NetPlumber and are verified by methods such

as reviewing flow history, examining possible paths in the graph (connectivity, length), or sending probe flows.

It is unclear as to what happens when a policy or invariant check returns a negative result. In any case, no attempt is made to solve conflicts.

NICE NICE [CVP+12] (No bugs In Controller Execution) is a tool that tests for bugs in controller programs. It takes the controller program, the network topology, and a set of correctness properties (e.g. no forwarding loops or other predefined system states). It then tests the program by performing model checking (to systematically explore the space of system states) and symbolic execution (to systematically explore all code paths generated by different inputs). NICE checks the predefined correctness properties after every transition and if there is a violation, the tool records the execution trace so that the problem can be recreated.

Although NICE was developed to find bugs in controller programs and not conflicts between applications per se, it checks for correctness properties, just like VeriFlow and NetPlumber, which could be utilized to detect whether a conflict has happened.

2.2.2 Proactive Conflict Handling

In general, handling conflicts in a proactive way means analyzing factors that might *lead* to a conflict instead of examining the (simulated) network after the conflict has happened. The great advantage to this is that the network state need not be analyzed. However, proactive conflict detection also requires detailed knowledge of the circumstances that lead to a conflict. As a result, these approaches to conflict detection only cover certain types of conflicts.

The OpenDaylight Controller's "Group Based Policy Service" The OpenDaylight Controller [Lin17a] implements a "Group Based Policy Service" that provides a mechanism for detecting policy conflicts by analyzing policies for overlaps.

Policies (defined in "contracts") have different "Endpoint Groups" (participants), including a *provider* and *consumer* of a contract. Each Endpoint Group can be a part of several contracts simultaneously. Contracts that are "in scope" (have an overlap of Endpoint Groups) are searched for. This is reminiscent of Sloman and Moffet's description of policy conflicts (Section 2.1.4), where conflicts can only occur when there is an overlap of either policy subject or target object.

Each contract can be associated with a set of rules that contain *classifiers* and a set of *actions*, comparable to the OpenFlow match and action fields. "Policy Resolution" is carried out using hierarchies, as can be seen in the following excerpt from the "Group Based Policy User Guide" [Lin17b]:

> *"Rules, subjects, and actions have an order parameter, where a lower order value means that a particular item will be applied first. All rules from a particular subject will be applied before the rules of any other subject, and all actions from a particular rule will be applied before the actions from another rule. If more than [one] item has the same order parameter, ties are broken with a lexicographic ordering of their names, with earlier names having logically lower order."*

Incorporating conflict detection into controllers seems like an intuitive approach, as the controller has a centralized view of, and control over, the network. However, apart from OpenDaylight, most controllers have no knowledge of management policies, let alone contain mechanisms for dealing with policy conflicts ([GKP+08], [Ryu17], [Pro16]).

FortNOX FortNOX [PSY+12] is a software extension for the Nox controller that checks new rules against existing ones in real time. The goal is to prevent different rules (presumably originating from different applications) from interacting in such a way that not all policies are fulfilled.

For example, a controller could be running a firewall application that has a rule that blocks traffic from a certain address (10.0.1.4) and a second application whose rule changes the source addresses of incoming traffic from the same address (10.0.1.4). If the second application's rule changes the address 10.0.1.4 to some other address before the packet is passed to the firewall application's rule, the firewall's rule will let the packet through even though the packet should have been dropped according to network policy. As a result, the firewall policy is broken.

To prevent situations like this, where the modification of a packet by one rule results in the inability of another rule to match the same packet, FortNOX implements policy-based flow rule enforcement. It does this by converting all rules, including any new or modified rule, into a representation called *alias reduced rules* (ARRs) and then analyzing the ARRs for conflicts. ARRs are created by expanding a rule's match criteria by possible header modification results and wildcards. In the case of the application above, the rule set would be:

$$1 : a \rightarrow b : \ set\ (a \implies a')$$

$$2 : a' \rightarrow b : \ forward\ packet$$

and the derived rule would be:

$$(a, a') \rightarrow (b) : \ forward\ packet$$

The firewall application rule would be represented as

$$(a) \rightarrow (b) : \ drop\ packet$$

When the two sets are intersected, it becomes clear that the same packet is matched by rules with different actions. When this happens, FortNOX declares a conflict. To resolve conflicts, an application hierarchy is used. Some applications can be pre-defined as "security applications" and have a higher priority than "regular applications". When human administrators insert rules, they are given the highest priority. If two opposing rules have the same priority, the conflict can be solved by human intervention, otherwise the new rule overrides the previous one.

The procedure of creating alias reduced rules, analyzing them for conflicts, and resolving detected conflicts is carried out every time a flow rule is installed or modified, keeping the network free of policy conflicts between rules.

2.2.3 Summary

Both proactive and reactive approaches to detecting conficts have advantages and disadvantages. Reactive detection can verify policies without any knowledge of what causes conflicts, which is currently an asset since research on conflicts in SDNs is scarce. On the downside, apart from being very complex, the only way reactive mechanisms can react to conflicts is by blocking new rules as they have no knowledge about what is causing the conflict.

A better understanding of the circumstances that cause conflicts would be helpful for limiting necessary network state analysis. This is already implemented in VeriFlow, which is one of the reasons it works in real time. More in-depth knowledge of conflicts could make the process even more efficient.

Proactive conflict detection mechanisms, on the other hand, do have in-depth knowledge about the conflicts they are designed for. As a result, they can determine which applications are involved in the conflict, making conflict resolution possible. However, since proactive conflict detection is specific to a certain type of conflict, it is not a satisfactory solution for situations in which conflicts other than the specified ones occur.

Neither the reactive nor the proactive conflict detection mechanisms reviewed are completely sufficient for detecting and resolving conflicts. The reactive approaches are incapable of solving the conficts and the proactive approaches are not comprehensive enough. Both could greatly benefit from a better understanding of conflicts in SDNs.

Gaining a better understanding of what circumstances lead to conflicts between applications is one of the objectives of the testbed, the design and implementation of which is described in the following chapter.

3 Design and Implementation of the Testbed

The steps that were taken to create an appropriate testbed for researching conflicts between applications in SDNs are described in this chapter. First a network management scenario is constructed, which provides several network policies (Section 3.1). Using the scenario as a starting point, a parameter study is conducted, yielding parameters that could feasibly cause conflicts (Section 3.2). The implementation of the testbed based on the policies of the management scenario and the results of the parameter study is then described (Section 3.3).

3.1 A Network Management Scenario

The computer science institute of a university wants to hop on the bandwagon of SDNs and sets up a small test SDN infrastructure. The resulting network provides access control, basic connectivity and routing. It also connects several server farms that students and faculty can use for projects. The network is used regularly, and thus has the following important QoS requirements: high availability, and medium throughput and loss. This is achieved by overprovisioning on paths.

A group of researchers needs to use several servers for a research project that will generate a steady stream of traffic for its duration. The computations done on the servers need to be returned with minimum delay and the researchers would like to have their workload balanced evenly on all the servers. To keep any lines from failing due to too high load generated by the project, network management would like to have all traffic associated with the project balanced over several paths.

From these requirements, the following network management policies can be derived:

- "Traffic related to the project needs to be balanced evenly over all available *paths*."
- "Traffic related to the project needs to be balanced evenly over all assigned *servers*."

To fulfill these policies, two load balancing applications are implemented. One balances traffic over endpoints (EpLB), the other over paths (PLB).

The concept of load balancing over endpoints is that one "front endpoint" distributes workload across several servers according to some kind of load balancing policy. The front endpoint (in this case a forwarding device) needs some sort of information so that it knows which packets it needs to balance and which ones it can ignore. It then takes the necessary steps to ensure that the relevant packets are distributed. In the case of connection-oriented communication, all packets of the same flow must go to the same server. [Tan02]

In this case the identifier for traffic that needs to be balanced by the "frontend point" of the EpLB is an IP address, referred to as the "target" t. The "address space" E of the EpLB consists of t and the IP addresses of the servers that are balanced over. Packets that need to be balanced are addressed to t.

