Data-Centre Traffic Optimisation using Software-Defined Networks

Dissertation

Arne Schwabe

UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Submitted to the
Faculty of Electrical Engineering,
Computer Science, and Mathematics

In partial fulfillment of the requirements for the degree of
Doctor rerum naturalium (Dr. rer. nat.)

Submission: 14.12.2017

Referees:

Prof. Dr. Holger Karl,	University of Paderborn, Germany
Prof. Dr.-Ing. habil. Falko Dressler	University of Paderborn, Germany

Additional committee members:

Dr. Simon Oberthür,	University of Paderborn, Germany
Prof. Dr. Christian Scheideler,	University of Paderborn, Germany
Prof. Dr. Christian Plessl,	University of Paderborn, Germany

Abstract

Traffic demands in data centres are steadily increasing. Simple, yet commonly used round-robin algorithms are not able to meet these demands since they distribute the demands unevenly. Manually planning and redistribution, also called traffic engineering (TE), alleviates this problem. Traffic engineering is common in wide area networks (WANs). The high amount of manual work and associated cost in special hardware makes traffic engineering unpractical in data centre networks.

This thesis looks into the question whether software-defined networking (SDN) can be used to bring traffic engineering to the data centre. To reach this goal, the amount of manual labour and requirements for hardware need to be reduced. I first show that label switching, which enables better traffic distribution, is possible with SDN without special hardware. I present SynRace that infers network status by combining probes with SDN. The GlobalFIB framework allows applications to report directly their demands to a central instance. Both allow traffic redistribution to be automated.

With running multiple SDN application on the network and trying to fulfil all their demands and rules, conflicts are inevitable. In this thesis I analyse the nature of these conflicts and possibilities to handle them. The NetIDE core implements this conflict resolution and dynamic reconfiguration of the network.

Zusammenfassung

Das Datenverkehrsaufkommen in Rechenzentren wächst stetig an. Die weit verbreiteten Round-Robin Verfahren können dieses Aufkommen nicht mehr bewältigen, da diese die Last ungleich verteilen. Manuelle Planung und Umverteilung, auch Traffic Engineering genannt, lindert dieses Problem und wird vielfach in Weitverkehrsnetzwerken (WANs) eingesetzt. Der hohe manuelle Aufwand und die damit verbundenen Kosten, als auch die Kosten für spezielle Hardware, machen Traffic Engineering in Rechenzentren jedoch unpraktikabel.
Diese Dissertation beschäftigt sich mit der Frage, ob Software-Defined Networking (SDN) genutzt werden kann, um Traffic Engineering im Rechenzentrumsumfeld praktikabel zu gestalten. Um dieses Ziel zu erreichen, muss der manuelle Aufwand minimiert werden und die Anforderungen an die Hardware reduziert werden. Dafür zeige ich zunächst in dieser Arbeit, dass SDN es ermöglicht Label Switching, das bessere Verteilung der Flüsse ermöglicht, ohne spezielle Hardware zu realisieren. Ich stelle ich SynRace vor, welches SDN mit Testpaketen kombiniert, um den Status des Netzwerkes besser beobachten zu können. Darüber hinaus, wird as GlobalFib Framework eingeführt, welches SDN Anwendungen erlaubt, direkt ihre Bedarfsanforderungen an eine zentrale Stelle zu melden. Diese Ansätze erlauben die Verteilung der Flüsse zu automatisieren.
Wenn mehrere SDN Anwendungen im gleichen Netzwerk ausgeführt werden und versuchen, gleichzeitig ihre Anforderungen und Regeln durchzusetzen, sind Konflikte unausweichbar. In dieser Arbeit analysiere ich die Art dieser Konflikte und Möglichkeiten, mit diesen umzugehen. Der NetIDE core implementiert diese Konfliktbehandlung und die dynamische Konfiguration des Netzwerkes.

Acknowledgements

I thank my supervisor Prof. Holger Karl for the opportunity to start my own research, for continuous and constant feedback and support during all research years. He patiently taught me scientific thinking and writing.

I thank my parents Iris and Erwin for their continuous support during my study and graduation helping me to concentrate on my education.

The work presented here has been partially sponsored by the European Union through the FP7 project NetIDE, grant agreement 619543.
This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center “On-The-Fly Computing” (SFB 901).

Contents

1 Introduction

Technology is rapidly advancing and data usage is steadily growing. The often cited “The Zettabyte Era” analysis from Cisco [50] predicts a threefold increase of all traffic from 2016 to 2020 and for the busiest hour an increase by 4.6 times. This Internet-wide trend is also occurring in data centres [113]. The introduction of more distributed server architectures and of big data is increasing the communication demand between servers. To handle these increased demands, additional capacity is added to data-centre topologies by adding redundant links and paths. A single 10-GBit/s link between two switches is replaced by four 10-Gbit/s links. Naively, one would assume that the network is now able to handle four times the traffic between these two switches. Unfortunately, this is often not the case. The reason that the additional capacity is underutilised, lies in the behaviour of TCP/IP. TCP/IP has replaced nearly all other protocols in Ethernet networks; the performance of TCP/IP suffers when packet reordering happens. To avoid this performance problem on unsynchronised parallel links, a single flow is only forwarded over a single path. To use the capacity of more than one path, multiple TCP/IP flows are distributed over all available paths. A typical approach is that each switch locally distributes flows over all paths with the lowest cost metric to the destination in a simple round-robin fashion. While easy to implement and reliable, the resulting capacity under this approach is less than the raw capacity of the network, as the distribution of the flows over those available paths is often far from perfect. Imagine small and large flows that arrive in a (strictly) alternating pattern, a small query for a data location, directly followed by the transfer of the data. The round-robin distribution will assign the big flows to two of the four links and the small flows to the other two links. This results in a poor use of the available capacity. The discrepancy between raw and resulting capacity highly depends on the network topology and the workload but can be as high as 50% [5] (also shown in Chapter 4).

The simple example in the previous paragraph showed that the remaining capacity is very workload sensitive. As the capacity, which the applications will be able to use, also depends on the concrete random distribution of the flows, the remaining capacity is a random variable rather than a fixed number. For simplicity, I will use the term “remaining capacity” in lieu of expected value of the remaining capacity for a workload. In some scenarios, the resulting capacity might still be sufficient to fulfil the requirements, even though it is smaller than the raw capacity the network provides. Otherwise the resulting capacity needs to be increased. Two options can be used to alleviate the problem. The first is to add even more raw capacity to the network to increase the usable capacity. The second is to use the existing capacity more efficiently.

In data centres, adding more capacity can be as cheap as adding an extra link between two switches that both have unused ports if an unused fibre for that link is readily available. Adding raw capacity when no ports are free might also involve adding additional switches to the network or even adding extra fibres to the data centre if all existing fibres are already in use. In wide-area networks (WANs), adding raw capacity involves laying or renting fibre that might span hundreds to thousands of kilometres and has a significantly higher cost associated with it than adding fibres in a data centre [37]. When cost prohibits adding more raw capacity or adding more raw capacity is not feasible, for example when adding more fibres would require construction work, more likely in WANs, or takes too long, the remaining option is to use the available capacity more efficiently. This involves using the underutilised paths in the network more and employing more sophisticated methods for choosing paths than the simple round-robin approach. To implement this, traffic is classified, analysed, rules for better resulting capacity are developed and finally implemented. This process is typically summarised as traffic engineering (TE). Figure 1.1 shows a summary of the process that is often iterative. At the current state of the art, all these steps are done manually or only partially automated, which contributes to a high cost. For this reason, TE is predominately present in WANs since the costs are favourable compared to the cost of adding raw capacity. A complementary technique used to partly solve the problem is quality of service (QoS) and prioritisation of traffic. Instead of trying to provide the needed capacity to all flows, the flows are prioritised and only the higher prioritised flows will receive the full data rate. Often networks, especially data centre networks, have services that can be reduced in data rate without having a negative effect on other services or on the quality of this service. A typical example are backups, which transfer large amount of data but a longer backup time is unproblematic.

While it is certainly possible to implement the traffic engineering of WAN networks in data centres, doing so has a few shortcomings. Many WAN TE techniques are coupled with virtual circuit switching based on Multiprotocol Label Switching (MPLS) [4]. Introducing MPLS into a network adds a fair share of complexity, especially for smaller setups. MPLS is often needed in WAN networks to separate different customers and create network-wide overlays [25]. In data centres, this task is typically implemented with the much simpler VLAN protocol [12]. And most data centre servers, software and middle boxes cannot handle MPLS as it is uncommon in data centres. A translation from/to MPLS is required for these components. Additionally, data centres typically already have QoS rolled out based on VLAN/IP based QoS and would also need to reimplement it with MPLS based QoS. Most other data centre-specific technologies are not based on MPLS but directly on VLAN and IP and therefore are not compatible with MPLS at all. For example, advanced QoS and lossless Ethernet/Fibre Channel over Ethernet (FCoE) [94] is tightly coupled to the VLAN features [42]. This inflexibility mainly exists because switch vendors implement protocols for a certain use case (WAN or data centre) and hardware and software products are also targeted to a specific market.

Software-defined networking (SDN) allows to deviate from the protocols and implementations provided from the switch manufactures [4]. For example, advanced per flow decision and QoS features are often tied to the MPLS implementation. In our case, using SDN allows traffic engineering tailored to data centres, keeping the VLAN protocol and not introducing MPLS and still provide the ability to decide the path per flow. An additional advantage of SDN is that it allows greater control of the network. With the finer control of flow statistics

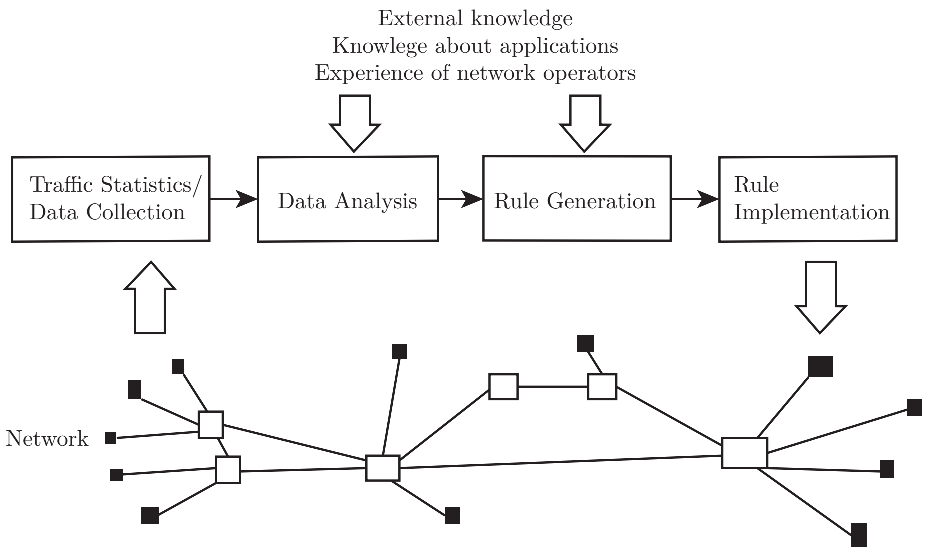

External knowledge
Knowlege about applications
Experience of network operators
Traffic Statistics/ Data Collection
Data Analysis
Rule Generation
Rule Implementation
Network

Figure 1.1: Steps of traffic engineering

in switches and more selective mirroring and duplicating of flows to a monitoring instance, it directly allows to implement new (and better) methods of data collection. Using the greater flexibility of SDN in two steps of the TE process that directly interacts with the network, allows a greater flexibility in the other two steps as their input, and output needs to match the capability of the network.

1.1 Resulting challenges

The main challenge with bringing traffic engineering to the data centre is to reduce the associated costs. Since the costs derive primarily from today's manual nature of TE, the natural consequence is to automate these steps as much as possible.

A major step in TE is the analysis of the traffic present in the network and gaining knowledge about the traffic patterns. Instead of doing manual analysis and classification, the knowledge about traffic should be obtained from network applications directly or by observing phenomena that can be automatically analysed. The next step is the automatic generation of the resulting rules to also eliminate that step.

Most data centres are not built for a single purpose and are running multiple different applications. These applications all have different characteristics and different interfaces. For that reason, to collect all needed available knowledge, the information gathering and analysis need to run in parallel for multiple different network applications. For example, one method will gather generic information of network behaviour while another is specialised for a specific application such as Hadoop [35, 112, 124]. Each of the methods can produce its own set of network rules.

Additionally, separately from traffic engineering, a data centre can have other services running that depend on SDN, e.g., monitoring, firewalls or intrusion detection system (IDS) [36]. These services also create network rules to fulfil their function. They are often implemented as SDN applications tightly integrated in an SDN controller. The network rules from all these different sources can contradict each other. Installing all these rules to the network switches can have bizarre and hard-to-debug problems and can cause part or the whole functionality of these applications to cease working. As an example, assume the IP rewriting rules of a load balancer that directs incoming servers to real IP addresses of the backend servers are not executed because a switch installs its own rules with higher priority. As consequence, some way of detecting and resolving these conflicts is needed.

1.2 Contributions

In this thesis I present possible solutions to the challenges outlined in the previous section.

The implementation of custom network rules to steer traffic is one of the enabling features that makes traffic engineering useful. As mentioned in the previous sections, a technology typically used for this, especially in WANs, is MPLS. MPLS is typically not supported by data centre switches or only at a premium price since it is considered a WAN/advanced feature (e.g., Cisco requires extra licenses [118, 119]). I show in Chapter 3 that it is possible to use SDN and MAC address rewriting to achieve label-based routing with an SDN network without specialised switches/switch software. Additionally, by exploiting Ethernet Address Resolution Protocol (ARP) semantics, ingress label tagging can be offloaded to the hosts.

SynRace, which I present in Chapter 4, uses SDN to extract information for TE that would not be available without SDN. SDN is used to turn the first packet of each connection (the SYN packet) into multiple probe packets. The observable timing difference and order in which these probe packets arrive, allows to infer the best path for a flow without requiring special hardware features from the switches. Using this novel way of acquiring information (the first TE step) and using label routing (e.g., MPLS or the MAC label routing from Chapter 3) increases the utilisation of the network.

I discuss the detection and resolution of conflicts needed for heterogeneous networks in Chapter 5. I describe the problem and possible solution of detecting and resolving these conflicts of rules of multiple sources in an SDN environment. I also show the limits and requirements in an SDN environment for composition of rules.

As a proof of concept, Chapter 6 presents a working implementation of these concepts that has been developed as part of my work and the NetIDE EU project and was the core component of that project.

Data that can be collected directly from the networks only provides a certain amount of information: data about the past and current state; for future flows only educated guesses and heuristics can be used. In order to have a better basis for handling upcoming flows the cooperation from the applications running in the data centre is needed. To do that in a structured way, I came up with the GlobalFIB framework idea that defines a framework to collect this information in a structured way and is described in detail in Chapter 7.

Finally, to further increase the usability of this system and concepts in demanding environments, In Chapter 8 I explore the options of dynamically detecting problems and reacting to these problem by reconfiguring the system. I conclude the thesis with a summary in the last chapter (Chapter 9).

In the following chapter I will often use "we" instead of I to emphasise that the work was done in collaboration with colleagues, as can be seen by the publications the chapters are based on.

Chapter 3 is based on

- Arne Schwabe and Holger Karl. "Using MAC Addresses As Efficient Routing Labels in Data Centers". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking.* HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 115–120. ISBN: 978-1-4503-2989-7. DOI: `10.1145/2620728.2620730`. URL: `http://doi.acm.org/10.1145/2620728.2620730`

Chapter 4 is based on

- Arne Schwabe and Holger Karl. "SynRace: Decentralized Load-Adaptive Multi-path Routing Without Collecting Statistics". In: *Proceedings of the 2015 Fourth European Workshop on Software Defined Networks.* EWSDN '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 37–42. ISBN: 978-1-5090-0180-4. DOI: `10.1109/EWSDN.2015.58`. URL: `http://dx.doi.org/10.1109/EWSDN.2015.58`

Chapter 5 is based on

- Arne Schwabe, Pedro A. Aranda Gutiérrez and Holger Karl. "Composition of SDN applications: Options/challenges for real implementations". In: *Proceedings of the 2016 Applied Networking Research Workshop.* ACM. 2016, pp. 26–31

- The chapter "Composition of network applications" in the journal paper: Elisa Rojes, Sergio Tamurejo Roberto Doriguzzi-Corin, Andres Beato, Arne Schwabe, Kevin Phemius and Carmen Guerrero. "Are we ready to tackle Software Defined Networks? A Comprehensive Survey on Management Tools and Techniques". In: *ACM Computing Surveys* (to be published)

  I was the lead author of that chapter.

Chapter 6 is based on

- The sections addressing the NetIDE core in NetIDE deliverable D2.7 NetIDE FP7 Project. “D2.7 NetIDE Manual”. In: *NetIDE reports* (2016)
- The NetIDE core implementation, which can be found at `https://github.com/fp7-netide/Engine/`.
- PA Aranda Gutiérrez, E Rojas, A Schwabe, C Stritzke, R Doriguzzi-Corin, A Leckey, G Petralia, A Marsico, K Phemius and S Tamurejo. “NetIDE: All-in-one framework for next generation, composed SDN applications”. In: *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE. 2016, pp. 355–356
- The implementation of the core has been based on Tim Niklas Vinkemeier’s master thesis:

  Tim Niklas Vinkemeier. “Composition and Orchestration of Network Control Applications”. MA thesis. University of Paderborn, 2015.

Chapter 8 is based on

- Arne Schwabe, Elisa Rojas and Holger Karl. “Minimizing Downtimes: Using Dynamic Reconfiguration and State Management in SDN Networks”. In: *3rd IEEE Conference on Network Softwarization (NetSoft 2017)*. Bologna, Italy, July 2017

2 Background

Before presenting the details of my contributions, I will briefly describe in this chapter the background knowledge that the remainder of thesis assumes the reader to be familiar with.

2.1 Data centres: Typical structures

Data centres are found in a variety of organisations and fulfil a variety of tasks. The building blocks used in these data centres are very similar and I will describe the typical building blocks here.

The most essential building block that constitutes a data centre is a server. Even the most basic server room, which can be an unused broom closet, has at least one server in it.

Storing servers on shelves without proper ventilation does not scale very well [47] and IT infrastructure in general is very power-intensive [73]. Also, accessibility and other practical concerns make this server housing unpractical for larger installations. Instead, servers are bought as rack-mounted servers which are installed in standardised racks [29, 70]. Server and rack space are measured in rack units (RUs). Standard-sized racks can accommodate 42 RUs with smaller racks also available for smaller installations, e.g. half racks with 21 RU and 4 RU racks for small office installations. With servers typically measuring 1-4 RUs, a full rack contains 13 to 42 servers.

In order to operate as expected, servers have to be provided with electric power, cooling and connectivity to other systems. In this thesis, I only look at the network connectivity of servers. In a data centre, connectivity is provided to the servers for communicating with the other servers and to communicate to the systems outside the data centre, usually the Internet. Ethernet is the de facto standard and has largely replaced any other network interconnect [115]. Additional specialised interconnects might be present, e.g., a high-speed interconnect for compute clusters like InfiniBand or a specialised storage interconnect like Fibre Channel but these are also slowly replaced by Ethernet [95], the latter by iSCSI [61] and Fibre Channel over Ethernet (FCoE) [94].

Each rack typically has its own Ethernet switch that is connected to each server inside the rack, often a 1 RU switch located at the top of the rack, which coined the term “top-of-rack switch” or simply “ToR switch”. When servers with 2 RUs or more are used, fewer but larger servers are in a rack. Since they require fewer Ethernet connections, one ToR switch is sometimes shared between two neighbouring racks.

All Ethernet switches together with the links between these switches are called the *backbone* of the data centre. The conventional design, which is also found in network design guides and textbooks [10], is a tree in which the ToR switches are the leafs and the switch at the root of the tree is called the “core” switch. The number of layers mainly depends on the size of the data centre and the number of ports available on the switches used in each layer. For a small data

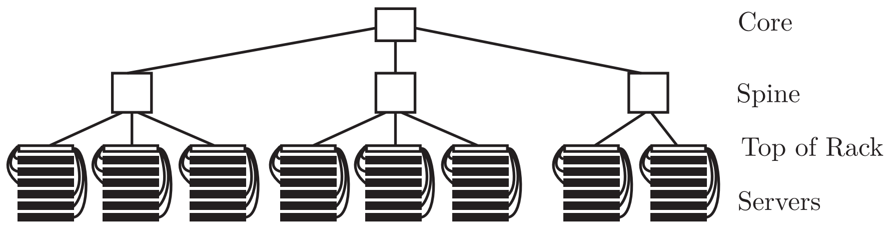


Figure 2.1: Ethernet data centre backbone with core, spine and ToR layer

centre, a two-layer tree with only a single core switch and corresponding ToR switches is a reasonable solution. A larger data centre might introduce a layer consisting of "end-of-rack" switches that bundle multiple ToR switches and even further introduce a layer called "distribution layer" or "spine switches" (coming from the backbone analogy) of switches below the core switch to multiplex the core switch's ports. Figure 2.1 gives an example with core, spine and ToR layer.

This simple structure has two main weaknesses: redundancy and scaling. Should any of the switches or links fail, the network is partitioned and a server cannot reach servers outside its own partition anymore. The other weakness is that the size of the network only scales with the number of layers. If the core or any other switch is not able to accommodate more links, another layer needs to be added. The achievable bisection bandwidth of this topology is limited by the maximum data rate between two switches. This data rate can be increased by bundling multiple links (usually up to eight [49]). Bundling multiple links between switches can further make the limited number of ports on the switches an issue.

Modern architectures avoid these weaknesses by not limiting themselves to a strict tree topology and use more flexible approaches to connect the switches in the topology [123]. Especially the notion of a single root node is abandoned and, as consequence, these topologies are not loop-free anymore and have multiple paths between leafs. This allows alternative paths to be used if a link should fail. Since the topology is not loop-free anymore, the network has to ensure that packets are not forwarded in a loop. The traditional Ethernet spanning tree protocol [41] eliminates alternative paths and reduces the active network to a tree subset, thereby also removing any loop. This still allows higher redundancy but eliminates performance benefits of topologies with alternative paths. To use these alternative paths not only as standby paths, either IP routing with a routing protocol using multiple paths like Open Shortest Path First (OSPF) [75] or an advanced Ethernet forwarding protocol is required. Examples for Ethernet forwarding protocols using multiple paths are Transparent Interconnection of Lots of Links (TRILL) [2, 87], RBridges [87], vendor-specific protocols [19] or the use of software-defined networking (SDN), which will be introduced in Section 2.3.

Many new topologies have been proposed to replace the traditional tree topology, including but not limited to HyperX [3], Dcell [44], BCube [43] and

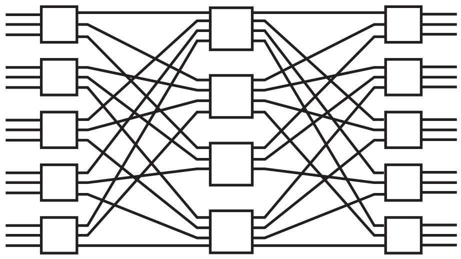


Figure 2.2: 15 port Clos switch built from 3 and 4 port switches

Jellyfish [114]. While these topologies promise good performance, their wiring requirements make them difficult to implement, e.g., the Jellyfish topology requires random connections between the ToR switches. This creates an unstructured wiring with few cables that can be bundled and few short distance links that are generally preferred for their lower cost.

A typical compromise between practicality and performance are the FatTree topologies [64], and networks topologies based on Clos switch [24, 113], which I will call just Clos topologies in the following. Since the basic ideas of these topologies are very similar and the names are often used interchangeably, I will describe them together.

The Clos network comes from the telecommunication world; its idea is to build a large non-blocking (telephone) switch from smaller non-blocking switches [24] by arranging the switches in three stages, the ingress, middle and egress stage. The special way of interconnecting these stages allows Clos networks to be non-blocking. In a circuit-switched system with inputs and outputs, non-blocking in a strict sense means that from a currently unused input to an unused output a free path always exists. The rearrangeable non-blocking property is weaker and only requires that for any combination of inputs and outputs a mapping without blocking exists. If a new connection arrives, the existing connections might need to be rearranged to provide a free path from input to output for the new connection [54]. Figure 2.2 shows an example for a 15-port switch built from smaller 4-port and 3-port switches. For Ethernet topologies, which are packet-switched, I cannot directly use the circuit switching-based definition, so instead I use the rearrangeably non-blocking property.

The concept of a Clos-like switch network to replace the big monolithic switch as core switch with multiple small switches is common to the FatTree and Clos topologies. In computer networks, Clos networks are often not designed to be fully non-blocking to trade off a lower bisection bandwidth against a smaller number of switches and links. This is done by using a smaller number of switches in the middle stage but keeping the main structure of a Clos network. Figure 2.3 shows a realisation of a data centre network using a Clos topology. Note that the spine switches are both the ingress and the egress stage and the core switches are the middle stage.

Closely related to Clos topologies is the FatTree topology, which also uses

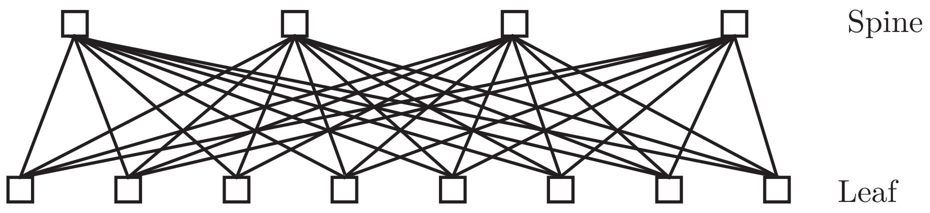


Figure 2.3: Data centre Clos-like topology

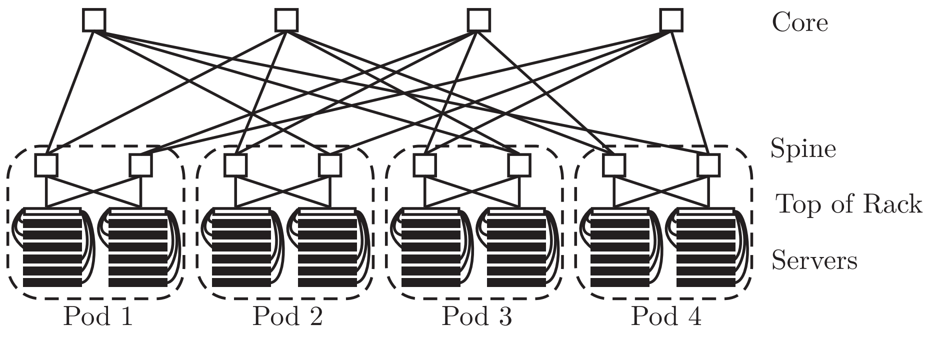


Figure 2.4: FatTree topology with four pods and four core switches.

a Clos network for its core but uses a different approach to bundle racks. In a FatTree, a number of racks with their spine switches form a so called pod. Figure 2.4 shows an example of a FatTree topology. In the example, each pod has two racks but pods are not limited to two racks. The two spine and edge switches in each pod form a small 4-port non-blocking switch. The connections of these 4-port switches to the core switches form a Clos switch.

Implementations of these topologies may opt to use fewer core switches or fewer links between spine and core switches to save costs. This does not guarantee the non-blocking property anymore but it allows to later add an additional core switch, once the capacity is needed. In the same way a pod will typically not have only two edge switches but a larger number, like four or eight. With more than two ToR switches in each pod, switching inside each pod is still non-blocking, but the available number of links to the core cannot provide a non-blocking network anymore. Figure 2.5 shows such a cost-conscious FatTree network, which uses two instead of four core switches and five instead of two racks per pod.

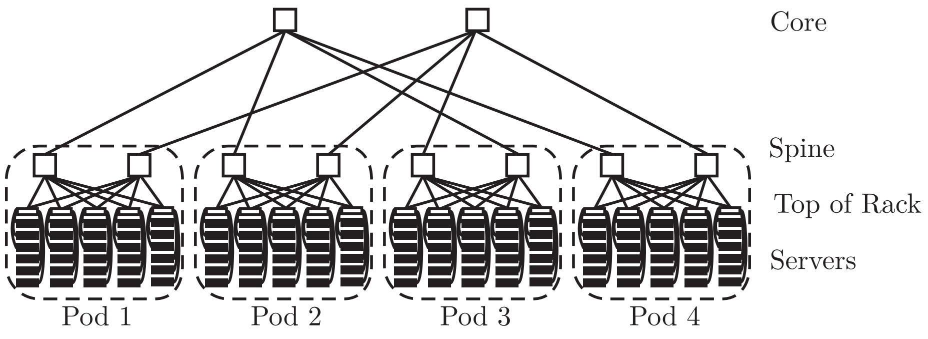


Figure 2.5: Cost-conscious FatTree topology

2.2 From conventional networking to SDN

Software-defined networking (SDN) is a new technology that promises to provide access to network control and replace closed and/or proprietary protocols with open protocols, allowing a network administrators and network software developers to define network switch behaviour themselves. Unfortunately, SDN is not a technically precise definition; it often describes the collection of various technologies and is being used more and more as a marketing term. As a result, everyone has a different interpretation of what SDN actually stands for. This section will detail what I understand of the term SDN.

Having software-defined systems is not a completely new idea. The main idea behind all software-defined technologies is that as little as possible is predefined by the hardware but instead can be implemented in software. For example, software-defined radios (SDRs), another technology that has "software-defined" in its name, is a much older technology [122]. Unlike SDN, the distinction between a software-defined radio and a conventional radio is much easier to make. A conventional radio is designed and implemented for one or more specific radio protocols, e.g. Bluetooth, and the majority of signal processing is done in hardware. Using a new or another protocol means replacing the hardware. On an SDR, the actual hardware does as little signal processing as possible but instead has an interface to send and receive raw samples of modulated radio signals. Behaviour of the radio module, especially the physical layer and modulation being used, can then be defined solely by software. This allows users to design and use their own radio protocol implementations instead of choosing only between the protocol implementations that the hardware chip manufactures offer.

SDN brings this idea to computer networking. Traditionally, switches and routers only support the protocols and features their manufacturers have implemented. These devices have no possibility to replace the software that implements these protocols. SDN adds interfaces to the switch that make the switches' behaviour programmable by software. Software in this context is the software the administrator chooses. This software can be written by the administrator him/herself, by the manufacturer of the switch, or by a third party. SDN switches provide an application programming interface (API) to allow this programmability. Different APIs exist for SDN; in this thesis, I will focus on OpenFlow, the SDN protocol that is typically used in academic research and is in widespread use in production SDN networks. Its interface focuses on having very little logic in the switch itself (just as with an SDR).

As a side note, implementing switches/routers in software was always possible (unlike to radio protocol design); a normal Linux system with multiple network cards can implement a switch. At first glance, this also allows the same features that SDN promises. The question arises whether SDN and special protocols like OpenFlow to interface switch hardware are really needed or whether a system built from commodity computer hardware could also be used. OpenFlow has two advantages here: the first is to provide a clear API that also serves as common protocol to interface different switch implementations. Projects like Open vSwitch [88] implement an SDN switch on a normal Linux PC, providing the same interface as a dedicated SDN switch. The other one is that packet forwarding at a high rate is very difficult to achieve without specialised hardware. For forwarding a packet, a switch needs to extract the header fields from a packet, look up these header fields in a table, apply the result of the lookup to the packet and output the packet to destination port. With modern network speeds a switch has to execute this task million times per second; a fully loaded 10-Gigabit Ethernet port equals to 833 333 to 19 531 250 packets per second depending on the packet size (64-1500 byte). Even high-end hardware with specialised software struggles to achieve these rates, especially if the forwarding tables are not unrealistically small [90]. To be able to achieve these high forward rates, switches use specialised and custom-built hardware. SDN also allows to use software switches (e.g. Open vSwitch) and hardware switches with the same protocol.

2.3 OpenFlow: Access to the switch data plane

OpenFlow was the first published SDN control protocol [69] and still is one of the most used SDN protocols [60]. OpenFlow's main idea is to allow external entities to access and modify the switch's Forwarding Information Base (FIB). It allows to install and remove and modify flow rules contained in this table. OpenFlow abstracts the entries of the forwarding table into rules with three parts: match, action list and counters. Every incoming packet is first matched by the rules. OpenFlow defines a number of header fields that can be used in a match; common options for matching include Ethernet MAC addresses, TCP/IP addresses and UDP/TCP ports. For the rule that matches the packet and has the highest priority, the counters are increased and the action list is evaluated.

In the action list, modifications of the packet can be specified, overwriting header fields, e.g. replacing MAC and IP addresses or adding/removing header fields like MPLS and VLAN tags. The action list also has an action that specifies what should be ultimately done with the packet. This can be to drop the packet or to output it to one or more ports. The port can also be a virtual port that forwards the packet over the control connection. This generates a “packet in” message. The controlling software can then react probably and for example install a new rule if the packet belongs to a new flow.

OpenFlow uses a standard TCP connection between the switch and the application controlling the switch, called the “SDN controller”. As OpenFlow itself is a network protocol that works over TCP, a single SDN controller can control multiple switches at the time, allowing SDN with OpenFlow to implement a central control instance, in stark contrast to traditional networks. These distribute the control over all switches and the traditional protocols, like(Border Gateway Protocol (BGP), OSPF, spanning tree,... are also designed as distributed algorithms. Often used with OpenFlow controllers is that an SDN controller implements the *reactive* installation of flow rules: Instead of installing all forwarding information before a packet arrives, the controller will purposely not populate the flow rules for the majority of flows and will only install the required flow rules when a packet triggers a miss in the flow table. This allows more fine-grained control of flows but makes the communication between a switch and the SDN controller a potential bottle neck. The alternative is to install flows before flows arrive which avoids the additional round-trip and performance hit but limits control of individual flows.

3 Efficient SDN Routing Labels

Traffic engineering (TE) depends on routing traffic via specific paths to distribute load over the available paths. Conventional traffic engineering (TE) often depends on Multiprotocol Label Switching (MPLS) [99] to implement forwarding over specific paths.

