UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Philip Wette

Optimizing Software-Defined Networks using Application-Layer Knowledge

Dissertation

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.)

Referees:
Prof. Dr. Holger Karl, Universität Paderborn, Germany
Prof. Dr. Stefano Salsano, Università degli Studi di Roma Tor Vergata, Italy

Additional committee members:
Prof. Dr. Friedhelm Meyer auf der Heide, Universität Paderborn, Germany
Jun.-Prof. Dr. Christian Plessl, Universität Paderborn, Germany
Prof. Dr. Heike Wehrheim, Universität Paderborn, Germany

Submission: June 2015
Examination: 07.07.2015

Abstract

Software-Defined Networking (SDN) is a game-changing new paradigm in telecommunications. Using its centralized control, SDN is able to reconfigure networks on time-scales of milliseconds instead of seconds or minutes, opening up possibilities for traffic engineering undreamed-of before.

This thesis studies the question *"To which extent does application-layer knowledge help traffic engineering to increase network performance?"*. To this end, it considers the two network types *circuit-switched* networks and *packet-switched* networks. Both types have different properties and feature different possibilities for reconfiguration. This is why this thesis approaches both of them individually, in very different ways. Prior to communication in a circuit-switched network, a dedicated circuit between both communication partners has to be established. This circuit is used exclusively, ensuring certain properties such as usable data rate or latency. In contrast, in the packet-switched domain, a shared medium is used for communication. Data is transferred using (small) data packets which are routed independently of each other over the shared network.

As a showcase for circuit-switched networks, this thesis considers optical wide area networks (WAN) and for packet-switched networks data-center (DC) networks are examined. WANs are covering large areas such as entire countries. They usually employ optical wavelength division multiplexing (WDM) technology providing high data rates to interconnect various smaller, diverse networks. DC networks, on the other hand, are deployed on a much smaller spatial area interconnecting servers owned by one operator.

To include application-layer knowledge into the optimization of WDM networks, first a new type of Routing and Wavelength Assignment (RWA) algorithm is proposed. The algorithm provides an interface between the management layer of a WDM network and the applications running on top of it. In a subsequent step, this interface is used to create an algorithm approaching the virtual topology design problem for software-defined IP-over-WDM networks. The algorithm leverages knowledge from the application layer to significantly increase the performance of optical WDM WANs.

To approach DC networks, first the necessary tools for experimental evaluation of traffic engineering in such networks had to be developed. MaxiNet is presented, which is a highly scalable emulation environment for SDNs. To mimic highly realistic DC traffic, DCT^{2}Gen was created which is the first publicly available generator for realistic data-center traffic. These tools are subsequently used to evaluate HybridTE, a new routing algorithm leveraging application-layer knowledge to build very low-cost SDN enabled data-center networks having higher performance than state-of-the-art TE techniques. At the showcase of HybridTE, this thesis shows how different qualities of information impact the quality of the traffic engineering scheme. This makes this thesis the first to show the relation between the amount of explicit information and the performance gained by this information in a realistic data-center environment.

Zusammenfassung

Software-Defined Networking (SDN) ist ein neues, wegweisendes Paradigma um Netze zu kontrollieren. Durch seine zentralisierte Kontrolle können Netze auf Zeitskalen von Millisekunden anstelle von Sekunden oder Minuten rekonfiguriert werden. Dies eröffnet vollkommen neue Möglichkeiten um Traffic Engineering zu betreiben.

Diese Dissertation widmet sich der Frage: „Welchen Nutzen besitzt Anwendungswissen für Traffic Engineering um die Leistungsfähigkeit von Netzen zu erhöhen?". Um diese Frage zu beantworten, beschäftigt sich diese Arbeit mit *leitungsvermittelnden*, sowie mit *paketvermittelnden* Netzen. Beide Netztypen haben unterschiedliche Eigenschaften und bieten unterschiedlichste Möglichkeiten der Rekonfiguration. Aus diesem Grund werden beide Typen individuell, jeweils mit verschiedenen Methoden, untersucht. Bevor Kommunikation in einem leitungsvermittelnden Netz möglich ist, muss zunächst zwischen jedem Kommunikationspaar ein dedizierter Übertragungskanal vermittelt werden. Jeder Übertragungskanal wird exklusiv genutzt. Dies stellt gewisse Eigenschaften der Kommunikation sicher, wie eine feste Latenz oder eine gewisse Datenrate. Im Gegensatz dazu wird in paketvermittelnden Netzen ein gemeinsam genutztes Medium verwendet in dem keine Garantien bezüglich der Kommunikationseigenschaften existieren. Daten werden in (kleinen) Paketen übertragen die unabhängig voneinander durch das Netz geleitet werden.

Als Beispiel für ein leitungsvermittelndes Netz dient in dieser Arbeit ein Weitverkehrsnetz (WAN). Als Beispiel für ein paketvermittelndes Netz wird ein Rechenzentrumsnetz genutzt. WANs decken sehr große Areale ab, wie beispielsweise ganze Länder. Es handelt sich dabei in der Regel um Glasfasernetze, die die Wavelength Division Multiplexing (WDM) Technologie verwenden. WANs werden in der Regel genutzt, um verschiedenste, kleinere Netze miteinander zu verbinden. Rechenzentrumsnetze decken eine wesentlich kleinere Fläche ab und verbinden eine Menge von Servern miteinander.

Um Anwendungswissen in der Optimierung von WDM Netzen zu nutzen, wird im Rahmen dieser Arbeit zunächst ein neuer Routing and Wavelength Assignment Algorithmus entwickelt. Dieser Algorithmus dient als Schnittstelle zwischen der Netzkontrollschicht und den Anwendungen, die das Netz zum Datenaustausch nutzen. Anschließend wird diese Schnittstelle von einem weiteren Algorithmus genutzt, der das Virtual Topology Design Problem von IP-over-WDM Netzen löst. Dieser Algorithmus nutzt Anwendungswissen und ist in der Lage, die Leistungsfähigkeit von

WANs signifikant zu erhöhen.

Um realistische Experimente auf Rechenzentrumsnetzen durchführen zu können, mussten zunächst die entsprechenden Werkzeuge entwickelt werden. Eines dieser Werkzeuge ist *MaxiNet*, eine hochgradig skalierbare Emulationsumgebung für SDNs. Für die Evaluation von neuen Traffic Engineering Techniken in Rechenzentrumsnetzen werden realistische Verkehrsmuster benötigt. Aus diesem Grund wurde *DCT*^{2}*Gen* entwickelt, der erste *öffentlich verfügbare* Generator für realitätsnahe Verkehrsmuster aus Rechenzentrumsnetzen.

MaxiNet und DCT^{2}Gen wurden anschließend verwendet, um HybridTE zu evaluieren. HybridTE ist ein neuer Routingalgorithmus, der Anwendungswissen ausnutzt, um leistungsfähige Rechenzentren mit Hilfe von preiswerten Netzkomponenten zu realisieren. Am Beispiel von HybridTE zeigt diese Arbeit, welchen Einfluss verschiedene Qualitäten von Anwendungswissen auf die resultierende Güte des Traffic Engineerings haben. So ist diese Arbeit die erste, die den Zusammenhang zwischen der Menge von verfügbaren, expliziten Informationen und der daraus entstehenden Güte in einer realistischen Rechenzentrumsumgebung zeigt.

Acknowledgment

I would like to thank Prof. Dr. Holger Karl for all the support and advice during the last years and Prof. Dr. Stefano Salsano for serving as a referee. Working at the Computer Networks Group was always fun; thanks to all my colleagues for making this a great time. I will never forget all the interesting and fruitful discussions, especially with my officemate Arne Schwabe. Thank you, Arne. In addition, I would like to thank Felix Wallaschek who contributed a lot of brilliant program code to the MaxiNet project.

This all would not have been possible without the constant mental support of my family. Thank you for always believing in me. I would like to especially thank my fiancée, Susanne, and my parents, Beatrice and Siegfried, for their support and patience throughout the whole course of my studies.

Contents

List of Figures

1 Introduction

Cisco's Visual Networking Index proclaimed the *Zettabyte Era* for 2016 because the annually transferred amount of global IP traffic will—for the first time in history—exceed a zettabyte (10^{21} bytes); even growing to 1.6 zettabytes in 2018. In turn, the Internet's infrastructure has to grow at a rapid pace; not only in quantity but also in quality. This thesis shows how to increase the efficiency of networking infrastructure by adding an intelligent interplay between the applications that generate traffic and the network infrastructure that transports this traffic.

This interplay leads to a new form of traffic engineering (TE) that tightly couples applications and network management, creating networks that adapt to application needs in virtually no time. In contrast to prior work, where applications adapt to the network infrastructure, this work develops networks that adapt to applications needs. The enabling technology here is *Software-Defined Networking* (SDN). With SDN, it is possible to define the behavior of a whole network using a central software component. This software component has full access to all internals of the physical network infrastructure enabling the redefinition of the behavior of any device on very short time scales. The possibilities for reconfiguration are so wide that each single flow in the network can be routed individually with routing paths computed using global knowledge.

The Internet consists of a large number of interconnected, autonomously managed networks. These networks fulfill different tasks and are built from different technologies using different methodologies. To scale up the Internet, all of these networks have to be scaled. The two main classes of networking methodologies are *circuit-switched* and *packet-switched* networks. Circuit-switched networks allocate resources exclusively for each pair of communicating endpoints. Exclusive allocation guarantees certain properties such as latency and throughput. In packet switched networks, all links are shared between all communicating pairs which means that the data transfers influence each other, hence no strict guarantees can be given for any transfer.

Due to the different technologies, methodologies, and properties of the networks, the possibilities for reconfiguration are diverse. This is why this thesis approaches both methods, circuit-switching and packet-switching, on their own. As a showcase for circuit-switched networks, wide area networks (WAN) are used. WANs span across large spacial areas such a entire countries or continents and interconnect other networks. For packet-switched networks, data-center (DC) networks are inspected, covering only a very small spacial area, directly interconnecting thousands of servers with each other.

Modern WANs employ optical Wavelength Division Multiplexing (WDM) technology that is mainly used to carry IP traffic directly without any other layers such as SONET or SDH in between. This is then called an IP-over-WDM network where multiple IP routers are interconnected using circuits established over the WDM network. Depending on the choice of the circuits, the neighborhood of the IP routers changes; communication between two routers that are not directly connected either requires a new circuit or a routing path over other routers. Obviously, for small amounts of traffic between a pair of routers a direct circuit is not necessary, however, for large amounts of data it might be beneficial to establish a new circuit, reducing traffic on other links. Obviously, the largest potential for reconfiguration of such a network is the choice of the circuits. To increase the efficiency of IP-over-WDM networks, this thesis proposes a scalable algorithm that at any time identifies which circuits to establish between which nodes in the network and which resources to allocate to establish these circuits. To find out which set of new circuits to establish and which set of old circuits to remove, a feedback loop between the IP routing layer (which in this case is the application using the network) and the WDM network control is created. Evaluation through a simulation study shows that by establishing the feedback loop to leverage knowledge from the application layer, the congestion in the network can be reduced by up to 50 % in the inspected scenarios.

While in circuit-switched networks the neighborhood can be changed by adding and removing new circuits, the topology of a packet-switched network is fixed and cannot be changed (apart from physical changes, i.e., laying new cables). The largest degree of freedom in such networks is routing. In traditional data-center networks, routing is calculated in a distributed fashion. The state of the art for data-center routing today is Equal-Cost-Multipath-Routing (ECMP). Here, each flow is assigned a random shortest path through the network. In symmetric networks, like data centers, ECMP spreads the flows equally over all links. If all flows had the same size, this would lead to equal load distribution on all links. If, however, multiple large flows are assigned to the same link, congestion occurs while there might be alternative paths with free residual data rate. In this situation, a load-aware routing algorithm with global knowledge of the current network state can help resolve congestion. This thesis shows how to design and evaluate a load-aware routing algorithm for data centers consisting of low-cost SDN network equipment. Such equipment has strong limitations on both its flow-table sizes and its processing capabilities. Evaluation shows that using the load-aware routing algorithm, the flow completion times can be reduced by up to 14.9 % compared to ECMP in a data-center environment.

1.1. Structure of this Thesis

This thesis is divided into the three different parts Circuit Switching, Packet Switching, and Emulation Tools. The first part describes a technique that allows optical WDM networks to automatically adapt to the applications running on top of them. This is different from what happens today where applications adapt their communication structure to the structure and properties of the underlying network. The peer-to-peer network Gnutella 0.6 [KM02] was one of the first applications with such an adaption process. Recently, the IETF published the protocol draft ALTO (Application-Layer Traffic Optimization) [APY+14] to provide applications with topology information of ISP networks. This enables applications to adapt their communication patterns to better fit to the underlying network. Note that this thesis investigates the problem the other way around. Here, networks are built that adapt to the communication structure of the applications. For the adaption process, information about the desired communication structure of the applications need to be available at the management plane of the network.

Chapter 3 presents a novel Routing and Wavelength Assignment (RWA) algorithm for optical WDM networks. The algorithm exposes an interface through which requests for lightpaths can be submitted. In case enough resources are available for the new lightpath, it is created. Otherwise, the algorithm computes sets of lightpaths which—in case they were to be removed from the network—yield enough free resources to fulfill the lightpath request. These sets are then passed back to the requester of the lightpath who can now choose which set to be removed, if any. The chapter first introduces a graph model along with the description and a complexity analysis of the novel RWA algorithm. To find out about the efficiency of the RWA algorithm in a WAN scenario, a simulation study is conducted.

Chapter 4 shows how the RWA algorithm developed in Chapter 3 can be used to build a WAN adapting to the requirements of the application running on top of it. A distributed algorithm for an IP-over-WDM network is presented where each IP router acts independently and selfishly. The approach is called *Selfish Virtual Topology Reconfiguration* (SVTR) and comprises three simple actions to adapt the virtual topology of the network to the requirements of the application. Evaluation shows that SVTR performs significantly better than a state-of-the-art virtual topology design algorithm which does not use any information from the application layer. The evaluation was conducted through a simulation study of the Deutsche Forschungsnetz (DFN) WAN topology along with real-world traffic patterns recorded in that network.

The second part of this thesis entitled PACKET SWITCHING focuses on routing in packet-switched networks. The leading question here is how much value application-layer knowledge has for traffic engineering in a data-center context.

Chapter 5 presents a routing algorithm called HybridTE which targets data-center networks. HybridTE distinguishes between small and large flows and handles them differently. To this end, HybridTE requires information about large flows which are either reported directly from the applications to HybridTE or estimated using so called *elephant detectors* (large flows as called *elephants* in the literature). Using this information, HybridTE has very low requirements on the switches while performing better than state-of-the-art traffic engineering techniques which have much higher requirements on the hardware. To study the effect of uncertain information on the quality of HybridTE (which might be induced by the elephant detector), different percentages of false positives, false negatives and different delays between the start of a large flow and its report to HybridTE are used in the evaluation. The results show that HybridTE is resilient against large numbers of false positives and that it still performs well even with 50% false negatives and reporting delays of up to one second.

Chapter 6 evaluates the quality of simple packet sampling as a possible elephant detection technique for data centers. To this end, first an extension for OpenFlow is presented. Using this extension, packet sampling capabilities of the switches can be controlled. Having control over packet sampling, it is possible to build a very simple elephant detector residing inside the OpenFlow controller. This chapter evaluates the quality of the generated information and concludes how valuable this information is for HybridTE.

While simulation is a reasonable and widely used technique for evaluation of circuit-switched networks, *emulation* is more suited to conduct research on packet-switched, low-latency networks such as data-center networks. Emulation is much more detailed than simulation, hence more realistic. High realism is very important for the evaluation of novel routing and traffic engineering algorithms for data centers because of the wide variety of effects caused by technical details such as a real TCP stack implementation. Although these effects are presumably negligible for large long-lasting flows, they have an essential influence on very small flows carrying only a few bytes. Traffic in data centers mainly consists of very small flows, which is why emulation is crucial in this scenario. Unfortunately, there was no network emulator available for emulating large networks such as a data center. Hence, to conduct the research presented in the second part of this theses, first the necessary tools had to be developed. These tools are presented in the third part of this thesis entitled EMULATION TOOLS.

Chapter 7 presents a highly scalable, distributed network emulator called *MaxiNet*. MaxiNet extends the famous Mininet emulation environment [LHM10] to span the emulation across several physical machines. This allows to emulate very large SDN networks on only a small number of physical machines each of which is called *worker*. In MaxiNet, each of these workers runs a Mininet instance and only emulates a part of the whole network. Switches and hosts are interconnected using GRE tunnels across different workers. MaxiNet provides a centralized API for controlling the emulation. The chapter discusses the design and presents an evaluation of the scalability of MaxiNet. The emulation showed that it is possible to emulate a mid-sized data center consisting of 3,600 hosts using only 16 workers in acceptable time.

Chapter 8 explains the design of DCT^{2}Gen, a highly realistic traffic generator for data-center traffic. To conduct highly realistic evaluations, a network emulator alone is not sufficient. The emulated traffic needs to be realistic, too. Unfortunately, no traffic traces from real data centers are publicly available. The two studies [BAM10, KSG$^+$09] published detailed statistical properties of data-center traffic on Ethernet level. DCT^{2}Gen takes these properties to create a schedule of TCP connections between a set of hosts. When this schedule is played out at a (emulated) data center, this creates Ethernet traffic with the same properties as given before. The chapter discusses the problems when computing such a schedule and explains the techniques required for solving them. Based on that, the design of DCT^{2}Gen is presented which is subsequently evaluated through MaxiNet emulations.

1.2. Contributions

The contents presented in this thesis were all developed in the time between October 2011 and March 2015. During that time, I authored nine research papers from which, by the time of writing, seven papers are published in peer-reviewed conference proceedings. Although I am the main author of all nine papers, in the remainder of this thesis I will change to first person plural to indicate that the corresponding findings are the result of joint work.

In addition, three open-source projects emerged from the work presented in this thesis, namely MaxiNet (Chapter 7), DCT^{2}Gen (Chapter 8), and NetSLS (Section 7.5). These projects can be found online under the URLs listed in Table 1.1.
Chapter 3 is based on the papers

- P. Wette and H. Karl. Using Application Layer Knowledge in Routing and Wavelength Assignment Algorithms. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2014
- P. Wette and H. Karl. Incorporating Feedback from Application Layer into Routing and Wavelength Assignment Algorithms. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013

Table 1.1.: URLs of the open-source projects emerged from this thesis.

MaxiNet	`https://www.cs.uni-paderborn.de/?id=maxinet` `https://github.com/MaxiNet/MaxiNet`
DCT^{2}Gen	`https://www.cs.uni-paderborn.de/?id=dct2gen`
NetSLS	`https://github.com/wette/netSLS`

Chapter 4 is based on the paper

- P. Wette and H. Karl. On the Quality of Selfish Virtual Topology Reconfiguration in IP-over-WDM Networks. In *Proceedings of the 19th IEEE Int. Workshop on Local and Metropolitan Area Networks*, 2013

Chapter 5 is based on the paper

- P. Wette and H. Karl. HybridTE: Traffic Engineering for Very Low-Cost Software-Defined Data-Center Networks. *arXiv preprint arXiv:1503.04317*, 2015. (submitted to the European Workshop on Software Defined Networks)

Chapter 6 uses parts of the paper

- P. Wette and H. Karl. Which Flows Are Hiding Behind My Wildcard Rule? Adding Packet Sampling to OpenFlow. In *Proceedings of the ACM SIGCOMM 2013 conference on Applications, technologies, architectures, and protocols for computer communication*, 2013

Chapter 7 is based on the papers

- P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Hassan Zahraee, and H. Karl. MaxiNet: Distributed Emulation of Software-Defined Networks. In *IFIP Networking Conference*, 2014
- P. Wette, A. Schwabe, M. Splietker, and H. Karl. Extending Hadoop's Yarn Scheduler Load Simulator with a Highly Realistic Network & Traffic Model. In *Proceedings of the 1st IEEE Conference on Network Softwarization*, 2015

Chapter 8 is based on the paper

- P. Wette and H. Karl. DCT^{2}Gen: A Versatile TCP Traffic Generator for Data Centers. *arXiv preprint arXiv:1409.2246*, 2014. (submitted to Elsevier Journal on Computer Communications)

and uses some parts of

- P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Hassan Zahraee, and H. Karl. MaxiNet: Distributed Emulation of Software-Defined Networks. In *IFIP Networking Conference*, 2014

2 Background

This chapter provides a brief overview of the basic terms and network technologies used throughput this thesis. In Section 2.1 the basics of modern optical networks are explained. Deep technical details which are not necessary to understand the concepts presented in this thesis are omitted and can be found in [RSS09]. Software-defined networking and one of its implementations for packet-switched networks, OpenFlow, is introduced in Section 2.2. Section 2.3 discusses properties of wide-area and data-center networks. This chapter does not intend to discuss related or competing work; this is done in the corresponding chapters of this thesis.

2.1. Optical Networks

Optical networks use light as a communication medium. As opposed to free-space optical communication, the optical networks considered in this thesis are communicating through fibers. These fibers are interconnecting optical network equipment. As the usable bandwidth in fibers is very large, very high data rates can be achieved. In addition, the attenuation in optical fibers is very low allowing large distances between network equipment. Both aspects favor optical networks as the key technology used in modern wide-area networks.

Before a pair of nodes is able to communicate via an optical network, the two communication partners need to setup what we call a *circuit*. A circuit is a pair of *lightpaths* between the two partners; one for each direction of the communication. A lightpath is a path through the optical network interconnecting its two endpoints. For technical reasons one single fiber can only be used unidirectional which is why fibers are commonly deployed in pairs; one fiber for each direction. If not stated otherwise, with the word fiber we always refer to such a pair of fibers. In turn, in this thesis lightpaths are always created in pairs, i.e., constructing a lightpath from node u to node v always implies the construction of a lightpath from v to u over the same path in the network. This makes model building simpler as with this assumption both fibers and lightpaths can be used for bidirectional communication.

2.1.1. Wavelength Division Multiplexing

In optical Wavelenth Division Multiplexing (WDM) networks, the available bandwidth on the fibers is divided into W independent communication channels denoted as $\Lambda = (\lambda_1, \lambda_2, ..., \lambda_W)$ for some fixed $W \in \mathbb{N}$. These channels are called *wavelengths*. Using different wavelengths allows concurrent data transmissions over the same fiber without causing any interference. Nevertheless, no two lightpaths must share the same wavelength on any shared fiber. In WDM, very high data rates can still be achieved; 10 Gbps and beyond per wavelength are not uncommon.

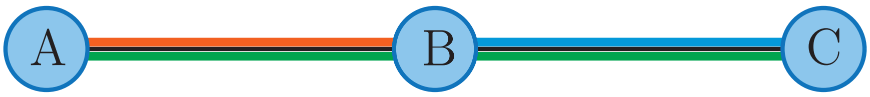


Figure 2.1.: Model of an optical WDM network with three established lightpaths.

Recalling the prior example, Figure 2.1 shows a small WDM network between three nodes. The example shows lightpaths between A and B, B and C, and A and C. As all lightpaths are using different wavelengths, there is no interference while all three nodes can communicate concurrently.

One of the more sophisticated optical network equipment used in modern WDM networks is called *reconfigurable optical add-drop multiplexer* (ROADM). Figure 2.2 shows the working principle of a ROADM with three connected fibers and one optical-to-electrical converter. Each attached fiber ends in a prism that breaks the light into its different wavelengths. From the prisms, the wavelengths encounter *wavelength selective mirrors* whose angle can be automatically adjusted to route the wavelength to the according outgoing fiber. Optical-to-electrical converters can be used to terminate a lightpath at a ROADM. In that case, the signal is converted and forwarded in the electrical domain to its final destination. Electrical-to-optical converters (missing in the figure) are used to create an optical signal corresponding to a given electrical signal.

ROADMs can also have the ability to convert between wavelengths. Thus, a signal arriving at wavelength λ_1 on one fiber can be converted to wavelength λ_2 and subsequently output on another fiber. Note that all these operations can be done in the

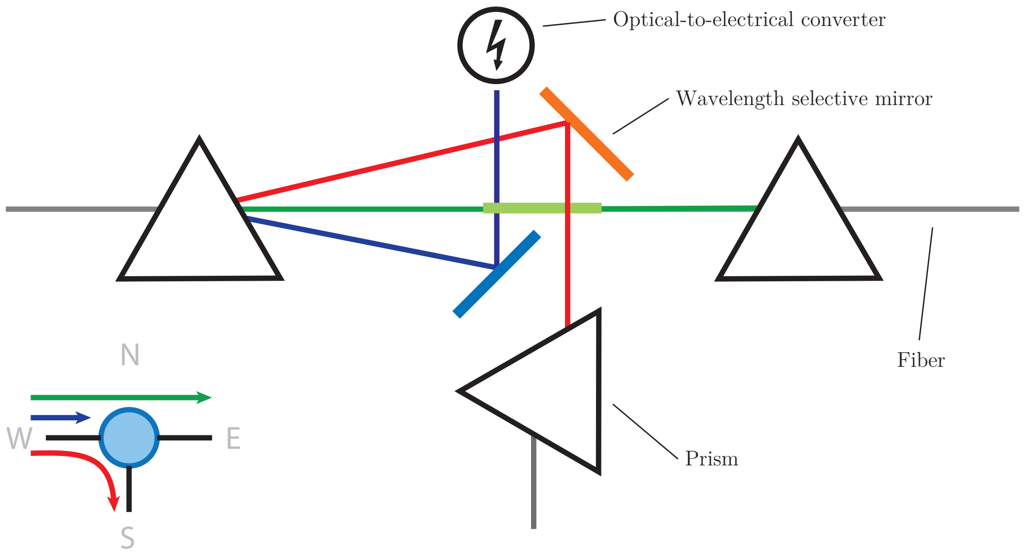


Figure 2.2.: Simplified working principle of a ROADM.

optical domain[1] which means no additional latency is induced by such operations. Depending on the particular ROADM design it might not be possible to convert between all wavelengths on all fibers; this is called limited wavelength conversion capabilities as opposed to full wavelength conversion capabilities where any conversion is possible. As with wavelengths, a ROADM might not be able to redirect light from every input port to every output port. This is referred to as limited light-switching (LS) capabilities [RSS09].

2.1.2. The Routing and Wavelength Allocation Problem

Whenever a new lightpath is to be added to a WDM network, this lightpath requires a path between its endpoints along with a wavelength for each fiber on that path. The problem of finding both a path and the corresponding wavelengths is called the *Routing and Wavelength Allocation Problem* (RWA Problem) [RSS09]. This problem has an online and an offline variant.

We define the *offline* Routing and Wavelength Allocation Problem as follows.

Given:

- An undirected simple graph $G = (V, E)$ representing the set V of ROADMs and the set $E \subseteq V \times V$ of fibers
- A set $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda_n\}$ representing the usable wavelengths per fiber
- A set $L \subseteq V \times V$ of lightpath requests

the offline Routing and Wavelength Allocation Problem asks for a route for each

[1] Although it is possible to conduct wavelength conversion in the optical domain, current commercial products stick to the optical-electrical-optical paradigm [RSS09]; it can be assumed that in the future all-optical conversion will be used in commercial products, too.

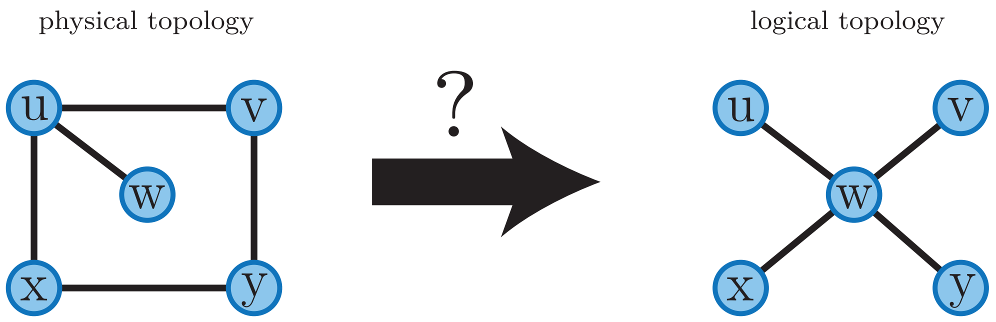


Figure 2.3.: The offline routing and wavelength allocation problem.

lightpath $l \in L$ along with a wavelength for each fiber on the route such that no two lightpaths use the same wavelength on any shared fiber. The problem is depicted in Figure 2.3 where the right-hand side depicts the demanded lightpaths and the left-hand side the physical topology.

The offline RWA problem can be reduced to finding the chromatic number of a graph [CGK92] and is thus $\mathcal{NP}$-complete. As the problem is highly relevant for running WDM networks efficiently, a lot of heuristics [XSBM08, WLYX07, WWH$^+$11, SBCL01, ZPC$^+$08] have been proposed.

While the offline variant of the problem asks to build a new network of lightpaths from scratch every time a new lightpath is requested, the online version of the problem asks to add a new lightpath to an *existing set* of already established lightpaths where the existing lightpaths must neither be rerouted nor allocated to different wavelengths.

The *online* Routing and Wavelength Allocation Problem is defined as follows.

Given:

- An undirected simple graph $G = (V, E)$ representing the set V of ROADMs and the set $E \subseteq V \times V$ of fibers
- A set $\Lambda = \{\lambda_1, \lambda_2, ..., \lambda_n\}$ representing the valid wavelengths per fiber
- A set $L \subseteq \mathcal{P}(E) \times \mathcal{P}(\Lambda)$ of established lightpaths
- A lightpath request between nodes $u, v \in V$

find a route and wavelength allocation for a new lightpath between u and v such that no two lightpaths use the same wavelength on any shared fiber. It is not allowed to modify any of the lightpaths $l \in L$ (in terms of the paths or assigned wavelengths). This is depicted in Figure 2.4. The dashed line between v and y on the right-hand side is the requested lightpath. The arrows on the left is the set of already established lightpaths. In the standard online RWA problem, a valid solution must contain all lightpaths $l \in L$ and the newly requested lightpath. In the *preemptive* RWA problem, not all $l \in L$ must exist in a valid solution, making is possible to preempt already existing lightpaths to free resources that can subsequently be used to establish the requested lightpath. But one should note that depending on the applications using the network, preempting a lightpath potentially has a negative effect on the performance

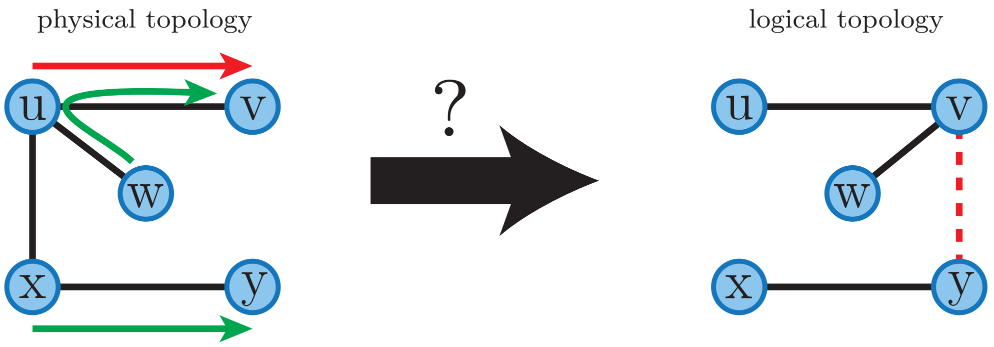


Figure 2.4.: The online routing and wavelength allocation problem.

of these applications.

A single instance of the online RWA problem is solvable in polynomial time. However, if we consider adding a series of lightpaths subsequently to the network using the polynomial online RWA algorithm, this solves the offline RWA problem. Hence, solving a series of the online RWA problem is $\mathcal{NP}$-hard.

2.1.3. Virtual Topology Design

As stated in Section 2.1.1, prior to communicating over a WDM network a lightpath has be be established between the two communicating partners. Thus, the set of established lightpaths in the network determines who can directly talk to whom. Figure 2.5 visualizes this for a small example. The lower layer depicts the ROADMs and fibers of the network; we call it the *physical topology*. The colored arrows in the physical topology are the established lightpaths. These lightpaths define the *logical topology* of the network. Each node of the physical topology has a corresponding node in the logical topology. The set of edges in the logical topology is defined as the set of endpoints of the established lightpaths in the underlying physical topology.

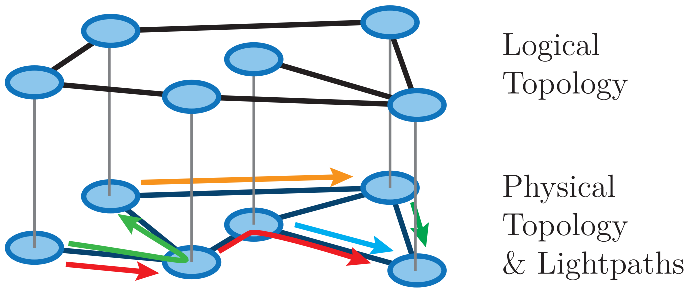


Figure 2.5.: The physical and logical topology of an optical WDM network.

The *virtual topology design problem* (VTD problem) asks for a set of realizable lightpaths defining a logical topology (over a given physical topology) that has certain properties. Commonly, these properties include, but are not limited to:

- The maximum perceived latency for any pair of nodes
- The diameter of the network
- Support for a certain traffic pattern without being congested
- Maximum node degree in the virtual topology

The desired properties of the logical topology are given as an input to the VTD problem. Note that solving the VTD problem requires solving the RWA problem as well which in turn means that VTD is $\mathcal{NP}$-hard [DR00]. There are a couple of heuristics solving the VTD problem. For a survey, see [DR00].

2.2. Software-Defined Networking

The term *Software-Defined Networking* (SDN) is primarily a marketing term that has no common technical definition in the scientific community. Driven by the control protocol *OpenFlow*, SDN recently became very popular for packet-switching networks. As of today, every large network equipment vendor has SDN products controllable via OpenFlow.

2.2.1. Characterization

For the scope of this thesis, SDN is characterized by the following three points:

- Separation of forwarding plane and control plane
- (Logically) centralized control through a controller-to-device protocol (such as OpenFlow)
- Capability to programmatically influence the behavior of the network

Separation of Forwarding Plane and Control Plane

Traditional network elements such as routers and switches consist of a *forwarding plane* and a *control plane*. As the name says, the forwarding plane is responsible for forwarding packets. To this end, it maintains a *forwarding table*. Whenever a packet arrives at the network element, the corresponding actions to be taken for the packet are looked up in the forwarding table. Possible actions include, but are not limited to, a port to output the packet on, rewriting IP or Ethernet header fields, or pushing/popping Multiprotocol Label Switching (MPLS) labels to the packet.

Whenever there is no entry in the forwarding table for a packet, the control plane is asked for the desired actions which are subsequently stored in the forwarding table. Depending on the desired routing behavior of the network, the control plane needs to run different routing algorithms to compute the actions. As traditional network elements do not have centralized control, routing algorithms are distributed.

To get as much logic as possible off the network elements, an SDN network element only consists of the forwarding plane and its forwarding table. This is why they are called *SDN forwarding element*, or short *SDN switch*. The forwarding table is exposed by an API to external devices. In SDN, forwarding elements do no execute any algorithms to fill the forwarding table on their own.

Centralized control through a controller-to-device protocol

To fill the forwarding tables of the switches, SDN introduces a logically centralized controller. This controller manages a set of switches and uses a control protocol to write to their forwarding tables. To make routing decisions, the controller requires information (for example about the topology, the connected devices, or the utilization of the links) from all managed switches. Using this information, the controller can make routing decisions with global knowledge about the network. The controller can quickly become a bottleneck which is why it is characterized as *logically* centralized. Distributed controllers (that may have a hierarchical structure) can be used to scale with increasing network size.

Capability to programmatically influence the behavior of the network

The centralized control makes SDN very versatile. In traditional networks, the single devices are closed-source; the only entities able to add new features to these devices are the manufactures. This makes it very hard (and expensive, too) to include new or custom routing protocols into an existing infrastructure. With SDN, this is different. The controller either has the capability to plug in new software components directly or it exposes a north-bound API that can be used by an external software to control the network. Writing both plugins and external control software can no longer be done solely by the vendors. This leads to lower cost and more custom-tailored solutions.

2.2.2. OpenFlow

OpenFlow [MAB+08] is currently the most successful open control protocol for SDN switches. It is maintained and actively developed by the Open Networking Foundation (ONF)[2]. At the time of writing this thesis, OpenFlow version 1.3 is the latest stable version. However, almost all OpenFlow-capable devices that can be bought today only support OpenFlow 1.0, which was released on December the 31th, 2009. All research in this thesis has been conducted with OpenFlow 1.0, which is why in the following we will use the term OpenFlow as a synonym for OpenFlow version 1.0.

[2]`https://www.opennetworking.org/`

Table 2.1.: The twelve OpenFlow header fields to identify a flow.

Input port (at the switch)
Ethernet source MAC address
Ethernet destination MAC address
Ethernet type
VLAN id
VLAN priority
Source IP address
Destination IP address
IP protocol
IP Type of Service bits
TCP/UDP source port
TCP/UDP destination port

Note that all concepts developed for OpenFlow 1.0 in this thesis are still valid and reasonable for OpenFlow 1.3.

OpenFlow-capable SDN switches are called OpenFlow switches and their forwarding tables are called *flow tables*. Each entry of a flow table consists of the three parts *header fields*, *counters*, and *actions*. Header fields in OpenFlow are summarized in Table 2.1 and consist of *twelve* fields from which eleven are actual packet header fields and the other one is the input port at the corresponding switch.

These header fields group the individual packets into *flows*. To lower the number of entries in the table, wildcards can be used that group multiple flows together and assign them to the same action. In the following, a flow table entry that does not use any wildcards is called an *exact-match* entry; otherwise, it is called a *wildcard* entry.

Counters are used to collect statistics about every flow. For details, see the OpenFlow 1.0 specification [ONF09]. OpenFlow defines four different types of actions: *Forward*, *Enqueue*, *Modify*, and *Drop*. The forward action specifies the port at which the packet leaves the switch. Enqueue can be used to enqueue the packet to a quality of service (QoS) queue. The modify action can be used to modify each of the header fields of the packet while the drop action simply drops the packet.

Upon the arrival of a packet at an OpenFlow switch, the header fields of the packet are looked up in the flow table. If a matching entry is found, the associated actions are applied to the packet. In case multiple entries match the same packet, the actions of the entry with the highest priority are executed. It is not possible to assign the same priority to multiple flow table entries. Exact-match entries always have a higher priority than wildcard entries. Priorities do not play any role in this thesis, thus they are omitted here. Details can be found in the specification.

If *no* matching flow table entry is found for a packet, this packet (both its header and payload) is encapsulated in a PACKET_IN message and forwarded to the OpenFlow controller. The controller then calculates an appropriate route for the packet through the network and uses the OpenFlow protocol to install corresponding flow table entries to the network.

2.3. Network Characteristics

2.3.1. Wide-Area Networks

Wide-area networks are used to interconnect multiple, geographically distributed networks. Therefore, WANs can spread over whole countries or even continents. Modern WANs are typically optical networks. This allows very high data rates and relatively low costs because the fibers can reach lengths up to several hundred kilometers. This, of course, causes quite high latencies because signals propagate in fibers at approximately $\frac{2}{3}\,c$ (c beeing the speed of light in a vacuum), which for example results in a signal propagation delay of 25 ms over a path of 5000 km.

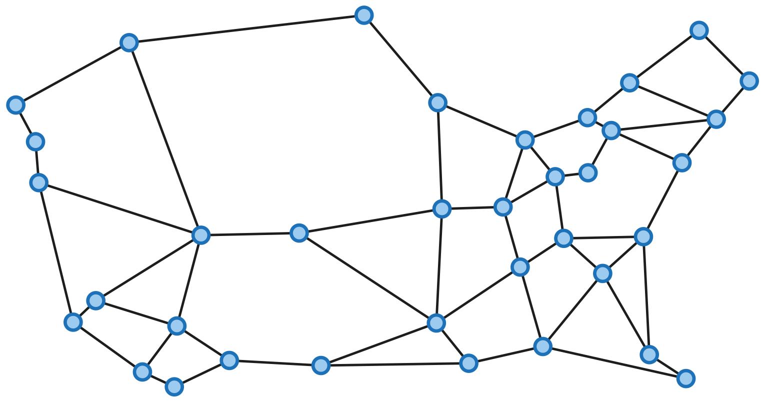


Figure 2.6.: Topology of the janos-us network (taken from `http://sndlib.zib.de`).

When designing WAN networks there are usually two leading design principals:

- Every location should be connected to at least two other locations to be resilient against failing links and locations.
- When integrating a new location to the network it should be connected to the geographically closest neighbors to save costs for cabling.

Due to these two principles, WANs are typically meshes at the physical layer as shown in Figure 2.6.

In the case of a WDM WAN, a logical topology is built based on such a meshed physical topology. The majority of modern WDM WANs use a technique called IP-over-WDM to build a packet-switched network layer directly on top of the circuit-switched WDM network [Liu02]. Finding an optimal logical topology along with an

IP routing on top of this logical topology is a multilayer optimization problem which is further addressed in Chapter 4 of this thesis.

2.3.2. Data-Center Networks

Data-center networks are used to interconnect a large number of servers. Depending on the type of the data center, the number of servers can add up to more than 100,000. As data centers typically host data-intense applications (such as Apache Hadoop [Apa]), a high-performance network infrastructure is crucial for efficient operation. In an ideal case, the network should be dimensioned such that at the same time, each server can talk to any other sever at full speed without creating congestion. Building such a network, however, is highly expensive. In addition, for real network workloads such a network would be overdimensioned and thus idle for most of the time, wasting resources.

To lower the number of required switches in a data center while providing high bisection bandwidths, hierarchical topologies are deployed in practice. An example of such a hierarchical topology is the *fat tree* (Figure 2.7). The fat tree has one root with a capacity to forward data at very high speed. The topology is called *fat* tree because moving up the tree towards the root a) the maximum data rate of the links is increasing and b) the switches have higher backplane capacities to process the higher data rates.

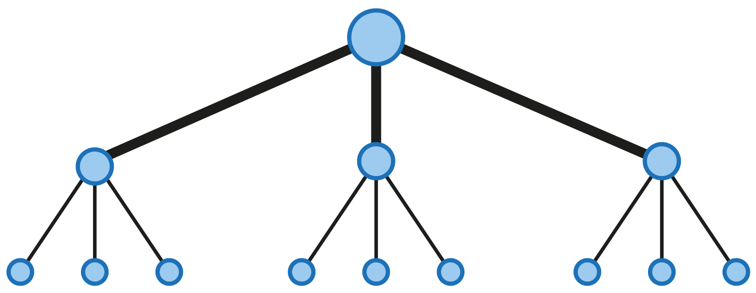


Figure 2.7.: Schematic view of a fat tree topology.

Obviously, the fat tree topology does not scale to a larger set of servers which is why large data centers usually employ a multi-rooted-tree topology, also called *Clos-like* topology named after Charles Clos [Clo53]. Clos-like topologies are typically built from three different layers, as depicted in Figure 2.8. The lowest layer comprises racks of servers with one top of rack (ToR) switch each. Traditionally, servers are connected to the ToRs with 1 Gbps but recently the amount of servers connected with 10 Gbps is raising due to falling prices for 10 Gbps network interface cards. Multiple ToR switches are grouped in *pods* with each pod having two pod switches. Each ToR is connected to both of the pod switches. Pod switches are then interconnected via the *core* layer of the network. In the depicted example each pod switch is connected to two core switches; of course, different cabling is possible, too. Typically, the links

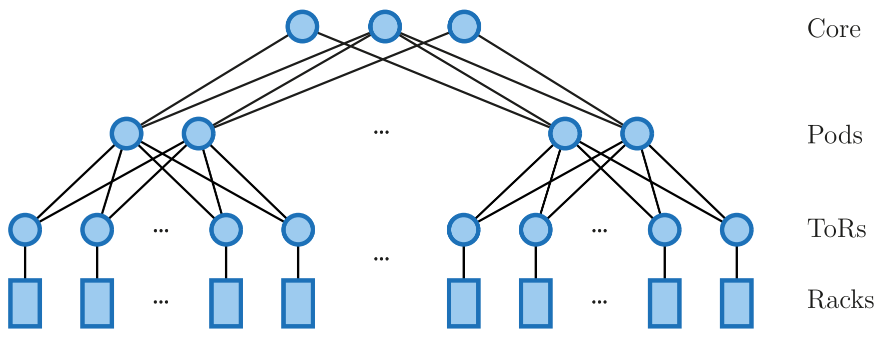


Figure 2.8.: Schematic view of a clos-like topology.

between core and pod, as well as between pod and ToRs are 10 Gbps. Due to the relatively short cable lengths, the latency in the network is primarily dominated by queuing delays at the switches and the network stack of the end hosts; it is (for empty queues) on the order of hundreds of microseconds.

In contrast to optical WDM networks, the degree of freedom for dynamical reconfiguration is lower in data-center networks. However, between any pair of nodes in the data center there are a lot of alternative routing paths. This makes routing the most promising degree of freedom for traffic engineering in data centers. More on traffic engineering in data centers can be found in Chapter 5.

Part I.

Circuit Switching

3 A Preemptive Routing and Wavelength Allocation Algorithm

Building networks that adapt to the requirements of arbitrary applications requires information about these particular requirements at the network level. For WDM networks, this information can subsequently be used to find a proper virtual topology. In a scenario where requirements change over time, the virtual topology has to adapt to these changes. This dynamic adaption process includes solving the online variant of the RWA problem. As mentioned in Section 2.1.2, an online RWA algorithm may preempt established lightpaths to free up resources that can subsequently be used to establish a new lightpath. Preemption only makes sense if the new lightpath has a higher value for the applications than the ones being preempted. As the RWA algorithm has not enough information to find out about the value of a particular lightpath for the application, an interface is required between the RWA algorithm and the applications using the network. This chapter describes how such an interface can be built.