Load balancing over paths is similar to load balancing over endpoints in that a certain load balancing policy is followed to distribute load over resources. In this case the resources

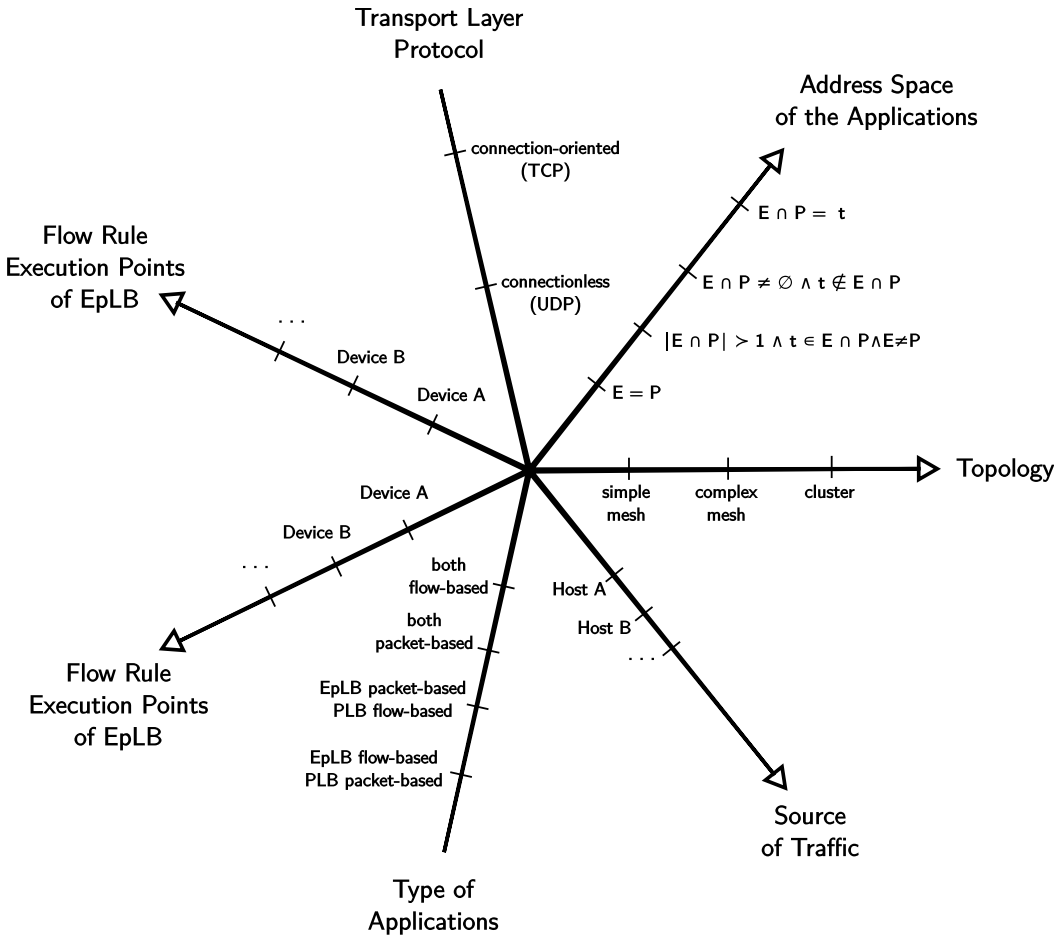

Figure 3.1: Possible dimensions

are several paths between the "load balancer" (in this case an SDN forwarding device) and the destination. Since only certain traffic should be load balanced (the traffic associated with the project), the PLB maintains an address space P that contains the IP addresses of the packets that need to be balanced.

This management scenario is used as a guideline for the parameter study in Section 3.2, and the actual implementation of the testbed (Section 3.3). It was chosen for the clarity of the policies and the fact that their target objects overlap, creating the grounds for observing conflicts.

3.2 Parameter Study

It is clear that conflicts between the two load balancing applications are possible, since their target objects (traffic related to the research project) are identical (Section 2.1.4). What remains to be seen is what circumstances (i.e. parameters) actually lead to conflicts. In the following section possible parameter dimensions are explored, an overview of which is shown in Figure 3.1.

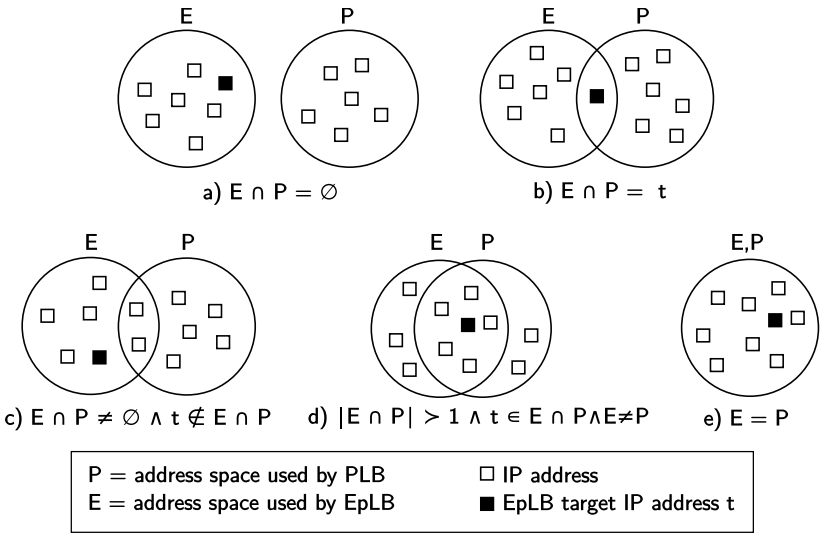

Figure 3.2: Address spaces of EpLB and PLB

Address Space of the Applications

As mentioned in Section 3.1, the EpLB and PLB both recognize the flows they need to perform their actions on by matching specific addresses. Since the target objects of the applications and thus, the address spaces overlap, it is possible that conflicts can occur. There are five different possible manifestations of the relationship between the address spaces, shown in Figure 3.2. Of these five possibilities, only the last four (b-e) are relevant, since the address spaces need to overlap for conflicts to be possible.

Application Execution Order

The order in which the applications are executed can be a cause for device-local conflicts (Section 2.1.4), therefore it is highly relevant. This includes processing order on the controller and the order of the tables on the forwarding devices.

Packet-based vs Flow-based Applications

There are two distinct types of connection service in networks: connection-oriented and connectionless [Tan02]. Connection-oriented services recognize flows and control them as a whole, whereas connectionless services are flow-agnostic and processes each packet separately. It is possible to implement applications so that they provide either a connection-oriented, or a connectionless service.

In order to provide a connection-oriented service, applications must be flow-based, meaning that for each new flow, they install table entries on devices. As a result, the same flow will always be processed by the same entries. The alternative is deploying packet-based applications, that do not install flows on devices, but always send their packets to the controller to be processed there. This results in a connectionless service, provided the controller does not maintain flow state internally.

Depending on the requirements of the traffic (connection-oriented protocols), changing this parameter in different applications could also create conflicts, for example if one appli-

cation providing a connectionless service breaks the connection-oriented service of another application.

Transport Layer Protocol

The transport layer protocol of traffic in the network (e.g. TCP, UDP) is a parameter that can affect network behavior, especially in the context of packet-based or flow-based applications. For example, if the EpLB is implemented as a packet-based application, it would make a difference whether a connectionless or a connection-oriented communication protocol is used. A connection-oriented protocol would try to set up a connection to a specific machine, and if the EpLB was a packet-based application, this would fail.

Flow Rule Execution Points

An interesting parameter to examine in light of the flexibility of SDNs is the flow rule execution point. Although applications can feasibly be executed (i.e. install rules and thus be "active") on any SDN forwarding device in the network, this is not necessarily desirable.

One reason for limiting the execution points of applications could be that some applications only make sense in certain areas of the network (e.g. firewall applications at the network edge). Alternatively, some devices could have more computing power and thus should have more tables and table entries to relieve other, less capable devices. It is also possible that the network contains both SDN forwarding devices and legacy devices (e.g. router, switch) that cannot change their behavior, thus limiting the points where an application can install its rules.

Conflicts could occur if one application's rules route traffic in such a way that it avoids another application's rules on a different device.

Source and Destination of Traffic

The source address of traffic could theoretically be any host in the network. For the EpLB to react to traffic, the destination address has to be the EpLB target. Since the source of traffic determines the path through the network to some degree, it may have an effect on whether a conflict occurs or not.

Topology

Since the topology of the network has an impact on certain parameters (e.g. flow rule execution point, number of paths through the network), it will most likely also influence whether applications will conflict or not. The range of different topologies is endless, from simple loop-free star topologies to manifestations of mesh topologies with various degrees of interconnectivity. For the PLB to be able to fulfill its policy, the network needs to have several paths from the source to the destination of traffic. Different mesh topologies could be relevant, as well as highly redundant cluster topologies.

Further Nonrelevant Dimensions

There are many more parameters that can be modified, such as the number of endpoint servers, or the number of servers the EpLB can balance on. The characteristics (size, number)

of packets that are sent by the packet generator are parameters that could be used to show the performance of the network under high load. However, the correct functionality of the applications is not dependent on these parameters, and thus they have no effect on the occurance of conflicts. As such, they will not be considered further.

3.3 Implementation

Now that a feasible management scenario for the testbed exists, and possible parameters have been examined, the testbed must be implemented. A base system is needed that realizes the network infrastructure in hardware or software. To be able to run an SDN, a controller and SDN forwarding devices are needed, along with several applications, and a traffic source and sink to create and receive packets, respecively. To be able to run tests and evaluate them, testing and evaluation tools are required.

3.3.1 System Architecture

Different possibilities of creating a testing environment exist, the closest to a "real" network being a physical network with forwarding devices and hosts in hardware. As this can be very expensive, virtual alternatives are often used. Different platforms for building virtual networks exist, for example Mininet [LHM10], which utilizes operating-system-level virtualization features to enable simulating networks with several hundred nodes on a single machine with constrained resources (e.g. laptop).

For this study, a different approach was used, specifically, a layered virtual infrastructure as described by Danciu et al. [DGK16]. An outer, physical machine (layer 0) serves as a hypervisor for a layer 1 virtual machine (VM). The layer 1 VM is also a hypervisor, running a group of VMs that represent the actual network. The advantage of using several layers of virtualization is that the virtual network is encapsulated in a further VM that can be migrated, duplicated, or restarted without interfering with other processes running on the hardware.