MPLS allows to assign labels to packets and to base forward decisions on the labels independent of the packet's contents. MPLS labels can also be stacked, which allows to bundle certain labels for parts of their paths. For a WAN network, a label stack consisting of three labels could express the fine-grained ingress forwarding, then the forwarding in the backbone and the last label could indicate routing in the egress network. Optionally, flows can be grouped by labels and stacking labels allows to keep the forwarding tables in the switches small. It also allows to shift classifying the packets of a flow to the edge of the network, typically called forwarding equivalency classes (FEC) in MPLS.

As indicated in the introduction (Section 1.1) MPLS is not available or sometimes conflicting with other technologies in a data centre. In this chapter, I will investigate two things: The first one is how well label forwarding can be emulated using SDN and switches that lack MPLS features. The second is to determine if structured MAC addresses can help to reduce the number of forwarding entries. For that I will formalise the problem, prove its $\mathcal{NP}$-completeness and present an approximation algorithm.

This chapter is based on

- Arne Schwabe and Holger Karl. "Using MAC Addresses As Efficient Routing Labels in Data Centers". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 115–120. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620730. URL: http://doi.acm.org/10.1145/2620728.2620730

3.1 Introduction

In a modern switch, forwarding is implemented in hardware to forward at line rate. A key component of forwarding is the lookup table which is implemented using Ternary Content-Addressable Memorys (TCAMs). TCAMs are expensive in energy and hardware. This limits the number of forwarding entries a switch can store in hardware. Forwarding in a data centre is based on the lookup of the destination MAC address in contrast to WAN networks that usually base their forwarding decisions on IP addresses or (MPLS) labels. MAC addresses of hosts are usually unstructured and each host needs an entry in the forwarding table. Reducing the number of entries in forwarding tables will save hardware costs and energy.

Usually, the number of required entries is reduced either by using compression/combining multiple entries or by storing only a subset of the entries in hardware tables. Combining multiple entries is difficult and even good approaches only achieve compression rates of 50% [101]. The other approach, to store only a subset of the entries in the hardware tables, introduces an additional delay whenever a packet arrives that has no corresponding entry in the hardware table and an entry has to be fetched, e.g., from a central controller (as often in SDN) or locally computed. Both solutions only alleviate the problem and provide no real solution. These reasons contribute to the need that commercial switches currently have to provide a very large number of possible entries.

The root of the problem is the lookup of unstructured addresses. For unstructured addresses in most cases one forwarding information base (FIB) rule per address is needed. One solution is to avoid the lookup of MAC addresses and instead perform the lookup on a label that can be controlled and given a structure. The idea of using labels to improve or enable forwarding hardware is quite old, dating back at least to ATM and MPLS. To benefit the most from using labels for forwarding, labels should be added to the packets as early as possible and removed as late as possible along the packet's path. Typically, the ingress switch adds the label to a packet and the egress switch strips the labels from the packets. This allows all switches between the ingress and egress switch to benefit from doing label lookups instead of MAC address lookups (or other unstructured/expensive header field lookups).

The label can be added in two ways. The first way is to encapsulate the packet, e.g., adding an MPLS header on the ingress switch and decapsulate the packet on the egress switch, removing the label again. The second way is to replace the content of a header field with the label at the ingress switch and, if necessary, reverse the replacement at the egress switch. Approaches specially designed for the problem in a data centre like Portland [81] use the second approach and replace the source and destination MAC addresses of the packets.

Using labels for forwarding shrinks the forwarding table when structured labels are used. Structured labels enable assigning similar labels to packets forwarded in the same way and thus one forwarding entry can be used to match multiple labels. This saving is obviously only true if labels have been already applied. The ingress and egress switches need multiple flow table entries for removing and adding labels, which are about the same number of entries as with regular Mac-based forwarding tables as the number of unstructured addresses that need to matched is the same. The work of adding a destination-specific address to the packets is also duplicated at the hosts and the ingress/egress switches. The hosts add source and destination MAC addresses according to their own addresses and ARP tables; the switches will then replace the addresses according to their tables.

Modifying the hosts (respectively, their operating system) to directly write the right MAC addresses (containing the labels) is the apparent solution, but modifying hosts (and their operating system) is often unacceptable or impossible. Without modifying the hosts, the source MAC addresses are always the hosts' own addresses, which cannot be controlled. But the ARP table of the hosts is filled from ARP replies and this can indeed be controlled from the outside without modification to the hosts. The question arises how far this limited control over the hosts can be exploited to eliminate the duplicate work and further improve the gain from using labels for forwarding. We will show in this chapter that this is indeed feasible and advantageous.

3.2 Background

This background section recaps the important aspects of lookup tables in switch hardware.

3.2.1 Lookup hardware

In a switch, an incoming packet is matched against the FIB of the switch. This is done by looking up multiple header fields from the packet itself as well as using meta information including the ingress port of the packet. The lookup is either made as an exact match, e.g., an IP address equalling 1.2.3.4, or as a wildcard match such as an IP address matching 5.6.0.0/16, often combined with a priority to give longer prefix entries a higher priority.

This matching is implemented using TCAMs and allows fields to match against 0, 1 and “don’t care”, usually written as X. As an example, the wildcard 11100XXX XXXXX1100 matches all 16-bit words beginning with 11100 and ending with 110.

Wildcard matches are an a indicator function for bit strings of length n:

$$t : \{0,1\}^n \mapsto \{0,1\}, \quad t(x) = \begin{cases} 1 & \text{wildcard matches } x \\ 0 & \text{otherwise} \end{cases}$$

Since the input domain ($\{0,1\}^n$) and output domain ($\{0,1\}$) of the function t are finite, the number of possible functions is also finite with 2^{n+1} possibilities. We define T_n to be the set of all possible wildcard functions with n bits input size. T_n represents all configurations that a TCAM with n bits can have.

3.2.2 Forward Information Base

Regardless of the routing algorithm/forwarding strategy used in the network, a FIB entry consists of a matching and a forward action.

To replace two FIB entries with one entry, two conditions have to be met. First, the matching of the new rule must match everything the two old rules matched and not match anything the old rules did not match. Second, the forwarding actions of both entries have to be the same. If the output actions differ, combining two entries will change the semantics of the combined rule. Merging multiple FIB entries is often impossible since the inputs (like the MAC addresses of hosts) have no structure that allows grouping them together in a wildcard.

3.3 Related work

Reducing the size of the FIB has been studied for various different scenarios and requirements. A very general approach is to use compact routing algorithms designed for arbitrary networks. For example, the compact routing scheme of [121] has a table size of $\tilde{\mathcal{O}}(n^{1/2})$ with a (worst case) stretch factor of 3. Introducing a stretch factor of 3 into data-centre forwarding would have the consequence that also the available data rate needs to be increased by a factor of up to three.

One very general proposal aiming to reduce the FIB table size is Pathlet routing [40]. Pathlet routing sets out to reduce the FIB size/complexity of inter-domain routing while retaining the flexibility of policy routing possible when using BGP. Pathlet routing achieves this flexibility by adding labels to the packets that encode parts of the path and allows each AS to decide trade-offs between the number of FIB entries and the complexity of the implemented routing policy.

The question arises if FIB reduction techniques aimed at inter-AS communication [31, 128] can be applied to the data center FIB reduction discussed here. Common to both problems is that the identifier used for forwarding, MAC addresses in the data center and IP prefixes in the inter-AS communication, are unstructured. Directly translating these proposals does not work well because it turns switches into ASes and the directly connected MAC addresses to the networks of the AS. Carefully adapting the proposals to data centre networks is not straightforward and thus creates new challenges.

There are a few key differences between these approaches and our approach: these approaches focus on future routers and Internet protocols. Their labels are not designed to be implemented in hardware or require hardware capabilities not present in current data centre hardware. And these approaches also assume a network (the Internet) that is managed by multiple entities whereas we can assume for our data centre network that only one entity manages the network; this single entity design is also reflected in the protocol design.

Instead of matching the header directly with TCAMs, the approach in [100] uses Bloom filters to match addresses. Bloom filters can have false positives. The approach avoids false positives by requiring a large number of alternative paths between communication pairs and implementing a strategy that avoids forwarding over paths which trigger false positives. The requirement of a special type of network and the need to modify hardware to support Bloom filters makes this approach only viable in scenarios where the hardware can be modified and either enough paths are available to offset the need to exclude certain forwarding paths or additional forwarding table entries can be installed, which can defeat the purpose of saving entries. In contrast, our approach is designed not to make assumptions about the topology or to need additional hardware modifications.

PortLand [81] is designed for FatTree topologies (see also Section 2.1) and assigns to each switch a hierarchical level of either core, aggregation or edge. A part of the network consisting of aggregation switches and edge switches is called a “pod”. For every host, its pod and edge switch are encoded into a “pseudo mac” address. Portland uses this pseudo MAC address for forwarding packets on the aggregation and core switches. Ingress switches rewrite the source MAC addresses to the pseudo MAC address. The egress switches replace the pseudo MAC address with the real address of the host again. PortLand maintains a one-to-one mapping between pseudo MAC addresses and hosts. The strict encoding of host positions into the MAC address limits PortLand to topologies matching PortLand’s network model (FatTree). Overall, PortLand is the closest to our own approach. The key differences between the approaches is that our approach does not assume a fixed topology and that we offload part of rewriting of MAC addresses to the connected end host by using ARP.

While PortLand maps one IP address to exactly one MAC address, first-hop redundancy protocols (HSRP [67] or VRRP [77]) let multiple IP addresses share a single MAC address for receiving packets on the failover IP gateway address but use their own MAC addresses for forwarding packets to the hosts. Using this “virtual” MAC address has the advantage that packets with this MAC address will always be accepted by the routers and the ARP table of the client always has a valid entry as long as one of the routers is still working. When forwarding a packet to a client, a router will use its own MAC address. The idea of answering an ARP request with multiple MAC addresses for the same destination host is also taken up by our approach but applied to all hosts in the network instead of only for the router.

3.4 Model and problem definition

To analyse and solve the problem of minimising the forwarding table size under these assumptions, we need a formal definition of the problem that simplifies it without abstracting away important details of the problem.

For our model, when a packet is forwarded in a network, it is forwarded to an outgoing port on each switch on its path until the packet arrives at its destination. We use the paths that the packets should be forwarded over and the graph itself as the input for our problem.

For Ethernet link layer forwarding (basic Ethernet switching) the forwarding action is “output packet on port x” and fits perfectly into the model. A more complex routing algorithm will install different forwarding actions with the same egress port. This can be represented in our model as two different outgoing edges going to the same switch (using a multi-graph-based model).

To minimise the forwarding tables, we need to assign the labels and construct the forwarding rules in a way that minimises the number of forwarding rules.

When constructing the wildcards to match the labels, only packets passing a switch need to be considered. For example, in Figure 3.1, a wildcard for the edge (u, v) must match the label of p_1 and must not match the label of p_2, but it is irrelevant if the wildcard matches p_3.

With these definitions, we can formulate:

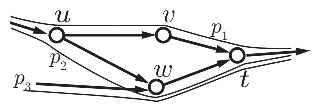


Figure 3.1: Small network with three paths (thin lines)

Path label assignment (PLA) Let n be the number of bits used for the label. Given a graph $G = (V, E)$ and a set of loop-free paths $P \subset \mathcal{P}(E)$. Does a mapping from the set of paths to the set of labels with n bits $m : P \mapsto \{0, 1\}^n$ exist so that for each edge $e = (u, v)$ a function $t_e \in T_n$ exists with $t_e(m(p)) = 1$ if $e \in p$ and $t_e(m(p)) = 0$ if $e \notin p$ for all paths $p \in P$ with $u \in p$ (all paths that have the same start vertex as e).

T_e is the set of all possible wildcard functions with n bits input size as defined in Section 3.2.1.

The definition of PLA asks for exactly one wildcard t_e per edge that should match all labels of paths $m(p)$ that traverse that link and should not match any other label $m(p)$ of a path that also crosses the same switch ($u \in p$ but not $e \in p$).

3.5 MAC addresses as labels

Using software-defined networking (SDN) gives a much greater control over the network. We use this greater freedom to repurpose the destination MAC address as a flexible forwarding label. This use of the MAC address has the big advantage that labelling packets can be offloaded to the host by using ARP rather than requiring a FIB entry to add the label on every ingress switch.

Hosts on the network rely on Ethernet Address Resolution Protocol (ARP) and Neighbor Discovery Protocol (NDP) [78, 89] to learn the MAC address of other devices. Instead of delivering the ARP query/neighbour discovery packets to the hosts, an SDN controller can intercept these packets. This allows us to respond with arbitrary MAC addresses instead of the MAC address of the host/network interface card (NIC) to which the IP address belongs. This arbitrary MAC address can then be used as a forwarding label. Intercepting and modifying the ARP request instead of attaching a separate label to the packets has an important advantage: the host will put the received MAC address into its own ARP cache and will put the MAC destination address into all outgoing packets to that particular IP address. This removes the need to label the packets on the ingress switch. Effectively we use the ARP table of the hosts to store the entries we otherwise need to store in the FIB tables of the ingress switches.

Answering the ARP queries allows us to answer with a label for the path to the destination IP address. If a different host uses ARP to query the same IP, we can respond with a different MAC address label. Effectively we have the possibility not only to use structured MAC addresses for destination hosts but even individually for each source and destination IP pair without requiring

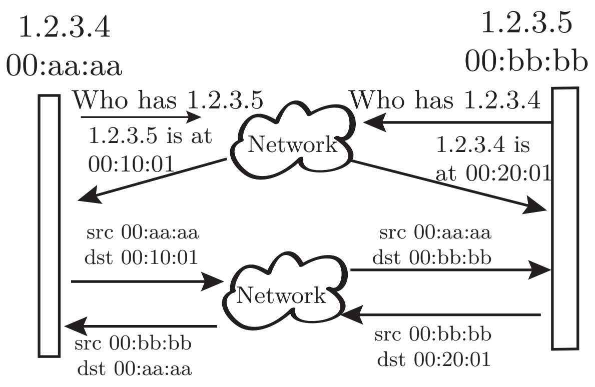


Figure 3.2: Intercepting and modifying ARP packets in an SDN network

additional FIB entries at the ingress and egress switches. Figure 3.2 shows an example of the resulting packet flows.

When a packet arrives at the egress, the packet still carries the label we applied earlier as destination MAC address. Without modifying the destination host operating system, the host will drop the packet since the destination MAC address is not matching its own. Hence, the egress switch needs to replace the destination MAC with the real MAC address. Having to do this extra step to undo the labelling seems to contradict the idea of using ARP to label packets. But the important difference is that labelling has to be done on every ingress switch, while rewriting the MAC address needs only to be done on the egress switch. Since the egress switch needs a FIB entry to forward the packets to the port in any case, this only adds an additional action to the already existing entry and does not consume an extra entry in the FIB.

Using the destination MAC address as a forwarding label breaks the assumption that the source and destination address of a host are always the same for the Ethernet layer. Our approach does not modify the source MAC address, which is the physical address of the sending host, to avoid adding FIB entries in the ingress switch. The ARP table of the receiving host contains a label MAC address for the source IP address instead of the real MAC address of the host. For a received packet the source MAC address will differ from the address stored in the ARP table. Nevertheless, the host will accept the packet; the first hop redundancy protocols work in a similar way, albeit in a much more limited scope, and we do not violate the standards for Ethernet end devices.

If we loosen our initial constraint of not allowing the modification of the host at all to allow the installation of an OpenFlow-capable software switch, like Open vSwitch, rewriting the MAC addresses can be done by the software switch, shifting the ingress and egress switch to the end hosts and removing these rules from the physical ingress and egress switches. Such software SDN switches are becoming more and more common, especially in virtualisation environments.

3.6 Solving PLA

In this section we present two methods for solving the PLA problem. We provide an exact solution with an integer linear program (ILP) and a greedy heuristic.

3.6.1 ILP solution

In this section we model the problem as an ILP. The variables of the ILP are defined as followed. All variables are binary (only values 0 and 1 are allowed).

t_{ej}	value of bit j of wildcard t_e
x_{ej}	bit j of wildcard t_e is a "don't care"
p_{ij}	value of jth bit of the label for path i
n_{ejk}	bit k of path j label is not matched by t_e
d_{ejk}	decision variable for n_{ejk}

To model the three-state nature of the wildcard bits we use two binary variables for each bit. The binary variable x_{ej} defines if the wildcard is in the "don't care" state and if it is 0, the variable t_{ej} specifies the value of the bit.

Path labels use normal bits and we model the path's n-bit label by binary p_{ij} variables.

To ease writing the following equations, we define the indicator function $s(p_j, e)$ with $e = (u, v)$ to be 1 if and only if p_j includes an edge (u, w) with $w \neq v$.

$$x_{ek} \geq p_{jk} - t_{ik} \tag{3.1}$$
$$x_{ek} \geq t_{ek} - p_{jk} \tag{3.2}$$

$k = 1 \ldots n,\ \forall e \forall p_j :\ e \in p_j$

$$\sum_{k=1}^{b} x_{ek} \leq n - 1 \qquad \forall e \tag{3.3}$$
$$n_{ejk} \leq t_{ek} - p_{jk} + (1 - d_{ejk}) \cdot M \tag{3.4}$$
$$n_{ejk} \leq p_{jk} - t_{ek} + d_{ejk} \cdot M \tag{3.5}$$
$$n_{ejk} \leq 1 - x_{ek} \tag{3.6}$$
$$\sum_{k=1}^{n} n_{ejk} \geq 1 \tag{3.7}$$

$k = 1 \ldots n,\ \forall e \forall p_j : s(p_j, e) = 1$

The first block of constraints (3.1–3.3) ensures that a wildcard for an edge matches all labels of paths containing that edge ($e \in p_j$). Constraints 3.1 and

3.2 ensure that t_{ek} and p_{jk} (bit k of wildcard and path label) are the same if x_{ik} is 0 (not a "don't care"). If t_{ek} and p_{jk} are different, the two equation force $x_{ek} = 1$, meaning that the k match bit of t_e is a "don't care" and thus the value of the variable t_{ek} is ignored.

Constraint 3.3 ensures that every wildcard has at least one bit that is not set to "don't care" by limiting the number of "don't care" to $n - 1$ bits.

With these constraints so far we know that every path that should be matched by a wildcard is actually matched. The rest of the constraints ensure that no other path is matched.

To ensure that the wildcard only matches labels it should match, i.e., not matching any other labels, Constraint 3.7 ensures that at least one bit of every other label has a value that is not matched by the wildcard. It uses the variable n_{ejk} for that; it is only 1 if a bit is not matched. The next constraints ensure the value of n_{ejk}. Constraints 3.4 and 3.5 ensure that n_{ijk} can be only 1 if p_{jk} and t_{ek} have different values (the constraints use a large value M construction with d_{ejk} as a helper variable to tie the constraints together). Constraint 3.6 furthermore ensures that n_{ijk} is 0 if the kth bit is a "don't care".

The ILP has no optimisation goal since the problem is either solvable or not.

As an optimisation to improve solving time, the n_{ijk} variables do not have to be binary but can be arbitrary float variables without changing the solution of the ILP since the constraints will force the variables to be either 0 or 1.

3.6.2 Greedy Heuristic

Unfortunately, calculating an optimal solution using the ILP does not yield a solution for any problem instances in a reasonable time, except for very small ones (less than 10 vertices). To find a solution for a larger (realistic) scenario, a faster algorithm is needed, trading off speed against a sub-optimal solution which may use more than one wildcard per edge.

The Ethernet MAC address has no variable length but a fixed number of 48 bits. Laying 16 bit aside to differentiate multiple (virtual) hosts behind a single switch port gives us a usable amount of 32 bit for the label. The split between the host and label part is somewhat arbitrary and is intended to be a good trade off but can be changed to accommodate other preferences.

Our approximation algorithm should gracefully adapt to a situation where a perfect solution requiring the one n-bit wildcard for every output port cannot be found and in this case use more than one wildcard rule per edge.

To achieve this goal we designed a greedy algorithm. The idea is to set one label bit after another for every edge in a way that brings the solution closer to requiring only one wildcard per link. For each bit, we will consider the switches in a random order and assign the bit values to the path that improves the situation at the most.

The greedy algorithm (Algorithm 1) gets as input the network graph and paths (of flows); its output labels for the paths. The algorithm works as follows: uniformly at random select an edge $e = (u, v)$. From all paths that include that edge e, determine from the so far assigned label bits the bits that all paths have in common. Apply these bits as a wildcard match on all paths that include an edge (u, v'), $v \neq v'$, i.e. all paths that traverse also the switch u. Put any path that matches the wildcard in the set U. This set U now contains the paths that cannot be distinguished from e at the switch u using the wildcard. If the set of U is empty, move to the next edge. Otherwise, set the unset bit of the paths in U so that the set U is minimised. Continue with the algorithm until either the set of indistinguishable edges is empty for every edge or when the number of bits is reached.

Algorithm 1 Greedy algorithm

```
    $P$ set of all paths, $E$ set of all edges, $n$ number of bits
 1: procedure assignLabels($P, E$)                    ▷ Initialise all label bits to unset
 2:     for all $p \in E$ do
 3:         for $i = 1 \dots n$ do
 4:             label[$p$][$i$]=unset

 5:     for  $i = 1 \dots n$ do
 6:         for $e = (u, v) \in$ shuffle($E$) do
 7:             $L = \{p \in P \mid e \in p\}$                    ▷ All paths that include $e$
 8:             $w$ = getMatch($L$)
 9:             $U = \{p \in P \mid \exists\ (u, v') \in p, u \neq$ and $w(\text{label}[p]) = 1\}$
                                ▷ All paths that have the switch $u$ in common with $e$
10:             if $U = \emptyset$ then
11:                 continue                                      ▷ Everything good
12:             for $k = 1, 2$ do
13:                 if $\forall\ p \in L :$ label[$p$][$i$] $= k \vee$ unset then
14:                     for all  $p \in L$ do
15:                         label[$p$][$i$] $= k$
16:                     for all  $\{p \in U \mid$ label[$p$][$i$] = unset$\}$ do
17:                         label[$p$][$i$] $= k$
18:     return labels

19: procedure getMatch($S$)
20:     $m \in T_n$                       ▷ indicator function (defined in Section 3.2.1)
21:     for $i = 1 \dots n$ do
22:         if $\forall\ p \in S,$ label[$p$][$i$] $= 0$ then
23:             $m[i] = 0$
24:         else if $\forall\ p \in S,$ label[$p$][$i$] $= 1$ then
25:             $m[i] = 0$
        return $m$
```

After the bits have been set for every path and U is empty for every edge, only one wildcard is needed per edge. For the infeasible case, we use a greedy second phase of the algorithm (Algorithm 2) to find a valid set of wildcards. Since using bits that are common to all paths does not create an empty set U,

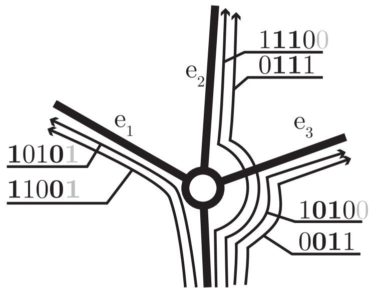


Figure 3.3: Node with six paths and four already assigned bits, common bits in bold, next bits in grey

we split the wildcard into two wildcards w_0 and w_1. We calculate the sets U_1 and U_0 for all bits that are not common between all paths. Then we choose the bit that minimises U_1 and U_0. We repeat this step until the sets U_i are empty for all wildcards w_i of the path.

As an example, consider Figure 3.3 where the first four bits are already set. A wildcard using the common bits of all paths including edge e_1 is 1XX0, which matches one path of e_2 and one of e_3. Adding 0 to the paths of e_1 and 1 to the paths of e_2 and e_3 makes the wildcard 1XX01 match only paths of e_1.

Algorithm 2 Second Phase of the greedy algorithm

```
procedure SECONDPHASE($P, E$, labels)
    for $e = (u, v) \in$ shuffle($E$) do
        minsize = $\infty$
        $\min_i = 0$
        $L = \{p \in P \mid e \in p\}$                ▷ All paths that include $e$
        for $i$ = shuffle($1 \ldots n$) do
            $w$ = getMatch($L$)
            $w_0 = t_L$, $w_1 = t_L$
            $w_0[i] = 0$, $w_1[i] = 1$
            $U_0 = \{p \in P \mid \exists\ (u, v') \in p, u \neq$ and $w_0(\text{label}[p]) = 1\}$
            $U_1 = \{p \in P \mid \exists\ (u, v') \in p, u \neq$ and $w_1(\text{label}[p]) = 1\}$
            if minsize $> |U_0| + |U_1|$ then
                $\min_i = i$, minsize $= |U_0| + |U_1|$
        for $p \in U_{0,\min_i}$ do
            label[$p$][$\min_i$] = 0
        for $p \in U_{1,\min_i}$ do
            label[$p$][$\min_i$] = 1
```

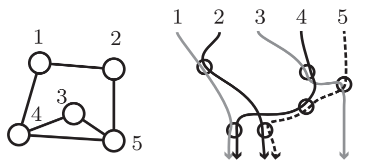


Figure 3.4: Example transformation of a graph for 4-colourability

3.7 Complexity of PLA

The path label assignment problem is $\mathcal{NP}$-complete for inputs of arbitrary networks. We present a proof in this section.

The existence of a polynomial-sized ILP (see Section 3.6.1) for the problem shows that the problem is inside $\mathcal{NP}$ since ILP problems are solvable by an $\mathcal{NP}$ algorithm. To establish $\mathcal{NP}$-completeness, we need to show that for the other direction a polynomial-time reduction exists as well. We will show that the 4-colourability problem [22, 23] (can four colours be assigned to vertices of a graph so that no edges connect vertices with the same colour) can be reduced to the path label assignment problem.

Let $G = (V, E)$ be the input for the 4-colourability problem. For each edge $e_i \in E$ we add a switch r_i in our model with two input and two output ports. Each vertex $v_j \in V$ is identified with a path p_j. The path p_j will traverse every r_i once for each edge e_i that is adjacent to v_j. Traversal can be in any order.

Figure 3.4 shows the idea of the reduction and shows reduction of a 5-vertices graph. The paths in the PLA problem have the same names as the vertices in the 4-colourability problem. For each common edge in the graph, the paths go through a common switch. If two connected vertices have the same colour, it is easy to see that then also the two associated paths with the same colour/label must cross the same switch.

By choosing the number of label bits (n) as two, four label values are possible for each path (00, 01, 10 and 11). For a switch with one path per output port each path must have a different label to be distinguishable. If the path label assignment problem has a solution, set the colours of the vertices v_i in G according to the bits of the corresponding paths p_i to solve the 4-colourability problem. To show that this is indeed a colouring solution, assume that this is not a valid solution for the 4-colourability problem. Then, an edge $e = (u, v)$ exists which connects two nodes with the same colour. The paths corresponding to u and v, p_u and p_v, have the same bit mask. Since p_u and p_v are distinguishable in r_e they cannot have the same bit mask. This contradicts the assumption above that the solution is valid for the path label assignment.

If the label assignment problem has no solution, the 4-colourability problem also has no solution. Again, assume the bit mask problem has no solution but the 4-colour problem has a solution. Assign each colour a bit mask and the bit masks to the paths in the bit mask problem. Then, for each switch the paths will have different bit masks and the bit mask problem has a solution, establishing the contradiction.

Since a polynomial-time reduction is shown in both directions the PLA problem is indeed $\mathcal{NP}$-complete.

3.8 Evaluation

Our evaluation consists of two parts. The first part shows that our method of using the destination MAC address as labels without rewriting the source address (Section 3.5) works as anticipated. The second part is an empiric evaluation of the greedy heuristic in multiple network scenarios.

While our method complies with the Ethernet standards, the behaviour of the network is unusual from the perspective of an operating system. We built a test bed using Mininet [63] and connected various virtual and physical hosts with different operating systems (Windows, Linux, Mac OS X and Cisco IOS) to it and implemented our approach on the test bed. The SDN controller would then reply to ARP queries that contain label addresses instead of the real addresses and install flow rules that use these label MAC addresses with a rule on the final switch to rewrite the label address to the real physical address.

Our findings confirmed that the operating systems will accept IP packets for their own IP address as long as the destination MAC address is right. The source MAC address can be arbitrary. Or, from an Ethernet layer-centric view, the operating systems do not make assumptions about the network addresses other than the receiving MAC address should be its own address. There is no attempt made to match a packet's IP and MAC address against the corresponding entry of that IP address in the receiving node's ARP cache.

The second part is a simulation to evaluate the possibility of reducing the number of needed flow table entries by using the greedy algorithm described in Section 3.6.2.

For small or simple structured networks (without many alternative paths, e.g. trees with less than 100 switches) the greedy heuristic achieves the optimal solution with one wildcard per edge.

As a more challenging example for the greedy heuristic, we built a CLOS network consisting of two core switches and 16 pairs of distribution switches (two uplinks each). Each pair of distribution switches had 8 top of rack switches connected to it. The network has 320 links between the switches (or 640 undirected links). To test the robustness of the heuristic, we modified the graph by randomly removing links and switches. As paths, we calculated all shortest paths between all switches. Using these paths, the SDN controller can choose the exact path between two end hosts purely by answering an ARP reply and without having to install or modify any FIB entries. The resulting number of wildcards compared to edges in the network is plotted in Figure 3.5. In this complex setup, the heuristic manages to achieve an average of about 4-5 wildcards per outgoing edge. The number of required wildcards is quite stable for the modified graphs as well.

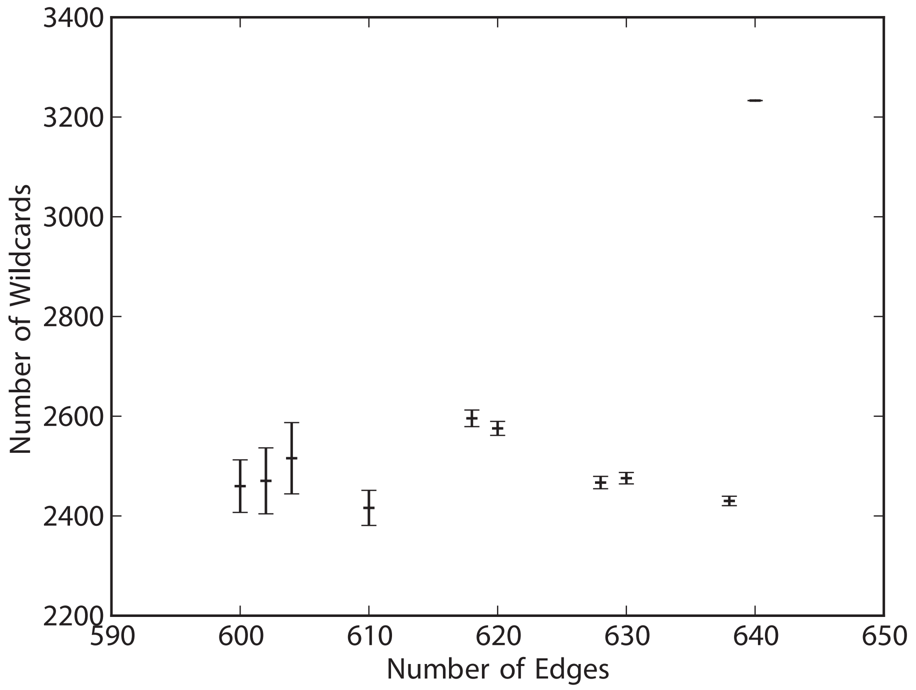


Figure 3.5: Average number of wildcards used by the greedy heuristic with 95% confidence intervals

3.9 Conclusion

We have formalised the Path label assignment (PLA) problem of finding the optimal number of FIB entries for a network and proofed the complexity of the problem.

We have shown that our techniques for reducing the number of needed flow table entries in a software-defined networks are viable. A centrally managed network makes it possible to use the destination MAC address as a very light-weight label that can be applied through ARP by the connected hosts allowing very small FIB tables and label routing.

We have provided a greedy algorithm that can be used to calculate labels for arbitrary networks.

For the goal of this thesis, the aspect of being able to redirect traffic in a better way is important. The ability to use label routing for that, even with inexpensive SDN switches, is an essential part of reaching that goal.

4 SynRace: Multi-path Routing

Multi-rooted trees are becoming the norm for modern data-centre networks. In these networks, scalable flow routing is challenging owing to the vast number of flows and paths. Current approaches either employ a central controller that can have scalability issues or a scalable decentralised algorithm only considering local information, which might only find a sub-optimal solution.

In this chapter I present a new decentralised approach to least-congested path routing in software-defined data centre networks that has neither of these issues: By duplicating the initial (or SYN) packet of a TCP flow and estimating the data rate of multiple paths in parallel, we exploit TCP's habit to fill buffers to find the least congested path. We show that our algorithm significantly improves flow completion time without the need for a central controller or specialised hardware.

This chapter is based on

- Arne Schwabe and Holger Karl. "SynRace: Decentralized Load-Adaptive Multi-path Routing Without Collecting Statistics". In: *Proceedings of the 2015 Fourth European Workshop on Software Defined Networks*. EWSDN '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 37–42. ISBN: 978-1-5090-0180-4. DOI: `10.1109/EWSDN.2015.58`. URL: `http://dx.doi.org/10.1109/EWSDN.2015.58`

4.1 Introduction

Modern data centres no longer use a single rooted tree topology with just a single path between hosts, but they move towards multi-path topologies like CLOS/FatTree topologies, which provide multiple redundant paths between end hosts. To support traffic-intense applications by providing them with a high data rate and low latency, it is critical to avoid congestion. To fulfil these demands the data rate of multiple paths between end hosts is required.

Ideally, this could be accomplished by distributing the traffic between communication partners over multiple paths and using all available capacity. Unfortunately, TCP traffic cannot be easily split arbitrarily and a routing decision has to be made to assign each individual flow to a particular path[65].

Our solution leverages the interdependence of congestion and latency. In a data centre we assume that the paths between a source and a destination are symmetrical. When a path is congested by multiple flows, the buffers along the path are being filled and this gives the path a higher latency compared to a path on which the buffers are empty. Hence, a path with a smaller latency than another path is likely to have a higher unused data rate. We reckon that we can use a path's latency as an indicator for its load and its (likely) unused data rate.

