Energy Efficient Traffic Engineering in Software Defined Networks

Radu Carpa

Numéro national de thèse 2017LYSEN065

THÈSE de DOCTORAT DE L'UNIVERSITE DE LYON

opérée par

l'École Normale Supérieure de Lyon

École Doctorale N° 512
École Doctorale en Informatique et Mathématiques de Lyon

Spécialité de doctorat :
Informatique

Soutenue publiquement le 26 octobre 2017, par :

Radu CÂRPA

Energy Efficient Traffic Engineering in Software Defined Networks

Ingénierie de trafic pour des réseaux énergétiquement efficaces

Devant le jury composé de :

Andrzej DUDA	Professeur – Grenoble INP-Ensimag	Examinateur
Frédéric GIROIRE	Chargé de Recherches – CNRS Sophia Antipolis	Examinateur
Brigitte JAUMARD	Professeure – Concordia University, Canada	Rapporteure
Béatrice PAILLASSA	Professeure – Institut national polytechnique de Toulouse	Rapporteure
Laurent LEFEVRE	Chargé de Recherches – Inria ENS Lyon	Directeur
Olivier GLUCK	Maître de Conférences – UCBL Lyon1	Co-encadrant

Abstract

As the traffic of wired networks continues to grow, their energy consumption becomes a serious efficiency and sustainability concern. This work seeks to improve the energy efficiency of backbone networks by automatically managing the paths of network flows to reduce the over-provisioning. Compared to numerous works in this field, we stand out by focusing on low computational complexity and smooth deployment of the proposed solution in the context of Software Defined Networks (SDN). To ensure that we meet these requirements, we validate the proposed solutions on a network testbed built for this purpose. Moreover, we believe that it is indispensable for the research community in computer science to improve the reproducibility of experiments. Thus, one can reproduce most of the results presented in this thesis by following a couple of simple steps.

In the first part of this thesis, we present a framework for putting links and line cards into sleep mode during off-peak periods and rapidly bringing them back *on* when more network capacity is needed. The solution, which we term "SegmenT Routing based Energy Efficient Traffic Engineering" (STREETE), was implemented using state-of-art dynamic graph algorithms. STREETE achieves execution times of tens of milliseconds on a 50-node network. The approach was also validated on a testbed using the ONOS SDN controller along with OpenFlow switches. We compared our algorithm against optimal solutions obtained via a Mixed Integer Linear Programming (MILP) model to demonstrate that it can effectively prevent network congestion, avoid turning-*on* unneeded links, and provide excellent energy-efficiency.

The second part of this thesis studies solutions for maximizing the utilization of existing components to extend the STREETE framework to workloads that are not very well handled by its original form. This includes the high network loads that cannot be routed through the network without a fine-grained management of the flows. In this part, we diverge from the shortest path routing, which is traditionally used in computer networks, and perform a particular load balancing of the network flows.

In the last part of this thesis, we combine STREETE with the proposed load balancing technique and evaluate the performance of this combination both regarding turned-*off* links and in its ability to keep the network out of congestion. After that, we use our network testbed to evaluate the impact of our solutions on the TCP flows and provide an intuition about the additional constraints that must be considered to avoid instabilities due to traffic oscillations between multiple paths.

Résumé

La consommation d'énergie est devenue un facteur limitant pour le déploiement d'infrastructures distribuées à grande échelle. Ce travail a pour but d'améliorer l'efficacité énergétique des réseaux de cœur en éteignant un sous-ensemble de liens par une approche SDN (Software Defined Network). Nous nous différencions des nombreux travaux de ce domaine par une réactivité accrue aux variations des conditions réseaux. Cela a été rendu possible grâce à une complexité calculatoire réduite et une attention particulière au surcoût induit par les échanges de données. Pour valider les solutions proposées, nous les avons testées sur une plateforme spécialement construite à cet effet.

Dans la première partie de cette thèse, nous présentons l'architecture logicielle "SegmenT Routing based Energy Efficient Traffic Engineering" (STREETE). Le coeur de la solution repose sur un re-routage dynamique du trafic en fonction de la charge du réseau dans le but d'éteindre certains liens peu utilisés. Cette solution utilise des algorithmes de graphes dynamiques pour réduire la complexité calculatoire et atteindre des temps de calcul de l'ordre des millisecondes sur un réseau de 50 nœuds. Nos solutions ont aussi été validées sur une plateforme de test comprenant le contrôleur SDN ONOS et des commutateurs OpenFlow. Nous comparons nos algorithmes aux solutions optimales obtenues grâce à des techniques de programmation linéaires en nombres entiers et montrons que le nombre de liens allumés peut être efficacement réduit pour diminuer la consommation électrique tout en évitant de surcharger le réseau.

Dans la deuxième partie de cette thèse, nous cherchons à améliorer la performance de STREETE dans le cas d'une forte charge, qui ne peut pas être écoulée par le réseau si des algorithmes de routages à plus courts chemins sont utilisés. Nous analysons des méthodes d'équilibrage de charge pour obtenir un placement presque optimal des flux dans le réseau.

Dans la dernière partie, nous évaluons la combinaison des deux techniques proposées précédemment : STREETE avec équilibrage de charge. Ensuite, nous utilisons notre plateforme de test pour analyser l'impact de re-routages fréquents sur les flux TCP. Cela nous permet de donner des indications sur des améliorations à prendre en compte afin d'éviter des instabilités causées par des basculements incontrôlés des flux réseau entre des chemins alternatifs.

Nous croyons à l'importance de fournir des résultats reproductibles à la communauté scientifique. Ainsi, une grande partie des résultats présentés dans cette thèse peuvent être facilement reproduits à l'aide des instructions et logiciels fournis.

Contents

Chapter 1

Introduction

Motivations

Advances in network and computing technologies have enabled a multitude of services — *e.g.* those used for big-data analysis, stream processing, video streaming, and Internet of Things (IoT) — hosted at multiple data centers often interconnected with high-speed optical networks. Many of these services follow business models such as cloud computing, which allow a customer to rent resources from a cloud and pay only for what she consumes. Although these models are flexible and benefit from economies of scale, the amount of data transferred over the network increases continuously. Network operators, under continuous pressure to differentiate and deliver quality of service, often tackle this challenge by expanding network capacity, hence always adding new equipment, or increasing the rates of existing links. Existing work argues that, if traffic continues to grow at the current pace, in less than 20 years network operators may reach an energy capacity crunch where the amount of electricity consumed by network infrastructure may become a bottleneck and further limit the Internet growth [1].

Major organizations have attempted to curb the energy consumption of their infrastructure by reducing the number of required network resources and maximizing their utilization. Google, for instance, created custom Software Defined Network (SDN) devices and updated their applications to co-operate with the network devices. In this way, it was possible to achieve near-100% utilization of intra-domain links [2]. With this smart traffic orchestration, Google was able to increase the energy efficiency while providing a high quality of service. However, such traffic management requires co-operation between the communicating applications and the network management, a technique that is out-of-reach of ordinary network operators.

As the network operators do not have control over the applications that use the network, they over-provision the network capacity to handle the peak load. Furthermore, to prevent failures due to vendor-specific errors, it is not exceptional for a big network provider to implement vendor redundancy in its core network. Vendor redundancy consists of using devices from multiple vendors in parallel. The Orange France mobile network blackout in 2012 [1] is an example of a vendor specific error where a software update introduced a bug both on the primary and on the backup device. To avoid these situations,

[1] http://www.parismatch.com/Actu/Economie/Orange-revelations-sur-la-panne-geante-157766

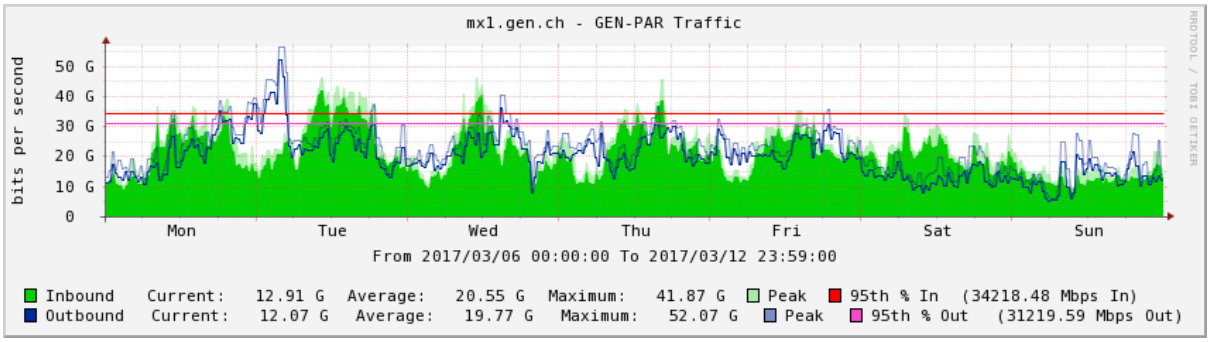


Figure 1.1: Traffic passing through the 200Gbps link between Paris and Geneva. Geant Network [3]

some network operators double the capacity by using backup devices from multiple vendors. All these approaches contribute to highly over-provisioned networks. For example, evidence shows that the utilization of production 200Gbps links in the Geant network can be as low as 15% even during the periods of highest load (Fig. 1.1). Moreover, during the off-peak times during night and weekends, the network utilization is even lower. It drops to almost 5% of the link capacity.

As a result of all this over-provisioning, the energy efficiency of operator networks is low: devices are fully powered all the time while being used only at a fraction of their capacity. This over-provisioning leaves a lot of potential for innovation and for reducing the energy that a network consumes by switching off unused devices and optimizing their usage.

This work is done as part of the GreenTouch consortium, which intends to reduce the net energy consumption in communications networks by up to 98% by 2020[4].

Research problem and objectives

Large-scale computer networks are major electricity consumers whose consumption is not yet optimized. A backbone network device can consume as much as 10kW[2] irrespective of its utilization. This thesis investigates means to improve the energy-efficiency of future backbone networks while retaining the performance constraints.

In particular, we target high-speed IP over Wavelength Division Multiplexing (IPoWDM) operator networks with extremely high data-rates, reaching several hundred gigabits per second per link in currently deployed systems. Unlike dedicated networks – such as those deployed in between data centers, enterprises or research institutes – the company that manages the network devices does not have control over the applications whose traffic traverses the network. Our main lever used to increase the energy efficiency of these networks is agnostic to the applications: we use "traffic engineering" techniques which consist in dynamically changing the paths of data flows transparently to the applications.

This thesis revolves around two ideas. The first natural idea consists in decreasing the energy consumption of the networks by reducing the number of resources that are

[2]Cisco CRS-3 data sheet: `http://www.cisco.com/c/en/us/products/collateral/routers/carrier-routing-system/data_sheet_c78-408226.html`

active during off-peak periods. In particular, we intend to turn *off* transponders and port cards on the extremities of a link. The status of these components is set to sleep mode whenever a link is idle, and they are brought back to the operational state when required. We also term this process of activating and deactivating network links as switching links *on* and *off* respectively. To free certain resources, the network traffic is dynamically consolidated on a limited number of links, thus enabling the remaining links to enter the low power consumption modes. Respectively, when the network requires more capacity, the corresponding links are brought back online to avoid congestion. In other words, we transparently change the paths taken by the application data in the network to accommodate network capacity to the load.

The second idea relies on the observation that a well-utilized network does not consume resources in vein. A proper load balancing technique allows transferring more data with the same resources, avoiding premature updates to higher data rates and thus avoiding the increase of energy consumption due to this update. As detailed later, we conclude that both these methods are necessary and must be used in parallel to achieve a good energy efficiency of operator networks.

We position our work in the context of Software-Defined Network (SDN)-based backbone networks and work within a single operator network domain. Although different methods have already been proposed for reducing the energy consumption of communication networks, our contribution is an end-to-end validation of the proposed solutions. These solutions were both validated through simulations and implemented in a real network testbed comprising Software-Defined Network (SDN) switches and the Open Network Operating System (ONOS) SDN controller. Moreover, we also used the testbed to evaluate the impact of frequent path changes on data flows and conclude that such techniques can be used with minimal impact on application flows.

Contributions and structure of the manuscript

Chapter 1 introduced the motivation of this work: the continuous growth of the number of connected devices relying on the network to access remote services puts a significant load on operator backbone networks. Faster speeds and bigger devices have to be deployed, continuously increasing the energy consumption.

In the rest of this chapter, we detail how to reproduce most of the results presented in this thesis. This work was performed considering the growing concern of reproducibility of scientific work in computer science. We also reference the Android applications which we created to provide an interactive way to test our algorithms and visualize their results.

Chapter 2 sets the context of our work. We provide a general overview of the architectures of backbone networks and explore the panorama of methods used to reduce their energy consumption. Most of these methods propose structural modifications of the network. For example, this includes completely re-designing the physical topology, the network devices, or the applications using the network. In contrast, we follow a less intrusive approach that relies on traffic engineering; a mature technology widely deployed by operators for other goals, such as differentiated services.

We continue the chapter with an extended introduction to the existing traffic engineering techniques and, in particular, to the concept of Software-Defined Network (SDN). In parallel, we present the related work that uses traffic engineering for increasing the energy efficiency of networks. This introduction allows us to conclude that there is still a need for an online solution, capable of reacting fast to changes in network traffic and, in particular, to the unexpected growth of network demand.

The chapter ends with a discussion on the particularities that must be considered when designing a SDN-based solution for high-speed backbone networks. The presented challenges underlie the design of our solutions in the rest of this work. We also introduce the Source Packet Routing In NetworkinG (SPRING)/"Segment Routing" protocol, intended to meet the requirements of traffic engineering in backbone networks, and hence circumvent the challenges above.

Chapter 3 provides the theoretical background needed to understand the results from this thesis. We give an introduction to linear programming and the classical network flow problem known as "maximum concurrent flow". We illustrate the two usual ways to model this problem as a linear program, namely the "node-arc" and the "edge-path" formulations. The first model is used throughout the work to provide a comparison baseline for the quality of our algorithms. At the same time, the edge-path is at the base of efficient approximate algorithms for computing the maximum concurrent flow in a network.

The chapter ends with some mathematical properties of the edge-path formulation and its dual, which is used in the later chapters of the thesis.

Chapter 4 presents the design, implementation and evaluation of STREETE, a SDN-based framework meant to reduce the energy consumption of backbone networks by switching links *off* and *on*.

The main particularity of this work is the capacity to react very fast to changes in the network conditions and, in particular, to unexpected growth of network demand. This is achieved thanks by incorporating a technique that uses two different views of the network to feed the heuristic. Moreover, we use dynamic shortest path algorithms at the core of our solution to speed up the computations and the SPRING protocol to enable low-overhead management of the network by the SDN controller.

We validate the proposed solution in the OMNeT++/INET network simulator and conclude that it performs very well under low network load, but there is room for improvement at high load. This conclusion served as motivation for the work presented in Chapter 5 of this thesis, which tries to make STREETE perform well in any network conditions while being able to compute a solution on the evaluated backbone networks quickly.

This chapter also presents the SDN-based testbed built for validating the proposed solutions experimentally. This testbed allowed us to highlight an additional weakness of STREETE. These results motivated the study done in Chapter 6, which allows for improving our framework further.

Chapter 5 presents two load balancing methods proposed for the STREETE framework to enable better utilization of active network resources.

The first method provides a consistent improvement over the shortest path routing, but the output quality is not very stable and did not satisfy our expectations. We present the reasons behind this behavior and continue with the second, more elegant, load balancing method that is based on state of the art for approximately solving the maximum concurrent flow problem. It builds on the work done by George Karakostas [5] to produce a complete solution for near-optimal Software-Defined Network (SDN) - based online load balancing in backbone networks. This solution also leverages the power of SPRING source routing protocol to reduce the complexity and cost of centralized management.

Chapter 6 starts by unifying the results from the two previous chapters and evokes the problem that the combination of STREETE with load balancing techniques may produce a lot of route changes for network flows which may affect network performance. In particular, the change in end-to-end delay can be interpreted by the TCP protocol as a sign to slow down its transmission. Taking into consideration that TCP is, by far, the most used transport protocol, this detail can have a severe impact on the overall network performance.

In the second part of the chapter, we use our network testbed to evaluate if frequent route changes can negatively impact the network performance. We conclude that the drop in the bandwidth of network flows can be significant if routes change too frequently. Moreover, network instabilities may happen as a result of bad inter-operation between the STREETE protocol and TCP if no special care is taken to avoid them.

We quantify the tolerable frequency based on data from our analysis and conclude the chapter with a discussion on the tuning to be done in the STREETE framework to achieve a stable, near-optimal, energy efficient traffic engineering solution.

Chapter 7 concludes the thesis with a summary of main findings, discussion of future research directions, and final remarks.

Reproducing the results

The simulation results from chapters 4 and 5 can be easily reproduced on a Linux system using *Docker*[3]. Some of the simulations rely on CPLEX. This LP solver is free for academic usage and can be downloaded from IBM after an academic verification. If the CPLEX 12.7.1 installation file is provided, it will be automatically detected, and the corresponding results will be generated. If the appropriate version of the installation file is not provided, those simulations will not run.

Fast mode

To make it easy to reproduce the simulations, we provide an archive containing all the necessary files except CPLEX. Reproducing the results becomes as easy as executing the following steps:

[3] `https://www.docker.com`

- Install dependencies (*docker*, *wget*) using the package manager of your distribution.
- Download the archive with all the files:
  `https://radu.carpa.me/these/streeteCode.tar`
- Extract the archive: `tar -xaf streeteCode.tar`
- *(optional)* Put the "cplex_studio1271.linux-x86-64.bin" file in the `streeteCode/installFiles` folder
- Build the docker image and run the simulations: cd streeteCode && ./buildAndRun.sh

In particular, on an Ubuntu host, you can just copy/paste the following lines into the terminal to follow these steps and run all the simulations except the CPLEX optimizations:

```
sudo apt-get install docker.io wget
wget https://radu.carpa.me/these/streeteCode.tar
tar -xaf streeteCode.tar
cd streeteCode
sudo ./buildAndRun.sh
echo "Finished!"
```

The raw simulation outputs will be placed into the `streeteCode/results` folder. An image is also generated for each simulated time point, thus providing a visual representation of the state of the network. An example of such a figure is given in Fig. 1.2 where

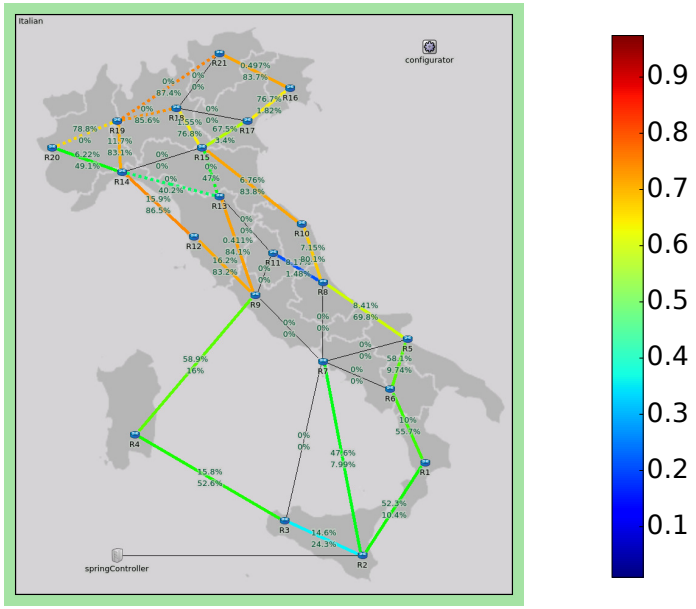


Figure 1.2: Image generated by simulations which illustrates the state of the links in the the Italian network.

the color and the percentage of the links represent their utilization. Two percentages are

shown (one per direction). The color is a helper for easier visualization and corresponds to the biggest of these two percentages represented using the "jet" color scheme (blue = low utilization; red = high utilization [4]). A dotted line means that the link is used only in one direction. A slim black line means that neither of the two directions is used.