3.3.2 Simulated Network

For the testbed itself, the outer layer of virtualization (layer 0) is insignificant and the first layer is only relevant for managing and evaluating tests. The most important part is the simulated network within the second layer.

The design of the topology of a simulated network is greatly dependent on the type of applications that should be run, which in turn is dependent on the type of services are to be "offered" on the virtual network. The choice of topology could also be modeled after a real network for testing purposes.

In the case of the management scenario, the services offered, and thus applications implemented, require a certain type of topology. Since load balancing over paths should be implemented, it is obligatory that the network have several paths from one point to another. For the endpoint load balancing application to make sense, there must be at least two servers and one host to generate traffic as well. Other than that, the exact form of the topology is very open.

Since several paths are needed in the topology, it could be assumed that the network needs to be split into several subnets in order to prevent loops in broadcast domains. But

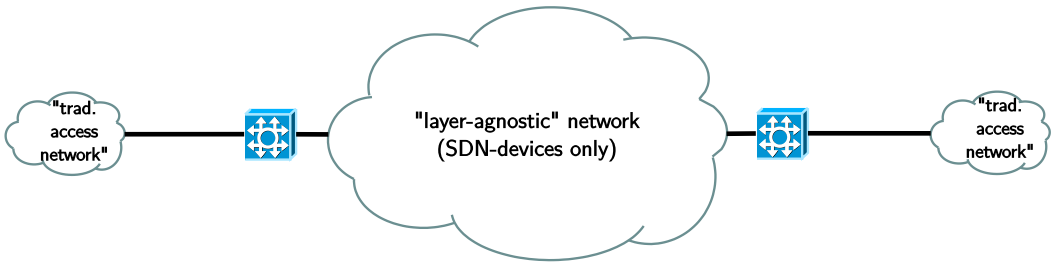

Figure 3.3: Concept of core and access networks

as this is not a traditional network and because SDNs enable "layer agnostic" networking (Section 2.1.1), it is possible to implement a given topology (even with loops) as a single subnet.

Nonetheless, some measures have to be taken for this type of topology to work smoothly, for although SDN devices do not need to function on the traditional OSI-layers, regular machines (e.g. servers, hosts) still do. To remedy this problem, the network can be logically split into one "layer-agnostic core network" and several "traditional access networks" (Figure 3.3). Theoretically, the traditional access networks could contain legacy switches and any traditional layer 2 topology (e.g. star, bus). As soon as packets are passed on to an SDN forwarding device, they enter the "layer-agnostic" area, where they can be processed with indifference to any traditional layer models.

The topology chosen for the testbed contains eleven VMs, comprising of:

- A controller (controller07). The Ryu controller (version 4.8) [Ryu17] was selected as it is designed as a framework for SDN applications, instead of as a monolithic controller, which is appropriate for testing different applications. It is also open source, well documented, and easy to install and use.

- Five SDN forwarding devices (ovs02, ovs03, ovs05, ovs10, ovs11). The OpenFlow-capable software switch Open vSwitch (version 2.6.2) [PPK+15] was used as SDN forwarding device software, as it is one of the only open source, free software switches.

- A traffic generator (traffic_generator04). The traffic generator sends UDP packets to a specified address. Although it uses UDP as its transport protocol, it can also send flows of packets, which is important for verifying whether the EpLB provides a connection-oriented service.

- Four servers. The servers are implemented as the counterpart of the traffic generator. They echo received packets back to the source, so that the EpLB can be verified in both directions.

The exact topology is shown in Figure 3.4. Apart from their connections to each other and to host machines, all the SDN forwarding devices are connected to a single bridge that is also connected to the controller. This facilitates controller-device communication and also traffic monitoring, as the "management traffic" does not mix with the traffic generated for testing purposes.

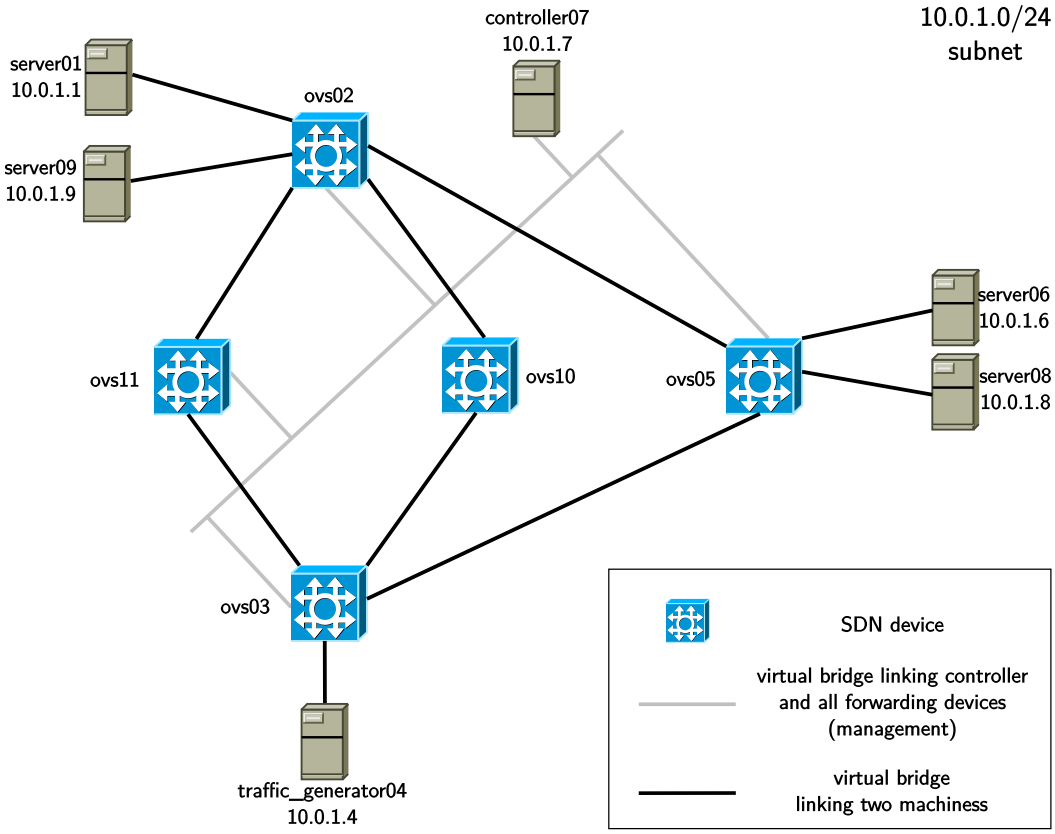

Figure 3.4: Topology of the testbed

3.3.3 Applications and Execution Context

Applications running on the Ryu controller can react to events, such as when a forwarding device connects to the controller. The applications implemented for the testbed install "base rules" on a preassigned table as soon as a new forwarding device has connected. Depending on the application, newly arriving flows are sent to the controller for processing, or passed on to the next table (signifying passing them on to the next application).

To prevent several applications from writing to the same table by accident (see Section 2.1.4), a configuration file was created for the testbed, assigning each application to a specific table.

Table Manager

The Table Manager (TM) is a simple application that clears all table entries from all forwarding devices when they first connect to the controller. This is convenient in a testing environment where the controller and forwarding devices are restarted often with different configurations, since no anomalies are generated by outdated rules. The TM does not install any rules itself and is only active at the very beginning of controller-device communication.

ARP-Responder

Although SDNs do not need to function on the traditional OSI-layers, regular machines (e.g. servers, hosts) still do. As a result, before sending an IP packet they broadcast ARP (Address Resolution Protocol) requests [Plu82] requesting the destination's MAC address (if the destination host is in the same subnet). In the case of the testbed topology, all machines are in the same subnet, but the forwarding devices disallow broadcasts in the "layer-agnostic network" because of loops. Without any further measures, the machine sending out the ARP request would never get a response and thus, no communication could take place at all.

This is where the ARP-Responder (ARP-R) comes in. It installs a table entry that matches any ARP-Request on each forwarding device. The matched ARP request is then sent to the controller, where an ARP response packet is built with the MAC address of the responsible forwarding device, implementing a kind of proxy ARP [CMQ87]. The ARP response is encapsulated and sent back to the forwarding device, which then sends it back out of the port it came in.

As a result, the source machine sends its IP packet to the forwarding device, which can then process it according to its "layer-agnostic" forwarding tables.

Shortest Path Forwarding Application

The Shortest Path Forwarding Application (SPFA) installs table miss entries that match any IP packet on each forwarding device. The first packet of a flow is sent to the controller, where the application utilizes a topology module to calculate the shortest path and get the relevant port the packet needs to be output on. It then installs a rule on the forwarding device, along with the instruction to forward the first packet out according the new rule. It forbids packets to be forwarded back out the way they came in, as this could lead to problems with endless loops.

With the ARP-R and the SPFA, basic connectivity throughout the network is achieved. These two applications are the base for the testing environment as both of them need to be

running for other applications to work. Thus, they will be viewed as the "execution context" for the applications that are tested.

Endpoint Load Balancing Application

As is described in Section 3.1, an endpoint load balancer balances flows fulfilling a certain criterium (e.g. the target address) over a number of servers.