To find the path with the smallest latency between a given source and destination, we use a low-overhead mechanism to measure the latency along *all* paths:

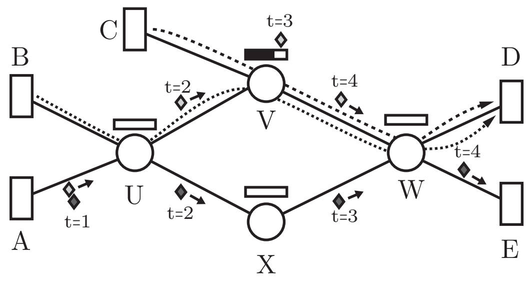


Figure 4.1: Example of SynRace

SynRace sends a TCP's connection initial SYN packet over all paths as a probe and then picks the path of the first arriving SYN probe. We assume that this is the fastest path and has the highest remaining data rate.

Figure 4.1 gives a simplified illustration how SynRace works: In this example, two flows from hosts B and C to host D are already established (dashed lines). These two flows cause a congestion at switch V, filling up the buffer of the switch. At time $t = 0$, host A decides to initiate a connection to host E and sends two probe packets (shown as diamonds) onto the two alternative paths. These travel on the upper and lower path at the same speed until at $t = 3$ the packet on the upper path is queued and delayed by the filled buffer. At $t = 4$, the probe packet of the lower path is the first probe packet to arrive at E and SynRace selects the path the faster probe packet travelled (U $\rightarrow$ X $\rightarrow W$) for the new connection. The example omits the recording of the travelled path on the probe packets and signalling back the chosen path to the sender A.

While this approach is conceptually simple (and variations of it have been introduced before), a major contribution of this chapter is the integration of this approach into SDN-based data centre networks and an evaluation using realistic traffic traces and network topologies.

The remainder of the chapter is structured as follows: We begin by comparing our approach to existing approaches in the related work section. In Section 4.3 we discuss the different sources of latency and their influence on the path latency, which is central to our approach. We then discuss the technical aspects in more detail in Section 4.4. We present an implementation of SynRace with the ubiquitous SDN protocol OpenFlow in Section 4.5. We follow up by describing some important corner cases and the applicability to WAN networks in Section 4.6 (e.g., multi-homed hosts). Finally, we give an experimental evaluation of our approach in Section 4.7 and draw a conclusion in the last section.

4.2 Related work

Routing flows in networks with alternative paths is a very old field. Unsurprisingly, there are multiple different approaches tailored to specific requirements.

The first class of approaches consists of decentralised algorithms that use only local information. A popular, robust example of this class is equal-cost multi-path routing (ECMP) [49, 76]. ECMP works by assigning each possible route on a switch a cost, usually derived from the number of hops to the destination. For each flow, one of the routes with the least cost is chosen in a random or round-robin fashion, depending on the implementation. The idea is that with enough flows the utilisation of the links equalises as a symmetric traffic pattern is assumed. Most ECMP implementations are completely agnostic to load. But even if the ECMP implementation uses locally available information and chooses the route with the least *local* utilisation, any bottleneck that is on the remainder of the route is completely ignored. A big advantage of ECMP is its simplicity and that no communication between network elements is required.

The second class of approaches uses a central instance to decide routes. By collecting statistics of utilisation of each link in the network, the central instance can make more informed decisions using global knowledge. This paradigm is especially popular in the software-defined networkings (SDNs) community [60]. This class of approaches, however, comes with its own problems: The central controller becomes the bottleneck of the network and statistics of flows and/or link utilisation need to be collected periodically or event based. The smaller the collection interval is, the more accurate these statistics become but also the higher is the load on switches and controller. Furthermore, either a separate control network between the switches and the controller is needed or the control messages and actual data traffic compete. An example of such a central flow scheduler is Hedera [30]. Hedera collects statistics only every 5 s and only schedules a subset of flows to avoid scalability problems.

Another decentralised approach is Cisco's CONGA [5]. The main idea is very similar to our approach: each leaf switch selfishly uses the least congested path in a decentral fashion. To find the least congested path, CONGA uses hardware acceleration and piggybacks queue status onto regular network packets, keeping the decentral path congestion information up to date. This requires specialised hardware supporting CONGA, which is the main difference between our approach and CONGA. We carefully designed SynRace to depend on nothing but standard OpenFlow SDN switches.

The idea to exploit the observed latency of packets to infer network characteristics is used in other approaches. Examples of approaches that estimate network performance are Packet Pairs or Packet Trains [53, 72]. By sending multiple probing packets and measure the delays between arriving packets, they estimate the queue size/link speed of the bottleneck link between sender and receiver. In contrast to our approach where we make assumptions on the network topology, these approaches do not as their focus is to estimate characteristics of a bottleneck link in the Internet with unknown characteristics. Our assumptions allow us to send only one probe packet, which gets duplicated along the path, whereas these approaches need multiple packets per path to determine the best path. In a time critical task, as is finding the path for a new flow, our approach has the advantage of needing much less time. It only needs to wait until the first packet arrives to make a decision (in contrast to waiting for all packets as the packet train/pair approaches).

A similar idea to SynRace, but realised in a vastly different scenario, is the dynamic source routing (DSR) protocol for wireless networks [57]. The idea of the protocol is very similar: Send out a probe in parallel over multiple paths and use the path on which the first probe (copy) arrives. But wireless networks have quite different characteristics than data-centre networks: wireless networks share the medium, local broadcast instead of unicast, high expected packet loss in wireless and no highly structured network (especially not completely symmetrical), making the two approaches quite different in implementation and technical details.

4.3 Latency as a proxy for data rate

Since we use packet latency delay to find the least congested path in the network, we first need to establish that the path with the smallest latency is indeed the one with the maximal residual data rate. This is due to TCP's congestion control dynamics: TCP will try to fill the buffer of the bottleneck link as part of its congestion avoidance mechanism [26, 93, 127]. Even more modern TCP variants like Data Center TCP (DCTCP) [6], which are designed to keep queue buffers small, will have some packets in a buffer. DCTCP has a parameter K to control this size, which is set between 20 and 65 in the original paper [6]. This in turn results in filled buffers on utilised links (like other TCP variants), which have a low residual data rate. Since packets travelling on paths with filled buffers will be delayed by queuing, packets on paths with empty buffers and otherwise the same characteristics will have a smaller end-to-end delay.

The question arises if the difference of queuing delays is large enough to be measurable and if it accounts for the lion's share in the observed delay difference. The delay of packets is influenced by multiple factors; we will look at the possible factors influencing the delay and evaluate how these affect our measurement.

We start by describing the queuing delay introduced by congested links in absolute time: Buffer sizes vary wildly from a few kilobytes in inexpensive top-of-the-rack TOR Gigabit switches up to hundreds of megabytes in modular core switches [21, 55]. An Ethernet packet of 1500 bytes takes between 120 ns (100 Gbit/s) and 12 µs (1 Gbit/s) to be transmitted. One megabyte of queued bytes on a 10-Gigabit link causes a delay of 800 µs. Based on these numbers, the total queuing delay is in the order of hundreds of microseconds (µs) to a few milliseconds (ms).

We compare this delay with other possible sources of delay differences and start with the physical connection. At 10 Gbps, even the delay of a single queued packet (1500 Byte, 1.2 µs) causes more delay difference than a 200 m difference in cable length ($200\,m \cdot c \cdot \frac{1}{1.62} \approx 1\,\mu s$ (speed of light in fibre optics)).

Different switch models and different forwarding strategies can influence the delay of a packet. Fortunately, these delays are identical for identical models and in a symmetrical data centre structure also the paths are symmetrical and each packet will be delayed by the same amount at each stage. Hence, this will not lead to a difference between the different paths.

In conclusion, the queuing delay dominates the delay difference of alternative paths in data centres.

4.4 Overview of SynRace

We have established in the last section that for each alternative path between communication partners in a reasonably designed data centre, the latency is the same. As hinted at in the introduction, we select the path with the lowest delay to route a flow by sending a probe packet on each path. By using only one probe packet we trade a fast connection setup time for accuracy. A single sample will not have perfect prediction, e.g., a path with bursty traffic will temporarily have empty queues but any traffic on a path increases the probability of delay and a really congested link will always have a high delay.

The order of the packets in which the packets arrive at the receiver should only depend on the queuing delays experienced by the probing packets. The naïve approach is to send one probe packet for every path from source to destination, but this approach favours the first packet sent. The solution to this problem is to start by sending only one probe packet for all paths and duplicate the probe packet only if paths diverge. When the probe packets arrive at the destination, we need a way to recover the path the packets travelled. As we start with only one packet, this information cannot be written into the packet by the sender and we need to add this information in transit. We realise this by adding different IDs to the different copies when the path branches. This allows to recover the switches traversed by the probe packet.

In contrast to broadcast networks (like wireless networks, for which DSR has been designed for), in a wired network like Ethernet the capacities of each direction are independent of each other. This entails two things: For bidirectional traffic we need to discover the best path for both directions and for each direction we need to deliver the information of the selected path from the receiver to the sender. For bidirectional flows we choose to piggyback the information about the best path to the first packet going in the opposite direction (typically the SYN/SYN-ACK packet).

SynRace has been designed to be a decentralised algorithm. For our approach, each switch can have a different (local) controller to ease scaling. We assume that every local controller has learned the network topology and that all switches have agreed on other fixed details like switch IDs. We consider this information exchange to be outside the scope of SynRace since they take place on large timescales and efficient algorithms for these problems are widely known. Other than these –almost static– information, the switches exchange information only through the probe packets. The actual chosen path must also be signalled to the other side of the flow. We try to piggyback that information as much as possible on the probe request themselves. See the discussion in Section 4.5.4 for a detailed discussion.

4.5 Implementation in OpenFlow

In this section, we present an implementation of our approach for the SDN protocol OpenFlow [69] (see Section 2.3).

For our implementation, we differentiate between two different types of packets: probe packets and normal packets. Normal packets belong to an already established connection and are sent over one path. Probe packets, on the other hand, are the packets that are duplicated over the possible paths and need special treatment. For a bidirectional connection like a TCP flow, probe packets in both directions are needed.

An SDN switch will perform a lookup of a packet's header in the forwarding table for each incoming packet. When a forwarding-table miss is triggered and the SDN controller is involved, the packet processing delay is often in the range of milliseconds to seconds and has a large variance. This voids our assumption that the measured delay is dominated by queuing delays. Therefore, our first objective is to avoid forwarding-table misses for probe packets at all cost. We need to install all packet rules needed to forward the probe packets before a probe packet is sent.

On the other hand, we also use the forwarding table miss mechanism on the *egress* switch to select the best path. By *not* installing flow rules for probe packets on the egress switch, we force this specific switch to forward all probe packets to the (local) controller and compute the path to install from the first arriving flow miss message.

Since timing constraints force us to ensure that forwarding is done in hardware, all packet modification is also limited to hardware features. This brings us to the next key design aspect of SynRace. The copies of the packets that arrive at the destination switch need to carry information about the travelled path. Otherwise, the receiver of the racing packets cannot differentiate the copies and cannot determine the path the packet used.

The conclusion is that we need to construct the forwarding tables entries such that these –without falling back to the controller– will

- forward and duplicate probe packets
- modify the probe packets to record the travelled path.

The following subsections will address each of these challenges.

4.5.1 Forwarding and duplicating the probe packets

Forwarding and duplicating the probe packets is astonishingly simple in OpenFlow. Instead of performing a lookup of the destination and setting the output port to one of the possible paths, the output port is set to the list of all ports where paths continue. This step also ensure that the copies of the incoming packet are processed and enqueued to the output buffers at the same time.

4.5.2 Triggering the race

There are multiple events that can trigger the sending of probe packets. The most basic event is the arrival of a new flow. An extremely fine grained way to deal with new flows is to send out probe packets for every new flow; the flow rules can be set up to forward every unicast packet not matching any other rule to initiate the SynRace by modifying the packet to be a probe packet. A less aggressive way is to first forward the packets to a local controller. The controller can then decide if a race should be started or if a cached result should be used.

A controller can also make educated guesses whether a flow might be big/important enough to warrant the sending of probe packets: For example, if the destination port belongs to a Map-Reduce framework's file system, a large data transfer can be expected. In contrast, a small HTTP flow that has a request and an answer that fit into one packet (e.g., to poll a status or check for updates) creates a lot of overhead and probe packets for this flow might not warrant sending probe packets.

Instead of or in addition to triggering the sending of probe packets by a new flow, the controller can also decide to send unsolicited probe packets. For example, the controller might decide to poll every few seconds for the best path to a few key communication partners to consider rerouting if the situation significantly changes.

4.5.3 Adding path information

To label the probe packet with hops of the path, we have to overcome a few challenges that are imposed by the capabilities of the OpenFlow standard and the switch hardware. To record the path of a probe packet, we need to modify the packet at each hop to be able to differentiate one packet copy from other packet copies that travelled other paths. Packet modification in OpenFlow is described by actions. Unfortunately, the only allowed actions (in all current OpenFlow versions (1.0 – 1.5)) are to replace/add a header (field) or to keep a header (field) unchanged; there is no possibility to reference an old header value in an action (the only exception are in/decrements of hop count fields). To summarise, we have to add/modify a header or a combination of headers to create a path label.

Headers that can be reused for the label consist of unused headers and headers that can be overwritten and restored to their original content in the last switch. Headers that qualify are, for example, the source and destination MAC address, MPLS labels and the VLAN ID.

Adding switch IDs can either be done by setting a new label on each hop of the path or by modifying an existing label to contain two or more IDs. To put two IDs into one header, we need to change that header from $\text{id}_{\text{switch1}}$ to $(\text{id}_{\text{switch1}}, \text{id}_{\text{switch2}})$. Since we cannot reference the first ID in the action, a separate flow entry for each possible value of $\text{id}_{\text{switch1}}$ is needed. Encoding three IDs like $(\text{id}_{\text{ToR}}, \text{id}_{\text{Distribution}}, \text{id}_{\text{Core}})$ into a header forces a switch to have forwarding table entries for all possible combinations of the first (id_{ToR}) and second ($\mathrm{id}_{\mathrm{Distribution}}$) part of label. We can work around this limitation by using more than one header for the label, e.g., using the MAC source address for the first and second hop and the MAC destination address for the third and fourth hop. We can further reduce the need for label space: When the probe packet arrives at the egress switch it is forwarded to the SDN controller. The switch forwarding the packet is obviously the last switch. The SDN controller can infer the penultimate switch by looking at the ingress port of the packet.

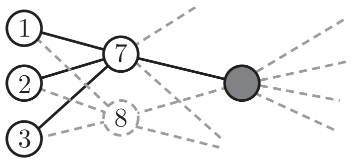
We now need to combine label adding and forwarding into forwarding table rules. Simple merging creates a large Cartesian product and increases the number of rules dramatically. Fortunately, OpenFlow (1.0 with extensions and 1.3+), allows to chain tables, allowing us to keep labelling and forwarding separate.

Figure 4.2 shows an example. Part of an example network and an excerpt of the forwarding tables for the highlighted switch (grey) are shown. The first three rules in Table 1 handle probe packets received on port 7: Depending on whether the probe has traversed switch one, two or three, the probe packet has been labelled either `f10000`, `f20000` or `f30000`. The rules in the first table will modify this label to also record the hop of the switch seven and delegate adding the forwarding actions to the second table. The second table will just check that a packet is indeed a probe packet (label begins with `f`) and then forward and duplicate the probe packet.

Since label combining needs the Cartesian product of path inputs and possible labels in the first table, it should be on switches where the possible number of input label combinations and ports are small, e.g., from the core switches to the distribution switches where the Cartesian product size (number of uplinks of a distribution switch times the number of core switches) is usually below twenty.

The total number of rules depends on how paths are calculated and how forwarding table rules are implemented. We give a calculation for a typical 3-tiered CLOS data centre network (core, distribution, ToR) in which all shortest paths between two hosts are allowed and structured addresses are used (e.g. pod and tor is encoded explicitly into the destination MAC address as in Portland [81] or implicitly as presented in Chapter 3 or encoded into the IP address like using 10.0.0.0/8 with the semantic 10.pod.tor.host). In the simpler first case, where combining IDs in labels is not used, each core switch needs one rule per pod: lookup pod ID from address, add own ID as label and forward packet to all ports that lead to that pod. Distribution switches need one rule per ToR and one rule that forwards probe packets to all (connected) core switches when the host is not local. ToR switches just need the rules to forward the probe packet to the controller. In the more complicated case where labels are combined, the switches that need to combine label need to add the number of the Cartesian product as explained above.

In summary, the number of rules that are needed for SynRace is very small and even if structured addresses are not used in normal forwarding table entries (for non probe packets), the structured addresses can be used for SynRace as probe packet forwarding is separate from normal forwarding. The rules on the ingress switch for the SynRace probe packet will transform the normal addresses



Forwarding table 1

Match	Action
label=f1000,inport=7	set label=f10700, next table=2
label=f2000,inport=7	set label=f20700, next table=2
label=f3000,inport=7	set label=f30700, next table=2
label=f3000,inport=9	set label=f30900, next table=2
...	...

Forwarding table 2

Match	Action
label=f****,dst=10.2.0.2	set outport=1,5
label=f****,dst=10.7.0.5	set outport=2,4
label=f****,dst=10.9.0.7	set outport=1,2
...	...

Figure 4.2: Example network and OpenFlow forwarding table setup for SynRace

into structured addresses. As the probe packet will be sent to the controller on the egress switch, the structured addresses are completely hidden from the end hosts.

4.5.4 Installing the flow entries

We have described how to choose the path based on the delay of the SYN packet (and return path based on SYN/ACK). But the information of the chosen path is only present at the last hop of the path (usually the ToR switch of the destination). For TCP flows we can piggyback the information on the probe packet containing the SYN/ACK packet, which is racing in the opposite direction. The controller at the other end can then retrieve the meta information, install the flow entry and forward the SYN/ACK packet to the connected host.

It is important to do the steps in the right order, in particular that the flow rule is installed before the SYN/ACK is delivered to the destination host. Otherwise, the arrival of the SYN/ACK packet moves the TCP's connection into the ESTABLISHED state; it could then start sending packets while the flow rules we have just computed are not yet installed, triggering needless flow table misses and PACKET_IN messages to the controller.

Installing the flow rule on the receiver side is more challenging. As indicated

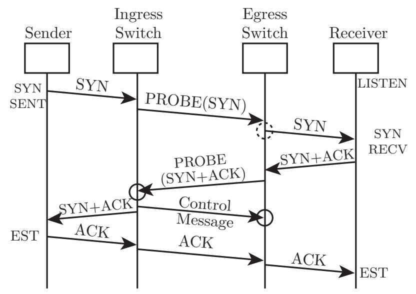


Figure 4.3: Message sequence chart for installing flow entries and three way-handshake, circles indicate the points in time when a flow rule is installed at the respective switch

in the previous paragraph, the information how to route the direction to the sender is only available after SYN/ACK has been received by the sender. The natural solution is to piggyback the path information to the first packet of the sender (the third packet of the handshake). But since the connection is in the ESTABLISHED state, the sender and the receiver start sending packets triggering flow misses at the receiver's switch.

A cautious solution is to delay the delivery of the first ACK until the flow is installed. Instead of delivering the SYN/ACK as soon as possible (and triggering the sender's ACK packet) the sender side's switch can send a control message containing only the path information and wait for an acknowledgement. This, however, comes at the cost of introducing an additional round trip time.

Instead, we can opt for a more opportunistic solution and live with the race condition but alleviate its effects as much as possible. We do not wait for the acknowledgement of the control message but instead avoid risking packets to be forwarded to the controller by installing a temporary rule as soon as we see the SYN/ACK packet. Figure 4.3 illustrates the information flow, TCP state on sender/receiver and order of packets with a message sequence chart for this opportunistic approach using a temporary flow rule.

Falling back to the temporary route, the packets are forwarded over the default path until the controller of the local switch can install the specific flow rule. Using this strategy can result in the first packets of a new connection to take another path than the following ones. This can result into packet reordering since the path selected by the probe packet is most likely faster and the packets of the second path might arrive earlier. Such reordering is known to negatively affect the performance of TCP [65]. In our own implementation and evaluation, we decided to use the first approach to avoid this effect.

4.6 Corner cases

This section looks into some corner cases and how SynRace affects them.

4.6.1 Asymmetric networks

Even though routing of individual flows is mostly done in a data-centre environment, there is no compelling reason why individual flow routing should not be done on campus or WAN network. In such networks, alternative paths typically have unequal delays and determining the least congested path by looking at the order of received probe packets is not sufficient anymore.

This can be avoided by adding accurate timestamps on probe packets on the last switch. Using an uncongested reference measurement for the links can be used to determine the difference in latency of the paths. By subtracting these times from the timestamps of the probe packets, we can infer the lowest queuing delay. Such a measurement can be produced in an active network if the switches have priority queues (QoS) that have priority over all other queues. Unfortunately, the `PAKET_IN` message of the current OpenFlow protocol does not have a timestamp field but such a field could easily be added. This then needs hardware modifications, which our approach is designed to avoid. But since only the last switch needs the hardware modification, adding a (small) extra switch per WAN node and keeping the other switches unmodified allows SynRace to be also used in this scenario.

4.6.2 Multi-homed end hosts

In data centres, reliability is often an important aspect. To protect against devices failures, end hosts are often connected to two ToR switches instead of one. When one link is only used as a backup, the scenario is the same as a single connected host scenario, i.e. nothing changes.

In this scenario the end hosts uses both links with a link aggregation like 802.3ad [49]. To avoid reorder problems, an individual flow is sent only to one of the links, often by some hashing algorithm. The host therefore controls the first hop link and thus chooses the first hop ToR switch. This implicit decision limits the number of paths to choose from but data-centre networks with dual-homed hosts have more redundant paths than normal data centres. Instead of one destination ToR switch, there are now two ToR switches. Ideally, both ToR switches would have perfectly synchronised clocks to timestamp the `PACKET_IN` messages and would be controlled by the same controller; this would allow to handle them like a single switch by sorting all probe packet's `PACKET_IN` by their arrival time and selecting the probe packet with the earliest time.

The order of the received probe packets can only be determined for each individual switch. A simple solution is to pick one ToR switch at random and only process the probe packet from this switch, effectively ignoring the pathes that contains the other switch. If both ToR switches are controlled by the same controller the controller might have additional information to make a more educated choice than blindly choosing at random.

Picking the destination ToR switch can also be done at different stages in the path selection process. The earliest selection point is to send the probe packets only to one of the destination switches. Alternatively, both end switches can send their result to the origin switch and let the controller assigned to that switch choose.

4.6.3 Effects on small flows

Having a large residual data rate is clearly beneficial to large flows. Small flows, which consist of only a few packets, are common and have no obvious benefit from a high data rate but are typically sensitive to latency. Sending probe packets for such small flows generates overhead. Knowing in advance if a flow is large or small is difficult, in many cases even impossible. We argue that we do *not* need to make this distinction since our approach is –although perhaps counter-intuitively– beneficial for small flows in typical scenarios.

Our approach duplicates the first packet of each direction. By choosing the path with the lowest latency, the expected average latency and completion time for the flow is lower than with a conventional path solution. For latency-bound small flows, this is a major improvement, also shown in our evaluation in the next chapter. In summary, we trade a higher network load for a better and more reliable performance of smaller flows.

4.7 Evaluation

For our evaluation we used Mininet [63] to emulate a typical multi-rooted tree data centre topology with a CLOS topology. We compared our approach with an ECMP implementation. We have chosen to compare our approach with ECMP since ECMP has also the advantage of having no central instance that decides which path a flow takes and scales well with the number of switches; it is the most established approach for using multiple links in a data centre. Other approaches (central and decentral) that improve data centre performance compare themselves with ECMP. As metric we use flow completion time as the flow completion time is a direct consequence of the data rate each flow experiences and the vast majority of flows have a fixed amount of data that they transfer rather than a fixed time that they run. So any improvement in data rate that SynRace gives should be directly visible in completion time. Furthermore, in data centres, completion time of jobs is the more interesting metric (e.g. compared to load on the network).

In our first scenario, we use large flows randomly between the hosts of the network. All flows have the same size. This traffic pattern is unrealistic but is comparable to the evaluation of similar approaches (e.g., Hedera [30] uses fixed 500 MB sized transfers for the simulation of the shuffle phase). We simulated different load levels by varying the number of flows; at a load level of 1 the data rate of the generated flows equals the bisection bandwidth. As can be seen in Figure 4.4, our approach significantly improves flow completion times. Improvement of the average flow completion times is between 6% (at 20% load) and 25% (at 100% load).

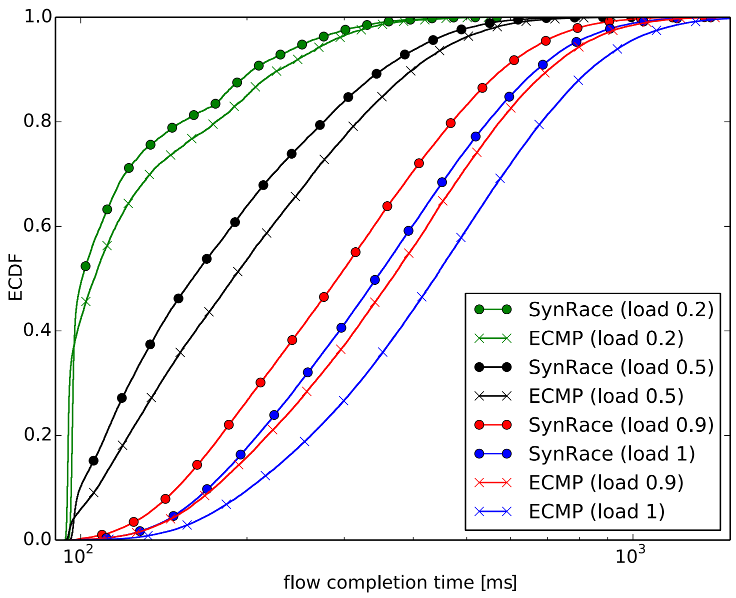


Figure 4.4: Empirical CDF of flow completion times under different loads

To simulate realistic data-centre traffic, we used the data centre traffic generator DCT^{2}Gen [126] and also confronted the algorithms with realistic data. Realistic traffic has a lot of very short flows and only a few large flows and a huge number of flows in total; these parameters favour ECMP as ECMP performs better with a large number of flows as a large number of flows is more likely to be evenly distributed over all links than the a small one. Nonetheless, as seen in Figure 4.5, SynRace manages to improve the completion time for all flows over ECMP. Overall, SynRace reduces average completion time by 5% even in this very ECMP-friendly scenario. Even centralised approaches (e.g. Hedera [30]) have only similar or smaller gains with more available knowledge.

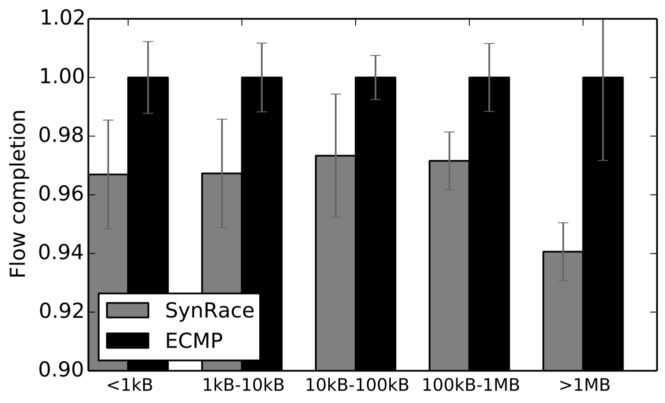


Figure 4.5: Comparison of flow completion times for realistic traffic using ECMP as baseline

4.8 Conclusion

We have shown that our method of cleverly using probe packets in SDN networks can choose optimal paths without needing specialised switches. It significantly improves flow completion times across all kinds of flows. Furthermore, SynRace is well isolated and can be used in conjunction with other techniques, which makes SynRace a candidate to further improve the quality of other approaches. In my thesis, this approach is particularly interesting because it provides a low-overhead method to collect information needed to make useful traffic engineering decisions.

5 Composition

This chapter defines and discusses composition of software-defined networking applications and shows the theoretical and practical approaches to composition in software-defined networks and explains the challenges associated with it. I explore feasibility of OpenFlow as an Application Programming Interface (API) for a composition engine and argue that its design as a Southbound controller interface makes it unsuitable for this task.

5.1 Introduction

Traditionally, network policy management is done manually: network administrators translate high-level network policies into low-level network configuration commands. Policy changes hence take a long time to plan and implement. Even with careful planning, side effects can be overlooked. Therefore, problems are typically detected only at runtime when users unexpectedly loose connectivity, security holes are exploited, or applications experience performance degradation [91].

Software-defined networking (SDN) promises higher flexibility in the way networks are managed. However, as we have seen in the previous chapters, by introducing user-defined software in networks, we also introduce the complexity and dangers of modern software systems into them.

One of the emerging problems in SDN is the heterogeneity regarding network applications, as many different entities want to push network rules to an SDN network, be it from traffic engineering (TE) or from an SDN application running on an SDN controller or by currently developed standardised interfaces to an SDN controller; these interfaces include the notion of an agent that accesses network devices – as proposed by Interface to the Routing System (i2rs) and other working groups (WGs) in the Internet Engineering Task Force (IETF). These interfaces foresee multiple applications accessing the same SDN controller. Similar to concurrent programming, a series of issues related with multiple access arises, including situations where several applications produce "conflicting" configurations. For example, a firewall can tell the network to drop a flow while a load-balancer will instruct the network to redirect the flow to a specific host. Intuitively, the firewall should be given higher priority and the end result should be to drop the packet; making this decision automatically requires a *composition logic*.

Contemporary approaches to defining and dealing with a "conflict" suggest that a generalised composition logic is not possible. In this chapter, I provide a new framework to describe the interactions between applications and the network based on transactions and examine what approaches to composition may make the problem tractable.

I start the next section by defining and explaining the different approaches to composition in software-defined networks in Section 5.3. I will continue by presenting general strategies to handle and implement composition in Section 5.4. I will look into the specific challenges of using OpenFlow in composition in Section 5.5. I look at related work in Section 5.6 and provide conclusions in Section 5.7.

This chapter is based on

- Arne Schwabe, Pedro A. Aranda Gutiérrez and Holger Karl. "Composition of SDN applications: Options/challenges for real implementations". In: *Proceedings of the 2016 Applied Networking Research Workshop.* ACM. 2016, pp. 26–31
- The chapter "Composition of network applications" in the journal paper: Elisa Rojes, Sergio Tamurejo Roberto Doriguzzi-Corin, Andres Beato, Arne Schwabe, Kevin Phemius and Carmen Guerrero. "Are we ready to tackle Software Defined Networks? A Comprehensive Survey on Management Tools and Techniques". In: *ACM Computing Surveys* (to be published)

  I was the lead author of that chapter.

5.2 Definitions

For the scope of this chapter, a software-defined network is a collection V of interconnected nodes (switches) forming a graph $G = (V, E)$, where E describes the connectivity between the switches. We define the *state* N_v of a node $v \in V$ as the state of its Forwarding Information Base (FIB). A FIB entry is defined as a tuple (p, m, i) specifying a priority p, a match m that specifies which packets this entry applies to, and a list of instructions i. A typical FIB entry used in IP forwarding is (100, {ingress_port==*, dst_ip∈{192.168.100.0/24}}, set {src_mac 00:00:00:ab:cd:ef, dst_mac aa:bb:cc:00:11:22, egress_port 3}). The network state N is then defined as the collection of all node states

$$N = \{N_v \mid v \in V\}$$

The network state can be changed by a command C. A command is a sequence of basic commands $C = [c_1, c_2, \ldots]$, $c_j \in \{F_{\text{inst}}, F_{\text{del}}\}$ that each modify the FIB of a switch:

- $F_{\text{inst}} = (v, p, m, i)$: Install a FIB entry on node v with priority p, match m and list of instructions i.
- $F_{\text{del}} = (v, p, m, i)$: Remove the FIB entry that was previously installed by $F_{\text{inst}}(v, p, m, i)$

We define the function $a\colon N \times C \to N$ as the function that applies a network command to a network state and produces the new network state:

$$a(N, [c1, \ldots, c_k]) = a(a(N, c_1), [c_2, \ldots, c_k])$$

$$a(N, c_j) = \begin{cases} \text{Add (p,m,i) at } v & \text{if } c_j = F_{\text{inst}} \\ \text{Remove (p,m,i) at } v & \text{if } c_j = F_{\text{del}} \end{cases}$$

for a basic command $c_j = (p, m, i, v)$.

In a network, these commands are generated by a control application or *module* M_j. We define a network module as a state-based function M_j that reacts to an event ev with a network command (and possibly modifies its internal state). State is here not used in the pure mathematical sense but that M_j can also have hidden state, for example remembering the network state or previous calls.

$$M_j : ev \to C$$

Examples for network events are the arrival of certain packet types (like ARP requests) or the arrival of a new flow currently not matched by any entry in the FIB.

5.3 Types of composition

The goal of composition is to run multiple modules on the same physical network and incorporate all their network commands into the network state.

5.3.1 Single module without composition

We start with the simplest case of a single module M_1 and an event ev. All commands are simply forwarded and applied to the network and the resulting network state N' is:

$$N' = a(N, M_1(ev))$$

5.3.2 Multiple modules without composition

A typical SDN controller runs multiple SDN modules and, commonly, all outputs of these modules are applied to the network without any explicit form of composition. In this case, each command is applied to the network as it happens, just like in the single-module scenario. The resulting network state $N^{(k)}$ for k modules all reacting to the same event ev looks like this:

$$\begin{aligned} N' &= a(N, M_1(ev)) \\ N'' &= a(N', M_2(ev)) \\ N^{(k)} &= a(N^{(k-1)}, M_k(ev)) \end{aligned}$$

The simplicity of this approach is also its biggest problem. Since every network command is applied when it happens, the results depends on the order of commands applied; $a(a(N, M_1(ev)), M_2(ev)$ is not necessarily the same as $a(a(N, M_2(ev), M_1(ev))$.