The figures which will be presented in chapters 4 and 5 will be generated in the sub-folders of the `streeteCode/figures` folder.

Fallback mode

Alternatively, a much smaller archive, which does not contain the third party installation files, is included with this PDF as an attachment on the first page of the manuscript. The same archive is also disposable via a couple of reliable cloud service:

- `https://s3-eu-west-1.amazonaws.com/me.carpa.archive/Thesis/streeteCode.light.tar.xz`
- `https://drive.google.com/open?id=0B-57pIpvqZLgUThqbTh3ZnNleFk`

The reader must manually provide the omnetpp-5.1-src-linux.tgz [5], inet-3.6.0-src.tgz [6] and qt-opensource-linux-x64-5.9.0.run [7] files in the *streeteCode/installFiles* folder. Furthermore, similarly to the previous section, the CPLEX (cplex_studio1271.linux-x86-64.bin) binary can be added to enable the corresponding simulations.

To be also noted that the *DockerFile* file is self-explanatory. The commands listed in this file can be run directly on an ubuntu 16.04 host machine instead of using *Docker*.

Demo using the Android app

A much more interactive way to visualize the execution and result of the proposed algorithms is done by our Android application which can be freely installed via Google play. The QR code from Fig 1.3 can be used to access the application on Play Store. A short instruction on how to use this application is provided in Appendix 7.2.

Figure 1.3: Access app using Google Play
`https://play.google.com/store/apps/details?id=me.carpa.streete`

[4] `https://en.wikipedia.org/wiki/Heat_map`
[5] `https://omnetpp.org/omnetpp`
[6] `https://github.com/inet-framework/inet/releases/`
[7] `https://download.qt.io/official_releases/qt/5.9/5.9.0/`

Chapter 2

Positioning and related works on energy efficiency and traffic engineering in backbone networks

In the previous chapter, we motivated the need for improving the energy efficiency of computer networks. Before digging into the core of this problem, we begin the current chapter by providing a general overview of the architectures of backbone networks and of solutions from the literature for reducing the energy these networks consume.

Afterward, we focus on a technique for reducing the energy consumption: intelligent traffic engineering. We present existing traffic engineering techniques and, in particular, the concept of Software-Defined Network (SDN). These practical details will allow us to understand better the state of the art on SDN-based traffic engineering in general and especially of energy efficient traffic engineering in backbone networks.

We finish the chapter by discussing the particularities that must be considered when designing a SDN-based solution for high-speed backbone networks. The presented challenges will define the design of our solutions in the rest of this work. This chapter also introduces the Source Packet Routing In NetworkinG (SPRING) protocol, designed for meeting the requirements of traffic engineering in backbone networks and for addressing some of the mentioned challenges.

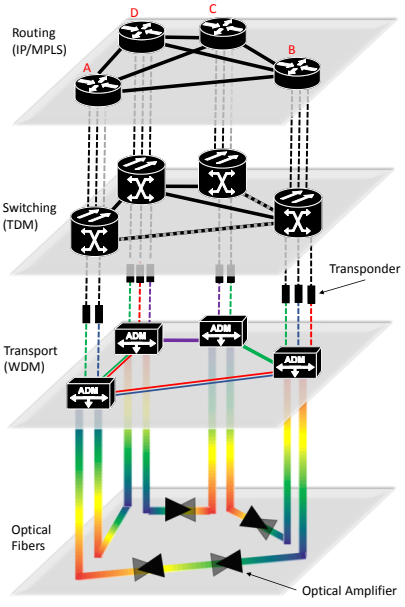


Figure 2.1: Architecture of a backbone network

2.1 Architecture of backbone networks

IP is now the *de-facto* protocol for transporting the data all around the world. It is also generally accepted that the optical technologies are the best solution for high-speed, long-distance communications. Current commercial products allow single-carrier optical transmissions at 200Gbps between nodes at 2000km distance [6]. To make the best use of the costly optical fibers, operators deploy multi-layered architectures. Fig. 2.1 shows a logical separation of the layers often found in an operator's network.

2.1.1 Physical layer

At the bottom of the architecture, we illustrate the actual optical fibers. Due to certain impairments, the signal passing through the fiber attenuates with the distance. As a result, optical amplifiers must be used to amplify the signal. Unfortunately, these devices also amplify the noise. There is hence a limit on the maximum number of amplifiers that can be used in series before the data becomes unrecoverable from the noise. An Optical-Electrical-Optical (OEO) conversion of the signal is needed to re-generate a new signal and therefore achieve longer distances.

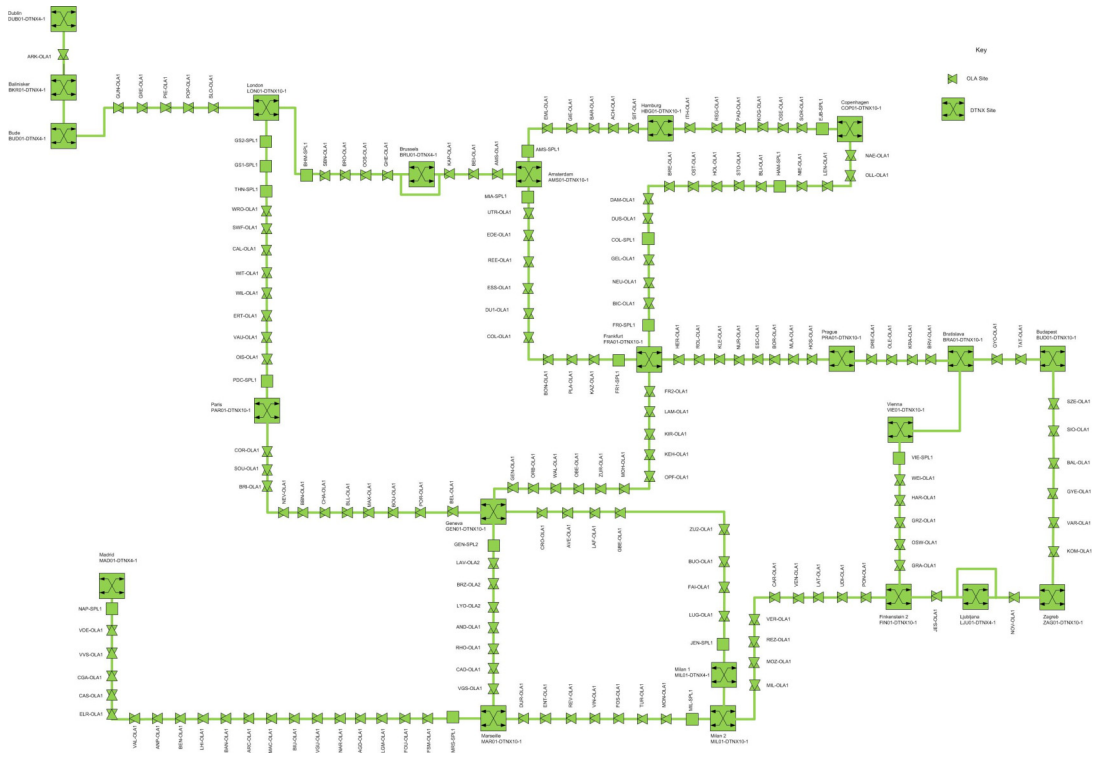


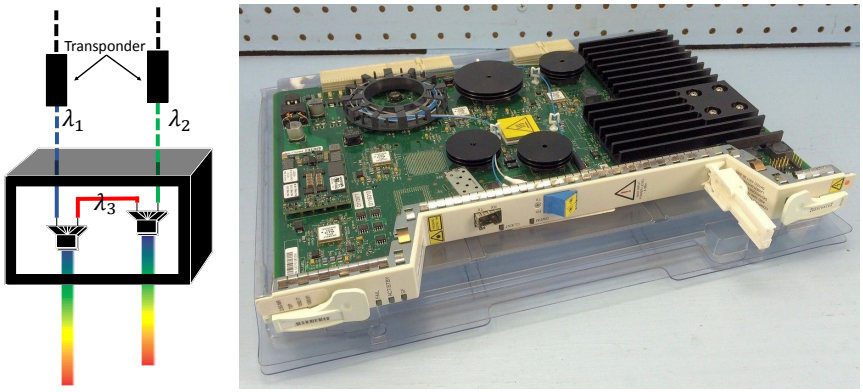
Figure 2.2: The terabit capable part of the Geant network
http://www.geant.net/Network/The_Network/Pages/Terabit-Network.aspx

The distance between the optical amplifiers is approximately 80km. By using multiple amplifiers in series, manufacturers advertise that their devices can transmit a 100Gbps signal at a 5000km distance without the need for OEO conversion [6]. As a result, OEO conversion is usually not required between the points of presence except for submarine communication cables.

The energy consumption of the amplifiers is small. For example, the authors of [7] affirm that the most common type of amplifier, the Erbium Doped Fibre Amplifier (EDFA), consumes approximately 110W to amplify all the wavelengths in the fiber. Nevertheless, the number of amplifiers may be considerable. Fig. 2.2 illustrates a small part of the physical topology of the Geant network. The little dots correspond to the various optical amplifiers along the fibers while the bigger squares correspond to WDM devices presented in the next section. As a result of their large number, the total power consumption of the optical amplifiers in a backbone network is not negligible.

2.1.2 Wavelength Division Multiplexing (WDM) layer

Second from the bottom is the Wavelength Division Multiplexing (WDM) layer. This layer operates on a wavelength (λ) granularity. Fig. 2.3a illustrates the operation of this layer at the node A from Fig. 2.1. The two signals from the upper layer are converted into two different wavelengths λ_1 and λ_2 and are multiplexed into the optical fibers by Add Drop Multiplexers (ADM). Historically, static optical splitters and cross-connects performed this job, but modern systems rely on Reconfigurable Optical Add Drop Multiplexers (ROADM) for remotely reconfiguring the optical layer.



(a) WDM layer at node A

(b) Transponder card

Figure 2.3: Insights on the WDM layer

It is important to notice that the wavelength λ_3 is directly switched from one fiber to another without any additional treatment. Jumping forward, we can see that this wavelength is perceived by the upper, TDM, layer as a direct connection between the nodes B and D; it "bypasses" the upper layers of the node A. That is why the literature refers to this kind of connections by the name of “optical bypass”.

The WDM technology enables the transmission of multiple terabits per second over a single optical fiber. Unfortunately, the number of wavelengths in a fiber is limited, and fine-grained traffic separation is difficult to achieve. This work is done by the higher levels of the core network hierarchy by multiplexing/demultiplexing multiple low-speed communications into/from an optical lambda.

We rely on the model of energy consumption presented in [7]. After consulting a large number of commercial product datasheets, the authors conclude that the consumption

of an Add Drop Multiplexers (ADM) device (the big box in Fig. 2.1) can be modeled as $d \cdot 85W + a \cdot 50W + 150W$, where d is the number of optical fibers entering the node; and a is the add/drop degree, the number of northbound interfaces connected to transponders. We must add to these values the power consumption of transponders, which are represented with small black rectangles on Fig. 2.1. They do complex electrical processing to generate the optical signal adapted for transmission over a long distance. The consumption of transponders, given by the same source [7], is summarized in Tab. 2.1.

Speed (Gbps)	**Consumption (W)**
10	50
40	100
100	150
400	300*

* mathematical projection

Table 2.1: Power consumption of transponders

To be noted that transponders correspond to cards (Fig. 2.3b) plugged into another network device. They are not independent devices as it can be viewed in Fig. 2.3a.

2.1.3 Time-Division Multiplexing (TDM) layer

The Time-Division Multiplexing (TDM) layer is usually built with the old SDH/SONET or the newer OTN(G.709) protocols. Their goal is to multiplex/demultiplex multiple low-speed connections into a higher speed λ for better spectral utilization of the optical fibers. In the absence of this layer, the point-to-point connection between the nodes A and C in the packet (IP/MPLS) layer would have to use an integral wavelength regardless of the needed capacity. As an example: assume that a client of the operator leases a dedicated connection between the nodes A and C with a capacity of 1Gbps. Without the TDM layer, the operator must reserve a whole wavelength (which has a potential of transmitting 400Gbps), to satisfy the clients' demand for 1Gbps. Fig. 2.4 schematically represents the work of the TDM layer at node A. The gray and the black signals are multiplexed into a single one, with higher data rate.

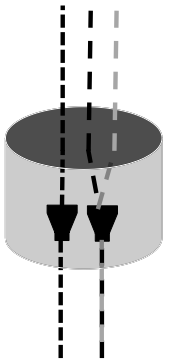


Figure 2.4: TDM layer at node A

The power consumption of the TDM ports is presented in Table 2.2.

2.1.4 Packet layer

Finally, the IP/MPLS layer corresponds to the packet based transmission. Similarly to the previous layers, the network topology seen by the packet layer corresponds to a virtual topology which does not necessarily matches the physical optical fibers. In particular, in

Speed (Gbps)	Consumption (W)
10	34
40	160
100	360
400	1236*

* mathematical projection

Table 2.2: Power consumption of TDM/OTN ports (bidirectional)

our example, what seems to be direct connections between the nodes A-C and B-D are virtual tunnels provided by the TDM and, respectively, WDM layers.

A particularity of the packet layer in operator networks is that it frequently relies on MultiProtocol Label Switching (MPLS) forwarding rather than IP routing. This is because the "best effort" service provided by IP is insufficient in today's convergent networks. Intelligent traffic management is needed to enable a network to be shared between delay-critical audio flows and best-effort traffic.

In this layer, a single device can consume as much as 10 kilowatts, 75% of which is due to the slot and port cards regardless of their utilization [7].

2.1.5 Layered backbone networks, from theory to practice

Unfortunately, there is no consensus on the best combination of the previously presented layers and on how to better map the IP traffic into the optical fibers. Multiple different solutions can be encountered in currently deployed operator networks as a result of searching for a balance between complexity, features, and cost. Small players will tend to choose convergent solutions to reduce the operational cost of their network. On the contrary, larger operators tend to have a very clear separation between the layers and even have separate teams responsible for the operation and management of each layer.

Moreover, each device manufacturer tries to push the technologies at which they are best. For instance, Cisco, one of the leaders in packet-based forwarding, attempts to encourage the use of entirely convergent packet over fiber solutions, promising a better link utilization at a fraction of the cost of implementing a network with TDM devices. On the other hand, manufacturers with a long background in the telecommunication industry promote TDM solutions for interoperability with legacy devices and to offload the transit traffic from the IP/MPLS routers.

The most common architectures are IPoOTN and IPoWDM. Each one of them has its strengths and its weaknesses which we present in the following sections.

IP over OTN (IPoOTN)

In theory, Optical Transport Network (OTN) is just a Time-Division Multiplexing (TDM) protocol standardized by the International Telecommunication Union (ITU) in the G.709 recommendation [8]. At the same time, in the networking lingo, IPoOTN refers to a group of integrated solutions: where the TDM sub-wavelength multiplexing and the management of WDM lambda wavelengths are both done by the same device. This allows

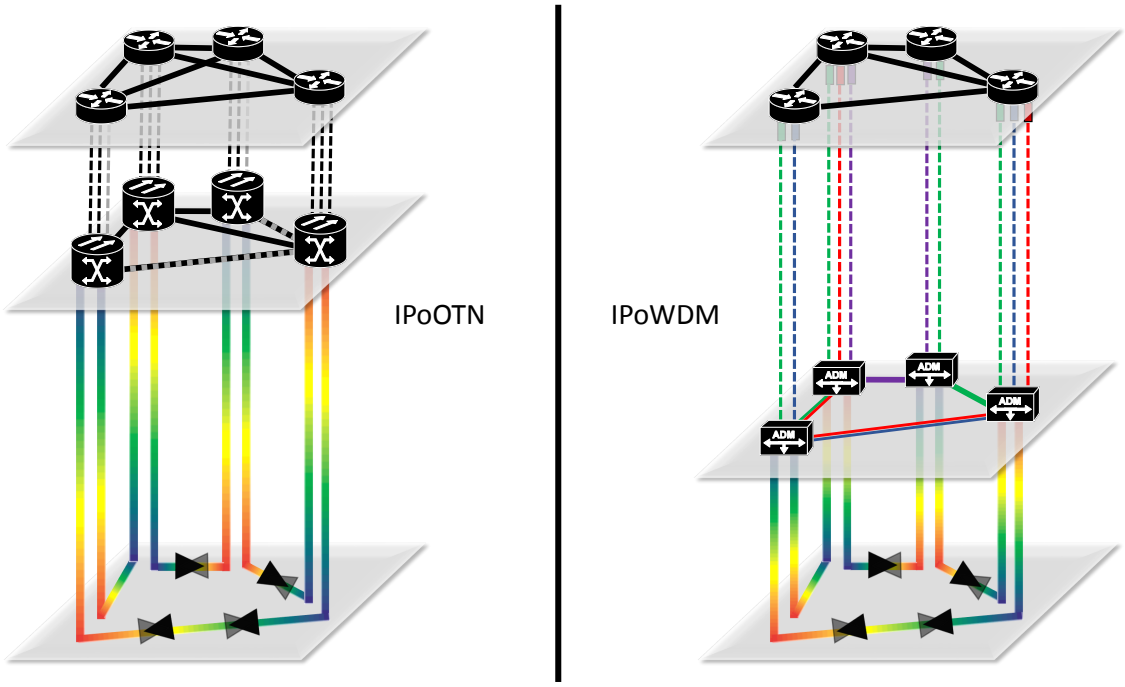


Figure 2.5: The architecture of IPoOTN and IPoWDM networks

leveraging the advantages of each layer while reducing the management overhead. The IP/MPLS packets are still processed by separate, specialized routers and are transmitted to the Optical Transport Network (OTN) devices via short-reach optical or copper links. The latter TDM-multiplexes multiple low-speed connections from the packet router into high-speed flows. Integrated transponders are afterward used to generate an optical signal, and an integrated Reconfigurable Optical Add Drop Multiplexers (ROADM) injects the lambda into the optical fiber.

For historical reasons, IPoOTN is the preferred choice of large transit operators. The OTN devices are designed to be backward compatible with legacy TDM protocols (SDH and SONET) which were deployed in times when most network traffic consisted of audio flows. OTN devices allow gradual transitions to higher speeds without having to completely re-design the existing network.

Another advantage of OTN solutions is the fact that transit traffic is kept at TDM layer instead of passing through multiple IP routers and the fact that existing protocols are very mature and allow end-to-end management through multi-vendor networks. Nevertheless, there are no technical limitations to implement these services, i.e. multi-tenancy and agile management of transit traffic, in the packet layer. Moreover, IP/MPLS packet forwarding can be seen as a Time-Division Multiplexing (TDM) technique. The services provided by the packet and the TDM layer are, to some extent, redundant. That is why in the modern era, with most network traffic being packet-based, IPoWDM architectures that we present in the next section are becoming increasingly popular.

IP over WDM (IPoWDM)

In IPoWDM solutions, the IP routers integrate re-configurable optics which allow generating the optical lambda wavelength directly. They are then mapped into the optical

fibers by a separate Reconfigurable Optical Add Drop Multiplexers (ROADM). This solution moves the cost of transponder/optics from optical layer equipment to the router.

The IP/MPLS packets are directly mapped into optical fibers without the need for the intermediate TDM layer. The aggregation of multiple small, low speed, flows into a high-speed flow is thus done at packet layer. The multi-tenancy and Quality of Service (QoS) are also implemented using the MPLS and RSVP-TE protocols at the packet level.

This level allows the most flexibility in traffic management. The disadvantage of this solution is in need to process all the transit packets in the IP layer. The packet processing per gigabit is, unfortunately, more costly than the TDM processing: the most power-efficient devices consume 7W per Gbps for IP/MPLS processing, which is slightly worse than TDM processing shown in Tab. 2.2. On the other hand, an optical bypass can be used when it is justified to avoid transit processing.