This is relatively straightforward to implement in an SDN: the Endpoint Load Balancing Application (EpLB) installs a table entry that matches a target address (IP destination, TCP/UDP port). When a packet (the first of a flow) matches this entry, the flow needs to be balanced. The forwarding device then sends this packet to the controller, where the EpLB selects a server from an internal list, depending on its algorithm (in this case round robin). The EpLB then basically performs Network Address Translation [SE01], installing a rule that matches each packet of this flow and changes its destination address from the target address to the address of the previously computed server. It also installs a rule that changes the address back on the return journey so that the load balancing remains transparent (i.e. unseen) to the host.

In both cases, after the header is changed, the packet is passed on to the next table (and thus to the next application). As a result, the EpLB requires some sort of forwarding application (in this case the SPFA) later on in the pipeline.

Since the EpLB operates on flows, connection-oriented communication (i.e. packets of the same flow are always directed to the same server) is enabled.

Path Load Balancing Application

Like load balancing over endpoints, load balancing over paths distributes load across resources. In this case the resources are not servers, but paths through the network. The Path Load Balancing Application (PLB) installs a flow table entry that matches any IP packet and sends it to the controller. On the controller, the PLB checks whether the destionation address of the packet is in its target list (list of destination addresses where corresponding packets should be balanced). If the destination address of the packet is in this list, the PLB computes all the paths from the responsible forwarding device to the destination and selects one, using a round robin algorithm. It retrieves the relevant output port from the topology module and sends the packet back to the forwarding device to be output.

Sending every IP packet to the controller is not necessarily the most elegant approach as it creates a lot of additional traffic, although it is a straightforward way to implement packet-based path load balancing. Another possibility would be to implement flow-based path load balancing (install a flow table entry for each new flow). However, the results of flow-based load balancing can be more difficult to observe, since even distribution of flows over paths does not necessarily imply even distribution of packets. And since ease of observation is an asset in the area of testing, packet-based load balancing is implemented.

3.3.4 Testing and Evaluation Tools

After the necessary parameter space has been established, the testbed requires testing tools that iterate over the different dimensions, and evaluation tools that take the output of the tests and summarize it in a way that occuring conflicts become easily recognizable.

Automatic Testing

There are several requirements for an automatic testing tool in this testbed: it must be able to start and stop the controller along with the desired applications, it must be able to iterate over the given parameter space, and it must be able to monitor the network in such a way that conflicts can be identified.

The automatic testing tool is implemented in a modular way. It consists of two parts, the first part ("generic test") implementing a single test with certain parameters, and the second part ("autotest"), running the generic test numerous times, iterating over the specified parameters.

When run, the generic test starts the controller with the specified applications, along with all the echo servers and the traffic generator. In addition, it monitors the traffic on each possible path from the source to the destination for later evaluation of the PLB policy. The traffic generator and echo servers also produce logs to be able to evaluate the EpLB policy. Once the traffic generator has sent all its packets, all processes are stopped so that the next test can be run without inheriting any problems from previous tests.

Automatic Evaluation

The evaluation tool uses the logs created by the testing tools to examine whether the network policies were upheld. To verify the EpLB policy it checks the following properties:

- Every server in the server list was used for balancing.
- All servers received a balanced number of flows.
- All packets from the same flow were received on the same server.

Although the connectionless protocol UDP is used, verifying connection-oriented service is still possible, as one flow consists of several packets and the evaluation tool can check whether all of them arrived on the same server.

To check the PLB policy, the evaluation tool:

- counts the number of packets to each destination address on each path
- verifies whether packets with a target address from the PLB were evenly balanced

It also checks whether the traffic generator sent all the packets it was meant to, and notes how many of the response packets returned.

4 Evaluation and Results

In this chapter, results of the tests that were designed in Chapter 3 are evaluated and analyzed. Occuring conflicts are examined and their causes deduced. Implications of these conflicts are discussed, along with possible solutions.

4.1 Configuration of the Tests

Before the tests can be evaluated, it is important to clarify their exact configuration. This includes defining the execution context as well as selecting relevant parameters from the parameter space defined in the parameter study in Section 3.2.

4.1.1 Execution Context

During testing, the EpLB and the PLB are never the only applications running at one time. As was discussed in Section 3.3, the network needs the ARP-Responder and the Shortest Path Forwarding Application in order to provide connectivity. However, these applications are viewed as execution context and not considered in the evaluation, unless their role as a supplement to the EpLB and PLB has an effect on the outcome of the test.

4.1.2 Selection of the Parameter Space

As it would be highly impractical to test every single parameter in the parameter space, and is also not always necessary to do so, the dimensions described in Section 3.2 will be narrowed down to a select parameter space that is relevant for the testbed and can be reasonably examined (Figure 4.1).

Address Space of the Applications The different possibilities where the address spaces have a non-empty intersection (Figure 3.2 b-e) are highly relevant for testing conflicts. As a result, one example from each of the four types will be used for testing (Table 4.1).

Description	Configuration	Descriptor (from Fig. 3.2)
$E \cap P = t$ (Fig. 3.2 b)	EpLB Servers: (10.0.1.9, 10.0.1.8) PLB Targets: (10.0.1.1, 10.0.1.6)	b)
$E \cap P \neq \emptyset \wedge t \notin E \cap P$ (Fig. 3.2 c)	EpLB Servers: (10.0.1.1, 10.0.1.9, 10.0.1.6, 10.0.1.8) PLB Targets: (10.0.1.1, 10.0.1.9, 10.0.1.8)	c)
$\mid E \cap P \mid > 1 \wedge t \in E \cap P \wedge E \neq P$ (Fig. 3.2 d)	EpLB Servers: (10.0.1.1, 10.0.1.9, 10.0.1.6, 10.0.1.8) PLB Targets: (10.0.1.9, 10.0.1.6, 10.0.1.8)	d)
E = P (Fig. 3.2 e)	EpLB Servers: (10.0.1.1, 10.0.1.9, 10.0.1.6, 10.0.1.8) PLB Targets: (10.0.1.1, 10.0.1.9, 10.0.1.6, 10.0.1.8)	e)

Table 4.1: Parameters used for the address spaces

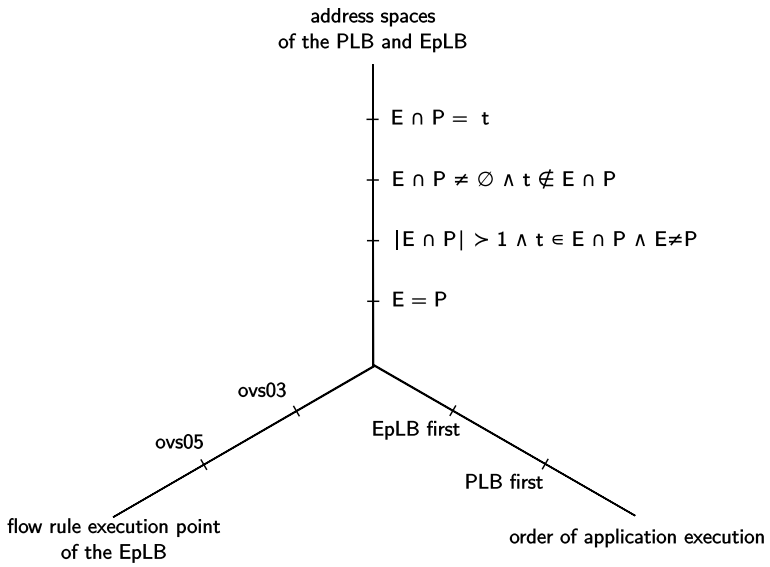

Figure 4.1: Dimensions selected for testing

Application Execution Order Both cases (EpLB first, PLB first) will be examined.

Packet-based vs Flow-based Applications Since one of the services of the network is to enable stateful communication on the servers (Section 3.1), the EpLB needs to be flow-based, balancing flows on servers instead of packets. The PLB is packet-based, since it is easier to evaluate evenly balanced packets than evenly balanced flows.

Flow Rule Execution Points The flow rule execution point of the EpLB has only one requirement: it must be somewhere where all traffic passes through on its way from the source to the target. If it is not, some flows will always be delivered to the target and never balanced (Figure 4.2 a). The possible flow rule execution points for the EpLB in the testbed topology are shown in Figure 4.2 b and c. It is only necessary to have one execution point, since as soon as the header is modified, it will no longer be matched by the EpLB's rules.

The flow rule execution point of the PLB is dependent on the topology. It only makes sense for the PLB to install rules where there are actually several paths to the target destinations. If there were only one path to the destination, the PLB could only choose the same path, which would not fulfill its policy. There also should be no more than one PLB execution point on any given path, otherwise random routing could occur, as shown in Figure 4.3 a. For the given topology, the only execution point that makes sense is ovs03 (Figure 4.3 b).

Further Parameters Parameters such as the number of flows and the number of packets per flow have no impact on conflicts so it is not necessary to change these during testing. Nonetheless, more than one flow should be sent to be able to test flow distribution on servers, and each flow should have more than one packet so that connection-oriented traffic (e.g. TCP) can be simulated. As a result, twenty flows with five packets each will be generated during each test, resulting in a total of 200 packets (including the echo packets). All the statically chosen parameters are shown in Table 4.2.