As an example, Module A sends the commands 'delete flow rule for 1.2.3.0/24; install new flow rule for 1.2.3.0/24' and module B sends the commands 'delete flow rule for 1.2.3.0/24; install new flow rule for 1.2.3.4/32". If applied in that order the second command will delete the flow installed from the first command set and only the rule for 1.2.3.4 is installed. If the second set of commands is executed first, the result is that both flow rules are installed.

Both cases might occur in a non-deterministic fashion in the same network, for example caused by differences in execution speed of modules M_1 and M_2.

Another problem are transient states. In the time after the first module has answered but not the second, the transient network state N' is active. This transient state is problematic since it only reflects the output of the first module but not the others. These ill-specified, non-deterministic transient network states are usually undesirable and constitute the main reason to explicitly define a composition logic.

5.3.3 Multiple modules with harmonising

The output of multiple modules might contain conflicting or overlapping commands. For example, two modules might instruct one switch to deal with the same packet by either forwarding or dropping it, at the same priority (see Section 5.4 for details). To deal with such conflicts, a stateful function

$$h\colon \text{command} \to \text{command}$$

can be used to modify or replace the commands. Stateful in this context means, in the same way as with the modules M_j, that h is not a pure function in the mathematical sense but that h but can have hidden state, for example remembering the network state, or previous calls.

The resulting sequence of network states can be expressed as:

$$\begin{aligned} N' &= a(N, h(M_1(ev))) \\ &\vdots \\ N^{(k)} &= a(N^{(k-1)}, h(M_k(ev))) \end{aligned}$$

An example of a harmonising function is the OpenFlow network hypervisor FlowVisor [111], which restricts control of individual modules to parts of the network (see the related work Section 5.6 for more examples and in-depth discussion). In this example, h would change the commands of each module to affect only the assigned part (by a user configuration) of the network. Commands that only affect parts of the network the module is allowed to interact with can be transmitted without change; commands that affect only parts of the network the module is not allowed not interact with are simply dropped. For commands that affect both assigned and not-assigned parts of the network, the function h needs to replace the commands with commands that affect only the assigned part. This can be as simple, as adding a condition on a VLAN tag if the module was assigned only to work on a particular VLAN. But complex harmonising will require also very complex modification of the commands.

Naturally, the interaction of a harmonising function may render a module non-working, especially the silent (from the module's perspective) dropping of commands. A simple learning switch will work fine when forcefully restricted to a certain subset, a more complex module might not.

When the harmonising function h is the identity function we get the same result as in the previous subsection.

This harmonising function can be used to allow multiple network module on the network and implement a composition logic that allows to modify the commands to prevent side effects but intermediate, hard-to-predict network states still exist. It is hence not a satisfying solution for all use cases. In the example of the hypervisor, the intermediate network state causes little to no problem as the intermediate states are only visible in the network as a whole and not visible in each partition.

5.3.4 Parallel composition

To overcome the problem with transient states and varying order of applied results, parallel composition collects all commands and then resolves all conflicts between these commands, composing the results into a *single* command to be applied to the network. This is done by a special *resolving function* r that gets *all* command outputs and generates a conflict-free version of these commands that can be applied to the network.

$$r\colon \text{command} \times \ldots \times \text{command} \to \text{command}$$

The new network state N' can then be expressed as:

$$N' = a(N, r(M_1(ev), M_2(ev), \ldots, M_k(ev)))$$

This composition requires all command outputs of an event to be available; it must also be possible to tie a command output of a module to a specific event (necessary when multiple events are passed to a module before commands have been produced, compare challenges of OpenFlow, Section 5.5).

A big difference between the parallel composition and the harmonising composition is that the parallel composition is reactive, i.e. it depends on the fact that the network commands generated by the modules are a response to a network event. The harmonising composition works without this assumption and can also be applied to network commands that are sent proactively without an event ($C = M_j(\emptyset)$). Just using r as an h function would not work. The signature of the function requires an event, and multiple outputs from the same network event. Network commands that are sent proactively have no corresponding events from the other modules. If we introduce a pseudo event and set all other outputs to an empty set, to match the signature of r, we also need to add special handling for pseudo events into r; this is basically adding a harmonising function to r.

5.3.5 Serial composition

For the serial composition, one module is fed the output of a previous module. The desired network state of the first module only exists as an input to the second module.

Instead of operating only on its state and the network event, a module in this scenario also operates on the command output of the previous module, implicitly on the state of the previous module. To support this behaviour, we need to change the signature of the modules to accept a command as an additional input:

$$M\colon \text{event} \times \text{command} \rightarrow \text{command}$$

With that, we can define, for two modules M_1 and $_2$, their serial composition ($\circ$) as new function M_{12}:

$$M_{12}\colon ev \mapsto (M_1 \circ M_2)(ev, \emptyset)$$

and use that in place of a normal module function (in the harmonising or parallel composition). For example, if only the serially composed function is used, the new network state will be:

$$\begin{aligned} N' &= a(N, M_{12}(ev)) \\ &= a(N, (M_1 \circ M_2)(ev, \emptyset) \\ &= a(N, M_2(ev, M_1(ev, \emptyset))) \end{aligned}$$

Chaining more than two network modules, e.g. $M_1 \circ M_2 \circ M_3$, is defined by obvious induction.

The main distinction of the serial composition from the parallel composition is that the last module in the composition chain can provide a consistent set of network commands. It also allows a module to incorporate the decisions of a previous module into its own decisions. The downside of the serial composition is, however, that modules need to be explicitly designed and programmed to be used in this way. Input and output need to have the same format. This is one of the reasons that network programming languages like Pyretic [96] that implement a chaining of functions, define their function signature to have symmetrical input and output: $f\colon \text{policy} \rightarrow \text{policy}$. We will take a look how useful serial composition is with existing modules in Section 5.3.7.

5.3.6 Network emulation

To be able to define a useful serial composition semantic, we need to explain network emulation. While not directly related to composition, it is a useful tool to implement a more complex composition logic. The idea is to implement a network emulator that understands and can parse the network commands (e.g. OpenFlow) and emulates a real network. The implementation of the network emulation is a specialised software switch. The software switch will parse and execute the network like a normal switch. The emulated switch or switches can then be used to see what effects certain packets or network commands have. The network command is sent to the network emulation and the results/changes to the emulated network are observed. By configuring the emulated network to mirror the real network, this can be used by a composition logic to answer how packet and flow would be affected by network commands without installing them in the network, e.g., determining if a packet would be dropped or how a packet would be modified.

5.3.7 Approximate serial composition

As most network modules are not designed for serial composition (i.e., they do not accept a command as an input), we define an approximate way to do serial composition with existing network modules. In this scenario, we need to incorporate as much as possible from the network command into the input event of the following module by a function

$$\alpha \colon \widetilde{N} \times \text{command} \to \text{event}$$

where $\widetilde{N}$ is the *approximated* network state resulting from applying the output of the first function to the current network state. This approximated state is a representation of the state in the controller; its manipulation does *not* involve manipulation of the actual state in network devices.

What can be incorporated into the new event is often very limited as we will see in Section 5.5. The new network state using this function can be expressed as:

$$\begin{aligned} N' &= a(N, (M_1 \tilde{\circ} M_2)(ev)) \\ &= a(N, M_2(\alpha(\widetilde{N}, M_1(ev)))) \end{aligned}$$

We use the $\tilde{\circ}$ here instead of $\circ$ to emphasise the difference from serial composition, denoted by $\circ$, and the approximate serial composition denoted by $\tilde{\circ}$.

Similarly, chaining three modules in an approximate serial composition works as well:

$$\begin{aligned} N' &= a(N, (M_1 \tilde{\circ} M_2 \tilde{\circ} M_3)(ev) \\ &= a(N, M_3(\alpha(\widetilde{\widetilde{N}}, \alpha(\widetilde{N}, M_1(ev))))) \end{aligned}$$

where $\widetilde{\widetilde{N}}$ is the approximated network state resulting from applying $M_1(ev)$ to $\widetilde{N}$. Extension to longer composition chains is again straightforward.

5.3.8 Using overlays for approximate serial composition

A conceivable variant to implement this approximation of the network state and the function α is to use an overlay of virtual switches to a real network as shown in Figure 5.1: For each physical switch, a number of virtual switches corresponding to the number of modules is emulated. Each module is assigned

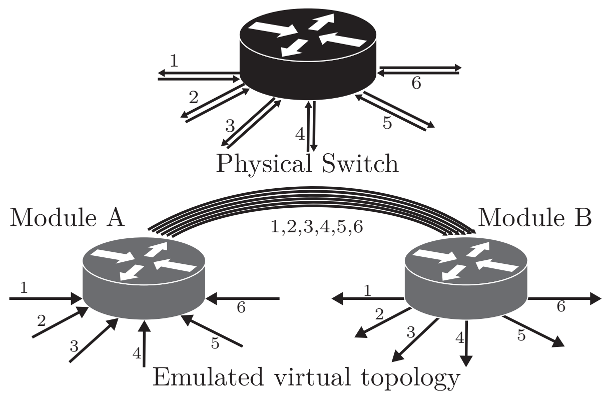


Figure 5.1: Using a virtual overlay network for composition of module A and B

to one virtual switch. The approximated network states are the state of the virtual switches and the function α would "process" the packet that the module sends to its virtual switch output ports as an event for the next module.

The advantage of this overlay semantic is that it is easier to understand how the approximate serial composition works and how the state of the first module affects the second module. The downside is that this approach looses flexibility as the transformation of output from one command to another is fixed. The other disadvantage is that if no extra steps are taken to hide the virtual topology from the modules, the modules can adapt to this virtual topologies, for example taking the extra virtual links into account for path calculations.

5.3.9 Composition and order of middle boxes

Often advanced network functions like intrusion detection systems (IDSs) and firewalls are implemented in dedicated hardware. These networking devices transform, inspect, filter, or otherwise manipulate forwarded traffic. As this hardware is often placed between two other components (such as routers or switches), the hardware boxes are often called middle boxes. When deploying SDN, the functionality of these middle boxes is often replaced by a software implementation on the SDN controller, in our terminology a module.

A common misconception is that the placement of (physical) middle boxes always carries over to the composition order. If multiple middle boxes, for example a firewall and a monitoring/IDS system, should act on all the same traffic, these boxes are set up in sequence to pass traffic to one box after another. Figure 5.2 shows an example of a traditional middle box setup with an IDS, a firewall and a load balancer. This also forms an implicit serial composition as each middle box gets the output of the previous middle box as input.

As SDN offers more flexibility here we can either set up these functions in serial or in parallel composition. Usually, for a composition with SDN the

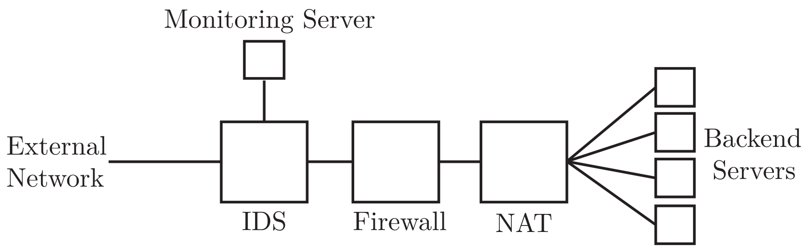


Figure 5.2: Typical order of IDS and Firewall and NAT load balancer middle boxes in a traditional network

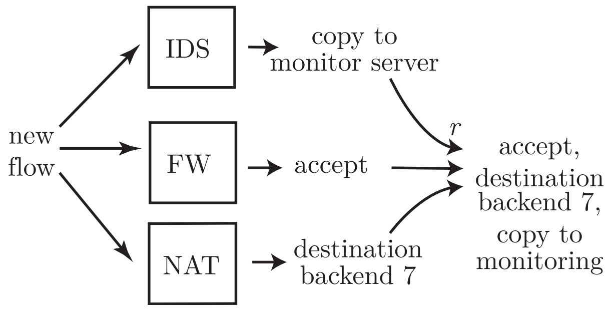


Figure 5.3: Parallel composition of IDS, FW and NAT load-balancer modules

modules would be set up in a parallel composition to allow all modules to base their decisions on the original input packets. The merging process of all the outputs will then give an equivalent solution to the middle box solution. Figure 5.3 shows the setup of Figure 5.2 implemented with parallel composition. The parallel composition here allows each module to be implemented in a simpler way since it does not need to interpret the output of a previous module.

5.4 Composition strategies

For the “true” serial composition, the mechanics required of the composition framework are simple and implemented in the modules themselves. For the more challenging approximate serial composition, we will discuss strategies in the OpenFlow implementation section.

For the remainder of this section, we look at some general composition strategies to implement the resolving function r for parallel composition. We concentrate on resolving multiple flow install commands since it is the most interesting composition part and the ideas used here can be used analogously for resolving other basic commands.

Our idea is to handle as much as possible in a generalized way but allow to fall back to developer-specified logic where a general approach cannot work. For this strategy, the function r performs the following steps:

1. Check for syntactic and general conflicts
2. Check for developer-specific conflicts, optionally using additional invariants specified by a user
3. If no conflicts are detected, perform generic composition
4. If a generic composition is not possible, abort or call developer-provided conflict resolution or, if not available, abort

The first step is to check for conflicts. If two commands do not act on the same switch, they do not conflict. Also, if one command has a higher priority than the other, the one with the higher priority "wins" and the other command is dropped. For the remainder of the section, we will consider two FIB install commands that act on the same switch and have the same priority.

Both basic commands c_1 and c_2 have a match m_1 and m_2, which in the general case are not identical. Hence, we have three different matches to consider for the composition. The match for packets matched only by $m_{1*} = m_1 \setminus m_2$, the analogous match $m_{2*} = m_2 \setminus m_1$, and the match for packets that are matched by m_1 and m_2: $m_{12} = m_1 \cap m_2$. For the generalized approach, we assume that network modules respond to a new flow event with a FIB install command that also matches the new flow. It directly follows, since both m_1 and m_2 need to match at least the new flow, that the common match m_{12} is not empty and only for the common match m_{12} we have instructions from both modules for the new flow of the event. As an example, one module might want to install policies per IP address while the other module installs policies per network. For the generalized approach, we therefore opt to ignore the matches m_{2*} and m_{1*} and only generate a new FIB install command for the composed rule on m_{12}. A new flow that falls under the match m_{1*} or m_{2*} will trigger a new flow event and we restart the composition with its new flow event.

For the instruction list of the install command, the general idea is to combine both instruction lists into one big list of instructions. When combining these lists, we can encounter different conflicts in the combined list. We differentiate these into semantic and syntactic conflicts. Syntactic conflicts can be automatically detected, like two instructions setting the same fields to two different values. As an example, a misconfigured composition enables two load-balancing modules and both try to rewrite the destination IP address of a packet to two different server IP addresses. Different instructions can also be mutually exclusive, like removing the VLAN tag and the same time changing the VLAN ID, or dropping the packet and any other action that modifies the packets. These syntactic conflicts can be detected by the generalized approach.

Semantic conflicts, in contrast, are not automatically detectable by a general approach but still cause problems. Assume again two load-balancing modules: the first module tries to redirect to a different TCP port but leaves the IP address unchanged and the second module sets a different IP destination address. Since no syntactic conflict exists, the action lists can be merged and will redirect the packets to an IP/port combination that will not work. The only way to detect such conflicts is to additionally call developer-provided logic on each conflict detection to detect these conflicts.

5.5 Composition with OpenFlow

OpenFlow is the protocol most commonly used in real-world SDN deployments and a lot of existing application logic is implemented using OpenFlow protocols. This makes OpenFlow desirable as a protocol underneath of composition and conflict resolution. On the other hand, OpenFlow itself was never designed to be used in a composition context. OpenFlow itself only allows multiple connections to a single switch for load balancing and failover. Reusing this mechanism to allow multiple controllers control the same network is not viable, as OpenFlow has the implicit assumption that there is only one entity controlling an OpenFlow device and multiple (synchronised) controller instances connecting to the same switch will only have partial view of the network and installed flows will be mixed from both controllers.

This problem is aggravated by the fact that OpenFlow is not only used as a control protocol for switches as the southbound interface. Instead, its semantic has also left its mark on the design of northbound interfaces, which often mirror the OpenFlow semantics. In this section, we will analyse the problems of OpenFlow in composition and conflict detection and detail how and to what degree they can be avoided and solved.

5.5.1 Definitions

We will briefly recapitulate the definition of the packet types in OpenFlow important for composition:

Packet_IN event The `PACKET_IN` event is the main event in OpenFlow and usually signifies the arrival of a new flow. Whenever a packet arrives at a switch and is not handled by one of the FIB entries (or a FIB entry explicitly states to generate a `PACKET_IN`), a copy of the packet and the meta information of the packet (ingress port, etc.) are forwarded to the controller.

FLOW_MOD This is the OpenFlow command that is analogous to our FIB entry install command F_i.

PACKET_OUT The `PACKET_OUT` allows an OpenFlow controller to craft and send a packet to the network. A typical use case for this command is to reply to an ARP request. The `PACKET_OUT` consists of a packet and action list that is identical in function and syntax to the `FLOW_MOD` action list.

5.5.2 Multiple modules

The "multiple modules" approach without harmonisation (Section 5.3.2) is easy to support with OpenFlow. Adding a harmonising function (Section 5.3.3) is possible but requires to intercept `FLOW_MOD` commands before sending them to the network. Depending on the specific controller architecture, this is a more or less easy task. But if the flow mods get modified, the actual and the expected behaviour of the switch can differ. The switch will not report the flow as being installed but instead reports successful installation of a different flow. Also events coming back from the switch might need to be modified back to the state that each application expects. A statistics request of the switch will report statistics for actually installed flows. A harmonising function then needs to generate statistics for the flows that the application thinks it installed to match the behaviour an OpenFlow application expects. Correctly treating timeouts of FIB entries is also not a trivial task. Hence, even the first non-trivial composition approach, the harmonising function h, is not entirely straightforward to support and needs to depend on the intended level of compatibility with OpenFlow applications to emulate normal OpenFlow behaviour rather than extensive implementation.

5.5.3 The run to completion problem

In the previous sections, we defined the parallel composition to combine all commands triggered by the same network event. The definition of network events in OpenFlow is straightforward and consists of a small list of unsolicited messages of which the most important one is the `PACKET_IN` event.

Unfortunately, in OpenFlow there is no relationship between a network event and the responses of a controller and thus there is no reliable way to tie the responses obtained from a module to the original network events. `PACKET_OUT`s may reference the original `PACKET_IN` as optimisation to avoid copying the packet but this captures only a fraction of the `PACKET_OUT`s. Also, there is no way to tell if an OpenFlow module will respond to an event at all.

Hence, the basic assumption of composition – actions can be tied to events across multiple modules – is not guaranteed by OpenFlow. The following sections will detail how we can still be able to achieve composition with OpenFlow as the protocol.

There is another use case for unsolicited messages: situations where it is more advisable to generate an initial configuration beforehand to put network elements in a known state. This behaviour is known as *proactive applications*. This behaviour can be supported by either implementing a harmonising approach for these messages or treat the initial configuration of modules with a special "initial" event and treat all initial messages as being triggered from this special event. The composition logic can then handle this event just as any other event.

5.5.4 Parallel composition

If ignoring the (major) run-to-completion problem, implementing a resolve function works as sketched in Section 5.4.

Usually, an (OpenFlow) SDN application responds to a `PACKET_IN` with a `FLOW_MOD` and `PACKET_OUT`. The payload of `PACKET_OUT` is often the same Ethernet packet as in the `PACKET_IN` but with the action from the `PACKET_OUT` applied. For these `PACKET_OUT`s, a generalised solution is not possible since there is no standard approach to combine two arbitrary Ethernet packets into one. Once more, either a developer logic is needed and/or a simple approach that prefers packets from one module and drops packets of other modules if more than one `PACKET_OUT` is present. As a special case, if all `PACKET_OUT`s of all modules are indeed the result of applying the action lists of the `FLOW_MOD`s, we can take the same approach and use our composed `FLOW_MOD` and apply it to the packet of the `PACKET_IN` and use that as `PACKET_OUT` instead of any of the other `PACKET_OUT`s.

5.5.5 Serial composition

With OpenFlow we can at best try to achieve approximate serial composition – actual serial composition is impossible as an OpenFlow-oriented northbound interface cannot express both events and commands as input.

The input event in OpenFlow is the `PACKET_IN`. The goal is to create a `PACKET_IN` that carries as much information from the outputs (`PACKET_OUT` and `FLOW_OUT`) of the previous module as possible.

The generated packets and functions involved in an OpenFlow serial composition chain with two modules looks like this:

$$\begin{aligned}\texttt{PACKET\_IN}_0 &\rightarrow M_1 \rightarrow \texttt{PACKET\_OUT}_1, \texttt{FLOW\_MOD}_1\\ &\rightarrow \alpha \rightarrow \texttt{PACKET\_IN}_1\\ &\rightarrow M_2 \rightarrow \texttt{PACKET\_OUT}_2, \texttt{FLOW\_MOD}_2\end{aligned}$$

The meta-information part of the new $\texttt{PACKET\_IN}_1$ (produced by the network emulation function α, see Section 5.3.6) is a match that only carries the input port and no other information. The input port is the same as the input port of the $\texttt{PACKET\_IN}_0$ unless an overlay composition is used in which case the input port is the output port designated by the first module.

For the packet part of $\texttt{PACKET\_IN}_1$ we have two options: (1) Modify the original packet of the original $\texttt{PACKET\_IN}_0$ or (2) use the packet of the $\texttt{PACKET\_OUT}_0$.

Option (2) can fail and stop the serial composition if there is no `PACKET_OUT` from the first module. Likewise, option (1) will fail if no `FLOW_MOD` is generated. If stopping the serial composition should be avoided, an option can be to fall back to the other option.

For the second option, applying the actions from $\texttt{FLOW\_MOD}_1$ to the packet of $\texttt{PACKET\_OUT}_0$ should not be considered as it would never happen in a normal SDN environment. Typically, if a flowmod and packet out are present, the packet out is the first packet of the flow and all following packets are handled by the flowmod, thus normally we can assume that all actions are either already applied or are in the action set of the packet out. As consequence, we will ignore $\texttt{FLOW\_MOD}_1$ in this case. If we decide to use the packet of the $\texttt{PACKET\_IN}_0$, we can apply the actions of $\texttt{FLOW\_MOD}_1$ to it and thus ignore $\texttt{PACKET\_OUT}_1$. No matter what option we choose, we always ignore a significant part of M_1's output.

In both cases we have to apply the instructions of the `PACKET_OUT` or the `FLOW_MOD` to preserve as much information as possible. Only the subset of instructions that mutates the packet itself (setting a header field or adding a header like VLAN ID) can be preserved. Everything that is not directly related to the content of the packet cannot be represented in the new packet, which includes instructions like setting the output queue, rate limits, goto table x, etc.

The workaround to preserve the information contained in instruction is to merge/intersect all actions from all modules of a sequential composition as the last step. But this creates an unintuitive, difficult to understand and predict hybrid between serial and parallel composition.

In summary, all these problems with generating a new `PACKET_IN` make sequential composition in an OpenFlow only usable in very limited circumstances; only if the limited information that are carried over from the first to the penultimate module as input to the last module are sufficient and the last module reacts accordingly, serial composition with OpenFlow is sensible.

In a wider sense, we can conclude that *an OpenFlow-oriented northbound interface is ill suited to support serial composition of control modules.* Composition is possible but can only be achieved with serious effort and some shortcomings. The Chapter 6 details an approach how to deal with these problems.

5.6 Related work

The idea of module composition in SDN is not new and has been presented in various forms. These approaches have the goal of allowing multiple applications to run concurrently on the same network by applying all network rules and combine them into a single set of network rules.

The objective of these approaches is twofold: (i) to allow the coexistence and cooperation among heterogeneous control programs and (ii) to have mechanisms in place for possible conflicts and errors so that they can be detected and solved automatically. We will categorise the approaches here and describe their differences. We conclude our comparison with a table in Section 5.6.3 that summarises the differences and similarities.

The approaches generally employ one or more of the following functions and can be classified accordingly.

5.6.1 Classification of approaches

1. **Merging of network applications:** The most basic category of composition is the merging of the outputs from applications that need to

be deployed in the same slice of the network. This requires the definition of criteria and languages, e.g., to define which application has a higher priority. Many current SDN controller frameworks, such as Floodlight [33], OpenDaylight (ODL) [71] and Open Network Operating System (ONOS) [13], already provide static priorities for SDN apps to be deployed; however, some issues like dynamically changing the priorities, creating more complex behaviour (not based only on those priorities) or allowing compatibility of different SDN applications from different frameworks still remain unresolved. Most of these approaches fall into the "multiple module without composition" category (Section 5.3.2) and have a weak form of parallel composition, as some approaches collect all outputs and only apply the output with the highest priority.

2. **Network partitioning and slicing:** A category of approaches orthogonal to basic merging is to avoid conflicts in the first place by partitioning the network into multiple slices. Network administrators assign each slice to the different applications in the network. To achieve this goal, many SDN network hypervisors have already been implemented [14] and we describe the ones related to composition in the next section. The approaches in this category correspond to a harmonising function (Section 5.3.3) by modifying the applications' output to target only a slice of the network and therefore not conflicting with each other. This is helpful when applications should be restricted to certain parts of the network or parts of the network are assigned to different, non-cooperating entities, e.g., different test setups or customers. The approaches do not help in the case when multiple SDN applications should run on the same network. Therefore, for my thesis these approaches are not sufficiently powerful.

3. **Conflict detection and resolution:** The most complete concept involves detecting and resolving conflicts. Approaches implementing of this category either use a harmonising function that relies on a complex state kept by the approach or a complex parallel composition. The approaches that implement their own programming language also tend to implement serial composition.

5.6.2 Existing approaches

As already mentioned the first category of approaches does not really implement composition and mainly consists of SDN controllers, we will therefore concentrate on approaches of the other two categories.

Representing the second category, *FlowVisor* [111] can be considered as the first approach in the SDN domain to allow multiple network controllers to run side-by-side on top of the same network infrastructure. As one of the first, it implements a basic form of network partitioning: Instead of allowing all controllers to operate on the same flows, FlowVisor partitions the network into smaller slices and gives each controller only the view of its own slice of the network. To achieve this goal, it sits as a centralised module between the network and the SDN controllers. Many other hypervisors are based on FlowVisor and are documented in a comprehensive survey on SDN hypervisors [14]. A followup to FlowVisor is *OpenVirteX* [109], which introduces the concept of virtual topologies. Virtual topologies allow to present an SDN application a different view of the switches and connection than the physical topology. OpenVirtex will then transparently transform network commands between these two topologies. These two approaches do not cover the scenario where network controllers cooperate to control the same traffic, therefore they do not implement any merging or conflict resolution mechanisms. In our terminology, the slicing of the network is a harmonising function that avoids conflicts by making the matches of all modules distinct. As already mentioned in the classification of approaches, these approaches cannot do any composition of multiple modules on the same flows and are less powerful than the approaches of the third category.

The approaches that fit into the third category of "conflict detection and resolution" all have some notion of parallel and serial composition and almost all approaches for composition claim to support arbitrary combinations of these two basic operations.

The parallel composition of SDN applications was originally introduced by **Frenetic** [34], a high-level language for OpenFlow networks. Similar to Frenetic, **NetKAT** [7] is a network programming language based on the so-called Kleene algebra that defines union and sequential operators plus the Kleene star operator to iterate applications. Grounded on Frenetic, **Pyretic** [96] is a domain-specific language embedded in Python that enables network programmers to develop SDN applications by leveraging high-level abstractions. Pyretic enhances Frenetic by introducing (i) sequential composition, and (ii) topology abstraction, which allows programmers to limit each module's sphere of influence. Pyretic applications can be executed on top of a modified version of the POX runtime system [39]. The Pyretic interpreter communicates with POX through a socket-based API; it can potentially run on top of any controller platform.

The languages described in this paragraph (Freenetic, NetKAT, Pyretic) were carefully designed to allow parallel and serial (sequential) composition on individual statements, for example by choosing a function signature that has a policy definition as input as well as output. This approach avoids the challenges and incompatibilities faced when using OpenFlow as basis for composition but pays the price of mandating a new programming paradigm. It is not feasible to translate existing OpenFlow-based network applications into Freenetic/Pyretic-based applications. Hence, this approach also falls short of requirements needed in this thesis of working with existing applications (for the technical details see Chapter 6).

Based on OpenVirtex, **CoVisor** [56] also acts as a hypervisor and is placed between the network and multiple controllers. CoVisor speaks OpenFlow on both SBI (with the network) and NBI (with the guest controllers). The main goal of CoVisor is to allow applications written for different controller platforms and in different programming languages to cooperate on controlling the same network traffic. In order to achieve this goal, CoVisor defines operators to combine policies of applications running on multiple controllers to produce a single flow table for each physical switch. Moreover, CoVisor exposes a virtual view of the topology to each controller and to the applications running on top of it. These topologies can be very simple, like presenting a topology with just one switch to a firewall that provides a "big virtual switch" abstraction, or they can mirror the real network for routing applications. We described this concept as using an overlay for approximate serial composition in Section 5.3.7. In summary, CoVisor assembles the policies of individual applications, written for a virtual network, into a composed policy for the virtual network consisting of virtual switches. Then, it compiles the "virtual" policies into policies for the physical network. CoVisor's use of virtual switches to implement serial composition is the approach we outlined in Section 5.3.8. In the OpenFlow section (5.5) we outlined a few challenges that are OpenFlow specific; the CoVisor paper leaves out many details how these challenges, most importantly the run-to-completion problem, are solved by CoVisor. Moreover, the available implementation of CoVisor only implements a very limited subset of the approach described in the paper, namely a static composition that gets all network commands at the start of the program and thus avoids all the challenges with dynamic events/network commands. Even though CoVisor came out when OpenFlow 1.3 was well established, it only supports OpenFlow version 1.0. It is hence difficult to ascertain how CoVisor actually intends to address the really difficult problems in composition.

Another SDN hypervisor is **FlowBricks** [28]. It is a framework that integrates heterogeneous controllers using only the standardised controller-to-switch communication protocol. While CoVisor only works with OpenFlow 1.0, FlowBricks is designed to support up to OpenFlow 1.4 and currently supports all OpenFlow 1.1 datapath features; for instance FlowBricks can work with multiple flow tables. Similar to CoVisor, a policy definition configured in FlowBricks specifies how different services from controllers are applied to traffic on the datapath. FlowBricks runs on an emulated environment with heavy modifications on the OpenFlow switches and cannot be used with standard network hardware. As with CoVisor, the paper does not address the issues we described with OpenFlow and there is no implementation publicly available that could be examined how the implementation deals with the challenges.

Corybantic [74] supports composition of network applications by resolving conflicts over specific OpenFlow rules. Corybantic acts as a *module* orchestrator, where modules are applications that implement particular network functions, such as end-to-end Quality of Service or flow latency control, and their impact on the network is evaluated in terms of cost and benefits. The *Corybantic Coordinator* implements an iterative approach to evaluate if the proposed changes of a particular module should be allowed on the network. Each round of the iteration is divided in four phases: (i) modules propose changes in the network, (ii) each module evaluates its own proposals in terms of cost and benefits, (iii) the Coordinator picks the best proposal and (iv) the modules install the chosen proposal onto the network. Its main difference is that it does not allow the use of different languages and controller platforms, as the OpenFlow-based approaches do, and requires the specific implementation of the modules to be coordinated in the sense that the cost and benefits of the modules need to be aligned with each since otherwise the system risks to degrade to a priority based system. For example, if a low benefit value of one module is always higher than the high value of another module, the proposal of the first module always wins.

Like Corybantic, **Statesman** [117] composes network applications by resolving conflicts. Statesman defines three views of the network: *observed state*, *proposed state* and *target state*. To prevent conflicts, applications cannot change the state of the network directly. Instead, each application applies its own logic to the network's observed state to generate proposed states that may change one or more state variables. Statesman merges all proposed states into one target state. In the merging process, it examines all proposed states to resolve conflicts and ensures that the target state satisfies an extensible set of network-wide invariants (both user specific as well as inherent in the system). Statesman may reject a proposed state from one application based on such invariants. In this case, the application has to handle the rejection and propose a new state.

The goal of **Athens** [11] is to ease coordination and automatic management of resource conflicts between SDN and controller applications. It proposes a revision of the Corybantic design but is essentially a compromise between Corybantic and Statesman, presented above. Athens sends the current state of the network to each application module. As a reply, all modules synchronously send a set of proposed changes to the Athens coordinator. After that, the coordinator asks each module to evaluate all proposals (by using the same evaluation method proposed by Corybantic). Based on the evaluation feedback, Athens runs its conflict resolution algorithm to elect the winning proposal, which is eventually implemented onto the network.

Policy Graph Abstraction (PGA) [91] leverages graph–based and "one big switch" abstractions to detect and resolve policy conflicts. Users and SDN applications independently generate their policies as graphs and submit them to the graph composer through a PGA User Interface (UI). The composer automatically composes input graphs into a combined conflict-free graph, resolving or flagging conflicts/errors and reporting them to users, possibly with suggested fixes. An initial prototype of PGA leverages VeriFlow [59] to verify whether the policies in the composed graph are correctly realised on the network.

Canini et al. [16] introduce the notion of *transactional network updates* and provide a formal model describing the interaction between the data plane and a distributed SDN control plane. The authors formulate the problem of consistent composition of concurrent network policy updates, the Consistent Policy Composition (CPC) problem. The CPC abstraction accepts concurrent policy-update requests and produces a sequential composition of these policies. In short, the proposed model ensures that every packet is processed by only one global policy, either using the policy in place before an update, or the policy in place after the update completes, never a mixture of the two. In the sequential execution, conflicting updates are rejected entirely. Unlike other works presented in this section, the proposed approach is more centred on the theoretical complexity of the CPC problem than on the actual implementation/description of a software architecture.

Our approach **NetIDE** [103], which I will explain in more detail in Chapter 6, provides a runtime Network Engine that allows the composition of multiple network applications from different controllers. The semantics are similar to the ones defined in CoVisor, but NetIDE differs from it in the following aspects: (i) the connection to the network is performed via an SDN platform (e.g., ODL or ONOS) instead of leveraging OpenVirtex, (ii) it supports OpenFlow 1.0 and 1.3, and potentially other protocols such as NETCONF, and (iii) apart from merging applications, it handles and resolves possible conflicts between them. As an OpenFlow-based solution, NetIDE also has to deal with the problems mentioned for OpenFlow in Section 5.5. Chapter 6 will describe the NetIDE approach in more detail and how we handle these challenges.