2.2 On improving the energy efficiency of IPoWDM backbone networks

In the previous section, we presented the different levels which can be encountered in backbone networks and described the two types of topologies deployed in production networks, namely IPoOTN and IPoWDM. Due to the relative simplicity of the IPoWDM and to the fact that this topology is better suited for the currently predominant IP traffic, we decided to target it in this thesis.

Following a pioneering work [9] on reducing the energy consumption of computer networks, numerous other authors dedicated themselves to improving their energy efficiency. In this section, we start by a quick general overview of the different approaches proposed in the literature and targeting IPoWDM networks. This introduction will lead us to the introduction of the domain of energy efficient traffic engineering in the IP layer.

2.2.1 Energy efficient network design

Some works approach the problem of network energy efficiency from a clear-state design perspective. [10], for example, proposes to incorporate the energy efficiency as a decision variable from the earliest stage of the network construction: the design of the physical network topology. The authors inspect how different physical topologies result in various overall power consumption. Direct fibers between the nodes reduce the average hop count and allow to avoid the power-hungry electrical processing of transit traffic. However, such direct long links need more optical amplifiers which consume power regardless of the amount of traffic passing through a link. The authors investigate the balance between these factors.

With the same goal of reducing the network hop counts, the GreenTouch consortium proposed [4] to re-design the modern Internet and use long reach point to point optical links to directly connect the end users and business to an extremely well-meshed network core.

On the other hand, most researchers follow a less radical approach and consider that the physical topology of optical fibers cannot be changed. A multitude of works [11][12][13] search to optimize the network design for energy consumption by selecting

the best combination of IP and optical devices at each point of presence. Such works usually use linear or nonlinear models because the computational time is not crucial in the network design phase.

Another approach to energy efficient network design relies on the following idea: the best way to reduce the energy consumption of the networks is to completely avoid using the network. A very attractive solution relies on caching the content as close to the end user as possible. In particular, [1] proposes to use Content Delivery Networks (CDNs) and FOG[1] computing to avoid the growth of the network energy consumption beyond what power grids can sustain. To be noted that CDNs are already widely used by content providers such as Netflix to both unload the network core and improve the user experience [14]. There are thus no technical limitations to implement this kind of solutions.

An alternative way to reduce the amount of network traffic, applicable when CDNs can not easily cache the content, is the redundancy elimination. Individual devices are placed in the path of network flows, and compression mechanisms are afterward used to reduce the amount of data sent trough the system. Unfortunately, when it comes to evaluating the energy consumption of solutions employing redundancy elimination, the research community diverges. [15], for instance, affirms that the increase of energy consumption due to compression may be higher than the reduction due to transmitting fewer data into the network. Other works are more optimistic [16] [17]. For example, the authors of [17] propose a solution which combines compression with energy efficient traffic engineering which we present in a later section. The authors take special care to avoid the unfavorable case shown in [15].

2.2.2 Energy efficient device design

The power consumption of network devices per Gbps of traffic is "naturally" decreasing thanks to the Moore's Law and continuous miniaturization of electronic components. Nevertheless, the energy proportionality of modern network devices remains surprisingly bad. For example, [11] measured the energy consumption of IP/MPLS routers and concluded that the load has a marginal impact on the power consumption. This result was also confirmed by measurements done by our team on a network switch (Fig. 2.6).

To increase the energy proportionality of network devices, the Institute of Electrical and Electronics Engineers (IEEE) standardized the 803.2az protocol[19]. This solution puts sub-components of network interface to sleep for short time periods when no packet is passing through the interface. When packets arrive into the interface's packet queue, the latter is awakened and the data is transferred. Small Office Home Office (SOHO) devices implementing the 803.2az protocol are already commercialized.

Unfortunately, transitions to/from the sleeping mode uses both energy and time. As a consequence, the efficiency of this approach is small when interfaces are highly utilized. For example, when waking from a low power state, the energy consumption of the interface may spike up to 4 times the power needed under normal operation [20]. Experimental analysis shows that the power savings are close to zero when the utilization of the interfaces reaches 20%[21]. Furthermore, this approach is only applicable in access networks and is not efficient on backbone links due to a very short inter-packet arrival time [22]. To illustrate the reason, assume a 100Gbps interface which is utilized at 10%.

[1] `https://en.wikipedia.org/wiki/Fog_computing`

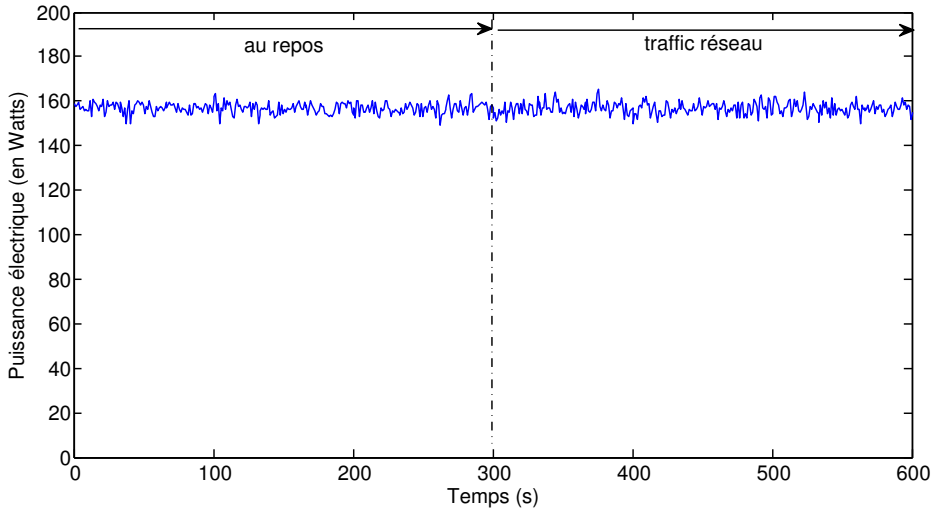


Figure 2.6: Energy consumption of a network switch measured at idle during 300s and at full load afterwards. Source : [18]

If it receives uniformly distributed 1000byte packets, it will receive a packet every 0.8 microseconds. With existing technologie, the time needed for an interface to transit to a low power state or to be awakened from sleep is much longer. Evidences show that modern commercial transponders need that tens of seconds, or even minutes, to become operational after a power cycle [2], [23], [24]. Fortunately, the research community affirms that it is possible to reduce this time and that transitions to/from a low power state are possible in milliseconds [25] by keeping part of the interface active, like in the 802.3az protocol.

Alternatively, Adaptive Link Rate (ALR) technologies propose to reduce the energy consumption of network devices and achieving a better energy proportionality by dynamically adapting the maximum rate of the network links. The port modulation is changed when lower data rates are needed and hence adjust the rate. For example, at low demands, a Gigabit Ethernet port will be passed to a 100Mbit mode. By doing that, the power consumption will be reduced. [26] analyses the possibility of doing modulation scaling in optical backbone networks.

New technologies may emerge soon to reduce the power consumption of network devices even further. The STAR project [27], for example, promise shortly to present a prototype of an all-optical switch which can be used to switch terabits worth of wavelengths while only consuming 8 watts.

We believe that the solutions which rely on energy efficient network design, CDNs, redundancy elimination, and energy-efficient device design are indispensable in a genuinely eco-responsible solution. However, these are long-term solutions. On a shorter time scale, it is possible to increase the energy efficiency of existing networks by using intelligent provisioning and traffic management, which we present in the next subsections.

[2] Juniper PTX Series Router Interface Module Reference

2.2.3 IPoWDM cross-layer optimization: leveraging optical bypass

The multi-layered architecture of backbone networks enables the possibility to save energy through a joint optimization of the IP and WDM layers. For instance, processing IP/MPLS traffic in the electronic domain consumes more power per Gbps than bypassing the intermediate node in the optical domain. The all-optical transmission is cheap: transporting an additional point-to-point WDM wavelength over existing infrastructure is virtually "free" regarding energy consumption of intermediate point of presence. However, each wavelength passing through the optical layer must be generated by energy-hungry transponders.

As a consequence, there are various works which search for a balance between routing the traffic in the IP/MPLS layer and bypassing the upper layers in the optical domain. The authors of [28] and [29] show, using mathematical models, that a proper technique to balance the traffic between the IP/MPLS layer and the optical bypass via λ-paths can increase the network performance and decrease the energy consumption.

Unfortunately, there are also works which show that power saving potential of this technique is not as clear as it seems. The authors of [30] conclude that optical bypass occurs to consume more energy than IP processing if wavelengths are filled at less than 50%. For example, using a dedicated wavelength, and thus dedicated transponders, to transport a small (100Mbps) flow and bypass IP processing at a couple of intermediate nodes will consume much more energy than processing the same 100Mbps of traffic in the IP layer of the intermediate nodes. The wavelengths must be adequately filled to profit from the advantage of the optical bypass. The authors of [31] also conclude that there is no general solution to the cross-layer optimization problem. Its usefulness always depends on the network topology (dimension, connectivity index) and the network load. The authors of [32] [33] used MILP models to analyze the saving potential in different layers and conclude that optimizing the traffic in the IP layer allows the most energy savings while dynamically re-configuring the optical domain has little impact on the energy efficiency in real networks.

To make things even more complicated, the number of WDM lambdas sharing a fiber is limited. Moreover, more transponders mean more place in the network device chassis. With current technological limitations on the number of wavelengths per fiber, it is impossible to efficiently create a well-meshed topology of point-to-point WDM wavelengths in a big network.

After reading all this evidence, we decided not to do cross-level optimizations and to concentrate instead solely on increasing the network energy efficiency via traffic engineering in the packet layer.

2.2.4 Energy efficient network operation

The energy efficient network operation techniques act by optimizing the utilization of resources which are already present in the network. These approaches contrast to the energy efficient network design, which is used to decide how to deploy resources and build an entirely new network.

A popular technique which allows reducing the energy consumption of computer net-

works rely on aggregating traffic flows over a subset of network resources, allowing to switch *off* a subset of network components.

In an ideal world with energy proportional network devices, the energy consumption would scale linearly with the utilization. As a result, there would be no way to improve the energy efficiency by extending the paths of network flows. However, we already presented in a previous section (2.2.2), that the consumption of the modern devices is not linear and maximizing the device utilization increases its energy efficiency.

In this thesis, we follow the trend of optimizing the energy consumption in the IP layer. First of all, the energy consumption of the IP layer was shown to dominate the one in the optical layer [34]. Secondly, the optical fibers are frequently shared among different operators and re-optimizations in the optical layer without turning amplifiers *off* has a marginal impact on the consumption of the network [32] [33]. It is also important to notice that, while transponders can be quickly shut down, switching amplifier sites on and off is a time-consuming task [35]. Lighting *on* an intercontinental fiber may take several hours.

We also limit ourselves to only turning *off* links, i.e. router ports and transponders, without touching to network devices. The main reason is that there is no clear separation between core and aggregation devices in production networks. Any core router is, at the same time, used to aggregate client traffic. Turning the device *off* would cut the connectivity of all connected clients. To be noted that some works proposed to virtualize the network devices and transparently forward optical access/aggregation ports to an active node using the optical infrastructure [36][37][38]. We believe that this solution has significant shortcomings. One of them is the fact that access ports are not adapted to long-distance transmission. We prefer thus to avoid switching *off* network nodes.

To switch *off* a subset of network resources, traffic engineering techniques are needed to wisely optimize the paths for traffic flows while satisfying a set of performance objectives. In the following section, we provide a detailed analysis of what traffic engineering is and off associated related work in the field of applied energy efficient traffic engineering via turning links *off*.

2.3 Traffic engineering

The goal of traffic engineering is to change the paths of network flows to satisfy a set of performance objectives. For example, good energy savings can be achieved by employing traffic engineering and turning *off* idle resources.

When it comes to implementations, two main different traffic engineering approaches exist: proactive and reactive. In the proactive case, actions are taken upfront to avoid unfavorable future network states. In contrast, reactive solutions will dynamically adapt to adverse events which already happened in the network. An example of such adverse event is network congestion. A network element is said to be congested if it experiences sustained overload over an interval of time. Congestion almost always results in the degradation of the Quality of Service (QoS) perceived by the end users. If it is possible, a traffic engineering solution will re-distribute the traffic to avoid congestion at any device in the network. For example, if a network resource becomes congested due to an increase in network traffic, the overall network may be reconfigured to reduce the load on this

resource.

Some of the most common traffic engineer optimization objectives are load balancing to avoid congestion; path protection in case of a link or device failure; differentiated service for distinct classes of network flows.

In this work we are particularly interested in two traffic engineering objectives: i) minimizing the energy consumption of the network by releasing links and turning them *off*; ii) increasing the efficiency of resources by increasing their utilization while avoiding congestion.

A significant challenge of traffic engineering solutions is to react to changes in the network automatically. To detect these changes, traffic engineering has to gather information about the state of the network. Therefore, once a change in network condition is detected, techniques are required to configure the behavior of network elements to steer traffic flows accordingly. Such functions, embedded into data and control planes, were traditionally performed in a decentralized manner, but more recently many traffic engineering schemes have re-considered the centralization of control functions.

In the remaining part of this section, we present the most popular traffic engineering techniques. We start with the decentralized and reactive solutions relying on the Interior Gateway Protocol (IGP). Afterward, we introduce the centralized techniques relying on the IGP weight assignment and those using MPLS with Resource reSerVation Protocol Traffic Engineering (RSVP-TE) or Path Computation Element (PCE). We finish with an introduction to SDN and the related works in the field of energy efficient traffic engineering.

2.3.1 Traffic engineering using the IGP

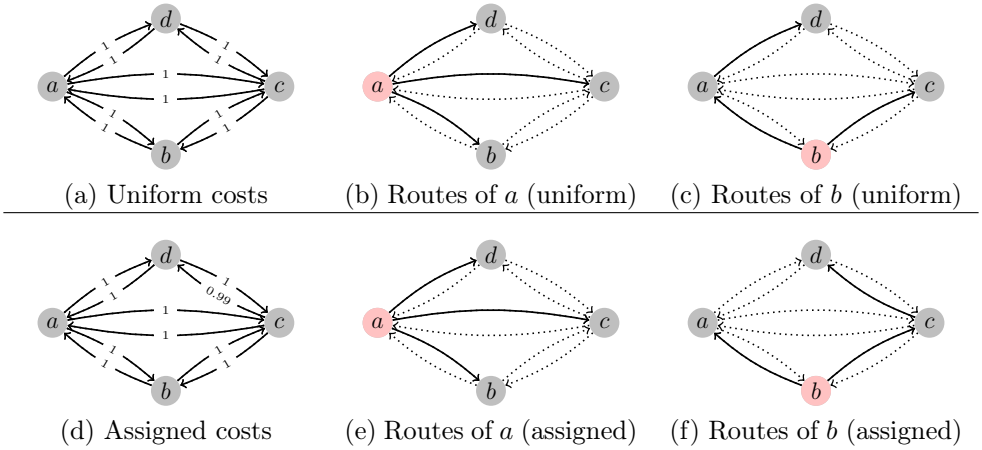
A well-known reactive distributed approach to traffic engineering relies on Interior Gateway Protocol (IGP), such as OSPF[39] or IS-IS[40]. It is easy to enable this kind of traffic engineering because it relies on a single protocol, which is almost always present in the network.

The Interior Gateway Protocol (IGP)-based traffic engineering allows to dynamically adapt the routes of traffic flows across the network based on available network resources; the routes are dynamically changed to meet QoS requirements. To achieve that, the network devices periodically flood the network with information about the load of the directly connected links. This information allows each network device to compute the paths through the network in a way which avoids heavily used spots.

A constrained shortest path algorithm is used for this purpose. However, special care must be taken to prevent oscillations: when all devices start using a lowly loaded link and congest it. This action, in its turn, makes all the devices avoid the link. For instance, the analysis of the OSPF-based traffic engineering shows that it may take whole minutes to converge to a steady state after an unfavorable network event [41]. Moreover, a detailed analysis of OSPF-TE indicates that its communication overhead is extremely high even under normal operation of the network [42]. As a result, the IGP-based traffic engineering is reserved for small networks and is seldom used in production.

2.3.2 Traffic engineering via IGP metric assignment

A way to profit of centralized network-wide optimization while maintaining a low management overhead is to assign metrics to network links. As a result of this action, the classical shortest path algorithms used by the IGP will diverge from the initial routes.



(a) Uniform costs (b) Routes of a (uniform) (c) Routes of b (uniform)

(d) Assigned costs (e) Routes of a (assigned) (f) Routes of b (assigned)

Figure 2.7: Example of IGP metric assignment for traffic engineering

Fig. 2.7 illustrates this method. The first 3 figures show the result of a shortest path computation done by nodes a (Fig. 2.7b) and b (Fig. 2.7c) with the uniform IGP metric from Fig. 2.7a. It so occurs that both the node a and b use the link ad to route traffic towards the node d, which may be sub-optimal if both nodes a and b send a lot of data towards d. By reducing the metric (cost) of the link cd in Fig. 2.7d-2.7f we ensure that b will prefer this link for reaching d.

The disadvantage of solutions relying on the IGP metric is in the fact that they are limited by the use of shortest path to compute the routes. However, a theoretical property affirms that an optimal routing is possible in the network by using only D+M paths in the network [43] , where D is the number of demand pair and M the number of links. As a result, such solutions can actually approach very close to the optimum.

2.3.3 Traffic engineering using MPLS

A more elegant reactive traffic engineering solution relies on MultiProtocol Label Switching (MPLS); a protocol frequently used in backbone networks [3]. One of the strengths of MPLS is its ability to explicitly set the path of the packets by using virtual tunnels instead of transmitting them over the shortest path.

Two different approaches to MPLS-based traffic engineering exist, namely distributed and centralized. In the distributed case, to achieve the TE objective, information about the network is propagated between the nodes in the same way as in the case of IGP-based TE. The convergence speed is drastically improved thanks to the fact that each device tries to reserve resources before actually changing the routes of network flows. The RSVP-TE

[3]Juniper PTX [44], one of the biggest backbone routers, is a pure MPLS router and is not capable of doing IP processing at its maximum advertised speed.

[45] protocol is used between nodes to reserve resources along a path in the network. MPLS is then used to create a virtual tunnel which bypasses the shortest paths. It is also possible to establish multiple disjoint tunnels from a source to a destination. Equal-cost Multi-path (ECMP) is afterward used to split the traffic across multiple tunnels towards an egress device. Even if, by using resource reservation, the convergence speed increases and the risk of oscillations decreases, this solution still leads to under-optimality due to the greedy local selection of network paths by each device.

The centralized MPLS-based traffic engineering is the precursor of SDN. In this case, a Path Computation Element (PCE) is used. A PCE is a special device that monitors the networks and programs the paths into the network devices. We do not spend time on this case because it is fundamentally equivalent to SDN, which is explained in the next section.

2.3.4 Traffic engineering using SDN

Software-Defined Network (SDN) is a novel paradigm which seeks to bring even more agility into the networks. It can be seen as an evolution of the Path Computation Element (PCE)-based TE and allows the integration of the system with custom-made software to enable resource management at a level which is unreachable by traditional network protocols. In particular, SDN allows a lot of flexibility when it comes to traffic engineering which can solve the shortcomings of the previous solutions by leveraging centralized traffic management.

There is no consensus on what Software-Defined Network (SDN) means. It is even usual to encounter conflicting points of view. For example, SDN is frequently considered to be a synonym of OpenFlow. In contrast, in this work, we adhere to a more general definition of SDN, given in the RFC 7426 [46]: *A programmable network approach that supports the separation of control and forwarding planes via standardized interfaces.*

An essential element of SDN is thus the separation of the control plane from the data plane. In practical terms, that means that network devices perform tasks that ensure data forwarding (*i.e.* the data plane) whereas management activities (*i.e.* the control plane) are factored out and placed at a central entity termed as the SDN controller.

Fig. 2.8 illustrates the layers of a software defined network.