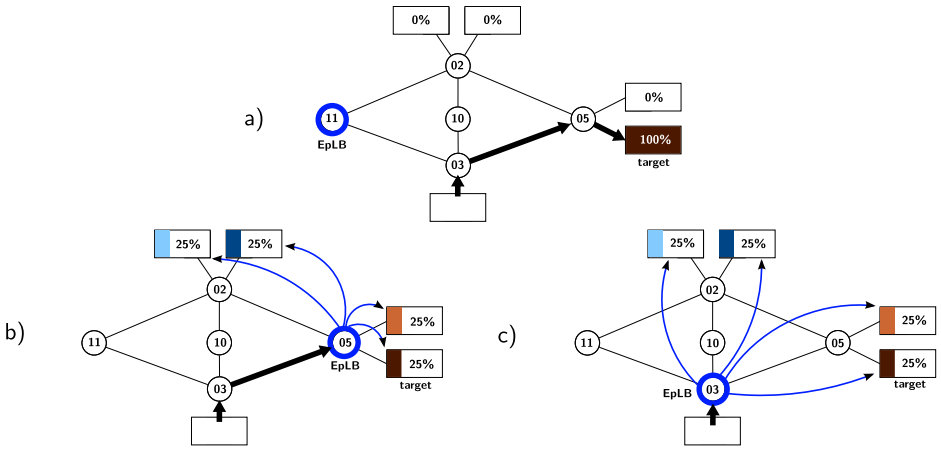

Figure 4.2: Possible Flow Rule Execution Points of the EpLB

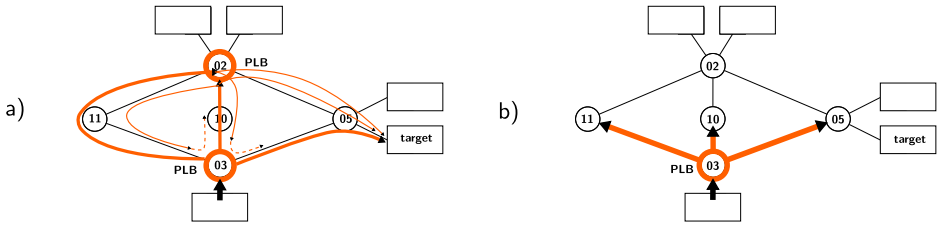

Figure 4.3: Possible Flow Rule Execution Points of the PLB

Dimension	Description	Parameter configuration
Source IP	host with several paths to destination	10.0.1.4
Destination IP	Target Server of EpLB	10.0.1.6
Number of flows	> 1	20
Number of packets per flow	> 1	5
Ex. Point for PLB	at path split/“crossroads”	“ovs03”

Table 4.2: Static parameters

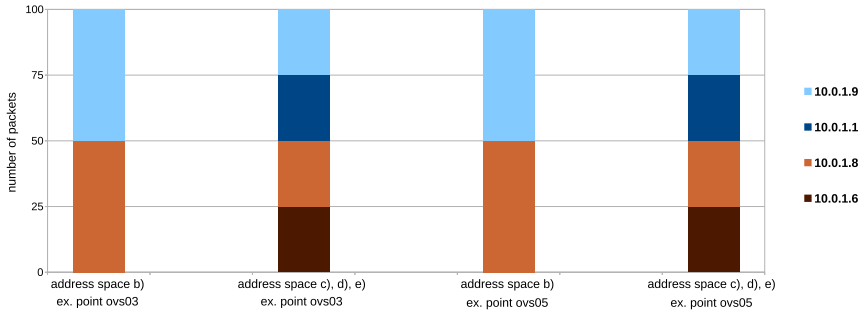

Figure 4.4: Benchmark case I: Endpoint Load Balancer

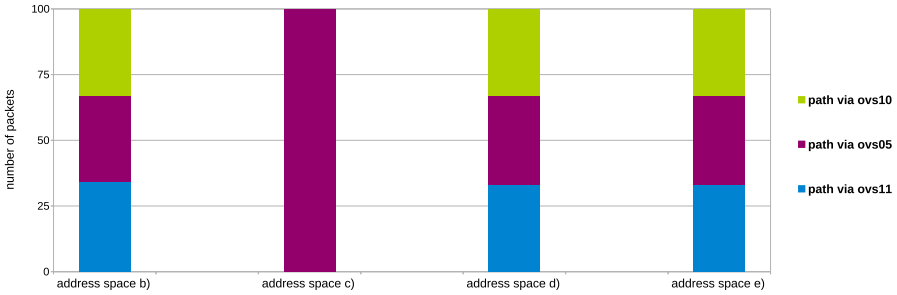

Figure 4.5: Benchmark case II: Path Load Balancer

4.2 Benchmark Cases I and II

To verify that the applications fulfill their policies when run individually and to provide a basis of comparison, the results of benchmark tests are evaluated. The parameter space used for the benchmark tests is smaller than the one used in the actual tests, as not all parameters are relevant when one application is run individually (e.g. order of application execution).

Benchmark Case I: Endpoint Load Balancer

Apart from omitting the application execution order in the benchmark test of the EpLB, the parameters are the same as the ones defined in 4.1. Specifically, different combinations of endpoint servers are used, and the flow rule execution point is changed.

As can be seen in Figure 4.4, packets are always evenly balanced over all available servers. The application upholds its policy in both cases of rule execution point placement, as well as all different cases of servers used. Although it is not visible in the image, connection-oriented communication is also ensured, as all the packets of a specific flow always arrive on the same server.

Benchmark Case II: Path Load Balancer

Since the flow rule execution point of the PLB is not changed, the only parameter that is tested in the PLB benchmark is the address space of the PLB. The results of these tests can be seen in Figure 4.5.

In three of these cases, the PLB upholds its policy by balancing all the outgoing packets

evenly on all paths. In one case however, packets are not balanced, as the destination address (10.0.1.6) is not in the PLB's address space. Nonetheless, this configuration will be tested in the case III tests, because although the target address is not in the address space of the PLB, there is still some overlap with the server addresses of the EpLB (Table 4.1).

4.3 Case III: EpLB and PLB

The iteration over the parameters defined in Figure 4.1 yields 16 separate tests. The results are shown in Figure 4.6. At first glance it becomes apparent that in only one of the 16 cases, the applications coexist in such a way that both policies are upheld (case 2.1, Figure 4.7). All other results show some kind of anomaly. In most cases, this results in either one *or* the other application fulfilling its policy, but one case exists where neither do (1.3).

Different effects of the occuring conflicts include: uneven balancing over paths (cases 2.3 and 2.7, Figure 4.8 i), complete failure to balance over paths (1.3, 1.4, 2.4, 2.5, Figure 4.8 ii), failure to balance over endpoints (1.1, 1.5, 1.7, Figure 4.8 iii), dropped packets (1.2, 1.6, 1.8, 2.2, 2.6, 2.8, Figure 4.8 iv), and failure to direct all packets of a flow to the same endpoint (1.3).