In particular, this chapter presents a family of preemptive RWA algorithms for WDM networks that request feedback from applications. These algorithms have two distinguishing features: a) they can handle dynamic traffic by on-the-fly reconfiguration, and b) applications can give feedback for reconfiguration decisions and thus influence the preemption decision of the RWA algorithm, leading to networks which adapt directly to application needs.

Our algorithms handle various WDM network configurations including networks consisting of heterogeneous WDM hardware. To this end, we use the layered graph approach together with a newly developed graph model that is used to determine conflicting lightpaths.

3.1. Introduction

A *Routing and Wavelength Assignment* (RWA) algorithm is used to solve the RWA problem, as defined in Section 2.1.2. Static RWA algorithms are used to solve the *offline* variant of the RWA problem. Dynamic RWA algorithms, on the other hand, can be dynamically queried for additional lightpaths by specifying a source node and a destination node. Dynamic RWA algorithms thus solve the *online* RWA problem. Whenever a lightpath cannot be created, it is possible to *preempt* existing lightpaths to free resources required to set up the requested lightpath. Numerous algorithms for the dynamic case exist which either work non-preemptively [WWH+11, SBCL01, ZPC+08] or preemptively [XSBM08, WLYX07]. Neither method takes advantage of application-layer knowledge and this is why preemption decisions of existing preemptive RWA algorithms can be arbitrarily poor.

We present a preemptive routing and wavelength assignment algorithm for the online variant of the RWA problem which is able to use feedback from the application layer for making preemption decisions. The work flow of our proposed RWA algorithm is depicted in Figure 3.1. Upon a lightpath request (that is made by an application) that cannot be fulfilled, our algorithm computes multiple candidates for preemption to choose from. These candidates are then passed to the application to choose which of the candidates to preempt, if any. This way the requester can rate the candidates using application-layer knowledge and present the best candidate to the RWA algorithm which then executes the preemption and establishes the newly requested lightpath by using the freed resources. This serves as a powerful building block when constructing networks that automatically adapt to the logical topology of the applications running on top of it.

When our RWA algorithm is used in a scenario where multiple applications run concurrently on top of the WDM network, these applications compete for lightpaths. Depending on the desired behavior of the network, it might be necessary to restrict the preemption such that a specific application can only preempt lightpaths that this very application requested earlier. Otherwise it is possible that applications mutually preempt lightpath set up by other applications. As in the remainder of this thesis we concentrate on networks exclusively used by a single application, it was not necessary to build such an authorization scheme. However, by record keeping which lightpath was requested by which application, such a scheme can by integrated to the `reduce()` function used in Algorithm 1.

We identified two exemplary showcases for the use of our findings: the first one is a distributed algorithm to solve the Virtual Topology Design (VTD) problem in an IP-over-WDM network. The algorithm is further explained in Chapter 4. The second showcase are modern wireless cellular networks like Long Term Evolution (LTE). LTE Advanced uses what is called Coordinated Multi-Point (CoMP) transmission/reception. In CoMP one *User Equipment* (UE) is jointly served by multiple basestations in bad connectivity scenarios. For a set of basestations to use CoMP, a certain data

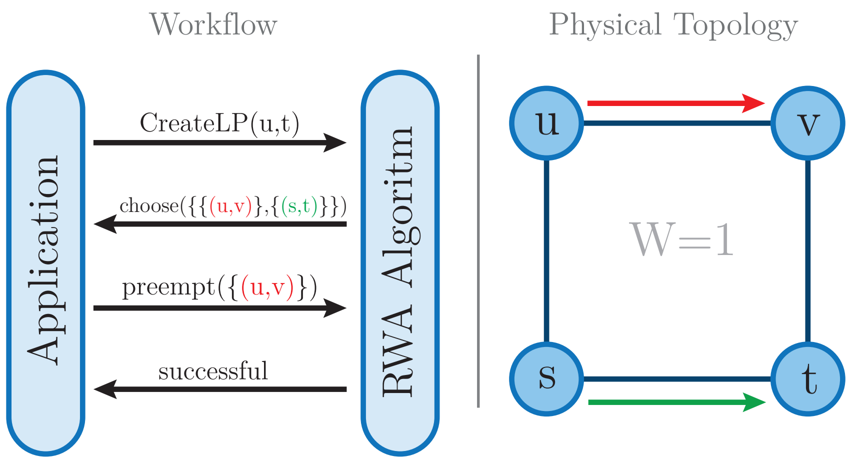


Figure 3.1.: Workflow of our proposed RWA algorithm. The right-hand side depicts the physical topology and established lightpaths.

rate between these nodes is required, where the data rate depends on a) the number of jointly served UEs and b) the quality of the wireless channel [DBKK12] between the basestations and UEs. Depending on these parameters, basestations query for new lightpaths in the backhaul network (a backhaul network is a wired network interconnecting basestations). The problem of selecting a proper preemption candidate in this scenario is a fairness problem which can *only* be solved with measurements of the wireless channel between basestations and UEs.

The rest of the chapter is structured as follows: Section 3.2 discusses related work in both RWA algorithms and lightpath rerouting algorithms. Section 3.3 presents assumptions and definitions along with the graph model that is used. Section 3.4 describes our preemptive RWA algorithm, which is evaluated by simulation in Section 3.5. Section 3.6 presents a conclusion.

3.2. Related Work

Existing routing and wavelength assignment algorithms can be differentiated by a) their assumption on the existence of wavelength converters in the network, b) their required knowledge about future lightpath requests, and c) their optimization criteria. In addition, they can be classified into algorithms which—given a new lightpath request—build new networks from scratch and algorithms trying to integrate the new lightpaths on-the-fly into the existing lightpath topology.

We are interested in preemptive RWA algorithms taking heterogeneous wavelength converters into account and handling new lightpath requests by on-the-fly network reconfiguration without knowledge about future lightpath requests. In addition to a low blocking probability, we are interested in a generic algorithm which is able to create lightpath topologies that comply with arbitrary custom constraints.

Several preemptive RWA algorithms with different strategies for choosing preemption candidates exist in the literature. In the following, we will give an overview over some of these algorithms.

In [GM02] a simple lightpath reconfiguration algorithm for IP over WDM networks is presented. The algorithm adapts the lightpath topology of a WDM network to the observed IP traffic. Each lightpath is assigned two so called *watermarks*: W_L and W_H. Whenever IP traffic on a lightpath is less than W_L this lightpath is considered as underutilized and therefore the lightpath is preempted. Analogously, whenever IP traffic on a lightpath is higher than the W_H watermark, a new lightpath is inserted into the network. Even though this algorithm uses preemption for network reconfiguration it is not possible to create lightpath topologies fulfilling any predefined constraints.

Ref. [XSBM08] studies the problem of service level agreement (SLA) violations in a WDM network under dynamic connection requests. SLAs specify a period in which a lightpath between two points in the network has to be active along with a availability requirement stating the amount of time the lightpath is allowed to be unavailable during that period.

A preemptive multi-class routing scheme with backup lightpaths is proposed in [WLYX07]. Lightpaths are assigned to different priority classes. Each Lightpath gets assigned a backup lightpath, which consists of a path through the network and a wavelength for each of the edges on the path that is not used by any other lightpath. In contrast to primary lightpaths, backup lightpaths are only planned but not active in the network. In case a primary lightpaths fails, for example due to a node or a fiber failure, the corresponding backup lightpath is created and takes over by handling all traffic previously carried by the failed lightpath.

In [XAG12] a routing and wavelength assignment algorithm for advance reservation of lightpaths is presented. The authors distinguish between lightpaths that are scheduled for future inclusion and lightpaths that are active. The difference between those two types of lightpaths is that scheduled lightpaths only *reserve* network resources while active lightpaths *occupy* resources. If, for a new lightpath request, there are not enough free resources left (where a resource is free when it is neither reserved nor occupied) the algorithm tries to rearrange the scheduled lightpaths to free enough resources for the requested lightpath. If this fails, the request is denied.

When building a lightpath topology incrementally, the available network resources undergo fragmentation. One method to deal with fragmentation is rerouting existing lightpaths in case a new lightpath request cannot be handled. Several lightpath rerouting algorithms exist, i.e. [XAG12, LL96, CByL07, CL05, YR04], assuming either the wavelength continuity constraint or exploiting wavelength conversions.

In [LL96] the “parallel move-to-vacant wavelength retuning” (MTV-WR) algorithm is proposed. MTV-WR is an RWA algorithm that does not exploit wavelength conversion. Upon arrival of a new lightpath request between two nodes u and v, MTV-WR first tries to find a route for the new lightpath on a free wavelength. If this is not possible the algorithm checks if it is possible to retune existing lightpaths to other wavelengths such that a route between u and v on a continuous wavelength is free. In [CByL07], MTV-WR is extended to MTV-OPA (move-to-vacant one-path-adjusting) for use in networks exploiting wavelength conversion. In addition to altering the wavelength of lightpaths, MTV-OPA is able to reroute existing lightpaths to different paths through the network. Unfortunately, the number of different rerouting candidates per request is bounded by one. Thus, many possible rerouting candidates are ignored, lowering the chances of successful rerouting.

Ref. [YR04] investigates rerouting based on calculating k shortest paths. Simulation results show that under light load rerouting based on the k shortest paths lowers the blocking probability significantly. But with increasing load the success rate of the presented rerouting algorithm decreases massively.

3.3. Generic Graph model

3.3.1. Assumptions and Definitions

We assume a system in which the user of the network has the possibility to actively tell the network operator to increase the possible data rate between two points in the network by creating a new lightpath. We call this a *request*. Every request has a (possibly unlimited) lifetime determining how long the corresponding lightpath has to be active. After a lightpath exceeds its lifetime or upon a release request, it is removed and its associated resources are freed for future use.

As introduced in Section 2.1.1, the physical topology of the optical network is given by an undirected graph $G_P = (V_P, E_P)$ where V_P are the nodes and E_P the fibers connecting those nodes. We assume each fiber can carry a total of W independent wavelengths, denoted by $\Lambda = \{\lambda_1, ...\lambda_W\}$, $W \in \mathbb{N}$, which are used for data transmission. Each of these wavelengths can be used for transmitting at a fixed data rate S.

A lightpath $l = \{(e_1, ..., e_m), (\lambda_1, ..., \lambda_m)\}$, $e_i \in E_P$, $\lambda_i \in \Lambda$, consists of a path through the network and a wavelength for each edge on this path. We do not require the lightpath to use the same wavelength on each edge; hence, conversion from one wavelength to another is possible.

3.3.2. Graph Construction

This section presents a variation of the layered graph model [SBCL01, ZPC+08]. The basic idea of the layered graph is that each of the W available wavelengths can be seen as an independent layer. This idea is visualized in Figure 3.2 and the construction is discussed in the following.

We construct the layered graph $G_L = (V_L, E_L)$ of a physical topology G_P as follows: As G_L consists of W different layers, for each node $v \in G_P$ the nodes $v_1, ..., v_W$ and $\tilde{v}_1, ..., \tilde{v}_W$ are added to V_L; they represent v on each layer. For each edge in G_P,

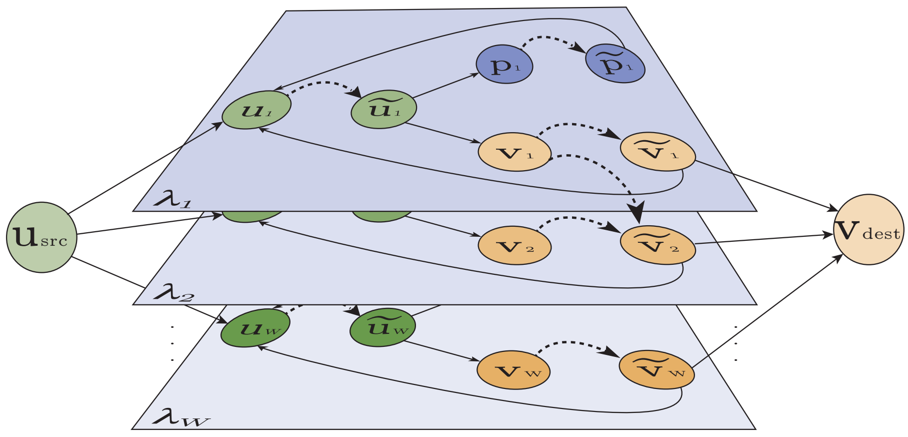


Figure 3.2.: Layered graph with W wavelengths. Wavelength edges dashed. In this example v is able to pass-through λ_i and to convert from λ_1 to λ_2.

$2W$ edges are added to G_L as follows: For each undirected edge $(u, v) \in E_P$, add the directed edges $(\tilde{u}_i, v_i)$ and $(\tilde{v}_i, u_i)$ to E_L for $1 \leq i \leq W$. In addition, for each node $v \in V_P$, add the nodes v_{dest} and v_{src} to V_L which are used to interconnect the different layers. These nodes are connected to the layers as follows: For each $v \in V_P$, add (v_{src}, v_i) and $(\tilde{v}_i, v_{dest})$ to E_L, $1 \leq i \leq W$.

We will now add the wavelength conversion capabilities of each node to V_L: If a node v is capable of converting an incoming signal on wavelength λ_i to λ_j, the edge $(v_i, \tilde{v}_j)$ is added to E_L. We call such an edge a *wavelength edge*. Note that edges between $\tilde{u}_i$ and v_j $(\forall i, j \in \{1, 2, .., W\})$ represent physical fibers, while edges between u_i and $\tilde{v}_i$ $(\forall i, j \in \{1, 2, .., W\})$ represent wavelength conversion capabilities.

This graph construction implies full LS capabilities and hence does not support ROADMs with limited light-splitting hardware. But as ROADMs with limited light-splitting support are built from multiple smaller interconnected ROADMs (which have full LS capabilities), the limitation can be modeled by replacing the limited ROADMs in G_P with its constituting parts.

3.4. Routing and wavelength assignment

Our RWA algorithm consists of four main parts: a) a greedy lightpath selection, b) a method to find conflicts with already established lightpaths in case the greedy strategy could not find a solution, c) a rerouting algorithm trying to reroute conflicting lightpaths to free resources for future usage, and d) a periodic cleanup module used to keep fragmentation low.

These parts are composed to constitute our algorithm as depicted by the flowchart shown in Figure 3.3. The flowchart is now described briefly; in the following, each building block is described in more detail. Upon a request for a lightpath between nodes u and v, at first the greedy first-fit algorithm is applied. If this algorithm could not find a free path, multiple sets of conflicting lightpaths R are computed. Such a set $c \in R$ is defined such that after removing *all* lightpaths in it, it is possible to create the requested new lightpath. Now it is checked if one of these sets can be rerouted such that a new lightpath between u to v can be established. If yes, R is passed to the requester of the new lightpath who can use its application-layer knowledge to choose which $c \in R$ is to be preempted, if any. Afterwards, the new lightpath between u and v can be constructed over a path and with a wavelength assignment computable by the greedy first-fit algorithm.

3.4.1. First-Fit: Greedy shortest path routing

When considering a scenario with dynamic request patterns where new requests are submitted and old requests expire, it is important that new requests can be handled by the system. As there is no information in advance about newly arriving requests, it seems prudent to grant a request using as few resources as possible, maximizing the free resources of the network.

One possibility to use minimal resources to create a lightpath is to use a shortest path between source and destination [RSS09]. This path can easily be found by applying a breadth-first search on the layered graph. As no two lighpaths must share the same wavelength on any edge, all edges of the layered graph have to be removed which are already in use by a lightpath. As pointed out in [SBCL01], it is even better to use the least loaded path between two points in the network, where the load of an edge is defined in terms of lightpaths traveling through the corresponding physical fiber. To find the least loaded path in the network, a weighted shortest path algorithm such as Dijkstra's can be used.

Depending on the desired behavior of the RWA algorithm, the greedy shortest path routing component either computes a shortest path on the layered graph using a breadth-first search or a least loaded path by using Dijkstra's algorithm.

3.4.2. Lightpath preemption

Establishing lightpaths between two points based on the greedy algorithm works as long as there are enough free resources available. But as more and more lightpaths are established the number of free paths in the graph decreases and at some point there is a request that cannot be handled. This section presents a method to find multiple sets of already established lightpaths such that the preemption of *all* lightpaths in *any one* of these sets allows the new request to be handled. In a subsequent step these sets can be analyzed to find the most feasible candidate for preemption. This can even be done in an interactive manner: Upon a new lightpath request that cannot be handled by the network, the RWA algorithm calculates multiple sets of preemption candidates. These sets are passed to the requester (which can be another program or even a human being) who is now able to analyze each candidate set. If all lightpaths

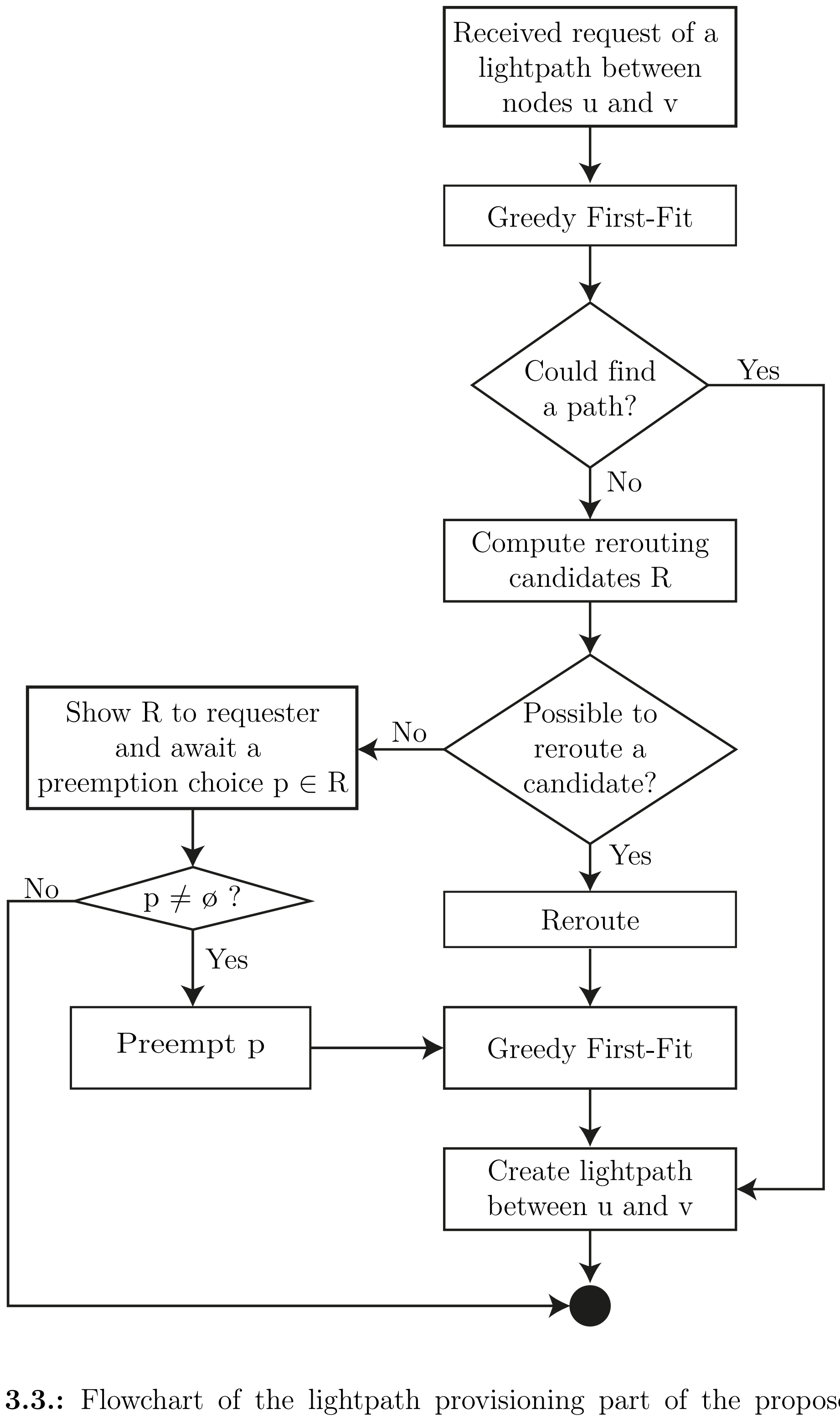
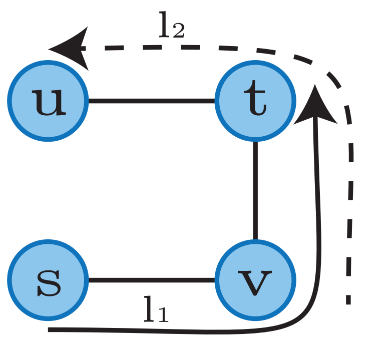
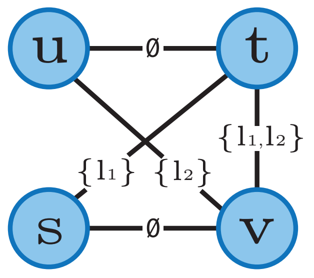

Received request of a lightpath between nodes u and v
Greedy First-Fit
Could find a path?
Yes
No
Compute rerouting candidates R
Possible to reroute a candidate?
No
Show R to requester and await a preemption choice p ∈ R
No
p ≠ ø ?
Yes
Yes
Reroute
Preempt p
Greedy First-Fit
Create lightpath between u and v

Figure 3.3.: Flowchart of the lightpath provisioning part of the proposed RWA algorithm.



(a) Simple graph with two established lightpaths l_1 and l_2 and $W = 2$ wavelengths.



(b) The corresponding conflict graph for lightpaths l_1 and l_2.

Figure 3.4.: Construction of the conflict graph G_C from a lightpath configuration.

in one of the sets are to be preempted this set is passed back to the RWA algorithm which preempts all lightpaths in it and finally creates the newly requested lightpath.

When multiple applications share the same network at the same time one application might want to preempt a lightpath previously established by another application. Depending on the desired behavior of the network this should not be possible. To implement an authorization scheme, the RWA algorithm has to keep track which application requested which lightpath. Then, each preemption request can be checked against the desired scheme.

To find the preemption candidates for a given request between nodes u and v, we first construct the *conflict graph* $G_C = (V_C, E_C)$ for all established lightpaths and the layered graph G_L. G_C is an undirected graph on the nodes of the physical topology and thus we set $V_C = V_P$. The edges of this graph are labeled with sets of lightpath identifiers where the labels of an edge (u, v) are denoted by $\text{labels}(u, v)$ and are defined by the following two rules:
(1) For each unused edge $(p_i, q_j) \in E_L$, $i, j \geq 1$, create an edge (p, q) with $\text{labels}(p, q) = \emptyset$ in the conflict graph. We call such an edge **unlabeled**. Thus for each pair of nodes in G_P, connected by a fiber having unused wavelengths left, there is an unlabeled edge in the conflict graph.
(2) To model the influence of taking down an already established lightpath l, the nodes on the path of l will form *one* strongly connected component in G_C. To this end, between all pairs of nodes on l, an edge is inserted in G_C. The edges of this strongly connected component are **labeled** with the corresponding lightpath identifier but only if there does not already exist an unlabeled edge.
Thus, edges inserted by rule (1) will never get assigned a label by rule (2) and we do not allow multiple edges between two nodes. An edge (u, v) in G_C with labels $L = \{l_1, .., l_m\}$ implies that by preempting one lightpath from L it is possible for the

greedy algorithm to create a new lightpath from u to v. Figure 3.4 shows the conflict graph for a small graph with two established lightpaths.

In the conflict graph, a path from u to v consisting only of unlabeled edges corresponds to an opportunity to create a new lightpath without interference by other lightpaths (provided that the required wavelength conversion capabilities are available). But since the greedy search did not find such a path in the layered graph, no such path in the conflict graph can exist, either. All paths from u to v in G_C hence have *at least* one edge with *at least* one label.

The labels of an edge in G_C describe the lightpaths that are forwarded over this edge. Thus, if we want to create a new lightpath between u and w we would have to remove one of the lightpaths from labels(u, v). Now suppose we have a path $p = \{(u, v), (v, w)\}$ in G_C. To create a lightpath between u and w we would have to remove one lightpath from the set of labels(u, v) and one lightpath from labels(v, w) (and not necessarily the same one). To obtain all possible combinations of lightpaths to remove, we have to calculate labels(u, w) $\times$ labels(w, v) where $A \times B$ is a variation of the Cartesian product, defined as:

$$A \times B = \begin{cases} \{a \cup b \mid a \in A, b \in B\} & \text{if } A \neq \emptyset \wedge B \neq \emptyset, \\ A & \text{if } A \neq \emptyset \wedge B = \emptyset, \\ B & \text{if } A = \emptyset \wedge B \neq \emptyset, \\ \emptyset & \text{else.} \end{cases}$$

This obviously generalizes to paths with more than 2 hops.

To determine the *complete* list of all possible preemption candidates, all possible paths in G_C between the nodes u and v are required. Then, for each of these paths, the possible preemption candidates can be computed by taking the cartesian product of the labels of each edge on the path and afterwards remove all combinations that are not feasible due to wavelength conversion limitations.

As it is not practical to compute all paths between two nodes in a network [Yen71] we concentrate on finding a fixed number c of paths. These paths are chosen such that there are no two paths p_1 and p_2 with labels$(p_1) \subseteq$ labels(p_2) (where labels$(p) = \bigcup_{(u,v)\in p}$labels$(u, v)$) because otherwise, if we were to preempt p_2, more lightpaths would be preempted than necessary. To find these c paths we use a slightly modified version of Dijkstra's shortest path algorithm, which has the following properties:

- The algorithm stops if c different paths between a source s and a destination d are found.
- During the computation for each node $n \neq d$ an unlimited number of paths from s to n are stored.
- If there are two paths p_1 and p_2 and it holds that labels$(p_1) \subset$ labels(p_2), then p_1 is shorter than p_2.
- There is no other notion of the length of a path.

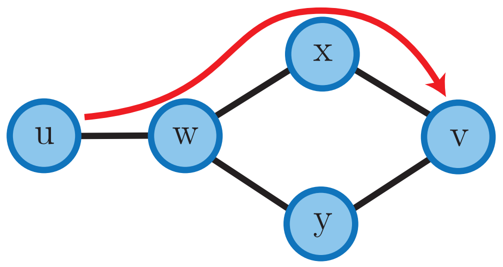


Figure 3.5.: To grant a request from x to v (with $W = 1$) the lightpath between u and v can be rerouted over y.

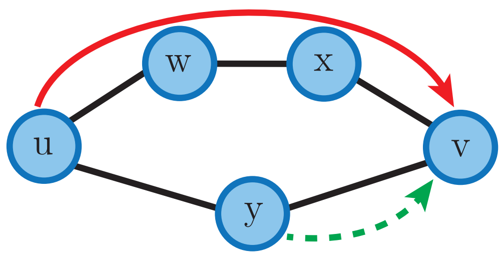


Figure 3.6.: If the solid shaped lightpath is older than the dashed one, cleanup will not be successful.

This modified Dijkstra's algorithm can be seen in Algorithm 1. The function `reduce` is used to remove obsolete elements from the set `newLabels`. To this end, it only keeps the smallest sets of labels where "smaller than" is defined by set inclusion. In addition, this function removes all lightpath combinations that cannot be merged due to wavelength conversion limitations.

Algorithm 1 FINDPREEMPTIONCANDIDATES(u, v)

```
1:  q.enqueue(u);
2:  for each node n do
3:     onWay(n) ← {}
4:  end for
5:  while not q.isEmpty() and |onWay(v)| < c do
6:     $s$ ← q.dequeue()
7:     for each $n \in$ neighbors($s$) do
8:        newLabels ← onWay(s) × labels(s,n)
9:        newLabels ← newLabels ∪ onWay(n)
10:       newLabels ← reduce(newLabels)
11:       if |newLabels| > |onWay(n)| then
12:          q.enqueue(n)
13:          onWay(n) ← newLabels
14:          if |onWay(v)| ≥ c then
15:             return onWay(v)
16:          end if
17:       end if
18:    end for
19: end while
20: return onWay(v)
```

3.4.3. Rerouting before preemption

Before any established lightpath is preempted it should be checked if it is possible to reroute one of the preemption candidate sets. Figure 3.5 shows a simple example where rerouting is possible. In the example there is a lightpath from u to v via w

and x. Now we want to add a new lightpath from x to v. As there are no more free resources to grant this request, we would have to preempt the lightpath from u to v. But as rerouting this lightpath over w and y yields free resources on the edge (x, v), the new lightpath request can be granted without preempting the existing one.

Our rerouting works as follows: Upon arrival of a request for a lightpath from u to v that cannot be handled, use Algorithm 1 to calculate the set R of preemption candidate sets. Now it is checked if it is possible to remove all lightpaths of one set $C \in R$, create a new lightpath from u to v, and afterwards reinsert all previously removed lightpaths again in order of their arrival into the network. If this step succeeds for some $C \in R$, then we found a way to insert a new lightpath from u to v without preemption. Note that the success of this operation depends on the order in which lightpaths are reinserted. As for large R it would be too time consuming to try every sequence we concentrate on the order of arrival only.

3.4.4. Periodic Cleanup

After successively granting new requests and removing expired lightpaths from the network, the free resources of the network undergo fragmentation. As fragmentation leads to uneven resource utilization and uneven resource utilization yields a higher blocking probability for new requests, we try to counteract fragmentation. This is done by periodically recalculating paths and wavelength assignments for all established lightpath. To this end, we first hypothetically remove all lightpaths from the network. Now we try to reinsert all lightpaths again in order of their arrival onto the empty graph. Note that, again, the success of this operation depends on the order in which the lightpaths are added to the network and thus this operation need not always be successful. See Figure 3.6 for a configuration where cleanup fails. Only if this hypothetical experiment succeeds, we apply the changes to the real network.

3.4.5. Complexity analysis

Our proposed RWA algorithm consists of the three phases greedy path selection, conflicting lightpath computation, and rerouting. We will give a complexity analysis for each of these phases and compose them to get the overall run time.

For the greedy path selection with "least loaded paths first" Dijkstra's shortest path algorithm is run on the layered graph. The layered graph $G_L = (V_L, E_L)$ of a physical topology $G_P = (V_P, E_P)$, $N = |V_P|$, $E = |E_P|$, with W different wavelengths, consists of $|V_L| = 2W(N+1)$ nodes and, in case of full WC capabilities, of $|E_L| = NW^2 + EW + 2NE$ edges. As W is a constant, it follows that $|E_L| \in \mathcal{O}(NE)$ and $|V_L| \in \mathcal{O}(N)$. The runtime of the greedy algorithm is thus $\mathcal{O}(N \log N + NE)$.

To compute conflicting lightpaths, first the conflict graph is built. This takes time $\mathcal{O}(NE + RN^2)$ where R is the number of established lightpaths in the graph. The conflict graph has N nodes and at most $\frac{(N-1)^2}{2}$ undirected edges, where each edge can have at most R ($\leq WE$) labels. Finding c preemption candidate sets based on the conflict graph is the same as finding c shortest paths on a multigraph where each single label $l \in \text{labels}(u, v)$ is represented by one edge (u, v). This graph would consist of N Nodes and $\frac{(N-1)^2}{2} \cdot R \in \mathcal{O}(N^2WE)$ edges. Thus, the runtime of finding c sets of preemption candidates is $\mathcal{O}(c(N^3WE + N^2 \log N))$ [Yen71].

The run time of the rerouting algorithm is cL times the run time of the greedy path selection, where L is the maximum size of a set of preemption candidates and upper bounded by R.

Cleaning up the network consists of two steps: a) remove all lightpaths and b) add all lightpaths with the greedy algorithm. This can be done in $\mathcal{O}(R \cdot (N \log N + NE))$.

The worst case in terms of run time is when neither the greedy algorithm can find any free path nor the rerouting algorithm finds a set of lightpaths that can be rerouted. As finding the preemption candidates is the dominant task the overall run time is $\mathcal{O}(c(N^3WE + N^2 \log N)$.

From the preceding discussion we know that the space required to store the layered graph is $\mathcal{O}(NE)$ and for the conflict graph $\mathcal{O}(N^2WE)$. The space required for calculating c preemption candidates based on the c shortest paths algorithm is $\mathcal{O}(N^2WE)$ [Yen71]. Thus, the space complexity of our proposed algorithm is $\mathcal{O}(N^2WE)$.

3.5. Simulation

3.5.1. Model

For the simulations of our proposed algorithm we used the janos-us network (Figure 2.6) which consists of 26 nodes and 42 bidirectional links. It is a (fictional) wide-area network comparable to large provider backbone networks. We assume $W = 16$ and full WC and LS capabilities.

We assume dynamic traffic in a circuit-switched network and perform a round-based simulation. At the beginning of each round, a source node chooses a destination node from all the other nodes uniformly at random and generates a lightpath request with a holding time t measured in rounds. If this request is granted, it will use the allocated lightpath for t rounds.

For the generation of requests, we use a Poisson process. This is a widely used assumption for traffic in WDM networks [SBCL01, ZPC+08, RS95]. We use a Poisson distribution with parameter λ_p to determine the arrival of new requests. The holding time of each request is exponentially distributed with rate λ_e. All arrivals and holding times are pairwise independent of each other. To create a given load l (in Erlangs) in the network with a mean holding time λ_e^{-1}, we set $\lambda_p = l \cdot \lambda_e$. To calculate confidence intervals, each experiment simulated 10,000 rounds and was repeated 20 times.

3.5.2. Results

The metric used to determine the performance of an RWA algorithm is the blocking probability: the probability of denying an incoming lightpath request. In our simulation we did *not* make use of the preemption feature. The given results thus show the performance of our RWA algorithm, where the blocking probability can equally be seen as the probability of having to make preemption decisions. We decided for this kind of evaluation because otherwise, the results in terms of blocking probability would *massively* depend on the kind of application that runs on the network and therefore neither be comparable with other work nor be intuitively interpretable. For example, an application that always preempts one of the preemption candidates would lead to a blocking probability of 0%.

To find out how the number of different rerouting candidates affects the resulting blocking probability of the RWA algorithm, we conducted the following experiment: We fixed the load in the network to 127 Erlangs, average holding time of lightpaths to 20 rounds and turned periodic cleanup off. Then we varied the number of rerouting candidates from 2 to 10 (parameter c) and looked at the resulting blocking probabilities. Figure 3.7 shows the outcome of this experiment: Higher values for c are leading to a lower blocking probability which means that–not surprisingly–rerouting works better with increasing number of choices.

The frequency at which the periodic cleanup is executed impacts the blocking rate, too: Figure 3.8 shows the influence of the number of rounds between two subsequent cleanup operations on the blocking probability. For the experiment we turned the rerouting feature off, fixed the load to 127 Erlangs, average holding time of a lightpath to 20 rounds and varied the number of rounds between two cleanups from 2 to 20. As neither preemption nor rerouting is used in the experiment, we set $c = 0$. It can be seen that with less frequent cleanups, the blocking probability increases.

To study the impact of holding time on the blocking probability, we set the rounds between two cleanups to $\frac{1}{4}\lambda_e$ and c to 7. We fix the average load (in Erlangs) and vary the holding time. Figure 3.9 shows the results of this experiment with and without rerouting: For $\lambda_e^{-1} > 15$ the blocking probability does not change significantly by increasing the holding time. Thus, the holding time does not have any significant influence on the quality of our presented algorithm.

Figure 3.10 plots the average hop count of the established lightpaths for the considered load levels. As it can be seen the cleanup routine has a significant impact on the average hop count of the established lightpaths since it decreases by up to 17% by periodically cleaning up the lightpaths. When using the rerouting feature in addition to the cleanup routine the average hop count does not increase significantly which is important because this implies that latency is not affected by using the proposed rerouting feature of our algorithm.

Figure 3.11 illustrates the success rate of the rerouting operation for different load levels. Even in a highly loaded scenario, rerouting is possible as the success rate of

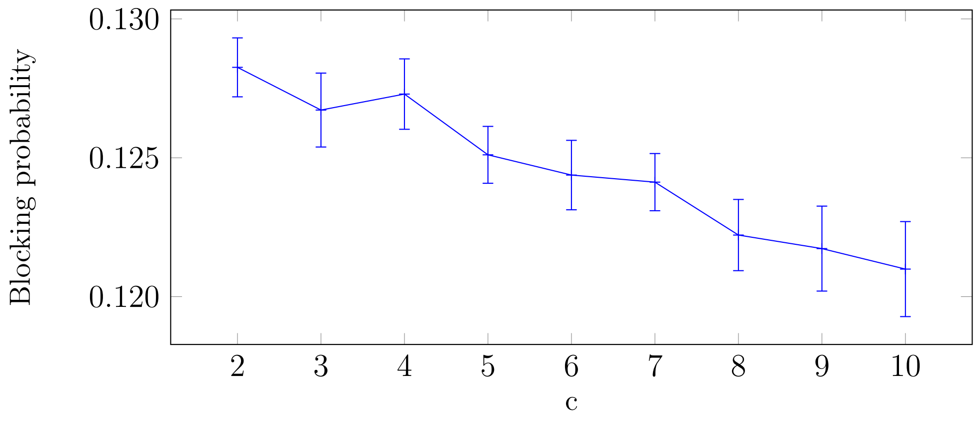


Figure 3.7.: Influence of number of rerouting candidates on the blocking probability. Algorithm run at 127 Erlangs, holding time 20 rounds and without periodic cleanups. Greedy routing configured to use least loaded paths first. Error bars illustrate the confidence interval for a confidence level of 95%.

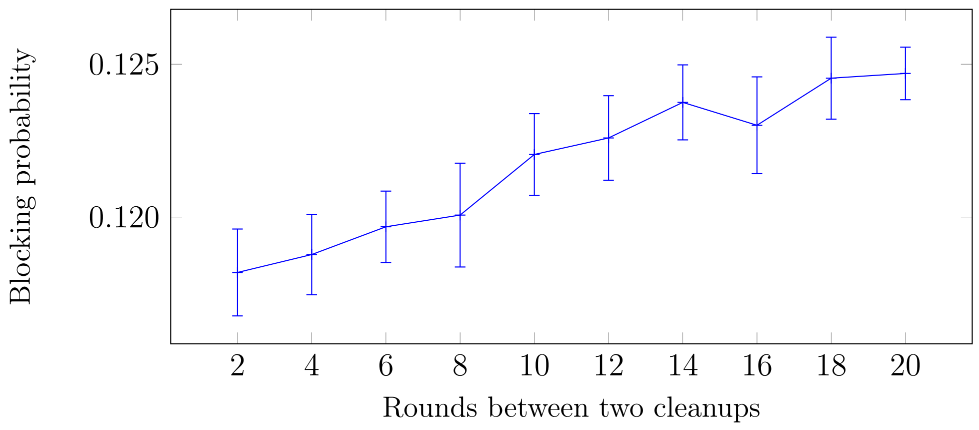


Figure 3.8.: Influence of the number of rounds between two subsequent cleanups on the blocking probability. Algorithm run at 127 Erlangs and holding time 20 rounds. Rerouting feature off. Greedy routing configured to use least loaded paths first. Error bars illustrate the confidence interval for a confidence level of 95%.

our algorithm decreases very slowly with increasing load. In Figure 3.12 the average cardinality of each rerouting candidate is plotted. It can be concluded that (for janos-us) even under high load it is sufficient to preempt only *one* existing lightpath to grant a new lightpath request.

For the performance analysis of our proposed method we considered six different configurations of our algorithm. These configurations can be seen in Table 3.1, where c corresponds to the number of considered rerouting candidates (see line 5 in Algorithm 1), “Cleanup” is the number of rounds between two cleanups (where ∞ corresponds to no cleanups at all) and “Path” is the method used to find free paths in the network. Note that the first two configurations correspond to the plain greedy algorithm using either the first found shortest path or a least loaded path to set up

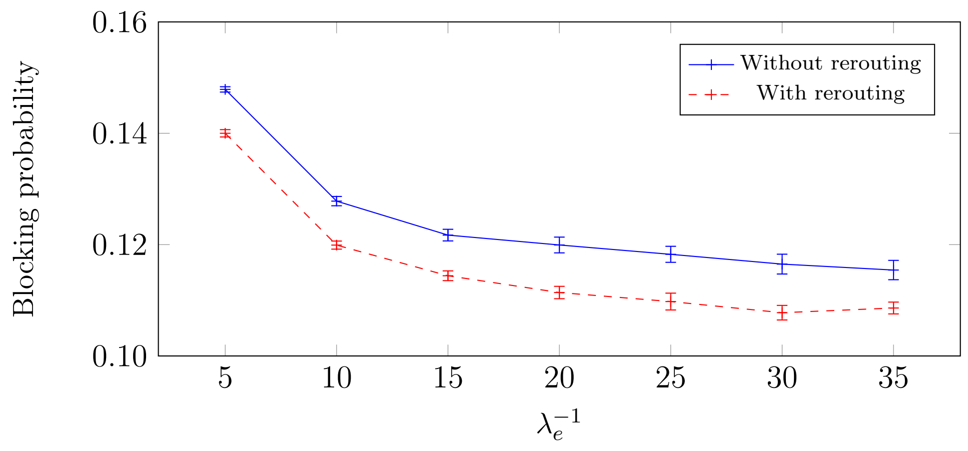


Figure 3.9.: Influence of the holding time on the blocking probability. Algorithm run at 127 Erlangs, $c = 7$ and $\frac{1}{4}\lambda_e$ rounds between two subsequent cleanups. Greedy routing configured to use least loaded paths first.

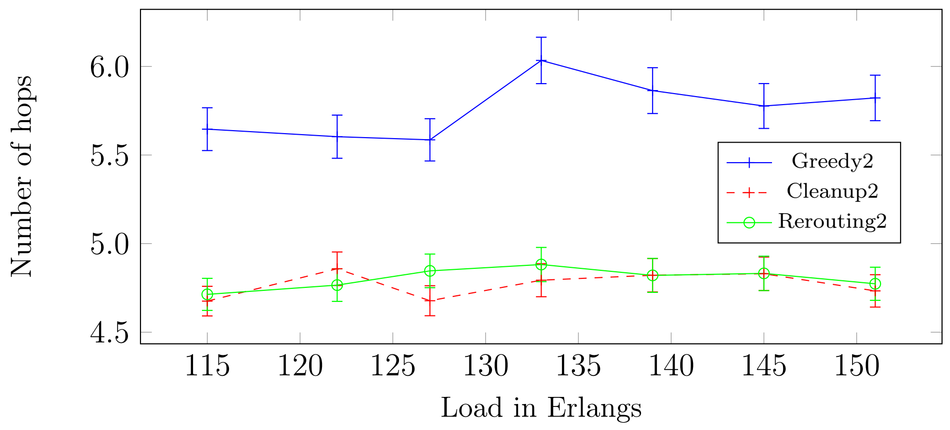


Figure 3.10.: Average hop count of established lightpaths. Algorithm configurations can be seen in Table 3.1. Holding time 20 rounds. Error bars illustrate the confidence interval for a confidence level of 95%.

a lightpath which, in this case, serve as a reference. The next two configurations additionally use periodic cleanup, which is executed every 5 rounds. The last two configurations include the full functionality by adding the rerouting algorithm with up to 7 different candidates. For the simulations we set the average holding time λ_e^{-1} of a lightpath to 20 rounds and simulated 10,000 rounds for each of the given loads. The simulation results are plotted in Figure 3.13. As before, each experiment was repeated 20 times; error bars show confidence intervals for a confidence level of 95%.

It can be seen that with a higher load in the network the blocking probability increases and there are clear differences in the performance of the considered algorithms. The worst blocking probability is achieved by the greedy algorithms. Their results can be significantly improved by using the cleanup routines as the plots of *Cleanup1* and *Cleanup2* illustrate. Best results are achieved by *Rerouting2*, which

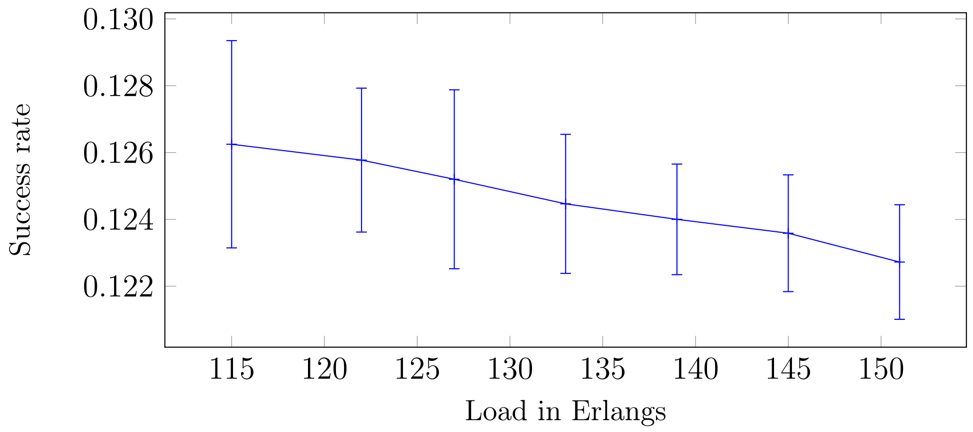


Figure 3.11.: Success rate of the rerouting operation. Algorithm run with $c = 7$, 5 rounds between two subsequent cleanups, and holding time 20 rounds. Greedy routing configured to use least loaded paths first. Error bars illustrate the confidence interval for a confidence level of 95%.

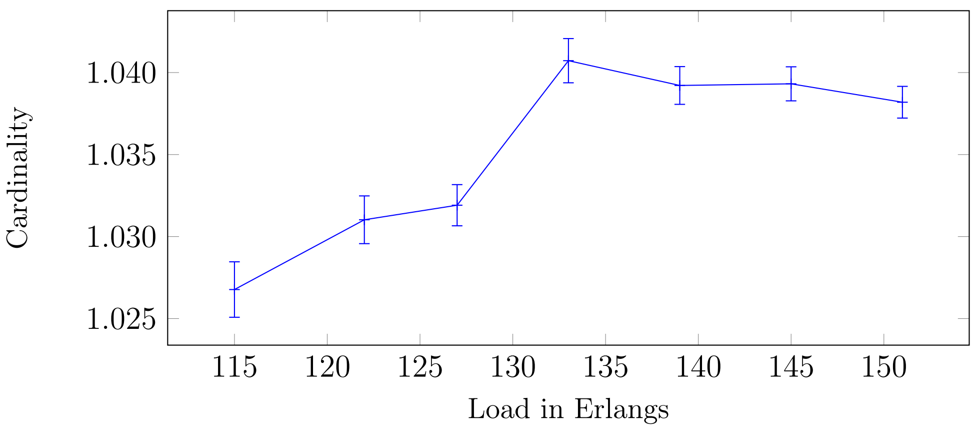


Figure 3.12.: Cardinality of each set of rerouting candidates. Algorithm run with $c = 7$, 5 rounds between two subsequent cleanups, and holding time 20 rounds. Error bars illustrate the confidence interval for a confidence level of 95%.

indicates that the rerouting feature can be used to lower the blocking probability confirming our conclusion derived from Figure 3.11.