- At the bottom, we represent the network devices which are simplified to the maximum. Instead of running complex distributed protocols, the network devices are only capable of forwarding packets that arrive through an incoming port towards the correct outgoing port. Henceforth data plane devices will be referred to as SDN switches.

- In the middle is the SDN controller, which executes many of the traffic-engineering requirements on gathering traffic information, performing management and control. SDN controllers enable a lot of flexibility when it comes to traffic engineering. They maintain and exploit network-wide knowledge to execute all required computations. In Fig. 2.8 the controller is illustrated running a topology service, which discovers the network topology; a statistics service to keep track of network status, and a flow rule service to apply routes into the network devices.

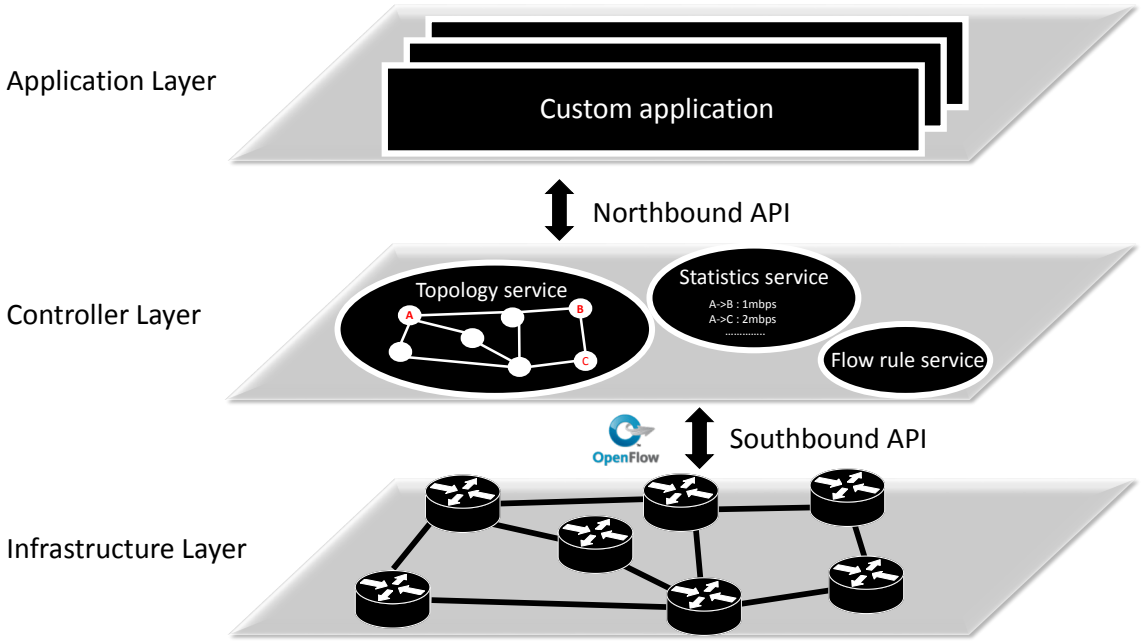


Figure 2.8: The architecture of SDN

- At the top, we show the custom business applications written in the network administrator's language of choice. An example application is cloud infrastructure utilizing a network to provision cloud services for customers.

The communication between the layers is done via abstraction APIs. This comes in contrast to the traditional networks, where the control and data planes are tightly coupled. An effort has been made towards standardizing the interface between the controller and the data plane, generally termed as *southbound* API, and the manner the controller exposes network programmability features to applications, commonly called *northbound* API. The most known *southbound* API is OpenFlow. When it comes to the *northbound* API, it is frequently exposed as HTTP-based RESTful APIs.

In a short period, Software Defined Networks have emerged from a concept into a product which is expected to revolutionize the network management. Some SDN pioneers, including Microsoft[47] and Google[2], have deployed software controlled networks in production environments, increasing the efficiency of their system.

In the next section, we present the challenges which appear when implementing SDN-based traffic engineering in backbone networks.

2.3.5 Energy efficient traffic engineering

Numerous works rely on traffic engineering to reduce the energy consumption of the computer networks. The idea is to modify the paths of network flows in a way to minimize the energy consumption. Some of the works, for instance, set the paths in a way to optimize the network for adaptive link rates, but the majority seeks to turn *off* the links or even devices completely. As noted in the previous section, we position ourselves in-between and concentrate on solutions which only turn-*off* links.

Distributed traffic engineering

One of the few works which rely only on the IGP for energy-efficient traffic engineering is [48]. In this work, every node monitors the utilization of adjacent links and decides whether to switch *off* a link by using a utility function that depends on the energy consumption and a penalty function which we describe later. If a link is chosen for switch *off*, the node floods into the network its will to turn it *off*. All nodes update their IP routing tables via the IGP. As a result, the data flows will avoid the links marked for sleep. Machine learning mechanisms are used to avoid choosing links whose extinctions provoked a congestion in the past. A per-link penalty is used for this purpose: each time a wrong decision caused a degradation of the network performance, the penalty increases, ensuring that the link will not be turned *off* in the near future.

Unfortunately, the authors drastically underestimate the overhead of their solution regarding exchanged messages. In a network with 30 nodes, they estimate that OSPF-TE would flood the system once every 10 seconds. However, a detailed analysis of OSPF-TE [42] shows that it will happen six times per second. If we recalculate the results from [48], we discover that the announced overhead of 0.52% will become a 30% increase in the number of OSPF-TE flooding. This is not negligible because OSPF-TE flooding is costly both in the number of messages and needed processing power.

Another work which relies on the IGP to do energy efficient traffic engineering is [49]. The authors propose to use OSPF-TE to announce the link load into the network. Afterward, if no link has a load greater than a certain threshold, the least loaded link is locally turned *off* by the corresponding node. If the threshold is violated, the last turned-*off* link is awakened. This solution has the weakness of potentially turning *on* links in the network even if this action does not help to solve the congestion.

[50] presents an original solution where a control node is elected in the network. This node then guides the construction of the solution to obtain better overall energy savings while still mostly relying on the distributed operation of the IGP. A subset of this algorithm was implemented using the Quagga software by the authors of [51]. Sadly, the paper provides little insights on the lessons learned from this implementation.

All previously mentioned papers suffer from the biggest problem of solutions relying entirely on the IGP: a slow convergence. That is why centralized solutions, relying on metric assignment or precise routes via MPLS or SDN, are more frequently proposed in the literature.

Centralized traffic engineering

Centralized energy efficient traffic engineering solutions use a *central controller* in charge of executing an optimization algorithm and apply the changes to the network. The centralized solution can be grouped into two categories: i) based on IGP metric assignment, and ii) based on explicit paths. When it comes to selecting between these two techniques, we must note that it was theoretically proven that, for any optimal solution that uses explicit paths, it is possible to find a link weight assignment under which these paths are the shortest [52]. As a result, it is possible to obtain very good solutions using link weight assignment.

The solutions relying on metric assignment act by setting a high metric on the links which have to be turned off. As a result, the traffic is rerouted to avoid these “costly”

links, allowing to turn them *off* effectively. Nevertheless, this approach has a major drawback: temporary network loops can be created during the re-configuration of IGP as a result of weight changes.

Alternatively, using explicit paths enables more flexibility in traffic management like, for example, easier unequal cost multipath forwarding and avoidance of network loops. Pioneer works in this field proposed to use MPLS for data forwarding[53]. Recently, the trend shifted towards using SDN for solutions relying on explicit path.

The panorama of research in these two fields is vast. For example, a recent survey [54], cites more than 50 works which propose to reduce the energy consumption of wired network via centralized traffic engineering. There is a trade-off to be found between the energy efficiency, network performance, and computational complexity. For instance, the optimal solution to the problem of energy efficient traffic engineering can be found using linear and non-linear models. Moreover, these models can easily be augmented to incorporate additional constraints like, for example, path protection [55]; limited number of network reconfiguration per day [56] [57]; redundancy elimination [17]; maximum propagation delay[58]; Quality of Service (QoS) [59].

A constraint which is frequently encountered in SDN based work is the limiting size of flow tables [60] [61]. These works target OpenFlow networks and, as a result, flow coalescing can overflow the flow table which is implemented in a size-limited ternary content addressable memory (TCAM). We believe that this costly constraint can be avoided by using the SPRING protocol instead of OpenFlow [62]. This point of view also seems to be shared by the authors of [63], who are among the first to propose a solution for energy efficient traffic engineering built with the SPRING protocol in mind.

All the previously mentioned linear and nonlinear models cannot be solved to optimality on any, but small, network. That is why the authors usually also propose heuristics intended for use in real networks. Unfortunately, few works report the computational complexity of their algorithms. Such works include [57], reporting up to one hour of computation; [53] who artificially limit the execution time to 300s; [61] reports 10 seconds on a 15 node network; [59] announce between tens of second, up to multiple hours depending on the desired precision.

In our work, we seek to design a solution which can react fast to network condition changes at the expense of trading off the additional optimization constraints. A similar work [64] reports 10 seconds of computation on a 400 node network. However, their algorithm heavily depends on the number of request and the authors evaluated the solution with less than 2000 requests, which is extremely small for such a big network.

2.4 Challenges in implementing SDN-based traffic engineering in backbone networks

Traffic-engineering schemes are increasingly relying on SDN to simplify configuration and management operations. In this thesis, we also rely on SDN for the purpose of network optimization. We make use of the centralized view of (i) the network topology, (ii) running applications and, (iii) traffic demands; to program a network and change its virtual topology according to traffic conditions.

Decoupling the control plane and the data plane produces its drawbacks. For instance,

the protocol traditionally used in SDN networks, OpenFlow, relies on the centralized management to the point that the network becomes nonoperational without a controller. Such a single critical point of failure is unacceptable in backbone networks.

In this section, we discuss the challenges and requirements in designing a SDN-based traffic engineering solution for backbone networks. This will allow us to introduce the constraints which will define the design of our algorithms.

2.4.1 Impact of traffic engineering on network flows

It is worth mentioning that rerouting the flows in the network changes the end-to-end delay perceived by network flows. Compared to Local Area Networks, where the delay variation is small and has little impact on performance, in backbone networks, the delay of two paths can differ significantly. As a result, packet reordering may occur, as depicted in Fig. 2.9 : packet 1 arrives at node b after the packets 2 and 3.

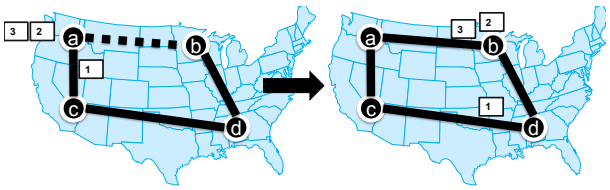


Figure 2.9: Rerouting towards a lower RTT ⇒ packet reordering.

This reordering may influence the speed of the flow. The Transmission Control Protocol (TCP), for instance, may assume a network congestion and reduce the sending rate. Due to that, the networking folklore implies that changing the route of TCP flows will have a severe negative impact on their throughput. The research community has extensively estimated the impact of random packet reordering on the performance of TCP flows. However, rerouting the flows is different from random packet reordering in the sense that reordered packets arrive in bursts at the receiver and may result in a different behavior compared to randomly reordered packets. Moreover, the joint evolution of the congestion window size of multiple TCP flows sharing a single router or link exhibit fluctuations due to the network as a whole and come on top of the standard behavior of TCP [65].

To the best of our knowledge, previous work has not specifically evaluated the impact of frequent flow rerouting on multiple aggregated transported flows. Some works which estimated the impact of reordering on TCP include Bennett *et al.* [66], who were among the first to highlight the frequency of packet reordering and its impact on the throughput of TCP flows. Their results showed that packet reordering has a powerful effect on a network's performance. As a result, the authors of [67], for example, proposed a traffic engineering algorithm which tries to optimize the congestion control and routing jointly. In the meantime, Laor and Gendel [68] have experimentally measured the impact of reordering on a testbed and also concluded that even a small rate of packet reordering could significantly affect the performance of a high bandwidth network. A lot of work was afterwards performed to increase the tolerance of TCP to packet reordering [69] [70] [71] [72].

Finally, a solution to detect unneeded retransmissions caused by packet re-ordering and undo the congestion window was proposed and implemented. The problem of packet reordering was considered solved until the introduction of Multi-Path forwarding, where packets of a single flow are split among different paths [73] [74]. An RFC was even created to prevent splitting TCP flows over multiple paths [75].

It is worth noting that different authors arrive at opposite conclusions concerning the impact of multi-path forwarding on TCP. Karlsson *et al.* [73], for example, concluded that multi-path forwarding reduces the throughput of TCP flows and that mitigation techniques implemented at the transport layer in the Linux kernel are not effective in reducing the impact of packet reordering. On the other hand, the authors [74] affirm that multi-path forwarding in fat tree data-center networks has little impact on the throughput. In any case, modern multi-path forwarding techniques try to act on a flowlet level and avoid reordering [76].

We performed an extensive evaluation on a testbed with the goal to answer the question whether route changes due to traffic engineering will have an adverse impact on the TCP flows and severely reduce the network performance. The detailed evaluation will be presented in the chapter 6. However, jumping forward, we can affirm that it is safe to reroute the flows as long as this action is not performed "too frequently".

2.4.2 Scalability of centralized management

SDN looks very promising in theory. Nevertheless, the first commercial implementation, OpenFlow had lots of drawbacks. One of them, which we rapidly mentioned in the previous section, is due to relying too much on the centralized controller.

Limited performance of the SDN controller was one of the issues that we faced in a testbed environment during preliminary tests. Although Open vSwitch software may achieve satisfactory switching rates, bursty traffic may drain the controller's resources and make it a network bottleneck even in small systems. Considering the time required to parse incoming *packet_ in* OpenFlow message, calculate efficient path and push flow information to all intermediate flow tables, particular care should be taken to prevent improper handling of network streams.

Studies have shown that the latency induced by the control plan can have a significant impact on the SDN performance. For example, the delay needed to update a rule in the switches can be as high as 30ms [77]. The well-known Google's WAN network, B4, had an outage due to the incapacity of the SDN switches to process the high number of incoming flow modification messages at a sufficient rate [78].

2.4.3 Control traffic overhead

An essential requirement of traffic-engineering comprises the ability to gather information about the state of the network and to configure the behavior of network elements to route traffic flows accordingly. This information has to be exchanged between network devices or, in the case of SDN, transferred to a centralized network controller. This control traffic may induce a major overhead. In this section, we discuss the trade-offs between the amount of control traffic and the usefulness of the information.

Gathering information about the state of the network

We are aware of the following approaches when it comes to gathering information about the network state:

- Traffic agnostic: the network traffic is not taken into consideration at all.
- Based on link utilization: the solution relies on knowing the utilization of each link.
- Relying on the traffic matrix (origin-destination flows): only the information about the accumulated bandwidth flowing from an ingress device towards each egress device is transmitted.
- Based on (micro-)flows: the throughput of each TCP/UDP flow is communicated to the SDN controller.

Each next level increases the management overhead but also enables more fine-grained optimization. Leaving aside the traffic agnostic solutions, the research community still does not agree on which approach is better [79]. Some authors prefer to gather the link utilization and run complex algorithms to reconstruct traffic matrices instead of directly collecting the traffic matrix [53][80], even though traffic matrix estimation was proven inefficient [81]. On the other hand, some production backbone networks are comfortably relying on OpenFlow-based micro-flow statistics [47][2].

We believe that micro-flow based statistic will not scale to the number of TCP/UDP flows traversing a backbone network. In the remaining part of this section, we estimate the control traffic overhead of solutions relying on traffic matrices to decide if it is a viable approach for our solutions.

In theory, the quantity of control traffic needed to update the origin-destination traffic matrix fully increases quadratically with the number of network devices: each ingress device must inform the SDN controller about how much data is originated towards every other (egress) devices. In practice, this behavior is confirmed in the current generation of SDN controllers, which adopt a polling strategy for statistics collection.

To provide a realistic estimation of the bandwidth needed, we tested the solution on a testbed network with the ONOS network controller. The tests showed that, on average, 122bytes were used by the OpenFlow protocol per flow to inform the controller of an update in the flow's demand. Knowing that there are exactly $n \cdot (n-1)$ origin-destination flows in a network, we obtain the estimations from Table 2.3. The results assume that the traffic matrix is updated every second.

Network nodes	50	100	200	1000
Control traffic	0.3Mbps	1.21Mbps	4.86Mbps	120Mbps

Table 2.3: Estimated statistics traffic with 1s polling interval

Given these results, we can affirm that, in a reasonably sized backbone network with 100 nodes, it is possible to maintain a fresh view of the network traffic matrix effortlessly even with any currently available SDN protocols. Moreover, in real-world networks, traffic entering the network does not usually change unexpectedly. A vast improvement is possible by assuming a temporal stability of the flows. Asynchronous notifications can

be used to inform the controller only about changes in flow sizes at the cost of having a slightly less accurate view of the network traffic [82].

Configuring network routes

SDN-based solutions proposed in literature frequently use explicit network paths for each origin-destination flow. A large number of forwarding rules which have to be transmitted from the network controller to the SDN switches may generate a lot of control traffic. Moreover, it can amplify the problem of the centralized management presented in the previous section.

We consider that a good solution must be somewhere in between the shortest path routing and explicit routing. Instead of sending explicit paths, the controller should send a small amount of meta information, allowing switches to rebuild the network paths locally as computed by the controller and to update their forwarding tables locally in batches.

To achieve this goal, our work uses a novel SDN-friendly source routing protocol: SPRING, which was initially introduced by Cisco in 2015 under the name of "Segment Routing". We present this protocol in the next section.

2.5 The SPRING protocol

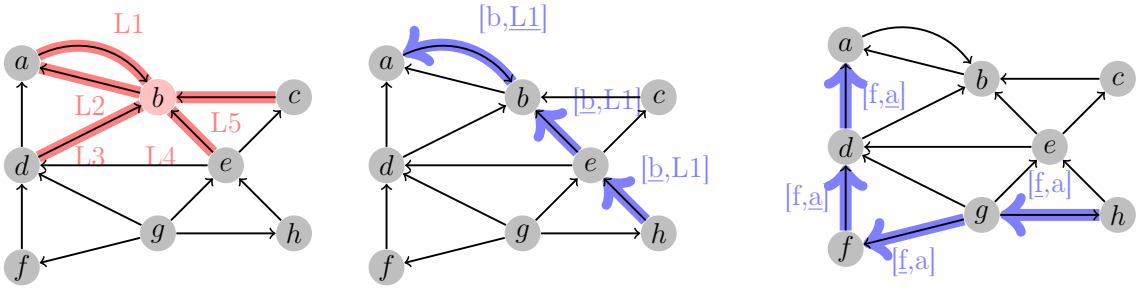
Source Packet Routing In NetworkinG (SPRING) [83] is an IETF draft protocol at the final stage of standardization is already implemented in the Linux kernel [84]. It was initially introduced by Cisco under the name of Segment Routing and is known under the latter name. We prefer however to adhere to the official IETF name in this work.

As presented by the creator itself [85], the SPRING protocol was inspired by the work done by Google[2] and Microsoft[47] and was designed as a SDN-ready protocol for Wide Area Network (WAN) networks. Instead of taking the path of OpenFlow, which heavily relies on the centralized network controller for a normal operation, SPRING tries to combine the best of two worlds: traditionally distributed control plane and centralized SDN.

The goal was to create a robust SDN-ready protocol, which does not rely on a centralized control plane for the most critical network function: packet forwarding. It is however designed to build on SDN for Traffic Engineering, and other Operations, Administration and Management (OAM) features. As a result, in the case of a failure in the SDN control plane, SPRING will still be able to operate the network with some QoS degradation.

SPRING keeps the traditional well-tested IGP (OSPF/IS-IS) control plane while replacing other complex and error-prone protocols. For example, traffic engineering, computation of fast reroute backup paths, network monitoring, and other network functions are exported to the centralized SDN controller. To facilitate the centralized management, SPRING relies on the source routing paradigm.