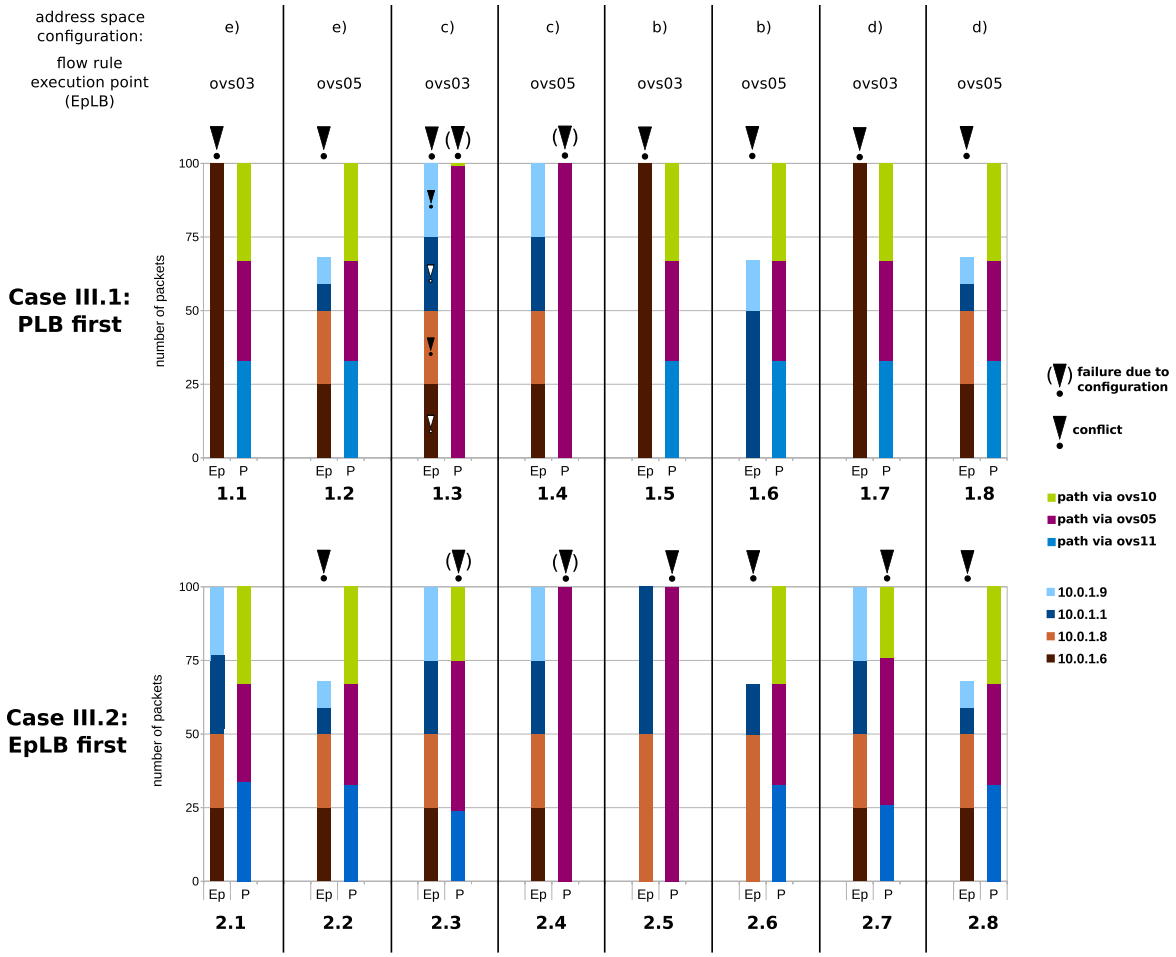

Figure 4.6: Test case III: EpLB and PLB

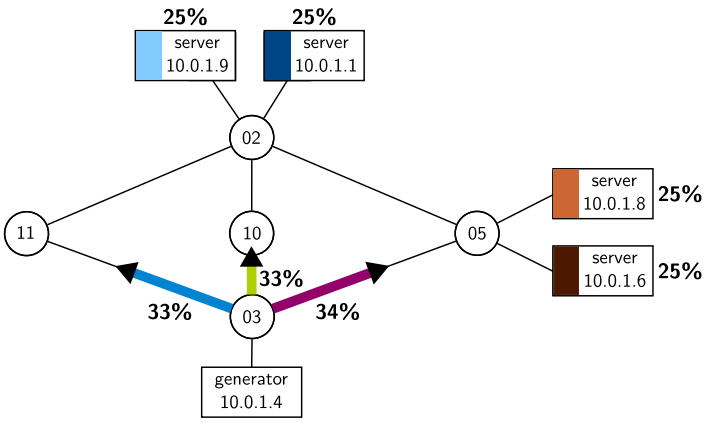

Figure 4.7: Successful execution of both applications (case 2.1)

4.4 Analysis

The parameter space examined has three dimensions: the flow rule execution point of the EpLB (either ovs03 or ovs05), the address space of the applications (one example per variation in Figure 3.2), and the order of application execution (PLB first, EpLB first). The upper row in Figure 4.6 shows all the tests with the PLB first and the EpLB second, and the lower row the opposite order.

It is significant that the results of x.2, x.4, x.6 and x.8 (even numbers) are identical. In these tests the flow rule execution points of the applications are on different devices. This implies that the order of application execution is insignificant when applications are executed on different devices. Inversely, since the results vary when the the applications *are* executed on the same device, this means that in these cases the execution order is significant.

4.4.1 Flow Rule Execution on the Same Device

To identify how the order of the applications along with the address spaces of the applications (E = address space of the EpLB, P = address space of the PLB) have an effect on whether or not a conflict occurs, the possible course of events is examined first for the case "EpLB first", and then for the case "PLB first". Every incoming packet has the destination IP address A of the EpLB target t.

EpLB First (Figure 4.9) When a packet arrives on the SDN device, it is processed by the EpLB table first. Since A = t, the EpLB table matches the packet, and changes the IP address to A'. The packet is then passed on to the PLB table. The PLB is a packet-based application, therefore it sends the packet to the controller where the it is processed by the PLB. Here the address space of the applications is the crucial factor in the occurance of conflicts:

- If $A' \in P$, the PLB matches the packet and forwards it to the port the PLB has computed. As a result, both applications fulfill their policies. This situation can be observed in case 2.1 (Figure 4.6).
- If $A' \notin P$, the PLB ignores the packet and passes it along the pipeline where it is

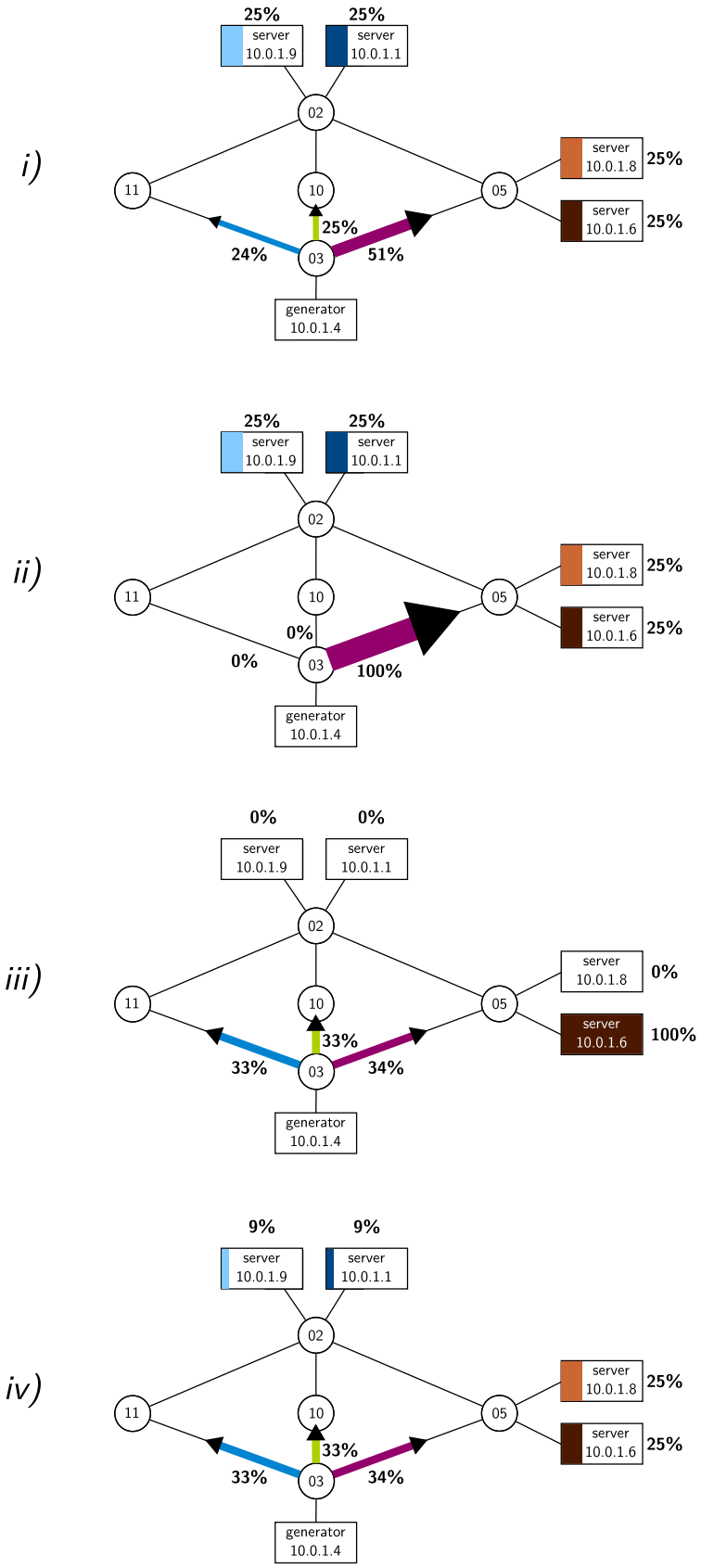

Figure 4.8: Examples of different conflicts

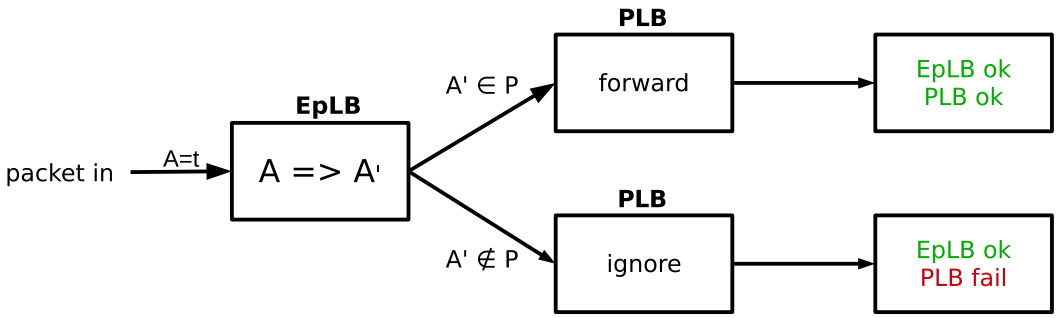

Figure 4.9: Packet processing pipeline: EpLB first, execution on the same device

forwarded by the SPFA. As a result, the EpLB fulfills its policy, but the packets all take the shortest path, breaking the PLB's policy. In case 2.5 the modified address A' is never ∈ P, resulting in no packets being balanced (Figure 4.8 ii). In cases 2.3 and 2.7, A' is sometimes ∈ P, depending on the server the EpLB has selected. The result is that some of the packets are balanced (Figure 4.8 i).