The last column titled “Time” of Table 3.1 shows the run time of the algorithms serving one request in a scenario with mean holding time 20 rounds and 127 Erlangs load. The results where obtained on a 2.2 GHz Intel i7 QM with 8 GB of DDR3 memory. As it can be seen in Figure 3.13, the performance of *Cleanup2* and *Rerouting1* do not differ very much. But as the runtime of *Cleanup2* is about 8.5 times lower, it is recommended to prefer *Cleanup2* over *Rerouting1*.

Table 3.1.: Six different configurations of our algorithm.

Name	c	Cleanup	Path	Time [ms]
Greedy1	0	∞	shortest	2.15
Greedy2	0	∞	least loaded	2.66
Cleanup1	0	5	shortest	1.79
Cleanup2	0	5	least loaded	2.26
Rerouting1	7	5	shortest	19.30
Rerouting2	7	5	least loaded	19.96

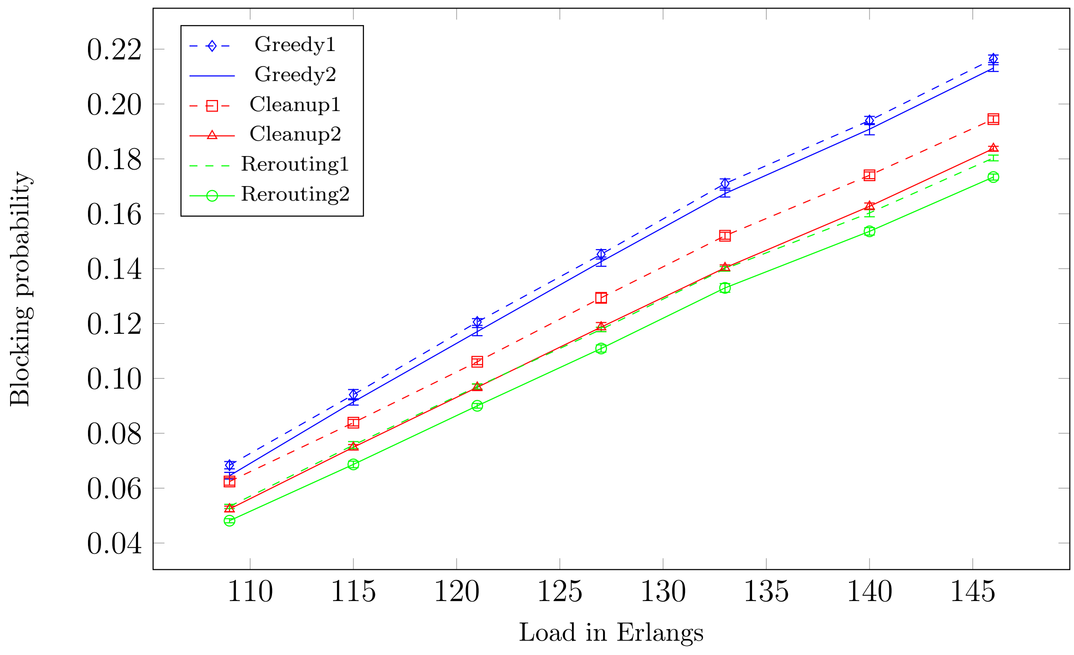


Figure 3.13.: Blocking probability of the algorithm configurations listed in Table 3.1.

3.6. Conclusion

Simulation results clearly show that the proposed algorithms can be used to lower the blocking probability in wider area backbone WDM networks. We have shown that rerouting of paths can be done at a good balance between algorithmic complexity, practical runtime for relevant example scenarios, and significant improvement of the blocking probabilities. In addition we studied the effect of periodically cleaning up the network on the resulting blocking probability in a dynamic traffic scenario: By recomputing the paths and wavelength assignments of all lightpaths periodically, the blocking probability for future lightpath requests drops significantly.

The RWA algorithm presented in this chapter is a step towards an interactive relationship between optical networks and the applications running on top of these networks. Using the findings of this chapter, the next chapter shows how to design a Virtual Topology Design algorithm that dynamically adapts the logical topology to the needs of the overlaying application, leading to higher network utilization and possibly improved quality of experience of the applications.

4 Selfish Virtual Topology Reconfiguration in IP-over-WDM

The process of planning a virtual topology for a Wavelength Devision Multiplexing (WDM) network is called Virtual Topology Design (VTD). The goal of VTD is to find a virtual topology that supports forwarding the expected traffic without congestion. In networks with fluctuating, high traffic demands, it can happen that no single topology fits all changing traffic demands occurring over a longer time. Thus, during operation, the virtual topology has to be reconfigured. Since modern networks tend to be large, VTD algorithms have to scale well with increasing network size, requiring distributed algorithms. Existing distributed VTD algorithms, however, react too slowly to congestion for the real-time reconfiguration of large networks.

We propose Selfish Virtual Topology Reconfiguration (SVTR) as a new algorithm for distributed VTD. It combines reconfiguring the virtual topology and routing IP traffic using a Software-Defined Network. SVTR is used for online, on-the-fly network reconfiguration. Its integrated routing and WDM reconfiguration keeps connection disruption due to network reconfiguration to a minimum. SVTR reacts very quickly to traffic pattern changes. Using application-layer knowledge, SVTR iteratively adapts the virtual topology of the optical network to the traffic patterns on IP layer without global traffic information and without future traffic estimations. The SDN is used to quickly adapt routing on IP layer to the topology changes of the WDM network. To this end, the SDN controller is notified when a circuit is to be removed from the network to route traffic off that circuit prior to deletion. Whenever a new circuit is established, the SDN quickly considers the new circuit for routing. We evaluated SVTR by simulation and found that it significantly lowers congestion in realistic networks and high load scenarios.

4.1. Introduction

Planning the virtual topology of an optical WDM network is a multilayer optimization problem called *Virtual Topology Design* (VTD). Depending on the traffic demands of the applications using the network, an appropriate virtual topology has to be found that is compatible with the physical topology of the network and allows routing of the traffic without congestion. As already mentioned in Section 2.1.3, VTD is an NP-hard problem commonly solved by using heuristics. These heuristics usually work offline using full knowledge about both the expected traffic patterns and the physical topology. The outcome of such a heuristic is a virtual topology and a routing of the traffic on that topology.

When using centralized algorithms to perform VTD, the global network state has to be gathered in one node. Since the network state contains the traffic matrix, it grows quadratically in the size of the network and fluctuates quickly. This makes it hard to acquire up-to-date information in real time. With increasing network size, centralized VTD algorithms cannot react to traffic changes quickly enough for fast network reconfiguration. Since this leads to worse resource usage, more network resources (i.e., ROADMs and fibers) are required, leading to both higher OPEX and CAPEX.

This chapter presents a distributed Virtual Topology Design algorithm for IP-over-WDM networks that can be used for iterative network reconfiguration with no need to determine global traffic demands. We call our approach *Selfish Virtual Topology Reconfiguration* (SVTR). In SVTR, switches selfishly manipulate the existing virtual topology of the WDM network using knowledge from the IP routing layer. Through extensive simulation with traffic demands collected from a real network, we show that SVTR reacts quickly to traffic pattern changes and that the achieved throughput is superior to both a well studied centralized VTD algorithm [LLP$^+$02] and a distributed VTD algorithm [SOI$^+$03].

Our algorithm differs from existing distributed VTD algorithms in the following ways: a) we have no need to gather global traffic information, b) we need neither synchronization nor election processes making the distributed application scalable, c) our approach does not require every node to have information about optical resource availability and d) we include backplane capacity limits of switches in our reconfiguration decisions.

The rest of this chapter is structured as follows: Section 4.2 discuses related work in Virtual Topology Planning. In Section 4.3 our assumed network model consisting of WDM equipment and SDN switches is presented. In Section 4.4 the Selfish Virtual Topology Reconfiguration paradigm is presented which is evaluated in Section 4.5. Section 4.6 concludes this chapter.

4.2. Related Work

Existing approaches for planning virtual topologies for IP-over-WDM networks can be classified according to several properties. These include using knowledge about future traffic demands, being distributed or centralized, the reconfiguration triggering method, and by proactively or reactively reconfiguring the network [Wu11]. In addition, there are approaches planning a topology from scratch and approaches which iteratively reconfigure a given topology into a target topology.

Reference [LLP$^+$02] presents three fundamental, centralized VTD algorithms that assume future traffic to be known. These algorithms build a topology from scratch and work by ranking each pair of nodes according to different metrics. The used metrics are all based on either the total traffic demands or the residual traffic demands of the individual nodes. Based on the ranking, lightpaths are created between pairs of nodes. Inspired by these algorithms, we adapted the use of residual demands for our distributed algorithm.

Distributed VTD algorithms spread the logic used to compute reconfigurations among the participants of the network (i.e., the switches) and promise faster decisions and better scalability than centralized algorithms. The distributed algorithm designed by Shiomoto et al. [SOI$^+$03] works by distributing traffic information to all nodes. Based on this information each node computes a new virtual topology and reconfigures the WDM network appropriately. A major drawback when using this algorithm in practice is that it requires all nodes to have the exact same traffic information, which is far from being trivial to achieve in real time. Nevertheless, among the few available distributed VTD algorithms, it is the one with most realistic requirements and–from our point of view–the only one which could be applied in practice. Unfortunately, the network assumptions used for the simulation in its original paper were too strong (uniform traffic distribution). We simulated the algorithm with our realistic network model and found out that the algorithm yields poor results under these assumptions.

Reference [HL06] introduces a distributed algorithm for connection rerouting at sub-wavelength granularity for WDM networks. Depending on the routing, lightpaths are added/removed from the virtual topology. Unfortunately, the rerouting calculations require all nodes to have global information about the routing of all sub-wavelength granularity flows. Obviously, this approach is not applicable even to medium-scale networks.

The distributed algorithm introduced in [WSQ$^+$03] aims at grooming SONET circuits to optical connections and could be adapted to IP-over-WDM networks. In contrast to [SOI$^+$03], not every node requires traffic information from other nodes. Nevertheless, all the nodes require information about the status of all WDM-related hardware and information about *all* established lightpaths. Additionally, for each reconfiguration of the network, all nodes have to participate in a decision process. Since this does not scale with increasing network size, it prevents the algorithm from reconfiguring networks quickly.

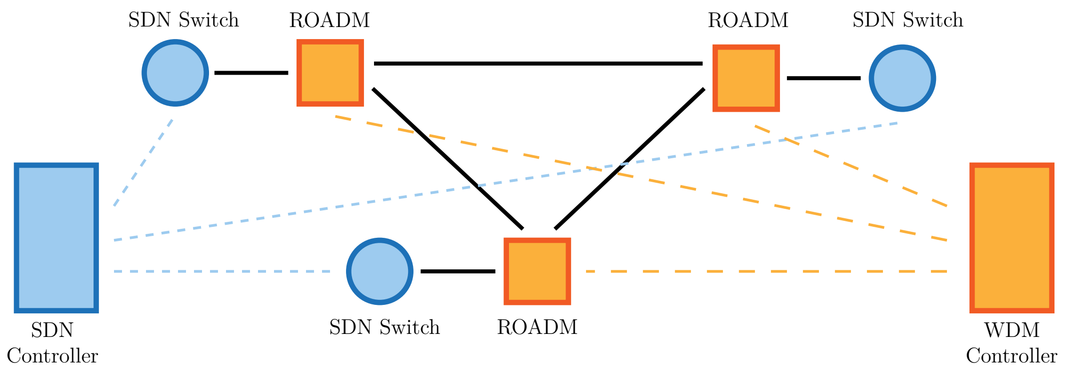


Figure 4.1.: SDN-over-WDM Network. Solid lines represent optical fibers. ROADMs are controlled by a WDM controller and SDN switches are controlled by an SDN controller.

4.3. Software Defined IP-over-WDM Networks

4.3.1. Network Architecture

We assume an "SDN-over-WDM network" as the future backbone network architecture; the SDN is used to route IP traffic over the virtual topology of a WDM network (this means—in contrast to MPLS—there is *no* additional IP routing happening on top of the SDN routing). Figure 4.1 shows an example SDN-over-WDM network consisting of 3 ROADMs and 3 SDN switches as well as a WDM controller and an SDN controller. The WDM controller decides wavelength conversion and lightpath switching at the ROADMs. Each ROADM has an SDN switch attached. These switches are controlled by an SDN controller. Based on the gathered information on the network (i.e., connected subnets), the SDN controller takes over IP routing and controls the switches appropriately. Depending on the lightpath configuration of the network, the switches see different network topologies while being attached to a static optical network. Thus, by thoughtful manipulation of the WDM lightpath configuration, the virtual topology can be adapted to meet traffic demands without changing any *physical* fiber.

4.3.2. WDM / SDN Assumptions

For the WDM network, we assume that every optical fiber is able to carry W wavelengths and that ROADMs have both heterogenous wavelength conversion and light switching capabilities. We do *not* impose the wavelength continuity constraint.

Switches are SDN enabled; each one offers a fixed number of physical interfaces. Due to hardware limitations, each switch can switch traffic only up to a fixed data rate called *backplane capacity*. In consequence, not all physical interfaces of a switch can be operated at full data rate in parallel. In addition to limited backplane capacity, there are strict limitations on the total number of flow entries (flow entries identify different traffic flows and specify where each flow is forwarded to). Due to this limitation a switch is able to handle only a fixed number of different flows at a time. The more flows it should handle, the more expensive it is.

4.3.3. Routing and Rerouting on IP level

Routing should maximize throughput while minimizing congestion. In SDN, whenever a switch receives a packet without having a matching flow entry, a matching route is calculated and pushed to all switches along this route. For route calculation we use a *Constrained Shortest Path First* (CSPF) routing algorithm [KB10] that takes the residual bandwidth capacities of each link and the residual switching capacities of the switches into account. This is possible since in addition to global information about the virtual topology of the network, the SDN controller collects information about each single switch and about the data rates on each link. To protect against traffic oscillations due to CSPF, the route of a flow does *not* adapt when residual link *or* backplane capacities change.

IP packets are routed over the VT of the WDM network; edges of the VT correspond to active lightpaths. Since these lightpaths can be removed, edges in the VT are deleted and all flows passing these edges are disrupted. To circumvent this, before a lightpath is removed, all affected flows are rerouted over other lightpaths. To implement such a rerouting feature, the SDN controller has to keep track of all active flows. This is simple to do since the SDN controller is informed whenever a new flow is created and when an old flow expires. When a lightpath is to be removed, first for each flow routed over this lightpath a new route is calculated by CSPF. Then, the network is updated by pushing the newly calculated flow entries to the switches. By doing so, the affected flows are seamlessly migrated to other routes. Since after updating the network, the lightpath to be deleted does not carry any more flows, it can be deleted without affecting the IP traffic.

To implement rerouting, the SDN controller needs an interface for receiving rerouting requests. Note that rerouting without introducing congestion need not always be possible. This is why the interface should be able to be used to *test* whether or not it is possible to reroute flows without creating congestion. With the interface, the influence of deleting a lightpath on congestion can be determined *before* the actual network reconfiguration is triggered.

4.3.4. Network Reconfiguration in WDM Networks

Whenever a new lightpath has to be created, wavelengths on a path through the physical topology have to be allocated. For this task we use the preemptive RWA algorithm introduced in Chapter 3.

Network reconfigurations are triggered by the SDN switches by querying the WDM controller for a new edge in the virtual topology. The WDM controller then runs the preemptive RWA algorithm to create a corresponding lightpath. If, to create the new lightpath, a set L of existing lightpaths has to be preempted, first all traffic routed over lightpahts in L has to be rerouted over other lightpaths because otherwise already established connections would be terminated.

4.4. Selfish Virtual Topology Reconfiguration

4.4.1. Overview

In SVTR, each switch tries to *selfishly* reconfigure the network to meet its own traffic demands. To this end, each switch has a residual demand vector, indicating the amount of traffic originating in this switch that is destined for another switch but not routed so far.

Let $D^t_{i,j}$ be the rate at which switch i *wants* to send data to switch j at time t. Let $T^t_{i,j}$ be the rate at which switch i *actually sends* data to switch j at time t. The residual traffic demand R^t for time t is defined as $R^t = D^t - T^t$. Based on this matrix R^t, the SDN switches request new lightpaths at the WDM controller using the interface developed in the last chapter (Figure 3.1). The individual switches know only their own residual demand vector (i.e., its row in the matrix) but *not* the whole matrix. Once a new lightpath is established or an existing lightpath is to be deleted, the WDM controller notifies the SDN controller to adapt the IP routing accordingly.

4.4.2. Rules

Selfish Virtual Topology Reconfiguration is distributed over the SDN switches. It is based on three simple rules: *Residual Demand Direct Link Establishment*, *Backplane Capacity Load Relaxation* and *Idle Lightpath Removal.* All three rules are executed *periodically* by each switch. We do *not* require synchronicity between switches.

Residual Demand Direct Link Establishment tries to lower the residual demands by establishing lightpaths to all switches with *positive* residual demands. Therefore, if at time t there are two switches u and v such that $R^t_{u,v} > 0$, then u requests a new lightpath from u to v by querying the WDM controller. After a lightpath is created, the corresponding traffic from u to v is routed and $R^{t+1}_{u,v}$ is lowered. In case existing lightpaths need to be preempted to create the new one, first all traffic passing these lightpaths is rerouted to other paths.

Backplane Capacity Load Relaxation (BCLR) is used to take load off the switches. Since switches have restricted backplane capacities, a selfish switch is interested in not being overloaded. We define a switch to be overloaded if its backplane is used by more than 90 %. To this end, each switch tries to minimize traffic it has to forward. This is done by creating lightpaths connecting two adjacent switches. Let $\mathrm{nei}(u)$ be the set of neighboring switches of u. If u is overloaded, it selects two

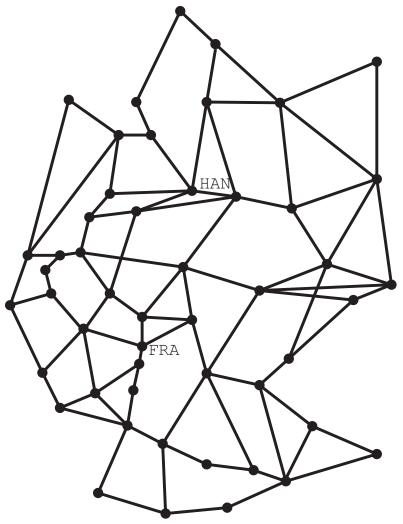


Figure 4.2.: Germany50 network with 50 nodes and 88 links.

switches $v, w \in \text{nei}(u)$ such that the amount of traffic u forwards from v to w is maximized. Then, u requests a lightpath from v to w from the WDM controller. If lightpath creation was successful, the SDN controller reroutes all traffic passing the (sub-)path (u, v, w) on the newly created link (u, w). After that, the load on the backplane of v is lowered and all rerouted flows experience a lower latency due to the reduced hop count.

Idle Lightpath Removal is used to free unused resources. We say a lightpath is underutilized if it carries data at less than a constant fraction T_L of its capacity. Whenever a switch detects that a lightpath is underutilized, the switch queries the SDN controller to reroute all corresponding flows over other lightpaths. If rerouting succeeds, the lightpath is removed.

4.5. Simulation

4.5.1. Network

For our simulations we use the germany50 network, depicted in Figure 4.2. The network has a structure comparable to the German National Research and Education Network (DFN) [ZIB]. We assume that every fiber provides 16 wavelengths, each operating at 10 Gbps, and that WDM equipment has full wavelength conversion capabilities and is able to switch optical signals from every input port to every output port. We do not use the real DFN network topology because the actual topology and traffic information in the DFN network are confidential. Thus, we use traffic information measured in the DFN network that were mapped to the germany50 network [ZIB].

The core IP routers present in 2010 at most locations of the DFN network are

Cisco 7600 Series routers which we assume to switch 360 Gbps. The locations at Hannover (HAN) and Frankfurt (FRA) are equipped with Cisco CRS-1 systems offering 1200 Gbps switching capacity. Since we use traffic measured in the DFN network for our simulations, we adopt these values for the germany50 network.

4.5.2. Traffic Model

We use IP traffic demands measured between the years 2004 and 2005 in the DFN network. From these measurements, the average traffic demand over a 24-hour period was calculated and afterwards mapped to the germany50 network.[3] For each 5 minute period, we have access to one traffic demand matrix. To create different load levels we multiply the traffic demand matrices with a scalar ϱ.

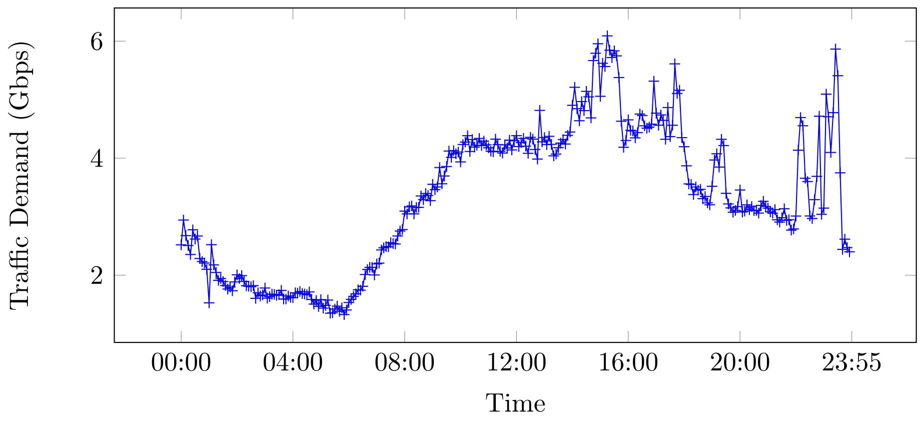


Figure 4.3.: Total traffic demand over a 24-hour period.

Amongst others, the DFN network connects various German universities with the internet. Since Frankfurt is DFN's main peering point with the rest of the internet, it is the bottleneck when scaling traffic. We do not have any information about the further destination of the traffic once it reaches Frankfurt. Thus, we cannot distinguish between traffic that is destined to a host in Frankfurt and traffic that is routed off the network. To get traffic demands for *intra-network traffic* only, we decided not to consider the traffic from and to Frankfurt in our simulation. The corresponding cumulative intra-network traffic is depicted in Figure 4.3.

Routing on IP layer is flow-based but since we only know the traffic matrices we do not have information about individual traffic flows. To divide traffic into independent flows, we assume that the lifetime of a flow is Pareto-distributed with $\alpha = 0.95$ [Cha00] and that all flows have the same natural demand $\gamma = 1\,Mbps$. The natural demand is the maximum data rate the flow will get even if it does not pass any bottleneck link.

To create flows between a pair of nodes u and v, we first compute a schedule defining which flow starts at which time. To this end, we compute the total amount

[3]We only have access to the averaged data.

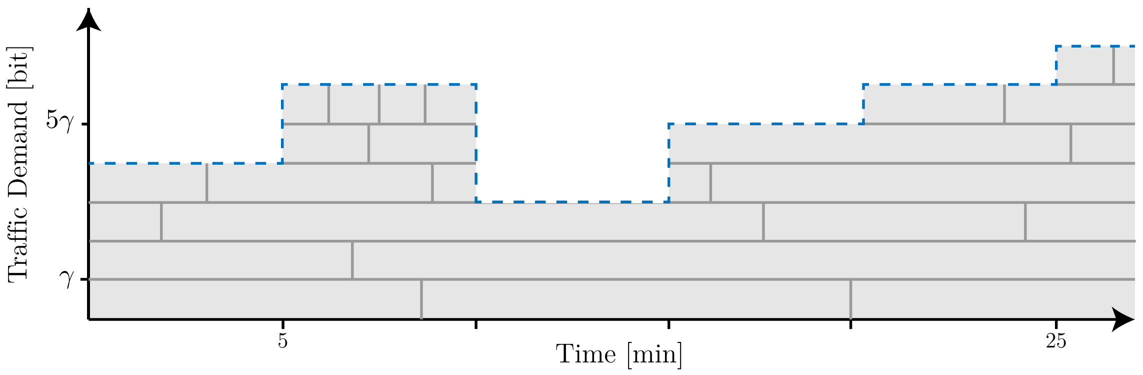


Figure 4.4.: Exemplary flow schedule for a pair of nodes. The dashed line denotes the total traffic between the nodes due to the traffic matrices. The gray blocks are the flows transporting the traffic.

of traffic between the two nodes over the whole 24-hour period. Then, we generate a set of flows matching this amount of traffic. Flows have Pareto-distributed lifetimes and each flow has a natural demand of γ. Next, we assign start times to flows. Start times have to be assigned such that in each 5 minute period, the traffic created by the flows exactly matches the traffic specified by the traffic matrix for that 5 minute period. We call such an allocation *exact*. Note, that such an allocation must not always exist.

We assign start times to the flows, one after another, using a first-fit algorithm. An example for such an allocation can be seen in Figure 4.4. In the figure, the dashed line depicts the traffic between a pair of nodes according to the traffic matrices. Each gray block depicts one flow between that pair of nodes. The depicted allocation is exact. However, even if such an allocation exists, the first-fit algorithm might not always find an exact solution. In that case, not all flows could be assigned and there are periods left which have less traffic (due to the assigned flows) than specified by the traffic matrices. Unassigned flows are afterwards assigned randomly to those periods. This introduces some error. For a discussion on the quality of such flow assignments, see Chapter 8.

4.5.3. Method

We compare SVTR to both standard textbook algorithms *Residual Demand Hop-Count Product* (RDHP) heuristic [LLP$^+$02] and the distributed algorithm by *Shiomoto et. al* [SOI$^+$03]. RDHP is a well-studied VTD algorithm that creates lightpath topologies from scratch. The algorithm works by first creating lightpaths forming a spanning tree over the switches. Then, the algorithm iterates through the following three steps: a) route as much IP traffic as possible over the network without creating congestion; b) compute the residual demands for each pair of switches and multiply it with their hop count (over the shortest interconnecting path) on IP level; c) in descending order of the residual demand-hop count product, try to establish lightpaths between switches; once a lightpath is established, continue at a); d) after no more lightpaths can be established, spend the free optical resources by establishing lightpaths randomly.

We also compared SVTR to the distributed algorithm designed by *Shiomoto et al.* [SOI+03]. Unfortunately, the algorithm yields poorly designed VTs under the assumptions used here; see Figure 4.5. Due to this shortcoming, the algorithm is left out in the further evaluation.

We measure the achieved *throughput* as the ratio of demanded traffic (as given by the flow assignment) to routed traffic (in our simulation). In the simulation, a flow is only routed, if the CSPF algorithm (Section 4.3.3) was able to find a non-congested path between the flows source and destination. Otherwise, it is periodically checked if a free path for that flow exists and the start of the flow is postponed until then. Different load levels are created by multiplying the traffic matrix by a scalar $\varrho \in \{100, ..., 800\}$.

To obtain the best possible results, for each traffic matrix, the rule set of SVTR has to be executed until the algorithm has converged, i.e., no additional flows can be established. A discussion on the influence of the number of iterations on the achieved quality is given in Section 4.5.4. For the *Idle Lightpath Removal* rule, we set T_L to 0.01 in our simulations, only removing lightpaths that are very close to idle.

4.5.4. Results

Figure 4.5 shows the achieved throughput over load levels. When scaling the original traffic by $\varrho = 800$, SVTR creates about 50% less congestion than RDHP. With increasing load the throughput gap between both algorithms increases. As in all scenarios SVTR generates a higher throughput than RDHP, it can be concluded that SVTR is able to react to traffic pattern changes and reconfigures the network appropriately. This is a very important result since it clearly shows that the idea of Selfish Virtual Topology Reconfiguration works in a realistic scenario.

Figure 4.6 illustrates the physical interface usage of individual switches for both VTD strategies. It can be observed that in a low-load scenario, RDHP needs fewer interfaces: for $\varrho = 100$ it yields a virtual topology with a 34% smaller maximum degree. But with increasing ϱ, the maximum number of physical interfaces in RDHP increases and for $\varrho \geq 200$ the maximum number of allocated physical interfaces is over 64 for RDHP. For SVTP, the maximum degree of the virtual topology seems to be independent of the load as it is between 38 and 48 in all considered load levels.

Figure 4.7 shows the *average* number of flow entries per switch. The average flow table usage is very comparable for both strategies. Additionally, the number of flow table entries scales linearly with increasing network load. Figure 4.8 shows the *maximum* number of flow entries used at a single switch. SVTR and RDHP have a similar flow entry usage which is between 300,000 and 350,000 flow entries for the

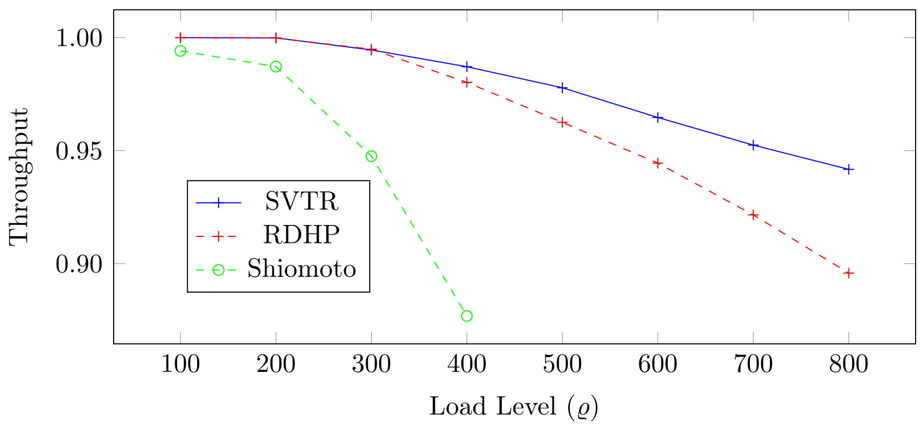


Figure 4.5.: Throughput over different load levels.

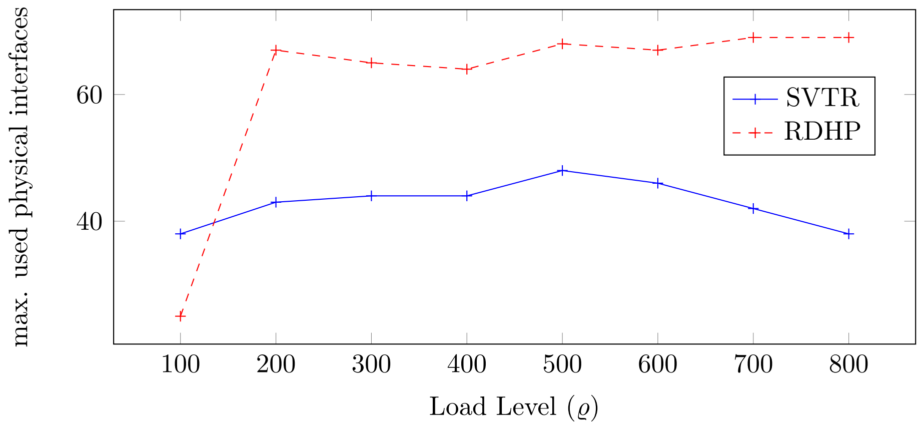


Figure 4.6.: Maximum used physical interfaces at a single switch.

considered load levels.

A frequently used argument against using SDN is that each SDN switch has a limited flow table size. If the amount of flows routed through a switch exceeds the flow table size, the performance is *massively* degraded because the switch has no possibility to hold the controller's decisions in its flow table. Figure 4.9 shows the impact of the flow table size on the achieved throughput. For both VTD algorithms, the routing algorithm (CSPF) used on IP level included the flow table size as a constraint. For the experiment, we fixed the load level to $\varrho = 800$, assigned different flow table sizes to the switches and calculated the corresponding throughput. Obviously, the smaller the flow tables, the worse the overall throughput is. SVTR is still superior to RDHP when limiting the maximum possible flows per switch.

Figure 4.10 shows the average hop count over different load levels where the hop count is the number of lightpaths a flow is routed over. The average hop count in a virtual topology designed by SVTR is—independent of the considered load level—very close to 2 hops, while for RDHP with increasing load the hop count decreases from 3.3 hops at $\varrho = 100$ to 2.2 hops at $\varrho = 700$. Thus, average end-to-end delay in

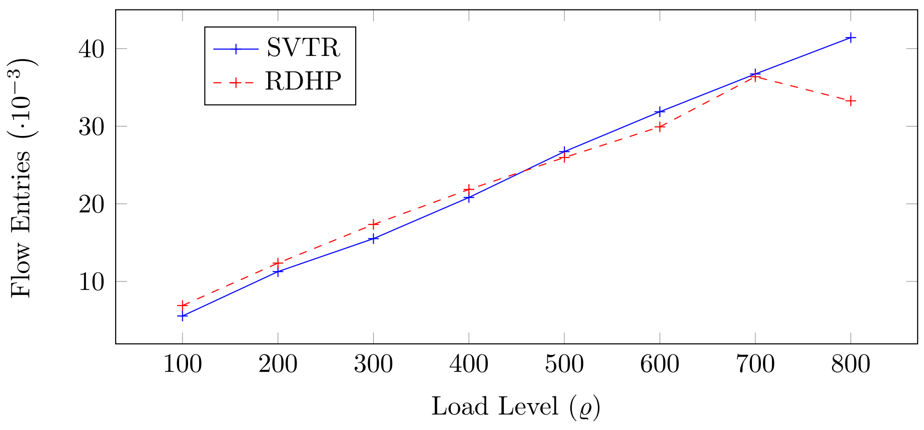


Figure 4.7.: Average flow entries per SDN switch.

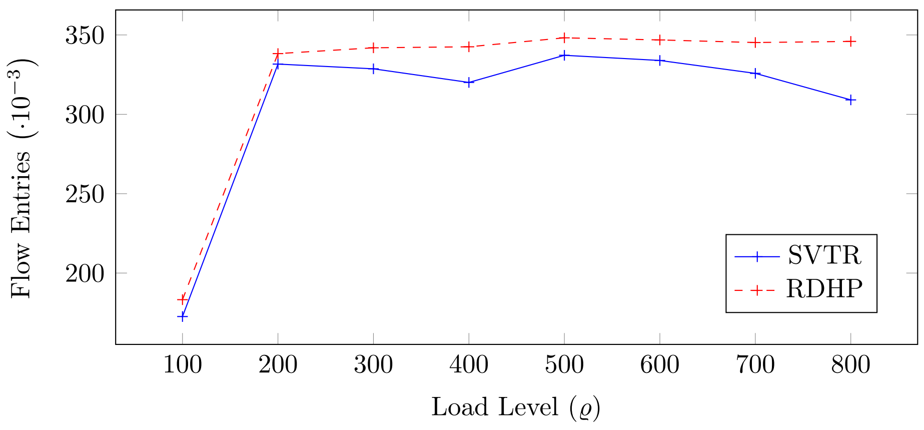


Figure 4.8.: Maximum flow entries used at a single SDN switch.

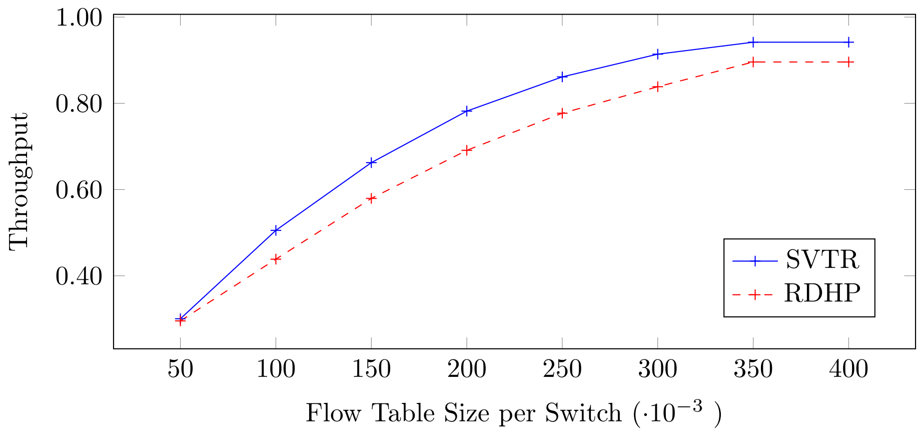


Figure 4.9.: Throughput over different flow size limits with $\varrho = 800$.

SVTR is comparable or even better than end-to-end delay in a virtual topology built by RDHP. The shape of RDHP's curve is due to the way RDHP constructs its virtual topology. Since RDHP starts with a spanning tree and creates edges between nodes with high traffic demands, nodes with low traffic demands are potentially connected

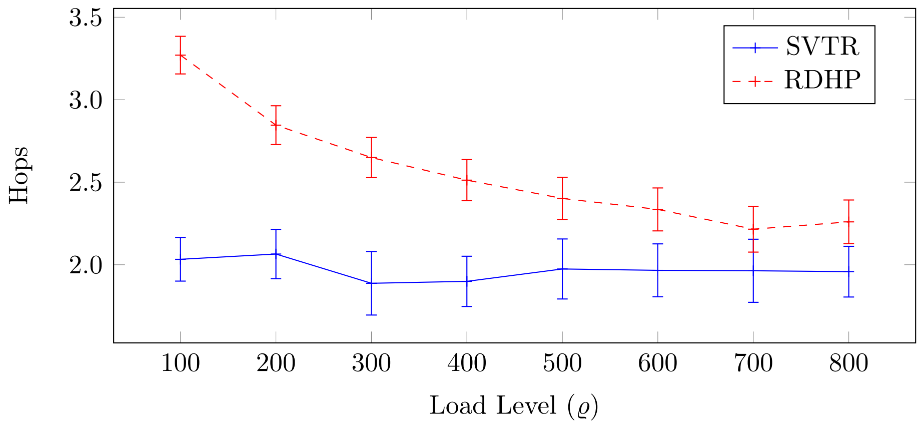


Figure 4.10.: Hop count over different load levels. Bars illustrate the confidence interval for a confidence level of 0.95.

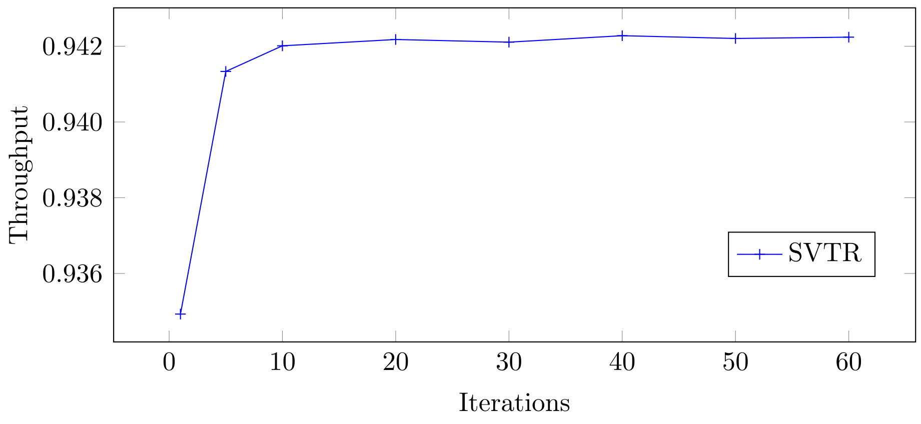


Figure 4.11.: Achieved throughput over iterations with $\varrho = 800$.

by only a few edges. This leads to a high network diameter and to a high hop count.

The applicability of an iterative algorithm in practice strongly depends on the number of iterations necessary for the algorithm to converge. We fixed $\varrho = 800$, set $\gamma = 1\,\text{Mbps}$ and varied the number of iterations the algorithm executes in a single 5-minute period. Figure 4.11 depicts the outcome of this experiment. We can conclude that SVTR requires only a very small number of iterations to reconfigure the network to meet the experienced traffic requirements. By increasing the number of iterations from 5 to 10 in a 5-minute period, the achieved throughput increases by less than 0.1%. Even more iterations have no noteworthy impact on achieved throughput. We can conclude that SVTR is very applicable in practice where only a small number of iterations seem feasible.

We could show that under our realistic network model SVTR clearly outperforms both other algorithms in throughput while offering at least the same performance in any other metric. To the best of our knowledge, the distributed VTD algorithm stated in this chapter is the only one that is able to reconfigure the VT of large backbone networks quickly in an iterative manner.

4.6. Conclusion

This chapter showed that, when using knowledge from the IP routing layer, adaptive networks can be built that are 50% less congested than traditional networks in which the virtual topology design process only uses information about the traffic matrix.

To this end, we proposed a simple, rule-based reconfiguration system for a backbone network employing SDN for an IP-over-WDM network as a possible future internet architecture. The combination of SDN together with a reconfigurable optical network makes the network highly flexible. This flexibility creates new opportunities to support a very large set of different traffic patterns without any human interaction, leading to a high-speed, self-adaptive network. A network equipped with this technology is able to transparently reconfigure very quickly from one topology to another without noticeable performance degradation during the transformation. It has no need for multilayer optimization processes and promises a much higher utilization than present algorithms.

Part II.

Packet Switching

5 A Low-Cost, Software-Defined Data-Center Network

The size of modern data centers is constantly increasing. As it is not economic to interconnect all machines in the data center using a full-bisection-bandwidth network, techniques have to be developed to increase the efficiency of data-center networks. The Software-Defined Network paradigm opened the door for centralized traffic engineering (TE) in such environments. Up to now, there were already a number of TE proposals for SDN-controlled data centers that all work very well. However, these techniques either use many flow table entries or a high flow installation rate that overwhelms available switching hardware, or they require custom or very expensive end-of-line equipment to be usable in practice.

This chapter present HybridTE, a TE technique that uses (uncertain) information about large flows. This information can either directly be reported by the applications running in the data center or it can be predicted by monitoring the traffic. Using this extra information, our technique has very low hardware requirements at better performance than existing TE techniques. This enables us to build very low-cost, high-performance data-center networks.

5.1. Introduction

To increase the throughput of data-center networks a lot of traffic engineering (TE) techniques have been proposed. Some of them are distributed in nature, such as Equal-Cost Multi-Path (ECMP), and some of them are using the Software-Defined

Network (SDN) paradigm, to make fine-grained routing decisions on a flow-based level by using global information.

Recent studies of data-center traffic [BAM10, KSG+09] revealed that the traffic in data centers is very diverse. 80 % of all flows are shorter than 10 KB, whereas more than half of all transported bytes reside in only a few flows. These flows are called *elephants* (as opposed to *mice*).

In ECMP, each single flow (both elephants and mice) is randomly assigned to one of multiple shortest paths between the source and destination. There are multiple variants of ECMP. In the context of this work, with ECMP we refer to the variant where each single Layer 3 flow (identified by the five header fields source IP address, destination IP address, source port, destination port, VLAN id) is routed individually. ECMP is the de facto standard in traditional data centers and aims at evenly distributing load to all links. If, however, multiple elephants are assigned to a shared link, these elephants are competing for data rate although there might be paths with free data rate. This is where most novel TE techniques [BAAZ11, AFRR+10, CMT+11] come into play. They detect colliding elephants and reroute them based on global information about the network.

According to [BAM10, KSG+09], the average flow size in contemporary data centers is approximately 146 KB. Thus, to saturate a full duplex 10 Gbps link, it requires approximately 20,000 new flows every second. All available SDN TE techniques either require one flow entry per flow (i.e., they do not use wildcard flows) or switches with custom logic which cannot be bought off-the-shelf. Without using custom logic, a 48-port switch requires 480K flow table entries on average when these TE techniques are used. Although there exist SDN-capable switches with such large flow tables, these switches *cannot* keep up with the high flow arrival rate in data centers. NoviFlow's NoviSwitch, for example, supports up to 1 million flow entries but it is only able to process 12,000 flow installations per second, which is less than what is required to saturate even a single full duplex link.

The contributions of this chapter are as follows: First, HybridTE is proposed, a novel TE technique for SDN-controlled data-center networks that uses information about elephant flows. In turn it only requires very few flow entries in the switches and has a very low flow installation rate. HybridTE allows us to build high-performance data-center networks using low-cost off-the-shelf SDN switches that need only limited space and processing capabilities without loosing network performance. We show the performance of HybridTE using a MaxiNet-based network emulation with realistic data-center traffic and compare our results to traditional ECMP and to Hedera [AFRR+10]. The necessary tools required for the emulation, i.e., MaxiNet and the DCT^{2}Gen traffic generator, are presented in detail in the next part of this thesis. The key idea of HybridTE is to handle mice differently than elephants. To this end, we require explicit knowledge about elephants which can be created either by leveraging application-layer knowledge or by elephant-detection techniques [LWPB07, CKY11].

Different elephant-detection techniques create different qualities of information. In a hypothetical ideal case, elephant detection reports all existing elephants without any delays. In reality, however, elephants will be detected some time after they start and with some amount of false positives and false negatives.

We show how the different levels of false positives, false negatives and delay impact the quality of our traffic engineering scheme. This makes this work the first to show the relation between the amount of explicit information and the performance gained by this information in a realistic data-center environment. Data shows that even with 50 % false negatives and a reporting delay of 1 s, the performance of HybridTE is comparable with Hedera and ECMP. Also, we could show that HybridTE withstands up to 95% false positives without suffering from performance degradation.

The rest of this chapter is structured as follows: Section 5.2 gives an overview of related work. Section 5.3 motivates the problem addressed by this paper and introduces its solution: HybridTE, a novel routing algorithm for software-defined data-center networks. This algorithm is evaluated under realistic traffic in Section 5.4. Section 5.5 concludes this chapter.