The data plane used by SPRING utilizes the same concept of label switching of MPLS, but its control plane has been completely redesigned. The distribution of labels is done via an extension to the IGP instead of using specific protocols such as LDP/RSVP-TE. Moreover, unlike MPLS, labels, called Segment IDentifiers (SID) in SPRING, have a



(a) Segment Identifiers (b) Force data over a link (c) Force data over a node

Figure 2.10: SPRING protocol.

global scope. In the present work, we are interested in two types of SIDs, namely "nodal" and "adjacency" as shown in Figure 2.10a.

- A nodal SID is globally unique and identifies a node (a,b,c,...,h).
- An adjacency SID is local to a node and identifies an outgoing interface (node b has the adjacency SIDs $L1$,$L2$,$L3$,$L4$ and $L5$).

After network discovery, sending a packet to node a through the shortest path requires encapsulating it into a packet with destination a. Unlike in IP, much more flexible traffic engineering is possible:

- If node h wants to send a packet to node a while forcing it over link $L1$, it adds the header [b,$L1$] (Figure 2.10b).
- If h wants to send a packet to a via f (Figure 2.10c), it uses the header [f,a].

Being a source routing protocol, SPRING enables fast flow setup and easy reconfiguration of virtual circuits with minimum overhead since changes must be applied only to the ingress devices. No time and signalling are lost re-configuring the midpoint devices.The policy state is in the packet header and is completely virtualized away from midpoints along the path. This means that a new flow may be created in the network by contacting only one network device: the ingress router.

This agility and atomicity of flow paths updates is essential in solutions that intend to perform link switch-off and improve energy efficiency. This comes in contrast with OpenFlow where the forwarding tables of all the devices along the path must be reconfigured. As already mentioned earlier, the need to synchronize the flow rule updates created a lot of problems for the research community and even was one of the causes that provoked an outage in the Google's SDN backbone network.

Following all this analysis, we decide to rely on the SPRING protocol for our traffic engineering, as it respects better the requirements for designing an SDN-based solution that is presented in the current section.

2.6 Conclusion

In this chapter, we have introduced the context of backbone networks. We did a quick overview of the architectures which are currently deployed in operator infrastructures. In particular, we compared the IPoOTN and IPoWDM architectures and motivated our choice to concentrate our research on the latter: we believe that it is better suited to route modern traffic, which is predominantly packet-based.

The number of research papers that try to reduce the energy consumption of IPoWDM networks has recently exploded in number. We sought to present the panorama of works in this field, both at the network design phase and the network operation phase. Evidence shows that the best way to reduce the energy consumption of such network is to manage the traffic intelligently in the packet layer; a technique known in the literature as "energy efficient traffic engineering"

We continued by presenting the existing traffic engineering techniques and the state of art works which apply these techniques to the problem of reducing the energy consumption of computer networks. This allowed us to conclude that there is a need in a online, reactive, framework with low computational overhead.

We finished the chapter with a discussion on the challenges which may define the design of such a framework. We concluded that a good SDN-based traffic engineering solution must avoid sending explicit paths to the SDN switches. Combining a centralized optimization with a local update of network paths can drastically reduce the amount of control traffic passing through the network. As a way to achieve this goal, we proposed to use the Source Packet Routing In NetworkinG (SPRING) protocol, which allows to change the paths of flows by atomically changing a single forwarding entry on the ingress network device.

Chapter 3

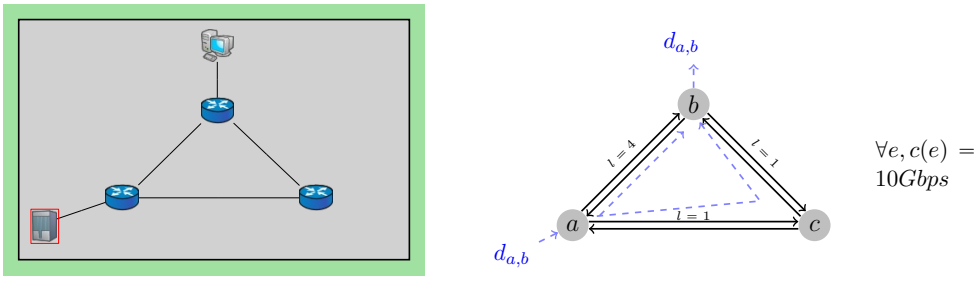
Theoretical background

In the previous chapter, we introduced the context of IPoWDM backbone networks and presented the multiple ways used to reduce their energy consumption. In particular, we are interested in solving this problem via intelligent traffic engineering in the IP layer.

In this chapter, we provide the necessary theoretical background on which we base our work. We first introduce the classical way to model an IP layer of a communication network via graph theory. After that, we provide a brief introduction to the linear programming. Armed with these basics, we dig a deeper into the theory of the maximum concurrent flow problem and introduce theoretical results that enable efficient algorithms for centralized traffic management.

3.1 Graphs 101: model of a communication network

This section introduces the formal notations used throughout this work. Consider the communication network from Fig. 3.1a. In this thesis, we model this network as a connected directed graph $\mathcal{G}(V,E)$ (Fig. 3.1b).



(a) Network (b) Graph model

Figure 3.1: Graph modelization of the network

The following notation is used:

- Each network device corresponds to a vertex $v \in V = \{a, b, c\}$.
- Each link represents a (directed) edge $e \in E$: $(a,b), (b,a), (a,c), (c,a), (b,c), (c,b)$. We frequently use the short notation ab to refer to the edge (a,b) going from node a towards node b.

- A function $c : E \to \mathbb{R}^+$ represents the capacities of the edges, *i.e.* the link speed.
- A function $l : E \to \mathbb{R}^+$ represents the length (cost) of the edge.
- The traffic entering the network at the source node $i \in V$ and flowing toward the egress destination node $j \in V$ is defined as the demand $d_{i,j} \geqslant 0$. We assume that there is no traffic from a node to itself, *i.e.* $\forall v \in V, d_{v,v} = 0$.
- A path $P = (e_1, e_2, ...)$ is a sequence of edges such that $(e_i = ab$ and $e_{i+1} = xy) \Rightarrow b = x)$. The length of a path $P = (e_1, e_2, ...)$ is the sum of the lengths of the edges in the path: $l(P) = \sum_{e \in P} l(e)$. (in Fig. 3.1b, the two dashed lines show 2 paths between a and b).
- A shortest path $SP_{i,j}$ from the node i to node j is a path such that there is no other path P between i and j with $l(P) < l(SP_{i,j})$. In Fig. 3.1, the path $P_1 = (ac, cb)$ is the shortest path between a and b under the given length function l, because $l(P_1) = l(ac) + l(cb) = 2 < 4 = l(ab)$.
- We introduce the notation $\mathcal{P}_{i,j}$: the set of all the paths between nodes i and j in $\mathcal{G}$. Let $\mathcal{P} = \cup_{(i,j) \in V^2 : i \neq j} \mathcal{P}_{i,j}$ be the set of all the paths in the network. Note that, by definition, $\mathcal{P}$ also contains the paths with cycles. To simplify the examples in the forthcoming sections, we do not consider these paths, as they do not influence the result and are avoided during the execution of the algorithms based on shortest path computations.

3.2 Linear Programming 101

This section is intended to help the readers with no Linear Programming background to grasp basic notions and go through the rest of this thesis more handily. For the ones that already have such background, you can skip this section. For a much more in-depth overview of the Linear Programming and the proof of the properties presented in this chapter, the reader can relate to the book [86], which served as a basis for our summary. It is freely available online[1].

Linear programming is a method for the optimization of linear objective functions subject to linear equality and inequality constraints. This technique can be used to model and solve a lot of optimization problems: which try to find the best possible allocation of limited resources while respecting a set of constraints. The objective and the constraints are defined by the nature of the problem being solved.

An elementary linear program is shown in Fig. 3.2. In this example, each dotted line represents a constraint expressed as an equality. The gray regions next to these lines indicate the directions of the inequality constraint. The shaded square area represents geometrically all the values of the variables x1 and x2 that simultaneously satisfy all the inequality constraints. Among the feasible solutions, the goal is to find the values of the variables x_1 and x_2 that maximize the objective function represented by the bold black lines. The sub-optimal solution is $(5.15, 1.2)$ and the optimal solution is (5.6,1.2).

[1] `http://web.mit.edu/15.053/www/`

Maximize $\mathbf{2 \cdot x_1 + 1 \cdot x_2}$ under the constraints:

$$\begin{aligned} 3 \cdot x_1 - 4 \cdot x_2 &\leqslant 12 \\ -1 \cdot x_1 - 2 \cdot x_2 &\leqslant -7 \\ -2 \cdot x_1 + 7 \cdot x_2 &\leqslant 0 \\ 6 \cdot x_1 + 7 \cdot x_2 &\leqslant 42 \\ x_1 &\geqslant 0 \\ x_2 &\geqslant 0 \end{aligned}$$

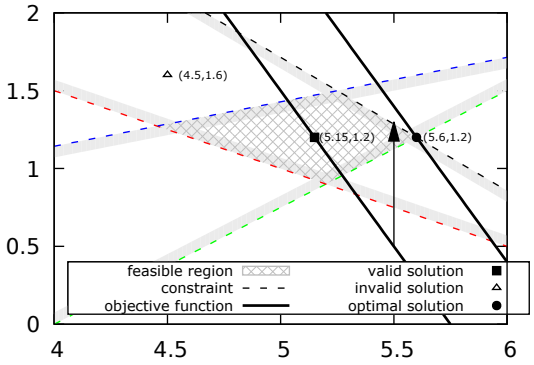


Figure 3.2: Example of linear program in $\mathbb{R}^2$

Any point that is not within the feasible region violates at least one of the constraints of the problem. For example, this is the case for the point represented by a triangle. It respects all the inequalities except the third one: $-2 \cdot 4.5 + 7 \cdot 1.6 = 2.2 \nleqslant 0$.

The previous linear program can also be represented in a matrix form $\{c^T x | Ax \leqslant b, x \geqslant 0\}$:

$$max \left\{ \underset{c^T}{\begin{pmatrix} 2 & 1 \end{pmatrix}} \underset{x}{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}} \middle| \underset{A}{\begin{pmatrix} 3 & -4 \\ -1 & -2 \\ -2 & 7 \\ 6 & 7 \end{pmatrix}} \underset{x}{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}} \leqslant \underset{b}{\begin{pmatrix} 12 \\ -7 \\ 0 \\ 42 \end{pmatrix}}, \underset{x}{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}} \geqslant \underset{0}{\begin{pmatrix} 0 \\ 0 \end{pmatrix}} \right\}$$

An important property of the Linear Programming is the *duality*. For any maximization linear program presented in the form $max\{c^T x | Ax \leqslant b, x \geqslant 0\}$, we can formulate a corresponding minimization problem $min\{b^T y | A^T y \geqslant c, y \geqslant 0\}$, called the *dual* problem. The initial problem is called *primal*.

In matrix form, the dual problem will be represented as follows:

$$min \left\{ \underset{b^T}{\begin{pmatrix} 12 & -7 & 0 & 42 \end{pmatrix}} \underset{y}{\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}} \middle| \underset{A^T}{\begin{pmatrix} 3 & -1 & -2 & 6 \\ -4 & -2 & 7 & 7 \end{pmatrix}} \underset{y}{\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}} \geqslant \underset{c}{\begin{pmatrix} 2 \\ 1 \end{pmatrix}}, \underset{y}{\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}} \geqslant \underset{0}{\begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}} \right\}$$

This notation makes it easier to observe that each variable of the dual corresponds to a constraint in the primal.

One of the fundamental duality properties is referred to as "**weak duality**" and provides a bound on the optimal value of the objective function of either the primal or the dual. The value of the objective function for any feasible solution to the primal maximization problem is bounded from above by the value of the objective function for any feasible solution to its dual minimization problem. Similarly, the objective function

of the dual problem is always greater than the value of the objective of the primal. Furthermore, in a case when the linear program is feasible and has an optimal solution, we have:

Strong Duality Property. *If the primal (dual) problem has a finite optimal solution, so does the dual (primal) problem, and these two values are equal. That is, if we denote by x^* the optimal primal solution and by y^* the optimal dual solution, we have:*

$$c^T x^* = b^T y^*$$

The importance of this property is that it indicates that we may, in fact, solve the dual problem in place of (or in conjunction with) the primal problem. A solution which we show in a future chapter does indeed use this property to construct the primal problem jointly with the dual.

A linear program in which some variables are restricted to be integers is called Mixed-Integer Linear Programming (MILP). The integrality constraints allow the model to capture the discrete nature of the decisions. For example, a variable whose values are restricted to 0 or 1, called a binary variable, can be used to decide whether or not some action is taken. Unfortunately, the introduction of the integer variables makes the problem NP-hard.

In the next section, we study a real problem modeled in a linear program. In particular, we provide the classical edge-path modeling of the maximum concurrent flow problem.

3.3 The maximum concurrent flow problem

Flow problems are at the core of many optimization problems in the field of computer networks.

A special case of a flow problem is the *maximum concurrent flow problem.* It takes a graph $\mathcal{G}(V,E)$ as an input, together with k different pairs of vertexes $(s_i,t_i) \in V^2, 0 \leqslant i < k$. Each pair i has an associated demand d_{s_i,t_i}. The goal is to maximize the total flow in the graph while respecting: i) the edge capacities given by a function $c : E \to \mathbb{R}^+$; and ii) the constraint that the ratio of the flow supplied between a pair of vertexes to their demand must be the same for all k pairs. The latest constraint ensures that the demands of all pairs are satisfied in equal proportion. No pair is "starving" because other pairs consumed all the bandwidth.

In the forthcoming sections we'll provide a step by step construction of the edge-path Linear Program model of the maximum concurrent flow problem. This will allow to illustrate and better understand the reasoning which should be used to formulate a simple real-world problem into a mathematical linear programming model. Afterwards, we'll quickly introduce the dual of the LP and the node-arc formulation of the primal. In parallel, we present the fundamental properties used in the chapter 5 of this thesis.

3.3.1 Constructing the edge-path LP formulation

Consider the network from Fig. 3.3. The operator observed that as much as $d_{a,b} = 9Gbps$ traffic is entering the network at node a and leaving at node b. Respectively, $d_{a,c} = 1Gbps$ traffic goes from a to c.

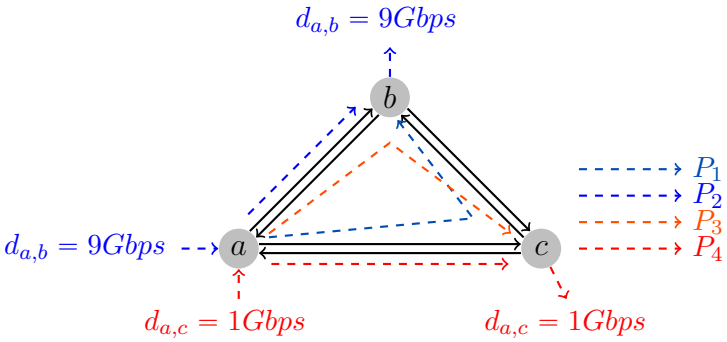


Figure 3.3: Paths which can be taken by the packets of the demands $d_{a,b}$ and $d_{a,c}$

The operator wants to find how much more traffic can be theoretically absorbed into the network before having to replace the network devices with faster, more expensive and more power hungry ones. It assumes that all demands will increase at comparable rate. For example: if $d_{a,b}$ doubles to $18Gbps$, $d_{a,c}$ will also double to $2Gbps$.

To find the solution, we use the following objective: maximize $\lambda \cdot 9Gbps + \lambda \cdot 1Gbps$ = maximize $\lambda \cdot 10Gbps$ = maximize λ [2] thus represents the traffic growth rate.

One can see that the only two possible ways to transport the data from node a to node b is by passing through the paths $P_1 = \{ac,cb\}$ and $P_2 = \{ab\}$. We ignore the paths containing cycles as, for example: $\{ac,ca,ab\}$.

Let $f_{a,b}(P_1)$ be the fraction of the demand $\lambda \cdot 9Gbps$ passing through the path P_1. By construction: $f_{a,b}(P_1) + f_{a,b}(P_2) = \lambda \cdot 9Gbps$. The same for the demand from a to c: $f_{a,c}(P_3) + f_{a,c}(P_4) = \lambda \cdot 1Gbps$.

The notation $f_{a,b}(P_1)$ is redundant as the path P_1 is known to start at node a and to end at node b. We therefore use a condensed notation $f(P_1)$. The previous equations can be rewritten as **Demand constraints**:

$$\begin{aligned} f(P_1) + f(P_2) \qquad\qquad & -9\lambda = 0 \\ +f(P_3) + f(P_4) \quad & -1\lambda = 0 \end{aligned} \tag{3.1}$$

Also, the flows passing through a link must be smaller than the capacity of this link. As a result, we have the following **Capacity constraints**:

$$\begin{aligned} f(P_1) \qquad\qquad + f(P_4) \quad & \leqslant c(ac) \\ + f(P_2) \quad +f(P_3) \quad & \leqslant c(ab) \\ f(P_1) \qquad\qquad & \leqslant c(cb) \\ +f(P_3) \quad & \leqslant c(bc) \end{aligned} \tag{3.2}$$

Finally, the amount of data passing through a path cannot be negative. **Flow non-negativity constraints**:

[2] "10Gbps" is a constant. As a result, maximizing $\lambda \cdot 10Gbps$ is equivalent to maximizing λ.

$$\begin{aligned} f(P_1) &\geqslant 0 \\ f(P_2) &\geqslant 0 \\ f(P_3) &\geqslant 0 \\ f(P_4) &\geqslant 0 \end{aligned} \tag{3.3}$$

These equations can be written in a condensed form:

$$\begin{aligned} &\text{Maximize: } \lambda \\ &\text{Subject to constraints:} \\ &\text{from 3.1}: \sum_{P \in \mathcal{P}_{i,j}} f(P) = \lambda \cdot d_{i,j} && \forall (i,j) \in V^2 : d_{i,j} > 0 \\ &\text{from 3.2}: \sum_{P \in \mathcal{P}: e \in P} f(P) \leqslant c(e) && \forall e \in E \\ &\text{from 3.3}: f(P) \geqslant 0 && \forall P \in \mathcal{P} \end{aligned} \tag{3.4}$$

The problem solved by the Linear Program 3.4 is known by the name of Maximum Concurrent Flow Problem (MCFP). In the following, we refer to any valid solution λ as "*concurrent flow of throughput* λ". We use the notation λ^* to designate the optimal solution to this problem.

3.3.2 The node-arc LP formulation

The edge-path linear programming formulation of the maximum concurrent flow problem is beneficial due to the relation between the primal and the dual, which we present in the next section. Moreover, it is better suited for use in conjunction with the method of column generation [87] for solving the maximum concurrent flow problem: the general idea is to start with solving a linear program using a limited subset of network paths. After that, the variables of the dual are used to find a new path that could improve the solutions, which is then added to the initial paths in the primal and the solver re-iterates to search another path.

On the other hand, when searching for an optimal solution, the edge-path formulation is not the best one, because the number of paths in the network (the number of primal variables) can potentially be exponential in the network size. The node-arc formulation is preferrable. The following variables are used:

V	set of nodes
E	set of edges
$c(e)$	capacity of the edge $e \in E$
$f_e^{i,j} \geqslant 0$	flow from node $i \in V$ to node $j \in V$ on link $e \in E$
w_v^+	set of outgoing links from the node $v \in V$
w_v^-	set of inbound links towards the node $v \in V$

The model objective is still to maximize: λ
Subject to the following constraints:
i) Flow conservation constraints. These constraints algebraically state that the sum of the flow through arcs directed toward a node plus that node's supply, if any, equals the

sum of the flow through arcs directed away from that node plus that node's demand if any.

$$\sum_{e \in w_v^-} f_e^{i,j} - \sum_{e \in w_v^+} f_e^{i,j} = \begin{cases} -\lambda \cdot d_{i,j}, & \text{if } v = i \\ \lambda \cdot d_{i,j}, & \text{if } v = j \\ 0, & \text{otherwise} \end{cases} \qquad \forall v \in V, \forall (i,j) \in V^2 : d_{i,j} > 0 \tag{3.5}$$

ii) Edge capacity constraints. The total flow passing through a link must be smaller than its capacity.

$$\sum_{(i,j) \in V^2 : d_{i,j} > 0} f_e^{i,j} \leqslant c(e) \qquad \forall e \in E \tag{3.6}$$

All the models used for validating our heuristics are based on this node-arc LP formulation.