Finally, the algorithm and approaches used in the composition strategies (Section 5.4) are similar in many aspects to the approaches used by SDN test tools such as SOFT [62] and NICE [17] to detect bugs in controllers. Composition strategies can reuse the work of these tools to enhance the semantic conflict detection.

5.6.3 Summary of the different composition approaches

In Table 5.1, we summarise and compare the composition approaches described in this chapter. The parameters that have taken into account are the following:

- **Application interface:** APIs used by SDN applications to communicate with the framework that implements composition and conflict resolution mechanisms.

- **Applications are modified:** For the approaches that reuse an existing API, i.e., OpenFlow or other standards, this column indicates whether preexisting application modules must be modified in order to meet the requirements of the composition framework/approach. This categorisation is not applicable to approaches that introduce new programming languages, as existing applications cannot be reused unless they are totally rewritten with the new language.

  Note that – at least for NetIDE – the “no modifications” check mark does not necessarily imply that any application will run unmodified. NetIDE places some restrictions on what SDN applications are allowed to do to handle the run-to-completion problem. As mentioned in the respective paragraphs, for the other approaches, this was difficult to determine unless explicitly mentioned in the paper as with FlowBricks.

- **Composition:**
  - **Slicing:** Network hypervisors, such as FlowVisor, OpenVirtex, force each application module to operate on a disjoint subset, or slice, of the traffic.
  - **Merging:** Multiple application modules can cooperate on processing the same traffic by merging their actions.
  - **Conflict:** It indicates whether the approach detects and resolves conflicts between individual policies generated by different application modules.
- **Composition specification:** This illustrates how the composition is specified. This ranges from fully automatic composition to a user-defined composition logic. In the case of approaches with custom APIs, parts of this task are also delegated to the modules themselves.

Approach	**Application interface**	**Applications are modified**	**Composition**			**Composition specification**
			Slicing	Merging	Conflict	
FlowVisor	OpenFlow	No	✓			User defined
OpenVirteX	OpenFlow	No	✓			User defined
CoVisor	OpenFlow	No		✓		User defined
FlowBricks	OpenFlow	Yes		✓		User defined
Frenetic	Programming language	-		✓		Union of all statements
NetKAT	Programming language	-	✓*	✓		Union of all statements
Pyretic	Programming language	-		✓		Union of all statements
Statesman	Custom API	-		✓	✓	Automatic, invariant checks
Corybantic	Custom API	-		✓	✓	Modules score proposals
Athens	Custom API	-		✓	✓	Modules score proposals
PGA	Custom policy language	-	✓	✓	✓	Automatic
NetIDE	OpenFlow	No		✓	✓	User defined

– Means *not applicable.*

∗ NetKAT supports slices in its programming language, but not multiple apps running at the same time.

Table 5.1: Comparison table of the different composition approaches

5.7 Conclusions

Composition can be a very useful tool if its current restrictions are understood and considered when using it. We have shown that for composition to work in a meaningful way, the underlying south-bound interface (SBI) should be designed to support it. While OpenFlow can be augmented and restricted to work for composition, as Section 5.5 points out, doing so results in a customised protocol that, even if the changes are not massive, is still lacking several aspects. The most notable aspect is that OpenFlow-based modules will get no feedback from composition about the conflict resolution result.

As main conclusions, we point out that an interface between controllers and control modules that is oriented towards OpenFlow *is not suitable to support any but the most trivial composition semantics.* The challenge is to find an interface that carries enough information (commands and events in our parlance; policies in Frenetic lingo) between modules, but hopefully without having to mandate its own programming style as done by Frenetic/Pyretic. Finding such an interface will make implementing the actual composition much easier as it does not need to add an extra layer of workaround like for OpenFlow.

6 NetIDE core

The previous chapters have looked at individual aspects of the problem of traffic engineering in data centres with software-defined networking (SDN) but have not brought the parts of the solution together. During the NetIDE EU FP7 project (NetIDE) [120], we developed a framework for running multiple SDN applications in parallel on the same network. NetIDE aimed at supporting SDN developers when developing applications through the help of better tools and IDEs as well as reusing existing SDN applications in a new environment. Central to the architecture is the NetIDE core as a central component implementing the composition and interfaces for tools and IDEs such as a network debugger or performance evaluator.

In this thesis, the core is used as a common platform to implement my approaches. It demonstrates that all individual aspects can indeed work together. This chapter is based on

- The sections addressing the NetIDE core in NetIDE deliverable D2.7 NetIDE FP7 Project. "D2.7 NetIDE Manual". In: *NetIDE reports* (2016)
- The NetIDE core implementation, which can be found at `https://github.com/fp7-netide/Engine/`.
- PA Aranda Gutiérrez, E Rojas, A Schwabe, C Stritzke, R Doriguzzi-Corin, A Leckey, G Petralia, A Marsico, K Phemius and S Tamurejo. "NetIDE: All-in-one framework for next generation, composed SDN applications". In: *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE. 2016, pp. 355–356
- The implementation of the core has been based on Tim Niklas Vinkemeier's master thesis:

  Tim Niklas Vinkemeier. "Composition and Orchestration of Network Control Applications". MA thesis. University of Paderborn, 2015.

6.1 Introduction

One of the goals of NetIDE is to allow multiple SDN applications to run alongside each other. Usually, SDN applications are written and tailored for a specific SDN controller like ONOS [83], Ryu [102], Floodlight [33] or OpenDayLight [82]. Even though SDN controller platforms have a lot in common, no standardised application programming interfaces (APIs) exist that applications can use. Porting an SDN application from one controller to another requires manual adaptation. In the NetIDE project, we investigated automatic translation of applications to be able to convert applications between controller platforms. That would have allowed to pick one of the controllers and run all applications on this controller. Unfortunately, this proofed to be an infeasible solution as even translation between the two imperative programming languages Java used by OpenDayLight and ONOS and Python used by Ryu. Automatic program translation between APIs with different semantics is an even more difficult, if not impossible task for arbitrary APIs. Even for the small set of three SDN controller platforms, the APIs differ enough and are not well enough documented to infer concise semantics that would be needed to create a machine-readable format, which is only the basic foundation for an automated translation.

Instead of trying to translate the application and run them on a foreign controller platform, NetIDE runs SDN applications in their original environment, i.e., their native SDN controller. When running the application in their native controllers, we are not directly interacting with the application anymore. To implement the composition, we will need to intercept the communication from the application at some layer to implement composition. The most basic one is to not modify the controller at all but instead provide an emulated OpenFlow to the controller (like CoVisor does, compare Section 5.6.2). This approach provides the least intrusion in a controller framework, but on the other hand also the least control and information about the application. Instead we opted to implement a module in each controller to implement this interception. The difference to normal execution is that the controller does not directly access the SDN switches but instead is amended to forward the SDN application requests and outputs to a central instance, the NetIDE core, bypassing as much as possible of the usual controller logic. In most controllers this amendment can be implemented as an add-on or plugin. The NetIDE core also provides the SDN controller with enough information and status messages to discover the topology and switches. In the NetIDE context, the add-on/plugin for an SDN controller that allows this integration is called *backend* and the SDN controllers running this backend are *client controllers*. To control the network, the NetIDE core uses a dedicated controller with a specialised interface. The component in the server controller that provides this interface to the core is called *shim* in NetIDE and the SDN controller that is running the shim is the *server controller*.

Figure 6.1 shows an overview of the NetIDE architecture and the place of the core in the architecture. The figure shows the SDN applications running on their own SDN client at the top and the SDN server controller with the shim module on the bottom. To allow the core to control/manage also the applications running on top of the server controller, the server controller in this example also includes a backend module. This way, by installing a backend and a shim in the server controller, a migration from a non NetIDE setup to a NetIDE setup can be made without moving the application from the server controller to a dedicated client controller or moving the switches to a dedicated server controller first. This of course is only possible if both shim and backend module for a controller exist and they can both be used at the same time.

The backends, shim and the core communicate via the NetIDE protocol, a custom protocol developed by the project. The protocol allows to tag all messages with an application ID to specify the application on the backend the message originated from. With this information, the core can treat the different SDN applications as individual applications – even if two or more are running on the same backend. Even then, the core cannot treat applications running on differ-

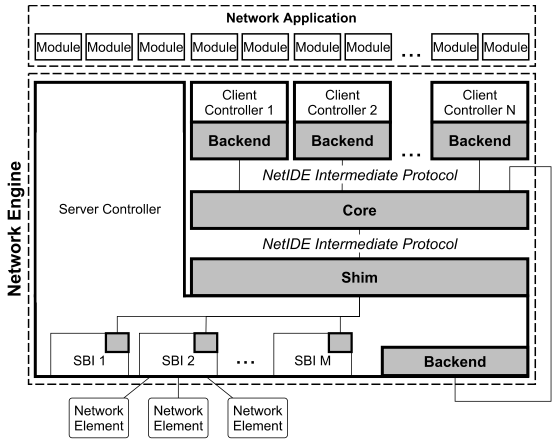


Figure 6.1: NetIDE architecture

ent controllers equally as it still needs to handle differences and idiosyncrasies of different controllers to provide a more uniform SDN application behaviour. For example, some controllers like ONOS actively query the flow table of the switches and delete all flows that were not installed by the controller. The independent nature of the core allows to implement conflict resolution and the GlobalFIB (Chapter 7) independent of any controller framework, improving reusability.

As the NetIDE protocol uses OpenFlow to pass network rules from the SDN application to the NetIDE core, the core needs to handle the problems associated with using OpenFlow as a north-bound protocol for composition highlighted in Section 5.5. The NetIDE protocol does so by adding meta information and restricting the OpenFlow protocol, which is described in detail in Section 6.4. The design decision for using a server controller and the role of the server controller is discussed in Section 6.3.

6.2 Challenges

During the development of the NetIDE core we faced and solved many challenges. The first challenge is to provide a stable platform for running SDN applications with composition while at the same time serving as an experimental research platform that could be extended and changed easily to try out/implement new techniques. And at the same time, the core needs also to support legacy applications with their APIs, while at the same time supporting newer applications and providing them with modern APIs.

In the rarest cases, software written during research and research projects is used without modifications in later projects. More often, only parts are reused in different projects or ported to other existing projects. Accepting this reality means that the reusability of modules of the software should be planned from the beginning. Also, rather than implementing from scratch, reusing existing components as much as possible should be attempted.

Another important aspect of software-defined networking is that software no longer is static but can be changed and modified in very rapid development and deployment cycles. Restarting the whole core and causing network downtime in these cases is often undesirable, so the core needs to have a mechanism to handle these cases. The need for this dynamic reconfiguration, dynamic reconfiguration in SDN in a more general way and the implementation of reconfiguration in the core are described in Chapter 8.

Composition of SDN applications is still in its early stages and Chapter 5 has already shown that composition has quite a few problems, especially when using OpenFlow, that an implementation needs to handle. Typical OpenFlow controllers and applications assume that they are the only entity that control an OpenFlow switch. The core needs to emulate this semantic as closly as possible to minimise the changes in other components.

The run-to-completion problem, mentioned in the Composition Chapter (Section 5.5.3), was solved by introducing and enforcing fence messages (Section 6.4).

Each OpenFlow request includes an identifier, called XID. OpenFlow controllers expect the responses to be answered with the same XID and only getting responses as a result of a request. Usually, the uniqueness of these XIDs is ensured by the controller, e.g. by just incrementing the XID for each request. With more than one controller, XID from the different controllers are not guaranteed to be unique anymore. Therefore, the core needs to guarantee uniqueness of the XID towards the switches; it also has to keep track of the request-to-backend mapping. This was implemented with the core routing module (Section 6.5).

6.3 NetIDE core relation to SDN controllers

The responsibilities of the core managing multiple SDN applications and acting as a supervisor for the SDN network are quite similar to the responsibilities of a normal SDN controller. Both have a concept of multiple applications that need to be managed and both need ways of controlling the SDN network by establishing and managing the connections to the SDN switches.

The core could talk directly to the switches, but we decided against this in order not to pointlessly reimplement these features. Implementation of the southbound interfaces, especially OpenFlow, has been done very well by existing SDN controller frameworks. To reuse these existing SDN controller's capabilities we identified two possible concepts. The first one, which NetIDE initially pursued, is to implement the different parts of the core directly as parts of the backend and shim. This led to a lot of code duplication for both the different backends and also the shim modules. The other possibility is to implement the core as an independent component that communicates with the SDN controllers using a well-defined protocol and handles the translation to the internal APIs of an SDN controller by a small, controller-specific module. This allows to implement the interface (the shim) for multiple SDN controllers with low effort compared to implementing the core inside multiple controllers. Another, even bigger advantage of keeping the core separate is that it becomes its own entity that allows to implement functions and interfaces for other tools to use.

The *shim* used above in NetIDE allows the core to forward/receive network commands from the switches without needing to implement the full OpenFlow specification. Managing the switch and handling all other OpenFlow message is still the task of the controller. From the perspective of the server controller, the shim is running an application that sends/receives OpenFlow messages to its switches.

6.4 Composition

The focus of NetIDE has been on a practically usable composition for *existing* SDN applications. Since existing applications almost exclusively use OpenFlow, the composition semantics realised have been focused on OpenFlow (version 1.0 – 1.4), too. For that, we had to overcome or work around the limitations of OpenFlow identified in Section 5.5 as much as possible. This has been mainly done by (1) a number of restrictions placed on the SDN application's behaviour and (2) protocol enhancements of OpenFlow using the NetIDE protocol.

The OpenFlow protocol allows an SDN controller to enforce a response when a switch has finished executing a command with the barrier message/response [116]; OpenFlow also defines asynchronous status messagess to inform the SDN controller when the state of the switch changes. Examples of these asynchronous status message are the packet-in message, port status changes, or flow mod timeouts.

Status updates in the other directions (from controller to switch) are, however, not designated in the protocol; a switch cannot determine if an SDN controller has finished processing a status message or if an SDN controller sends responses to asynchronous events. While this poses no problem in a normal SDN controller setup, for a composition the notifications and feedback in the other direction are needed as I pointed out in Section 5.5.3.

Determining if an unmodified OpenFlow SDN controller has finished processing an event is impossible as it is equivalent to the halting problem [68]. The following paragraphs explain how we how handle the different aspects of these OpenFlow related problems.

NetIDE protocol Modifying OpenFlow semantics creates a new protocol. Instead of using a protocol incompatible with OpenFlow, we created a custom protocol that wraps OpenFlow. That way, we can keep the OpenFlow messages and protocol unmodified and add our modification and additions strictly in our new protocol. This NetIDE protocol is used between the core and the backends. This protocol has its own messages as well as a message that encapsulates OpenFlow messages along with additional NetIDE-specific headers. It also allows to support different SDN protocols than OpenFlow later as it specifies the type of the encapsulated payload and can encapsulate another protocol than OpenFlow in its messages as long as the other protocol is also message based.

In this implementation, ZeroMQ (also spelt 0MQ) [1] is used as the transport protocol between the components (SDN controllers, core, extra tools). ZeroMQ has the advantage of already implementing transport and socket layer, freeing the components from implementing these tasks themselves and providing an easy to use interface to send messages between components.

Transaction IDs OpenFlow does not guarantee that an asynchronous message (from switch to controller) has a unique message ID that differs from other messages. Setting the ID to zero for all PACKET_IN messages is perfectly fine in OpenFlow. Tying a response to the request is difficult without having such unique ID. Even determining if two responses are reactions to the same request is not possible on the protocol level as the responses have no common ID. The NetIDE protocol header adds a unique transaction ID to every message. This allows the asynchronous message to be referenced in the response messages when a controller sends a response to this message.

Module IDs As the goal is to compose multiple applications, the core needs a way to identify the application responsible for a message. The backend adds a module ID to identify the sender of a message on every outgoing message to identify the application.

Fence messages To compose the network commands from all applications for an event (e.g. a PACKET_IN message), the composition needs to collect all responses to this event. The discussion in Section 5.5.3 has already shown that this is nearly impossible to achieve with unmodified OpenFlow applications. The SDN application or the SDN controller framework can determine if a request is finished, e.g. by checking if a handler returns. In NetIDE, the task is therefore delegated to the backend implementation. A backend signals the core that an event has finished processing by sending a fence message. The fence message ID repeats the transaction ID of the request message.

Using the transaction ID in the fence message, in the network command and in the request allows the core to group messages by the transaction ID. Multiple events can therefore be sent to a backend without waiting for a response, resulting in concurrent processing and avoiding being constrained by latency when waiting for a fence message before sending a new request.

Concentrate on parallel composition We did a case study in the NetIDE project and defined multiple use cases for NetIDE. All use cases were implemented with parallel composition. Since we did not require serial composition for the use cases and serial semantics are difficult to define and implement, the current implementation is focused on parallel composition but nevertheless realises a proof-of-concept serial composition.

Restrict OpenFlow control module behaviour One of the problems of parallel composition with OpenFlow is that handling a flow mod or packet out command from one module without results from other modules is problematic as the composition module needs results from all modules (compare Section 5.5.3). To work around this problem, NetIDE imposes restrictions on the modules' use of OpenFlow. A module must reply to events only and should not use proactive flow rule installations. We decide against working with pseudo events as this basically would have meant to implement a second composition that works like a harmonising function as explained in Section 5.3.4.

Modules also must not make assumptions on the state of the network, e.g., they need to always reply to a packet in and must not assume that previously installed flow rules already handle the flow.

For the serial composition we require always a packet out **and** a flow mod as output of all but the last module in the chain.

6.5 Modularity

This section is based on a section I wrote for: NetIDE FP7 Project. "D2.7 NetIDE Manual". In: *NetIDE reports* (2016).

When designing and implementing the core, we focused on modularity and reusability of its components, e.g., allowing the composition component to be reused in another project. As modularity is not a new concept, there are proven frameworks to support it. Our requirements in the core were that it should be Java-based and the framework should handle binding of the interfaces between the modules and manage module loading/unloading/upgrading. To be able to make quick adjustments or reconfigure the core with little effort, we wanted to include a command line interface (CLI). Furthermore, if possible, a framework that is already used in other contexts and software of our project would reduce effort to maintain multiple frameworks.

As result, we have chosen as framework for our modular implementation Apache Karaf [8]. Apache Karaf is an Open Services Gateway initiative (OSGi) runtime distribution for highly dynamic applications and supports runtime reconfiguration, service discovery and modularisation [8, 85]. We chose Karaf since it fulfilled all our requirements and is already heavily used by OpenDaylight (ODL) [82] and Open Network Operating System (ONOS) [83]. It also features an integrated CLI that is easy to extend. As an added bonus feature, it also allows the core to run on an existing ODL or ONOS Karaf instance.

As an OSGi-based application that adheres to the patterns of modular applications and service-oriented architectures, the core is built out of several bundles, each implementing a particular functionality of the core. An overview of the core bundles can be found in Figure 6.2. The bundles were chosen to get a high reusability and easy exchange against similar bundles as the bundles are almost self-contained. For example, putting the composition engine itself in its own bundle (core.coas) allows to reuse the composition engine in other contexts, e.g., in an SDN controller. Similarly, the communication is implemented in the core.connectivity bundle. If, for example, the communication to backends and shim should be done via HTTP (not a good idea, but as an example), a core.httpconnectivity bundle could replace it.

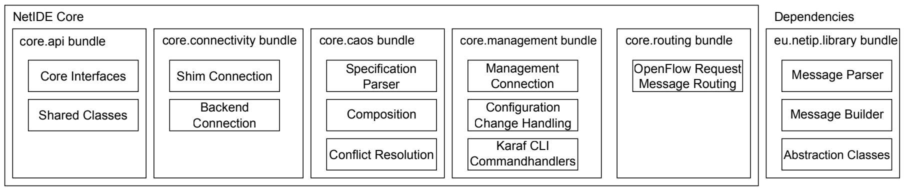


Figure 6.2: core Bundle Overview.

The individual bundles provide services to and consume services of other bundles. The service references are managed by the Apache Aries Blueprint Dependency Injection module [15]. It is integrated in Karaf and was therefore selected as the dependency injection framework for the prototype. A short overview of the bundles and their responsibilities follows:

core.api bundle The `core.api` bundle contains shared classes and interfaces for the core bundles. It does not contain any logic itself and does not provide or consume services. It is referenced as a dependency by the other bundles and provided at runtime in Karaf as part of the core.

netip.library bundle With the NetIDE Protocol also came the need for libraries that make handling the protocol's messages easy. This bundle contains a Java implementation of the NetIDE Protocol, i.e., it contains classes for parsing and creating NetIDE messages. It is implemented as a Java library so that other Java-based parts of NetIDE (e.g., shim module for OpenDayLight, shim and backend module for ONOS, and the backend module for Floodlight) can utilise it.

core.connectivity bundle This bundle contains all classes necessary for the connection to the shim and the backends. In our prototype, the connection to these are using ZeroMQ sockets. This bundle extensively uses the netip.library bundle to parse incoming messages and create outgoing messages.

core.caos bundle The `core.caos` (Composition and Orchestration) bundle contains the most important logic of the core, namely the composition and conflict resolution mechanisms.

core.routing bundle This bundle manages the mapping of requests and answers in order to give only the right backend/module the response to a request

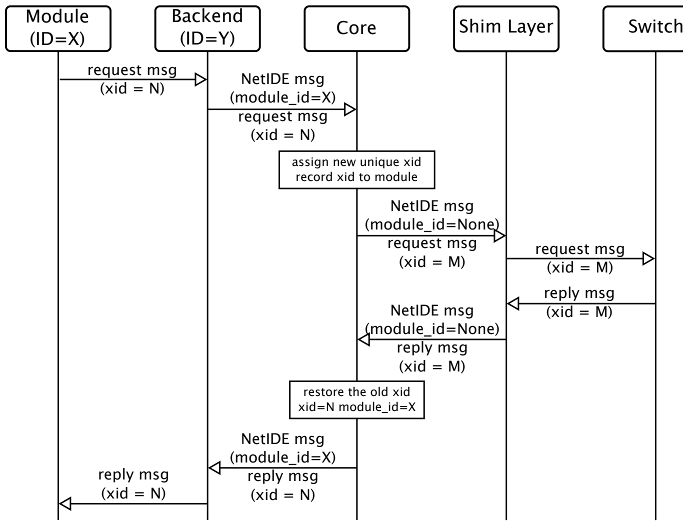

Module (ID=X)
Backend (ID=Y)
Core
Shim Layer
Switch
request msg (xid = N)
NetIDE msg (module_id=X) request msg (xid = N)
assign new unique xid record xid to module
NetIDE msg (module_id=None) request msg (xid = M)
request msg (xid = M)
reply msg (xid = M)
NetIDE msg (module_id=None) reply msg (xid = M)
restore the old xid xid=N module_id=X
NetIDE msg (module_id=X) reply msg (xid = N)
reply msg (xid = N)

Figure 6.3: Message diagram for XID remapping in the core

and not to broadcast the response to all backends, risking that a backend gets confused by a response that it did not request.

This bundle solves two problems. The first is that backends should only receive responses to requests that they initiated themselves and not responses originating from tools or other backends. The other problem is that request messages sent to the switch need to have unique IDs to pair responses to requests. Normally, this uniqueness is ensured by the SDN controller by generating unique XIDs, e.g. by just incrementing the XID by one for each request. Since the backends do not coordinate their XID, uniqueness needs to be established by the core. The core does so by replacing the original XID with its own XID, also recording the mapping to restore the original XID and direct answers only to the right backend. Figure 6.3 shows a message diagram of this process.

core.management bundle This bundle provides a management interface for the core using ZeroMQ and NetIDE Management messages. This interface allows external tools to change the configuration of the core at runtime (i.e. by installing a new composition specification). It also provides the special CLI command to examine the core status in the Karaf shell and modify its configuration.

core.globalfib The module core.globalfib implements the GlobalFib and the legacy implementation. For more details see Section 7.12.

6.6 Extending the core to non-OpenFlow protocols

Almost all modules of the core are written in a protocol-agnostic way. The only modules that know about and handle the content of NetIDE messages with OpenFlow payload are `core.caos` and `core.routing`. `core.routing` serves a very special purpose of mapping/remapping OpenFlow XIDs and is therefore OpenFlow-specific.

Most of the `core.caos` module is also protocol-agnostic. It will do sequential or parallel composition by instantiating the `ConflictResolvers` class, which in turn calls the specific conflict resolver depending on the message content. At the moment, this is only implemented for OpenFlow in the form of the `DefaultOFConflictResolver` class. Extending `caos` for other protocols simply needs implementing resolver classes for the other protocols.

6.7 Composition specification

The specification language has to be able to express all the semantics that specify how to compose software-defined networking (SDN) application modules running in the SDN controllers in the core.

We used an XML-based composition specification. That allows us to offload checking the validity of the file to an XML validation tool and enables us to use well established XML tools and libraries. The underlying XML schema is available as part of NetIDE [79].

The first part of the specification is to define what modules are available and how these are identified. Then the composition needs to specify when and how the modules are being used, e.g., if the modules are used in parallel or serial in the composition. Then finally the specification needs to specify how the results are composed and what behaviour the core should have if conflicts are detected.

An example for such a specification is given in Listing 3. The `Modules` element (line 3) fulfils the rule of defining the modules used in the composition. The `Composition` element (line 12) defines the execution flow on network events.

```
1  <CompositionSpecification
2         xmlns="http://netide.eu/schemas/compositionspecification/v1">
3   <Modules>
4     <Module id="fw" loaderIdentification="ryu-fw.py"/>
5     <Module id="appA" loaderIdentification="appA.py"/>
6     <Module id="appB" loaderIdentification="appB.py"/>
7     <Module id="lb" loaderIdentification="loadbalancer.jar">
8       <CallCondition events="packetIn" datapaths="0 42 43 45"/>
9     </Module>
10    <Module id="log" loaderIdentification="logger.py"/>
11  </Modules>
12  <Composition>
13    <ModuleCall module="fw" />
14    <ParallelCall resolutionPolicy="priority">
15      <ModuleCall module="appA" priority="1"/>
```

```xml
      <ModuleCall module="appB" priority="2"/>
    </ParallelCall>
    <Branch>
      <BranchCondition events="flowMod"/>
      <If>
        <ModuleCall module="log"/>
      </If>
      <Else>
        <ModuleCall module="lb"/>
      </Else>
    </Branch>
  </Composition>
</CompositionSpecification>
```

Listing 6.1: Composition Specification Example.

In the following section we will describe the most important elements of the composition XML format in detail.

6.7.1 Modules

The modules section specifies the modules to be used in the composition. Each module can have the following attributes: attributes,

id The ID of the module is used to identify the module throughout the rest of the configuration and also how the module identifies itself during the handshake with the core.

loaderidentification This optional attribute specifies a binary that implements the module. A loader application can start the not running modules by looking up this attribute to start the modules.

noFenceSupport As fence support may require modification of the modules, the core implements a workaround/hack for modules that do not implement support. Instead of using fence messages to determine when a module has finished processing and which events are tied together, in this mode the core assumes that everything sent after an event belongs to that event and that processing an event is finished when a new event is sent to a module. This mode is intended only for testing and when implementing fence support in a module/controller.

To restrict when a module is used, the module tag can also have a CallCondition tag. When a module is used in a composition but not being called but used due to the CallCondition tag, the module is instead treated as if it had an empty output.

Attributes to the CallCondition tag are:

datapaths If specified, this restricts the module to being called for only the switches (called data paths in OpenFlow) with the IDs in this list.

events Specifies a list of events for which the module is called. This list includes the events packetIn, which is for PACKET_IN events, connectionUp and connectionDown, which are the link status events.

headers The CallCondidition tag has also been prototyped to allow restriction to certain packet type, e.g., a specific TCP or UDP destination port. Although the specification allows specifying these restriction the current prototype does not evaluate them.

6.7.2 Composition

The composition allows to specify how to the composition of the modules specified in the Modules tag should be executed. If modules are listed in a *ParallelCall* tag, the modules are called at to the same time and a parallel composition is performed. If modules are listed without an enclosing *ParallelCall* tag, a serial composition is performed. The *Branch* tag has the same attributes as the *CallCondidition* as above for the Module and an If and Else sub tag. If the packet matches the attributes of Branch tag, the contents of the If tag branch is used for the composition of the packet, otherwise the Else tag.

A ModuleCall specifices that a specific module should be called. It has the following attributes:

module ID of the module, this references the ID of the Module tag in the Modules section.

priority Priority of the module, used in some of the conflict resolution policies, see below (Section 6.7.3).

6.7.3 Conflict resolution policies

Currently, the following conflict resolution policies for a parallel composition that is specified with ParallelCall are defined:

Ignore A baseline policy, simply ignores all potential conflicts and returns the union of the commands.

Pass If two commands conflict, discard them both. Also a baseline policy; currently not implemented (but trivial to do).

Priority Assuming that priorities have been assigned to the applications in the composition specification (even if only implicitly by the order in which they appear in the specification file), then pick the command resulting from the application with the higher priority (and ignore the other commands).

Auto The *auto* policy tries to determine a smart resolution. It is a simplified implementation of the conflict resolution outlined in Chapter 5.4. The policy determines the largest match that all commands (flow mods) have in common by using the following algorithm:

- each match field that has wildcards in all commands, keep the wildcard
- if there is a wildcard in one command and a specific value in the other, keep the specific value
- if there is the same specific value in all commands, keep that value
- Otherwise the algorithms determines the match to be empty and the conflict resolution fails.

Action resolution is currently implemented as trying to create a conflict free union of all action sets:

- Create the union of all action
- if the union has duplicate items, remove the duplicates
- if the union has two items with the same action but different parameter, (e.g. Add-Vlan 1, Add-Vlan 7), the conflict resolution fails.

If the conflict resolution fails for two conflicting commands C_1 and C_2, fall back to the default policy, which is usually reject/ignore the rules.

Other merge policies are easy to add to the core if needed, for example automatic rewriting of rules by priorities.

6.8 Evaluation

Evaluation of the individual aspects of the core can be found in the individual chapters; the evaluation of reconfiguration of the core is described in Section 8.6.

An evaluation of the core as the whole system configuration has been done in the course of the NetIDE project. The NetIDE project was a use-case driven project; one of the most important, if not the most important, goal for the software of the project was to implement the use cases specified at the beginning of the project. These use cases were designed by industrial partners and are modelled from realistic scenarios. The scenarios have in common that they require composition of existing applications in order to achieve the correct functionality. As an example, one of the scenarios consisted of a data centre with multiple networks that each have a switch or router connecting them. The networks are connected by firewalls and finally the whole data centre is connected to the Internet with a load-balancer. This scenario can be implemented with multiple SDN switches and an SDN controller with a firewall/switch or router SDN application on the respective controller. Using the composition implemented in the core, all modules can be run on the same switch and the scenario can be implemented by reusing the existing software modules. The other scenarios had similar requirements where multiple existing SDN software should be combined.

This evaluation of the approach yielded several results. The first and most important one is that the composition implemented here works with realistic scenarios. The second observation is that the modules have to be carefully selected (or modified) to comply with the restrictions laid out in Section 6.4, something that might be more difficult with third-party modules rather than modules developed in-house like in our scenario. Another result of the evaluation is that a controller independent interface has proven useful to develop debugging tools.

While a performance evaluation of the core would be critical for a productive system, the prototype developed during this thesis has been developed with a focus on functionality to explore if the concepts do work. We do not believe that a performance evaluation of the core does not bring any good insights.

6.9 Conclusion

This chapter has presented the NetIDE core and its surrounding architecture.

The core has proven to be a reliable tool in NetIDE. The challenges that present themselves for practical composition were overcome and I found solutions for them. At the same time, the implementation features a modular design that is extendable for further improvements. This practical implementation has indeed proven that it is feasible to run multiple applications of different controllers simultaneously on the same network. The concept is also general enough to be extended to protocols other than OpenFlow later. But the design and implementation of the composition has also shown that composition with OpenFlow needs special care of selecting or even modifying existing applications.

7 GlobalFIB

In this chapter, I will present GlobalFIB, an approach to bring a higher level of abstraction to composition and flow management to an SDN controller framework. This layer keeps information about all flows in a network-wide representation independent of information stored inside any particular framework. This representation allows better composition and enables easy integration of additional features such as verification and routing that includes application semantics – even across controller frameworks.

7.1 Introduction

Software-defined networks (SDN) are constantly evolving, and as the applications do, the software that runs and uses SDN evolves rapidly, too. In the beginning, SDN software was much simpler than it is today and we can still see this legacy in many ways. For instance, the programming model for SDN application is still very basic in some aspects; every application is allowed to send arbitrary rules to the network. This uncomplicated and rapid programming model is well suited for a single application and fast development. Using this programming model, which focuses on a single application; running only a single application is not sufficient for most networks anymore: a single monolithic application should not implement all required functionality in networks of non-trivial complexity.

Prior to software-defined networking (SDN), multiple parts of the network that had different roles were also kept physically separated. Data centres have a separate part of the network – the border routers that manage forwarding from and to the outside (the Internet). Examples of decisions made in that network are which egress path/provider to take or how to share the load between the uplinks. These border routers are then connected using dedicated physical links to the data centre network switches. To signal different treatment of flows between the two domains, in-band signalling is usually used, e.g. QoS parameters are set or special (MPLS or VLAN) tags are set and the configuration of both sides has agreed on what the tags mean, e.g., forwarding for best latency or data rate.

Even within the data centre network, there are multiple applications (or even dedicated middle boxes in non-SDN setups) that influence forwarding of packets: load balancers that translate a public IP address into an address of a backend server, QoS applications that classify or reclassify the flows' priority, or firewalls that block or allow flows. When migrating from a conventional network to an SDN setup, all these applications ideally would run on a single SDN controller platform and work together without any problem or manual configuration. Unfortunately without coordination, the individual components' flow rules and actions that are installed on the network can be contradictory and conflict with each other. This will render one or more of the applications to be inoperational. This situation is particularly problematic as current SDN controllers have no mechanism to detect such conflicts and they have barely any mechanism to avoid and solve such conflicts. Furthermore, as most applications are designed to run standalone, they provide more information and take more decisions than their core necessitates: a load balancer SDN application's purpose is to map an incoming flow to one of the backends. Since there is (usually) no API to communicate that, the load balancer applications (or its SDN controller) will also calculate a path for the flow.