5.2. Related Work

To speed up applications running in the data center, various techniques have been proposed recently. The research area most related to HybridTE is *traffic engineering* (TE). TE focuses on the routing of individual flows. In the traditional sense, it works on time scales of hours to days. Using SDN, however, traffic engineering can be used on a second or even sub-second scale.

Hedera [AFRR+10] was one of the first TE schemes proposed for the SDN protocol OpenFlow. Hedera uses a centralized scheduler to dynamically compute new routes for elephant flows. To identify elephant flows, the scheduler periodically polls data about all installed flows from all switches. Unfortunately, the centralized control of Hedera does not keep up with the high flow arrival rate of real data centers.

Just like Hedera, DevoFlow [CMT+11] uses a centralized scheduler to do TE. To remove load from this controller, DevoFlow uses custom switching hardware and a modified version of the OpenFlow protocol. The evaluation with realistic data-center traffic, however, could not show any significant performance gain over ECMP.

MicroTE [BAAZ11] uses short-term predictions of the traffic matrix to perform TE in data centers. Evaluation results show that MicroTE performs well under realistic traffic assumptions. To predict traffic, MicroTE relies on modified end hosts to collect server-to-server traffic matrices for sub-second time scales. As this does not scale with increasing number of servers, MicroTE is able to fall back to rack-to-rack traffic matrices. However, evaluation shows that in rack-to-rack mode MictoTE does not outperform ECMP.

CONGA [AED+14] is a fully distributed congestion-aware load balancer for data centers which requires hardware support. It has been implemented in custom ASICs and will be available soon in Cicso's top-of-the-line data-center switches.

As already stated in the introduction, we want to leverage low-cost off-the-shelf SDN switches for building data centers. Therefore, HybridTE is designed for low resource usage and compatibility with the OpenFlow standard. To the best of our knowledge, none of the existing techniques complies with these requirements.

5.3. HybridTE

5.3.1. Problem

Efficiently handling data-center traffic using SDN is complicated due to the limited scalability of the centralized SDN approach. The traffic patterns in data centers are changing very frequently [BAAZ11] and traffic mostly consists of very small flows ($90\,\% < 1\,\mathrm{MByte}$), whereas most bytes are transported by a minority of elephant flows. The aggregated flow arrival rate is too high to be handled in a centralized manner and most of the decisions (i.e., for mice) have only very little influence at all on the overall load situation.

Thus, when using SDN in the data center, it does not pay off to make individual routing decisions for each single flow. Even if we did, it would lead to performance penalties. 10 KB sent over a 10 Gpbs line takes $1\,\mu$s, which means that even if decision making only takes an additional $1\,\mu$s, the flow finishing time is doubled.

For a TE scheme to be usable in practice, the requirements on the switching hardware should be as low as possible which is why HybridTE uses only few wildcarded and exact-match flow-table entries while having a very low flow installation rate.

5.3.2. Idea

To a) remove load from the central controller and b) reduce delay in handling mice flows, HybridTE handles mice flows locally at the switches by providing proactively installed static routes. When handling elephant flows using local decisions as well, this leads to avoidable congestion whenever two or more elephants are assigned to the same link while alternative routes with free capacity exist that could have been chosen using global knowledge. To this end, HybridTE requires knowledge about elephant flows. Once HybridTE is informed about an arriving elephant, this flow is routed individually using global knowledge.

In this chapter, we study this question: Assuming we know which flows are elephants, is it sufficient to have a very simple proactive routing scheme when performing traffic engineering on the elephants *only*? As in reality there is no explicit knowledge about *all* elephants available, we determine what number of elephants leads to which performance level, and how many elephants we have to be aware of to create a routing with better quality than existing TE techniques.

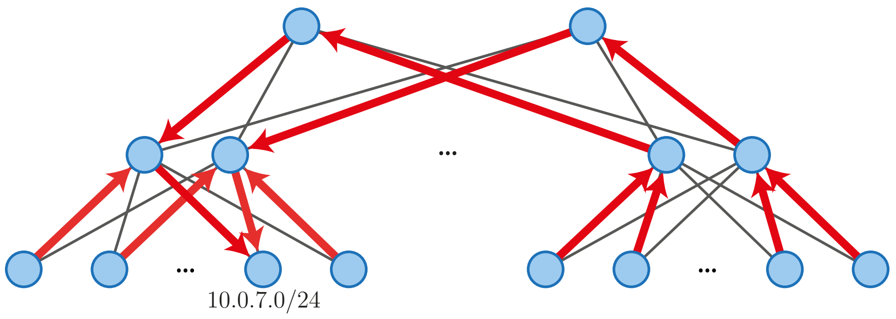


Figure 5.1.: Clos-like data-center topology with an exemplary forwarding tree for subnet 10.0.7.0/24.

5.3.3. Static Routing

The static routing component in HybridTE provides basic connectivity and aims to spread traffic evenly over all links in the network and flow table entries evenly over all switches. To provide low latency while using as few resources as possible, the scheme forwards traffic along shortest paths only. Therefore, the scheme constructs one forwarding tree per destination rack. Figure 5.1 shows a typical data-center architecture consisting of a core layer (top), a pod layer (middle) and a layer of top-of-rack (ToR) switches. Each ToR switch connects one rack of servers (typically 20 to 40 servers per rack). ToRs are grouped within pods. Each pod has two pod switches which are interconnected through the core switches. We assume that within each rack, servers use a unique IP subnet. This assumption allows us to implement one forwarding tree using one OpenFlow wildcard flow-table entry per switch only. In Section 5.3.4 we show that when using label routing techniques, HybridTE works even without this assumption.

HybridTE uses Algorithm 2 to install static routes. In the algorithm, R is the set of all racks and $\mathcal{S}_{i,j}$, $i, j \in R$, is the set of all shortest paths between racks i and j. To create a forwarding tree for rack $i \in R$, for each rack $j \in R$, $i \neq j$, one path $p \in \mathcal{S}_{i,j}$ is chosen uniformly at random. Then, on each switch s on the path p that has not yet a rule to handle traffic destined to i, one wildcard rule is installed to forward all traffic destined to i to the next switch on the path p. Note that this algorithm creates one forwarding *tree* per destination. Figure 5.1 gives an example for such a tree.

With these default routes in place, HybridTE creates basic connectivity and already spreads traffic over the whole topology. When using the OpenFlow protocol, this only requires $|R| + M$ wildcard flow entries at each ToR switch and $|R|$ entries at each core and pod switch, where M is the number of machines per rack.

Algorithm 2 Install Static Routes

```
for $(i, j) \in R \times R$ do
    choose Path $p \in \mathcal{S}_{i,j}$ uniformly at random
    for each Switch $s$ on $p$ do
        if $s$ has no entry matching destination $i$ then
            on $s$, install a wildcard entry matching all packets destined to $i$ with the
            action to forward the packet to the next switch on $p$
        end if
    end for
end for
```

5.3.4. Support for Unstructured IP Addresses

Using novel label switching techniques [SK14, ADRC14], the assumption about structured IP addresses made in the previous section can be dropped while keeping the same number of OpenFlow flow table entries at each switch. This allows endhost mobility without having to change IP addresses after moving a machine from one rack to another, which is a very important feature for upcoming cloud data centers.

The core idea in [SK14] is using spoofed ARP replies to create structure in the MAC addresses. Whenever a host sends an ARP request for a host residing in rack i, the SDN controller intercepts the request and answers with a faked MAC address which uses the first bytes of the MAC address to encode the rack ID i and the last bytes to encode the endhost. In addition, on the ToR of rack i, a flow entry is installed to rewrite the faked MAC address back to the host's original MAC before delivering any packet. Whenever a host moves from one rack to another, gratuitous ARP is used to promote the new MAC address of the host. To deliver any packets from flows to a recently moved host (which are addressed to the now outdated MAC), we install a rule to rewrite the MAC accordingly at the ToR of rack i.

As the rack ID is encoded in the first bytes of the MAC addresses, the static routing component of HybridTE can use these bytes for wildcard routing instead of the IP subnet as before. Note that this does not change the number of required OpenFlow flow table entries at the switches.

5.3.5. Reactive Elephant Routing

Using the static routing scheme only, congestion is very likely to appear. Assume we have two elephant flows between the same pair of ToR switches but between distinct hosts. As we are using a forwarding tree to route traffic, both elephants are sharing the available data rate on the path while there could be another path with free residual data rate.

To shift traffic from heavily used links to unused links, we reroute elephants off the static routes onto individually computed routes. To this end, we need to know which flow becomes an elephant in the first place. As HybridTE uses wildcard flow entries to route traffic, we do not know which flows are hiding behind these rules and thus depend on external elephant detection techniques [WK13c]. In the scope of this chapter, we do not care where this information comes from; there are a lot of different possibilities to detect elephants. More details are given in Section 5.3.7.

Whenever a new elephant between ToRs i and j is reported to HybridTE, this elephant is routed individually through the network. The new route can be computed by different rerouting algorithms that can make use of all information present at the SDN controller. For simplicity, HybridTE reroutes the elephant to the shortest path p between i and j that is least loaded. Rerouting is done by installing exact matching flow entries on each switch on the new path p.

5.3.6. Periodic Elephant Rerouting

As old flows complete and new flows arrive, the load situation of the network is constantly changing. To account for this, HybridTE periodically reroutes all known active elephants. Most traffic is transported using TCP. As rerouting a TCP flow introduces packet reordering and changes the round trip time perceived by the flow, rerouting has a negative influence on the flow's data rate, which is why flows should not be rerouted too often. We decided to reroute every 5 s which is in line with what is discussed in [AFRR+10].

To receive a list of all reported but still active elephant flows, all core and pod switches are queried for their installed exact matching flow table entries. Using this information, we can compute the average data rate of each elephant and the path over which each elephant is routed.[4] As we are only installing elephant flows as exact matching flows and only a few elephants exist concurrently, this will not become a performance bottleneck. Next, all switches are queried for their number of bytes transferred over each physical switch port. By record keeping, HybridTE computes the average data rates of all links over the last 5 seconds. By subtracting the average data rates of all elephants from the link data rates we can express the data rate on the links consumed by mice only; this can be seen as the background traffic for rerouting elephants.

Finding an optimal routing for the active elephants is an $\mathcal{NP}$-hard problem. Just like Hedera, we use the *Global First Fit* [AFRR+10] heuristic to solve it. Global First Fit starts by computing the *natural demand* of all elephant flows. The natural demand is the data rate the flow would achieve when the only bottleneck in the network was the host's network interfaces and all the rest of the network is non-blocking. Based on the link data rates we computed before, Global First Fit reroutes elephants by subsequently finding non-blocking paths for each elephant. For details on Global First Fit, see [AFRR+10].

[4]There are different ways to get the routes of all reported still active elephant flows. We decided for this soft-state solution because it is simpler to implement and allows for very simple and fast fail-overs in case the HybridTE controller fails.

5.3.7. Is Elephant Detection Realistic?

HybridTE leverages information about elephant flows in the network. Our evaluation shows (unsurprisingly) that with more accurate information, the performance of HybridTE increases (Section 5.4.2). But to which extent is information about elephant flows available in typical data centers?

There are two different types of elephant detectors: the invasive type and the noninvasive type. The invasive type requires changes to the software running in the data center or the operating systems on which this software is executed. With the gathered information it can be told which data transfer is an actual elephant. Imagine a file download from an HTTP server. As the server knows the file size in advance, this information can directly be passed to HybridTE. HadoopWatch [PCW$^+$14] is another example for a (minimally) invasive detector. It monitors the log files of Hadoop for upcoming file transfers. This way, the traffic demand of the Hadoop nodes can be forecast with nearly 100% accuracy and ahead of time.

The noninvasive type of elephant detectors does not require any changes to software and is much easier to deploy. Packet sampling is a noninvasive technique to identify elephants that is independent of the software running in the data center. Using middleboxes or built-in features of switches, packet samples can periodically be taken and analyzed. Choi et al. [CPZ04] showed that using a reasonable number of samples, elephants can be identified quickly with reasonable accuracy.

Different elephant detection techniques have different characteristics. HadoopWatch, for example, is able to predict elephants ahead of time, while packet sampling usually takes more time to identify a flow as an elephant. In addition, different techniques yield different numbers of false positives, i.e., mice that are labeled as elephants by mistake. On the other hand, some elephants will not be detected at all. We give more insight into the effects caused by various values of false positives, false negatives and delays in Section 5.4.2.

5.4. Evaluation

5.4.1. Emulation Environment

In contrast to most other work in the context of traffic engineering in data-center networks, we are using highly realistic traffic patterns for evaluation. Traffic is generated by DCT^{2}Gen, a novel generator that uses the findings of two studies [BAM10, KSG$^+$09] about traffic patterns in data centers. DCT^{2}Gen is explained in detail in Chapter 8.

We use MaxiNet to emulate a mid-sized data center. MaxiNet is a distributed Mininet [LHM10] version that uses multiple physical machines to emulate a network; MaxiNet is described in detail in Chapter 7. To support emulation of a 10 Gbps data-center network we rely on the concept of time dilation [GYM$^+$05], which basically means slowing down the time by a certain factor (called time dilation factor). For our experiments, we used a time dilation factor of 150, meaning each 10 Gbps link was emulated by a 66.6 Mbps link; in turn, the overall running time of each experiment increased by a factor of 150.

For evaluation, we implemented HybridTE, ECMP and Hedera on top of the OpenFlow controller Beacon [Eri13] using OpenFlow 1.0. Both the connection between the switches and the controller and the controller itself were *not* subject to time dilation, leading to nearly instant decision making. This removes any effects possibly resulting from a bad controller placement or hardware too slow to host the controller.

The physical machines used to compute the emulation are four Intel i7 960 CPUs with 24 GB of memory and five Intel i5 4690 CPUs with 16 GB of memory. The OpenFlow controller is hosted at an Intel Core 2 Duo E8400 running at 3 Ghz with 8 GB of memory. All physical machines are wired in a star topology with 1 Gbps Ethernet.

5.4.2. Experiments

To find out how HybridTE performs, we emulate 60 seconds of data-center traffic on a Clos-like, full duplex 10 Gbps network consisting of 1440 hosts organized in 72 racks. During that time, approximately six million Layer 2 flows are emulated. Each pod consists of eight racks and each pod has two pod switches. The core of the network consists of two switches. Due to the used time dilation factor of 150, one experiment takes two and a half hours walltime plus time for setting up the emulation and compressing and storing the results.

The metric used to compare the different routing algorithms is the *average flow completion time*, i.e., the average time between the first packet of a flow is sent until the last packet of that flow is received. As argued in [CZS14, DM06], the flow completion time is one of the most important metrics for data centers running multi-staged applications where one stage can only start when the prior stage is completed.

For all experiments, we defined an elephant to be a flow transmitting more than 10 MByte payload.

As already argued in Section 5.3.7, different elephant detection techniques have different properties. To account for this, we evaluate HybridTE under different numbers of false negatives, false positives and reporting delays.

False Negatives: Actual elephants not reported

To see how HybridTE performs under different rates of false negatives, we evaluated it with information about a) 100 %, b) 75 %, and c) 50 % of all elephants. In the following, we reference these cases as HybridTE_{100}, HybridTE_{75}, and HybridTE_{50}, respectively. These correspond to 0 %, 25 %, and 50 % false negatives. False negatives were drawn uniformly at random from all elephants. We compare the results

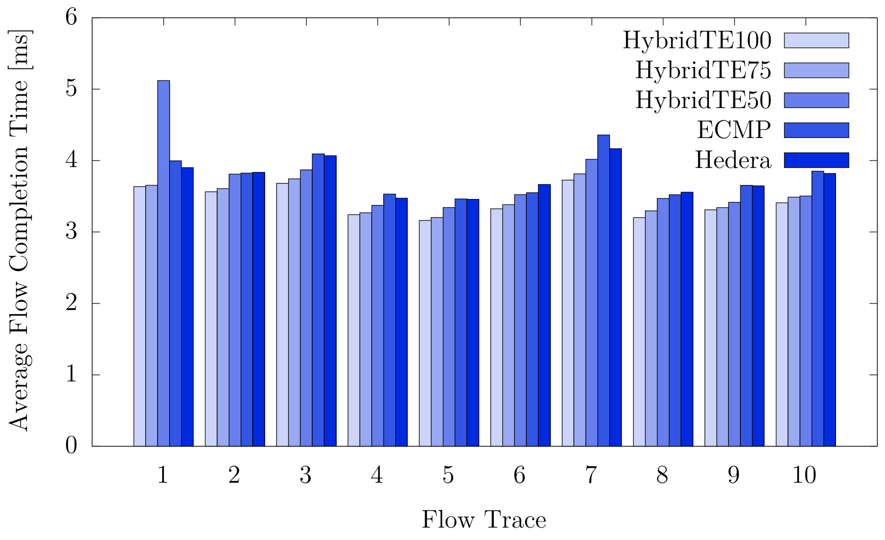


Figure 5.2.: Average flow-completion time over all ten flow traces on load level 1.

of HybridTE against ECMP and Hedera under four different load levels (×1, ×1.5, ×1.75, and ×2). A load level was created by scaling down the data rates of the emulated links while keeping the same emulated NIC speeds. We used DCT^{2}Gen to generate ten independent *flow traces* describing which host sends how much payload via TCP to which other host at which time. For each load level, we repeated the experiment with each of the ten flow traces, which resulted in 200 experiments. Including all overheads, the experiments took more than one month to complete.

As data-center traffic follows heavy-tailed distributions, the ten experiments we conducted for each configuration are not enough to compute meaningful averages and confidence intervals. This is why in the following we show the results for each run of the experiment individually. In our experiments never more than 423 elephants are existing concurrently, which can easily be handled by HybridTE running on a single machine. Due to the small number of elephant flows, the flow installation rates on the switches are very low.

Our results show that Hedera does *not* outperform ECMP in terms of the average flow completion time. This result is in line with what was stated in the original Hedera paper when considering realistic traffic. Thus, in the following discussion we omit Hedera. In contrast to HybridTE, Hedera is not informed about elephants; it classifies elephants from information gathered from the switches. We suspect that this classification does not work well with the traffic used in our evaluation. Otherwise, if Hedera had full information about all elephants, we would suspect it to perform as well as HybridTE$_{100}$. However, even then (due to its high flow installation rate), Hedera would be unusable on real hardware.

The average flow completion time in load level 1 can be seen in Figure 5.2. For each experiment, the average was computed from the approximately 3 million Layer 4

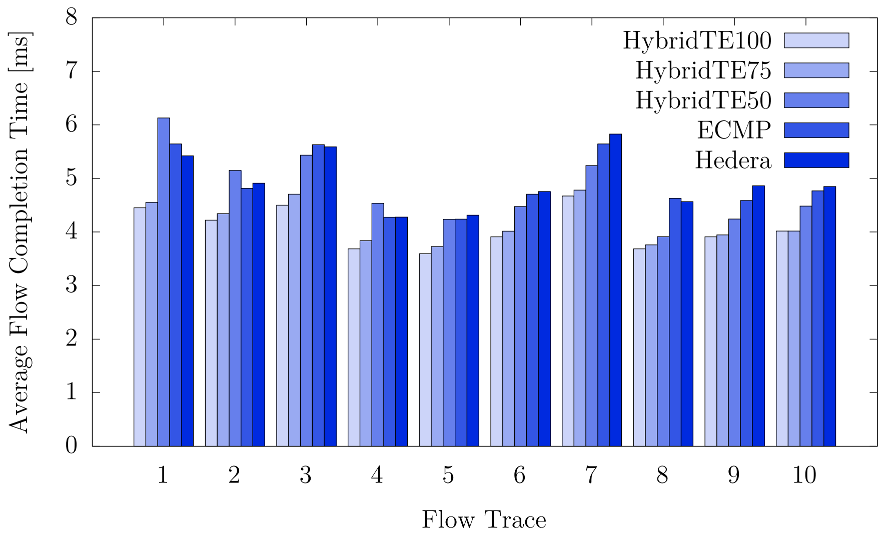


Figure 5.3.: Average flow-completion time over all ten flow traces on load level 1.5.

flows contained in each flow trace. It can clearly be seen that HybridTE achieves lower flow completion times than ECMP and Hedera. The flow completion time resulted from ECMP is on average 7% higher than for HybridTE$_{100}$. HybridTE$_{100}$ and HybridTE$_{75}$ have nearly the same performance: On average, HybridTE$_{75}$ yields 1% higher flow completion times than HybridTE$_{100}$. The difference between HybridTE$_{100}$ and HybridTE$_{50}$ is 6.2%. But still, for most flow traces the flow completion times resulting from HybridTE$_{50}$ are lower than from ECMP.

The flow completion times for load level 1.5 are depicted in Figure 5.3. In this load level the emulated network is already congested, which increases flow completion times. In turn, the results created by ECMP are getting worse: on average ECMP yields 20% higher flow completion times than HybridTE$_{100}$. We see that with increasing knowledge about elephants, the results are getting better: the gap between HybridTE$_{100}$ and HybridTE$_{50}$ increases from 6.2% (load level 1) to 14.3% (load level 1.5).

Figure 5.4 and Figure 5.5 show the results for load levels 1.75 and 2. With further increasing congestion in the network, HybridTE outperforms ECMP by larger margins. At load level 1.75, ECMP yields 28.7% longer flow completion times, and at load level 2, the gap even increases to 29.1%. At the same time the gap between HybridTE$_{50}$ and ECMP decreases; it seems that in such a heavily loaded network, information about only 50% of the elephants is not enough to provide significantly better results than ECMP.

In summary, our results show that a) with an increasing amount of information, HybridTE yields better results, and b) HybridTE clearly outperforms ECMP and Hedera in terms of flow completion time when the network is congested. Even with 50% false negatives, HybridTE still yields better results than ECMP and Hedera.

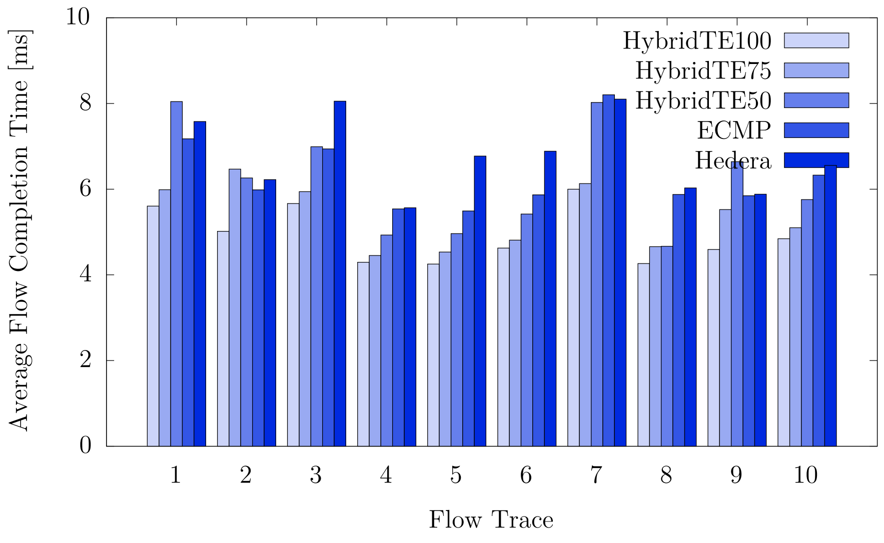


Figure 5.4.: Average flow-completion time over all ten flow traces on load level 1.75.

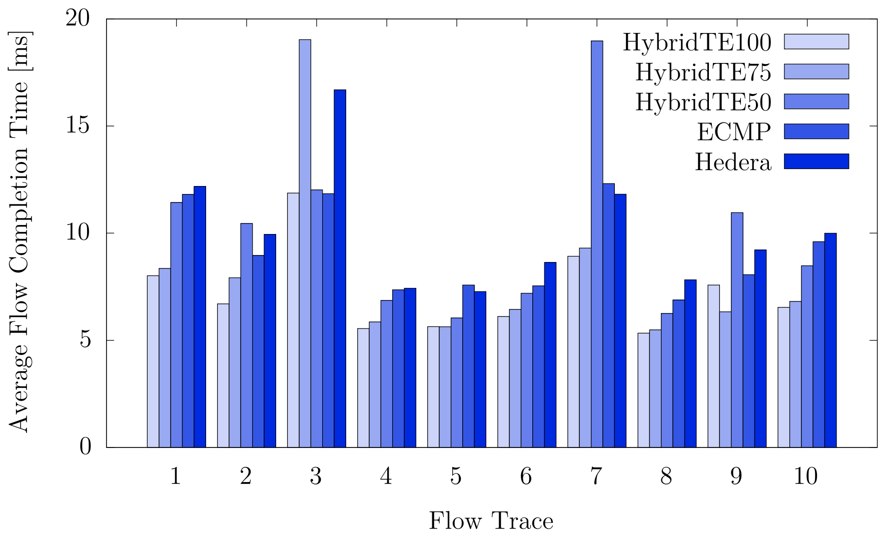


Figure 5.5.: Average flow-completion time over all ten flow traces on load level 2.

False Positives: Mice reported as elephants

As stated in Section 5.3.7, elephant detection techniques sometimes falsely report mice as elephants. Whenever a mouse is reported to HybridTE, it will be assigned an individual path through the network. As the mouse only transfers a few bytes it will either already have finished when the individual route is installed on the switches or it will use the new route and complete quickly. As a result, the newly installed route will time out quickly, too. Although handling more elephant reports will increase the workload on the controller running HybridTE, we do not see any reason for serious

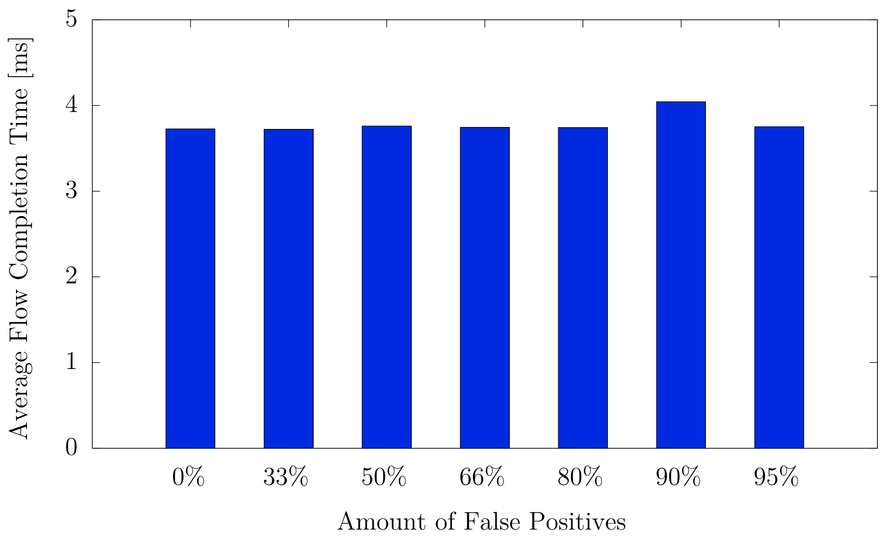


Figure 5.6.: Average flow completion time created by HybridTE$_{100}$ under different levels of false positive elephant reports on flow trace 7.

performance degradation as long as switch tables do not fill up and flow installation rates stay manageable. If, however, the rerouting phase of HybridTE starts between the report of a false positive and its timeout, then this false positive will be part of the input to the rerouting phase and possibly lead to worse rerouting decisions for the actual elephants.

To determine the impact of false positives on the quality of HybridTE$_{100}$, we conducted the following experiment: We fixed the load level to 1 and varied the number of false positives from 0 % to 95 % where false positives are drawn uniformly at random from all mice. Figure 5.6 depicts the outcome of this experiment on *flow trace 7*. We decided for this particular flow trace because it produced the most distinct results in the previous experiment (Figure 5.2). It can clearly be seen that HybridTE is resilient against false positives, which is mainly because only very few of the false positives live long enough to be rerouted in the first place. The time between the start of a mouse, its report to HybridTE and the corresponding route calculation and installation of the corresponding flow table entries at the switches takes longer than the mouse flow lives. In turn, the corresponding flow table entries will never count any packets and time out. Even if the rerouting phase starts between the end of the mouse and the timeout of its corresponding flow table entries, the mouse will still *not* be regarded by our rerouting algorithm because we filter all entries with zero packet counts out of the set of elephants before rerouting.

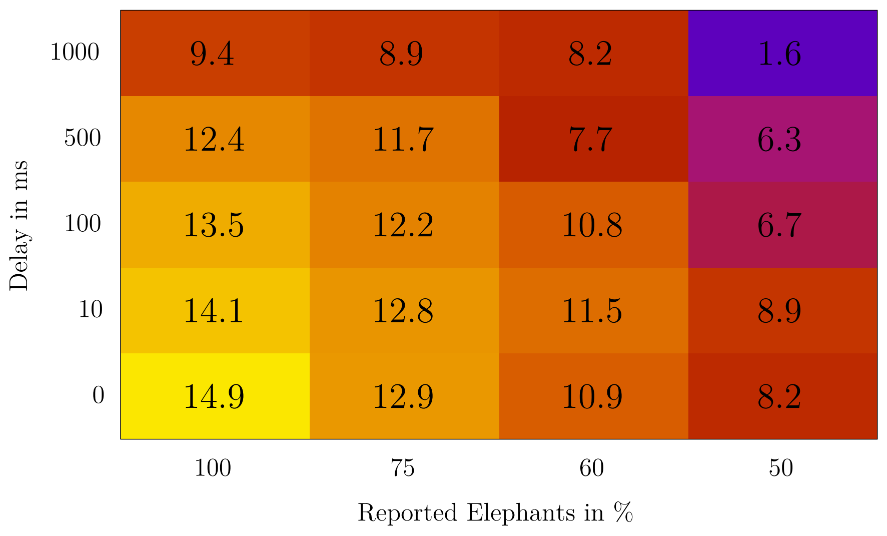


Figure 5.7.: Reduction (in percent) of the average flow completion time created by HybridTE relative to ECMP under different delays and false negatives.

Delays of Elephant Detection

In the experiments shown in Section 5.4.2, HybridTE was informed about elephant flows right when they started. In reality, however, the information that a flow is an elephant will likely be reported to HybridTE during the lifetime of the elephant. This way, in the first phase of its lifetime, the elephant will be handled by static routing and after some time it will be reported and rerouted, if necessary. We are now going to study the influence of this delay on the performance of HybridTE. We fixed the load level to 1 and varied two parameters: a) the percentage of false negatives and b) the delay between starting an elephant and reporting it to HybridTE.

The results depicted in Figure 5.7 show the average flow completion time of one run for each parameter tuple on flow trace 7. As we did not do any repetitions, it is not enough to make general statements. However, it is enough to get an idea on how the combination between delay and the number of reported elephants relates to the performance of HybridTE. The figure shows the reduction of the average flow compleption time created by HybridTE relative to ECMP. The delay shown on the vertical axis is without time dilation, i.e., a value of 1000 ms was translated to 150 s for our emulation. These experiments confirm the statement of the previous section: HybridTE performs better with an increasing number of reported elephants. As expected, with increasing delay between starting and reporting an elephant to HybridTE, the performance decreases.

In Chapter 6 we show that even with very basic packet sampling techniques, the amount of reported elephants is between 75 % and 100 % while the delay will be between 100 ms and 1000 ms. This means a reduction of the average flow completion time between 13.5 % and 8.9% compared to ECMP, which is pretty close to HybridTE$_{100}$ when reporting elephants right away.

Flow Installation Rates

HybridTE is specifically built to support switches with limited flow installation rates. Most flows in HybridTE are installed by the periodic rerouting where all reported (and still alive) elephants are rerouted. To find out about the actual number of flows to be rerouted in our scenario, we logged the number of elephant flows given as input to HybridTEs rerouting phase for each execution of the rerouting algorithm over all experiments we conducted so far. Figure 5.8 shows the largest number of elephants considered at once in the rerouting phase over all flow traces on load level 1. The largest number of elephants rerouted at once on load level 1 is 123. Figure 5.9 plots the number of elephants for load level 1.5. Here, the largest number of concurrently living elephants is 199. For load level 1.75 (Figure 5.10), the largest number of concurrent elephants is 257 and for load level 2 (Figure 5.11) it is 423. Even if all new routes for the elephants would share one switch, the flow installation rate on that switch is still very low, even for load level 2.

To find out which number of false positives HybridTE can handle on our flow traces without installing flow entries at a too high rate, we assume the following scenario:

- Mice flows that are falsely reported to HybridTE live not longer than the actual elephants.
- All new routes for the elephants have to be written to the switches flow tables one second after the rerouting phase ends at the latest.
- All new routes share one single switch.
- All false positives make it into the rerouting phase.
- Switches can write 12,000 flows to their flow table every second.

In this scenario, on load level 1, HybridTE is able to handle $\frac{12000}{123} - 1 \approx 97$ times more elephants before the flow installation rate at the switches gets a bottleneck. Hence, HybridTE could handle 99 % ($\frac{97}{97+1} \approx 0.99$) false positives in this scenario. Even on load level 2, the system would withstand up to 96 % false positives ($\left(\frac{12000}{423} - 1\right) / \frac{12000}{423} \approx 0.965$). In a more realistic scenario, where a) not all false positives make it into the rerouting phase (as discussed earlier), b) mice finish much faster than elephants, and c) *not all* new routes pass the *same* switch, the resulting flow installation rates on the switches are much lower.

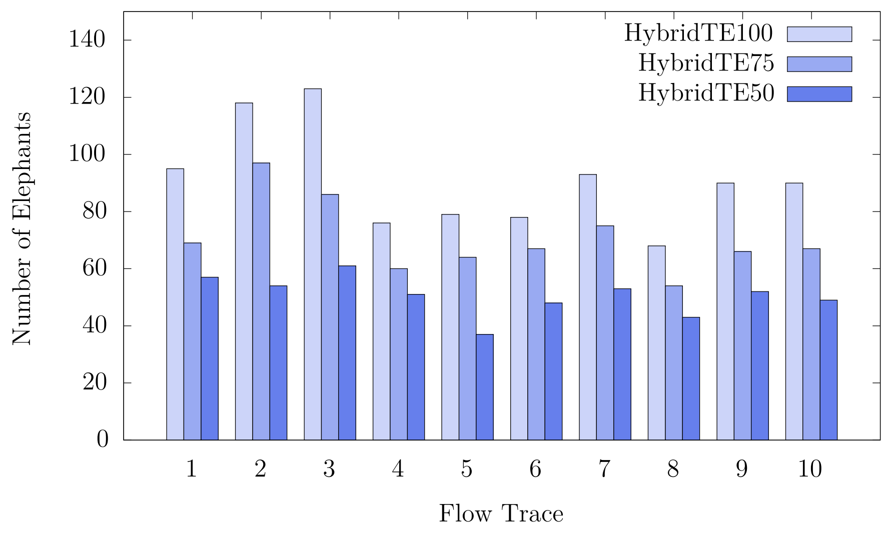


Figure 5.8.: Maximum number of concurrently living reported elephants over all ten flow traces on load level 1.

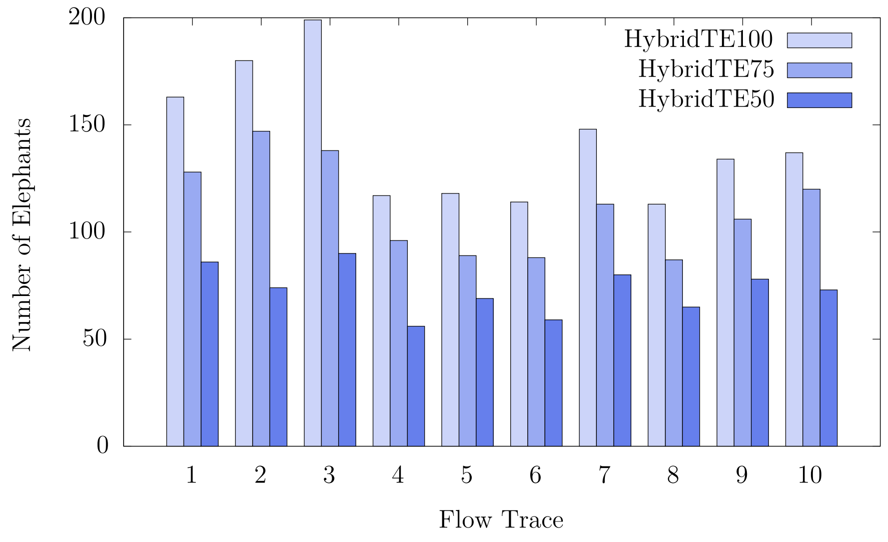


Figure 5.9.: Maximum number of concurrently living reported elephants over all ten flow traces on load level 1.5.

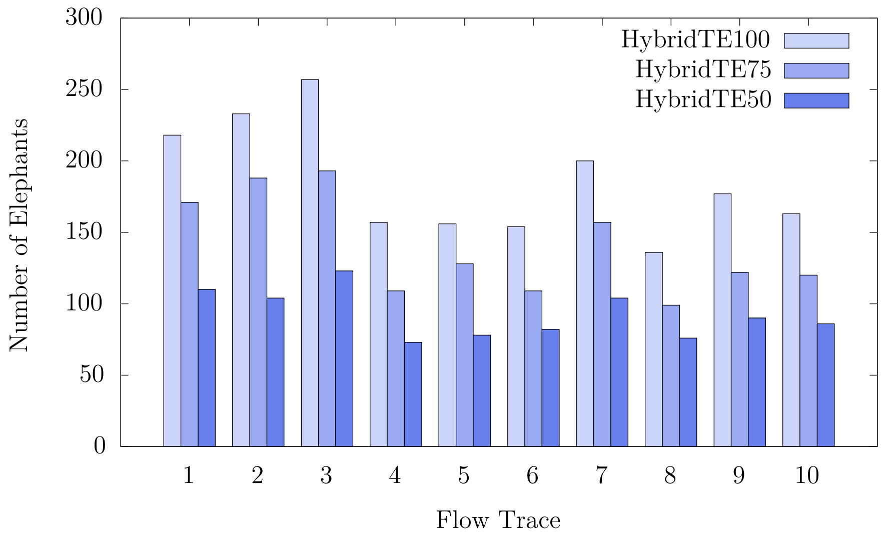


Figure 5.10.: Maximum number of concurrently living reported elephants over all ten flow traces on load level 1.75.

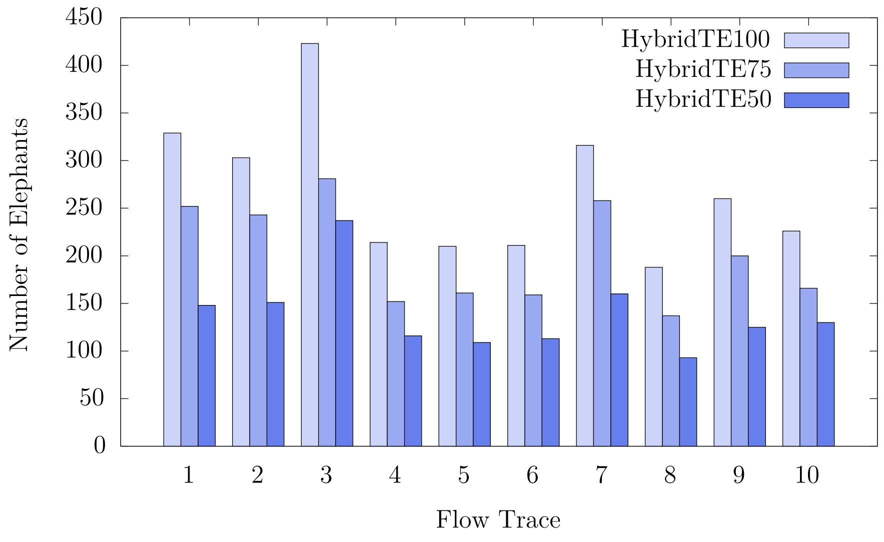


Figure 5.11.: Maximum number of concurrently living reported elephants over all ten flow traces on load level 2.

5.5. Conclusion

We showed that when using application-layer knowledge about elephant flows, we can construct a simple routing algorithm that is very efficient in resource usage while achieving better performance than state-of-the-art TE techniques leveraging uncertain information about elephant flows. Moreover, we have shown that our approach is extremely robust against false positives and can withstand even adverse false negatives of 50 % and elephant reporting delays of up to one second. Using this algorithm one can construct SDN networks using low-cost data-center hardware with similar or even better performance characteristics as networks from expensive end-of-line equipment.

To conclude, this chapter shows that by leveraging application-layer knowledge, one can not only create networks that decrease the flow-completion times by up to 14.9% but also reduce CAPEX because the hardware requirements on the switches are less than for other TE schemes.

6 Elephant Detection

HybridTE presented in Chapter 5 outperforms ECMP even when only 50% of all elephants are reported with a reporting delay up to 1 second. However, HybridTE mandates that applications communicate the existence of elephant flows to the network, which traditional applications do not. To cope with such non-participatory applications and to simplify the transition from a traditional data center to a data center hosting participatory applications, another source of information for flows from arbitrary applications is needed that does not require changes to the applications.

This chapter shows how to discover implicit information about elephant flows. To this end, a so-called elephant detector is required. Elephant detectors can either be middleboxes deployed in the data center or software components making use of the upcoming technology of *Network Function Virtualization*. Both alternatives require additional hardware resources either in addition to or inside the switches. We show that generating information about elephant flows is also possible using a very basic packet sampling approach which does not impose significant additional hardware requirements because most off-the-shelf switches already have hardware support for packet sampling. We first propose a small extension to the OpenFlow standard to configure packet sampling via OpenFlow and subsequently use that extension to generate implicit information about elephant flows. Evaluation shows that using a small numbers of samples, elephants in data-center traffic can be detected with high quality.

6.1. Introduction

In the OpenFlow model, switches only consist of a forwarding plane that is equipped with a *flow table*. As already mentioned in Section 2.2, a flow table is a collection of flow entries that identify individual traffic flows and determine how each packet of a flow is processed. There are 12 packet header fields to match on where matching can either be done exactly, where each single field has to match, or wildcarded, where particular header fields can be omitted. Wildcarded flow entries thus group several individual flows together and treat them in the same way.

Whenever a switch does not have an appropriate flow entry for an incoming packet, this packet is encapsulated in a `PACKET_IN` message and sent to the controller. The controller then computes an appropriate route for the flow and pushes the corresponding flow entry to the switch. This way, all subsequent packets of that flow are handled by the newly installed rule. For a network with a high arrival rate of new flows the amount of `PACKET_IN` messages is very high. However, even current high-end switches cannot keep up with such a high rate. To cope with these limitations, wildcard flow entries have to be installed in the switch. Thus, default routes for different groups of flows are defined in a proactive manner. But due to grouping, the fine-grained control of the network is lost. The only way to see which single flows are hiding behind a wildcarded flow is to delete the wildcard flow entry from the switch. Then, each subsequent packet that was previously covered by the wildcard flow entry creates a `PACKET_IN` that is sent to the controller. Clearly, this is a very inefficient way to look behind a wildcard flow entry.

This chapter proposes to add a packet sampling mechanism to the OpenFlow standard to efficiently unveil which individual flows are hiding behind a wildcard flow entry. Since we use packet sampling, the obtained results are only approximations. Unlike in conventional packet sampling techniques (like sFlow), samples are not taken from all incoming packets but only from the wildcarded flow entries that are under suspicion. This is more economical: for a given level of accuracy, less samples are required.

6.2. OpenFlow Packet Sampling Extension

To add sampling support to OpenFlow, we propose the following extensions to the standard: To invoke the sampling process on a switch for a specific flow entry we extend the `ofp_stats_type` by the new type `OFPST_SAMPLING`. The body of such a message is defined in Figure 6.1. On reception of this message, a switch marks every matching wildcard entry with the requested sampling probability and duration. In the message, `sampling_period` specifies the average sampling period. If set to p, 1 out of p packets are chosen at random and sent to the controller. `duration` specifies the sampling duration in milliseconds.

Sampled packet headers are sent to the controller in an `OFPT_FLOW_SAMPLE` message (Figure 6.2), where `total_len` is the length of the encapsulated ethernet frame (without payload) and `sample_len` is the length of the originally sampled packet (including payload). `in_port` is the port on which the packet was received. The

```
struct ofp_flow_sampling_request {
    struct ofp_match match;  /* Fields to match. */
    uint8_t table_id;        /* Table ID (from ofp_table_stats) */
    uint8_t pad;             /* Padding */
    uint16_t bucket_size     /* Bucket size */
    uint32_t max_samples;    /* Max number of samples */
    uint16_t out_port;       /* Output port. A value of OFPP_NONE
                               indicates no restriction. */
    uint16_t sampling_period;/* Average sampling period */
    uint16_t duration;       /* Sampling duration in ms */
};
```

Figure 6.1.: `ofp_flow_sampling_request` message.

```
struct ofp_sample_flow {
    struct ofp_header header;
    uint16_t total_len;     /* Full length of frame. */
    uint16_t sample_len;    /* Length of sampled packet. */
    uint16_t in_port;       /* In port */
    uint16_t bucket_empty;  /* Indicator, if bucket empty */
    uint64_t cookie;        /* Opaque controller-issued ID. */
    uint8_t data[0];        /* Ethernet frame */
};
```

Figure 6.2.: `ofp_sample_flow` message.

message additionally contains the `cookie` of the wildcard entry that was matched. This way, it is possible to have one switch sample multiple rules at a time without large processing overhead at the controller to find out from which wildcard entry the sample was taken. If the wildcard flow entry from which a packet is sampled does not have a cookie, the value -1 (`0xffffffffffffffff`) is used.

When a wildcard flow suddenly increases its data rate during a sampling period, the controller is flooded with `OFPT_FLOW_SAMPLE` messages. To counteract this, we use the *token bucket* algorithm to limit the emission of samples. This is depicted in Figure 6.3. The token generation rate is defined as $r = \frac{\texttt{duration}}{\texttt{max\_samples}}$ where a token is added to the bucket every $1/r$ seconds. `bucket_size` denotes the size of the bucket. When the bucket gets empty, the controller is informed by sending an `ofp_sample_flow` with `bucket_empty` set to 1 (`0x0001`); by default `bucket_empty` is always set to 0 (`0x0000`).