3.3.3 Properties of the edge-path LP formulation

The edge-path formulation allows proving a set of useful properties, which will be used in this thesis. The first property is introduced by the authors of [88] under the name "distance(length) upper bound". It affirms that, for any lengths assigned to the links, the ratio $\frac{\sum_{e \in E} l(e)c(e)}{\sum_{(i,j) \in V^2 : d_{i,j} > 0} l(SP_{i,j}) d_{i,j}}$ provides an upper bound for the maximum concurrent flow λ. Formally, it is defined as follows:

Claim 3.3.1. *For any link length function $l : E \to \mathbb{R}^+$, and any feasible concurrent flow of throughput λ, we have $\frac{\sum_{e \in E} l(e)c(e)}{\sum_{(i,j) \in V^2 : d_{i,j} > 0} l(SP_{i,j}) d_{i,j}} \geqslant \lambda$.*

Note 1: We assume $\sum_{(i,j) \in V^2 : d_{i,j} > 0} l(SP_{i,j}) d_{i,j} \neq 0$. This condition is true in all the instances of the problem which are used in this thesis.
Note 2: Some authors use the term "volume of the network" for referring to the term $\sum_{e \in E} l(e)c(e)$

Proof. As per link capacity constraints (3.2) of the primal LP, if a concurrent flow of throughput λ is feasible, the amount of flow which passes through a link never exceeds the capacity of this link: $c(e) \geqslant \sum_{P \in \mathcal{P} : e \in P} f(P)$. As a result, we have:

$$\begin{aligned} \sum_{e \in E} l(e)c(e) &\geqslant \sum_{e \in E} (l(e) \sum_{P \in \mathcal{P} : e \in P} f(P)) = \\ &= \sum_{e \in E} \sum_{P \in \mathcal{P} : e \in P} l(e) f(P) = \\ &= \sum_{P \in \mathcal{P}} (\sum_{e \in P} l(e)) f(P) = \\ &= \sum_{P \in \mathcal{P}} l(P) f(P) \end{aligned} \tag{3.7}$$

Furthermore, the length of a shortest path between a source node i and a destination node j is, by definition, shorter or equal to the length of any other path between these nodes, i.e.: $\forall P \in \mathcal{P}_{i,j}, l(P) \geqslant l(SP_{i,j})$. As a result:

$$\begin{aligned}\sum_{P\in\mathcal{P}} l(P)f(P) \geqslant & \sum_{(i,j)\in V^2:d_{i,j}>0} (l(SP_{i,j}) \sum_{P\in\mathcal{P}_{i,j}} f(P)) = \\ = & \sum_{(i,j)\in V^2:d_{i,j}>0} l(SP_{i,j})\lambda d_{i,j} = \\ = & \lambda \sum_{(i,j)\in V^2:d_{i,j}>0} l(SP_{i,j})d_{i,j}\end{aligned} \tag{3.8}$$

From equations 3.7 3.8, and under the assumption that $\sum_{(i,j)\in V^2:d_{i,j}>0} l(SP_{i,j})d_{i,j} \neq 0$, we prove the claim. □

The previous results can be applied to the particular case of an optimal maximum concurrent flow λ^*:

$$\frac{\sum_{e\in E} l(e)c(e)}{\sum_{(i,j)\in V^2:d_{i,j}>0} l(SP_{i,j})d_{i,j}} \geqslant \lambda^* \tag{3.9}$$

3.3.4 The dual of the edge-path LP and its properties

The dual of 3.4 has a variable $z(i,j)$ for each demand constraint (3.1) and a variable $l(e)$ for each capacity constraint (3.2) and is defined as follows:

$$\begin{aligned}&\text{Minimize: } \sum_{e\in E} l(e)c(e) \\ &\text{Subject to constraints:} \\ &\sum_{e\in P} l(e) \geqslant z(i,j) \qquad \forall (i,j) \in V^2 : d_{i,j} > 0; \forall P \in \mathcal{P}_{i,j} \\ &\sum_{(i,j)\in V^2:d_{i,j}>0} z(i,j)d_{i,j} \geqslant 1 \\ &l(e) \geqslant 0 \qquad e \in E\end{aligned} \tag{3.10}$$

Lemma 3.3.2. *Setting $z(i,j)$ equal to the length of the shortest path between i and j under the length function l, i.e. $z(i,j) \leftarrow l(SP_{i,j})$, neither changes the cost nor impacts the feasibility of the LP.*

Proof. The first group of constraints forces each $z(i,j)$ to be *at most* equal to the length of the shortest path between i and j. In any valid solution to the LP 3.10, if some $z(i,j)$ is strictly smaller than $l(SP_{i,j})$, we can safely increase it without invalidating the constraint $\sum_{(i,j)\in V^2:d_{i,j}>0} z(i,j)d_{i,j} \geqslant 1$ or changing the objective value.

□

Lemma 3.3.2 allows us to rewrite the LP:

$$\begin{aligned}
&\text{Minimize: } \sum_{e\in E} l(e)c(e) \\
&\text{Subject to constraints:} \\
&\sum_{e\in P} l(e) \geqslant l(SP_{i,j}) \qquad \forall (i,j) \in V^2 : d_{i,j} > 0; \forall P \in \mathcal{P}_{i,j} \\
&\sum_{(i,j)\in V^2 : d_{i,j}>0} l(SP_{i,j})d_{i,j} \geqslant 1 \\
&l(e) \geqslant 0 \qquad e \in E
\end{aligned} \tag{3.11}$$

Lemma 3.3.3. *Let $D(l) := \sum_{e\in E} l(e)c(e)$ be the quantity minimized by the dual. For any valid solution of the LP 3.11,*

$$D(l) \geqslant \frac{\sum_{e\in E} l(e)c(e)}{\sum_{(i,j)\in V^2 : d_{i,j}>0} l(SP_{i,j})d_{i,j}} \geqslant D(l)^* \tag{3.12}$$

Proof. The first inequality is a direct application of the constraint $\sum_{(i,j)\in V^2 : d_{i,j}>0} l(SP_{i,j})d_{i,j} \geqslant 1$ of the LP 3.11. The second inequality is a direct application of the strong duality theorem on Equation 3.9 from the previous section.

□

3.4 Conclusion

This chapter rapidly introduces the theoretical background of this thesis. Our primary goal was to present the "maximum concurrent flow problem", a problem which received extensive attention from the academic research community. One of these works [5] presented an efficient algorithm, which is adapted to our needs in Chapter 5.

We also highlight the importance of Equation 3.12 which provides an upper bound to the dual optimization objective. It is at the core of various algorithms for approximately solving the maximum concurrent flow problem, including our work.

Chapter 4

Consume Less: the STREETE framework for reducing the network over-provisioning

In this chapter, we present the SegmenT Routing based Energy Efficient Traffic Engineering (STREETE) framework, an online SDN-based method for switching links off/on dynamically according to the network load. To react quickly to traffic fluctuations in operators' core networks, we trade off optimality for speed and create a solution that relies entirely on dynamic shortest paths to achieve very fast computational times.

One of the main contribution of this chapter is the ability to react very fast in order to awake sleeping links. The literature has thus far mostly overlooked the problem of turning links *on* assuming that the traffic varies slowly in backbone networks due to the high aggregation level. Nevertheless, unexpected burst still happen and must be rapidly assimilated by the network. For example, the real traffic matrices which we use for validation on the Germany50 network contain such a sudden burst of traffic volume.

This chapter starts by giving a high-level overview of our framework, and then details the implemented algorithms. It then presents the methodology and the techniques used to evaluate the proposed solution. In particular, we provide a quick overview of network topologies; the software used for evaluation; the analyzed metrics; and the results obtained by simulation. We highlight the observed strengths and limits of STREETE, thus making a transition towards the next chapter, which presents a mean to make STREETE work under extremely high network load at the expense of increased complexity.

A large part of this chapter also presents the Software-Defined Network (SDN) testbed built for experimental evaluations and discusses on the problems revealed by the experimental evaluation.

4.1 General overview

At an abstract level, the SegmenT Routing based Energy Efficient Traffic Engineering (STREETE) framework works as follows:

- All ingress devices inform the network controller of their demand towards each egress node. That way, the controller maintains an up-to date network-wide traffic

matrix.

- The controller, which has an up-to-date and global view of the network topology and the traffic matrix, executes an algorithm to determine the network links that must be turned *on* or *off*.
- The controller sends the previously computed information to the SDN switches.
- The switches locally recompute the new routes to avoid the links which are *off* and update their forwarding tables.

The algorithm that the controller runs continuously simulates the routing of the latest network traffic matrix while trying to detect network under-utilization and/or highly utilized links. Under-utilization offers a chance to reduce the energy consumption by turning links *off*. Hence, the algorithm performs an in-memory simulation of turning the candidate links *off* and estimating the impact of such a decision, without actually turning them *off* in the physical network. Whenever the result of the simulation shows that it is possible to turn a link *off* without creating congestion, it is scheduled for switch *off*. Afterwards, the algorithm re-iterates to, potentially, turn-*off* other links. On the other hand, under high link utilization an imminent congestion is assumed and the algorithm acts in the same way to turn-*on* network links and solve the congestion if possible.

While the solution is conceptually simple and corresponds to a greedy, "generate and test", solution, it includes a couple of important optimizations that improve the performance and the quality of the final solution. The rest of this section presents the optimizations included into STREETE.

Provide an order for the greedy algorithm. The algorithm keeps two different views of the network: i) the current network view, with some links being *off*; and ii) the full network topology, where all the links are assumed active. The second view allows guiding the construction of the solution by providing an order for the greedy algorithm.

Using a fully active view of the network provides useful metrics for the algorithm. This is especially important for turning links *on*. Compared to the phase of switch off, it is much more difficult to correctly select a good subset of links to turn *on* that can solve congestion. It may occur that the best candidate link is far from the place of congestion. It is also usual to have to turn-*on* more than one link for this purpose.

For the phase of turning *off*, providing an order to the execution is less important. Simple local heuristics generally lead to good results [79]. Nevertheless, our double view of the network is still beneficial by avoiding chaining of link extinction. This case will be detailed latter in the current section.

The intuition behind the method that we propose is depicted in Figure 4.1. Starting from the network view with links being turned off (Figure 4.1a), the algorithm uses the traffic matrix to find the highly loaded links (Figure 4.1d). In the meantime, the demands from the traffic matrix are simulated being routed into the full network topology (Figure 4.1b). This action allows to sort the links in the decreasing order of their estimated utilization in the fully active network. Figure 4.1e illustrates the result of this computation where the thickness of the links represents their relatively high utilization compared to other links. The previous action enables defining the order in which the links will be

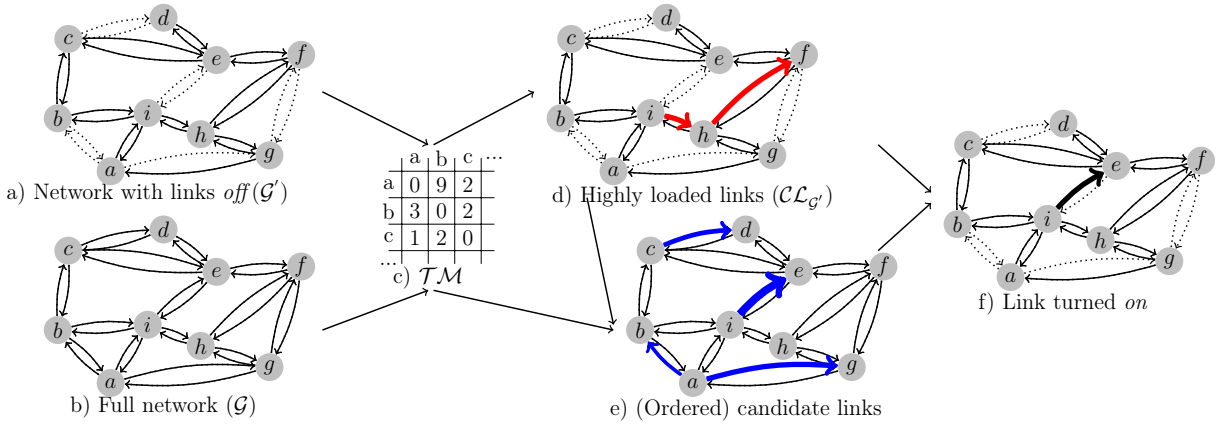


Figure 4.1: STREETE-ON: simulating the all-on network view to select the best link(s) to turn on.

turned on until, if possible, the congestion is resolved. For example, in Figure 4.1f, the link ab is selected as the best candidate for turn on because it is *off*, but it would transfer a lot of data in the all-*on* network.

Reduce the control traffic overhead. The algorithm executed by the SDN controller relies entirely on shortest path computations. This particularity allows to keep the communication overhead as low as possible because the SDN controller does not need to send explicit routes to the SDN switches. After receiving instructions from the controller, each network device locally computes the routes by using the same constrained shortest path algorithm as in the controller and naturally avoids disabled links. The only information which has to be sent from the SDN controller to the switches contains the state of all the links in the network: *on*/*off*. Moreover, there is no need to coordinate the updates between devices to avoid transient network loops. This problem is avoided by relying on the Source Packet Routing In NetworkinG (SPRING) protocol presented in a previous chapter.

Fast computations. To considerably reduce the computational overhead in the SDN controller, our solution relies on dynamic shortest path algorithms. The general idea is to partially update the shortest paths between the steps of the algorithm instead of computing a solution from scratch. We'll present one of the used dynamic shortest path algorithms in the section 4.2.4 of the present chapter.

Reduce the impact of historical decisions. As mentioned earlier, the two views of the network also reduce the influence of the historical decisions on the execution of the algorithm for the phase of extinction. This allows to avoid chains of turned-*off* links.

To provide an example on why a historical decision may have a negative impact on the final solution, we consider an algorithm from a related work [49], which works as follows: "*while there are no highly utilized links in the network, turn-off the least loaded link*". Figure 4.2 zooms in a subset of 4 nodes from a hypothetical network. Because the link AB was turned off in the first figure, the flow through the links BC and CD is reduced. That action made BC the best candidate for switch off. As a result, after

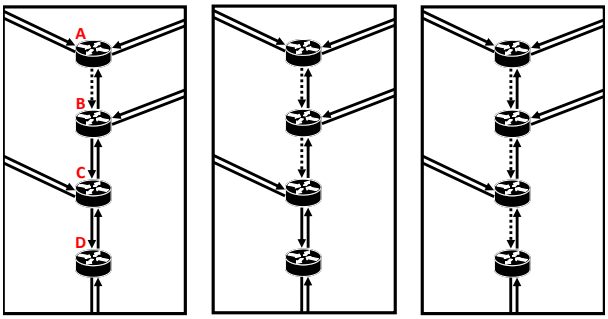


Figure 4.2: Chain of turned-*off* links

turning it *off* in the second figure, the flow on CD is reduced even further, making it the best candidate.

After doing some preliminary tests, we observed that this chain of events generates unnecessarily long routes in the network. In contrast, STREETE computes the order of extinction based on the all-on topology. This way, we ensure that the algorithm is not impacted by the actual state of links in the network.

Avoid route oscillations. The STREETE framework also incorporates a technique of double thresholds to avoid oscillations in the state of network links as a result of small traffic variations. In particular, it turns *on* a link only if this action reduces the utilization of any link which is used at more than a threshold α. At the same time, a link is turned *off* only if this action does not increase the utilization of any link beyond another threshold β. These thresholds can be modified at will in seek for a balance between the conservativeness and the energy efficiency of the algorithm. In this work we fix them to $\alpha = 0.8$ and $\beta = 0.6$. STREETE will though try to maintain the utilization of links between 60% and 80%, avoiding to load the links dangerously close to congestion.

4.2 Algorithms

In this section, we present the details of the algorithms executed by the SDN controller.

4.2.1 Notations

To facilitate the formal description, we use a few custom notations besides the classical graph notations introduced in the previous chapter. For instance, we use the notation $\mathcal{G}$ for the initial network topology with all the links being *on*, and $\mathcal{G}'$ for referring to the modified topology, with a subset of links switched *off*: $V' = V$ and $E' \subset E$. An example of $\mathcal{G}$ and $\mathcal{G}'$ can be seen in Figure 4.1*a* and 4.1*b* respectively.

We use the symbols α and β for referring to the two thresholds which trigger turning links *on* and, respectively, *off*.

Based on these thresholds, for a given network $\mathcal{G}$ and a traffic matrix $\mathcal{TM}$, we define the set of links at critical load $\mathcal{CL}_{\mathcal{G},\mathcal{TM}} = \{e \in \mathcal{G} \mid u_{\mathcal{G},\mathcal{TM}}(e) > \alpha\}$. It contains the links with utilisation higher than α in $\mathcal{G}$. For example, if $\alpha = 0.8$, $\mathcal{CL}_{\mathcal{G}'}$ will contain the links

with utilization greater than 80%. Hereafter we refer to links in $\mathcal{CL}$ as "links close to congestion". In our example, Figure 4.1d, $\mathcal{CL}_{\mathcal{G}',\mathcal{TM}} = \{ih, hf\}$.

Respectively, we use $\mathcal{ML}_{\mathcal{G},\mathcal{TM}} = \{e \in \mathcal{G} \mid u_{\mathcal{G},\mathcal{TM}}(e) > \beta\}$ for referring to links at load more than β. The turn-*off* phase will avoid creating such links. To be noted that $\mathcal{ML}_{\mathcal{G},\mathcal{TM}} \supseteq \mathcal{CL}_{\mathcal{G},\mathcal{TM}}$.

4.2.2 Initialization and main loop

The following steps must be computed only once, at the initialization of the framework.

- *Pre-computing a spanning tree.* To avoid disconnecting the network, the minimum spanning tree in terms of delay is pre-computed using Prim's MST algorithm. The links on the spanning tree are never turned *off*.

- *Pre-computing the shortest paths.* Dijkstra algorithm is used to compute the $|V|$ shortest path trees in the network.

The choice to never turn-*off* the links located on the minimum spanning tree may be criticized, because some links can remain enabled without being used to route network traffic. However, in the context of backbone networks, with aggregated network traffic the probability of this to happen is low. Fixing an always active spanning tree allows to keep the network connected for management purposes. Moreover, it may even be interesting to use graph spanners [89] instead of spanning trees to guarantee that no unnecessarily long paths are created in the network as a result of turning links *off*. A t-spanner of a graph $\mathcal{G}$ is a subgraph in which the distance between any two nodes is at most t times longer than the distance between the same nodes in $\mathcal{G}$. Such a solution for grid networks was proposed in a related work [90].

The main execution loop is self-explanatory (Alg. 1). It consists of an infinite loop which will periodically call STREETE-*ON* to find links which must be turned *on* and apply the change into the in-memory graph structure $\mathcal{G}'$, followed by STREETE-*OFF* to find the links which must be turned *off*. The state of links will afterwards be sent to the network devices. Note that the algorithm can, in fact, shut down some links while turning *on* others. This happens when the load is increased in one part of the network while being reduced in another.

Algorithm 1 Main loop

```
1: while Energy Efficient Traffic Engineering is active do
2:     STREETE-ON(G, G', TM)
3:     STREETE-OFF(G, G', TM)
4:     SendToDevices(StateOfLinks(G'))
5: end while
```

The following section presents the details of STREETE-*ON*. We do not enter into the details of STREETE-*OFF*, as its behavior is exactly the same as the turn-*on*, with the difference that links will be sorted in the ascending order of their utilization.