PLB First (Figure 4.10) In the cases where the PLB is executed first, the PLB table processes an arriving packet first. The packet is immediately sent to the controller, where it is passed to the PLB. Once again, the address space of the PLB affects the outcome:

- If $A = t \in P$, the PLB sends the packet back to the SDN device, along with the instruction to forward it according to the PLB's policy. The packet is immediately forwarded, omitting the EpLB entirely. As a result, the PLB's policy is fulfilled, but no endpoint load balancing takes place. This is the case in 1.1, 1.5, and 1.7.
- If $A = t \notin P$, the PLB passes the packet on to the EpLB. Since A = t, the EpLB matches the packet and assumes that it is the first packet of a flow, because according to the EpLB, the only packets sent to the controller are the packets that do not match an entry. The EpLB then installs an entry on the SDN device that matches the "new" flow and sends the packet back to the device, along with the instructions to modify the header according to the new table entry. The problem with this course of events is that *every* packet is sent to the controller, not just the first of a flow. As a result, although the EpLB distributes the packets evenly to the servers, it does not recognize flows, and breaks the part of its policy that promises connection-oriented service. Since $A = t \notin P$, the packets are not balanced by the PLB, as the target is not in the address space. This is not a conflict, as the same thing happens when the PLB is run individually (Figure 4.5). This situation is exemplified in case 1.3.

4.4.2 Flow Rule Execution on Different Devices

Now that the cause of the conflicts in the cases of flow rule execution on the same device have been examined, the remaining cases (Figure 4.11) are analyzed. In these cases, the PLB flow rule execution point is "ovs03" and the EpLB flow rule execution point is "ovs05".

Case x.4 is fairly straightforward to analyze. Packets arrive on ovs03 first, where they are passed to the PLB. Since $t \notin P$, it is impossible for the PLB to match any packet. As a

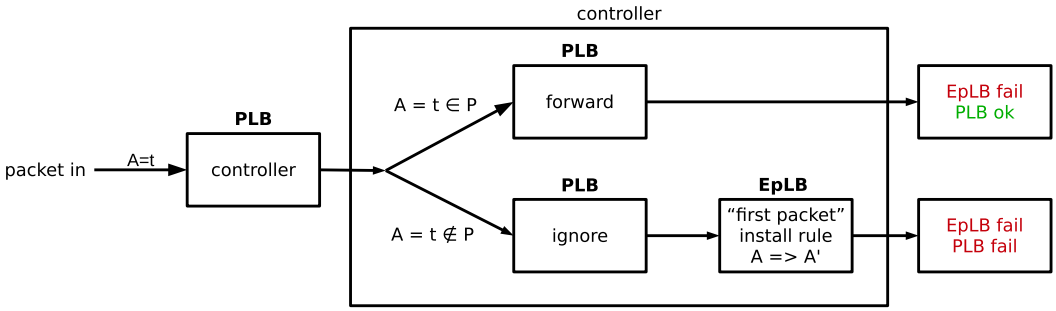

Figure 4.10: Packet processing pipeline: PLB first, execution on the same device

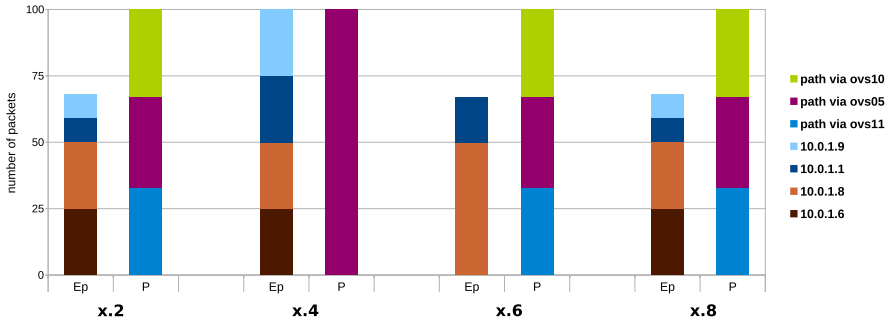

Figure 4.11: Cases of flow rule execution on different devices

result, the PLB ignores the packets and no path load balancing takes place. Like in case 1.3, this is not a conflict, as the same thing happens if the PLB is run individually. The packets are then forwarded by the SPFA to ovs05, where the EpLB modifies the address.

The fact that packets are dropped in the other three cases is striking, as it is not an effect of any of the other conflicts. Upon closer examination, it becomes clear that this is actually not a conflict between the EpLB and the PLB. When packets are balanced by the PLB, they take the paths via ovs11, ovs10, and ovs05. The packets that are forwarded via ovs05 are handled as expected. The packets that take paths via ovs10 and ovs11 are then forwarded by ovs02 towards ovs03 and then to ovs05. When they arrive on ovs05, they are processed by the EpLB, which modifies their addresses. Some of these packets are then forwarded on to the servers that are directly connected to ovs05. This is successful, as is visualized in Figure 4.11: server_06 (10.0.1.6) and server_08 (10.0.1.8) receive all the packets they should. However, if the EpLB modifies the address to that of one of the servers connected to ovs02, when the packet is passed to the SPFA it is dropped due to the fact that the SPFA disallows sending a packet out by the port it arrived on (Figure 4.12). This is not a conflict between the EpLB and the PLB, but between the EpLB and the SPFA.

4.5 Findings

One of the first distinctions made in the analysis was that the configured application execution order has no effect on the outcome of the test when applications are not executed on the same device. This implies that the configured application order is a *device-local* characteristic.

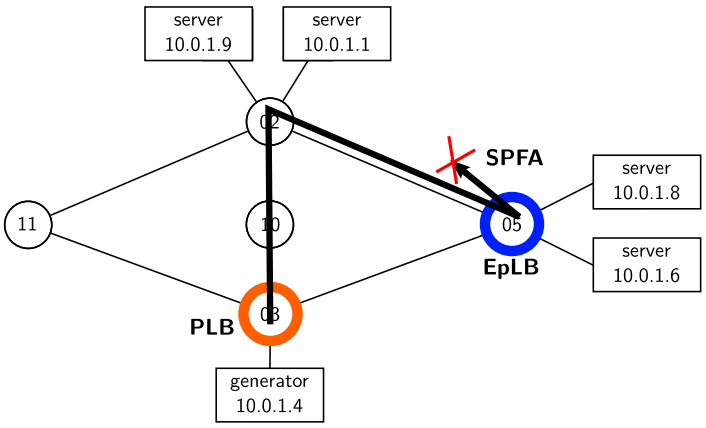

Figure 4.12: Conflict between the EpLB and the SPFA

However, when applications are executed on different devices, they are executed in a specific order as well, defined by the position of the devices in the topology. For example, in case 2.4 the EpLB is executed on ovs05, and the PLB on ovs03. When packets pass through the network, they are always handled by the PLB first.

The parameter "flow rule execution point" defines where applications are executed in the network. In order to better understand the implications of this parameter when examining two applications for possible conflicts, it can be represented by the combination of "application proximity" (if the applications are executed on the same device or on different devices) and "global application order" (if they are on the same device, what the configured order of execution is, and if they are on different devices, what the topological order of execution is).

The effects of using this abstraction to describe the tests that were carried out on the testbed are shown in Figure 4.13. The observed conflicts can be described by four general cases:

I Application A modifies the address of a packet so that it is no longer in the address space of application B (device-local).

II Application A forwards the packet before application B can process it (device-local).

III Application A sends the packet to the controller, where it is also unintentionally processed by application B (device-local).

IV Further conflict not related to the EpLB and the PLB:
Application A modifies the address of a packet in such a way that causes application B to drop it.

Unlike conflict types I – III, conflict type IV is not a result of the parameters shown in the graph, as it is a conflict between the "execution context" SPFA and the EpLB. This shows that applications running in the execution context can have an influence on the applications being tested in the testbed.

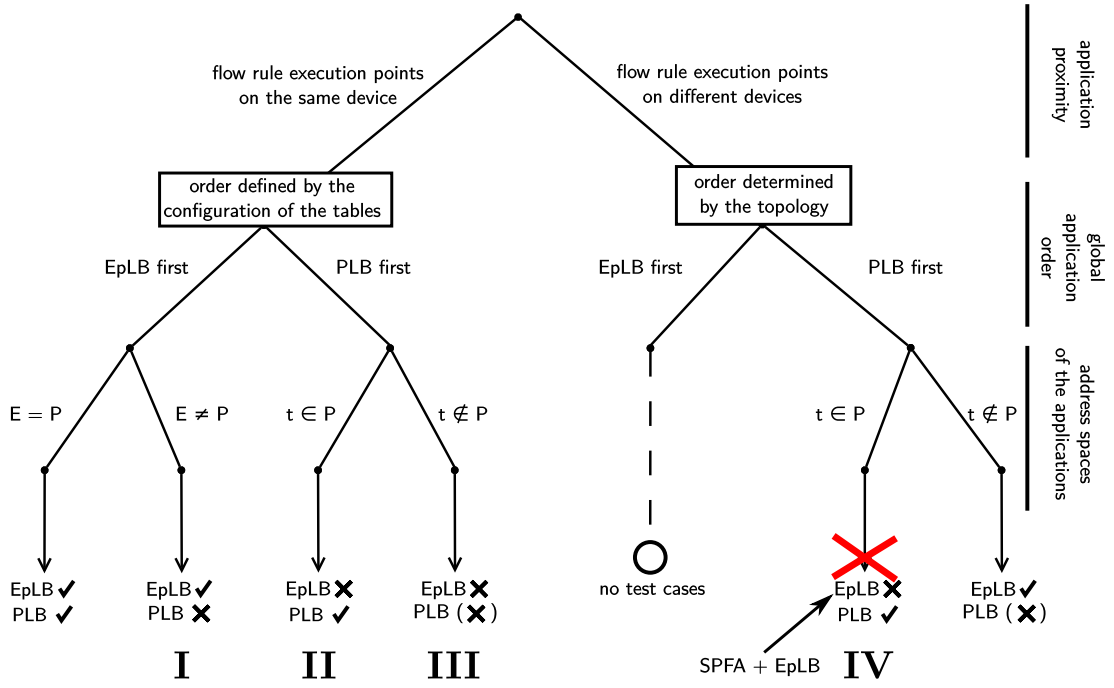

Figure 4.13: Visualization of occurring conflicts in the testbed

4.6 Discussion

The close analysis of the conflicts discovered in the testbed makes it possible to deduce their cause and to discuss possible methods of detection and in some cases, resolution.