The packet sampling extension is very simple to implement since all required building blocks are already used in the OpenFlow implementation. Only a random generator is required in addition. Our extension requires 208 bits storage overhead per installed wildcard flow entry. We implemented this proposal in the OpenFlow 1.0 userspace reference implementation. Figure 6.4 plots the forwarding performance of our implementation where "modified" refers to the reference implementation enhanced with the packet sampling extension and "Non-modified" is the original reference implementation. We installed two wildcard rules on a switch connecting two hosts (one for each direction). The plot shows the maximum achievable data rate between both hosts under different sampling periods. It can be seen that there is no significant decrease in the achievable data rate for sampling periods larger than 100. The switch

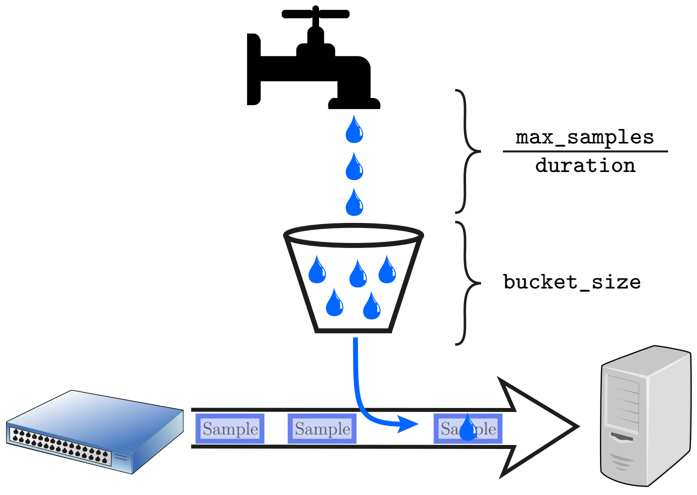


Figure 6.3.: The token bucket algorithm for packet sampling.

was emulated using Mininet on a Intel Core i7 QM 2.2 GHz with 8 GB memory. The experiment was repeated 20 times to compute confidence intervals.

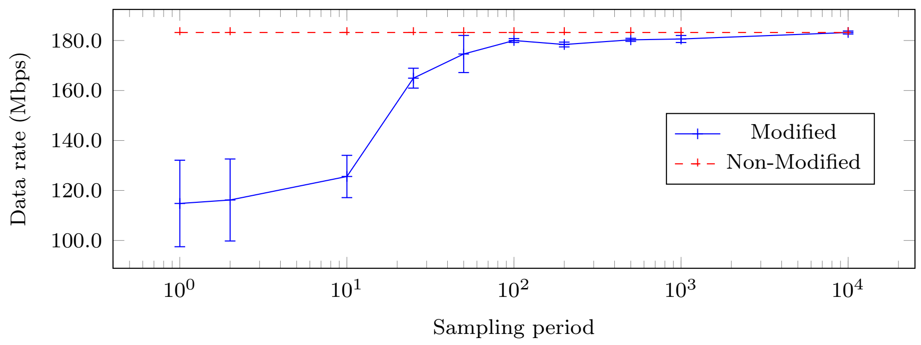


Figure 6.4.: Throughput over different sampling rates. Error bars show confidence intervals with confidence level of 95%.

6.3. Using Packet Sampling for Elephant Detection

With the packet sampling extension proposed in Section 6.2 it is not only possible to look behind a wildcard rule but also to detect elephants: A packet is considered ("sampled") with probability $\frac{1}{p}$ and we assume a flow to be an elephant if we sampled at least n packets of the same flow. Evaluation of the samples subsequently takes place at the OpenFlow controller. Using the parameters p and n, the rate and quality at which elephants are reported can be controlled.

6.4. Evaluation

To evaluate how packet-sampling-based elephant detection techniques perform in a data-center environment, we set up the following experiment: We emulate a data center employing ECMP using MaxiNet and DCT^{2}Gen as explained in Section 5.4. At one of the core switches of the topology, we use `tcpdump` to write a transcript of all packets routed through the first interface.[5] We subsequently evaluated the packet sampling technique described in Section 6.3 on these transcripts for $p \in \{100, 1000, 5000, 10000, 100000\}$ and $n \in \{1, 2, 5, 10, 25\}$. The reporting delay for an elephant flow is the time between the first packet of the flow arrived at the switch (independently whether this packet was sampled or not) and the time we sampled the n-th packet of the flow. This introduces a small error because naturally, the starting time of a flow is when the first packet of the flow is generated by the source. However, since the core switch is only three hops away from the end hosts and latency in data centers is very low, the error is small (in the order of sub-milliseconds).

We repeat the experiment ten times using ten different transcripts each containing traffic from different schedules that were independently generated by DCT^{2}Gen. Figure 6.5 shows the resulting rate of false negatives, i.e. the percentage of elephants that are not detected by the elephant detection. Figure 6.6 shows the corresponding delay between the detection of an elephant and its start. Finally, Figure 6.7 shows the resulting percentage of false positives.

Together with Figure 5.7, the performance of HybridTE under the different configurations of the elephant detector can be predicted. Our data shows that even if only very few packets are inspected, the predicted performance of HybridTE is very high: With $p = 10{,}000$ and $n = 1$, 92% of all elephants are detected with a mean reporting delay of only 378 ms. The resulting percentage of false positives is 86%. Recalling Figure 5.6, this amount of false positives does not impact the performance of HybridTE, which is why for this set of parameters we suspect HybridTE to reduce the average flow completion time relative to ECMP by 11.7 % to 12.4 % (Figure 5.7).

Using higher numbers for n, the amount of false positives can be reduced significantly (Figure 6.7). However, as can be seen in Figure 6.6, this comes at the cost of a higher reporting delay. In addition, values of $n > 1$ as an indicator for an elephant require a more complex design of the elephant detector. For each flow from which a packet is sampled, a counter has to be created and updated. This can be problematic for systems with limited memory. Hence, it requires a strategy to remove entries representing old flows from the memory. Trivial examples for such candidates are *First in First out* (FIFO), *Least Recently Used* (LRU), or a *random* selection. The strategy itself can influence the quality of the elephant detector. But as the data shows that $n = 1$ yields high quality results, we did not further study the effect of

[5]Due to the performance of our testbed it was not possible to employ packet sampling on all interfaces of the emulated switches concurrently. In turn, to compute the percentage of false positives/negatives we only considered flows that were routed over the above-named interface.

possible strategies on the quality. Thus, the results shown in this chapter are for a scenario with unlimited memory resources. When elephant detection is performed by the SDN controller, the available memory will be sufficiently large such that even when strategies like FIFO or LRU are used, the results will be comparable with the unlimited memory case because only *very* old entries will be removed from memory.

6.5. Conclusion

With our OpenFlow packet sampling extension it is efficiently possible to find out which individual flows are hiding behind a wildcard flow entry. This is a powerful tool when building OpenFlow applications that reside on wildcard flow entries but require to know detailed traffic information after installing a wildcard flow entry.

We could show that packet sampling can also be used as a very basic elephant detection technique that allows to use HybridTE in data centers with non-participatory applications and without additional hardware. From the data it can be forecast that the performance of HybridTE is up to 12.4 % higher than ECMP in this scenario with non-participatory applications. In a scenario with mixed non-participatory and participatory applications, we suspect the performance is even higher.

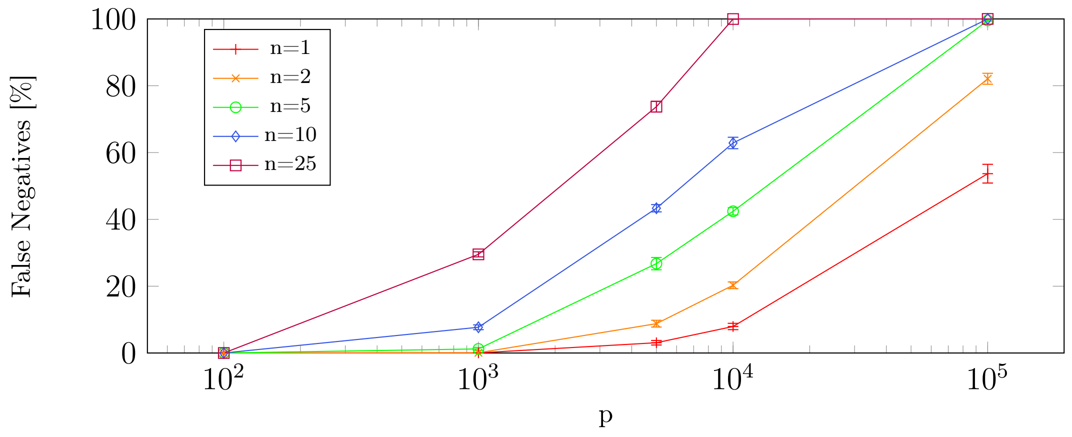


Figure 6.5.: Rates of false negatives, i.e. elephants not reported, for different parameters. Error bars illustrate the confidence interval for a confidence level of 95%.

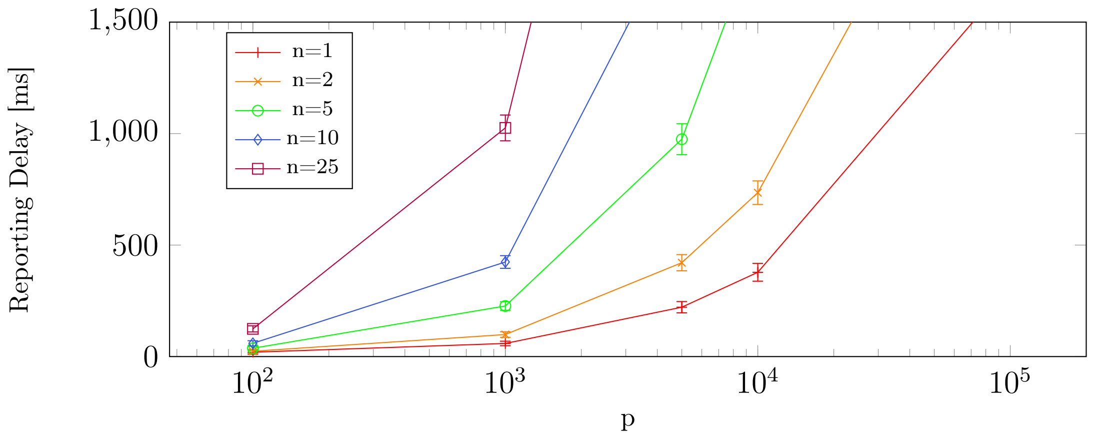


Figure 6.6.: Reporting delay for different parameter sets. Error bars illustrate the confidence interval for a confidence level of 95%.

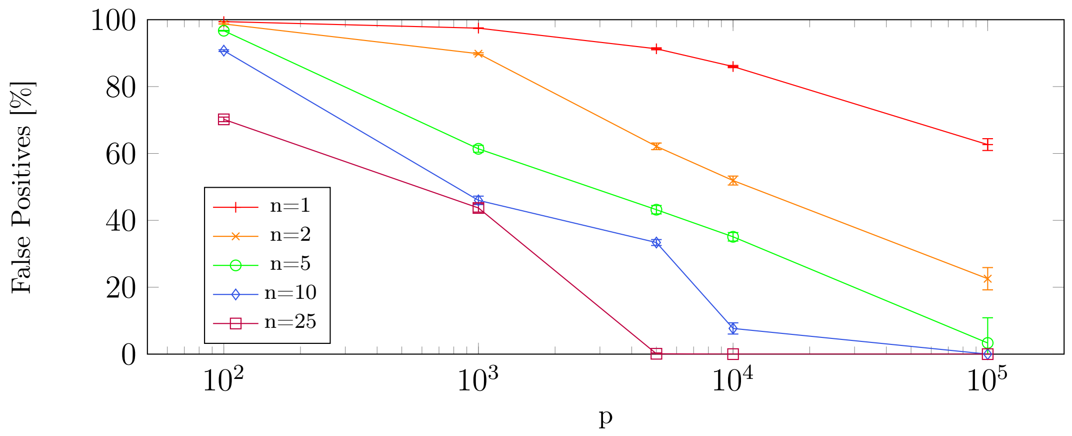


Figure 6.7.: Rates of false positives, i.e. mice falsely reported as elephants, for different parameter sets. Error bars illustrate the confidence interval for a confidence level of 95%.

Part III.

Emulation Tools

7 MaxiNet: A Scalable Network Emulator

Network emulations are widely used for testing novel network protocols and routing algorithms for packet-switched networks because they enable using an actual implementation of a network stack provided by a real operating system. This allows to include all effects arising from the actual implementation of network protocols to the timing and scheduling of interrupts by the operating system into an evaluation. To conduct the experiments shown in Chapter 5, it was necessary to emulate a whole data center. Unfortunately, no emulation environment was scalable enough to emulate a network as large and with as many concurrent flows as in a data center.

Mininet [LHM10] is the most common tool to emulate Software-Defined Networks of several hundred nodes. This chapter shows how to extend Mininet to span a network emulation over several physical machines, making it possible to emulate networks of several thousand nodes on just a handful of physical machines. This enables us to emulate, e.g., large data-center networks.

To demonstrate the scalability of MaxiNet, we emulated a data center consisting of 3200 hosts on a cluster of only 12 physical machines. This chapter shows the resulting workloads and the trade-offs involved.

7.1. Introduction

When evaluating new algorithms or protocols for Software-Defined Networks (SDN), emulation using Mininet [LHM10] is the first choice. Mininet uses network namespaces to create separate network contexts for each process running together on one physical machine. With this technique, several OpenFlow-enabled software switches can be run on one machine interconnected by virtual network interfaces. In low-traffic scenarios, Mininet scales to several hundred nodes on state-of-the-art hardware. But when emulating large networks with high traffic volume, the computational complexity of the emulation overwhelms today's computers. In such scenarios it is still possible to successfully emulate the network by using a technique called *time dilation* [GYM+05]. Time dilation slows down the emulated time with respect to the walltime by a constant factor (called *dilation factor*) but keeps the speed of peripheral devices constant. By setting the dilation factor to 10, one second of a 10 Gbps link can be emulated by using 10 seconds of a 1 Gbps link. Of course, using time dilation, the amount of walltime required for the experiment is multiplied by the dilation factor.

We show that the relation between the size of the emulated network and the required dilation factor is *not linear*, leading to unacceptable run times for large network emulations. We suspect this non-linearity is due to the huge amount of required network namespaces, processes and virtual network interfaces, which adds a high amount of overhead to the system and causes the host operating system to work inefficiently. The required dilation factor can be reduced by *distributing the emulation* over multiple physical machines. This is a very time-consuming and error-prone process when done by hand because it has to be decided a) how to partition the virtual network, b) which partition is emulated on which physical machine, c) how to invoke commands at the emulated nodes to keep the experiment synchronized, and d) how to collect the resulting data from the machines for evaluation. In addition, when using such hand-crafted solutions, measurements have to be made to ascertain that the built system works properly (for example, does not distort the latencies between nodes).

This chapter presents a framework called MaxiNet to use multiple physical machines for large-scale SDN emulations. The whole process of mapping and deploying the network to be emulated onto the physical environment is transparent to the user. Using MaxiNet is like specifying an experiment with Mininet. We evaluated our system through several experiments and found that it properly replicates the properties of a single-machine emulation environment.

We built MaxiNet to allow us to test new routing algorithms for data-center networks. This did not only require a scalable emulation environment but also realistic data-center traffic. Today, there are no publicly available traffic traces from data centers but recent studies [KSG+09, BAM10] reported several properties of such traffic. Using their findings we built a traffic generator that produces traffic at flow level. This traffic generator is called DCT^2Gen and is described in Chapter 8. Using MaxiNet we were able to emulate a data center interconnecting 3200 servers under realistic traffic using only a dilation factor of 200 on 12 physical machines. MaxiNet and DCT^2Gen are open source. They can be downloaded from our websites[6]. We also provide preconfigured virtual machine images that can be used to test MaxiNet.

[6]`https://www.cs.upb.de/?id=maxinet` resp. `https://www.cs.upb.de/?id=dct2gen`

The rest of the chapter is structured as follows: Section 7.2 presents work related to large-scale SDN evaluation. In Section 7.3 we present MaxiNet itself. Section 7.4 presents the experience we gained in large-scale data center emulation with MaxiNet. Section 7.6 concludes this chapter.

Throughout the text, with *worker* we refer to physical machines that are used to compute the emulation. The term *node* is used for switches and end hosts that are emulated. The network that is to be emulated (consisting of interconnected nodes) will be denoted as *virtual network*.

7.2. Related Work

When evaluating new algorithms or protocols for SDN it usually comes down to the choice between simulation and emulation for a test implementation. For both directions there is already a number of tools available. For simulation there are OMNeT++ [Var], NS3 [HLR+08, GRL05] and some commercial tools. NS3 already has a built-in module for OpenFlow and a comparable extension for OMNeT++ [KJ13] also exists.

For our approach we wanted to be able to run native SDN controllers and to include the effects from a native network stack into our evaluation. It is technically possible to implement both in a simulation framework, but from our point of view it is more straightforward to use emulation instead of simulation.

For emulating SDNs there are two possible tools to be considered: Mininet [LHM10, HHJ+12] and EstiNet [WCY13]. EstiNet is a commercial tool, which contradicts our idea of building a system that is freely extensible and usable by other researchers. Thus, we focus on Mininet for our implementation.

The option to simulate or emulate large SDNs has been researched in a number of papers: In [JN13] the authors describe how OpenFlow can be run on top of a simulation engine called *S3F*. This approach is limited to one physical machine, thus the performance of this approach is limited. The authors of [GSB13] show how replacing Mininet with their own tool *fs-sdn* can speed up the simulation of SDNs, but again this approach is limited to one physical machine. In [DHM+13] an elastic OpenFlow controller is proposed that grows and shrinks with the amount of routing decisions that have to be made. For evaluation the authors built a hand-crafted emulation testbed based on several Mininet instances that were interconnected by GRE tunnels. The authors of [AS13] claimed they built a distributed version of Mininet for the purpose of malware propagation analysis. However, they omit a description of how their system works and do not show the adequacy of the results obtained with their system.

With respect to our main use case to emulate data-center traffic, the work in [LSN+09] investigates the distributed simulation of a multi-tier data center but does not include the simulation of SDN.

7.3. MaxiNet

7.3.1. Requirements

When designing MaxiNet we had the following requirements:

- Centralized programming model that is similar to specifying an emulation with Mininet
- Linear scaling of the virtual network size with the number of physical machines
- Leverage original Mininet
- Small (physical) network footprint

To achieve these goals we had to solve different problems. The first problem is how to partition the virtual network onto the workers such that a) the physical network does not become a bottleneck and b) the workload is evenly distributed over all workers. Traffic from one partition of the virtual network has to reach all other partitions, requiring forwarding across multiple physical machines. This forwarding process must not introduce a noticeable penalty to the latencies experienced by the nodes. Otherwise the partitioning could influence the outcome of an experiment.

As we want to have a centralized programming model, we need to access nodes across the different workers and these workers have to be synchronized. To achieve this, we decided to run all the control logic of an experiment (the course of when to run which command at which node) on one specialized physical machine called the *frontend*. The frontend partitions and distributes the virtual network onto the workers and keeps a list of which node resides on which worker. This way we can access all nodes through the frontend. The frontend itself can also act as a worker and is manually selected prior to the experiment.

7.3.2. Overview

MaxiNet is an abstraction layer connecting multiple, *unmodified* Mininet instances running on different workers. A centralized API is provided for accessing this cluster of Mininet instances. GRE tunnels are used to interconnect nodes emulated on different workers. MaxiNet works as a front end for Mininet that sets up all Mininet instances, invokes commands at the nodes and sets up the tunnels required for proper connectivity.

Figure 7.1 shows a schematic view of MaxiNet. A network experiment can use MaxiNet to set up, control and shut down a virtual network by using the MaxiNet API. This API is designed to be very close to the Mininet API to ease using MaxiNet when already familiar with Mininet. The emulation as such happens on a pool of workers. Workers are controlled by MaxiNet using the Mininet API. Communication between MaxiNet and Mininet happens through RPC calls.

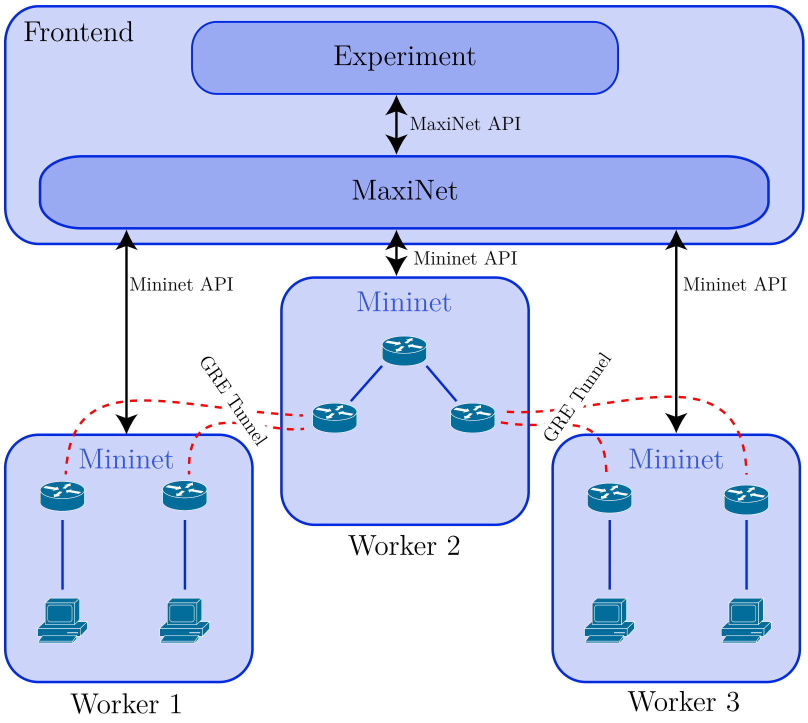


Figure 7.1.: Schematic view of MaxiNet.

For partitioning a virtual network onto several workers we use the graph partitioning library METIS [KK95]. For n workers, METIS computes n partitions of near equal weight. The goal of our partitioning process is to confine most of the emulated traffic locally to the workers. The optimization criteria we use for partitioning is minimal edge cut. Edge weights in the partitioning process are proportional to the data rate limits specified in the virtual topology. Node weights in the partitioning process are chosen to be proportional to the corresponding node degree in the virtual topology. This makes sense because a node with a higher number of links is likely to forward more traffic, thus causing more load to the worker.

7.3.3. Using MaxiNet

To acquaint the reader with MaxiNet, this section provides a minimal example on how to use MaxiNet. Figure 7.2 shows the complete Python code (error checking omitted) required to set up and run an experiment on a tree network that is emulated on three different physical machines. First, a `MininetCluster` object is built (line 4). This object holds a list of all n physical hosts' network addresses (here: DNS names pc1, pc2, and pc3) that will be used as workers. The `start` function (line 5) prepares the workers for the emulation by starting a Mininet instance on each machine.

The `Emulation` object (line 7) is the interface to MaxiNet's API. It's creation requires the `MininetCluster` object and a `mininet.topo.Topo` object describing the virtual network topology. In the example we use a tree topology from the Mininet

```
 1 import maxinet
 2 import TreeTopo from mininet.topolib
 3
 4 cluster = maxinet.MininetCluster("pc1","pc2","pc3")
 5 cluster.start()
 6
 7 emu = maxinet.Emulation(cluster, TreeTopo(3,2))
 8 emu.addController("192.168.0.1", "6633")
 9 emu.setup()
10
11 print emu.get("h2").cmd("ping -c5 10.0.0.3")
12
13 emu.stop()
14 cluster.stop()
```

Figure 7.2.: Minimal example experiment using the MaxiNet Python API.

library specified as `TreeTopo(3,2)`. The `addController` function (line 8) is used to specify the OpenFlow controller for the experiment. By calling the `setup` function (line 9), the topology is clustered into n partitions using METIS. After clustering, from each partition a `mininet.topo.Topo` object is built and emulated at a worker. For each edge in the topology that is between nodes in different partitions a GRE tunnel is set up that directly connects to the interfaces of the nodes.

After invoking the `setup` function of the `Emulation` object the emulation begins. Now, during the run time of the emulation, it is possible to invoke commands at the nodes (line 11). To do so, first the node object is fetched via the `get` function. Afterwards, the `cmd` function can be used to pass a shell command that is executed in the node's environment.

7.4. Performance Evaluation of MaxiNet

7.4.1. Scalability

Environment

The small-scale experiments presented in this section are all run on a cluster of 4 Intel i7 3.3 Ghz quadcore servers with 24 GB RAM interconnected by 1 Gbps Ethernet in a star topology. The software switch we used is the Openflow 1.0 userspace reference implementation. We did not use OpenVSwitch because even in its latest version (2.0) OpenVSwitch does not scale well in scenarios where a high amount of switches is emulated at a single host. When emulating a fat tree of depth 7 the throughput of the single switches was highly fluctuating, leading to very bursty traffic patterns. We thus recommend using the userspace reference implementation when aiming at emulating a very high amount of switches, even though the forwarding performance is not as high as for OpenVSwitch.

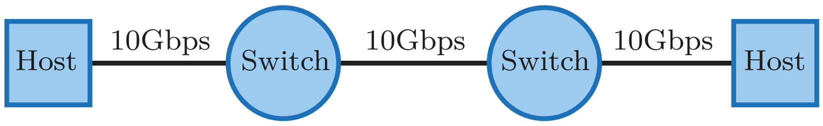


Figure 7.3.: One switch pair consisting of 2 switches and 2 hosts connected in a line.

Single Worker

To understand the scalability of a single Mininet instance, we run the following experiment: We create pairs of switches with one host each that are interconnected in a line by 10 Gbps links (see Figure 7.3). Two flows are created (one for each direction) to fully utilize the links. To find out the lowest possible dilation factor we monitored the data rates of the links and lowered the dilation factor as long as all links were fully utilized. We repeated the experiment ten times to compute confidence intervals. The dashed line in Figure 7.4 plots the number of emulated 10 Gbps links against the required dilation factor (by choosing a smaller factor the processing power of the worker does not suffice to forward all packets). Note that each emulated switch pair (like the one shown in Figure 7.3) corresponds to 30 Gbps. It can be seen that due to the introduced overhead per node the scaling is *not linear*.

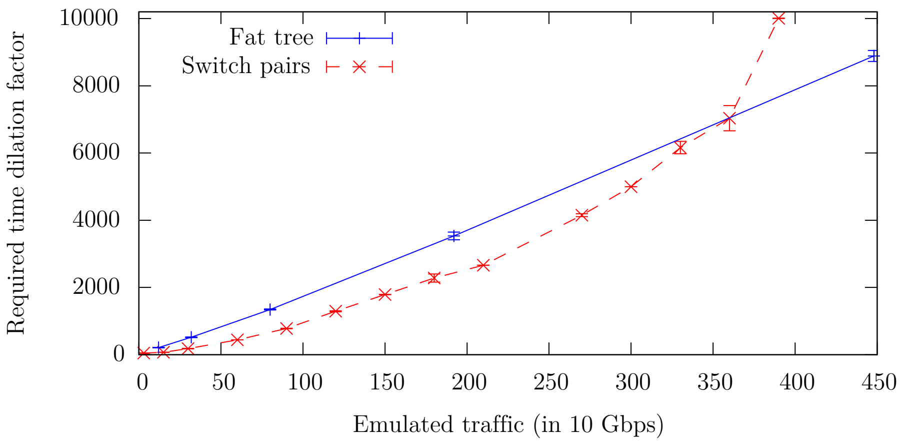


Figure 7.4.: Required dilation factor when emulating on a single physical machine. Confidence interval for a confidence level of 95 %.

We now look at fat trees with full bisection bandwidth and a data rate of 10 Gbps on the lowest (ToR) level. Each ToR switch is connected to one host for traffic generation. We again aim at fully utilizing every single link while keeping the dilation factor as low as possible. For a fat tree with n ToR switches we create flows between ToR i and $n - i$ to accomplish full utilization of every link in the topology. The solid line in Figure 7.4 shows the outcome of this experiment: the fat-tree scenario scales nearly linear, which is due to the small amount of emulated switches and hosts in comparison to the line scenario. The largest fat tree used in this experiment had a

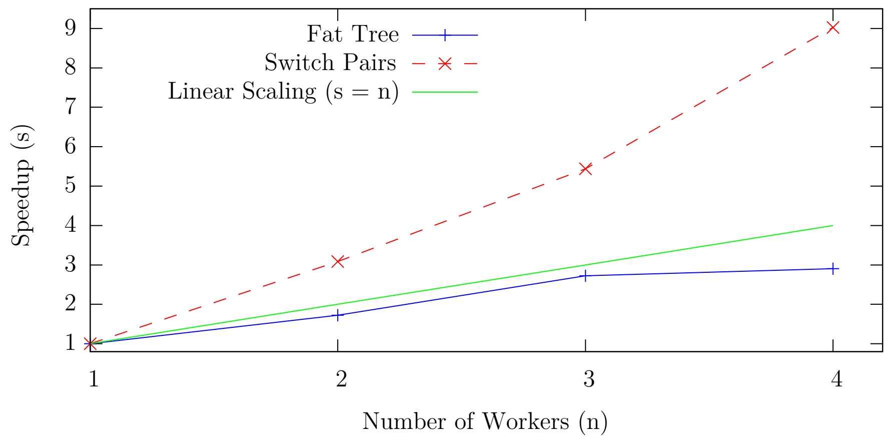


Figure 7.5.: Speedup gained from an increasing number of workers in the fat-tree and switch-pair scenarios.

depth of 6 resulting in 128 switches and 64 hosts. The largest switch-pair scenario consisted of 150 pairs, which results in 300 switches and 300 hosts.

Multiple Workers

We now show the effect of distributing the two scenarios onto multiple workers using MaxiNet. For the experiment we use 120 switch pairs in the switch-pairs scenario. This results in 240 switches, 240 hosts and 360 emulated 10 Gbps full-duplex links with an aggregated traffic of 7200 Gbps. For the fat-tree scenario we fix the number of leaf nodes to 64, resulting in a fat tree of depth 6. The number of switches is 128 and the number of hosts is 64. The aggregated data rate in this scenario is 8960 Gbps and hence beyond the capabilities of a single Mininet instance. The two scenarios are chosen to be the best case for the emulation, respectively the worst case because in the switch-pair scenario the virtual network consists of many isolated and thus independent network components and in the fat tree *all traffic* is routed through one single switch (the root of the tree). Since the root can only be emulated at one worker this affects the performance of the whole network and restricts the possible scalability when distributing to a cluster of workers.

Figure 7.5 plots the speedups over the number of used physical workers. For the switch-pairs scenario, the speedup is better than linear which is due to the lower overhead of smaller virtual networks. When using only one worker, the amount of nodes and virtual links is very high, leading to severe performance penalties (the Linux kernel has to handle 720 virtual interfaces and 360 veth pairs *and* do traffic shaping on them). When distributed over multiple workers the overhead gets lower and no longer hurts performance.

It can be seen from Figure 7.5 that the effect of adding more workers to the fat-tree scenario does not have a large effect on the speedup. There is basically no gain from

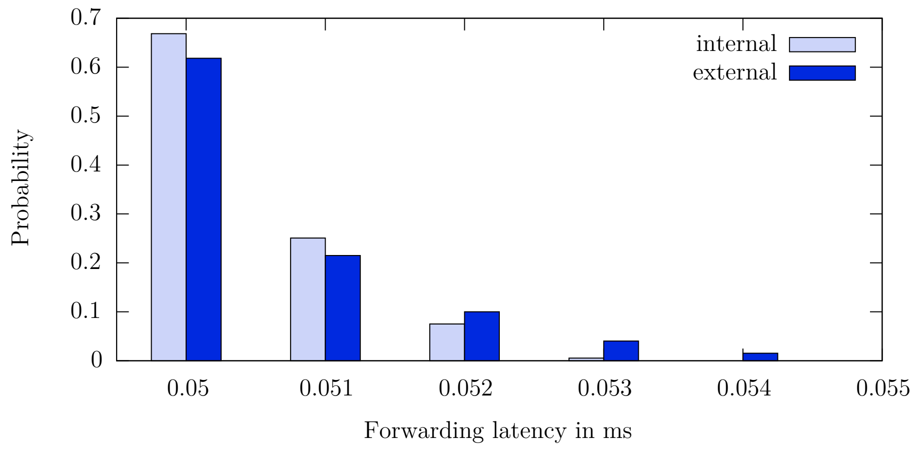


Figure 7.6.: Forwarding delays between nodes emulated at the same/different worker nodes.

adding a fourth worker to the cluster. As already said, this is due to the root node of the fat tree. The experiment is designed such that all traffic is routed over the root switch. That switch can only be emulated at one single worker and thus its CPU becomes the bottleneck of the emulation.

Latency distribution

Since we are using GRE tunnels over a physical network infrastructure to interconnect the different workers, we now take a look at the latencies between hosts emulated at the same physical machine and nodes emulated at different machines. To make latency measurements we use the fat-tree topology described above with 128 leaf nodes (depth of 7) and a dilation factor of 5000 emulated at 4 worker nodes. We decided for a dilation factor of 5000 due to the results shown in Figure 7.4 and Figure 7.5.

Each link has an emulated delay of 0.05 ms. With the dilation factor this means a latency of 250 ms was configured for each link. We use UDP flows and let them create a utilization of 30% on each link. During the experiment we perform all-to-all latency measurements. The latency histogram between nodes emulated at the same worker (internal) and nodes emulated at different workers (external) are plotted in Figure 7.6. Both distributions do not differ significantly from each other. This is because the physical network only adds a negligible latency in comparison to the used dilation factor.

Effects of a distributed emulation

When distributing an experiment over multiple workers with MaxiNet, the results should be the same as when running the experiment with original Mininet. To show that MaxiNet does not distort results we ran the following experiment on both original Mininet and MaxiNet: We emulated a data center consisting of 300 servers intercon-

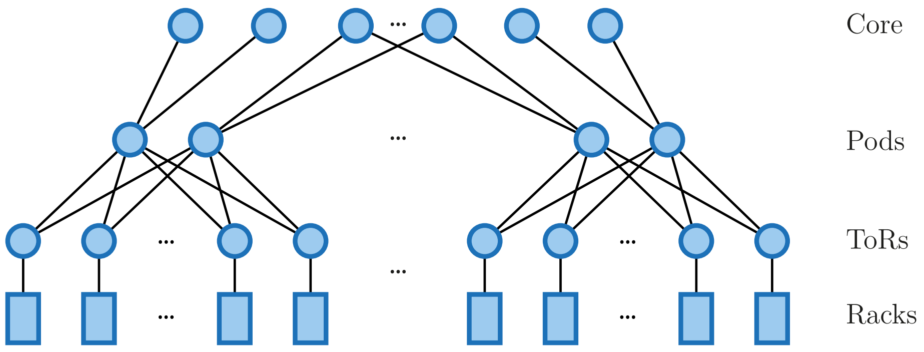


Figure 7.7.: Sketch of the emulated Clos-like topology.

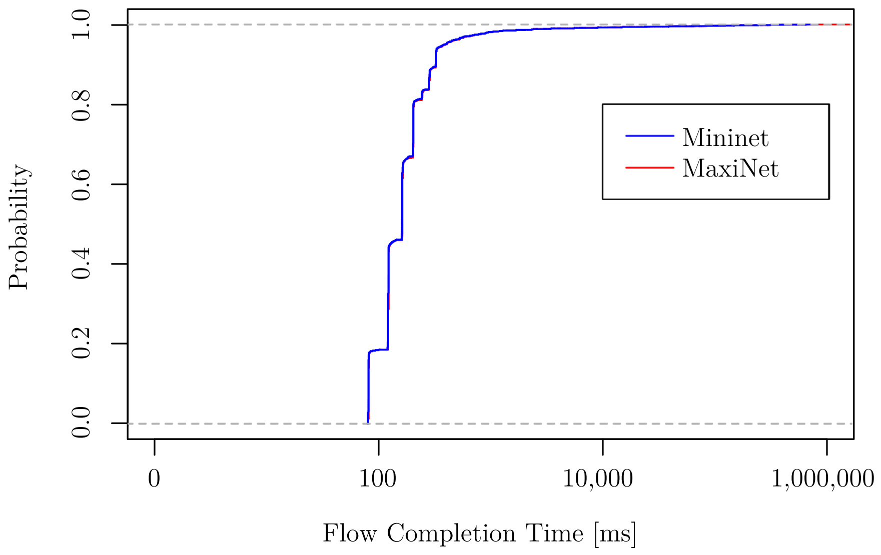


Figure 7.8.: Distribution of flow completion times between 300 servers emulated at a) one worker (Mininet), and b) four workers (MaxiNet) in a Clos-like network with a dilation factor of 200.

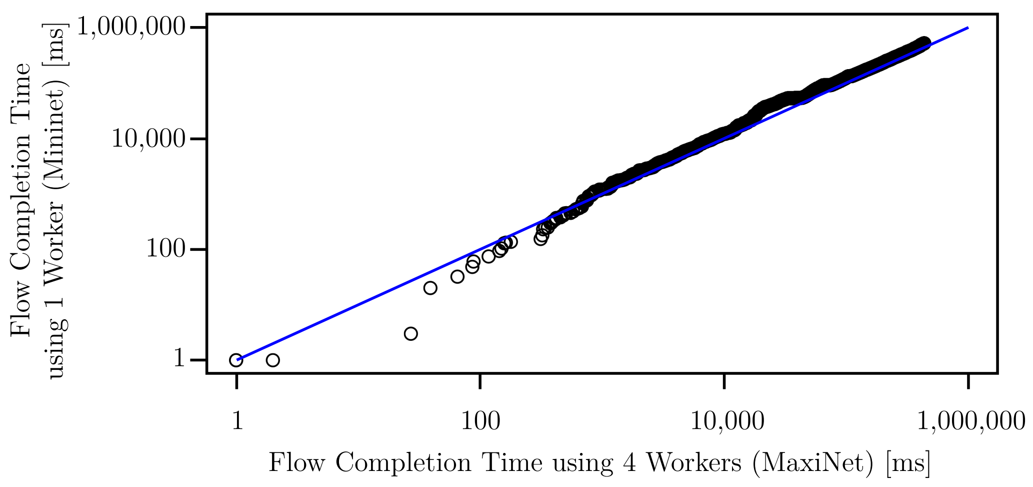


Figure 7.9.: QQ-plot of flow completion times between 300 servers emulated at a) one worker (Mininet), and b) four workers (MaxiNet) in a Clos-like network with a dilation factor of 200.

nected in a *Clos-like topology*. Figure 7.7 shows a sketch of this topology. The lowest layer in the topology are racks of servers. Each rack consists of 20 servers and one top-of-rack (ToR) switch, resulting in 15 racks. Servers are connected with 1 Gbps links to the ToR switches. *Pods* are formed by grouping three ToR switches and connecting them to two pod switches with 10 Gbps links. Each pod switch is connected to two *core switches* with 10 Gbps links. In the experiment we used two switches at the core layer. Note that for comparison, the emulation has to run at a single machine. This is why we decided for such a small scenario.

We emulated 60 seconds of traffic that was generated by DCT^{2}Gen as described in Chapter 8. The dilation factor was set to 200 which means our experiment completed after 200 minutes. We assumed a forwarding delay of 0.05 ms per switch.

Figure 7.8 shows the CDFs of the *flow completion times* for a) the experiment emulated with original Mininet and b) emulated with MaxiNet on four workers. Note that the results include the dilation factor of 200. The flow completion times in both scenarios follow the same distribution which means that for this particular scenario there is no difference between the results from MaxiNet and Mininet. This is also confirmed by the corresponding Quantile-Quantile plot (QQ-plot) shown in Figure 7.9. QQ-plots are plotting the quantiles of two distributions against each other. If the plot shows the identity function, this is an indicator that the distributions fit [CCKT83]. Figure 7.9 shows the identity function for all values larger than 10 ms. For all other values, there are too few measurements to tell if the distributions match for those values or not. Hence, we consider the results of experiments with MaxiNet to be *not* distorted when being distributed over multiple workers.

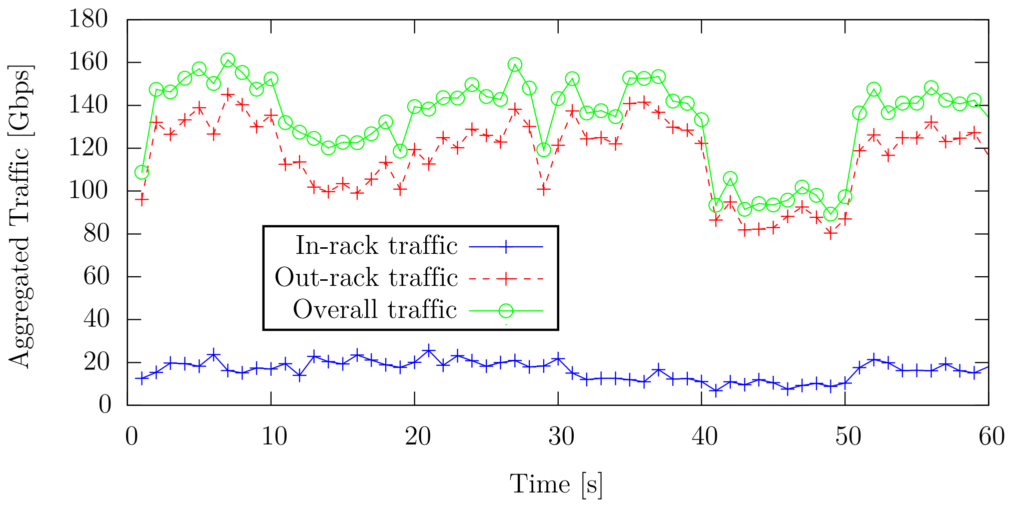


Figure 7.10.: Aggregate traffic demand used in the large-scale experiment.

7.4.2. Large-Scale Data Center Emulation

To learn about the scalability of MaxiNet we choose to emulate a data center consisting of 160 racks employing a *Clos-like topology* as before. Each rack consists of 20 servers and one ToR switch, which makes 3600 servers overall. Servers are connected by 1 Gbps links to ToR switches. Pods consist of *eight* ToR switches and are connected to two pod switches with 10 Gbps links. Pod switches are connected to two *core switches* with 10 Gbps links. The core layer in our topology consists of *seven* switches, which makes 207 switches overall. We assume a forwarding delay of 0.05 ms per switch.

We emulated 60 seconds of traffic that was generated by DCT^{2}Gen as described in Chapter 8 and used a dilation factor of 200 which means the emulation completed after 200 minutes. Figure 7.10 shows the aggregated traffic demand for both traffic that stays within one rack and traffic that is destined to servers located in other racks. It can be seen that the TM for the 40 s – 50 s period contains significantly less traffic than the TMs for other periods, which mimics a low-load phase in the data center. We did not force that to happen; it is just a variation in the random process.

The emulation took place on 12 physical worker nodes that were equipped with Intel Xeon E5506 CPUs running at 2.16 GHz, 12 Gbytes of RAM and 1 Gbps network interfaces connected to a Cisco Catalyst 2960G-24TC-L Switch. For routing, we implemented equal cost multipath (ECMP) routing based on the Beacon controller platform [Eri13]. As the controller was placed out-of-band and did not use any kind of time dilation, the routing decisions of the single controller were fast enough for the data center network. In addition, the latency between the controller and the emulated switches was not artificially increased. This means that in relation to all the other latencies in the emulated network, the controller decisions were almost immediately present at the switches and did not add any noticeable delay to the flows. Please note

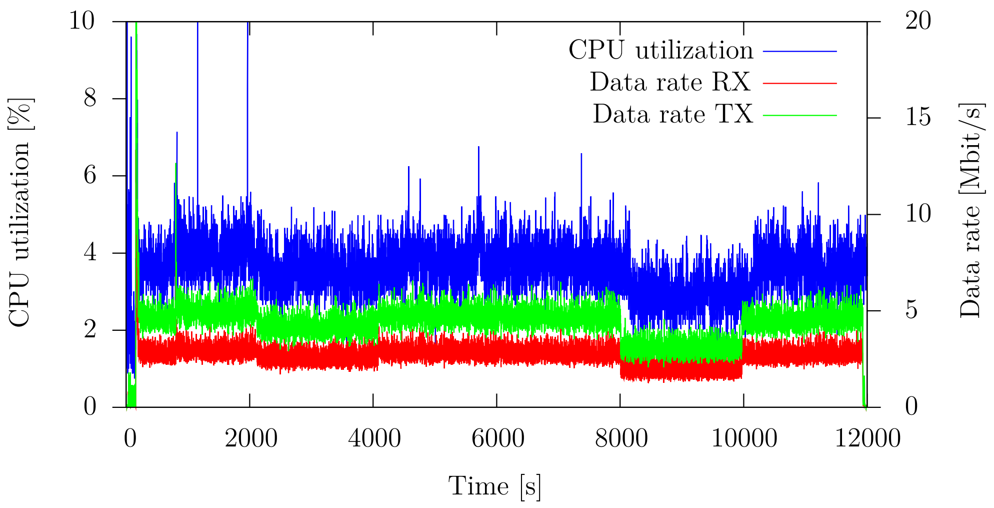


Figure 7.11.: CPU utilization and network usage of the OpenFlow controller over the course of the simulation.

that for a real data center (without using time dilation) an ECMP implementation based on only one centralized controller would not keep up with the high flow arrival rates. Figure 7.11 shows the system utilization of the OpenFlow controller we used for the experiment (which was also run at a 2.16 GHz Intel Xeon E5506). The CPU utilization of the controller is around 4% on average and the data rate required to install rules at the switches is around 5 Mbit/s. When using no time dilation these values would rise to 800% CPU utilization and 1 Gbps data rate on average. This means for a data center of this size it would require a distributed OpenFlow controller with more than 8 physical machines to run our ECMP implementation. Even assuming a perfect linear scaling of the distributed OpenFlow controller.