4.2.3 STREETE-*ON*

Algorithm 2 STREETE-*ON*

```
1: procedure STREETE-ON(G, G', TM)
2:     result ← ∅
3:     candidateLinks ← E\E'
4:     congested ← CL_{G',TM}
5:     changed ← true
6:     while congested ≠ ∅ and changed = true do
7:         changed ← false
8:         for all e ∈ sorted(candidateLinks, u_{G,TM}) do
9:                                   ▷ sorted in descending order of utilisation in G
10:            if congested ≠ ∅ then
11:                E' ← E' ∪ {e}                          ▷ Turn on the link e
12:                congested_after ← CL_{G',TM}
13:                if congestion decreased then
14:                    changed ← true
15:                    candidateLinks ← candidateLinks\{e}
16:                    congested ← congested_after
17:                    result ← result ∪ {e}
18:                else
19:                    E' ← E'\e                          ▷ Do not keep e on
20:                end if
21:            end if
22:        end for
23:    end while
24:    return result
25: end procedure
```

We mentioned in a previous section that our greedy heuristic first tries to turn-*on* the links which are *off* in the energy-optimised network $\mathcal{G}'$, but would have a high utilisation in $\mathcal{G}$. This action is performed by Algorithm 2 at lines 3 and 8. We prioritise turning *on* the links that would create shortcuts for large flows.

At lines 11-12 the algorithm simulates turning the link *on* and estimates the congestion after this operation. At lines 13-20 it tests if congestion decreases. If so, the link is scheduled to be turned *on* in the physical network; otherwise the link is disregarded.

"congestion decreased" at line 13 is defined as:

- solving all the congestion
- or decreasing the utilization of any congested link while avoiding both i) to create new congested links and ii) to increase the utilization of any congested link.

Sometimes a set of links must be turned *on* to avoid congestion. The while loop at line 6 ensures that the algorithm is repeated until the congestion is resolved or it becomes impossible to solve without creating another congestion.

The previously presented algorithm hides a complex operation when accessing $\mathcal{CL}$ at lines 4 and 12, namely estimating the load of links to find the congested ones. To

estimate the load of the link, we must simulate the routing in the network by first finding the shortest paths and, afterwards, simulating the demands flowing through the network. The next section presents how we use dynamic shortest path algorithms to reduce the complexity of the first step. However, for the second step, it is also possible to achieve an $O(|V|)$ complexity for routing all the |V|-1 flows from a common source. As this is not trivially intuitive, the algorithm 3 presents a way to achieve such a running time.

Algorithm 3 Compute how much data passes through the links of the spanning tree $\mathcal{T}$ rooted at node v_{root} if the demands $d_{v_{root},*}, * \in V$ are routed into the tree

```
1: function RouteInTree(T(V, E_T), v_root ∈ V, d_{v_root,*})
2:     sortedV ← SortTreeNodes(T, v_root)
3:     initialize f_v ← 0, ∀v ∈ V
4:     for i ← |V| − 1; i > 0; i ← i − 1 do
5:         v ← sortedV[i]
6:         find (u,v) ∈ E_T                ▷ Find the unique parent node u of v
7:         f_v ← f_v + d_{v_root,v}
8:         f_u ← f_u + f_v
9:         f_(u,v) = f_v
10:    end for
11:    return f_(*,*) = {f_(u,v) : uv ∈ E_T}
12: end function
```

The main idea relies on the fact that, in a directed tree, the traffic crossing a link (u,v) can be recursively defined as the sum of i) the traffic going to the destination endpoint v; plus 2) the traffic flowing through the links $\{(v,*)|* \in V\} \subset E_{\mathcal{T}}$ outgoing from v. The algorithm can be imagined as starting with the leaf nodes, which have no outgoing traffic. Afterwards, the algorithm climbs into the tree and computes the flow on each link using the previously mentioned recursive relation.

Algorithm 4 Topologically sort the nodes of the (directed) tree $\mathcal{T}$ starting from the root node v_{root}. Returns the ordered list of nodes.

```
1: function SortTreeNodes(T(V,E_T), v_root ∈ V)
2:     sorted = [v_root]                ▷ List of nodes with one element (v_root)
3:     i = 0                            ▷ Index in sorted of the next node to analyze
4:     while sorted.size() < |V| do
5:         u ← sorted[i]
6:         for all v ∈ V s.t. (u, v) ∈ E_T do
7:             sorted.append(v)
8:         end for
9:         i ← i + 1
10:    end while
11:    return sorted
12: end function
```

To achieve that, the algorithm proceeds in two phases. First, the tree is topologically sorted, i.e. the nodes are ordered such that there is no link going from a node towards any of its predecessors. This is easy to achieve in a tree and consists in traversing it starting

from the root node, as shown in Algorithm 4. The algorithm 3 afterwards traverses the sorted nodes in decreasing order (lines 4-10). This order ensures that, when a node is encountered, the flow passing through all its children was already computed.

4.2.4 Dynamic shortest paths

A naive approach for recomputing the shortest paths uses n static Dijkstra computations. As we want our algorithm to react fast to network congestion, we used an optimised Dynamic All-Pairs Shortest Path (DAPSP) algorithm proposed in the literature [91]. It is designed to execute only partial re-computation of the shortest paths in case of edge insertion/deletion. We use *D-RRL*, a slightly modified version of the algorithm initially proposed in [92]. The worst case theoretical complexity of this algorithm is the same as static Dijkstra computations. It is higher than the complexities of its competitors [91]. Nevertheless, aside the fact that it is the easiest to implement, this algorithm also happens to have the best performance on almost planar graphs which are encountered in computer networks.

Algorithm 5 Update the shortest path tree $\mathcal{T}$ to maintain the shortest paths from the root node v_{root} towards all the other nodes after the insertion of a new edge $e_{\mathcal{G}'}$ into the graph $\mathcal{G}'$

1: **procedure** D-RRL-ADDEDGE($\mathcal{G}'(V, E_{\mathcal{G}'}), \mathcal{T}(V,E_{\mathcal{T}}), v_{root} \in V, e \in E_{\mathcal{G}'}$)
2: $(u, v) \leftarrow e$ ▷ Get the extremities of the edge in $\mathcal{G}'$
3: **if** $v = v_{root}$ **then**
4: **return**
5: **end if**
6: **if** $dist(v) \leqslant dist(u) + l(e)$ **then**
7: **return**
8: **end if**
9: initialize $dist(w) \leftarrow l(v_{root}, w), \forall w \in V$
10: initialize $pq \leftarrow PriorityQueue()$
11: $dist(v) \leftarrow dist(u) + l(e)$
12: $E_{\mathcal{T}} \leftarrow (E_{\mathcal{T}} \backslash \{(\_,v)\}) \cup \{e\}$ ▷ Delete the unique edge going to v in $\mathcal{T}$ and add the new edge e
13: $pq.add(v, dist(v))$
14: **while** $pq.notEmpty()$ **do**
15: $v = pq.extractTop()$
16: **for all** $w \in V$ t.q. $(v, w) \in E_{\mathcal{G}'}$ **do**
17: **if** $dist(w) > dist(v) + l((v,w))$ **then**
18: $dist(w) \leftarrow dist(v) + l((v,w))$
19: $E_{\mathcal{T}} \leftarrow (E_{\mathcal{T}} \backslash \{(\_,w)\}) \cup \{(v,w)\}$
20: $pq.addOrUpdate(w, dist(w))$
21: **end if**
22: **end for**
23: **end while**
24: **end procedure**

The *D-RRL* algorithm consists of two parts:

- Initialization of the initial in-memory data structures by pre-computing the $|V|$ shortest path trees. This step is done only once, at the initialization of the network after the initial topology discovery.

- Partial update of shortest paths each time an edge is inserted or deleted. The algorithm 5, details this step for a partial update following an edge insertion to $\mathcal{G}'$.

The algorithm tests at line 6 whether the insertion of the edge does reduce the distance from the root node to the destination end of the edge. If the distance is not decreased, there is no need to do any changes to the spanning tree.

If the insertion of the edge does decrease the distance towards the node v, the algorithm executes a Dijkstra-like construct (lines14- 22) to update the impacted branch of the shortest path tree while leaving intact the rest of the tree. The actual reconstruction of the shortest path tree is done at lines 12 and 19: we delete from the shortest path tree the edge connecting the affected node and its predecessor on the path from the root. It is safe to assume that exactly one such edge exists, because every node except the root has exactly one inbound edge in a directed tree. The root node would never be examined due to the condition at the beginning of the algorithm.

The use of the dynamic shortest path algorithm has two advantages. First of all, it only recomputes the solution for the impacted subtree of each $|V^{'}|$ trees. Secondly, the biggest improvement is due to the fact that only a fraction of these trees will be, at all, impacted. Most of them will not pass the condition at line 3.

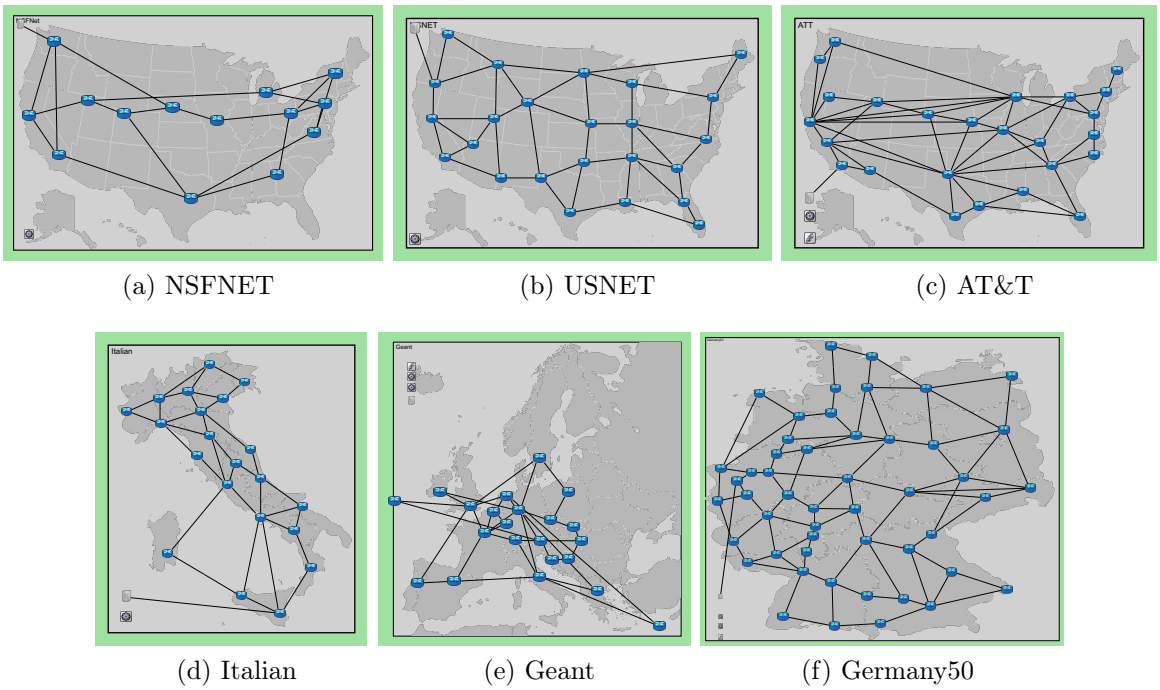
4.3 Evaluation by simulation

Although we made an extensive effort to evaluate the feasibility of the proposed solutions, access to real backbone network devices and real traffic information are difficult to obtain. Network operators and device manufacturers keep the details about their networks and their devices secret. Considering that experimenting with large scale production networks was not possible in our case, we resorted to simulations, combined with validation on a small-scale network testbed to assess the proposed techniques.

In our initial work [NC1][C3][J2], we used the discrete event simulator OMNeT++, together with the network simulation framework INET, to simulate real packets flowing through the network and through the software stacks of network devices. However, due to extremely long simulation times and low benefits of the packet-based simulation in the context of backbone networks, we changed our testing methodology. We present some lessons learned from the packet-based simulations in a forthcoming section (4.3.4).

The subsequent work was evaluated on an algorithmic level: we executed the algorithms on emulated input data without simulating packets transiting through the network. In parallel, we designed and implemented a proof of concept in the ONOS SDN controller and validated it on a small network testbed.

This section presents the evaluation methodology and the architecture of our testbed.



(a) NSFNET (b) USNET (c) AT&T

(d) Italian (e) Geant (f) Germany50

Figure 4.3: Topologies evaluated with real traffic traces

4.3.1 Evaluated networks

We employ the network topologies shown in Figure 4.3 to evaluate our solution. By using real traffic matrices for these networks, we evaluate our solutions in close to real scenarios. Although we do not simulate packets flowing through the network, we execute the algorithms and compare their estimation to a MILP baseline which is presented in the next section. The input of an algorithm is the traffic matrix at a discrete time t, and the output is the routing that must be applied to the network.

Moreover, a few details are worth mentioning:

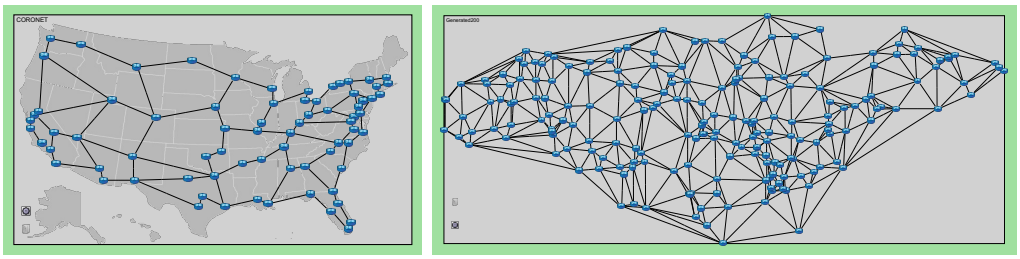
- The traffic matrices for the Geant and Germany50 networks are taken at 15 minute and, respectively, 5 minute intervals. We have no insight on the evolution of network flow in-between this sampling intervals. Moreover, the matrices date from 2007 and may not reflect the state of traffic in modern networks. Over the past years, the cloud has been extensively adopted by numerous companies and access to video streaming on demand has become commonplace in households around the world.

- The traffic matrices for the *NSFNet*, *USNet*, *AT&T* and *Italian* are issue from the GreenTouch project [93][94] and represent the average business Internet traffic expected between nodes of the networks in year 2020. For each network, we have 12 traffic matrices which reflect a day of operation with a 2 hour sampling interval.

- Most of these networks are small and may not correspond to the size of modern networks regarding the number of devices. Unfortunately, the operators keep the information about their network topology confidential. The only backbone network

who's topology is not secret is Geant[1]. The size of this network didn't change much compared to the one provided by SNDLib in 2007. For this reason, we follow the same path as the rest of the research community and assume that the sizes of networks presented in Fig. 4.3 are representative of the reality.

Nevertheless, we believe that the continuous growth of network traffic may lead to a growth in number of devices. To evaluate the computational time on bigger networks, we also executed the algorithms with uniform all-to-all traffic on the networks from Figure 4.4. The first one, Coronet, was taken from the Internet[2]. We also artificially generated a 200 node network using the igen[3] toolbox.



(a) CORONET (b) Generated200

Figure 4.4: Topologies evaluated with generated traffic

- We do not know the real capacities of the network links. We empirically adapted their capacity to obtain a maximum link utilization around 80% at the moment of the highest utilization.

The properties of these networks are summarized in Table 4.1.

Network	**Number of nodes**	**Number of links**
NSFNET	14	42
Italian	21	70
Geant	22*	72
USNET	24	86
AT&T	25	108
Germany50	50	176
CORONET	75	198
Generated200	200	1148

*the New York node is not visible in the Geant network, but the two links connecting it to the rest of the network are shown.

Table 4.1: Summary of the network topologies.

[1] `https://www.geant.org/Networks/Pages/Home.aspx`
[2] `http://www.monarchna.com/topology.html`
[3] `http://igen.sourceforge.net`

4.3.2 The MILP baseline

The MILP model used to generate a baseline for comparison, do not necessarily model the same problem as the described algorithms. We neither seek to design complex MILP models nor search for advanced techniques for solving them. Such techniques, for example, include the column generation method. For us, the Linear Programming provides a way to establish an upper bound for the solution quality against which our solutions can be compared.

The MILP model used in this chapter uses the following variables:

V	set of nodes
E	set of links
$c(e)$	capacity of the link e
$d_{i,j}$	demand from source node $i \in V$ towards the destination node $j \in V$
$f_e^{i,j}$	flow from node $i \in V$ to node $j \in V$ passing through link $e \in E$
x_e	boolean: 1 if the link $e \in E$ is ON; 0 if it is OFF
w_v^+	set of outgoing links from the node $v \in V$
w_v^-	set of inbound links towards the node $v \in V$

The model objective is to minimize the number of links being kept active.

$$\text{minimize:} \sum_{e \in E} x_e$$

Subject to:

Flow conservation constraint:

$$\sum_{e \in w_v^-} f_e^{s,d} - \sum_{e \in w_v^+} f_e^{s,d} = \begin{cases} -d_{s,d}, & \text{if } v = s \\ d_{s,d}, & \text{if } v = d \\ 0, & \text{otherwise} \end{cases} \qquad \forall v \in V, \forall (s,d) \in V^2 : d_{s,d} > 0 \tag{4.1}$$

Link capacity constraint:

$$\sum_{(s,d) \in V^2 : d_{s,d} > 0} f_e^{s,d} \leqslant c(e) \cdot x_e \qquad \forall e \in E \tag{4.2}$$

Likewise STREETE, we force an always-active spanning tree in the network:

$$x_e = 1 \qquad \forall e \in E : e \text{ is on spanning tree} \tag{4.3}$$

The MILP model has a small advantage over STREETE: flows are allowed to follow multiple paths in the network. This was done to reduce the computational complexity. Without this simplification, the MILP solver was sometimes unable to find the exact result in 2 hours even for the 24 node USNET network.

To produce the optimal baselines for comparison, we use the CPLEX 12.7.1 C++ API to build the linear programs directly from our C++ code and to solve them. The biggest network for which CPLEX executed without running out of RAM was Germany50.

4.3.3 Evaluation metrics

Energy consumption. We consider uniform link speeds and, as explained in a previous chapter, the energy consumption of network devices does not depend on their utilization. As a result, we assume that the energy a network consumes is proportional to the number of active links. Although we recognize that it is important to handle heterogeneous link speeds and energy consumption, this scenario is left for future work.

Distribution of network load. The utilization of the links is another parameter that illustrates the quality of the solutions based on switching links *on* and *off*. This includes not only the utilization of the most loaded link, but also the distribution of the load across all links. The last case is especially interesting for the load balancing purpose in the following chapter of this thesis. We believe that having a single highly utilized link is worse than having two links at medium utilization. In the latter case, the network is less likely to become congested if the traffic increases.

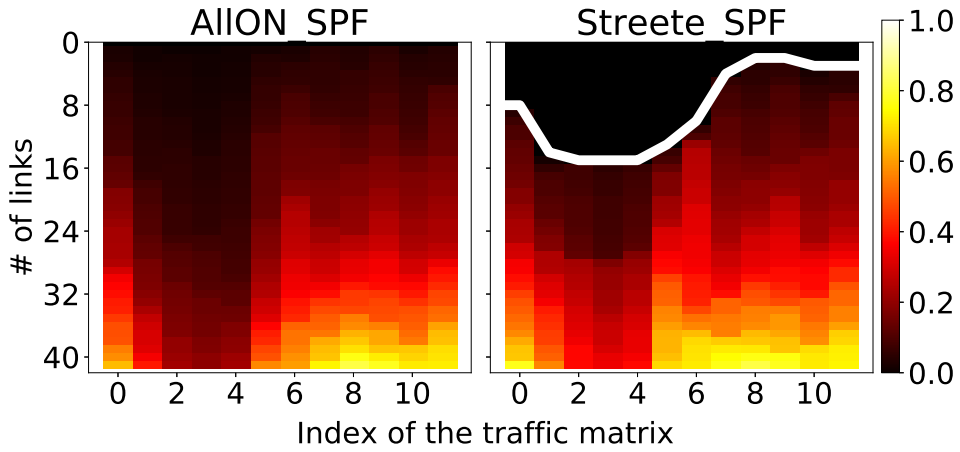


Figure 4.5: The heat-map of the link utilization during a day in the NSFNet network. Also illustrating the links which are *off* (shown in black)

Hereafter we present figures (e.g. Figure 4.5) that condense a lot of useful information: they illustrate the proportion of links that are active, their utilization, and the distribution of the utilization across the whole network. In particular, Figure 4.5 illustrates a day of the NSFNet network with two different routing techniques: i) classical shortest path routing; and ii) a technique that tries to turn links *off*.