As in this traditional SDN programming model each application generates – based on its decisions – flow rules for each individual switch. Applying these changes all individually on each switch leads to problems with conflicting rules and unexpected behaviour (see also Chapter 5).

To overcome the problems posed by running many uncoordinated network applications on the same network, we propose the following approach: Gather the information and changes that each applications wants to apply to any flow and use that abstract information to merge the flow descriptions coming from any application that concerns itself with that flow.

Controller frameworks are already going in this direction of providing these application programming interfaces (APIs) (see also Section 7.2). But these APIs are optional and SDN applications are free to choose whether to use the high level APIs or use the traditional low level APIs.

We believe that this is the right direction but the approaches are not going far enough. We also think that a cross controller framework is needed (as with the composition in Chapter 6). Therefore, we present an approach that does two things compared to traditional frameworks: Make the high-level flow description rule the central element in the description of flow rules and force these flow rules to be in a network wide format (unlike OpenFlow flow mods, which describe only flows on a single switch). We will show that having this format for flow rule definition has multiple advantages.

We propose GlobalFIB that stores information and meta data about end-to-end flows as a solution. The idea is to have a forwarding information base (FIB), which stores all information that is needed to take forwarding decisions on the scope of a whole network and all applications. This global FIB then provides all information to fill the individual FIB tables of the switches. And since GlobalFIB is a network-wide table, all flow rule descriptions in this table should also be in network-wide format, which we call end-to-end flow rule descriptions. This solution exploits the advantage that SDN controllers typically have a global view of the network, and use it as the key building block for our solution.

An end-to-end flow description is similar to a normal OpenFlow flow description (as used in an OpenFlow table) but with the important difference that it describes the whole way of a packet through the network. In contrast, OpenFlow flow descriptions focus on matching flows on a single switch.

Implementing GlobalFIB inside a controller framework is at first glance the natural way to implement it. The other option is to implement it externally as an additional layer between SDN controllers and SDN switches. As we have already implemented the core (Chapter 6), it is more reasonable for us to realise GlobalFIB also in the context of the core. As with the core and composition, this allows to provide this API with multiple different controllers running at the same time.

In the remainder of the chapter we will show how GlobalFIB is defined in Section 7.3 after looking at related approaches in Section 7.2. We continue by showing how GlobalFIB can be used to improve various areas of conflict detection (Section 7.4), validation (Section 7.5), conflict resolution and composition (Section 7.6) and as API for collecting statistics (Section 7.8), debugging tool (Section 7.10) and traffic engineering (TE) (Section 7.9). We present details about our proof-of-concept implementation of GlobalFIB in Section 7.12 and discuss aspects of integrating GlobalFIB into a distributed platform in Section 7.11 and finally draw a conclusion in the last section.

7.2 Related work

The idea to generalise individual flow rules into a more generalised concept is widespread. The concept of looking at individual flow mods and rules on a switch that is found in SDN is rather an artefact that originates from the low-level perspective of south-bound SDN protocols like OpenFlow and the history of distributed autonomous packet switches and their usage of distributed algorithms. In stark contrast, especially circuit-switched networks have always employed an end-to-end semantic as focusing on a single network element would break the semantic of a circuit.

In SDN, there are multiple ideas to provide high-level APIs, for example in the IETF IB-NEMO [51] project or the Neutron API of OpenStack [80]. The idea is to relieve the consumer of the API from the need to specify the fine details and only specify the rules with as few details as possible. Typical requests to these APIs are “VMs A, B and C should be in the same network” or “a tunnel from location D to E with at least 100 MBit/s”. Details of realisation (e.g. by using VLans or by creating a GRE tunnel) are not important for the consumer of the API.

The need to implement these more abstract descriptions of requests has also been realised by the various controller frameworks. Both ONOS [13] and OpenDayLight [82] implement higher level APIs. The goal of these APIs is mainly to relieve the applications running on the controller of the details that are needed to generate the flow rules. For this, the ONOS controller framework [13, 84] adds *intents* that allow controller applications to express intents like the HostToHost Intent, which instructs the controller to provide connectivity between two hosts. An SDN application does not need to specify a path between these hosts or other details. If the application wants to also specify the path of the flow, ONOS offers the PathIntent for that. The required parameters for a flow rule that an application (intentionally) does not specify are filled out by ONOS before generating flowRules (ONOS’ abstraction of Flow mods) and installing the flowRules onto the switches. This API only works inside the controller and the goal is to simplify the creation of flow mods. ONOS does not implement any conflict resolution or other optimisations based on these intents.

Running multiple apps on top of an SDN controller is also trying to be solved by multiple approaches I already compared in Section 5.6. The approaches either focus on single-switch semantics when implementing “legacy” protocols like OpenFlow, or only support their own protocol that is not switch-centric. An example for creating a language/interface that focuses on programming the network instead of programming individual devices is SNAP [9]. It abstracts the network into a “Big Switch abstraction”, thereby also creating only flows in its programming language that are from edge to edge port. To split the statements and its states onto the switches, SNAP uses Gurobi and solves an integer linear program (ILP). The drastic change of changing the programming language is only suited for new deployments. The new programs can then be written in SNAP.

Most of these approaches that introduce new programming languages (like SNAP) are designed to run solely on the network but introducing GlobalFIB as additional layer allows these approaches to coexist.

7.3 Architecture

In this section I start defining the network definition we are working with and continue with explaining the architecture of GlobalFIB.

7.3.1 Network

On a high level abstraction a network infrastructure can be seen as a collection of SDN switches, end hosts and links between these components.

For these switches and links we make a number of assumptions. The SDN controller is the only authoritative instance for any switch it controls and no other entity is manipulating the state of the switch. This allows the SDN controller to expect that the switch does exactly what the controller has instructed it to do. We assume that the network allows us to discover its topology, as GlobalFIB heavily depends on a correct topology. Furthermore, links between switches forward data unmodified or, if there is a transparent firewall/IPS, that these do not change the packets headers as we assume that packets on direct links between (SDN) switches are unmodified.

We divide switch ports (on controlled switches) into two categories: fabric ports and edge ports. We call a port a *fabric port* if it connects to another controlled switch, i.e. every packet received/sent from such a port is coming from/going to another fabric port. All other ports that connect to anything not managed by the controller are *edge ports*. For the purpose of this definition, virtual ports to the controller (e.g. PACKET_IN in OpenFlow) are also (virtual) edge ports. By differentiating between these port types, we can also differentiate between handling of packets that interact with other systems and packets that are handled purely internally. We call the ports, switches and links that are directly and solely controlled by the controller the fabric of the network. This fabric of the network is analogous to the fabric of a hardware switch,

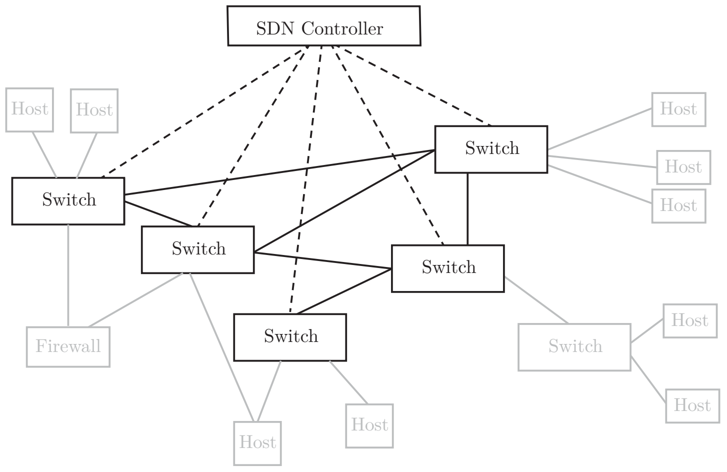


Figure 7.1: Overview of a network with distinction between fabric and edge links

which inner workings are also invisible to any outside system. Figure 7.1 gives an overview of a network. Every link and switch that is black is considered part of the fabric; the grey switches links are not part of the fabric and their ports are edge ports.

7.3.2 End-to-end flow rule definition

The key point of the GlobalFIB design is the description of *end-to-end flow rules*. An intuitive definition: it specifies everything that happens with a packet from entering until leaving the network. A packet forwarding is decided by exactly one end-to-end flow rule (or a composed end-to-end flow rule, see Section 7.6). In contrast, in a decentralised environment and often in SDN controllers, handling packets is specified on a per-switch basis and forwarding of a packet is determined by multiple flow rules, one per switch (or more with OpenFlow table support in Openflow 1.3+). An end-to-end flow rule is defined to have an input specification, that defines which packets belong to the flow, and an output port. Optionally, a flow rule can also have actions/mutations, constraints and meta data. We keep the specification as close to the OpenFlow flow mod definition as possible but extend it to carry the end-to-end semantic. The individual parts are defined as follows:

Input specification The input specification specifies the packets that are matched by this flow rule. The match consists of the edge input ports and an OpenFlow-like input match that specifies what header values of a packet are considered to be matched by this specification. The difference to the OpenFlow input specification is that inport of the match specifies a list of pairs (switch ID, port ID) instead of a list of only port IDs.

Priority a single application has multiple end-to-end flow definitions that overlap, e.g. by having overlapping input specifications, the priority will be used to resolve the conflict.

Output ports The edge port(s) that packets matching this rule should ultimately be forwarded to. As with the input ports this is a list of tuples in the form of (switch id, port id).

Actions/mutations Modification and actions that should be performed on the incoming packet and visible on the outgoing packet. These include changes in the packet header like VLAN tags, IP addresses or MAC addresses.

Constraints Constraints are optional and specify anything that changes the behaviour of forwarding the packet but is not (directly) visible on the outgoing packets. These constraints can specify a specific path in the network or specific QoS handling. Also more high-level constraints are possible. A high-level constraint would be the need to pass the packets of the flow through an IDS device.

- *path constraint* If the application want to use a specific path, this constraint will contain a list of the switch IDs that describe that path.

Meta data The meta data is information for this flow but does not influence packet processing directly. This can include the SDN application that installed the flow or an internal priority of the rule or the original rule(s) if the rule was converted from another format. This is implemented as a simple key-value list, with the representation of the value depending on the key.

So far we defined the following meta data:

- *importance* One of optional or required. By default flow rule definitions are considered to be required to be installed as they are needed for an application to work correctly. Optionally an application can specify that a flow rule can be ignored if it conflicts with another flow rule that is not optional. This is used in the conflict resolution (Section 7.6).

- *characteristics* The characteristics for the flow (data rate, burstiness, duration). This information can be used to allow better decision from TE (Section 7.9).

Note, it is possible to specify more than one port in the input specification and also more than one port in the output specification. As a packet that matches any of the input ports will be forwarded to all of the output ports, this feature allows specifying many-to-one or incast, multicast/broadcast and many-to-many (or all-to-all) flows.

An example for a rule of a connection to a load-balanced HTTP server is as follows:

input specification:	inport=7,dpid=5,dst_port=80, dst_ip=10.0.0.1
output port:	outport=3,dpid=2
actions:	set dst_ip=192.168.0.34
meta data:	app_id=http

7.3.3 GlobalFIB database

Storing the information about all the end-to-end flows creates the GlobalFIB database, which serves as the authoritative database of flows for the network. When no additional optimisation/verification is requested, these can be directly translated into FlowMods for the southbound layer:

We calculate a path (or spanning tree if multiple in/out ports are involved) from the input port(s) to the output port(s). On the input switches the FlowMods can be generated by simply copying the input specification and reducing the inport list to the ports of the switch itself. For the output port(s) the next switch(es) in the calculated path are used. We also tag the packet with a label (e.g. a MPLS label) that we link to the end-to-end flow rule. On all other switches (in between or egress) we set the match of the flow_mod to match this label and the outport to the next switch(es) in the path. Finally on the egress switch, the flow_mod rule also removes the label.

The example assumes, for simplicity, that we have unique labels that we can apply to the packets. If these are not available other header fields or a combination can be used to identify the packets or the packets have to be modified on the first switch to ensure this uniqueness and modified back on the egress switches.

The additional knowledge present in the GlobalFIB can be used by southbound frameworks to optimise the rules being pushed to the network: The southbound layer can use GlobalFIB as a database from which the southbound layer (or another consumer, like a debugging frontend as described in Section 7.10) can query for flows matching certain criteria and gets back all end-to-end flows matching the query. For example, instead of merely asking for the best possible match, the consumer can ask for any flow definition matching an input spec. The consumer could also ask only for rules that target specific flows. For example, it might be useful to ask for everything that affects TCP flows with port 80. Or when generating flows that are specific to a switch, a query of all flows that have a specific output port can help generating fewer rules as multiple end-to-end flows can potentially be grouped into a single rule on that switch. When a large number of flows are going to a central destination in the network (like the uplink or a central component like the DNS server), flows can be marked by a special tag on the ingress switches and then all handled the same with only one flow mod per switch in forwarding based on the tag. Another example is to ask if there are other rules that have similar or identical actions on the egress port. Such knowledge enables a south-bound layer to create better flow rules if multiple end-to-end flows only differ in the input specification but are otherwise handled identically. That distinction can be eliminated when generating the flow rules for the switches.

Having such a flexible database as the GlobalFIB and allowing other components to access it, begs the question how to access this database. The simplest option is to present all flows to the consumer. This option does not scale very well, so a mechanism to select a subset is needed. As the examples above already highlight, consumers might be interested in different subsets. As the number of parameters that we can choose from is very large, predefining different subsets does not make sense either. To satisfy these needs we think a simple query language does the best job. For the design of the query language, we wanted to stick to an established and known notation. Therefore, we designed it to be loosely based on the syntax of the flows rule definition, which is already based on the OpenFlow flow rule definition. To get an idea what we think such query language needs to support, we list a few examples of what we consider typical queries: “flows that match the packets from 1.2.3.4 to 5.6.7.8”, “all flows that start at the switch with DPID 23”, “any flows that matches at least one source IP address from 10.8.0.0/16” or “flows that affects a VLAN between 200 and 300”. In the few examples, the relation of the queried element and the expected result can have very different relationships: the answer can be coarser as in the example of matching a flow from one IP, it can be more specific as in VLAN query examples or even be a query for a non-empty intersection as for the flows affecting the 10.8.0.0/16 subnet. To support all these variations we define the following operators that specify the scope of the result:

$=$ Query and result must be equal.

$<$ The result should be equal to or coarser than the queried element, for example a query for $<$ 1.2.3.0 can result in a flow that matches 1.2.3.0/24.

$>$ The result should be equal to or more specific than the queried element, a query for $>$ 4.5.0.0/16 can result in a flow that matches 4.5.6.0/24.

$\emptyset$ The query and the result must have an empty intersection. The query $\emptyset$ 1.1.1.0/24 would return a flow that matches 3.0.0.0/16.

$\cap$ The result and the flow must have an non-empty intersection. A query for $\cap$ 192.168.*.1 would return a flow matching 192.168.23.0/24.

The language is quite simple so far but fits our needs. The operators presented here are a good foundation to make the language more flexible; in Section 9.2 we describe how to extend the language to be more flexible and SQL like.

Figure 7.2 shows how the GlobalFIB (and its database) is embedded into the SDN controller architecture. It sits, as an additional layer, between the southbound interface and the controller framework and application that generates rules. For the controller framework and applications that already specify rules in a GlobalFIB compatible flow format (e.g. the ONOS intents), no further adaptation is required; for all other applications, we add a legacy rule emulation, which is described in the next section. For existing controller framework this integration is similar to the backend modules for the core (see Chapter 6), which also eases the implementation of GlobalFib within the core.

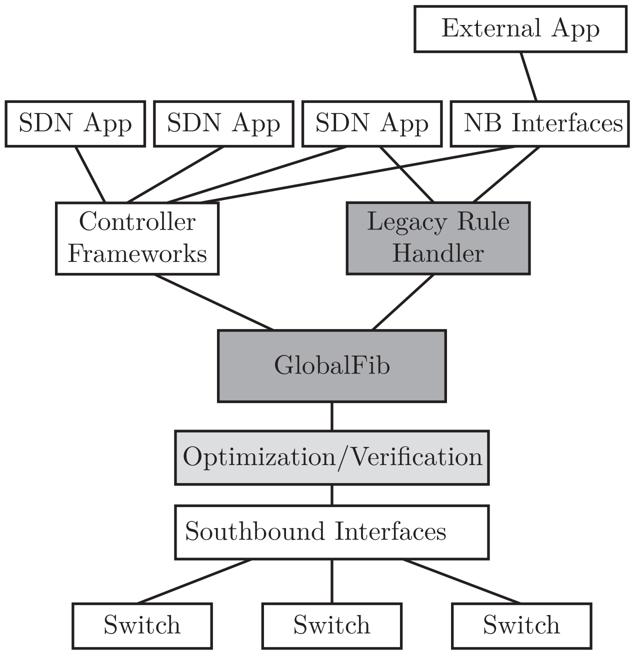


Figure 7.2: Architectural overview of GlobalFIB, new/modified components GlobalFIB components are dark/light grey

7.3.4 Handling legacy rules

Not all SDN software will generate rules that can be directly converted to the end-to-end flow rule representation; especially existing SDN controller apps will generate flow rules individually for each switch.

Flow rules that have edge ports as input and output port(s) already describe an end-to-end flow rule. All other flow rules, where input port, output port or both are fabric ports, are invalid as an end-to-end flow rule. The solution is to transform these flow rules into end-to-end flows.

We do this by emulating packet forwarding and inferring an end-to-end flow rule definition from that. For now, assume that a controller installs all flow rules we need for this emulation. We begin with a flow rule that matches packets on an edge port. Using that rule we emulate packet forwarding. We start with a virtual packet that comes from the edge port, apply the action from the flow rule, record the changes made to the packet and the designated output port. We also record the match used to match the packet as a tentative match for the end-to-end flow rule. Using the topology of the network, we first check the type of output port. If the output port is a fabric port, we look up the switch on the other end of the link, check which (legacy) flow rule on that switch matches our virtual packet. The match of the legacy flow rule might be only a subset of the match we recorded so far. In this case we set the (common) subset of both matches as new match and create copies of the virtual packet that match the rest of the original match. For these copies we will then repeat the process. After applying these changes to our virtual packet we repeat this step. For the other case, that the output port is edge port, we have successfully traced the virtual packet form an edge port to another edge port. From all the flow rules that matched our virtual packet we can infer an end-to-end flow rule in the following way: we already have the input port and the output port and the input match. To create the set of action and mutations, the action/mutations of the flow rules are combined into a single set of actions/mutations. Actions that cancel each other out or do not have an effect are removed from this set. Actions that cancel each other out are generally actions that reverse a previous action; for example, first adding and then removing a VLAN tag. This adding and then removing is not visible outside the switches that are controlled by the SDN controller. Actions that have no effect are actions whose effects on a packet are removed by a later action or do not modify the packet at all. For example a “decrement-TTL” action has no effect if later a “set-TTL” action is applied to the packet. Encapsulating a packet in an IP tunnel that discards the Ethernet, discards any modification to MAC field before the packet is encapsulated. Optionally, if we want end-to-end rules created from legacy to emulate as close as possible we also store the path of the switches as a path constraint, so the end-to-end flow definition is forced to take the same path as the legacy flow rules we emulate.

When the controller proactively installs all flows rules, end-to-end flow rules can be inferred by the way described before. But that raises the question what happens if no flow rule matches on a switch, when trying to look up the next step, e.g. because the controller does not install rules proactively. To get the rule for the switch in question, we need to trigger a rule installation from the SDN controller. This can usually be done by sending a `PACKET_IN` to the controller. The controller will then usually answer with a rule that matches the packet. To create a `PACKET_IN` packet, we have two options. The first option is to create an artificial packet. For that, we have to guess all packet headers and contents that we cannot infer from the rules, e.g. setting them to random values or zero. Creating such an artificial packet, which may not even be valid, that comes at an unexpected time might confuse the controller. The second option is to defer the creation of an end-to-end flow until a `PACKET_IN` from the network arrives (for a real packet). Again we emulate applying the flow rules to `PACKET_IN` until we arrive at the switch that has no matching flow. We then send the `PACKET_IN`, which has all the modifications from the flow rules so far applied, to the controller. As the second option is how a real network would behave and does not have the problem of random packets, we prefer this option over the first.

This emulation and transforming from legacy rules is naturally not always perfect, especially since we can install the end-to-end flow rule definition in a different way than the legacy application expects, statistics and network behaviour will also differ from what a legacy application expects. For example, if we choose different paths for the flows, the network utilisation will be different from what the application might expect. But this is a trade-off between emulating everything precisely and using the new possibilities that we have to make.

7.4 Conflict detection

As a reminder from Chapter 5, I recapitulate what constitutes a conflict of flow rules in an SDN network; a conflict generally occurs when packets should be handled in two different ways: two flow rules match on the same packets (or have a common subset) and specify different actions to be performed. Such conflicts can exist between flow rules of just one application or between flow rules of multiple applications. For a single application, these conflicts are often intentional since one of the rules is more specific than the other one (a canonical example is the default route and a local subnet route). Priorities are used to resolve the conflicts between these rules, either explicitly or implicitly like in longest-prefix matching. When comparing flow rules of different applications, only these effective rules that are not ignored have to be considered.

Ideally, all flow rule definitions should coexist and not conflict with each other. But applications are written by different developers and have different logic therefore conflicts are inevitable. A conflict might not create erroneous behaviour since the conflicting rules have similar behaviour. For example, one application normally forwards ARP requests/answers directly between connected devices and another application implements proxy ARP, which redirects ARP requests to the SDN controllers and generates ARP answers from the SDN controller. Irrespective of the winning rule, the ARP queries are answered; and even though ARP can still work without problems the SDN application proxy

ARP module might not work correctly since it is missing the information otherwise acquired through the ARP packets, leading to errors that are hard to debug.

Without using end-to-end flow rule definitions, flow rules will be generated on a per-switch basis. This can also create conflicts on a per-switch basis and will detect conflicts that are artefacts of an implementation that does not use end-to-end semantics. As an example, two applications have decided to use the same switch as part of a path to tunnel traffic. With end-to-end semantics this switch is only a hop in the path constraints and does not represent a conflict. With individual flow rule definitions, both applications can tunnelling with the same VLAN id to identify the tunnel. If these rules have different output ports, the path diverges, creating an unsolvable conflict. By having end-to-end flows for each application – either by directly having them in this format or being converted from legacy rules first – we can entirely avoid conflict detection on a per switch basis.

7.5 Validation/Verification

Having a central database for all flows in networks enables new ways of run-time validations. As we can assume that only flow rules exist in the network that are also in the GlobalFIB network, we can check for problems and irregularities as a validation step before installing rules to the network and reject rules if they violate our invariants. The same is true for other SDN deployments but the representation of all flows in end-to-end flow format gives a better foundation to do verification as this section will explain.

The first validation we always perform is whether an end-to-end flow rule is valid. That means checking whether the input specification and the output ports are all edge ports. This ensures that there is no flow that just ends in the middle of the of the network.

There are several problems that can arise in SDN. Most of them are inherited from the problems of traditional networks. But the nature of SDN also brings new problems.

A critical problem in any network are routing/forwarding loops. A forwarding loop is when a packet is forwarded in a circle. As the packets never leave the network, these forwarding loops will eventually consume all forwarding performance. This phenomenon is also often called “broadcast storm”, as broadcast packets are the packets most likely to trigger the loop behaviour as these are typically forwarded to all ports. By utilising the end-to-end semantics of GlobalFIB we do not allow any kind of rules that create a loop in the network fabric, as a loop in the network fabric would violate the requirement for end-to-end flows to never begin or end in a fabric port. This, of course, depends on the topology discovery layer to correctly discover the network topology (which is one of the main reasons why we require that the network topology can be discovered in Section 7.3.1). Applications directly producing end-to-end flow rule definitions have no way to specify loops. As the legacy rule emulation also produces end-to-end flows, the “loop free guarantee” carries over to legacy SDN applications. The legacy emulation will already detect a loop in the flows when trying to create an end-to-end flow. Even if an application introduces a loop as a path constraint (e.g. specifying a path constraint like 1,3,2,3,4) the SB layer still has the option to either remove the loop from the path, refuse to install the path or install flow rules in a way that the packets can be differentiated and forwarded twice over the same switch/link without creating a loop by using different tags for each pass over the switch.

The central authority of the GlobalFIB and its clear semantics allow to write live validations of the rules being installed. Instead of relying on a firewall to filter out the unwanted traffic, all rules can be checked against a set of predefined invariants. If a new flow rule in the GlobalFIB violates an invariant, the controller can refuse the new rule. These invariants can be seen analogous to unit tests in software development. An invariant can specify traffic between certain hosts has to be forwarded or not. Other possible invariants include having a certain IP range on an interface, for example disallowing any rule to forward a private IP to/from an uplink.

Instead of creating another completely new mechanism to specify and implement the invariants, we reuse the existing end-to-end flow rule definition and conflict detection. To specify an invariant the user (or an SDN app/framework) has to specify an end-to-end flow rule and an expected conflict resolution result. To specify an invariant that disallows traffic between certain hosts and a port basis: Specify a flow rule description that may never conflict from the ports the of the first network to the ports of the second network. This can be helpful to verify that traffic never leaves a certain zone, e.g. no direct flows between the DMZ and other parts of the networks without involving a firewall. The expected conflict detection result is set to having no intersection. For validation on network level the end-to-end flow can match on source and target IP addresses instead.

7.6 Conflict resolution and composition

Conflict detection is a powerful tool in itself but only detects problems and does not solve them. Conflict resolution is just as important for a dependable system. The most straightforward automatic way to deal with conflicts is to use priorities to resolve them and give each application a different priority and only apply the rule with the highest priority.

But this priority-based system ignores rules of an application with a lower priority if conflicts exist. Even if the priority of the ignored application is lower than the other applications, the ignored application’s flow rules can be important for the correct functionality of the application. The priority in this scenario merely ensures that the more important application will work in a case of conflict.

Using the end-to-end flows can have multiple benefits, conflicts that occur when using a per-switch conflict detection, may not even exist when the same

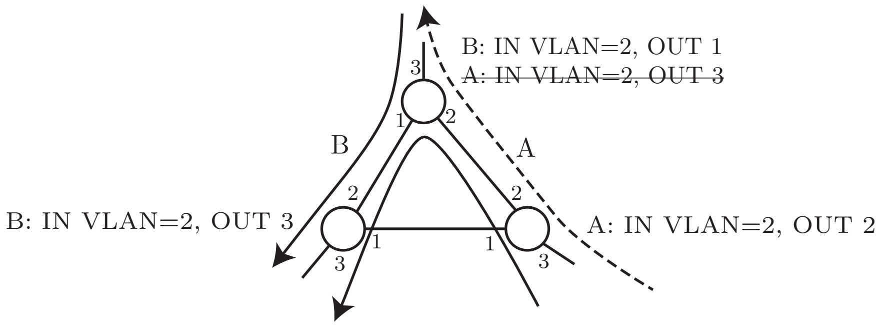


Figure 7.3: Applications A and B both try to install flow rules, application B has higher priority

flows are expressed as end-to-end flows. Two applications might decide to use the same VLAN tag to tunnel a flow. On an intermediate switch both would then add a rule that matches all packets with the VLAN tag, naturally creating a conflict.

A per-switch rule can alter the packet in a way, that the packet is not matched by any rule on the next switch anymore. Assume one rule changed the destination IP address of the packet and the other rule adds a VLAN tag; since the two actions do not conflict with each other, a composed rule can have both actions and apply both to an outgoing packet. The next switch might then not have a rule that matches the packets at all since the flow rules installed on the next switch are too specific to match the packets with two actions applied. Or worse, an unrelated flow rule matches and forwards the packets, potentially creating a loop. Figure 7.3 shows an example where a per-switch conflict resolution with simple priorities misdirects packets to a destination they were never intended to reach. Even with this simple priority based resolution system to resolve conflicts on individual switches, the resulting packet flow is completely unintended by all participating applications. With a real composition that actually merges the flows, limiting the scope to a switch is even more dangerous since the resulting packet flows can be even more confusing and very hard to debug. To overcome these problems a composition or conflict resolution has to consider not only the rules on each individual switch but also the rules on the neighbouring switches. With our end-to-end flow rule definition approach rule, considering all switches for a flow is intrinsically included in the approach.

A second advantage of using end-to-end flows is that the constraints of end-to-end flows are optional, so an application can leave out constraints it does not need to enforce, allowing easier merging of the rules. For the same end-to-end flow one application can have a path constraint and other one a QoS constraint. Since these two constraints do not contradict each other, merging the two flows is simple. When using per-switch flow rules, a path has to be always calculated and needs to be considered in the composition. Unlike the previous example, conflicts are expected here.

Having a higher level format also allows to add meta-information from the application to the composition. An application can indicate if certain flow is required or optional (e.g. a flow that is only for update checks), so conflicts with a firewall can be resolved automatically.

7.7 Calculating paths with GlobalFIB

Calculation of routing and forwarding paths in a network is nothing new. To avoid loops in networks with distributed control, usually all participants use the same path calculation, e.g. Dijkstra's algorithms in OSPF [76]. In an SDN network, usually the SDN controller is the central and sole instance for route calculation. Without the need to coordinate with other instances, it has more freedom in the routing decision. This freedom benefits a routing algorithm when using the information from GlobalFIB.

For the flows where the path is already predetermined through the application by specifying a path constraint entry, no route calculation is needed. For entries that specify the need of crossing some specific nodes or a virtual network function (VNF), the GlobalFIB allows to incorporate this information into the routing algorithm and to pick the best path through the network if multiple choices for the VNF are available.

We will show a solution that uses a shortest path algorithm to find a good path. We duplicate the network into multiple layers and assign a VNF to each layer. This layer is connected to the previous layer on nodes that have a VNF instance. This way, a path from the start node on the first layer to the end node on the last layer will traverse all required VNFs. In the example in Figure 7.4 a path from s to t should be found that traverses an instance of VNF A first and then an instance of VNF B. The three layers are (from top to bottom) "nothing traversed", "A traversed" and "A, B traversed". Since the layers are only connected at nodes that provide the functions, a valid path from s in the top layer to t in the bottom layer is guaranteed to fulfil the constraints. Further annotating the edges with weights allows to further control the selection of a possible path.

This approach to find a path depends on the fact that it is known that these VNF need to be crossed. GlobalFIB provides this information when it is included in the flow rule definition. If the information are available through another mechanism, the algorithm will work as well.

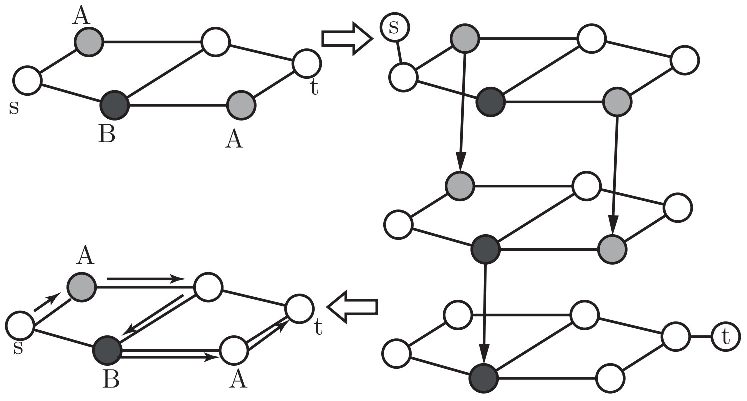


Figure 7.4: Transforming the network graph for a constraint that specifies "first A then B" and a possible resulting path from s to t

7.8 Statistics

Statistics are an important part of network management. The counter of flow rules on switches can give further insight into the traffic handled by the network. When finer distinction of flows is needed for statistics rather then for forwarding, flow rules are solely installed to gather statistics. Statistics-collecting rules usually take up space in the limited flow rule table on switches and can create a bottleneck on a busy switch. For a normal point-to-point flow the difference in number of packets on different switches on its path are negligible when the network is not overloaded. Using the end-to-end flow semantics helps here since it allows to pick a switch with a low load to collect the required statistics by allowing to infer which switches are eligible to collect the statistics by just looking at the hop list of a flow. This allows to install the rule that collects statistics on the switch with the most free space in its table on the path.

7.9 Traffic Engineering

Traffic engineering is one of the applications that benefits from more information. Since traffic engineering is also looking to steer traffic to paths with better performance characteristics for a flow, the GlobalFIB is a good fit as input for TE. Calculating demands and possible paths for flows works much better if all flows are available as end-to-end flows as this is similar to the circuit-based calculation approach that is predominant in TE systems. Since our end-to-end flows can be annotated with meta information, this allows applications to send expected traffic characteristics (data rate, burstiness, duration) and the required quality of service (QoS) class. Having this information in the GlobalFIB (and by that the TE layer) readily available instead of having to guess it or manually add it, allows for better automatisation of TE.

7.10 Debugging tool

Installing GlobalFIB as a completely new layer in the network architecture is a very radical step as it changes the behaviour of the network. Even if such a radical change of the network is not desired, GlobalFIB can still be used as useful tool in helping to debug the network.

When feeding all flow mods into GlobalFIB's database, GlobalFIB will be able to recover end-to-end flows if possible. These can be checked against the invariants, as specified in Section 7.5 to passively monitor the network's flow rules. Furthermore the database can be used to query if and how a certain packet would be handled by the rules currently installed in the network.

7.11 Distributed GlobalFIB

As SDN controllers provide a central instance they also provide a single point of failure and scalability tends to be a concern that needs to be addressed. This is usually solved by using distributed systems for the controller; multiple instances of the SDN controller share a common database but control different parts of the network, i.e. each controller instance controls a different switch and maintains the authoritative FIB database for that switch. Other instances will share a copy of that database [27].

Any new approach introduced into an SDN controller needs to support a distributed controller environment to be relevant for deployments that use/need this feature. The challenge when distributing a database with multiple reader and writers, is how to ensure the performance of the system as preferring certain reads/writes makes other operations slower.