Figure 7.12 shows the CPU utilization of the 12 worker nodes during the experiment. It can be seen that the load is distributed evenly over the worker nodes and that load per worker is very stable. No worker was running at its capacity which means the experiment was not affected by hardware limitations. Figure 7.13 plots the corresponding network usage over the course of the experiment. It can be seen that the network also does not run at its capacity and that data rates are *not* subject to heavy fluctuation. The later is due to the ECMP routing algorithm and the type of traffic which results in an even distribution of load over the emulated switches. Please note that Figure 7.13 only plots the portion of traffic that used the physical network and does not include traffic that was exchanged on links connecting nodes emulated at the same worker. This is why there is no strong correlation between Figure 7.12 and Figure 7.13.

From the data it can be seen that 12 physical worker nodes are sufficient to emulate a data center consisting of 3600 servers interconnected in a Clos-like topology. The required physical network footprint is very low and the CPU utilization of each worker

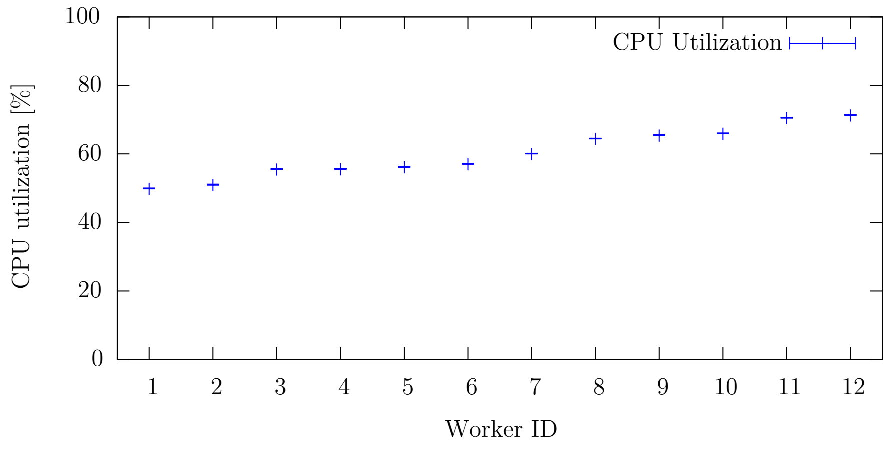


Figure 7.12.: Average CPU usage of the 12 workers during the large-scale experiment. Confidence intervals with confidence coefficient of 0.95 given.

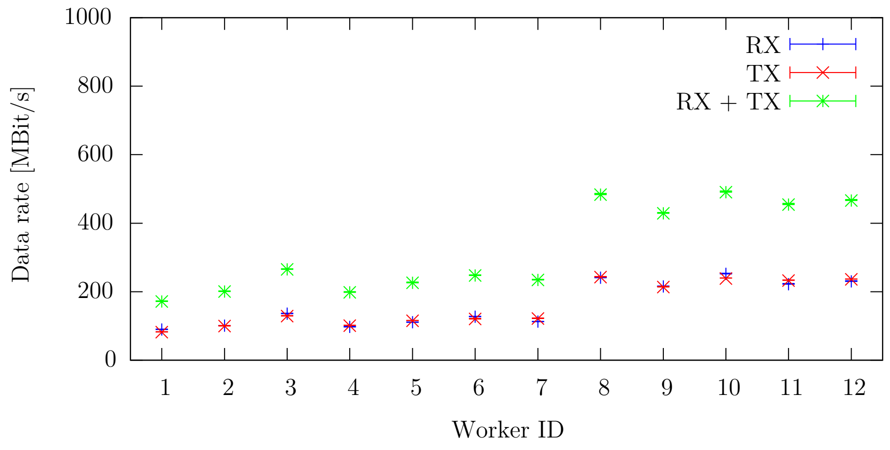


Figure 7.13.: Average network usage of the 12 workers during the large-scale experiment. Confidence intervals with confidence coefficient of 0.95 given.

in this experiment is below 80%. When emulating larger networks, however, either more workers have to be used or a larger dilation factor has to be chosen. Otherwise there are not enough free CPU cycles to compensate for peaks in the CPU usage, which otherwise will bias the outcome of the experiment.

7.4.3. Lessons Learned

To ease the setup of the large-scale experiment we first used MaxiNet in virtual machines (VMs) where each physical worker hosted one VM. We used Linux KVM as the virtualization environment. KVMs network virtualization module, however, is restricted to only one RX/TX queue per network interface. This led to an uneven CPU utilization inside the VMs and thus limited the scalability of MaxiNet. We assume that the same effect will also occur when using physical machines with network interfaces limited to one RX/TX queue. We thus recommend to use network interfaces with multiple RX/TX queues to reduce the required dilation factor.

Choosing the right dilation factor is crucial for the emulation. When the dilation factor is chosen too large, the emulation takes unnecessarily long to complete, and by choosing it too low, the results of the emulation are distorted by limitations of the physical network or by the limited processing power of the workers. Unfortunately, the required dilation factor strongly depends on the used virtual topology, the amount of traffic and the software running at the emulated hosts. This makes it hard to give a general statement about the required dilation factor. To find out the smallest possible dilation factor we started with a high dilation factor and decreased it subsequently as long as the result of our experiment did not change.

7.5. NetSLS: A MaxiNet Application

As a showcase for MaxiNet, we now show how to create a new simulation tool to evaluate ideas that jointly solve the job and flow scheduling problem for big-data applications. The tool combines the Yarn Scheduler Load Simulator with MaxiNet. With this combination, the interdependency between the network and the jobs running on top of it can be included into the evaluation of new ideas, leveraging research on big-data applications with *joint* job and flow scheduling.

Research on accelerating big-data applications can be divided into *job scheduling* and *flow scheduling*. Job scheduling focuses on the timely and spacial placement of jobs on execution units. Flow scheduling, on the other hand, concentrates on routing of flows originating from actively running jobs. Although both job scheduling and flow scheduling work on accelerating big-data applications, their view on the problem and the available information is very different.

7.5.1. Introduction

Recently, research on scheduling flows from big-data frameworks has gained a lot of attraction. Flow schedulers like Varys [CZS14] and Barrat [DKBR14] are built to increase the performance of big-data applications by optimizing the network for the running parts of the application. To this end, they use information about the mapping of flows (on the network level) to jobs (on the application level). This information is then used to route flows such that job-related metrics (like the job completion time or the makespan) are optimized.

On the other hand, some work focuses on optimizing big-data *job schedulers*. Each job has certain requirements on computing power, disc space, and communication. Based on these requirements, the job schedulers decide which part of the application is to be executed at which machine at what time. Tetris [GAK$^+$14] is an example of a modern job scheduler that optimizes average job completion time while keeping a

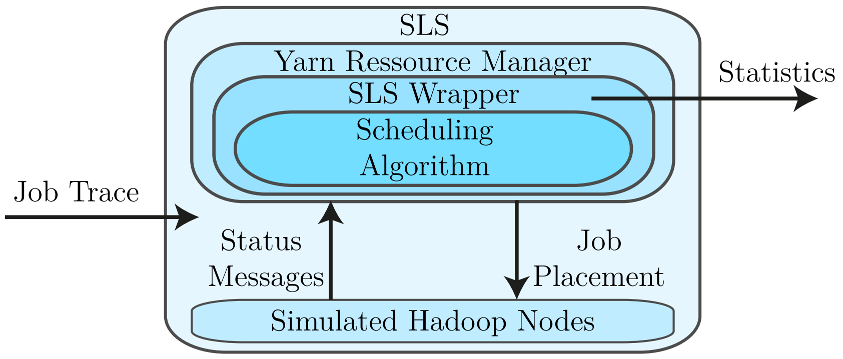


Figure 7.14.: Schematic view of SLS.

certain level of fairness between jobs.

So far, only little work has been done on combining job scheduling and flow scheduling by creating a job scheduler that is network-aware or by creating a feedback loop between flow and job scheduler. One reason for the absence of such an integration is that it is very complicated to evaluate. Modeling the exact behavior of the interaction between distributed applications and an interconnecting network is a complex task. To create trustable results, a testbed of reasonable size is required, leading to very high costs. To overcome the hurdle, the Yarn Scheduler Load Simulator[7] (SLS) has been built. SLS is a tool for simulating Hadoop clusters to predict the behavior of new job scheduling algorithms. SLS, however, does not model the interconnecting network, which makes it the wrong tool for evaluating network-related scheduling features.

In this work, we extend SLS by a highly realistic network model making it possible to conduct research in combined job and flow scheduling. SLS is used to simulate new scheduling algorithms on real world traces of big-data jobs. Whenever the scheduler decides to start a new job on a set of machines, the network emulator emulates the data transmissions corresponding to the job.

7.5.2. Yarn Scheduler Load Simulator

SLS is built to mimic a running Hadoop cluster to the Resource Manager (the entity running the scheduling algorithm). This allows the execution of unmodified Hadoop-scheduling algorithms in the simulated environment.

To benchmark a scheduling algorithm using the Resource Manager, a job trace is given as input to SLS. A job trace is a record of various parameters and statistics of jobs from a real Hadoop cluster. Among other data, a trace contains statistics of every run job and how much data was generated (and transferred) by the job. Job traces can be replayed inside SLS.

[7]`http://hadoop.apache.org/docs/r2.3.0/hadoop-sls/SchedulerLoadSimulator.html`

Fig. 7.14 shows a high-level view of SLS. SLS simulates Hadoop by running an unmodified instance of the Resource Manager. The Resource Manager controls a set of simulated Hadoop nodes by assigning jobs and monitoring the cluster. SLS inserts a shim layer between the Resource Manager and its scheduling algorithm. This layer is solely used to collect statistics about the jobs and the scheduler itself. In contrast to a real Hadoop cluster, in SLS there are no nodes actually computing map or reduce tasks. On arrival of a task assignment, nodes wait a certain time (as specified in the job trace) and report back the successful execution of the task.

7.5.3. Adding a Realistic Data-Center Model

We extend SLS to *NetSLS* by adding actual data transmissions to the simulated tasks. In Hadoop, processing a task consists of the three sequential phases fetching data, processing data, and storing data. Both fetch and store requires to access data. Depending on the location of the data, this involves the network. To model the network interconnecting the Hadoop nodes, NetSLS uses MaxiNet.

Fig. 7.15 shows how NetSLS works. On startup, NetSLS fetches the cluster topology from a file supplemental to the job trace and sets up MaxiNet to emulate the network. For each simulated Hadoop node N_i, NetSLS emulates a corresponding node, $\tilde{N}_i$ using MaxiNet. Whenever the scheduler decides to execute a task on N_i, NetSLS looks up information about data transfers from the job trace and initiates the corresponding transfers at $\tilde{N}_i$. The completion of the transfer is subsequently reported from $\tilde{N}_i$ to N_i which, in turn, reports a successful completion to the job scheduler when the three phases of the job have finished.

As MaxiNet emulates an OpenFlow-capable SDN, an OpenFlow controller manages the network. The controller is external to NetSLS, thus any OpenFlow controller can be used to control the behavior of the network. The flow scheduler resides in the OpenFlow controller. Being a part of the controller, it has access to any information present at the controller. To exchange information with the job scheduler, a custom interface between the job and flow scheduler can be established.

For evaluating novel ideas under different network loads we also integrated DCT^{2}Gen, a traffic generator for highly realistic data-center traffic presented in detail in Chapter 8. In NetSLS, DCT^{2}Gen creates background traffic mimicking cross traffic from other applications.

By turning SLS into NetSLS, novel job and flow scheduling algorithms can be tested using little time and effort. Along with a job trace and a topology description, the only two inputs to NetSLS are a job scheduling algorithm and an OpenFlow controller implementing the desired flow scheduling algorithm. NetSLS will then simulate the interplay between the flow and job scheduler on the job trace, leading to fast and realistic evaluation of novel scheduling algorithms without requiring an expensive testbed or a custom simulation.

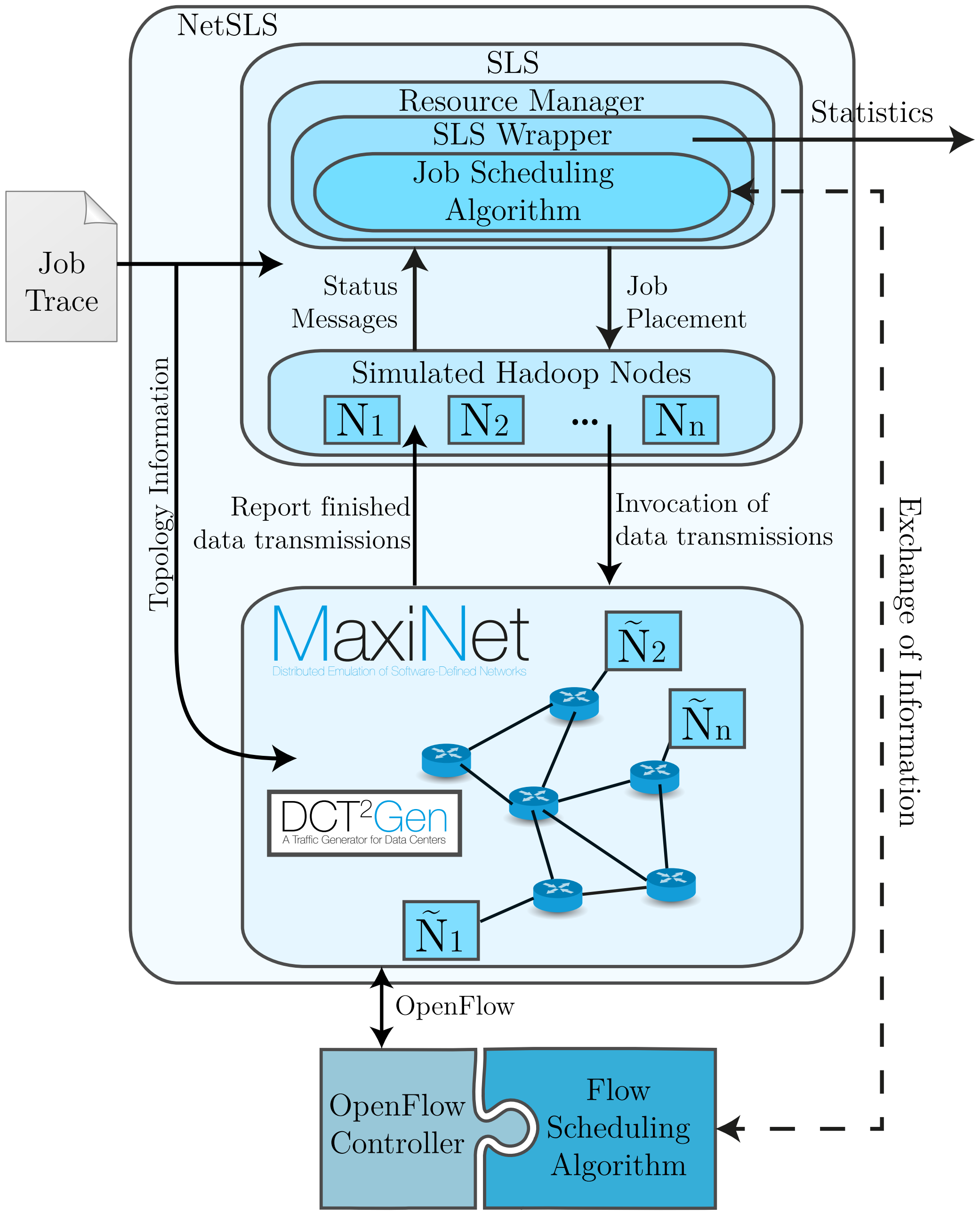


Figure 7.15.: Schematic view on NetSLS.

7.6. Conclusion

With MaxiNet it is possible to emulate data center topologies at scale in a reasonable amount of time using a cluster of physical machines for the computations. This, together with the traffic generator presented in the next chapter, opens the door for realistic evaluation of novel data-center routing protocols through emulation that can be used for rapid prototyping evaluations. We could show that the physical resources required for the emulation of a mid-sized data center are very low even when using acceptable time dilation factors. Our whole emulation environment consisted of old Intel Xeon processors. It can be assumed that by using state-of-the-art hardware even larger data centers can be emulated without using more physical resources or increasing the dilation factor.

8 DCT^{2}Gen: A Traffic Generator for Data Centers

Only little is publicly known about traffic in non-educational data centers. Recent studies made some knowledge available, which gives us the opportunity to create more realistic traffic models for data-center research. We used this knowledge to create the first publicly available traffic generator that produces realistic traffic between hosts in data centers of arbitrary size. This traffic generator was subsequently used to generate the traffic we used for the evaluations in Chapter 5 and Chapter 7.

We characterize traffic by using six probability distribution functions and concentrate on the generation of traffic on flow-level. The distribution functions are easily exchangeable to enable using up-to-date traffic characteristics whenever new data is available from publications or own experiments. Moreover, in data centers, traffic between hosts in the same rack and hosts in different racks have different properties. We model this phenomenon, making our generated traffic very realistic. We carefully evaluated our approach and conclude that it reproduces these characteristics with accuracy.

8.1. Introduction

Traffic traces from data-center networks are very rare. This leads to problems when evaluating new networking ideas for data centers because it is not possible to find proper input. We propose a method to generate realistic traffic for arbitrarily sized data centers from only a set of statistical properties of data-center traffic. This enables us to generate traffic for networks where only limited information is available.

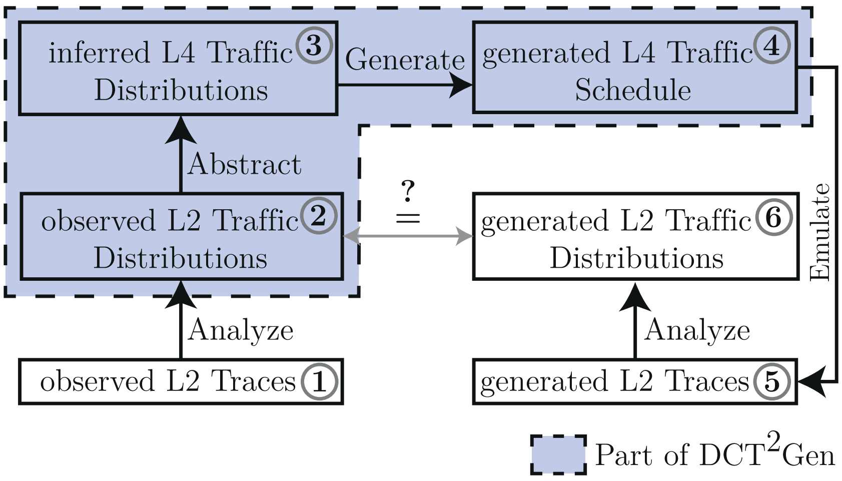

inferred L4 Traffic Distributions 3
Generate
generated L4 Traffic Schedule 4
Abstract
observed L2 Traffic Distributions 2
?
=
generated L2 Traffic Distributions 6
Emulate
Analyze
observed L2 Traces 1
Analyze
generated L2 Traces 5
Part of DCT^{2}Gen

Figure 8.1.: High-level overview of DCT^{2}Gens work flow.

Recent studies [KSG$^+$09, BAM10] investigated traffic patterns in today's data centers on flow level. They gave a detailed statistical description for both the traffic matrices and the flows present on Layer 2 in data centers. These studies were the first to give a detailed insight into the communication patterns of commercial data centers and reported that different parts of the traffic matrix have different statistical properties. This is due to software tuned for running in data centers (like Hadoop [Apa]). Such software tries to keep as much traffic in the same rack as possible to achieve a higher throughput and lower latency. We call this property *rack-awareness*.

This chapter proposes the Data Center TCP Traffic Generator (DCT^{2}Gen) which takes a set of Layer 2 traffic descriptions and uses them to generate Layer 4 traffic for data centers. When the generated Layer 4 traffic is transported using TCP, it results in Layer 2 traffic complying with the given descriptions. With the Layer 4 traffic at hand, TCP dynamics can be included into the evaluation of novel networking ideas for data centers with pre-described properties of Layer 2 traffic without having to know the exact applications running in the data center. This allows using realistic TCP traffic patterns in experiments conducted at testbeds or network emulations where real network stacks are used. Our generator is highly realistic; e.g. it reflects rack-awareness of typical data-center applications, which enables highly realistic evaluation of novel data-center ideas.

The work flow required to generate artificial TCP traffic and to prove its validity is depicted in Figure 8.1. First, Layer 2 traces from the targeted data-center are collected (1). Then, these traces are analyzed to obtain a set of probability distributions describing the traffic (2). These distributions include the number of communication partners per host, the flow sizes, the sizes of traffic matrix entries, and others. From the observed Layer 2 traffic distributions (2) we infer the underlying Layer 4 traffic distributions (3). Using these Layer 4 distributions, we generate a Layer 4 traffic schedule (4). This schedule describes for each host when to send how much *payload* to which other host in the data center. We claim that by executing our calculated schedule, the resulting traffic on Layer 2 (5) has the same stochastic properties as the original Layer 2 traffic traces (1). To prove that, we are using a network emulation to execute the computed Layer 4 schedule. We capture the resulting traffic at Layer 2 (5) from the emulation and analyze its statistical properties (6) to show that these are the same for both the original Layer 2 traces (1) and the generated traces (5).

To compute the Layer 4 traffic schedule, we do not even need to know the Layer 2 traces (1). It is sufficient to know the Layer 2 traffic distributions (2). Also, even if for a data center it is possible to directly obtain Layer 4 traffic distributions (3), DCT^{2}Gen serves as a useful tool to generate Layer 4 schedules (4) from this data. Because even then, it is still necessary to generate traffic matrices from the data and create TCP flows complying with the given distributions.

Finding a Layer 4 traffic schedule (4) is a challenging task because of the bidirectional nature of TCP. TCP is the most common Layer 4 protocol. For this work, we assume that all Layer 4 traffic is transported using TCP and that all TCP connections are *non-interactive* (i.e., payload is only transported into one direction). In TCP, each flow transferring payload between a source s and a destination d also creates a flow of acknowledgments (ACKs) from d to s. The size of this ACK flow is roughly proportional to the size of transferred payload. Thus, half of all flows in the schedule cannot be scheduled arbitrarily. The properties of these flows depend on the other half of flows. This poses a lot of interesting problems that we solved when creating our traffic generator.

DCT^{2}Gen is open source and available for download from our website[8]. We supply all necessary inputs required to generate a traffic schedule complying with the distributions reported in [KSG$^+$09, BAM10]. We emphasize that DCT^{2}Gen is independent of these two studies and thus can keep up with the ever changing properties of data-center traffic. Whenever new studies about data-center traffic are published, their results can be used with DCT^{2}Gen, too. To do so, solely the probability distributions (which are given as step functions and are part of the input) have to be replaced. Although we explain, in some parts of this chapter, some of our design choices with the behavior of a Map-Reduce-style workload (backlogged data transmissions, almost all TCP connections unidirectional), we strongly believe that our assumptions on the traffic in data centers do not depend on Map Reduce and are also valid for traffic produced by other data-center applications.

The rest of this chapter is structured as follows. In Section 8.2 we give a short overview of the landscape of traffic generators. Section 8.3 discusses traffic properties of data-center networks. These properties have to be replicated by our traffic generator whose architecture is presented in Section 8.4. Section 8.5 deals with one of the main challenges of this work: A method is described to find the distribution of the sizes of Layer 4 traffic matrix entries from the distribution of the sizes of Layer 2

[8] `https://www.cs.upb.de/?id=dct2gen`

traffic matrix entries. Section 8.6 describes the process of traffic matrix generation. Section 8.7 explains how to use these traffic matrices to create a schedule of Layer 4 traffic. In Section 8.8 we evaluate our traffic generator and conclude this chapter in Section 8.9.

8.2. Related Work

Past research has created a large number of different traffic generators, all with different aims and techniques. From our point of view, there are four key characteristics of available traffic generators:

- Flow-level vs. packet-level
- Traffic on one link only vs. traffic on a whole network
- Automatic vs. manual configuration
- Topology awareness vs. non-topology awareness

We give a short overview of each characteristic and afterwards use them to categorize existing traffic generators.

8.2.1. Flow-level vs. packet-level generators

There are traffic generators [APV04, AGE$^+$04, LSP] that output traffic on packet level, formatted due to certain communication protocols. The mix of these packets follows certain rules and probability distributions that are configurable beforehand. However, these traffic generators do not usually implement flows, i.e. packets that logically belong together and that share certain properties like source and destination addresses. A traffic generator that is flow-aware [VV09, BTI$^+$03, SKB04] always generates packets organized in flows. Flow generation is done such that the flows meet certain statistical properties.

8.2.2. Traffic on one link only vs. traffic on a whole network

The majority of existing traffic generators concentrates on generating traffic originating from one interface only. For performance evaluation of whole network topologies it is required to know the packet stream that is created by each single device in the network. As typically these streams are correlated, it is not sufficient to generate traffic for each interface separately but a traffic generator that creates correlated traffic for a whole network is required.

8.2.3. Automatic vs. manual configuration

Network traffic has various properties depending on the type of the network. To specify desired traffic properties, traffic generators can be parameterized by hand

[Hee00], automatically by feeding *traffic traces* from a real network whose traffic has to be mimicked [WAHC+06, SKB04, SSKB10], or by a combination of both [SKB04, SSKB10]. For the automatic case either algorithms are used to extract parameters from the given traffic or the given traffic itself is part of the generated traffic. In that case it is often used as background traffic that is superimposed by special traffic that has properties based on the desired test case.

8.2.4. Topology awareness vs. non topology awareness

Traffic generators can be *topology-aware.* In that case, the topology of the target network influences the traffic patterns produced by the generator. In the context of data-center networks, the traffic matrices are typically dense in intra-rack areas and coarse in inter-rack areas. Thus, a traffic generator for data center networks has to account for the placement of servers in racks.

8.2.5. Existing Traffic Generators

Harpoon [SKB04] is an open-source traffic generator that creates traffic at flow level. It creates correlated traffic between multiple endpoints and automatically derives traffic properties from supplied packet traces. Harpoon is able to generate both TCP and UDP traffic. The general concept of Harpoon is a hierarchical traffic model. Traffic between any pair of endpoints is exchanged in sessions where each session consists of multiple file transfers between that pair of hosts. Sessions can either be TCP or UDP. Harpoon can be parametrized in terms of inter-arrival and holding times for sessions, flow-sizes, and the ratio between UDP and TCP. These parameters are automatically derived from supplied packet traces. As Harpoon is not topology-aware it cannot be used to replicate the special properties of data-center traffic.

Ref. [SSKB10] proposes a flow-level traffic generator for networks. It uses a learning algorithm that automatically extracts properties from packet traces. That work focuses on generation of traffic from different applications each with different communication patterns. To this end, *Traffic Dispersion Graphs* are used to model the communication structure of applications. The generator reproduces these communication structures accurately but is less accurate in modeling the properties of flows. In addition, this traffic generator does not capture any structural properties of the traffic matrix.

The Internet Traffic Generator [APV04] and its distributed variant D-ITG [AGE+04] focus on traffic generation on packet-level. Both generate a packet stream that can be configured in terms of the inter-departure time and the packet size. A similar traffic generator is presented in [BTI+03]. It generates traffic on flow level for a single internet backbone link.

Swing [VV09] is a closed-loop traffic generator that uses a very simple model for generating traffic on packet level. Swing aims at reproducing the packet inter-arrival rate and its development over time. Packets are logically organized in flows. However, Swing only generates a packet trace for a single link.

Up to now, there exists no traffic generator that computes a schedule of TCP payload transmissions that can be used to produce Layer 2 traffic that complies with statistical traffic properties given beforehand. DCT^{2}Gen is the first generator to compute such a schedule.

8.3. Traffic Properties

To describe and generate traffic, DCT^{2}Gen uses several stochastic traffic properties. Some of these properties are observed from L2 traces that are given as input, some of them are properties of inferred Layer 4 traffic. The traffic description hinges on how flows behave inside the network.

Throughout the chapter, the term *flow* describes a series of packets on Layer 2 between the same source and destination that logically belong together. We distinguish two types of flows. A *payload flow* is a flow which transports payload from source to destination – looking from Layer 4, it transports TCP data packets. Since we assume non-interactive TCP traffic, a payload flow does not include any acknowledgments. Acknowledgments are sent in separate *ACK flows*, which only include TCP ACK segments but no data. In consequence, each TCP connection results in two flows. The structure of these flows is captured by *traffic matrices* described in the following.

8.3.1. Traffic Matrices

A traffic matrix (TM) describes the *amount of data* in bytes (not number of flows) that is transferred between a set of end hosts in a fixed time interval. The entry (i, j) of a TM tells how much data is sent from server i to server j in the considered time interval. This amount of data is transferred by a (possibly large) number of flows, including payload flows and ACK flows. In the following, we will discuss the characterization of traffic at flow level, i.e., how the amount of data captured in the traffic matrix is decomposed into TCP flows.

We distinguish several types of traffic matrices. The primary one is the traffic matrix describing the *observed,* actual traffic on Layer 2; we denote this matrix as $\mathrm{TM}^{(\mathrm{obs})}$. The next matrix corresponds to the *generated* traffic on Layer 2 that is a result of DCT^{2}Gen (compare Box 5 in Figure 8.1). Generated L2 traffic is described by the traffic matrix $\mathrm{TM}^{(\mathrm{gen})}$.

Layer 2 traffic is juxtaposed to Layer 4 traffic. Layer 4 traffic is usually not observed (or, at least, not reported in publications) but only generated. We have to distinguish between the *payload* traffic matrix describing the actual data flows and the *acknowledgement* traffic matrix for the flows containing only acknowledgement packets; they are called $\mathrm{TM}^{(\mathrm{PL})}$ and $\mathrm{TM}^{(\mathrm{ACK})}$, respectively. Since the payload flows from i to j give raise to the acknowledgement flows from j to i, these two matrices are interrelated:

$$\mathrm{TM}^{(\mathrm{ACK})}(i,j) = \beta \cdot \mathrm{TM}^{(\mathrm{PL})}(j,i)$$

for some value β to be discussed in Section 8.6.

Moreover, a Layer 2 traffic matrix is the sum of a Layer 4 payload TM and a Layer 4 ACK TM plus overhead; Section 8.5 discusses the overheads involved here.

In addition to the layer, traffic matrices also reflect the traffic structure inside a data center. Because of rack-aware applications, traffic inside a rack has different stochastic properties than that between racks – we reflect these differences by separate stochastic distribution functions for the *intra-rack* and the *inter-rack* parts of a traffic matrix.

For either part, we have to describe first the stochastic distribution of the number of nodes a given node i talks to (either in its own or in any other rack). Second, we need a stochastic distribution to describe the amount of bytes that is transferred from i to j, for each node j that i talks to; the intra- and inter-rack cases will in general have different distribution functions.

In summary, we need stochastic distributions separately for (a) the cases of observed and generated Layer 2 traffic and for payload and ACK traffic on Layer 4, (b) the distinction between intra- and inter-rack traffic, and (c) the description of number of communication partners vs. number of transferred bytes. This results in $4 \cdot 2 \cdot 2 = 16$ distribution functions so far.

8.3.2. Flow Sizes

The *flow size* denotes the number of bytes transported by a flow including all protocol overhead. An entry of a traffic matrix describes how much traffic is exchanged in total between a pair of nodes in a given time but it specifies neither number nor size of the individual flows transporting this traffic. The *flow-size distribution* specifies how likely a flow of a certain size occurs on Layer 2.

We distinguish between the flow-size distribution of payload flows, the flow-size distribution of ACK flows, *and* the flow-size distribution of both payload and ACK flows combined. Figure 8.2 depicts the relationship between payload flows, ACK flows, and all flows on Layer 2. Each flow transporting payload from node i to node j implies an ACK flow from node j to node i. The size of the ACK flow depends on the size of the corresponding payload flow. Hence, the payload-size distribution implies a certain ACK-size distribution and the convolution of both distributions is the flow-size distribution on Layer 2.

To summarize, we distinguish between three different flow-size distributions.

- The *payload-size distribution* specifies how likely a flow of a certain size occurs on Layer 2 which results from a payload flow on Layer 4.

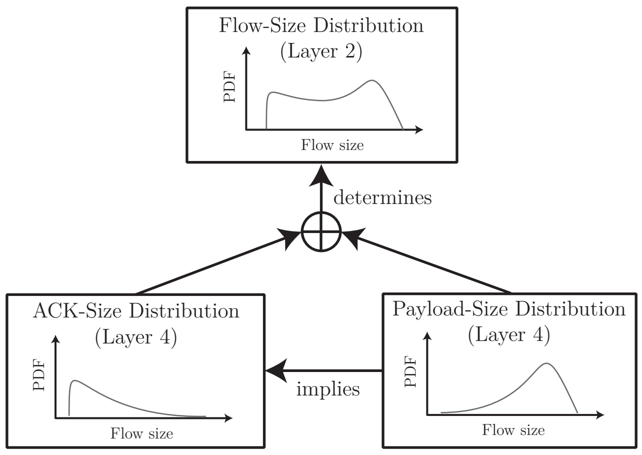


Figure 8.2.: Relationship between the observable flow-size distribution function at Layer 2 (top), the distribution function of ACK sizes (bottom left) and causal payload sizes (bottom right).

- The *ACK-size distribution* specifies how likely a flow of a certain size occurs on Layer 2 which results from an ACK flow on Layer 4.
- The *flow-size distribution* specifies how likely a flow of a certain size occurs on Layer 2.

Looking from Layer 2, we cannot tell if a flow is a payload flow or an ACK flow. In the process of traffic generation, DCT^{2}Gen takes the flow-size distribution of all flows and computes the corresponding payload-size distribution. To be able to infer flow sizes at Layer 4 (from the flow-size distribution) we need to assume that all TCP sessions in the data center are non-interactive. For data centers running mostly Map-Reduce workload this assumption is true for most of the flows.

8.3.3. Flow Inter-Arrival Time

The flow inter-arrival time distribution describes the time between two subsequent flows arriving at the network. Together with the flow-size distribution, the distribution of the flow inter-arrival time specifies the distribution of the total amount of traffic for a given time interval. This amount of traffic must match the total traffic specified by the corresponding TM. Otherwise, not enough (or too many) flows exist, which means it is not possible to use these flows to create a TM with the desired properties.

8.3.4. Nomenclature

We shall indicate the distribution functions (*not* the random variables) as follows:

- N represents the number of communication partners per node, B represents the total bytes exchanged between a pair of nodes, S represents the flow size and

Table 8.1.: Overview of the distribution functions.

Distributions		observed	generated	inferred at Layer 4	
		Layer 2	Layer 2	PL	ACK
Bytes	intra-rack	B^{obs}_{intra}	B^{gen}_{intra}	B^{PL}_{intra}	B^{ACK}_{intra}
	inter-rack	B^{obs}_{inter}	B^{gen}_{inter}	B^{PL}_{inter}	B^{ACK}_{inter}
Number	intra-rack	N^{obs}_{intra}	N^{gen}_{intra}	N^{PL}_{intra}	N^{ACK}_{intra}
	inter-rack	N^{obs}_{inter}	N^{gen}_{inter}	N^{PL}_{inter}	N^{ACK}_{inter}
Flow Size		S^{obs}	S^{gen}	S^{PL}	S^{ACK}
Flow Inter-Arrival time		IAT^{obs}	IAT^{gen}	IAT^{PL}	IAT^{ACK}

IAT represents flow inter-arrival times.

- The subscript specifies either the intra- or inter-rack case, if needed.
- The superscript specifies the case of observed or generated (on Layer 2) vs. payload or ACK (on Layer 4) traffic.

Table 8.1 summarizes all the stochastic distribution functions that we use in the remainder of the chapter.

8.4. Architecture of DCT^{2}Gen

We generate a schedule of Layer 4 traffic that specifies at which time how much *payload* has to be transmitted from which source node to which destination node. When transported using TCP, the generated Layer 2 traffic on the network shall have the same properties as the observed Layer 2 traffic. Many possible schedules with this property exist. We aim at finding one of these with the following approach.

First, a $TM^{(PL)}$ is generated. To this end, we need to infer the distribution of payload bytes exchanged by a pair of nodes for the inter-rack case (B^{PL}_{inter}) from the observed distribution of total bytes exchanged by a pair of nodes for the inter-rack case (B^{obs}_{inter}) and the distribution of payload bytes exchanged by a pair of nodes for the intra-rack case (B^{PL}_{intra}) from the observed distribution of total bytes exchanged by a pair of nodes for the intra-rack case (B^{obs}_{intra}). From these distributions, along with the distribution of the number of communication partners per node for the inter-rack case (N^{obs}_{inter}) and the distribution of the number of communication partners per node for the intra-rack case (N^{obs}_{intra}), a $TM^{(PL)}$ can be generated (for details, see Section 8.6). The next step is to assign flows to all non-zero $TM^{(PL)}$ entries. For this task, we need to infer the distribution of payload-flow sizes (S^{PL}) from the observed distribution of flow sizes on Layer 2 (S^{obs}). The former describes the distribution of the sizes of payload flows that (together with the implied S^{ACK}) generates the given flow-size distribution on Layer 2 (Figure 8.2). Using S^{PL}, a set of payload flows is generated which are mapped to the non-zero $\mathrm{TM}^{(\mathrm{PL})}$ entries in a subsequent step (Section 8.7).

At the end of this process we know how many payload bytes to send from which node to which other node in TCP sessions such that it holds that: $\mathrm{B}^{\mathrm{gen}}_{\mathrm{intra}}$ equals $\mathrm{B}^{\mathrm{obs}}_{\mathrm{intra}}$, $\mathrm{B}^{\mathrm{gen}}_{\mathrm{inter}}$ equals $\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$, $\mathrm{N}^{\mathrm{gen}}_{\mathrm{intra}}$ equals $\mathrm{N}^{\mathrm{obs}}_{\mathrm{intra}}$, $\mathrm{N}^{\mathrm{gen}}_{\mathrm{inter}}$ equals $\mathrm{N}^{\mathrm{obs}}_{\mathrm{inter}}$, $\mathrm{S}^{\mathrm{gen}}$ equals $\mathrm{S}^{\mathrm{obs}}$, and $\mathrm{IAT}^{\mathrm{gen}}$ equals $\mathrm{IAT}^{\mathrm{obs}}$.

The modular design of our traffic generator can be seen in Figure 8.3. It consists of the five different modules *Deconvolver*, *Payload Extractor*, *Traffic Matrix Generator*, *Flowset Creator*, and *Mapper*. This section gives a short description of each single module. In the subsequent sections, complex modules (Deconvolver, Traffic Matrix Generator, Mapper) are explained in detail.

8.4.1. Deconvolver

The *Deconvolver* takes the observed distribution of total bytes exchanged by a pair of nodes for the intra-rack case ($\mathrm{B}^{\mathrm{obs}}_{\mathrm{intra}}$) and the observed distribution of total bytes exchanged by a pair of nodes for the inter-rack case ($\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$) as inputs. From these, it computes the distribution of payload bytes exchanged by a pair of nodes for the intra-rack case ($\mathrm{B}^{\mathrm{PL}}_{\mathrm{intra}}$) and the distribution of payload bytes exchanged by a pair of nodes for the inter-rack case ($\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$), which enable us to generate $\mathrm{TM}^{(\mathrm{PL})}$. As the name suggests, the Deconvolver uses a deconvolution technique which is explained in detail in Section 8.5.

8.4.2. Payload Extractor

We need to compute a set of payload flows that, together with the implied ACK flows, generate flows on Layer 2 which comply with $\mathrm{S}^{\mathrm{obs}}$ (Figure 8.2). Flows on Layer 2 are the union of payload flows and ACK flows. As we are only given $\mathrm{S}^{\mathrm{obs}}$ we need to infer S^{PL} (which itself implies a certain $\mathrm{S}^{\mathrm{ACK}}$). S^{PL} is computed in the *Payload Extractor*.

For the Payload Extractor to work, we need to assume that the ratio of payload packets to ACK packets in TCP is fixed at a value r. We substantiate this assumption in Section 8.5.2 and use r to calculate the ratio of payload bytes to ACK bytes $\beta = \frac{|\mathrm{ACK}|}{|\mathrm{PAY}|} \cdot \frac{1}{r}$, where $|\mathrm{ACK}|$ is the size of an ACK packet and $|\mathrm{PAY}|$ is the size of a payload packet. $|\mathrm{PAY}|$ is the MTU of the network (which in our case was 1500) plus the size of an Ethernet header (14 bytes). $|\mathrm{ACK}| = 66$ because TCP Cubic, which is default in Linux, tends to use the TCP Time Stamp option resulting in an ACK packet size of 20 bytes IP header, 32 bytes TCP header and 14 bytes Ethernet header. Once concrete values r and β are known, the Payload Extractor transforms $\mathrm{S}^{\mathrm{obs}}$ into S^{PL}.

Algorithm 3 is used to infer S^{PL} from $\mathrm{S}^{\mathrm{obs}}$. Let $\mathrm{Pr}^{\mathrm{obs}}(x)$ be the probability (according to $\mathrm{S}^{\mathrm{obs}}$) that the size of a flow is x and $\mathrm{Pr}^{\mathrm{PL}}(x)$ the probability (according to S^{PL}) that the size of a payload flow is x. $\mathrm{ACK}(x) = 66 \cdot \left\lceil \frac{x}{\mathrm{MSS} \cdot r} \right\rceil$ is the size of an ACK flow acknowledging the receipt of x payload bytes. MSS is the maximum segment size which in our setup was 1448. To convert $\mathrm{S}^{\mathrm{obs}}$ into S^{PL}, the algorithm iterates over all flow sizes in descending order and removes the corresponding ACK flow from S^{PL}. This works because it always holds that the ACK-flow size is smaller than or equal to the corresponding payload-flow size.

Algorithm 3 Transform $\mathrm{S}^{\mathrm{obs}}$ to S^{PL}.

```
1: **Algorithm** InferPayloadSize( $Pr^{\mathrm{obs}}(\cdot)$ ):
2: $\mathrm{Pr}^{\mathrm{PL}}(\cdot) \leftarrow Pr^{\mathrm{obs}}(\cdot)$
3: **for each** flow size $x$ in decreasing size **do**
4:     $\mathrm{Pr}^{\mathrm{PL}}(\mathrm{ACK}(x)) \leftarrow \mathrm{Pr}^{\mathrm{PL}}(\mathrm{ACK}(x)) - \mathrm{Pr}^{\mathrm{PL}}(x)$
5: **end for**
6: $\mathrm{Pr}^{\mathrm{PL}}(\cdot) \leftarrow \mathrm{Pr}^{\mathrm{PL}}(\cdot) / \sum_x \mathrm{Pr}^{\mathrm{PL}}(x)$        $\triangleright$ normalizes $\mathrm{Pr}^{\mathrm{PL}}(\cdot)$
7: **return** $\mathrm{Pr}^{\mathrm{PL}}(\cdot)$
```

8.4.3. Traffic Matrix Generator

From the outputs of the Deconvolver ($\mathrm{B}^{\mathrm{PL}}_{\mathrm{intra}}$ and $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$) together with the observed distribution of inter-rack communication partners per node for the intra-rack case ($\mathrm{N}^{\mathrm{obs}}_{\mathrm{intra}}$) and the observed distribution of inter-rack communication partners per node for the inter-rack case ($\mathrm{N}^{\mathrm{obs}}_{\mathrm{inter}}$) , the *Traffic Matrix Generator* creates a $\mathrm{TM}^{(\mathrm{PL})}$. This $\mathrm{TM}^{(\mathrm{PL})}$ specifies payloads such that, when exchanged using TCP, this results in a $\mathrm{TM}^{(\mathrm{gen})}$ having the same statistical properties as $\mathrm{TM}^{(\mathrm{obs})}$. Matrix generation is explained in detail in Section 8.6. After the $\mathrm{TM}^{(\mathrm{PL})}$ has been calculated, the payload bytes exchanged between any pair of hosts are divided into single payload flows.

8.4.4. Flowset Creator

Flows are generated by the *Flowset Creator*. The Flowset Creator gets S^{PL} from the Payload Extractor, $\mathrm{IAT}^{\mathrm{obs}}$, and a target traffic volume (which is the sum over all entries of the $\mathrm{TM}^{(\mathrm{PL})}$ generated in the previous step). It outputs a set of flows whose flow sizes sum up to the target traffic volume.

To this end, the Flowset Creator creates payload flows with sizes distributed according to S^{PL}. When the payload flows are transferred over the network, the generated Layer 2 flows have to comply with $\mathrm{IAT}^{\mathrm{obs}}$. For this task, we need to infer $\mathrm{IAT}^{\mathrm{PL}}$ from $\mathrm{IAT}^{\mathrm{obs}}$ such that $\mathrm{IAT}^{\mathrm{PL}}$ and $\mathrm{IAT}^{\mathrm{ACK}}$ result in $\mathrm{IAT}^{\mathrm{obs}}$. However, the resolution of the data provided by [KSG$^+$09] (or any other sources we are aware of) for $\mathrm{IAT}^{\mathrm{obs}}$ is so low that we could not draw any conclusions on $\mathrm{IAT}^{\mathrm{PL}}$. This is why we use $\mathrm{IAT}^{\mathrm{obs}}$ as an approximation for $\mathrm{IAT}^{\mathrm{PL}}$; this is a rough approximation, however, we do not have any better data at hand.