Conflict type I can apply to a number of different applications, as it is not related to the addresses themselves, but to the fact that applications use values such as the address to match packets. As a result, conflict type I can be generalized to "Application A modifies a *field* of a packet so that it is no longer matched by application B". This kind of conflict has already been addressed by FortNOX (Section 2.2.2), which proposes a mechanism for detection and solution that could feasibly be implemented for other controllers, as well.

Conflict type II is a device-local conflict that occurs due to the fact that the table with the forwarding rule was not at the end of the pipeline. This problem involves the device-local order of the applications, which, at the moment, must be configured by a human administrator or programmer. Leaving such a task up to manual configuration is not an optimal solution, as SDNs are dynamic in nature, possibly demanding the reordering of applications faster than a human administator could keep up with. A possibility for automization could be a more sophisticated controller that is capable of reordering table entries in such a way that applications with forwarding actions are placed at the end of the pipeline.

Conflict type III is caused by the difference in the types of service the applications offer. The PLB offers a connectionless service, operating on packets, whereas the EpLB operates on flows. In general, SDNs operate on flows and not on packets, making this specific scenario unusual. As a result it is unlikely that an application will require every packet to be sent to the controller. However, it is possible that some applications require more than just the initial packet of a flow to be sent to the controller, for example for monitoring reasons. In this case, the conflict that was observed could occur. A more sophisticated controller might

be able to prevent this conflict from happening by maintaining some sort of state where it can deduce whether a packet is the first of a flow or not, and pass it to relevant applications accordingly.

Since conflict type IV is not a conflict between the EpLB and the PLB, the only possibility of detecting it without any further knowledge is by using reactive policy conflict handling tools such as VeriFlow or NetPlumber (Section 2.2.1).

Figure 4.13 clearly visualizes which further test cases need to be investigated in order to show the effects of the topological order of the two tested applications on the outcome of a conflict. Specifically, this requires a test setup in which the EpLB flow rule execution point can be on a device that processes packets first, and in which the SPFA does not conflict with the EpLB, inhibiting the observation of a possible conflict between the PLB and the EpLB. Given the nature of the system architecture of the testbed (Section 3.3), it is possible to extend the topology in such a way that these cases can also be examined.

5 Conclusion and Future Work

The advantages that Software-Defined Networks offer by enabling dynamic network behavior come at the price of having to dynamically handle network policy conflicts, as well. To date, there is scant research on conflicts in SDNs and the little research that has been done has focused mostly on how to recognize that a conflict has occurred based on the examination of the network state, instead of on the causes of conflicts.

In order to better understand the causes and effects of conflicts in SDN, a testbed was created to examine the circumstances that can lead two SDN applications that function perfectly well individually, to create conflicts when run simultaneously. This was done by constructing a management scenario, and using the resulting policies as a basis for examining parameters that could be responsible for creating conflicts. The testbed was implemented in such a way that a selected subset of these parameters could be systematically tested. The conflicts observed in these tests were then analyzed.

The testbed provided a context where the occurring conflicts as well as the circumstances that lead to them, could be analyzed. The understanding of possible parameters that lead to conflicts enabled discovering the abstract representation of the "flow rule execution point". With this abstraction, it was possible to identify additional cases that would be worth testing in future work.

As the parameters explored were partly application specific (e.g. address spaces), and the tests covered a select parameter space, it would be worthwhile to utilize the developed testbed in a modified form to conduct more tests with other applications and parameters.

Even though the number of applications were limited, and only a select parameter space was tested, several different kinds of conflicts were observed. Since different applications have different parameters, and interact in different ways, it becomes apparent that the potential for conflicts between applications is enormous.

Nevertheless, the results of this thesis are a step in the direction of building a foundation for the automatic detection and resolution of conflicts, which will eventually enable SDNs' full potential to unfold.

List of Figures

Bibliography

[BAM09] Benson, Theophilus, Aditya Akella and David A Maltz: *Unraveling the Complexity of Network Management.* In *NSDI*, pages 335–348, 2009.

[CMQ87] Carl-Mitchell, Smoot and John S. Quarterman: *Using ARP to implement transparent subnet gateways.* RFC 1027, RFC Editor, October 1987. `http://www.rfc-editor.org/rfc/rfc1027.txt`.

[CVP+12] Canini, Marco, Daniele Venzano, Peter Perešíni, Dejan Kostić and Jennifer Rexford: *A NICE Way to Test OpenFlow Applications.* In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 127–140, San Jose, CA, 2012. USENIX.

[DGK16] Danciu, Vitalian A., Tobias Guggemos and Dieter Kranzlmüller: *Schichtung virtueller Maschinen zu Labor- und Lehrinfrastruktur [Layering of Virtual Machines for Lab and Teaching Infrastructure].* In *9. DFN-Forum - Kommunikationstechnologien, 30. Mai - 1. Juni 2016, Rostock, Germany*, pages 35–44, 2016.

[GB14] Goransson, Paul and Chuck Black: *Software Defined Networks: A Comprehensive Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[GKP+08] Gude, Natasha, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown and Scott Shenker: *NOX: Towards an Operating System for Networks.* SIGCOMM Comput. Commun. Rev., 38(3):105–110, July 2008.

[KCZ+13] Kazemian, Peyman, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown and Scott Whyte: *Real Time Network Policy Checking Using Header Space Analysis.* In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 99–112, Berkeley, CA, USA, 2013. USENIX Association.

[KRV+14] Kreutz, Diego, Fernando M. V. Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky and Steve Uhlig: *Software-Defined Networking: A Comprehensive Survey.* CoRR, abs/1406.0440, 2014.

[KZZ+13] Khurshid, Ahmed, Xuan Zou, Wenxuan Zhou, Matthew Caesar and P. Brighten Godfrey: *VeriFlow: Verifying Network-Wide Invariants in Real Time.* In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 15–27, Lombard, IL, 2013. USENIX.

[LHM10] Lantz, Bob, Brandon Heller and Nick McKeown: *A Network in a Laptop: Rapid Prototyping for Software-defined Networks*. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[Lin17a] Linux Foundation: *OpenDaylight Boron Documentation*. `http://docs.opendaylight.org/en/stable-boron/index.html#`, 2017.

[Lin17b] Linux Foundation: *OpenDaylight Boron Documentation, Group Based Policy User Guide*. `http://docs.opendaylight.org/en/stable-boron/user-guide/group-based-policy-user-guide.html`, 2017.

[MS94] Moffett, Jonathan D. and Morris S. Sloman: *Policy Conflict Analysis in Distributed System Management*. Journal of Organizational Computing, 4(1):1–22, 1994.

[NHW12] Natarajan, S., X. Huang and T. Wolf: *Efficient Conflict Detection in Flow-Based Virtualized Networks*. In *2012 International Conference on Computing, Networking and Communications (ICNC)*, pages 690–696, Jan 2012.

[Ope] Open Networking Foundation: *OpenFlow Switch Specification, Version 1.3.1*. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf`.

[Plu82] Plummer, David C.: *Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit Ethernet address for transmission on Ethernet hardware*. STD 37, RFC Editor, November 1982. `http://www.rfc-editor.org/rfc/rfc826.txt`.

[PPK+15] Pfaff, Ben, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon and Martín Casado: *The Design and Implementation of Open vSwitch*. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association.

[Pro16] Project Floodlight: *Floodlight Documentation*. `https://floodlight.atlassian.net/wiki/display/floodlightcontroller/For+Developers`, 2016.

[PSY+12] Porras, Philip, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson and Guofei Gu: *A Security Enforcement Kernel for OpenFlow Networks*. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 121–126, New York, NY, USA, 2012. ACM.

[Ryu17] Ryu SDN Framework Community: *Ryu 4.13 Documentation*. `http://ryu.readthedocs.io/en/latest/index.html`, 2017.

[SE01] Srisuresh, P. and K. Egevang: *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022, RFC Editor, January 2001.

[SGY+10] Sherwood, Rob, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown and Guru Parulkar: *Can the Production Network Be the Testbed?* In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 365–378, Berkeley, CA, USA, 2010. USENIX Association.

[She11] Shenker, Scott: *The Future of Networking and the Past of Protocols.* Open Networking Summit, 2011.

[Tan02] Tanenbaum, Andrew: *Computer Networks.* Prentice Hall Professional Technical Reference, 4th edition, 2002.

[Ver02] Verma, D. C.: *Simplifying Network Administration Using Policy-Based Management.* IEEE Network, 16(2):20–26, Mar 2002.