In an SDN network, flow install happens mainly in two ways, reactive and proactive. For proactive flow install, the latency of the flow install is not critical as the flow rules are installed well before the flow arrives. For reactive flow install, the flow rules are installed when the first packet of a flow arrives. We reckon that therefore the database should be optimised to allow fast reactive flow installs.

The SDN controller instance that controls the switch where a new flow arrives should be able to make a fast decision and be able to install the flow mods after that and if needed instruct the other SDN controller instances to install flow mods on switches it does not control itself. To achieve that, a controller instance should be able to always have an up-to-date view of end-to-end flows that have the controlled switch as origin. This avoids communication with other controller instances for lookups. To ensure this property, every SDN controller instance serves as read/write master database for all flows that originate from one of its controlled switches. Additionally, each instance keeps a copy of the database of every other controller instance. This way each controller has a full copy of the GlobalFIB while for the latency critical reactive flow installation no communication with other controller instances is needed.

When installing the flow rules the SDN controller can have setup proactive flow mods on the switches that allow forwarding to a destination by setting the right header on the ingress switch (using MPLS or SDN approaches like [81, 107]). In this case the end-to-end flow approach allows the ingress instance to install the flow without coordination with the other instances.

7.12 GlobalFIB implementation

As a proof of concept, we implemented a centralised version of GlobalFIB in the Core (Chapter 6). As the Core is a central component, it is an ideal place to implement GlobalFIB and its legacy rule handling (Section 7.3.4). Flow installation and composition events are all routed through the GlobalFIB module in order to enable the GlobalFIB module to have a complete view of all flows that are installed in the network. The GlobalFIB module will keep a table of installed legacy flow rules. From this table, a second table that holds all calculated end-to-end flows is constructed.

In the current implementation the GlobalFIB is not used to calculate the routing of the network. The implementation is focused on showing that the legacy rule to end-to-end rule conversion is indeed possible and can be used to debug the network. For this, the GlobalFIB module provides two command line interface (CLI) commands. The first command lists the OpenFlow rules and the second one lists the calculated end-to-end flow rules.

7.13 Conclusion

This chapter shows that the introduction of GlobalFIB as a new layer in existing SDN frameworks and its focus on end-to-end semantics enables and eases the implementation of additional services in an SDN controller. GlobalFIB especially allows the integration of high-level features that have a wider scope than an individual application as seen with the integration of VNF locations into the routing process. In our eyes, it is clear from this work that this focus on end-to-end flow should be the general direction that controllers are heading to.

GlobalFIB is the attempt to bring this idea to existing controller frameworks with minimal intrusion and changes. However, these changes are still major and GlobalFIB will only achieve its full potential if all SDN applications use the enhanced end-to-end flow semantics instead of trying to recover these semantics from the "legacy" flows. However, if application are rewritten to use a new API, a more radical API change can be made that is better suited at supporting composition and running multiple applications cooperatively. Multiple of the proposed approaches in Section 5.6 are following this approach. GlobalFIB offers only an incremental improvement.

8 Reconfiguration

Software-Defined Networks (SDN) are constantly evolving and so is their software. One of the key advantages of SDN over traditional networks is the ability to rapidly develop and deploy new features. However, updating features often still requires restarting the SDN controller and causes network downtime. One reason for such updates could be some of the changes to the network proposed in this thesis; these major changes could take multiple iterations to implement.

In addition to such planned updates, unforeseen, accidental downtime is also a risk for SDN. While commercialised SDN controllers are adding mechanisms to deal with both planned and accidental downtime, they still are not competitive with conventional approaches, which typically use redundant hardware and special software to address these problems.

In this chapter I will investigate how these two challenges to SDN networks can be addressed with dynamic reconfiguration and show how the state of the network can by managed by reconfiguration. Finally, I present an evaluation based on the implementation of the NetIDE core presented in the previous chapter.

This chapter is based on

- Arne Schwabe, Elisa Rojas and Holger Karl. "Minimizing Downtimes: Using Dynamic Reconfiguration and State Management in SDN Networks". In: *3rd IEEE Conference on Network Softwarization (NetSoft 2017)*. Bologna, Italy, July 2017

8.1 Introduction

Conventional networks consist of dedicated hardware switches and proprietary software (firmware) running on these switches. The choice of software for these devices is limited; most times there is only one software image available. Even if multiple software versions are available, the choice is usually small and license-driven (like "basic IP services" and "advanced services"); changing licensing is a rare occurrence.

Upgrading to a new software version is usually the only time in the lifetime of a device when its software changes. For this special case, switch vendors have usually incorporated specialised routines in software and hardware that allow upgrades without losing connectivity, often called "in-service software upgrade" [20, 58]. This feature usually works by upgrading the software on a standby supervisor engine to a new one, then transferring all important state to the standby supervisor engine, and finally switching to said engine.

In this world of fixed components and monolithic software images, the need or even potential to change the running software on a short timescale does not exist. The user of a software cannot simply exchange one part of the software, be it for the addition of features or to solve a problem with the code of that part of the software. A reconfiguration/new composition of the running software is not foreseen.

In stark contrast to that, SDN controllers, rather than having a monolithic software image, are typically composed of multiple modules (often also called “apps”). Changing the composition of the modules – or also the modules themselves – is much more dynamic than in conventional scenarios. For example, since modules may come from different vendors, stability might be different and sometimes not ideal, requiring frequent tests and reconfigurations. The interactions within a specific module composition might not have been tested at all and has issues that only manifest in this specific scenario.

If that is the case, one option might be a controller restart, without a scheduled maintenance to change or reinitialise the running software. But in most environments, this is not an acceptable solution since it causes network downtime and intermediate failures.

Other components depend on the network to work correctly and this dependency is comparable to conventional networks. Other services of a data centre can be even more directly linked to an SDN than to a traditional network, e.g. the virtualisation network services using a network API of an SDN controller to configure its virtual network on demand instead of relying on a preconfigured set of VLANs. To provide a similar dependability of the network using an SDN application, we need to use paradigms that work in SDN, but also incorporate the strengths of SDN and leverage them to ensure the required dependability and availability.

Instead of treating the volatility of an SDN as a risk and a problem, we believe that this is a strength that can be capitalised on by introducing dynamic reconfiguration at run-time as a core component in an SDN deployment. Rather than having to stop/start the whole system, dynamic reconfiguration enables to only restart the modified parts of the system and can minimise the effects of restarts even further.

In this chapter, we consider existing solutions and approaches to dynamic reconfiguration. First, we collect requirements of a dynamic reconfiguration in the context of SDN (Section 8.2). Then we analyse related work (Section 8.3) and how these approaches address reconfiguration. We discuss how we designed our dynamic reconfiguration in Section 8.4 to fulfil these requirements by showing how malfunctioning modules can be detected (Section 8.4.4) and handled (Section 8.4.5). Having established the behaviour in the unexpected case, we use this as basis to develop dynamic reconfiguration behaviour for a scheduled or manual reconfiguration in Section 8.4.6. After that, we present our specification language for dynamic reconfiguration in Section 8.5 and how this approach integrates into the core (Chapter 6). Lastly, we evaluate our implementation in Section 8.6 and give a final conclusion in the last section.

8.2 Requirements for dynamic reconfiguration

We define the reconfiguration of an SDN network as the possibility to replace, reload, add or remove a module in the currently running controller configuration without restarting or affecting the remainder of the SDN network. For now, a module is a synonym for an SDN controller app, but we will show in the design section (Section 8.4) later that a reconfigurable module can also be used more generally and even correspond to an entire SDN controller.

As outlined in the introduction to this chapter, the reasons for dynamic reconfiguration can be placed into two categories: (1) planned reconfigurations and (2) unplanned reconfigurations. The planned reconfigurations also include moving the SDN controller to a different hardware or software platform, or, generally, changing the context: reconfiguring the services and resources based on changes in user needs, business goals, and/or environmental conditions. Supporting both categories is equally important and implies three key requirements.

While a reconfiguration takes place (e.g., even implying a controller restart), the controller might not be able to process requests, impeding network traffic. This is clearly undesirable. Hence, the **first requirement** is to minimise the time for reconfiguration itself in which no requests are answered. If a reconfiguration has an (unavoidable) disruption time, this time should also be minimised.

The reconfiguration itself needs to be triggered by an event. This is trivial for a planned reconfiguration. But for an unplanned configuration, it will be caused by detecting that the system is not working as intended: a part of the system is malfunctioning or has crashed. This leads to a **second requirement**: detection of malfunctioning parts or modules in the system.

The network should be disrupted as little as possible when a reconfiguration takes place; therefore, probably the most important aspect of dynamic reconfiguration is the transition from the old state to the new state. Therefore, the **third requirement** is to gracefully handle the state during the reconfiguration.

8.3 Related work

As stated in the introduction, SDN controllers help to break monolithic software into pieces called modules or “apps”. SDN controllers are usually written in a single language, and to synchronise their apps, they usually leverage dynamic frameworks already developed for the specific language they are based on. We will focus on existing modularity and reconfigurability of the existing controllers here.

The **OSGi** [85] technology facilitates the componentisation of software applications, for the specific case of the Java programming language. OSGi allows applications or components (so-called *bundles*) to be remotely installed, started, stopped, updated, and uninstalled without requiring a reboot, which is not feasible in standalone Java environments. For example, Apache Felix [32] implements the OSGi framework and service platform in a basic form. It is extended by **Apache Karaf** [8], providing some additional features on top of a standard OSGi implementation, such as folder-based hot deployment, remote SSH access to the console, or centralised logging. In the case of the Python language, iPOPO [52] is a Python-based Service-Oriented Component Model (SOCM) based on Pelix [86], which is an OSGi dynamic service platform written in Python. Finally, Apache Celix [18] is an implementation of the OSGi specification adapted to C. Both Pelix and Celix follow the OGSi API as closely as possible, with some unavoidable differences because the OSGi specification is written primarily for Java.

The benefit of these OSGi frameworks is that they provide the programmer with a framework that can be used to implement mechanisms that act upon on updating or detecting a crash of a component. Though not specific to reconfiguration, these frameworks can be leveraged to implement the reconfiguration we a striving for (see also Section 8.4). The reconfiguration of the modules has its limits as the implementations are limited to the (Java) process they are running in.

Currently, there are two SDN frameworks based on OSGi, specifically on the Karaf OSGi framework: OpenDayLight (ODL) [82] and Open Network Operating System (ONOS) [83]. **ODL** is based on a Model-Driven Service Abstraction Layer (MD-SAL) architecture, which lets different components – called *projects* – share information. The integration of these projects together constitutes ODL. The projects have different priorities (called offset $0, 1\dots$ in ODL) and projects with lower priority depend on the higher ones. However, ODL lacks a well-defined API to share common structures for dynamic reconfiguration of the network; e.g., if an ODL project needs to save information about flow entries in the network before being restarted, it is itself responsible for saving them. **ONOS** follows a well-defined architecture with specific APIs. For example, it is possible to obtain information about flow entries in the network and if a module is uninstalled, the flows related to it will disappear in the network as well. Although ONOS has some basic handling of flow entries when its configuration changes during run time, it is still very oriented towards its own modules and is Java-oriented. These frameworks still lack a way to detect malfunctioning modules and have no generalised API or configuration that allows fine tuning the behaviour.

Research on how to handle state during network updates has been carried out, albeit without focusing on reconfiguration of the whole system as in [97] or [66]. While we focus on the state handling between different apps, the ideas and concepts of this research can still be applied on top of our concepts for handling the state while reconfiguring the system.

8.4 Design

This section describes the design decision and the resulting design of our approach.

8.4.1 External or internal?

We identified two approaches of implementing the dynamic reconfiguration of modules in an SDN network; Figure 8.1 gives a simplified overview. The first approach is to keep the SDN controller as the central instance in the SDN stack and integrate reconfiguration mechanisms inside the controller platform. The

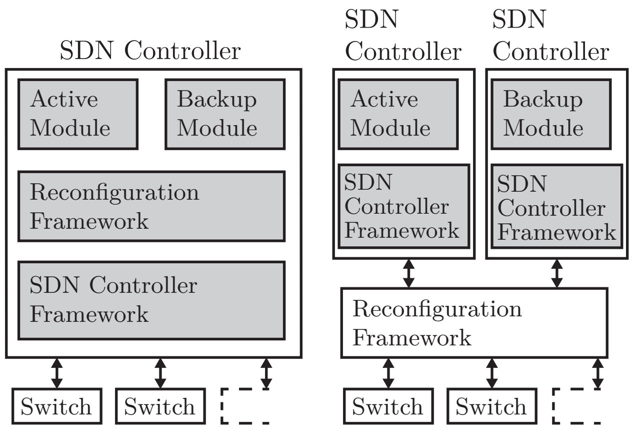


Figure 8.1: Simplified comparison of approaches to implement reconfiguration as SDN controller component (left) and as independent component (right)

other approach is to have a separate entity that acts as central component to handle the actual controller and to control the restarting and reconfiguration of the controller and its apps. Notice that these approaches are orthogonal to whether the SDN controller is implemented in a distributed manner or not.

The controller-based approach has the advantage that the controller's own mechanisms can be extended and the dynamic reconfiguration can be more tightly coupled with the controller's features. In practice, it will be difficult to maintain such an approach when the underlying controller framework keeps developing.

The approach with a separate entity controlling the reconfigurable SDN controller may make the integration with the SDN controller harder but offers more flexibility and possibilities. In particular:

- Different modules can be separated into individual processes, which in turn improves reliability.
- The reconfiguration process becomes controller framework-agnostic. This allows to extend reconfiguration even to cases where multiple controllers (from different frameworks) work in parallel under the control of this central component. In fact, the central component becomes a bit more powerful; it will not only deal with reconfiguration but also with the composition of multiple modules in general as demonstrated with our central component, which is the NetIDE core (see Chapter 6).
- Composition mechanism for multiple controllers (as demonstrated by CoVisor [56], OpenVirtex [110] or in Chapter 6) can be integrated.
- Radical reconfigurations are possible since the SDN controller itself is no longer an entity that must never be stopped.

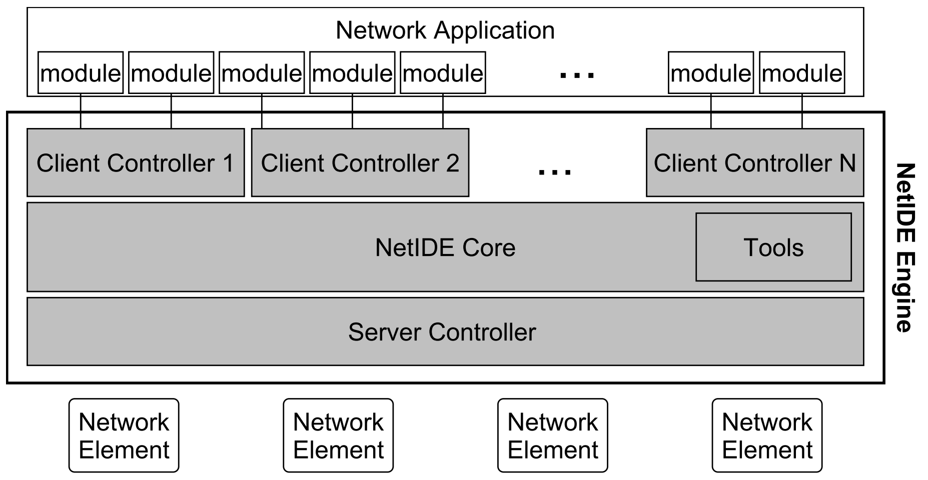


Figure 8.2: Architecture of NetIDE

There is ample evidence for the feasibility of such an external approach. For example, using a small external coordinator to improve the reliability of the whole system is also used in high availability systems in which multiple instances of a software, or even different software/hardware implementations, are used and the external component monitors the instances and selects a working instance to control the whole system (e.g., [38]).

In this chapter, we are hence looking in detail at the second approach since its greater flexibility facilitates more powerful reconfiguration. Figure 8.2 summarises this architecture (already presented in Chapter 6): the Core component deals with modules running in various controllers; the Server Controller does not implement application logic but simply deals with the network elements (e.g., parsing of OpenFlow messages). Both the NetIDE Core and Server Controller in Fig. 8.2 represent what is called the Reconfiguration Framework in Fig. 8.1, while the Client Controllers and their modules correspond to the actual SDN Controllers. Furthermore, modules can either be in active or backup mode.

We expect the line between these approaches will be blurred in the future by distributed SDN controllers that are a hybrid of the two approaches and incorporate a distributed version of the reconfiguration framework. That framework can treat the different instances individually and thus can benefit from most of the advantages we mentioned.

8.4.2 Granularity

As we base our design around the idea that the SDN network is controlled by multiple modules that run on one or more controllers, we have to take this idea into account for the central component's design as well. On the coarsest granularity, a module is a whole SDN controller. Enhancing the SDN controller to communicate more information about its loaded SDN applications and annotating the network commands with an identifier of the SDN application allows the central component to treat the individual applications of a controller as individual modules and allows reconfiguration to treat them as individual modules instead of a having to treat a whole controller as module.

8.4.3 Specification of (Re-)Configurations

Reconfiguration at run-time requires developers or deployers of an SDN system to be able to specify the configuration somehow, for example with some kind of specification language. First, this specification needs to be available at deployment time when an SDN controller along with is modules is brought online. Later on, the configuration might change during run-time; such changes then are reflected (if necessary) by a reconfiguration action.

Therefore, this specification language should support not only static configuration as such, but also the **re**configuration steps from one configuration to another. Ideally, the specification should only require reconfiguration specifications that cannot be calculated automatically from the differences between an old and a new configuration specification.

We provide more details in Section 8.5 about this specification language as extension of the specification language presented in Section 6.7.

8.4.4 Detecting malfunctioning modules

One of the events for reconfiguration is handling malfunctioning components. For handling these run-time problems, we have to go through three steps. Firstly, we have to detect the problem to trigger the reconfiguration. Secondly, we need to choose (from several possible configurations) one configuration that alleviates the problem. Thirdly, we perform a run-time reconfiguration to this new configuration.

Detecting a problem is in general a very hard problem (and strictly speaking, not even computable, like the halting problem). Instead of checking if a module ceased to work, we hence decided for a pragmatic approach and to check for the *liveliness* of the deployed modules. In this section, we focus on methods to determine the liveliness of modules; in Section 8.4.5 we will describe handling events describing the death of a module.

Malfunction of deployed modules can manifest itself in very different ways. The most classic and basic form of a malfunction is a *dead module* that crashes and ceases to respond to requests altogether. This can be detected using a simple heartbeat or timeout mechanism. If the module is using the OpenFlow protocol (e.g. a client controller connected to the core), the switch will periodically send `EchoReq` messages to the module. If the module is still alive, it will respond with an `EchoRes` message. Missing `EchoReq` responses from the module indicate a dead module. If the switch does not request `EchoReq` messages or the `EchoReq` message interval is too long for the required reaction time, the central component

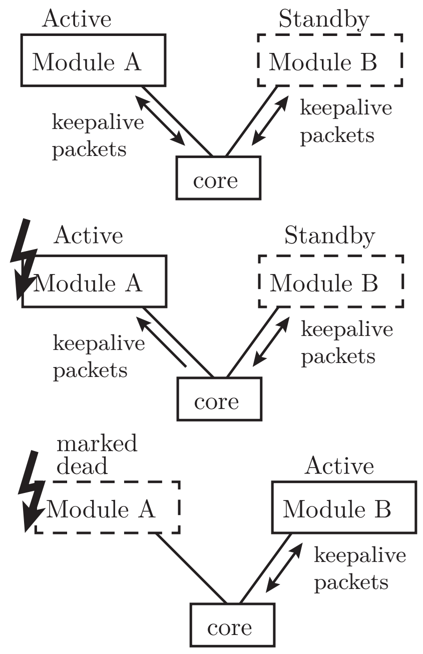


Figure 8.3: Switch-over between two modules after crash of primary module

will send additional `EchoReq` messages to the module. Other protocols have similar mechanisms or a separate mechanism can be implemented.

Apart from these implicit events of missing heartbeats, explicit events can also trigger a module to be marked as dead. The module can send an exit message or some failure message that indicates that the module is no longer available (in the sense of a fail-stop error model). This message can also be intentional, e.g. because the user explicitly stopped the module. Modules might also be restarted by external mechanisms. In this case, the arrival of a new module with the same name and configuration is another event that implicitly marks the old module as dead and to be replaced by the new module.

Figure 8.3 shows a simple example of an active module with a standby module. After the active module stops to send keepalive packets, it is marked as "dead" and the central component switches over to the standby module.

A concept similar to the *dead* module, but much harder to detect, is a *partially dead module* or *malfunctioning module*. These are modules that still respond to some events but not others or send invalid or erratic messages. As an example, think of an ARP handler that uses multiple threads. The main thread that also handles the `EchoReq` from the OpenFlow protocol is still alive and working and therefore the module is not marked as dead. To handle actual ARP requests, the module dispatches the request to a worker thread. If, for example, the worker is stuck in an infinite loop, the module will not answer any ARP requests, rendering it partially dead. To check liveliness in this scenario, the central component needs not only to check the general liveliness of the module but also the liveliness of a specific function. A method specific to the module to check its liveness is needed. This can be done by either passive observation (e.g. ARP requests to the module should be followed by ARP answers) or by active checks. An active check for the ARP function of the module is to send a test ARP request to the module and see if the central component obtains a valid answer.

Using external checks to monitor the network status and its components is already implemented in most data centres with the help of a specialised network monitoring software like Icinga [48]. Certain failure conditions in these systems, like a failing DHCP server test, can also be considered to mark a module as dead, e.g. the DHCP server SDN app. We hence foresee that a developer can provide, along with a module as such, also a (hopefully simpler and more robust) *failure detector function* to be executed by the central component.

8.4.5 Handling malfunctioning modules

To reach the goal of run-time reconfiguration with as little downtime as possible, malfunctioning modules also have to be handled as soon as possible. This requires an automatic solution that does not depend on manual intervention of an operator. But since not all failing modules that are the same, the dynamic reconfiguration process needs some hints (from module developer or SDN deployer) on how to handle the module. These hints can be provided offline, e.g., at deployment time, well before a possible malfunction.

For a malfunctioning module, there are a number of actions that can be taken. We categorise them into the following categories:

1. Repair the original module.
2. Replace the module with another (hopefully better working) module/configuration.
3. Control possible damage; do not restore full functionality but try to limit the impact of the malfunction as much as possible.

Repairing the original module is usually accomplished by restarting the affected module (and potential dependencies). Note that this state transition is different from the usual “restart the controller” scenario since the central instance still manages the state of the module while the module is restarting. Replacing the module is usually an option if an alternative module is available that can accomplish the same task but perhaps not as well as the primary module. An example for such a scenario is one very sophisticated forwarding module that selects the forwarding paths using advanced utilisation-based algorithms; the backup module is a simpler forwarding module that only selects a path randomly among the shortest paths (leading to inferior utilisation) but is much less likely to fail. Controlling possible damage is an option to contain a malfunctioning module without affecting the rest of the network. As a possible

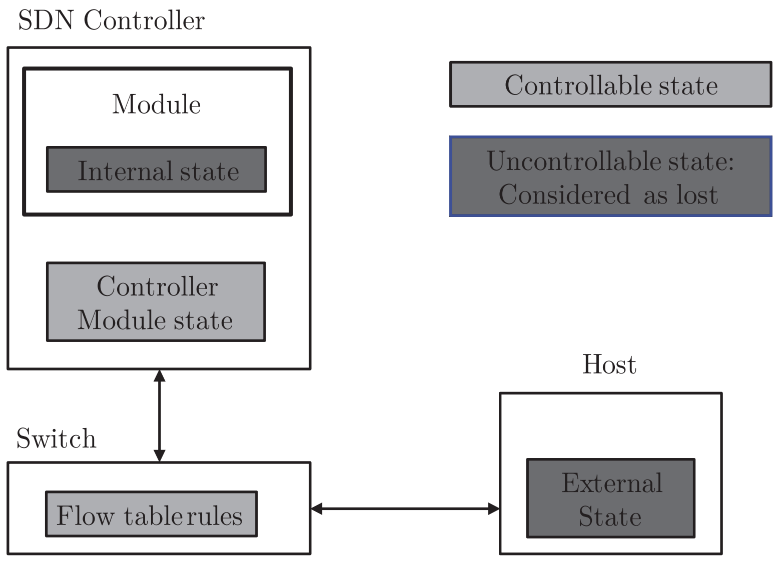


Figure 8.4: Overview of the different states that a module has in the system

scenario, the malfunctioning module is a monitoring module that affects the network stability but can be disabled/contained without causing problems.

For each of these options we have to treat the original module and its state. The state of a module can be broadly categorised into three different categories that can be seen in Figure 8.4. The categories are based on how we can manage the state:

1. *non-observable state*, this is mainly the internal state of a module
2. *the observable, controllable state* of the module, e.g. the state of the module in the SDN controller
3. *observable but not controllable state* of the module, e.g. through interaction with external systems

Let us consider an example to explain these states for a simple forwarding module. The non-observable, internal state of the module is everything that the module holds in its own memory, like the path decisions made, internal statistics or its own configuration. The observable and controllable state consists mainly, but is not limited to, the installed flow rules. Controllable state can also be state in other controller services. These installed flow rules can be tracked by the central component via the interaction of the module over its south-bound interface. In the reconfiguration event, this state can be changed and controlled, e.g. removing all the installed flow rules. The last category is the state in external systems. In this example, the forwarding module could have answered the queries for the default gateway IP address with a specific MAC address, which created an entry in the ARP table of the connected hosts, or in other words a state in an external system. The example of ARP shows that a non-controllable state is not a fixed definition. By sending more and additional ARP replies and requests the state of the ARP table in hosts can be controlled to some degree. In this grey area, non-controllable is defined by what is feasible to implement.

For a malfunctioning module, its internal state has to be considered as invalid or lost. If some of this internal state is important the module needs to have its own mechanism to checkpoint and restore this state. Most modules will already have most of this implemented as restarting a module without dynamic reconfiguration also requires reading in configuration values and other persistent data stores.

For the controllable state we have two basic options: keep the state or remove the state. For a more advanced scenario, a combination is also conceivable, e.g. remove or keep only flow mods that match a certain pattern. There is no general rule to decide whether keeping or removing the state is preferable. The best option depends on the function of a particular module. For example, when replacing one forwarding module with another, it is better to keep the old state and let the rules be slowly replaced with new rules whenever a new flow arrives that triggers a new rule install. On the other hand, a firewall requires a consistent set of rules and mixing flow rules from different firewall applications might create unforeseeable problems, which means that we should provide a clean state on the switch-over. Hence, we again foresee an option for the module developer to specify the behaviour desired from the central component. The right action is not only dependent on the modules but also the way they are used in a composition, so we leave this configurable.

In some instances (e.g. the mentioned forwarding module) it might be the best solution to keep the existing forwarding paths active to minimise network disruption. The failing module might not even be under the control of the central component and the central component can only wait for the module to (hopefully) recover.

8.4.6 Adding/removing modules

When adding or removing a module, the behaviour is a little bit different from handling an exception. For a planned dynamic reconfiguration the user of the SDN network will supply a new configuration that supersedes the old configuration.

To accept the new configuration, the run-time system has to make sure that the new configuration is a valid configuration and that switching to the new configuration does not cause additional downtime. We verify that new configuration, to only allow valid configuratios to be installed. Checking and activating the new configuration has to go through the following steps:

- Checking the syntax of the new configuration.

- Checking if all (required) modules for the new configuration are present or, if necessary, wait for the new modules to connect.
- For removed modules the state of this module has to be handled. Handling this is almost identical to the exception case. But since a module is usually removed intentionally, the default, unless specified otherwise, is to remove as much of its state as possible. An example for keeping the old flows would be the forwarding module example already mentioned above.
- Replace the old configuration with the new configuration with handling the state like specified in the configuration.

8.5 Specification language

The specification language needs to express the configuration of modules, how they should be composed into meaningful behaviour, which error checking should be applied, what default rules should be used or which custom behaviour (e.g., failure detector) should be used instead, and how the actual **re**configuration from one scenario to another should take place. We extend here an XML-based specification language that is presented in Section 6.7. proposed earlier [45] with composition of modules into complex network applications in mind.

For the reconfiguration, the **Modules** tag has been extended to provide options to configure reconfiguration. Listing 8.1 provides an example:

```xml
<CompositionSpecification
       xmlns="http://netide.eu/schemas/compositionspecification/v1">
       <Modules>
              <Module id="firewall">
                     <liveliness type="ofPing"/>
                     <recovery type="restart" />
              </Module>
              <Module id="fwd">
                     <liveliness type="timeout">3000</liveliness>
                     <liveliness type="plugin">eu.netide.arpchecker
                            </liveliness>
                     <recovery type="replace">slowfwd</recovery>
              </Module>
              <Module id="slowfwd">
                     <recovery type="ignore" />
              </Module>
       </Modules>
       <ExecutionPolicy>
              <ModuleCall id="firewall" dpid="2 7 9"/>
              <ModuleCall id="fwd" dpid="1 3 4">
       </ExecutionPolicy>
</CompositionSpecification>
```

Listing 8.1: (Condensed) Specification Example

8.5.1 Modules

The first block, the **Modules** block specifies the modules themselves along with their behaviour vis-à-vis reconfiguration. Here, the following module attributes are relevant:

`id`: Unique identifier for each module.

`liveliness`: One or several tests that check if the module is considered as still alive. A few simple ones are available as internal checks, e.g. OpenFlow ping or failure to respond to packet_ins in a given time (`timeout`). For more sophisticated checks we allow to specify external classes that implement the check.

If more than one test is specified, a module is considered to be alive only if all tests succeed.

As we heavily use Apache Karaf, we specify the external function as bundle name, so that Open Services Gateway initiative (OSGi) can automatically find the checker and load it with its dependencies. The module has complete freedom how to implement the detection. The module gets a reference to the Core to report back the status of the checked module.

`recovery`: This element specifies the action that is performed when the liveliness tests fail. Available options are to restart the module, to ignore the module henceforth (do not accept any input from the module and also do not forward information to the module), or to replace it, in which case the module is ignored and instead another module whose identifier is specified here is used in its place. "Used in its place" means, in effect, that the replacement's module ID is used instead of the replaced module's ID in the ExecutionPolicy.

As a last option we also provide the possibility of having a custom class handle the recovery, similar to the liveliness check.

`state`: This basically tells the central component how to handle the controllable state of the app. So far we specified `remove|keep|keepUntilRecover`. The first two are self explanatory and the third is a bit of a hybrid, in that it keeps the rules until the module is replaced or recovered.

Again, a bespoke state handling by a developer-provided class is easy to conceive and integrate (but not yet realised in the proof-of-concept implementation).

8.6 Evaluation

Doing a quantitative evaluation of such a reconfiguration approach is difficult and even if done has not much significance since metrics like the number of packets lost, time to detect a malfunction etc., are all dependent on the timeouts and other configuration parameters that have been chosen and the result will just reflect the configured values if the implementation is working correctly. Instead we opted to evaluate if the reconfiguration behaves like expected to the specified parameters. That means we are checking if the reconfiguration time and failure detection matches with the configured settings.

To evaluate reconfiguration, we used simple learning switches implemented in Ryu [102]. For a planned reconfiguration, we start a second instance of the Ryu controller with a second learning switch and loaded a new configuration that directs all traffic to the new switch. As expected, the new configuration was applied instantly with no measurable delay or packet loss.

To test an unplanned reconfiguration scenario, we configured two separate Ryu controller instances, each running a learning switch. The configuration file specified the second switch as a backup for the first. As malfunction detection we used a timeout of 10000 ms for any response of the failing switch, as the interval we are monitoring (heartbeat of OpenFlow) is 5 s. To simulate the non-responding controller, we used the Unix SIGSTOP signal to pause the first Ryu process after a random time rendering it unable to respond anymore. Once that has been detected the (queued) answers of the backup controller were sent to the network.

The detection time naturally depends on the frequency of packets. The smaller the packets' inter-arrival time, the quicker a non-response is detected. The normal OpenFlow heartbeat was 5 s in our test setup. A simple application we used the standard ping command and used its reported latency. The reported latency of ping with an interval of 0.2 s shown in Figure 8.5. Since the ping interval is much shorter than the heartbeat interval, most timeouts are triggered by ping itself and most responses are therefore in the 10 s timeout as well. The main peak being at 9.8 s rather than 10 s but this is rather expected as 10 s should be the maximum time an application should be seeing. And the timeout is 10 s, so a ping packet that is sent 200 ms after the last reaction from the controller is expected to be waiting 9.8 s before the central component marks the module as dead as this time (200 ms + 9.8 s wait time) is the timeout period (10 s).

In contrast to these results, when using a 20 s interval test as shown in Figure 8.6, the timeouts occurring from the heartbeat interval (5 s) are more significant. This makes a distribution from 5 s to 15 s of reported latency visible in the plot again with timeouts triggered by the ping packets themselves visible at 10 s. In addition, the larger interval allows ARP timeouts to happen. So before sending a ping packet often an ARP request is sent to the network. These extra packets can trigger a timeout waiting for a response from the controller. As the ping utility only records the time between sending a request until the answer, the additional time the ping utility waits before it actually sends its packet is not included in ping's latency output and therefore these timeouts are measured as 5 s instead of the 10 s, leading to a large spike at 5 s instead of 10 s.

The results have some minor irregularities, which also are expected in an emulated experiment setup like MiniNet, but otherwise confirm what is expected for a timeout based system, which shows that our approach is working.