The generated flows only add up to the target traffic volume if $\mathrm{IAT}^{\mathrm{PL}}$ and S^{PL} are chosen such that the sum of all generated flow sizes matches the traffic volume of the

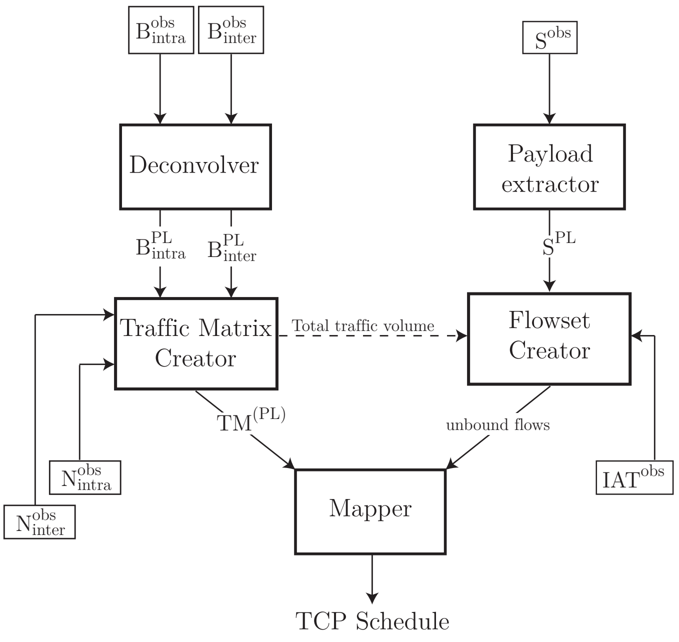


Figure 8.3.: Architectural overview of DCT^{2}Gen.

generated TM$^{(PL)}$. If this is not the case, we scale the inter-arrival times by a linear factor to generate more or less flows depending on the situation.

8.4.5. Mapper

In the last step of our traffic generator the flows generated by the Flowset Creator are mapped to the source-destination pairs specified by the TM$^{(PL)}$ as computed by the Traffic Matrix Creator. This mapping is done by the *Mapper* which uses a newly developed assignment strategy. The Mapper and our mapping strategy are explained in detail in Section 8.7.

8.5. Deconvolving Traffic Matrix Entries

8.5.1. Problem description

The outcome of the traffic generation process is a schedule of payload transmissions specifying when which amount of payload is sent from one machine to another. In TCP, whenever a certain amount of payload is transferred over the network, this payload flow causes a second flow called ACK flow. The ACK flow acknowledges the correct reception of the payload flow but does not transmit any payload itself.[9] However, it adds traffic to the network. The traffic seen on Layer 2 is the sum of the payload flows and the ACK flows. We only have information from the observed

[9]Data exchange between two hosts over different TCP connections is of course supported by DCT^{2}Gen.

$\text{TM}^{(\text{obs})}$ but we want to build the inferred $\text{TM}^{(\text{PL})}$. To this end, we need to compute $\text{B}^{\text{PL}}_{\text{intra}}$ from $\text{B}^{\text{obs}}_{\text{intra}}$ and $\text{B}^{\text{PL}}_{\text{inter}}$ from $\text{B}^{\text{obs}}_{\text{inter}}$. Or, put in other words, we need to infer the Layer 4 distributions of non-zero TM entry sizes from the corresponding Layer 2 distributions. This section shows how to do that.

The individual non-zero traffic matrix entry sizes of a $\text{TM}^{(\text{obs})}$ can be expressed as random variables $Z = X + Y$ where X and Y specify the amount of outgoing payload bytes (X) and the amount of outgoing ACK bytes (Y). The distribution of Z is given as $\text{B}^{\text{obs}}_{\text{inter}}$ resp. $\text{B}^{\text{obs}}_{\text{intra}}$ and it is the *linear convolution* of the distributions of X and Y, which we neither know.

When assuming that the ratio between payload packets and ACK packets is a constant r and by ignoring the facts that a) the TCP protocol adds overhead to each single packet and b) TCP uses additional messages for establishing and terminating sessions (TCP handshake), we can write Z as

$$Z = X + \beta Y$$

where $\beta = \frac{|\text{ACK}|}{|\text{PAY}|} \cdot \frac{1}{r}$ as before. We treat X and Y as independent and identically distributed (iid) random variables although X and Y might be correlated. However, we assume that this correlation is very low for the kind of software that runs in a data center.

RFC 1122 [Bra89] states that for TCP is is not required to acknowledge the correct receipt of every single payload packet. Instead, one ACK can acknowledge multiple payload packets at once. This is called *delayed ACK*. However, the acknowledgement of payload must not be arbitrarily delayed. According to RFC 1122, the delay must be less than 0.5 seconds and there has to be at least one ACK packet for every second payload packet. Unfortunately, TCP implementations in modern Operating Systems do not follow this specification strictly. In experiments with Linux Kernel 3.11, e.g., the number of outstanding unacknowledged payload packets ranged up to 16 for a backlogged 1 GByte flow over a 1 Gbps link.

In the following, we show that for the TCP connections transferring most bytes in a data center, the ratio between payload packets and ACK packets is on average $r \approx 2.5$.

8.5.2. Estimating Payload to ACK Ratio

We now show that r is nearly constant in our data-center scenario. Clearly, the value of r depends on a) the available link speeds, b) the TCP implementation, and c) the distribution of flow sizes. We want to calculate r for payload-flow sizes distributed according to S^{PL}. To determine r, we have to compute a $\text{TM}^{(\text{PL})}$ and divide its non-zero entries into payload flows. Then, this traffic can be emulated using a network emulator and the resulting r value can be observed. However, we know neither $\text{TM}^{(\text{PL})}$ nor S^{PL}. To compute both we need to know r first, which means we are stuck in a

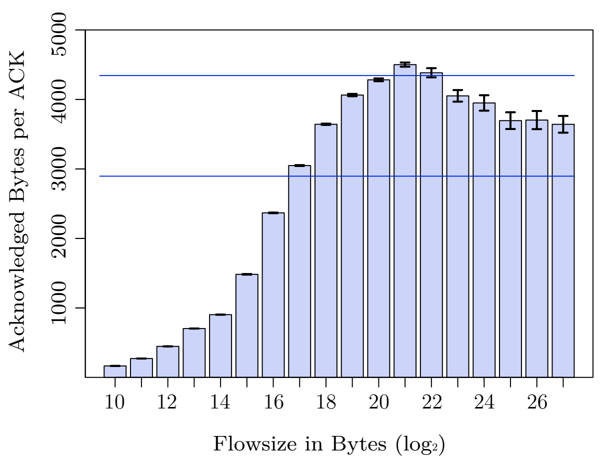


Figure 8.4.: Number of acknowledged payload bytes per ACK packet plotted over different flow sizes. Error bars show confidence intervals of level 95%.

vicious circle.

A pragmatic way of breaking the circle is to use $\mathrm{TM}^{(\mathrm{obs})}$ as an approximate for $\mathrm{TM}^{(\mathrm{PL})}$, divide the non-zero entries into payload flows distributed according to $\mathrm{S}^{\mathrm{obs}}$ and emulate this traffic to determine r. This of course has a negative influence on the accuracy of the estimated r value. However, since a) there is no reason why the traffic matrix should have a large effect on r (as long as the payload sizes stay the same), and b) $\mathrm{S}^{\mathrm{obs}}$ and S^{PL} are not too way off, the introduced error will be acceptable.

To estimate r we calculate a $\mathrm{TM}^{(\mathrm{obs})}$, generate TCP traffic with payload sizes distributed according to $\mathrm{S}^{\mathrm{obs}}$, and emulate this traffic using a network emulator. In a subsequent step we analyze the generated ACK packets to approximate r.

We generated 60 s of TCP traffic for a data center consisting of 1440 servers organized in 72 racks of 20 servers each, interconnected in a Clos-like topology (for details, see Section 8.8). We emulate this data center with MaxiNet as presented in Chapter 7. We use a time dilation factor of 300 on a cluster of 12 servers equipped with Intel Xeon E5506 CPUs running at 2.16 Ghz.

On both emulated core switches we used `tcpdump` to write a trace of all packets passing the first interface.[10] In a subsequent step we analyzed all ACK flows in the trace to determine the ratio between the transferred payload and the number of ACK packets. Figure 8.4 plots the number of payload bytes (on TCP level) acknowledged by each ACK packet against the size of the 429,491 identified payload flows. The two horizontal lines mark 2896 bytes and 4344 bytes, which is one ACK packet for every second resp. every third payload packet (we used an MTU of 1500 which means the MSS was 1448). It can be seen that for each flow larger than $2^{16} \approx 65$ KB the ratio between payload packets and ACKs is between 2 and 3. For larger flows the ratio stabilizes at 2.5.

[10]For performance reasons, it was not possible to create traces at all switches or interfaces. So we decided to use the core switches to be able to see traffic from all parts of the network.

Note that the observable TCP dynamics depend on various environment characteristics. TCP always adapts to the current network situation by delaying ACKs and by enlarging or decreasing the TCP window size resulting in different data rates for the single flows. Thus, the ratio between payload packets and ACKs we found for our scenario can differ from the ratio in other network setups. In that case a different r value has to be given to DCT^{2}Gen.

8.5.3. Deconvolving TCP Traffic

Once we know r, which tells how much overhead is created by the ACK packets, we can easily calculate β as it only depends on the MTU. Using the results of [Bel03], we are now going to show how to extract the distribution of X from

$$Z = X + \beta Y$$

when β and the distribution of Z are known and X and Y are independent and identically distributed (iid). This result can then be used to to infer both $\mathrm{B}_{\mathrm{intra}}^{\mathrm{PL}}$ from $\mathrm{B}_{\mathrm{intra}}^{\mathrm{obs}}$ and $\mathrm{B}_{\mathrm{inter}}^{\mathrm{PL}}$ from $\mathrm{B}_{\mathrm{inter}}^{\mathrm{obs}}$.

Let $\mathrm{f}(t)$ denote the characteristic function of X which we want to calculate. Since the characteristic function of the sum of two independent random variables is the product of both their characteristic functions, we can write the characteristic function $\mathrm{g}(t)$ of Z/β as

$$\mathrm{g}(t) = \mathrm{f}(t)\,\mathrm{f}(\gamma t)$$

where $\beta = \frac{|\mathrm{ACK}|}{|\mathrm{PAY}|} \cdot \frac{1}{r}$ and $0 < \gamma = \frac{1}{\beta} < 1$. Due to [Bel03], we can write

$$\mathrm{f}(t) = \prod_{k=0}^{\infty} \frac{\mathrm{g}(\gamma^{2k} t)}{\mathrm{g}(\gamma^{2k+1} t)} \quad .$$

Evaluating $\mathrm{f}(t)$ on each point from the range of $g(\cdot)$ yields an approximation of the characteristic function of X. From the characteristic function, the density can be calculated by an inverse Fourier transformation.

8.5.4. Results

To ascertain that the deconvolution yields reasonable results, we now show the results of the deconvolution of $\mathrm{B}_{\mathrm{inter}}^{\mathrm{obs}}$ (as reported in [KSG$^+$09]). We set r to 2.5, thus $\beta = \frac{66}{\mathrm{MSS}} \cdot \frac{1}{2.5}$ and compute the deconvolution to retrieve $\mathrm{B}_{\mathrm{inter}}^{\mathrm{PL}}$.

Then, we compute the implied $\mathrm{B}_{\mathrm{inter}}^{\mathrm{ACK}}$ based on $\mathrm{B}_{\mathrm{inter}}^{\mathrm{PL}}$ (Figure 8.2). The function $\mathrm{ACK}(p)$ is used to compute the size (in bytes) of an ACK flow corresponding to a payload flow of size p:

$$\mathrm{ACK}(p) = 66 \cdot \left\lceil \frac{p}{\mathrm{MSS} \cdot r} \right\rceil$$

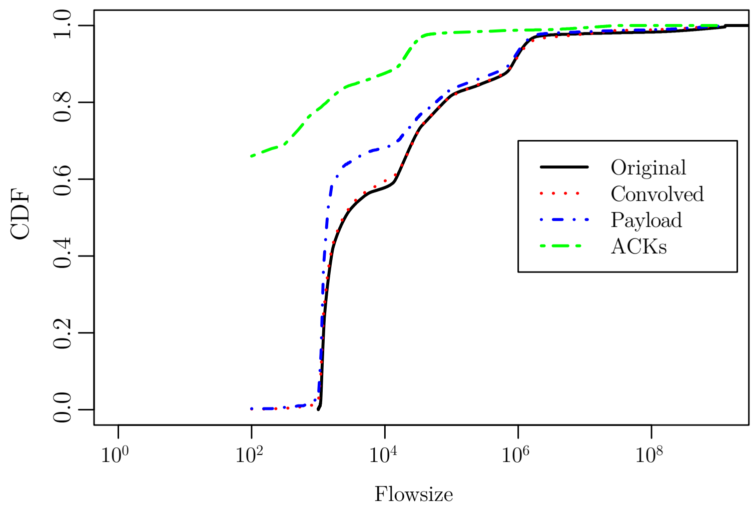


Figure 8.5.: Result of the deconvolution operation. The solid line shows $\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$ which is decomposed into $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ and the $\mathrm{B}^{\mathrm{ACK}}_{\mathrm{inter}}$. The dotted line depicts the convolution of $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ and $\mathrm{B}^{\mathrm{ACK}}_{\mathrm{inter}}$.

where in our setup MSS was set to 1448. 66 is multiplied because in TCP an ACK packet has a size of 66 bytes. We calculate the resulting $\mathrm{B}^{\mathrm{gen}}_{\mathrm{inter}}$ as the convolution of $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ and $\mathrm{B}^{\mathrm{ACK}}_{\mathrm{inter}}$ and compare it to $\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$ to ascertain that the deconvolution was successful.

Figure 8.5 shows the result of the deconvolution operation. It depicts the following four CDFs: a) $\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$ as the original CDF, b) $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ as the inferred payload-size distribution, c) $\mathrm{B}^{\mathrm{ACK}}_{\mathrm{inter}}$ as the implied ACK-size distribution, and d) $\mathrm{B}^{\mathrm{gen}}_{\mathrm{inter}}$ as the convolution of both $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ and $\mathrm{B}^{\mathrm{ACK}}_{\mathrm{inter}}$. One can see that the CDFs of the original and the derived Layer 2 distribution are almost identical which shows that the deconvolution was successful.

One should note that the deconvolution only yields an approximation of $\mathrm{B}^{\mathrm{PL}}_{\mathrm{intra}}$ and might be noisy depending on the resolution of the input data. In our case, we extracted $\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$ from a log-scaled figure published in [KSG+09] which has a very bad resolution. The resulting noise is even more amplified by the transformation from the characteristic function to a probability density function. We thus had to perform some manual filtering on the density function to retrieve the function depicted in Figure 8.5. This filtering basically removed negative values, smoothed the function, and scaled it to sum up to 1. We suspect that with proper data for $\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$, this manual filtering is not necessary.

8.6. Generating Traffic Matrices

In this section we present our approach to generate traffic matrices. The mix of applications run in a data center has a strong influence on the resulting communication structure (and thus on the TMs). If only one single application were running, its communication pattern would be reflected by the resulting traffic matrix. But with more and more applications running simultaneously, the impact of any single application on the overall traffic will reduce and the aggregate behavior will appear more and more random. Moreover, as we stated in the introduction, we do not necessarily have the knowledge which applications are in fact running from the observed and available data. Therefore, we aim at generating TMs that express such (perhaps only seemingly) random traffic patterns. We do highlight that with the techniques developed in [SSKB10], DCT^{2}Gen can be extended to include application-specific communication patterns into the TM generation process. [SSKB10] analyzes packet traces to construct so-called *traffic dispersion graphs* modeling the communication structures of different applications between a set of end hosts. However, this requires access to packet traces from the network in question which are not available to us.

We generate a TM$^{(\mathrm{PL})}$ that specifies the amount of payload exchanged between server pairs within a fixed period. When this payload is transported using TCP, this generates a traffic matrix TM$^{(\mathrm{gen})}$ on Layer 2. For this TM it holds that:

- $\mathrm{B}^{\mathrm{gen}}_{\mathrm{intra}}$ equals $\mathrm{B}^{\mathrm{obs}}_{\mathrm{intra}}$
- $\mathrm{B}^{\mathrm{gen}}_{\mathrm{inter}}$ equals $\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$
- $\mathrm{N}^{\mathrm{gen}}_{\mathrm{intra}}$ equals $\mathrm{N}^{\mathrm{obs}}_{\mathrm{intra}}$
- $\mathrm{N}^{\mathrm{gen}}_{\mathrm{inter}}$ equals $\mathrm{N}^{\mathrm{obs}}_{\mathrm{inter}}$

To create a TM$^{(\mathrm{PL})}$, we first determine each node's number of inter- and intra-rack communication partners by computing a random variable from the corresponding distributions. Then, we use the numbers as node degrees and look for such a undirected simple graph[11] G. Finding a graph with a given inter- and intra-rack node degree is the *k-Partite Degree Sequence Problem* which is a variant of the intensively studied *Degree Sequence Problem.* We give an Integer Linear Program (ILP) to solve the k-Partite Degree Sequence Problem and study its run-time behavior.

In a subsequent step we transform the adjacency matrix of G into a traffic matrix by computing a random variable for the traffic volume for each edge using $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ resp. $\mathrm{B}^{\mathrm{PL}}_{\mathrm{intra}}$.

8.6.1. Problem Formalization

The problem of creating traffic matrices for n nodes with given intra- and inter-rack node degrees can be formalized as follows: The inter-rack node degree of a node is defined as the number of edges to nodes in different racks whereas the intra-rack node degree of a node is defined as the number of edges to nodes in the same rack. Let $V = \{v_1, v_2, ..., v_n\}$ be a set of vertices organized in racks of size m where $v_{i\cdot m}, v_{i\cdot m+1}, ..., v_{(i+1)\cdot m-1}\ \forall\ 0 \leq i < \lceil n/m \rceil$ are located in the same rack i. Let $D^{\mathrm{int}} =$

[11] A simple graph is an undirected graph where no node has an edge to itself and no more than one edge between the same pair of nodes exists.

$(d_1^{\text{int}}, d_2^{\text{int}}, ..., d_n^{\text{int}})$ be the desired intra-rack node degrees and $D^{\text{ext}} = (d_1^{\text{ext}}, d_2^{\text{ext}}, ..., d_n^{\text{ext}})$ the desired inter-rack node degrees for all n nodes. We are looking for an undirected simple graph $G = (V, E)$ where the node degrees follow the intra- and inter-rack degrees given by D^{int} and D^{ext}.

8.6.2. The Degree Sequence Problem

Let $G = (V, E)$ be a simple graph on n vertices. We call the decreasing order of the node degrees of V the *degree sequence* of G.

Problem 1 (The Degree Sequence Problem) Let $V = (v_1, v_2, ..., v_n)$ be a set of nodes and $D = (d_1, d_2, ..., d_n)$, $d_i \geq d_{i+1}\ \forall\ 0 < i < n$, the desired node degrees. Find a simple graph $G = (V, E)$, $E \subseteq V \times V$, where the degree sequence of G is equal to D. If such a graph exists, D is called *realizable.*

Problem 1 is extensively studied [Hak62, Hav55]. The main results on the Degree Sequence Problem for simple graphs are Theorem 1 and Theorem 2.

Theorem 1 (Erdős–Gallai) D is a realizable degree sequence for a simple graph on n nodes if and only if

1. The sum of all desired node degrees is even
2. $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k)\ \forall\ 1 \leq k \leq n.$

Theorem 2 $D = (d_1, d_2, ..., d_n)$ is realizable as a simple graph if and only if $D' = (d_2 - 1, d_3 - 1, ..., d_{d_1+1} - 1, d_{d_1+2}, d_{d_1+3}, ..., d_n)$ is realizable as a simple graph.

From Theorem 2 the iterative algorithm stated in Algorithm 4 can be deduced to create a simple graph with a given node degree. The algorithm creates a graph for which it holds that $\deg(v_i) = d_i$ (if such a graph exists).

8.6.3. The k-Partite Degree Sequence Problem

Creating inter-rack edges is different from Problem 1 because here there exist sets of nodes between which no edges are permitted. These sets are the sets of nodes located in the same rack. This leads us to Problem 2, called the *Degree Sequence Problem on k-Partite Graphs.*

Problem 2 (Degree Sequence Problem on k-Partite Graphs) Given k degree sequences $D_1, D_2, ..., D_k$, find an undirected k-partite Graph G where each partition i

Algorithm 4 ConstructGraph($D = (d_1, ..., d_n)$)

```
1: $G = (V, E)$, $V = \{1, 2, ..., n\}$, $E = \{\}$
2: Let the initial residual node degree of node $v_i$ be $d_i$.
3: Let $U = (u_1, u_2, ..., u_n)$ be the list of vertices decreasing in the order of their
   residual node degree.
4: Create edges between $u_1$ and the next $d_1$ nodes in $U$.
5: if no $d_1$ nodes exists with residual node degree $> 0$ then
6:     return Error
7: end if
8: Update $D$ and corresponding $U$ as stated by Theorem 2.
9: If $U$ is not empty, goto 4.
10: return $\mathrm{Pr}'(\cdot)$
```

consists of $|D_i|$ nodes and for each node v in partition i it holds that $\deg(v) = D_{i_v}$. We call $D_1, D_2, ..., D_k$ *realizable* if such a graph exists.

Problem 2 is a special case of the *Restricted Degree Sequence Problem* [EKMS13] in which arbitrary edges are forbidden to use. The best known algorithm to solve this problem requires to find a perfect matching on a simple graph of $\Omega(n^2)$ nodes. This makes this approach inapplicable to our problem: It already took more than 36 minutes on an Intel i7 2.2 Ghz processor to calculate a graph for seven racks (each consisting of 20 servers) using the Boost graph library.

Reference [MV02] presents the *Degree Sequence Problem with Associated Costs* where the goal is to find a minimum cost realization of a given degree sequence. It is possible to model Problem 2 when setting all costs for intra-rack edges to infinity. However, the running time to solve the problem is also dominated by finding a perfect matching on a graph with $\Omega(n^2)$ nodes. We thus model our problem as an ILP with n constraints, which is faster to solve for the problem instance sizes in this context.

ILP 1 models Problem 2 where $D^{\text{ext}} = D_1 \cup D_2 \cup ... \cup D_k$. In case that D^{ext} is realizable, ILP 1 computes a graph G^{ext} with degree sequence D^{ext}. If D^{ext} is not valid it will compute the Graph G^{ext} which has the highest possible edge count under the condition that no node has a higher degree than specified by D^{ext}.

ILP 1 (Constructing an Inter-Rack Graph)

$$\text{maximize} \qquad \sum_{0<i<n} \sum_{0<j<i} b_{i,j} \qquad\qquad b_{i,j} \in \{0, 1\}$$

w.r.t.

$$\sum_{j \in \mathrm{inter}(i)} b_{\max(i,j),\min(i,j)} \leq d_i^{\text{ext}} \qquad\qquad \forall 1 \leq i \leq n$$

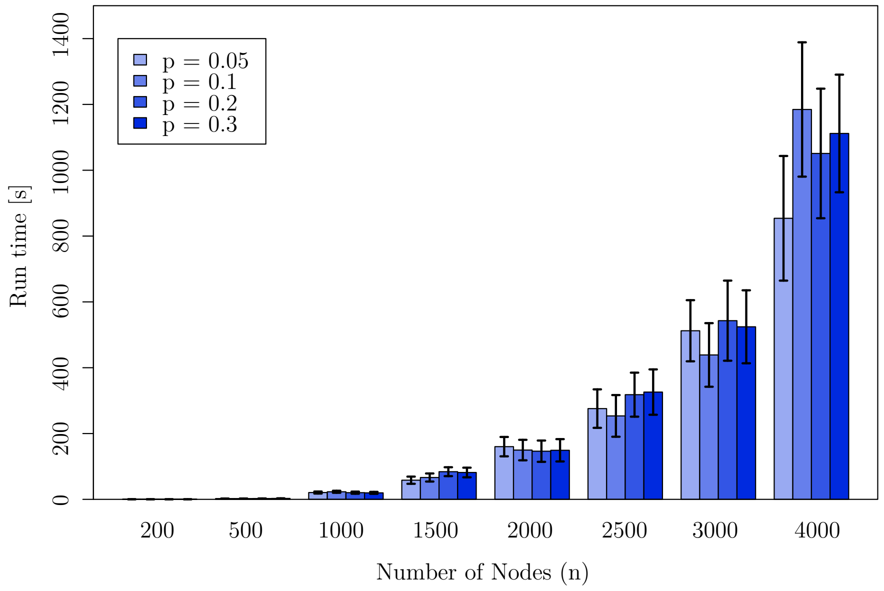


Figure 8.6.: Run time of ILP 1 for different problem sizes. Errorbars indicate the confidence intervals for a confidence level of 95%.

In ILP 1, $b_{i,j}$ equals 1 if an undirected edge exists between nodes i and j. Note that ILP 1 only models the lower triangular matrix of the adjacency matrix of G^{ext} because G^{ext} is undirectional. inter(i) describes the set of nodes that are not in the same rack as node i and is defined as

$$\text{inter}(i) = \left\{ j \mid \left\lfloor \frac{j}{m} \right\rfloor \neq \left\lfloor \frac{i}{m} \right\rfloor \;\; \forall\, 0 < j \leq n \right\}.$$

We use ILP 1 to compute G^{ext}. To show that the running time is acceptable for practical instances we are conducting the following experiment on an Intel i7 960 CPU running at 3.2 GHz with 24 GB of DDR3 memory. The ILP 1 is solved using Gurobi 5.6. We generated problem sequences D^{ext} for different number of nodes n and a fixed rack size $m = 20$. The single d_i^{ext}'s are drawn uniformly at random from the set $\{0, 1, ..., \lfloor n \cdot p \rfloor\}$ for values of $p \in \{0.05, 0.1, 0.2, 0.3\}$. For each parameter pair $(n,\, p)$ we solved 40 problem instances and measured the running times. Figure 8.6 plots the required running time against the size of the problem instance n and the different values for p. One can see that the running time is exponential in n and that the number of edges (controlled by parameter p) has no significant influence on the running time.

8.6.4. Generating a Traffic Matrix

The process of creating a traffic matrix can be divided into the two steps a) finding the positions of non-zero traffic matrix entries, and b) assigning traffic volumes to non-zero traffic matrix entries.

We are finding the positions of non-zero traffic matrix entries by creating a graph G with given intra- and inter-rack node degrees. To this end, two graphs G^{int} and G^{ext} are constructed. $G^{int} = (V, E^{int})$ only contains intra-rack edges with degree sequence D^{int} and $G^{ext} = (V, E^{ext})$ only contains inter-rack edges with degree sequence D^{ext}. We construct G by setting $G = (V, E^{int} \cup E^{ext})$. Whenever there is an edge between a pair of nodes i and j we make (i, j) a random variable in the traffic matrix which is distributed according to $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ resp. $\mathrm{B}^{\mathrm{PL}}_{\mathrm{intra}}$.

To create G^{int}, each rack can be examined separately using Algorithm 4. This leads to k unconnected subgraphs which model the communication between servers in the same racks. However, this only works if the degree sequences are realizable. But, as we draw the degree sequences randomly from the given distribution and the rack sizes are relatively small, in most cases the demanded degree sequences are not realizable. This is problematic as we are not allowed to redraw the degree sequences in that case because this would lead to a wrong distribution of intra-rack node degrees. Note that this problem is specific to the intra-rack case where the number of nodes is very small and the demanded node degrees are very high. For the inter-rack case, the probability of sampling a non-realizable degree sequence is much lower because of the large number of nodes and the relatively small demanded node degrees.

To compute intra-rack edges with degrees following $\mathrm{N}^{\mathrm{obs}}_{\mathrm{intra}}$, we developed ILP 2. ILP 2 assigns penalties to each node in case it does not meet its demanded node degree. The penalty of a node is defined as the absolute difference between the demanded node degree (given by D^{int}) and the node degree in the solution calculated by the ILP itself. ILP 2 minimizes the sum over the penalties of all nodes i divided by the probability of degree d_i (according to $\mathrm{N}^{\mathrm{obs}}_{\mathrm{intra}}$). This way, the sum of the *relative distances* between the degree distribution computed by the ILP and $\mathrm{N}^{\mathrm{obs}}_{\mathrm{intra}}$ is minimized.

ILP 2 (Constructing an Intra-Rack Graph)

$$\text{minimize} \quad \sum_{0<i<n} \frac{p_i}{Pr(d_i)}$$

w.r.t.

$$\begin{array}{llll}
p_i & \geq & 0 & \forall 1 \leq i \leq n \\
p_i & \geq & \sum_{j \in \mathrm{intra}(i)} b_{i,j} - d_i & \forall 1 \leq i \leq n \\
p_i & \geq & d_i - \sum_{j \in \mathrm{intra}(i)} b_{i,j} & \forall 1 \leq i \leq n
\end{array}$$

In ILP 2, p_i is the penalty assigned to node i. The demanded node degree of i is denoted d_i and $b_{i,j}$ is 1 if there is an edge between nodes i and j in the calculated graph. For practical instances, the run time of the ILP 2 is not critical as each rack can be examined separately and racks typically consist of up to 40 servers only.

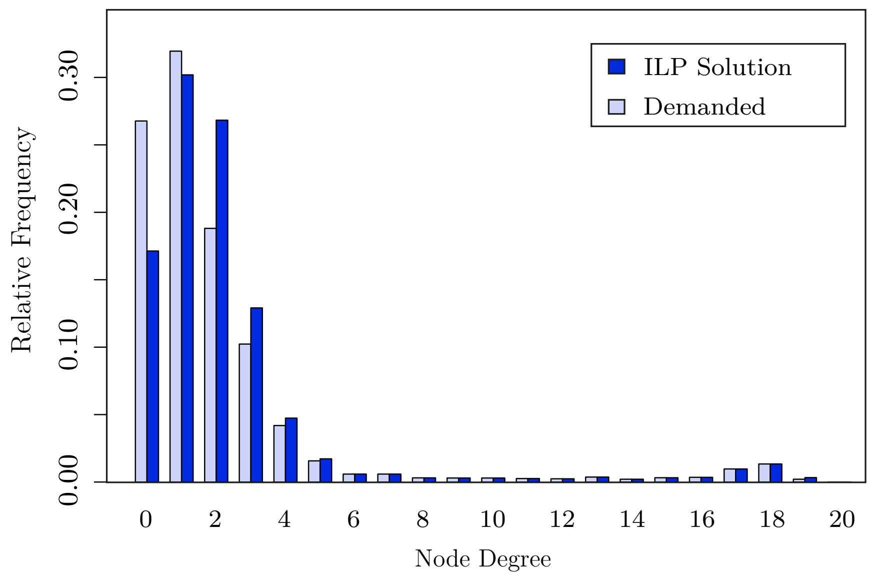


Figure 8.7.: Comparison of a) the distribution of node degrees in the demanded degree sequence and b) the distribution of node degrees of the solution computed by ILP 2

Figure 8.7 shows the solution quality of ILP 2 in an experiment with 10,000 servers organized in racks of 20 servers each. The 10,000 demanded node degrees are distributed according to N^{obs}_{intra}. One can see that the distribution of node degrees computed by ILP 2 is comparable to N^{obs}_{intra}. The distributions match very well for node degrees with small densities. For larger densities, the gap between the distribution of demanded degrees and the distribution of node degrees computed by ILP 2 is larger. However, the solution of ILP 2 is of sufficient quality for our purpose as can clearly be seen in our evaluation (Figure 16 and Figure 17).

8.7. Generating TCP Flows

8.7.1. Overview

A traffic matrix computed by the Traffic Matrix Creator states the amount of bytes exchanged by node pairs in a fixed time. Data-center traffic consists mostly of short-lived flows [KSG+09, BAM10]. Thus, for each communicating node pair (non-zero TM entry), the bytes have to be separated into different flows. We describe a Layer 4 flow as the 4-tuple *(start time, source, destination, size)*. This section only deals with payload flows.

Given a $TM^{(PL)}$ determining how many bytes to transfer between every pair of nodes, the question to answer is: How to separate the non-zero entries of a $TM^{(PL)}$ into flows such that flow sizes are following S^{PL} and the flow inter-arrival times are distributed according to IAT^{PL}?

Our strategy is to first generate a set of flows complying with S^{PL} and $\mathrm{IAT}^{\mathrm{PL}}$ and afterwards map these flows to the non-zero entries of the $\mathrm{TM}^{(\mathrm{PL})}$.

8.7.2. Generating Flows

Generating flows complying with S^{PL} and $\mathrm{IAT}^{\mathrm{PL}}$ for a given traffic matrix is a challenging task. A simple approach would be to go through all non-zero TM pairs (u, v) and generate flows for them according to S^{PL} and $\mathrm{IAT}^{\mathrm{PL}}$. But this approach raises some questions, for example:

When to stop generating new flows for (u, v)***?***
: We could stop assigning flows to (u, v) when the sum of flow sizes for (u, v) is larger than specified by the TM. But then, more traffic would be generated than is specified by the TM. Another way would be to stop generating flows for (u, v) when the next flow that is to be generated would exceed the amount stated by the TM. This way, the traffic generated by the flows would be less than specified by the TM. Hence, no matter how we decide, the resulting $\mathrm{TM}^{(\mathrm{gen})}$would not follow $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ and $\mathrm{B}^{\mathrm{PL}}_{\mathrm{intra}}$.

What if for a small TM entry a huge flow size is generated?
: Generating a new flow size in this situation distorts the resulting flow-size distribution. And by assigning the too large flow, the resulting $\mathrm{TM}^{(\mathrm{gen})}$would not comply with $\mathrm{B}^{\mathrm{PL}}_{\mathrm{inter}}$ and $\mathrm{B}^{\mathrm{PL}}_{\mathrm{intra}}$.

So generating flows for each host pair individually is not practical.

One way to get around these issues is to first create the TM and then a set of "unmapped" flows following S^{PL} and $\mathrm{IAT}^{\mathrm{PL}}$ (where "unmapped" means the flow is not yet assigned to a source-destination pair, *s-d pair*). Afterwards, flows get mapped to s-d pairs such that the sum of flow sizes mapped to each s-d pair matches the amount given by the traffic matrix. However, this mapping has to be done very carefully. Since there is no information known about inter-flow dependencies, the mapping must not introduce any artificial patterns to the generated traffic (such a pattern could, for example, be a higher probability to map large flows to node pairs with large TM entries). Thus, the goal is a random assignment of flows to host pairs (u, v) where the amount of traffic given by the flows between u and v is equal to the TM entry (u, v). We call such a mapping an *exact mapping.* Note that it is not guaranteed (and actually unlikely) that an exact mapping exists. Nevertheless, a good mapping strategy assigns flows such that the sum of flow sizes between nodes i and j is as close as possible to TM entry (i, j).

To create flows, we first determine the overall required traffic s_M of the TM (as the sum of all entries) and then create a set of unmapped flows such that flow sizes sum up to s_M. We denote the sum of all generated flow sizes as s_F. As s_M is a random variable it will hold that $s_M = s_F \cdot \varepsilon$, $\varepsilon \in \mathcal{R}_0^+$, where ε is the imbalance factor between the size of the flows and the TM. Of course, ε should be very close to 1 (meaning there is no imbalance at all), which is why we start over to generate the *whole set* of unmapped flows with adjusted flow inter-arrival times as long as $| \varepsilon - 1 | > 0.01$. This means that the sum of all generated flow sizes deviates at most 1% from the traffic specified by the TM. We assume this to be a reasonably small error.

We will now present two different strategies to map the unmapped flows to node pairs. The first one is a purely random process and the second one uses a variation of the queuing strategy *deficit round robin* (DRR) [SV95]. Afterwards, we study the quality of both strategies.

The randomized assignment uses the TM as a probability distribution and, for each generated flow, draws a node pair from this distribution. In this process, we define the initial probability to assign a flow to node pair (i, j) as the TM entry (i, j) divided by s_M. After a flow has been assigned, the probability distribution at the point of the node pair is lowered proportionally to the size of the flow.

The second strategy is inspired by DRR. DRR schedules jobs of different sizes and classes onto a shared processor. The goal of DRR is to share the processor among all classes according to the ratio of their priorities. To this end, each class is assigned a *priority* and a *credit*. DRR loops Round Robin through all classes. In each iteration of the loop, the credit of each class is raised by some constant (called quantum) weighted by the priority of the class. If for a class there exists a job with a size smaller than the current credit of the class, this job is scheduled to the processor and the credit of the class is lowered by the size of the job.

We use a DRR variant to map flows to node pairs. In this variant, node pairs correspond to classes and flows correspond to jobs. The only difference in our variant is that we do not schedule flows onto a shared processor; we schedule flows on node pairs. The priority of a node pair is proportional to the size of its residual traffic matrix entry. We loop Round Robin over all node pairs and raise their credit proportional to their residual TM entry. Whenever the unmapped flow under consideration is smaller then or equal to the credit of the node pair, this flow is mapped to the node pair and the credit is lowered accordingly.

Our adapted version of the DRR strategy can be seen in Figure 8.8. In this algorithm, i always corresponds to a source, j to a destination and R is the *residual* traffic between i and j as specified by the TM: whenever a flow is assigned to (i, j), $R_{i,j}$ is decreased by the size of the flow. F is a queue that initially contains all flows in a randomized order. $C_{i,j}$ is the credit (akin to DRR) of the node pair (i, j). $C_{i,j}$ is decreased whenever a flow is assigned to (i, j) by the size of the flow. The algorithm iterates Round Robin over all node pairs and tries to assign the flows queued in F. For each flow f the algorithm iterates as long over the node pairs (i, j) as no valid candidate has been found. (i, j) is a valid candidate for flow f if $C_{i,j}$ is larger than or equal to the size of f. After a pair (i, j) has been inspected its deficit counter is increased by $\max(\alpha \cdot R_{i,j}, \omega)$; α and ω control the increase of the deficit counter over

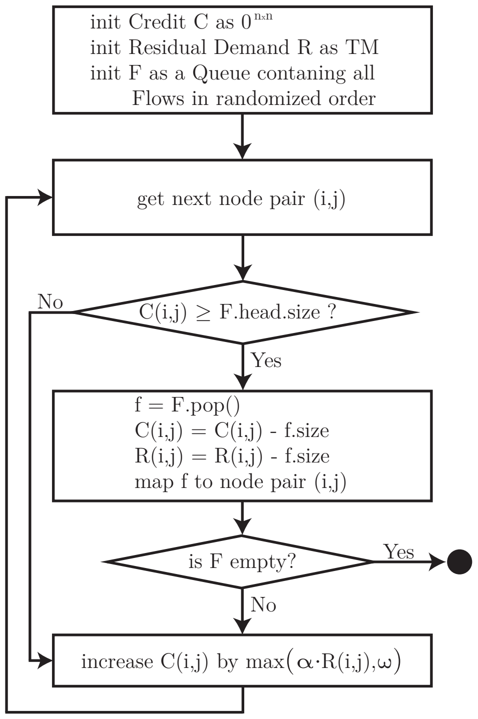

init Credit C as $0^{n \times n}$
init Residual Demand R as TM
init F as a Queue contaning all
Flows in randomized order
get next node pair (i,j)
C(i,j) ≥ F.head.size ?
No
Yes
f = F.pop()
C(i,j) = C(i,j) - f.size
R(i,j) = R(i,j) - f.size
map f to node pair (i,j)
is F empty?
Yes
No
increase C(i,j) by max(α·R(i,j),ω)

Figure 8.8.: Deficit Round Robin inspired algorithm for selecting s-d pairs for flows.

time. Ideally, both parameters are chosen to be very small. We found that setting them to values below $\alpha = 0.1$ and $\omega = 100$ cause no significant improvement of the flow assignment and only increases the run time of the algorithm. Thus, we consider $\alpha = 0.1$ and $\omega = 100$ to be a good choice.

8.7.3. Quality of Flow Assignment

In an optimal flow assignment, each node pair is assigned flows which exactly sum up to the amount of traffic stated by the given TM. In reality, we will produce a traffic matrix with slight derivations. To express the difference between the given TM M and the TM M' produced by the flow assignment we interpret both M and M' as probability distributions of exchanging traffic. Then, we express the distance between these two distributions by the relative entropy. The relative entropy is naturally defined as the *Kullback-Leibler divergence* (KL), but KL requires that $M'_{i,j} = 0 \;\Rightarrow\; M_{i,j} = 0 \;\forall\; (i,j) \in n \times n$, which does not hold in our case. However, the symmetric form of KL, called *Topsøe distance* (Equation 8.7.1) [JS+01] does not require this implication and can be used instead to compute the distance between two probability distributions.

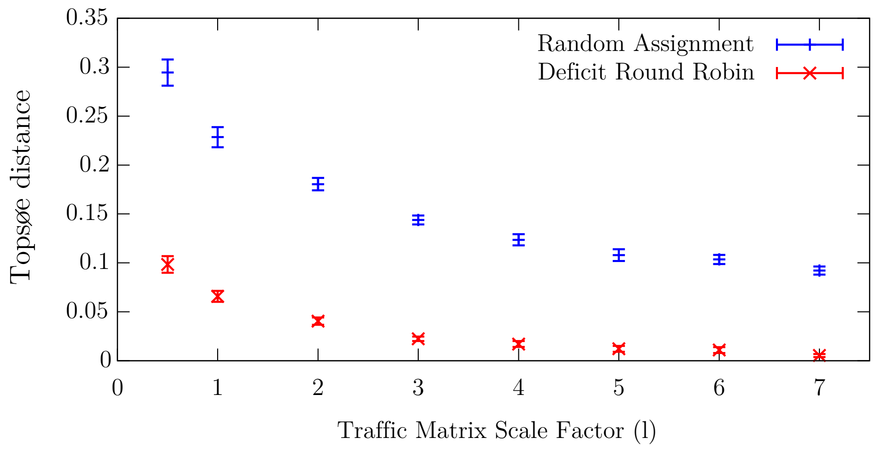


Figure 8.9.: Topsøe distance of flow assignment methods over different traffic volumes. Error bars show confidence intervals for a confidence level of 95%.

$$\text{Topsøe}(M, M') = \sum_{(i,j)} \left(M_{i,j} \ln \frac{2M_{i,j}}{M_{i,j} + M'_{i,j}} + M'_{i,j} \ln \frac{2M'_{i,j}}{M_{i,j} + M'_{i,j}} \right) \qquad (8.7.1)$$

We look at the Topsøe distance for *different* load levels of a network because given a fixed flow-size distribution, an increasing communication volume (TM size) will influence the results of the flow assignment methods: If the total traffic volume tends towards infinity, a single flow gets very small compared to a TM entry. In such a scenario it is very easy to find matching flow assignments. A load level is created by multiplying the TMs with a factor l; we denote the corresponding TM by lM. We then assign flows for lM to s-d pairs and calculate the TM $(lM)'$ based on that flow assignment.

We use lM as the ground truth and express the difference between lM and $(lM)'$ as the relative entropy of both matrices. Figure 8.9 shows the relative entropy obtained via either the random strategy or the Deficit Round Robin strategy calculated as averaged over the *Topsøe distance* of 40 matrices of 10 s generated traffic each $\left(\frac{1}{40}\sum_{i=1}^{40} \text{Topsøe}\left(lM, (lM)'\right)\right)$. The data center for which we generate traffic consists of 75 racks with 20 servers each. It is the same size that was used in the study [KSG$^+$09]. It can be seen that for both methods the Topsøe distance decreases with increasing load but for Deficit Round Robin the relative entropy is much lower, thus the method achieves a better flow assignment than the random mapping process. We will only consider the DRR-based scheme henceforth.

8.8. Empirical Evaluation

8.8.1. Approach

DCT^{2}Gen works properly if it is able to compute a schedule of TCP payload transmissions where (when transferred over a network) a) the generated $\mathrm{TM}^{(\mathrm{gen})}$ has the same properties as $\mathrm{TM}^{(\mathrm{obs})}$ and b) the generated flows have the same properties as the observed flows.

We use a stochastic analysis of the generated TCP schedule to confirm that $\mathrm{TM}^{(\mathrm{gen})}$ follows the same probability distributions as $\mathrm{TM}^{(\mathrm{obs})}$. To this end, we compute $\mathrm{N}^{\mathrm{gen}}_{\mathrm{intra}}$, $\mathrm{N}^{\mathrm{gen}}_{\mathrm{inter}}$, $\mathrm{B}^{\mathrm{gen}}_{\mathrm{intra}}$, $\mathrm{B}^{\mathrm{gen}}_{\mathrm{inter}}$, and $\mathrm{IAT}^{\mathrm{gen}}$ based on the Layer 4 schedule and compare them to $\mathrm{N}^{\mathrm{obs}}_{\mathrm{intra}}$, $\mathrm{N}^{\mathrm{obs}}_{\mathrm{inter}}$, $\mathrm{B}^{\mathrm{obs}}_{\mathrm{intra}}$, $\mathrm{B}^{\mathrm{obs}}_{\mathrm{inter}}$, and $\mathrm{IAT}^{\mathrm{obs}}$. A network emulation through MaxiNet is used to capture the effects of TCP when the generated traffic is replayed on a data-center topology. From the results of the emulation, we compute $\mathrm{S}^{\mathrm{gen}}$ and compare them to $\mathrm{S}^{\mathrm{obs}}$.

8.8.2. Traffic Properties used in the Evaluation

According to [KSG+09], $\mathrm{N}^{\mathrm{obs}}_{\mathrm{intra}}$ and $\mathrm{N}^{\mathrm{obs}}_{\mathrm{inter}}$ are heavy-tailed in typical data centers. It is reported that for a pair of servers located in the same rack, the probability of communicating in a fixed 10 s period is 11 % whereas the probability for out-of-rack communication for any pair of servers is only 0.5 %. In addition, a server either talks to the majority of servers in its own rack or to less than one forth of them. The amount of traffic that is exchanged between server pairs is distributed based on their relationship: Servers in the same rack either exchange only a small amount or a large amount of data, whereas traffic across racks is either small or medium per server pair.

Kandula et al. [KSG+09] found that 80 % of the flows in the data center last no longer than 10 s and that only 0.1 % of the flows last longer than 200 s. More than half the traffic is in flows shorter than 25 s and every millisecond 100 new flows arrive at the network.

An independent study [BAM10] looked at traffic from 10 different data centers. They showed that across all 10 data centers $\mathrm{S}^{\mathrm{obs}}$ is nearly the same. Most of the flows were smaller than 10 KB and 10 % of the flows are responsible for more than half of the traffic in the data centers.