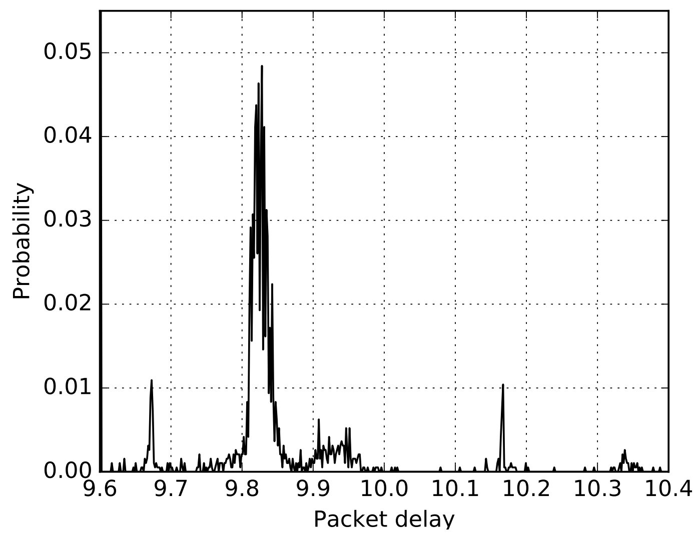


Figure 8.5: Recovery time as seen from the ping network application with 0.2 s interval

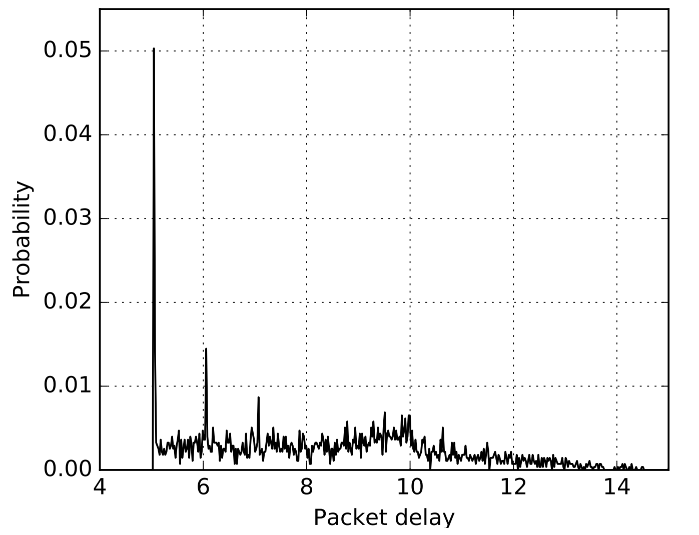


Figure 8.6: Recovery time as seen from the ping network application with 20 s interval

8.7 Conclusion

The chapter has shown that the proposed reconfiguration methods and the implementation of the proposed mechanisms are an essential step to making SDN controllers and networks more resilient and reduce the planned and unplanned downtimes in SDN networks.

Almost all of the concepts in this chapter do not require a specific SDN protocol like OpenFlow. The existing implementation could be extended to cover other non-OpenFlow SDN protocols without conceptual difficulties.

The ability to quickly change the modules of a running SDN network allows quicker changes to the software running of the network. This way it minimises the risks of introducing new techniques, like the ones introduced in this thesis. Reconfiguration is possible in both directions and a non working approach/implementation can always be replaced with an earlier version or a more simple/robust approach, again minimising the risks when introducing new features into the network.

9 Conclusion and future research

This chapter summarises the most important conclusions and reflects on the questions that were given at the beginning of the thesis. It will also look into possible further research and open questions.

9.1 Conclusion

In this thesis I have looked into many challenges that exist with the current SDN technology and its current implementations. For these challenges I presented solutions. I investigated challenges from low-level problems, as the lack of hardware features and forwarding table space in Chapter 3 all the way up to software design and architecture in Chapter 6 and Chapter 8. All these problems are related in one way or other to the challenges of better traffic engineering in data centres.

I have shown in Chapter 3 that software-defined networking (SDN) allows us to use the resources of existing network equipment more flexible and better. It also enables using powerful techniques as selecting a forwarding path selection for individual flows that would not be possible without the use of SDN. This path selection allows bringing the advanced traffic engineering even to small deployments. Furthermore, to give a better understanding of the problem, the chapter includes a complexity analysis of problem and shows its $\mathcal{NP}$-completeness.

To demonstrate that SDN can be used to collect additional information of the network in a novel way that help understanding the load of the network and help finding underutilised paths, I showed in Chapter 4 SYNrace, a method of intelligently using probe packets to exploit TCP's behaviour of filling buffers to infer network characteristics. The advantage of this approach is that no modifications to applications running on the network are needed. But the information that can be gathered by this approach is limited.

To overcome these limitations and allow collection of various information from applications and other methods (as SYNRace) in a central component, I introduced the concept of GlobalFIB that serves as a central database for the network that includes all information about the flows and allows APIs to use the information to optimise the forwarding of the network.

As a typical network will not only run one application and GlobalFIB has information collected about multiple applications running in a network, this information may conflict with each other. I have analysed these conflicts in Chapter 5 and provided insights and solution for detecting and solving conflicts between SDN applications.

As a software platform to implement and bring together all these building blocks, I have shown that the core I developed with other NetIDE members is a good platform to use and build upon.

Finally, I reckon that an SDN network is not static but the network and its software is undergoing frequent changes. As these freuqent changes might cause unwanted down-times, I have shown in Chapter 8 a reconfiguration approach, which is also implemented in the core, that allows dynamic reconfiguration in SDN networks.

9.2 Future work

Based on the insights I gained by working on topics of my thesis I see the following research areas for future research:

Future OpenFlow/SDN protocols

OpenFlow is a protocol that mainly models capabilities of *existing* switches. This traditional model did not foresee all the opportunities created by SDN. For example, as already mentioned in Section 4.6.2, the protocol lacks the possibility for time-stamped PACKET_IN messages as traditional switches do not need them. Adding more features into OpenFlow like that can create potential for other clever use of the network resources like SYNrace (Chapter 4).

Port Conflict resolution to ONOS/ODL

The focus in this thesis has been to implement conflict resolution outside the scope of an SDN controller. From the research perspective I have seen this as the more interesting and promising approach. In real SDN deployment often only a single SDN controller is used, which also still lack conflict resolution. Integrating the conflict resolution into the SDN controller would bring conflict resolution to these deployments.

Explore predication approaches

The methods presented here to optimise the traffic and collect statistics have used real-time data or statistics about the past. Adding models that incorporate predication about the future would a good extension of that work.

Extend GlobalFIB implementation

The current GlobalFIB implementation (Section 7.12) is only implemented as a proof-of-concept implementation. While this is sufficient to show that the approach indeed works, it does not allow using the idea in a productive network. A finished implementation would allows to use all the benefits this scenario.

The language defined in Section 7.3.3 is still very simple but the operators I defined can be used to create a much more powerful language or to use an existing query language and include the operator like Structured Query Language (SQL). That would allow statements like "SELECT * FROM flows WHERE destIP $<$ 192.168.0.23 AND sourceIP $\cap$ SELECT destIP FROM flows WHERE application = 'firewall'".

Bibliography

[1] *0MQ Distributed Messaging.* http://zeromq.org/. 2014.

[2] D. Eastlake 3rd, T. Senevirathne, A. Ghanwani, D. Dutt and A. Banerjee. *Transparent Interconnection of Lots of Links (TRILL) Use of IS-IS.* RFC 7176 (Proposed Standard). Internet Engineering Task Force, May 2014. URL: http://www.ietf.org/rfc/rfc7176.txt.

[3] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren and Robert S Schreiber. "HyperX: topology, routing, and packaging of efficient large-scale networks". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* ACM. 2009, p. 41.

[4] Ian F. Akyildiz, Ahyoung Lee, Pu Wang, Min Luo and Wu Chou. *A roadmap for traffic engineering in SDN-OpenFlow networks.* 2014. DOI: http://dx.doi.org/10.1016/j.comnet.2014.06.002. URL: //www.sciencedirect.com/science/article/pii/S1389128614002254.

[5] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav and George Varghese. "CONGA: Distributed Congestion-aware Load Balancing for Datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM.* SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 503–514. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626316. URL: http://doi.acm.org/10.1145/2619239.2626316.

[6] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta and Murari Sridharan. "Data Center TCP (DCTCP)". In: *Proceedings of the ACM SIGCOMM 2010 Conference.* SIGCOMM '10. New Delhi, India: ACM, 2010, pp. 63–74. ISBN: 978-1-4503-0201-2. DOI: 10.1145/1851182.1851192. URL: http://doi.acm.org/10.1145/1851182.1851192.

[7] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger and David Walker. "NetKAT: Semantic Foundations for Networks". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.* 2014.

[8] *Apache Karaf.* http://karaf.apache.org/.

[9] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford and David Walker. "SNAP: Stateful Network-Wide Abstractions for Packet Processing". In: *CoRR* abs/1512.00822 (2015). URL: http://arxiv.org/abs/1512.00822.

[10] Mauricio Arregoces and Maurizio Portolani. *Data center fundamentals.* Cisco Press, 2003.

[11] Alvin AuYoung, Yadi Ma, Sujata Banerjee, Jeongkeun Lee, Puneet Sharma, Yoshio Turner, Chen Liang and Jeffrey C. Mogul. "Democratic Resolution of Resource Conflicts Between SDN Control Programs". In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia: ACM, 2014, pp. 391–402. ISBN: 978-1-4503-3279-8. DOI: 10.1145/2674005.2674992. URL: http://doi.acm.org/10.1145/2674005.2674992.

[12] M. F. Bari, R. Boutaba, R. Esteves, L. Z. Granville, M. Podlesny, M. G. Rabbani, Q. Zhang and M. F. Zhani. "Data Center Network Virtualization: A Survey". In: *IEEE Communications Surveys Tutorials* 15.2 (bi2 2013), pp. 909–928. ISSN: 1553-877X. DOI: 10.1109/SURV.2012.090512.00043.

[13] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow and Guru Parulkar. "ONOS: Towards an Open, Distributed SDN OS". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA, 2014, pp. 1–6.

[14] Andreas Blenk, Arsany Basta, Martin Reisslein and Wolfgang Kellerer. "Survey on Network Virtualization Hypervisors for Software Defined Networking". In: *CoRR* (2015). URL: http://arxiv.org/abs/1506.07275.

[15] *Apache ARIES Blueprint*. https://aries.apache.org/modules/blueprint.html.

[16] M. Canini, P. Kuznetsov, D. Levin and S. Schmid. "A distributed and robust SDN control plane for transactional network updates". In: *Computer Communications (INFOCOM), 2015 IEEE Conference on*. 2015.

[17] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford et al. "A NICE Way to Test OpenFlow Applications". In: *NSDI*. 2012.

[18] *Apache Celix*. https://celix.apache.org/.

[19] Cisco. *Cisco FabricPath*. URL: https://www.cisco.com/c/en/us/solutions/data-center-virtualization/fabricpath/index.html.

[20] *Cisco IOS In Service Software Upgrade and Enhanced Fast Software Upgrade Process*. https://www.cisco.com/c/en/us/td/docs/ios/12_2sb/feature/guide/sb_issu.html.

[21] Cisco. *Nexus 7700 F3-Series 24-Port 40 Gigabit Ethernet Module Data Sheet*. http://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/data_sheet_c78-728410.html.

[22] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the third annual ACM symposium on Theory of computing.* STOC ’71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[23] David P. Dailey. “Uniqueness of colorability and colorability of planar 4-regular graphs are NP-complete”. In: *Discrete Mathematics* 30.3 (1980), pp. 289–293. ISSN: 0012-365X. DOI: http://dx.doi.org/10.1016/0012-365X(80)90236-8.

[24] W Dally and B Towles. *Principles and practices of interconnection networks.*

[25] Bruce S. Davie and Yakov Rekhter. *MPLS: Technology and Applications.* 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000. ISBN: 9781558606562.

[26] Amogh Dhamdhere and Constantine Dovrolis. “Open Issues in Router Buffer Sizing”. In: *SIGCOMM Comput. Commun. Rev.* 36.1 (Jan. 2006), pp. 87–92. ISSN: 0146-4833. DOI: 10.1145/1111322.1111342. URL: http://doi.acm.org/10.1145/1111322.1111342.

[27] *Distrubed ONOS.* https://wiki.onosproject.org/display/ONOS/Distributed+ONOS.

[28] A. Dixit, K. Kogan and P. Eugster. “Composing Heterogeneous SDN Controllers with Flowbricks”. In: *Network Protocols (ICNP), 2014 IEEE 22nd International Conference on.* Oct. 2014, pp. 287–292. DOI: 10.1109/ICNP.2014.50.

[29] *EIA/ECA-310 Cabinets, racks (including 19-inch racks, rack units), panels and associated equipment standard.* Tech. rep. Electronic Industries Alliance.

[30] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang and Amin Vahdat. “Hedera: Dynamic Flow Scheduling for Data Center Networks”. In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation.* NSDI’10. San Jose, California: USENIX Association, 2010. URL: http://dl.acm.org/citation.cfm?id=1855711.1855730.

[31] D. Farinacci, V. Fuller, D. Meyer and D. Lewis. *The Locator/ID Separation Protocol (LISP).* RFC 6830 (Experimental). Internet Engineering Task Force, Jan. 2013. URL: http://www.ietf.org/rfc/rfc6830.txt.

[32] *Apache Felix.* http://felix.apache.org/.

[33] *Floodlight OpenFlow Controller.* URL: http://www.projectfloodlight.org/floodlight/.

[34] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola and David Walker. “Frenetic: A High-level Language for OpenFlow Networks”. In: *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow.* 2010.

[35] Apache Foundation. *Apache Hadoop*. URL: http://hadoop.apache.org/.

[36] Jérôme François, Lautaro Dolberg, Olivier Festor and Thomas Engel. "Network Security Through Software Defined Networking: A Survey". In: *Proceedings of the Conference on Principles, Systems and Applications of IP Telecommunications*. IPTComm '14. Chicago, Illinois: ACM, 2014, 6:1–6:8. ISBN: 978-1-4503-2124-2. DOI: 10.1145/2670386.2670390. URL: http://doi.acm.org/10.1145/2670386.2670390.

[37] RL Gallawa. "Estimated cost of a submarine fiber cable system". In: *Fiber & Integrated Optics* 3.4 (1981), pp. 299–322.

[38] Tal Garfinkel, Mendel Rosenblum et al. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". In: *NDSS*. Vol. 3. 2003, pp. 191–206.

[39] GitHub. *POX Controller*. 2011. URL: https://github.com/noxrepo/pox.

[40] P. Brighten Godfrey, Igor Ganichev, Scott Shenker and Ion Stoica. "Pathlet Routing". In: *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*. SIGCOMM '09. Barcelona, Spain: ACM, 2009, pp. 111–122. ISBN: 978-1-60558-594-9. DOI: 10.1145/1592568.1592583. URL: http://doi.acm.org/10.1145/1592568.1592583.

[41] IEEE 802.1 Working Group. *IEEE 802.1D MAC Bridges*. URL: http://www.ieee802.org/1/pages/802.1D-2003.html.

[42] IEEE 802.1 Working Group. *IEEE 802.1Qbb - Priority-based Flow Control*. URL: http://www.ieee802.org/1/pages/802.1bb.html.

[43] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang and Songwu Lu. "BCube: a high performance, server-centric network architecture for modular data centers". In: *ACM SIGCOMM Computer Communication Review* 39.4 (2009), pp. 63–74.

[44] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang and Songwu Lu. "Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers". In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: ACM, 2008, pp. 75–86. ISBN: 978-1-60558-175-0. DOI: 10.1145/1402958.1402968. URL: http://doi.acm.org/10.1145/1402958.1402968.

[45] A. Aranda Gutiérrez, E. Rojas, A. Schwabe (Paderborn University), C. Stritzke, R. Doriguzzi-Corin, A. Leckey, G. Petralia, A. Marsico, K. Phemius and S. Tamurejo (IMDEA Networks). "All-in-one framework for next generation, composed SDN applications". In: *Proccedings of the 2nd IEEE Conference on Network Softwarization (NetSoft 2016)*. 2016, pp. 1–2.

[46] PA Aranda Gutiérrez, E Rojas, A Schwabe, C Stritzke, R Doriguzzi-Corin, A Leckey, G Petralia, A Marsico, K Phemius and S Tamurejo. “NetIDE: All-in-one framework for next generation, composed SDN applications”. In: *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE. 2016, pp. 355–356.

[47] Magnus K Herrlin et al. “Rack cooling effectiveness in data centers and telecom central offices: The rack cooling index (RCI)”. In: *Transactions-American Society of Heating Refrigerating and Air conditioning Engineers* 111.2 (2005), p. 725.

[48] *Icinga | Open Source Monitoring*. https://www.icinga.org/.

[49] IEEE. “Amendment to Carrier Sense Multiple Access With Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications-Aggregation of Multiple Link Segments”. In: *IEEE Std 802.3ad-2000* (2000), pp. i–173. DOI: 10.1109/IEEESTD.2000.91610.

[50] Cisco Visual Networking Index. “The zettabyte era–trends and analysis”. In: *Cisco white paper* (June 2016).

[51] *"Intent-Based Network Modeling"*. http://nemo-project.net/.

[52] *iPOPO: A service-oriented component model for Python*. http://ipopo.coderxpress.net/,

[53] RAJ JAIN and SHAWN A. ROUTHIER. “Packet Trains-Measurements and a New Model for Computer Network Traffic”. In: *IEEE Journal on Selected Areas in Communications* (1985).

[54] A. Jajszczyk. “Nonblocking, repackable, and rearrangeable Clos networks: fifty years of the theory evolution”. In: *IEEE Communications Magazine* 41.10 (Oct. 2003), pp. 28–33. ISSN: 0163-6804. DOI: 10.1109/MCOM.2003.1235591.

[55] Juniper. *Data sheet covering EX8200 Ethernet line cards*. http://www.juniper.net/us/en/local/pdf/datasheets/1000262-en.pdf.

[56] Xin Jin, Jennifer Gossels, Jennifer Rexford and David Walker. “CoVisor: A Compositional Hypervisor for Software-Defined Networks”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 87–101. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/jin.

[57] David B. Johnson and David A. Maltz. “Dynamic Source Routing in Ad Hoc Wireless Networks”. In: *IEEE Transactions on Mobile Computing* (1999). DOI: 10.1007/978-0-585-29603-6_5.

[58] *Juniper: Unified In-Service Software Upgrade*. https://www.juniper.net/documentation/en_US/junos/topics/concept/issu-oveview.html.

[59] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar and P. Brighten Godfrey. "VeriFlow: Verifying Network-wide Invariants in Real Time". In: *SIGCOMM Comput. Commun. Rev.* 42.4 (2012), pp. 467–472.

[60] D. Kreutz, F.M.V. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky and S. Uhlig. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76. ISSN: 0018-9219. DOI: 10.1109/JPROC.2014.2371999.

[61] M. Krueger and R. Haagens. *Small Computer Systems Interface protocol over the Internet (iSCSI) Requirements and Design Considerations*. RFC 3347 (Proposed Standard). Internet Engineering Task Force, July 2002. URL: http://www.ietf.org/rfc/rfc3347.txt.

[62] Maciej Kuzniar, Peter Peresini, Marco Canini, Daniele Venzano and Dejan Kostic. "A SOFT way for OpenFlow switch interoperability testing". In: *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. 2012.

[63] Bob Lantz, 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. Monterey, California: ACM, 2010, 19:1–19:6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868466. URL: http://doi.acm.org/10.1145/1868447.1868466.

[64] C. E. Leiserson. "Fat-trees: Universal networks for hardware-efficient supercomputing". In: *IEEE Transactions on Computers* C-34.10 (Oct. 1985), pp. 892–901. ISSN: 0018-9340. DOI: 10.1109/TC.1985.6312192.

[65] Ka-Cheong Leung, Victor O. K. Li and Daiqin Yang. "An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges". In: *IEEE Trans. Parallel Distrib. Syst.* 18.4 (Apr. 2007), pp. 522–535. ISSN: 1045-9219. DOI: 10.1109/TPDS.2007.1011. URL: http://dx.doi.org/10.1109/TPDS.2007.1011.

[66] Dan Levin, Andreas Wundsam, Brandon Heller, Nikhil Handigol and Anja Feldmann. "Logically Centralized?: State Distribution Trade-offs in Software Defined Networks". In: *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*. HotSDN '12. Helsinki, Finland: ACM, 2012, pp. 1–6. ISBN: 978-1-4503-1477-0. DOI: 10.1145/2342441.2342443. URL: http://doi.acm.org/10.1145/2342441.2342443.

[67] T. Li, B. Cole, P. Morton and D. Li. *Cisco Hot Standby Router Protocol (HSRP)*. RFC 2281 (Informational). Internet Engineering Task Force, Mar. 1998. URL: http://www.ietf.org/rfc/rfc2281.txt.

[68] Davis Martin. *Computability and unsolvability*. 1958.

[69] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: http://doi.acm.org/10.1145/1355734.1355746.

[70] *Mechanical structures for electronic equipment – Dimensions of mechanical structures of the 482,6 mm (19 in) series.* Standard. Geneva, CH: International Electrotechnical Commission, 2004.

[71] J. Medved, R. Varga, A. Tkacik and K. Gray. "OpenDaylight: Towards a Model-Driven SDN Controller architecture". In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. June 2014, pp. 1–6.

[72] B. Melander, M. Bjorkman and P. Gunningberg. "A new end-to-end probing and analysis method for estimating bandwidth bottlenecks". In: *Global Telecommunications Conference, 2000. GLOBECOM '00. IEEE.* Vol. 1. 2000, 415–420 vol.1. DOI: 10.1109/GLOCOM.2000.892039.

[73] Mark P Mills. "The cloud begins with coal: Big data, big networks, big infrastructure, and big power". In: *Digital Power Group* (2013).

[74] Jeffrey C. Mogul, Alvin AuYoung, Sujata Banerjee, Lucian Popa, Jeongkeun Lee, Jayaram Mudigonda, Puneet Sharma and Yoshio Turner. "Corybantic: Towards the Modular Composition of SDN Control Programs". In: *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks.* HotNets-XII. College Park, Maryland: ACM, 2013, 1:1–1:7. ISBN: 978-1-4503-2596-7. DOI: 10.1145/2535771.2535795. URL: http://doi.acm.org/10.1145/2535771.2535795.

[75] J. Moy. *OSPF Version 2.* RFC 1247 (Draft Standard). Obsoleted by RFC 1583, updated by RFC 1349. Internet Engineering Task Force, July 1991. URL: http://www.ietf.org/rfc/rfc1247.txt.

[76] J. Moy. *OSPF Version 2.* RFC 2328 (INTERNET STANDARD). Updated by RFCs 5709, 6549, 6845, 6860, 7474. Internet Engineering Task Force, Apr. 1998. URL: http://www.ietf.org/rfc/rfc2328.txt.

[77] S. Nadas. *Virtual Router Redundancy Protocol (VRRP) Version 3 for IPv4 and IPv6.* RFC 5798 (Proposed Standard). Internet Engineering Task Force, Mar. 2010. URL: http://www.ietf.org/rfc/rfc5798.txt.

[78] T. Narten, E. Nordmark, W. Simpson and H. Soliman. *Neighbor Discovery for IP version 6 (IPv6).* RFC 4861 (Draft Standard). Updated by RFCs 5942, 6980, 7048. Internet Engineering Task Force, Sept. 2007. URL: http://www.ietf.org/rfc/rfc4861.txt.

[79] NetIDE. *NetIDE composition XML schema specification file.* https://github.com/fp7-netide/Engine/blob/master/core/specification/CompositionSpecification.xsd.

[80] *Neutron's developer documentation.* http://docs.openstack.org/developer/neutron/.

[81] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya and Amin Vahdat. "PortLand: a scalable fault-tolerant layer 2 data center network fabric". In: *SIGCOMM Comput. Commun. Rev.* 39.4 (Aug. 2009), pp. 39–50. ISSN: 0146-4833. DOI: 10.1145/1594977.1592575.

[82] *The OpenDaylight Platform.* https://www.opendaylight.org/.

[83] *ONOS - Open Network Operating System.* http://onosproject.org/.

[84] *ONOS - Intent Framework.* https://wiki.onosproject.org/display/ONOS/Intent+Framework.

[85] *OSGi: The Dynamic Module System for Java.* https://www.osgi.org/. 2015.

[86] *Pelix/iPOPO: an OSGi framework for Python applications.* https://www.eclipsecon.org/europe2013/sites/eclipsecon.org.europe2013/files/osgi2013-pelix-prez.pdf.

[87] R. Perlman, D. Eastlake 3rd, D. Dutt, S. Gai and A. Ghanwani. *Routing Bridges (RBridges): Base Protocol Specification.* RFC 6325 (Proposed Standard). Updated by RFCs 6327, 6439, 7172, 7177, 7357, 7179, 7180, 7455, 7780, 7783. Internet Engineering Task Force, July 2011. URL: http://www.ietf.org/rfc/rfc6325.txt.

[88] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar et al. "The Design and Implementation of Open vSwitch." In: *NSDI.* 2015, pp. 117–130.

[89] D. Plummer. *Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware.* RFC 826 (INTERNET STANDARD). Updated by RFCs 5227, 5494. Internet Engineering Task Force, Nov. 1982. URL: http://www.ietf.org/rfc/rfc826.txt.

[90] G. Pongrácz, L. Molnár and Z. L. Kis. "Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK". In: *2013 Second European Workshop on Software Defined Networks.* Oct. 2013, pp. 62–67. DOI: 10.1109/EWSDN.2013.17.

[91] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma and Ying Zhang. "PGA: Using Graphs to Express and Automatically Reconcile Network Policies". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication.* SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 29–42. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787506. URL: http://doi.acm.org/10.1145/2785956.2787506.

[92] NetIDE FP7 Project. “D2.7 NetIDE Manual”. In: *NetIDE reports* (2016).

[93] Gaurav Raina, Don Towsley and Damon Wischik. “Part II: Control Theory for Buffer Sizing”. In: *SIGCOMM Comput. Commun. Rev.* 35.3 (July 2005), pp. 79–82. ISSN: 0146-4833. DOI: 10.1145/1070873.1070885. URL: http://doi.acm.org/10.1145/1070873.1070885.

[94] M. Rajagopal, E. Rodriguez and R. Weber. *Fibre Channel Over TCP/IP (FCIP)*. RFC 3821 (Proposed Standard). Updated by RFC 7146. Internet Engineering Task Force, July 2004. URL: http://www.ietf.org/rfc/rfc3821.txt.

[95] M. J. Rashti and A. Afsahi. “10-Gigabit iWARP Ethernet: Comparative Performance Analysis with InfiniBand and Myrinet-10G”. In: *2007 IEEE International Parallel and Distributed Processing Symposium*. Mar. 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370480.

[96] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford and David Walker. “Modular SDN Programming with Pyretic”. In: *USENIX ;login* 38.5 (Oct. 2013), pp. 128–134.

[97] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger and David Walker. “Abstractions for Network Update”. In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM ’12. Helsinki, Finland: ACM, 2012, pp. 323–334. ISBN: 978-1-4503-1419-0. DOI: 10.1145/2342356.2342427. URL: http://doi.acm.org/10.1145/2342356.2342427.

[98] Elisa Rojes, Sergio Tamurejo Roberto Doriguzzi-Corin, Andres Beato, Arne Schwabe, Kevin Phemius and Carmen Guerrero. “Are we ready to tackle Software Defined Networks? A Comprehensive Survey on Management Tools and Techniques”. In: *ACM Computing Surveys* (to be published).

[99] E. Rosen, A. Viswanathan and R. Callon. *Multiprotocol Label Switching Architecture*. RFC 3031 (Proposed Standard). Updated by RFCs 6178, 6790. Internet Engineering Task Force, Jan. 2001. URL: http://www.ietf.org/rfc/rfc3031.txt.

[100] Christian Esteve Rothenberg, C Macapuna, F Verdi, M Magalhães and András Zahemszky. “Data center networking with in-packet Bloom filters”. In: *Proc. SBRC*. 2010, pp. 553–566.

[101] Ori Rottenstreich, Marat Radan, Yuval Cassuto, Isaac Keslassy, Carmi Arad, Tal Mizrahi, Yoram Revah and Avinatan Hassidim. “Compressing forwarding tables”. In: *IEEE Infocom*. 2013.

[102] *The Ryu Project*. http://osrg.github.io/ryu/.

[103] Arne Schwabe, Pedro A. Aranda Gutiérrez and Holger Karl. "Composition of SDN Applications: Options/Challenges for Real Implementations". In: *Proceedings of the 2016 Applied Networking Research Workshop*. ANRW '16. Berlin, Germany, 2016, pp. 26–31. ISBN: 978-1-4503-4443-2. DOI: 10.1145/2959424.2959436. URL: http://doi.acm.org/10.1145/2959424.2959436.

[104] Arne Schwabe, Pedro A. Aranda Gutiérrez and Holger Karl. "Composition of SDN applications: Options/challenges for real implementations". In: *Proceedings of the 2016 Applied Networking Research Workshop*. ACM. 2016, pp. 26–31.

[105] Arne Schwabe and Holger Karl. "SynRace: Decentralized Load-Adaptive Multi-path Routing Without Collecting Statistics". In: *Proceedings of the 2015 Fourth European Workshop on Software Defined Networks*. EWSDN '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 37–42. ISBN: 978-1-5090-0180-4. DOI: 10.1109/EWSDN.2015.58. URL: http://dx.doi.org/10.1109/EWSDN.2015.58.

[106] Arne Schwabe and Holger Karl. "Using MAC Addresses As Efficient Routing Labels in Data Centers". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 115–120. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620730. URL: http://doi.acm.org/10.1145/2620728.2620730.

[107] Arne Schwabe and Holger Karl. "Using MAC Addresses As Efficient Routing Labels in Data Centers". In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN '14. Chicago, Illinois, USA: ACM, 2014, pp. 115–120. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620730. URL: http://doi.acm.org/10.1145/2620728.2620730.

[108] Arne Schwabe, Elisa Rojas and Holger Karl. "Minimizing Downtimes: Using Dynamic Reconfiguration and State Management in SDN Networks". In: *3rd IEEE Conference on Network Softwarization (NetSoft 2017)*. Bologna, Italy, July 2017.

[109] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori and Bill Snow. "OpenVirteX: Make your virtual SDNs programmable". In: *Proceedings of the third workshop on Hot topics in software defined networking*. ACM. 2014, pp. 25–30.

[110] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori and Bill Snow. "OpenVirteX: Make your virtual SDNs programmable". In: *Proceedings of the third workshop on Hot topics in software defined networking*. 2014.

[111] Rob Sherwood, Michael Chan, Adam Covington, Glen Gibb, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Masayoshi Kobayashi, Jad Naous et al. “Carving research slices out of your production networks with OpenFlow”. In: *ACM SIGCOMM Computer Communication Review* 40.1 (2010), pp. 129–130.

[112] Konstantin Shvachko, Hairong Kuang, Sanjay Radia and Robert Chansler. “The hadoop distributed file system”. In: *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE. 2010, pp. 1–10.

[113] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano et al. “Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network”. In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 183–197.

[114] Ankit Singla, Chi-Yao Hong, Lucian Popa and Philip Brighten Godfrey. “Jellyfish: Networking Data Centers, Randomly.” In: *NSDI*. Vol. 12. 2012, pp. 17–17.

[115] J. Sommer, S. Gunreben, F. Feller, M. Kohn, A. Mifdaoui, D. Sass and J. Scharf. “Ethernet - A Survey on its Fields of Application”. In: *IEEE Communications Surveys Tutorials* 12.2 (Second 2010), pp. 263–284. ISSN: 1553-877X. DOI: 10.1109/SURV.2010.021110.00086.

[116] OpenFlow Switch Specification. *Version 1.3.1 (Wire Protocol 0x04)*. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.3.1.pdf. Sept. 2012.

[117] Peng Sun, Ratul Mahajan, Jennifer Rexford, Lihua Yuan, Ming Zhang and Ahsan Arefin. “A Network-state Management Service”. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM ’14. Chicago, Illinois, USA: ACM, 2014, pp. 563–574. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626298. URL: http://doi.acm.org/10.1145/2619239.2626298.

[118] Cisco System. *Cisco IOS Packaging*. URL: http://www.cisco.com/en/US/products/sw/iosswrel/ps5460/products_qanda_item09186a00801af2c6.shtml.

[119] Cisco Systems. *Cisco NX-OS Licensing Guide*. URL: http://www.cisco.com/c/en/us/td/docs/switches/datacenter/sw/nx-os/licensing/guide/b_Cisco_NX-OS_Licensing_Guide/b_Cisco_NX-OS_Licensing_Guide_chapter_01.html.

[120] The NetIDE consortium. *NetIDE: An integrated development environment for portable network howpublished*. Jan. 2014. URL: http://www.netide.eu.

[121] Mikkel Thorup and Uri Zwick. “Compact Routing Schemes”. In: *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures.* SPAA ’01. Crete Island, Greece: ACM, 2001, pp. 1–10. ISBN: 1-58113-409-6. DOI: 10.1145/378580.378581.

[122] Walter HW Tuttlebee. *Software defined radio: enabling technologies.* John Wiley & Sons, 2003.

[123] Amin Vahdat, Mohammad Al-Fares, Nathan Farrington, Radhika Niranjan Mysore, George Porter and Sivasankar Radhakrishnan. “Scale-out networking in the data center”. In: *Ieee Micro* 30.4 (2010), pp. 29–41.

[124] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O’Malley, Sanjay Radia, Benjamin Reed and Eric Baldeschwieler. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing.* SOCC ’13. Santa Clara, California: ACM, 2013, 5:1–5:16. ISBN: 978-1-4503-2428-1. DOI: 10.1145/2523616.2523633. URL: http://doi.acm.org/10.1145/2523616.2523633.

[125] Tim Niklas Vinkemeier. “Composition and Orchestration of Network Control Applications”. MA thesis. University of Paderborn, 2015.

[126] P. Wette and H. Karl. “DCT2Gen: A Versatile TCP Traffic Generator for Data Centers”. arXiv:1409.2246.

[127] Damon Wischik and Nick McKeown. “Part I: Buffer Sizes for Core Routers”. In: *SIGCOMM Comput. Commun. Rev.* 35.3 (July 2005), pp. 75–78. ISSN: 0146-4833. DOI: 10.1145/1070873.1070884. URL: http://doi.acm.org/10.1145/1070873.1070884.

[128] Xiaowei Yang, David Clark and Arthur W. Berger. “NIRA: A New Interdomain Routing Architecture”. In: *IEEE/ACM Trans. Netw.* 15.4 (Aug. 2007), pp. 775–788. ISSN: 1063-6692. DOI: 10.1109/TNET.2007.893888. URL: http://dx.doi.org/10.1109/TNET.2007.893888.

List of Figures

List of Tables

List of Algorithms