For evaluation, we used the observed distributions by [KSG+09, BAM10] as an input to our traffic generator. Both studies reason about all the traffic in data centers. In addition to traffic transported with TCP, this includes ARP, DNS and many more protocols that do *not* use TCP for transport. This results in traffic characteristics that cannot be reproduced using TCP only. A flow resulting from an ARP request, for example, has a size of 60 bytes which was also the smallest reported flow size. Due to the three-way handshake used to establish and tear down TCP sessions the smallest possible flow size (on Layer 2) TCP can produce is 272 bytes. For evaluation

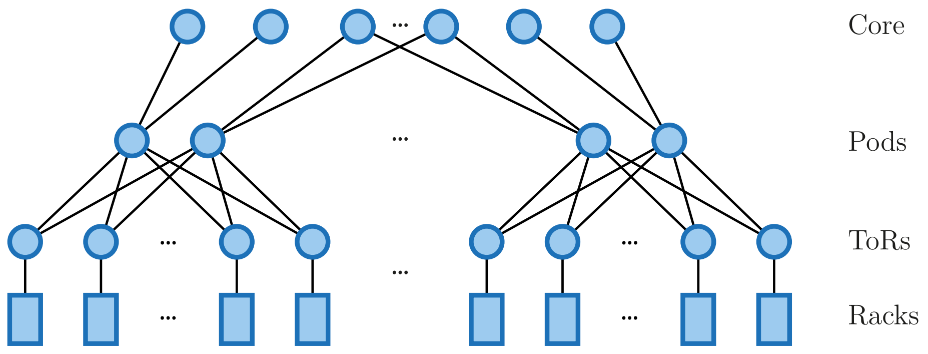


Figure 8.10.: Sketch of the Clos-like topology that was used in our experiments.

we increased all flow sizes by 212 bytes to remove this mismatch.

As a result of that increase, B^{obs}_{intra} and B^{obs}_{inter} no longer match the enlarged S^{obs}. This makes it impossible to have a good flow assignment because there are not enough small flows to be mapped to the small non-zero TM entries. To counteract this, we increased B^{obs}_{intra} and B^{obs}_{inter} by 1000 bytes.

Note that the performed changes are only minor. The average flow size extracted from [BAM10] is 142 KB. Thus, increasing the size of each flow by 219 bytes is an increase of 0.15 % on average. The average non-zero intra-rack traffic matrix entry has a size of 12.6 MB, the average non-zero inter-rack traffic matrix entry 12.4 MB. Thus, an increase of 1000 bytes per non-zero traffic matrix entry is negligible (about 0.1 %).

8.8.3. Topology and Emulation Environment

To include the effects of TCP into our evaluation, we choose to emulate a data center consisting of 72 racks employing a *Clos-like topology*. From the emulation, we are able to determine S^{gen} and IAT^{gen}. A sketch of the emulated topology can be seen in Figure 8.10. Each rack consists of 20 servers and one ToR switch, which makes 1440 servers overall. Servers are connected by 1 Gbps links to ToR switches. Pods consist of *eight* ToR switches which are connected to *two* pod switches with 10 Gbps links. Pod switches are connected to two *core switches* with 10 Gbps links. The core layer in our topology consists of *two* switches. We assume a forwarding delay of 0.05 ms per switch. In each experiment, we emulated 60 seconds of traffic. This traffic was generated from the statistics reported in the previous section. We used a time dilation factor of 200, which means one experiment completed after 200 minutes.

For emulation, we used 12 physical worker nodes equipped with Intel Xeon E5506 CPUs running at 2.16 GHz, 12 Gbytes of RAM and 1 Gbps network interfaces connected to a Cisco Catalyst 2960G-24TC-L Switch. Routing paths are computed using equal cost multipath (ECMP) implemented on the Beacon controller platform [Eri13]. As the controller was placed out-of-band and did not use any kind of time dilation, the routing decisions of the single controller were fast enough for the whole data center network. In addition, the latency between the controller and the emulated switches was not artificially increased. This means that in relation to all the other latencies in the emulated network, the controller decisions were almost immediately present at the switches and did not add any noticeable delay to the flows. Please note that for a real data center (without using time dilation) an ECMP implementation based on only one centralized controller would likely not keep up with the high flow arrival rates; for details see Section 7.4.2.

8.8.4. Results

To verify that DCT^{2}Gen produces a reasonable flow schedule, the traffic created by the schedule (box 5 in Figure 8.1) must have the same properties as the observed traffic (box 1 in Figure 8.1). As a) we do not have access to the observed Layer 2 traces and b) it is unclear how to directly compare two packet traces with each other, we compare the statistical properties of the two traces with each other (boxes 2 and 6 in Figure 8.1). Comparison is done throughout the following sections where each statistical property is inspected individually. Due to the huge amount of samples (our collected packet traces contain 7,060,194 flows, 330,155 distinct intra-rack and 1,675,305 inter-rack TM entries; each of our 16 generated Layer 4 schedules contains around 6 million flows) it is not easily possible to use any goodness of fit test to judge whether the generated distributions match the corresponding observed distributions. This is because there exist small statistical differences between both distributions that together with the large set of samples are big enough for the goodness of fit tests to reject, but too small to be of practical importance for our purpose (these differences are statistically significant, but not relevant). We instead analyze the distributions by using Quantile-Quantile plots (QQ-plots)[12].

Generated Flow-Size Distribution

To determine S^{gen} we emulated 60 seconds of data-center traffic consisting of 1440 hosts as described previously. A packet trace was captured on the first interface of each emulated core switch. We conducted 16 independent experiments (with 16 different Layer 4 schedules) and used the corresponding 32 traces to compute S^{gen}. The number of captured flows over all experiments is 7,060,194.

Figure 8.11 plots S^{obs} and S^{gen}. It can be seen that the distributions clearly match for flow sizes larger 1000 bytes. The distributions of smaller flows, however, do not match well. We suspect this is partly due to the behavior of TCP and partly due to our assumptions on the size of ACK flows as most flows smaller than 1000 bytes are ACK flows (Figure 8.5). As discussed in Section 8.5.2, smaller flows tend to have a lower ACK-to-payload ratio. The Flowset Creator, however, calculates the size of

[12]QQ-plots are plotting the quantiles of both distributions against each other. If the plot shows the identity function, this is an indicator that the distributions fit [CCKT83].

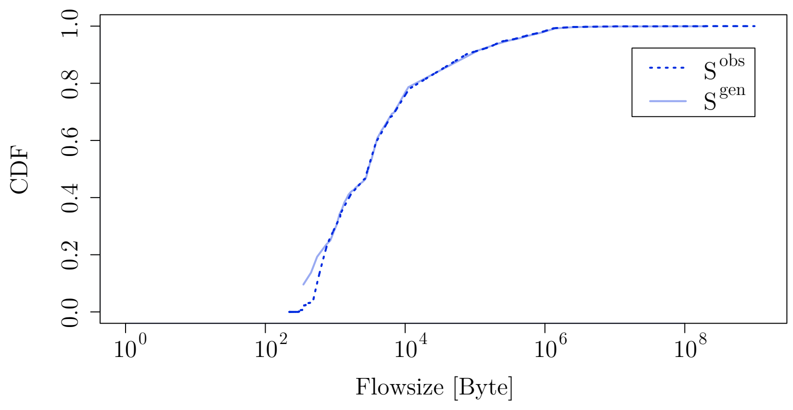


Figure 8.11.: Comparison between S^{gen} and S^{obs}.

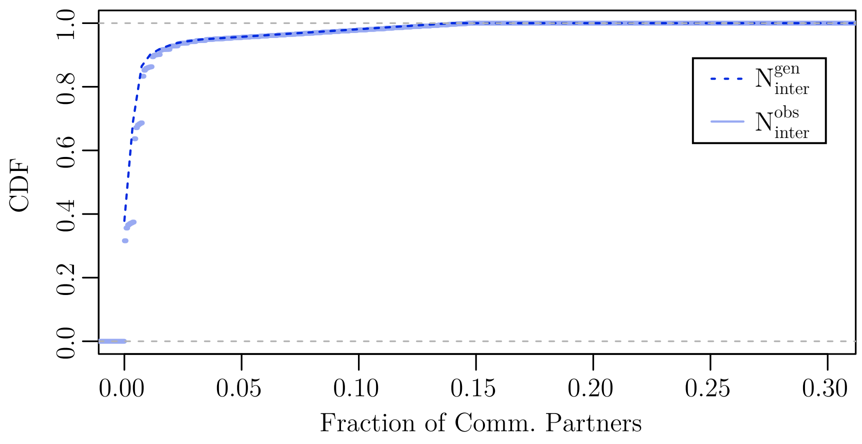


Figure 8.12.: Comparison between N^{gen}_{inter} and N^{obs}_{inter}.

each induced ACK flow with a fixed ratio of r which results in the slightly wrong distribution of ACK-flow sizes.

Inter-Rack Comm. Partners

To determine N^{gen}_{intra} and N^{gen}_{inter} we used the same 16 traffic schedules as before. N^{gen}_{inter} and N^{obs}_{inter} are plotted in Figure 8.12. From the plot no difference between the two distributions is discernible. The corresponding QQ-plot (Figure 8.13) also does not show any significant differences between N^{obs}_{inter} and N^{gen}_{inter}.

Intra-Rack Comm. Partners

The comparison between N^{obs}_{intra} and N^{gen}_{intra} (Figure 8.14) shows that our generated traffic contains a little too many intra-rack communication partners with a low degree. Despite that, both CDFs are nearly identical. This can also be confirmed by looking at the corresponding QQ-Plot (Figure 8.15). The plot shows an almost straight line that lies a bit above the identity function. This result is in line with what is discussed in Section 8.6.4.

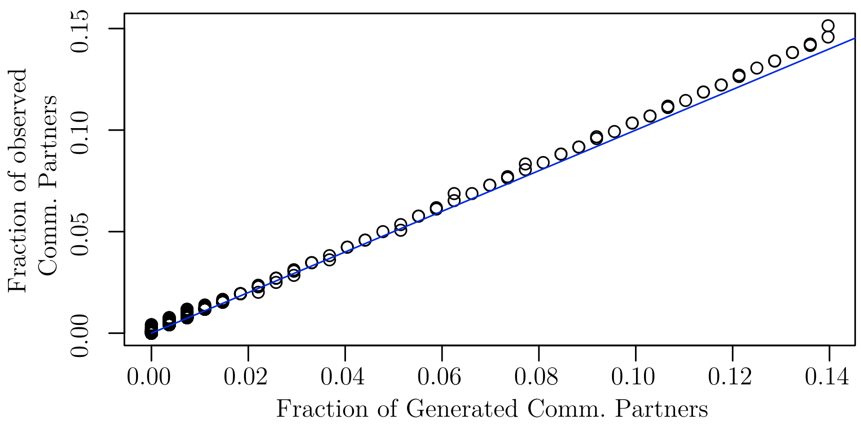


Figure 8.13.: QQ-plot of $N_{\text{inter}}^{\text{gen}}$ and $N_{\text{inter}}^{\text{obs}}$.

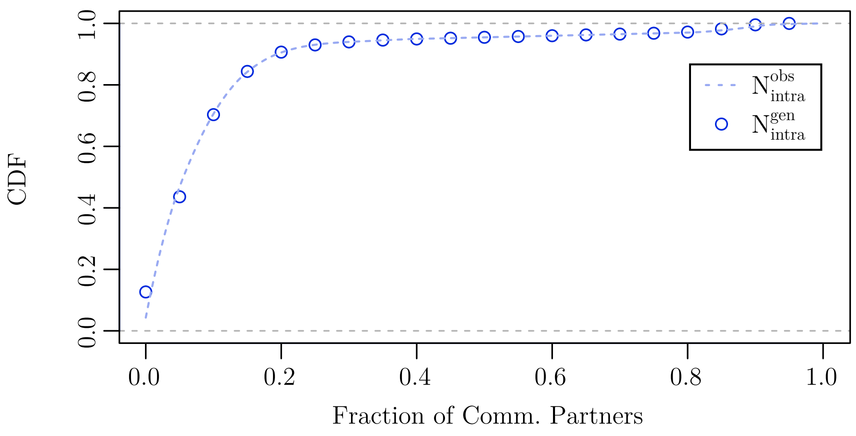


Figure 8.14.: Comparison between $N_{\text{intra}}^{\text{obs}}$ and $N_{\text{intra}}^{\text{gen}}$.

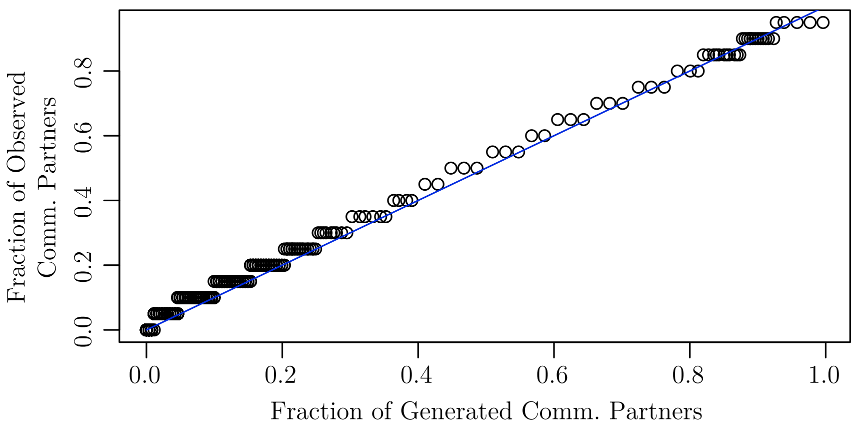


Figure 8.15.: QQ-plot of $N_{\text{intra}}^{\text{obs}}$ and $N_{\text{intra}}^{\text{gen}}$.

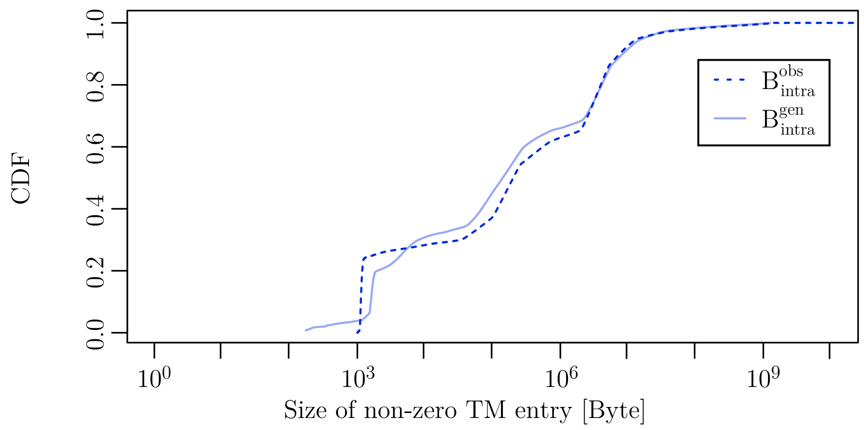


Figure 8.16.: Comparison between B^{obs}_{intra} and B^{gen}_{intra}.

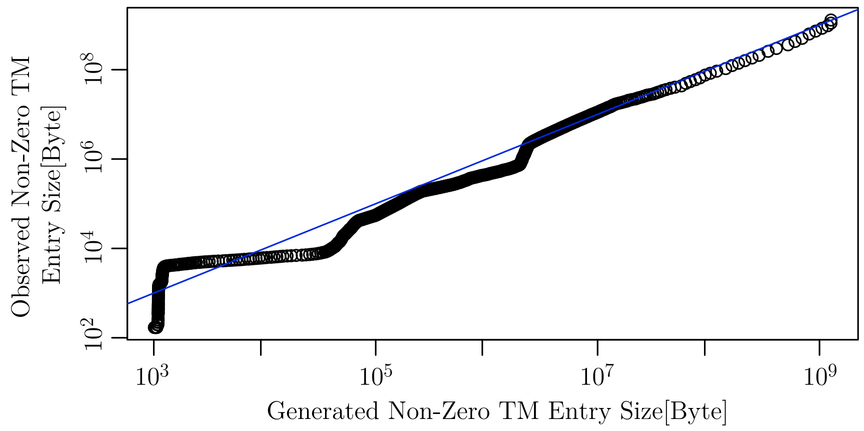


Figure 8.17.: QQ-plot of B^{obs}_{intra} and B^{gen}_{intra}.

Intra-Rack Traffic

The TM$^{(obs)}$s used in this section are deduced from the same 16 traffic schedules we used in Section 8.8.4. To compute the single traffic matrix entries, we fixed the payload-to-ACK ratio r to 2.5 (see Section 8.5.2) and computed the size of the flows on Layer 2 between each pair of servers. From that, we calculated the respective 96 TM$^{(obs)}$s (each for a period of 10 s).

The corresponding B^{gen}_{intra} is compared to B^{obs}_{intra} in Figure 8.16. Except for entries smaller than 10^4 bytes, B^{gen}_{intra} is strictly following B^{obs}_{intra}. This can further be confirmed by the QQ-Plot (Figure 8.17) which additionally only shows a small anomaly of the distribution for entries around 10^6 bytes.

The difference between both distributions in the smaller entries is due to the process of mapping single flows to traffic matrix entries. The goal of the Mapper is to distribute flows to traffic matrix entries such that for each node pair the difference between their TM entry and the sum of flow sizes between that nodes is minimized per server pair. The smaller the TM entry, the fewer flows can be mapped onto the corresponding node pair which means it is harder to find a well fitting mapping.

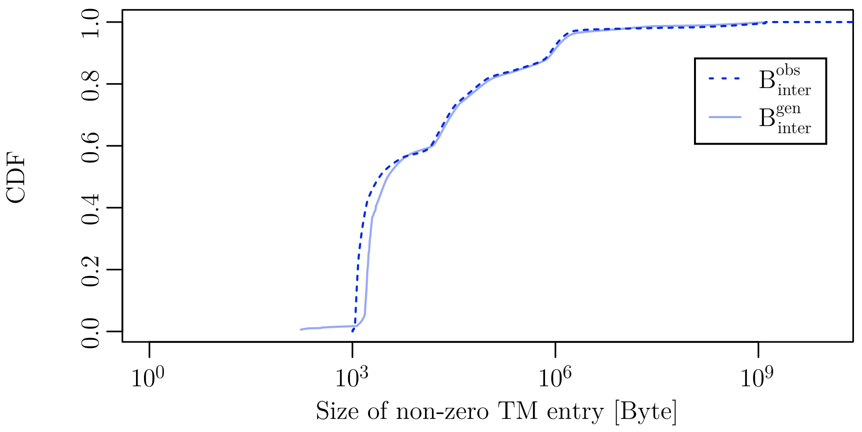


Figure 8.18.: Comparison between B^{obs}_{inter} and B^{gen}_{inter}.

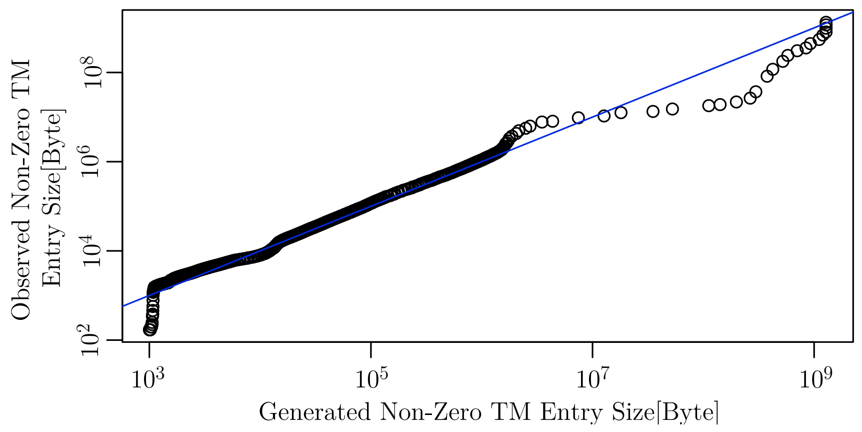


Figure 8.19.: QQ-plot of B^{obs}_{inter} and B^{gen}_{inter}.

Inter-Rack Traffic

B^{obs}_{inter} and B^{gen}_{inter} are plotted in Figure 8.18; the corresponding QQ-plot can be seen in Figure 8.19. From Figure 8.18, we observe the same situation as in the *intra*-rack case. The QQ-plot additionally exposes differences for the distribution of large entries ($> 10^7$). This effect in the QQ-plot is caused by only a slight difference between the tails of both distributions. As the tails of both B^{obs}_{inter} and B^{gen}_{inter} are very long, slight differences in the probabilities have a huge impact on the QQ-plot.

Flow Inter-arrival Time

To compute IAT^{gen}, we used the same 16 traffic schedules as before. In the Flowset Creator, IAT^{gen} is manipulated such that the bytes contained in all generated flows are matching the total traffic of the traffic matrix generated in the Traffic Matrix Generator. Both IAT^{gen} and IAT^{obs} can be seen in Figure 8.20. Apparently, these distributions do not match. The reason for this mismatch is the manipulation done in the Flowset Creator. With IAT^{obs} extracted from [KSG+09] it was not possible to create enough flows to fill up the generated traffic matrices. This can have two

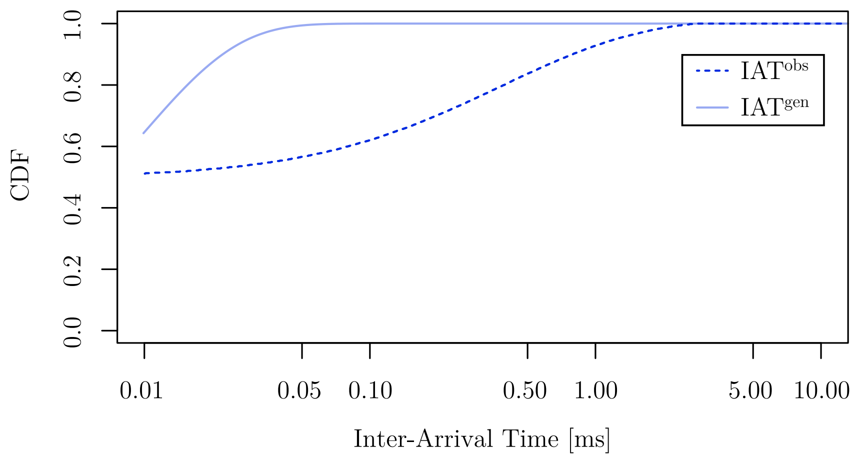


Figure 8.20.: Comparison between IAT^{obs} and IAT^{gen}.

causes: Either the IAT^{obs} reported in [KSG$^+$09] does not match the used S^{obs} or the data provided in [KSG$^+$09] has such a low resolution that we were not able to fully recover it. It would be interesting to repeat this work based on data with better quality.

8.9. Conclusion

The traffic generator DCT^{2}Gen presented in this chapter creates a Layer 4 traffic schedule for arbitrary sized data centers. When the scheduled payloads are transported using TCP, this produces Layer 2 traffic with properties that can be defined in advance using a set of probability distributions. Our evaluation showed that DCT^{2}Gen reproduces these properties with high accuracy. Solely the generated flow inter-arrival time distribution does not match our chosen target distribution. As DCT^{2}Gen manipulates the inter-arrival time distribution to adjust the amount of flows to the given traffic matrices, this is not surprising. We suspect that this difference will be significantly smaller when using input data of higher quality.

Given that DCT^{2}Gen generates a schedule of payload transmissions between all hosts in a data center it is suitable for simulations, network emulations, and testbed experiments. Using our generated traffic schedule combined with a large-scale network emulator such as MaxiNet, novel networking ideas can be evaluated under highly realistic conditions which brings new ideas a step closer to deployment in production environments.

9 Conclusion and Future Work

This chapter summarizes the most important conclusions and gives an answer to the question stated in the beginning of this thesis. Afterwards, possible directions for future research are given.

9.1. Conclusion

This thesis investigated how application-layer knowledge can be used to improve the behavior of networks. The leading question of this thesis is “To which extent does application-layer knowledge help traffic engineering to increase network performance?”. To answer this question, both software-defined circuit- and packet-switched networks are investigated.

In particular, a software-defined IP-over-WDM network is considered as a representative for future circuit-switched networks. In such networks, a software-defined IP routing layer resides directly on top of a circuit-switched WDM network. A special feature of optical WDM networks is the reconfigurability of the logical topology. This thesis shows how to design a feedback loop between the IP routing layer and the WDM network control that is both practical and efficient. Using that feedback loop, it is possible to include knowledge from the IP routing layer into the design of the logical topology. By leveraging this knowledge, congestion can be reduced by up to 50 % compared to the state-of-the-art topology design algorithm using no additional knowledge at all. When using a simple rerouting approach, the traffic on IP layer is not affected by service interruptions due to reconfiguration of the logical topology. This makes the approach a promising component for the WAN of the future.

To investigate the value of using application-layer knowledge in packet-based routing, this thesis investigates an OpenFlow-controlled data-center network. Data centers commonly host diverse distributed applications that require at lot of communication. Modern OpenFlow switches have a lot of limitations, i.e., limited flow-table size, limited memory write speeds, and limited processing capabilities. All this limits the granularity of routing because in scenarios with high flow arrival rates, we cannot use individually computed routing paths for each single flow. In consequence, flows have to be grouped using wildcards and then routing decisions for each group of flows have to be made. As this limits the possibilities for routing, this thesis investigated how application-layer knowledge can help working around these limitations. In particular, this thesis presents a routing algorithm called HybridTE that proactively installs wildcard entries at all switches. Using these wildcard entries, initially all traffic follows default routes. For traffic engineering, elephant flows are reported from the applications running in the data center to HybridTE, which in turn computes a custom route for each reported elephant. This way, load can be routed around congested parts of the network. HybridTE is tested using different percentages for false positives and false negatives in reporting elephants. In addition, different reporting delays are considered. It could be shown that HybridTE achieves up to 14.9 % lower flow completion times than the state-of-the-art data-center routing algorithm ECMP while having very low requirements on the switching hardware. To deal with non-cooperative applications, i.e., applications that do not report any flows, a very simple elephant detector based on packet sampling was evaluated. With the resulting quality of information about elephant flows, HybridTE is able to lower the flow completion time by up to 12.4 % in the considered scenarios.

With respect to the initially formulated question, the two key conclusions of this thesis are:

- With active support from the application layer, the efficiency of WAN networks can be increased significantly. Congestion in realistic WAN scenarios can be decreased by up to 50 %.
- By leveraging even uncertain information about large data transmissions, data-center networks can be built from low-cost equipment while reducing the average flow completion time by up to 14.9 % compared to ECMP.

9.2. Future Work

Leverage Knowledge About Flows Starting in the Future

This thesis shows how to design HybridTE, a routing algorithm leveraging certain and uncertain information about flows *currently* active in the network. Creating information about *future* flows, i.e., flows that will start some time in the future, is—depending on the application—definitely possible. HadoopWatch [PCW$^+$14], for example, forecasts future flows originating from Hadoop worker nodes. This information can be used to create a flow scheduling algorithm incorporating information about future flows. Research in this area should clarify *if* and to *which extent* knowledge about future flows is advantageous.

Actively Stalling Low Priority Flows

The traffic engineering approaches considered in this thesis handle flows as if they all have the same priority. In practice, networks carry traffic originating from different applications and hence different flows might have different priorities. By considering this in traffic engineering, it might be beneficial to actively stall flows with low priority if this is advantageous for higher prioritized flows. Stalling could either happen without application support by actively rate limiting the corresponding flows in the network or with application support using an interface between network control and the applications. However, possible traffic engineering schemes have to create a certain level of fairness to prevent low prioritized flows from starving. Google already uses a similar approach in their B4 WAN network [JKM$^+$13] where they are able to reschedule whole backup jobs in favor of higher prioritized traffic. Research on this idea in the data-center context is the next natural step.

Advanced Interplay Between Job and Flow Scheduler

As already stated in Section 7.5, only little work has been done on combining job scheduling and flow scheduling. Whenever multi-tier applications are to be placed in a network (which we call job scheduling), these applications a) will create traffic patterns between the single parts which can (to a certain extent) be forecast and b) the single parts have certain requirements on the quality of communication between each other (such as data rate, latency or jitter). With a traffic forecast, the quality of traffic engineering can possibly be improved, leading to less congestion in the network. Hence, with information from the job scheduler, the flow scheduler may be able to make better decisions.

The placement of application parts on a set of physical machines at the data center does not only depend on the performance characteristics of the physical machines themselves but also on the communication characteristics between the set of machines. In turn, the job scheduling problem cannot be solved properly without information from the flow scheduler. Future research should concentrate on joint job- and flow scheduling. Note that this problem is not specific to big-data applications but also exists in the placement of virtual network functions (VNF) where a service is composed of multiple VNFs. A data flow typically has to pass all virtual network functions on the way from its source to destination.

Network Oracle

There are various sources of information to predict traffic patterns from. Such sources include, but are not limited to:

- Application-layer knowledge
- Elephant detectors
- Specifications of multi-tier applications / VNFs
- Analysis of big-data application specification languages, such as Apache Pig (`https://pig.apache.org`; a Pig-Latin implementation)
- Analysis of log files, such as HadoopWatch [PCW$^+$14]

All these sources can be used to obtain information used to create traffic forecasts. However, the type of information may be different along the different sources. An elephant detector, for example, reports uncertain information about large flows, while the specification of multi-tier applications may feature information about the flow arrival rate or the maximum data rate between different components. Application-layer knowledge, on the other hand, can result in various types of information.

Future research should concentrate on creating a network component which is able to aggregate all these different sources of information to create a combined traffic forecast. This forecast can then be exported to a traffic engineering component. The new network component hence abstracts from the different sources and can be used by traffic engineering algorithms as an *oracle* for the future network state.

Bibliography

[ADRC14] Kanak Agarwal, Colin Dixon, Eric Rozner, and John Carter. Shadow MACs: Scalable Label-switching for Commodity Ethernet. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 157–162, New York, NY, USA, 2014. ACM.

[AED+14] 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, pages 503–514, New York, NY, USA, 2014. ACM.

[AFRR+10] 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, Berkeley, CA, USA, 2010. USENIX Association.

[AGE+04] Stefano Avallone, S Guadagno, Donato Emma, Antonio Pescapè, and Giorgio Ventre. D-ITG Distributed Internet Traffic Generator. In *Proceedings of the First International Conference on the Quantitative Evaluation of Systems (QEST 2004)*, pages 316–317. IEEE, 2004.

[Apa] Apache. Hadoop. http://hadoop.apache.org.

[APV04] Stefano Avallone, Antonio Pescape, and Giorgio Ventre. Analysis and Experimentation of Internet Traffic Generator. In *ACM Workshop on Models, Methods and Tools for Reproducible Network Research*, pages 70–75, 2004.

[APY+14] R. Alimi, R. Penno, Y. Yang, S. Kiesel, S. Previdi, W. Roome, S. Shalunov, and R. Woundy. Application-Layer Traffic Optimization (ALTO) Protocol. RFC 7285, RFC Editor, September 2014. `http://www.rfc-editor.org/rfc/rfc7285.txt`.

[AS13] Vitaly Antonenko and Ruslan Smelyanskiy. Global Network Modelling Based on Mininet Approach. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 145–146. ACM, 2013.

[BAAZ11] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12, New York, NY, USA, 2011. ACM.

[BAM10] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC '10, pages 267–280, New York, NY, USA, 2010. ACM.

[Bel03] Denis Belomestny. Rates of convergence for constrained deconvolution problem. preprint arxiv math.st/0306237 v1, 2003.

[Bra89] Robert Braden. Requirements for Internet Hosts - Communication Layers. STD 3, RFC Editor, October 1989. `http://www.rfc-editor.org/rfc/rfc1122.txt`.

[BTI$^+$03] Chadi Barakat, Patrick Thiran, Gianluca Iannaccone, Christophe Diot, and Philippe Owezarski. Modeling internet backbone traffic at the flow level. *IEEE Transactions on Signal Processing*, 51(8):2111–2124, 2003.

[CByL07] Xiaowen Chu, Tianming Bu, and Xiang yang Li. A Study of Lightpath Rerouting Schemes in Wavelength-Routed WDM Networks. In *Proceedings of the International Conference on Comunications*. IEEE, 2007.

[CCKT83] J. M. Chambers, W. S. Cleveland, Beat Kleiner, and Paul A. Tukey. *Graphical Methods for Data Analysis*. Wadsworth, 1983.

[CGK92] I. Chlamtac, A. Ganz, and G. Karmi. Lightpath Communications: An Approach to High Bandwidth Optical WAN's. *IEEE Transactions on Communications*, 40(7):1171–1182, Jul 1992.

[Cha00] J. Charzinski. Internet Client Traffic Measurement and Characterisation Results. In *Proceedings of the 13th International Symposium on Services and Local Access (ISSLS 2000)*, 2000.

[CKY11] A.R. Curtis, Wonho Kim, and P. Yalagandula. Mahout: Low-overhead Datacenter Traffic Management Using End-Host-Based Elephant Detection. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 1629–1637, April 2011.

[CL05] Xiaowen Chu and J. Liu. DLCR: a new adaptive routing scheme in WDM mesh networks. In *Proceedings of the IEEE International Conference of Communications (ICC)*, 2005.

[Clo53] Charles Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32(2):406–424, March 1953.

[CMT$^+$11] Andrew R Curtis, Jeffrey C Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow : Scaling Flow Management for High-Performance Networks. In *Proceedings of the ACM SIGCOMM Conference*, pages 254–265, 2011.

[CPZ04] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive packet sampling for accurate and scalable flow measurement. In *Proceedings of the Global Telecommunications Conference (GLOBECOM)*, volume 3, pages 1448–1452 Vol.3, Nov 2004.

[CZS14] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient Coflow Scheduling with Varys. In *Proceedings of the ACM SIGCOMM Conference*, SIGCOMM '14, pages 443–454, New York, NY, USA, 2014. ACM.

[DBKK12] M. Dräxler, T. Biermann, H. Karl, and W. Kellerer. Cooperating Base Station Set Selection and Network Reconfiguration in Limited Backhaul Networks. In *Proceedings of the IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*, 2012.

[DHM$^+$13] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. Towards an Elastic Distributed SDN Controller. In *Proceedings of the second ACM SIGCOMM Workshop on Hot Topics in Software-Defined Networking (HotSDN)*, pages 7–12. ACM, 2013.

[DKBR14] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized Task-aware Scheduling for Data Center Networks. In *Proceedings of the ACM SIGCOMM Conference*. ACM, 2014.

[DM06] Nandita Dukkipati and Nick McKeown. Why flow-completion time is the right metric for congestion control. *ACM SIGCOMM Computer Communication Review*, 36(1):59–62, 2006.

[DR00] R. Dutta and G.N. Rouskas. A Survey of Virtual Topology Design Algorithms for Wavelength Routed Optical Networks. *Optical Networks Magazine*, 2000.

[EKMS13] Péter L. Erdös, Sándor Z. Kiss, István Miklós, and Lajos Soukup. Constructing, sampling and counting graphical realizations of restricted degree sequences. *arXiv math/1301.7523 v3*, 2013.

[Eri13] David Erickson. The Beacon OpenFlow Controller. In *Proceedings of the Second Workshop on Hot Topics in Software Defined Networking (HotSDN)*. ACM, 2013.

[GAK$^+$14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-resource Packing for Cluster Schedulers. In *Proceedings of the ACM SIGCOMM Conference*. ACM, 2014.

[GM02] Aysegul Gencata and Biswanath Mukherjee. Virtual-Topology Adaptation for WDM Mesh Networks Under Dynamic Traffic. *IEEE/ACM Transactions on Networking*, 2002.

[GRL05] S. Guruprasad, R. Ricci, and J. Lepreau. Integrated Network Experimentation Using Simulation and Emulation. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, 2005.

[GSB13] Mukta Gupta, Joel Sommers, and Paul Barford. Fast, Accurate Simulation for SDN Prototyping. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics on Software Defined Networking*, 2013.

[GYM$^+$05] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C Snoeren, Amin Vahdat, and Geoffrey M Voelker. To Infinity and Beyond: Time Warped Network Emulation. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. ACM, 2005.

[Hak62] S. L. Hakimi. On realizability of a Set of Integers as Degrees of the Vertices of a Linear Graph. *Journal of the Society for Industrial and Applied Mathematics*, 10:496–506, 1962.

[Hav55] Václav Havel. Poznámka o existenci konečných grafů. *Časopis pro pěstování matematiky*, 080(4):477–480, 1955.

[Hee00] Poul E Heegaard. GenSyn - A Java Based Generator of Synthetic Internet Traffic Linking User Behaviour Models to Real Network Protocols. In *ITC Specialist Seminar on IP Traffic Measurement, Modeling and Management*, 2000.

[HHJ$^+$12] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible Network Experiments Using Container-based Emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012.

[HL06] Quang-Dzung Ho and Man-Seop Lee. Connection Level Active Rerouting in WDM Mesh Networks with Traffic Grooming Capability. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2006.

[HLR+08] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and JB Kopena. Network Simulations with the NS-3 Simulator. In *Proceedings of the ACM SIGCOMM Conference*, 2008.

[JKM+13] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined WAN. *SIGCOMM Computer Communication Review*, 43(4):3–14, August 2013.

[JN13] Dong Jin and David M. Nicol. Parallel simulation of software defined networks. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, 2013.

[JS+01] Don H Johnson, Sinan Sinanovic, et al. Symmetrizing the Kullback-Leibler Distance. *IEEE Transactions on Information Theory*, 1(1):1–10, 2001.

[KB10] Manayya KB. Constrained Shortest Path First. Internet-Draft draft-manayya-constrained-shortest-path-first-02, IETF Secretariat, February 2010. `http://www.ietf.org/internet-drafts/draft-manayya-constrained-shortest-path-first-02.txt`.

[KJ13] Dominik Klein and Michael Jarschel. An OpenFlow extension for the OMNeT++ INET framework. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, 2013.

[KK95] George Karypis and Vipin Kumar. METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0. Technical report, 1995.

[KM02] T. Klingenberg and R. Manfredi. Gnutella 0.6 RFC. online, June 2002. `http://rfc-gnutella.sourceforge.net/src/rfc-0_6-draft.html`.

[KSG+09] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference (IMC)*, pages 202–208, 2009.

[LHM10] 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*, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[Liu02] K.H. Liu. *IP Over WDM.* Wiley, 2002.

[LL96] K. Lee and V. Li. A Wavelength Rerouting Algorithm in Wide-Area All-Optical Networks. *Journal of Lightwave Technology*, 1996.

[LLP+02] Kevin H. Liu, Changdong Liu, Jorge L. Pastor, Arunendu Roy, and John Y. Wei. Performance and Testbed Study of Topology Reconfiguration in IP Over Optical Networks. *IEEE Transactions on Communications*, 2002.

[LSN+09] Seung-Hwan Lim, B. Sharma, Gunwoo Nam, Eun Kyoung Kim, and C.R. Das. MDCSim: A Multi-Tier Data Center Simulation Platform. In *Proceedings on the IEEE International Conference on Cluster Computing (CLUSTER)*, 2009.

[LSP] Juha Laine, Sampo Saaristo, and Rui Prior. Real-time UDP Data Emitter. `http://rude.sourceforge.net`.

[LWPB07] Yi Lu, Mei Wang, B. Prabhakar, and F. Bonomi. ElephantTrap: A Low Cost Device for Identifying Large Flows. In *Proceedings on the 15th Annual IEEE Symposium on High-Performance Interconnects*, pages 99–108, Aug 2007.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 2008.

[MV02] Milena Mihail and Nisheeth K. Vishnoi. On Generating Graphs with Prescribed Vertex Degrees for Complex Network Modeling. In *Proceedings of the 3rd Workshop on Approximation and Randomization Algorithms in Communication Networks*, 2002.

[ONF09] Open Networking Foundation. Openflow switch specification version 1.0.0. online, 2009. Accessable at `https://www.opennetworking.org`.

[PCW+14] Yang Peng, Kai Chen, Guohui Wang, Wei Bai, Zhiqiang Ma, and Lin Gu. HadoopWatch: A First Step Towards Comprehensive Traffic Forecasting in Cloud Computing. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, pages 19–27, April 2014.

[RS95] Rajiv Ramaswami and Kumar N. Sivarajan. Routing and Wavelength Assignment in All-Optical Networks. *IEEE/ACM Transactions on Networking*, 1995.

[RSS09] Rajiv Ramaswami, Kumar Sivarajan, and Galen Sasaki. *Optical Networks: A Practical Perspective, 3rd Ed.* Morgan Kaufmann Publishers, 2009.

[SBCL01] Gangxiang Shen, Sanjay Bose, Tee Hiang Cheng, and Chao Lu. Efficient Heuristic Algorithms for Light-Path Routing and Wavelength Assignment in WDM Networks Under Dynamically Varying Loads. *Elsevier Journal on Computer Communications*, 2001.

[SK14] 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)*, pages 115–120. ACM, 2014.

[SKB04] Joel Sommers, Hyungsuk Kim, and Paul Barford. Harpoon: A Flow-Level Traffic Generator for Router and Network Tests. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2004.

[SOI$^+$03] K. Shiomoto, E. Oki, W. Imajuku, S. Okamoto, and N. Yamanaka. Distributed virtual network topology control mechanism in gmpls-based multiregion networks. *IEEE Journal on Selected Areas in Communications*, 2003.

[SSKB10] Peter Siska, Marc Ph. Stoecklin, Andreas Kind, and Torsten Braun. A flow trace generator using graph-based traffic classification techniques. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, pages 457–462, New York, NY, USA, 2010. ACM.

[SV95] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. *ACM SIGCOMM Computer Communication Review*, 25(4):231–242, 1995.

[Var] A. Varga. OMNeT++ Discrete Event Simulation System. `http://www.omnetpp.org/`.

[VV09] Kashi Venkatesh Vishwanath and Amin Vahdat. Swing: Realistic and Responsive Network Traffic Generation. *IEEE/ACM Transactions on Networking*, pages 712–725, 2009.

[WAHC$^+$06] Michele C Weigle, Prashanth Adurthi, Félix Hernández-Campos, Kevin Jeffay, and F Donelson Smith. Tmix: a tool for generating realistic tcp application workloads in ns-2. *ACM SIGCOMM Computer Communication Review*, 36(3):65–76, 2006.

[WCY13] Shie-Yuan Wang, Chih-Liang Chou, and Chun-Ming Yang. EstiNet Openflow Network Simulator and Emulator. *IEEE Communications Magazine*, 51(9):110–117, 2013.

[WDS$^+$14] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Hassan Zahraee, and H. Karl. MaxiNet: Distributed Emulation of Software-Defined Networks. In *IFIP Networking Conference*, 2014.

[WK13a] P. Wette and H. Karl. Incorporating Feedback from Application Layer into Routing and Wavelength Assignment Algorithms. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013.

[WK13b] P. Wette and H. Karl. On the Quality of Selfish Virtual Topology Reconfiguration in IP-over-WDM Networks. In *Proceedings of the 19th IEEE Int. Workshop on Local and Metropolitan Area Networks*, 2013.

[WK13c] P. Wette and H. Karl. Which Flows Are Hiding Behind My Wildcard Rule? Adding Packet Sampling to OpenFlow. In *Proceedings of the ACM SIGCOMM 2013 conference on Applications, technologies, architectures, and protocols for computer communication*, 2013.

[WK14a] P. Wette and H. Karl. DCT^{2}Gen: A Versatile TCP Traffic Generator for Data Centers. *arXiv preprint arXiv:1409.2246*, 2014. (submitted to Elsevier Journal on Computer Communications).

[WK14b] P. Wette and H. Karl. Using Application Layer Knowledge in Routing and Wavelength Assignment Algorithms. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2014.

[WK15] P. Wette and H. Karl. HybridTE: Traffic Engineering for Very Low-Cost Software-Defined Data-Center Networks. *arXiv preprint arXiv:1503.04317*, 2015. (submitted to the European Workshop on Software Defined Networks).

[WLYX07] Xuetao Wei, Lemin Li, Hongfang Yu, and Du Xu. Dynamic Preemptive Multi-class Routing Scheme Under Dynamic Traffic in Survivable WDM Mesh Networks. In *High Performance Computing and Communications*, volume 4782 of *Lecture Notes in Computer Science*. Springer, 2007.

[WSQ$^+$03] I. Widjaja, I. Saniee, L. Qian, A. Elwalid, J. Ellson, and L. Cheng. A New Approach for Automatic Grooming of SONET Circuits to Optical Express Links. In *Proceedings of the IEEE International Conference on Communications (ICC)*, 2003.

[WSSK15] P. Wette, A. Schwabe, M. Splietker, and H. Karl. Extending Hadoop's Yarn Scheduler Load Simulator with a Highly Realistic Network & Traffic Model. In *Proceedings of the 1st IEEE Conference on Network Softwarization*, 2015.

[Wu11] Jing Wu. A Survey of WDM Network Reconfiguration: Strategies and Triggering Methods. *Elseview Journal on Computer Networks*, 2011.

[WWH$^+$11] Xin Wan, Lei Wang, Nan Hua, Hanyi Zhang, and Xiaoping Zheng. Dynamic Routing and Spectrum Assignment in Flexible Optical Path Networks. In *Proceedings of the Optical Fiber Communication Conference and Exposition*, 2011.

[XAG12] Chongyang Xie, Hamed Alazemi, and Nasir Ghani. Rerouting in Advance Reservation Networks. *Elseview Journal on Computer Communications*, 2012.

[XSBM08] Ming Xia, Lei Song, M. Batayneh, and B. Mukherjee. Event-Triggered Reprovisioning with Resource Preemption in WDM Mesh Networks: A Traffic Engineering Approach. In *Journal of the Optical Fiber communication/National Fiber Optic Engineers Conference*, 2008.

[Yen71] J Y Yen. Finding the k shortest loopless paths in a network. *Management Science*, 1971.

[YR04] Wang Yao and B. Ramamurthy. Rerouting Schemes for Dynamic Traffic Grooming in Optical WDM Mesh Networks. In *Global Telecommunications Conference*, 2004.

[ZIB] Zuse-Institute Berlin. SNDlib `http://sndlib.zib.de`.

[ZPC$^+$08] Y. Zhou, G.-S. Poo, S. Chen, P. Shum, and L. Zhang. Dynamic Multicast Routing and Wavelength Assignment Using Generic Graph Model for Wavelength-Division-Multiplexing Networks. *IET Journal on Communications*, 2008.