For the NSFNet network, we have 12 traffic matrices at 2 hour intervals. For example, traffic matrix 0 represents the network at 00:00 o clock, while the traffic matrix 1 at 02:00AM. Each column on the x axes represents the distribution of the load on the links with a fixed traffic matrix. The black area corresponds to the lightly utilized links, while white points show an utilization close to 1 (100%). The black links in the rightmost figure are *off*. The bold white line shows the border between the active and inactive links. The y axes allows to rapidly evaluate the number of inactive links.

The figure makes it easy to see the distribution of the load in the network. At 8AM ($x = 4$), not only we were able to reduce the energy consumption of the network, but we also avoided to create new white (highly utilized) links. With the traffic matrix 4,

the utilization of most active links in the rightmost figure is between 20% and 40%. To generate this figure, for each traffic matrix, the links were sorted by their utilization. As a result, the most utilized link is always shown at the bottom of each column and does not necessarily correspond to the same link in the adjacent columns.

Computational time. Another important parameter is the time needed by the algorithms to generate a solution. The solution proposed in this chapter has a high theoretical complexity but while behaves well in practice. For this reason, we measure the actual time the algorithm takes on the previously presented network topologies.

All our algorithms were implemented in C++ within the OMNeT++[95] discrete event simulator. Even if we no longer make use of the packet-based simulation, we rely on the powerful tools included with the simulator to facilitate the description of network topologies; generation of simulation traces; and their analysis. All simulations were performed on servers with two Intel Xeon E5-2620 v2 processors and 64Gb of RAM. The RAM was never a limiting factor for the execution of our algorithms.

Path stretch. The fact that we put links to sleep to save energy certainly impacts the network performance. First of all, both turning links *off* and load balancing the traffic come at the cost of probably taking longer routes and thus increasing the end-to-end delay of the network traffic. Longer network paths are particularly problematic in the context of backbone networks, where a detour path can, potentially, go around a whole continent. To evaluate this fact, we introduce the parameter *path-stretch*, which defines how much the path length is increased compared to the shortest path routing. Taking into consideration that the sub-flows of an origin-destination flow can take multiple routes in some of our solutions, we consider the mean path length. Formally, *path-stretch*$_{i,j} \equiv \frac{\text{avg}\{l(P):P \in \mathcal{P}_{i,j} \text{ and } f(P)>0\}}{l(SP_{i,j})}$ with $l(e) = 1$. A *path-stretch* value of 1 means that the longest path and the shortest path have equal length. Respectively, a *path-stretch*=2 means that the length of the paths is twice as long as the shortest path.

4.3.4 Evaluation results

Lessons learned from packet based simulations

As mentioned previously, our initial work was evaluated using packet-level simulations. However, the simulation times were prohibitive. In particular, the results which we published in our first papers [NC1][C3][J2] took up to a weak of computation per simulation scenario. Obviously, this approach had some benefits, including: the possibility to easily measure the packet losses in router queues resulting from unexpected burst of network traffic; the possibility to measure the influence of the management delay on the overall reactivity of the solution.

For example, our results from Fig. 4.6, illustrate the impact of the management delay when energy efficient traffic engineering is used to reduce networks' energy consumption. In particular, we injected the network load shown by the red filled curve into the Germany 50 network and executed STREETE.

With the goal to react proactively, before any congestion really occurs in the network, the algorithm turns link on as soon as a link is utilized at more than 80% of its capacity.

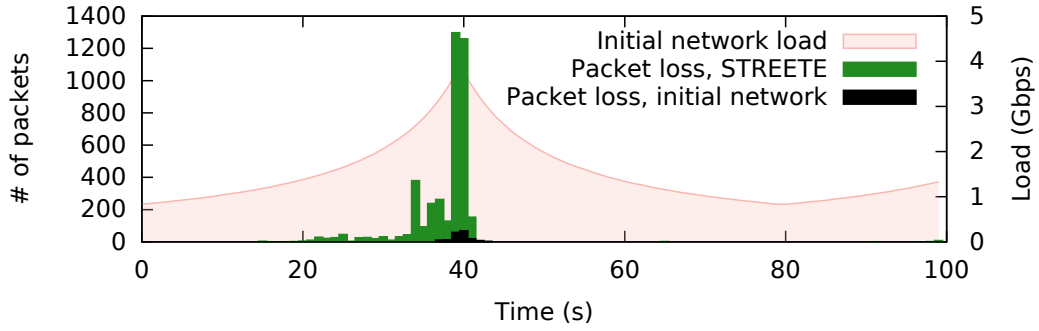


Figure 4.6: Packet loss.

However, even wit a safe margin, the network drops packets. There is a noticeable increase in the number of lost packets, especially at $t = 40$, when the initial network was already saturated. The packet loss on the most congested link at this moment is as high as 7.6%.

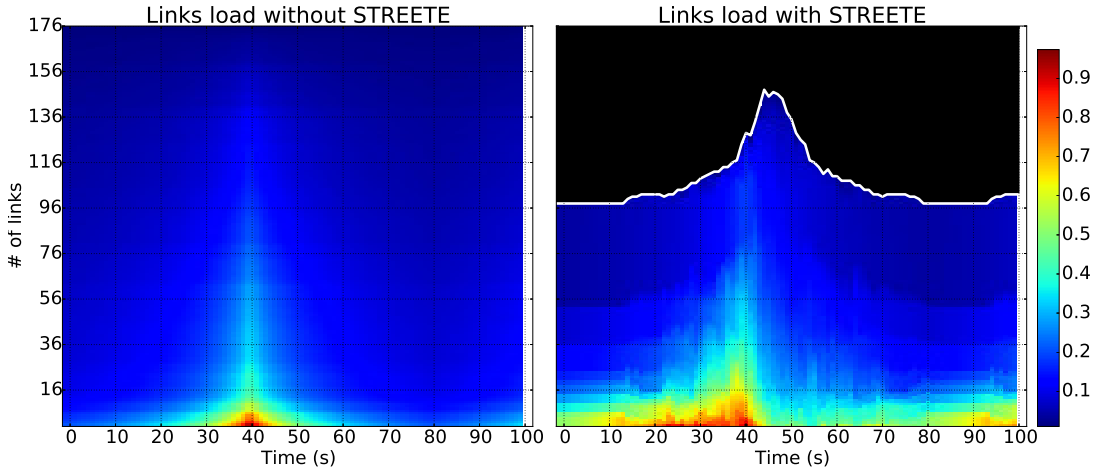


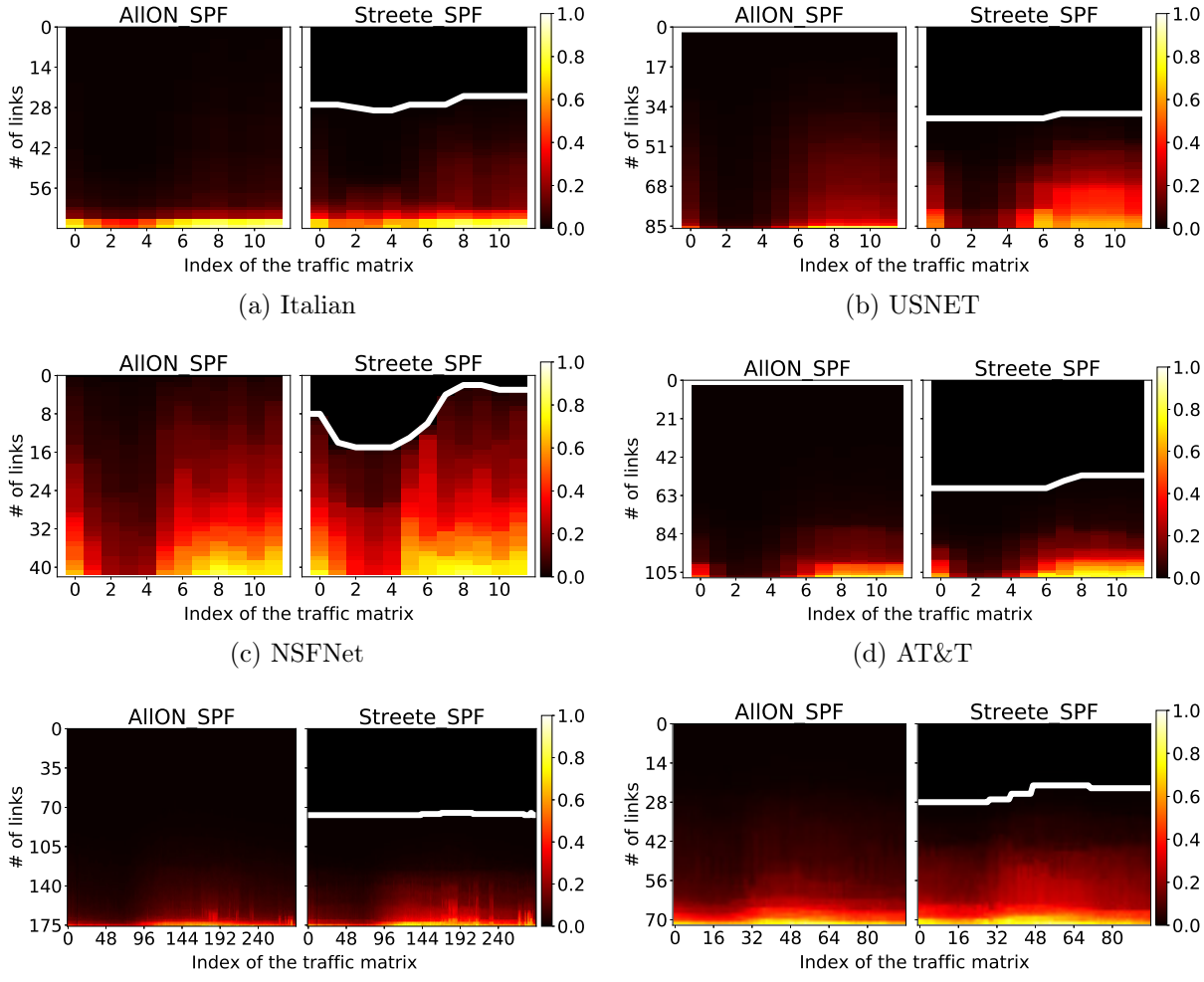
Figure 4.7: Links utilisation without and with STREETE.

This behavior is due to an approximately two-second delay between the increase of network load and the moment when the new forwarding rules are applied into the network devices by the SDN controller. Fig. 4.7 gives more insights on the matter. It shows the utilization map of each link in the Germany 50 network, where the blue area has an utilization close to 0 and red points present utilization close to 1. The white line in the right-hand side figure represents the limit between the links which are *on* an those that are *off*. It can be clearly seen that the network reacts with a delay to the congestion occurring at $t = 40$. Links are turned *on* only at approximately $t = 42$. This delay is almost entirely due to: i) the time needed for the traffic increase to be detected by network devices; ii) the time needed to notify the SDN controller about this; and iii) time needed to update the forwarding rules on network switches.

While packet based simulation allow to closer emulate the behavior of production network and detect problematic cases like the one presented in this section, we transited to the less expensive algorithmic evaluation which emulates only the work done by the SDN controller and not the entire network.

State of network links

Figure 4.8 shows the state of network links in each of the 6 network topologies evaluated with realistic traffic traces. For each network, the leftmost figure shows the utilization of the links when classical shortest path routing is used. The rightmost figure corresponds to the state of the network when our STREETE framework is used. For each figure, the black area represents the links with utilization close to 0% and the white points correspond to utilization close to 100%. The bold white line shows the border between the active and inactive links. The black points above the white line are switched *off*. More information about how this figure was generated was presented in the previous section (4.3.3).



(a) Italian

(b) USNET

(c) NSFNet

(d) AT&T

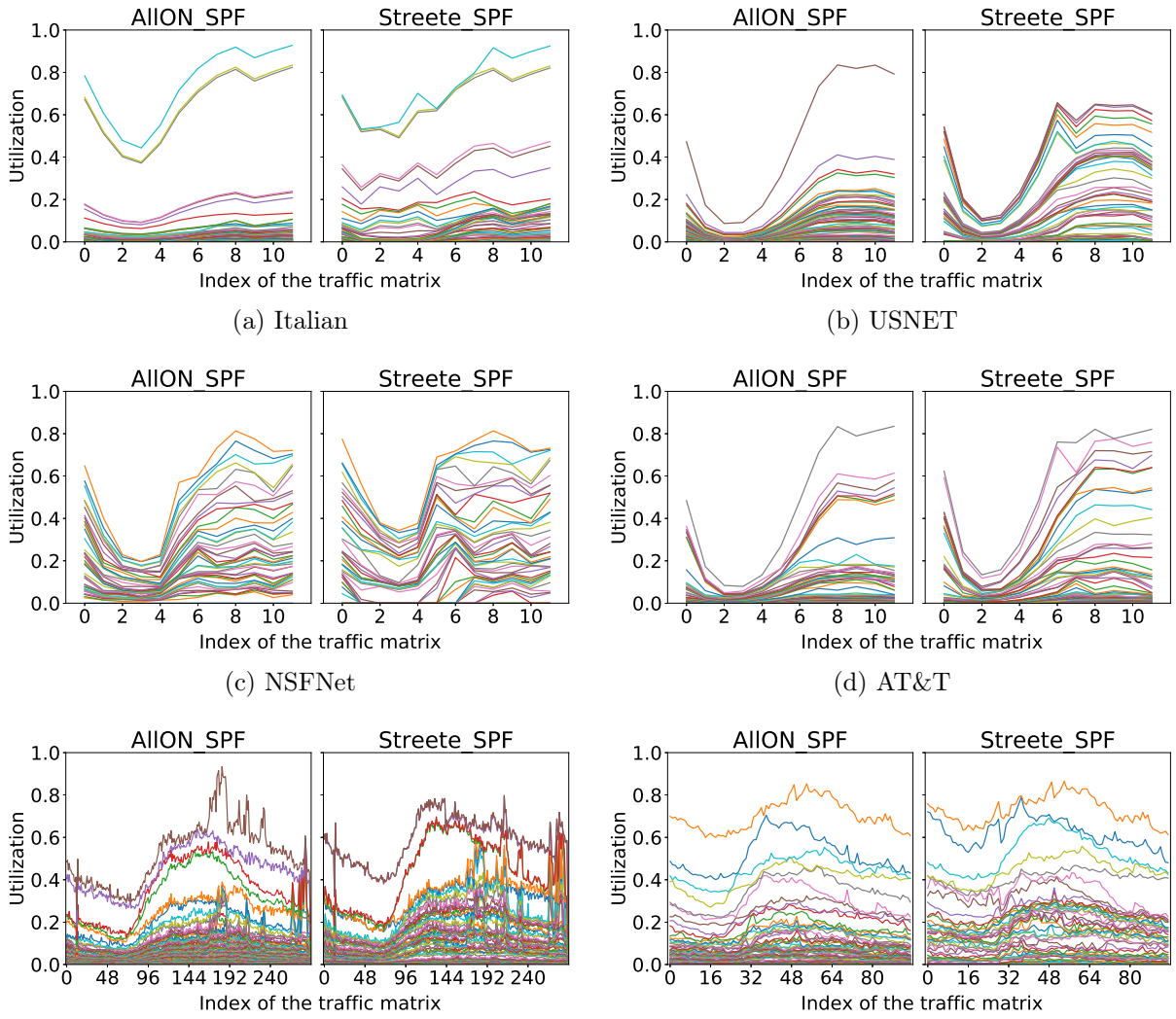
(e) Germany50

(f) Geant

Figure 4.8: The heat-map of the link utilization. Also illustrating the links which are *off* (shown in black)

Each figure corresponds to a day of operation of the network evaluated at fixed time points. For example, the first figures (4.8a - 4.8d) are generated using traffic matrices captured every two hours, producing 12 timestamps on the x axis. The latest two networks (4.8e - 4.8f) were evaluate at 5 minute and, respectively, 15 minute intervals.

Fig. 4.9 shows a more link-centric way to illustrate a part of the results condensed in the heatmaps. It allows to visualize the utilization of each link separately and, in particular, to see how the ignition of some sleeping links reduces the utilization of the most loaded ones. This is particularly well visible for the USNET network: at traffic matrix #7, the ignition of links reduced the utilization of other links to keep it below 80%.



(a) Italian (b) USNET

(c) NSFNet (d) AT&T

(e) Germany50 (f) Geant

Figure 4.9: The utilization of each link in the evaluated networks

The figures illustrate very well that shortest path routing does not suit these networks. Very few elephant flows consume most of the bandwidth. These flows rapidly congest some of the links while most links have very low utilization. Note that there are less highly utilized links when the STREETE framework is active. This result is particularly well visible on the USNET network (Fig. 4.8b and 4.9b) and may be counter-intuitive.

An illustration of the reason is given in Figure 4.10 where two flows are routed in the network and, when the link hd is down, the flows take a detour over longer paths. As a result, none of the links is at critical load. However, if the link is *on*, the flows are routed over hd and its utilisation rises above the threshold α.

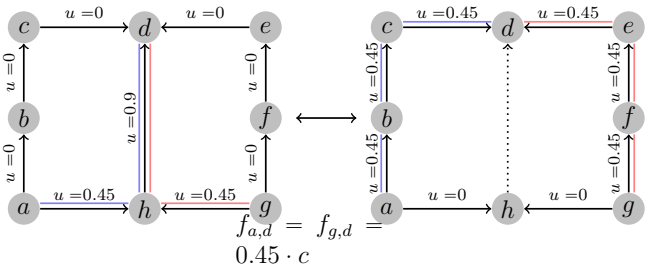


Figure 4.10: Congestion avoidance by turning *off*

We can conclude that STREETE behaves very well in all these scenarios. A large number of network links is turned *off*, enabling effective energy savings of up to 56% of the power consumed by the network links. We only consider uniform link speeds in this work. During the hours of low utilization, most networks are reduced to spanning trees. Moreover, STREETE is able to adapt the network against rises of traffic demands by turning links *on* when capacity is needed. Nevertheless, these results also indicate that STREETE was tested using "comfortable" conditions because it never faces a highly utilized network. In the next section we test the limits of the algorithm.

Pushing the limits of shortest path routing

In this section we evaluate STREETE in a more stressful condition. For this purpose, we execute the following experiment: we take the first traffic matrix from the data-set and initialize the algorithm. Afterwards, the evaluation follows the following pattern: at each validation time point, corresponding to an iteration of the STREETE framework, we increase the initial network load by an additive factor. The value of a flow at iteration t of the simulation is equal to $d_{a,b}(t) = d_{a,b}(0) * t$. This ensures that the load in the network increases continuously. We stop the simulation when the load of any link in the network reaches 100%. We compare STREETE against the optimum provided by the MILP solver (called "CPLEX" for simplicity) regarding both the number of active links and how long the algorithm can keep the network out of congestion.

Earlier we concluded that STREETE performs very well with the realistic traffic matrices. However, the results from Figure 4.11 reveal some shortcomings. The x axis corresponds to the t in the formula $d_{a,b}(t) = d_{a,b}(0) * t$, normalized to the moment when the shortest path routing in the full network revealed a congestion. As a result, $x = 5$ means that it was possible to absorb 5 times more traffic than what was possible to absorb with shortest path routing. The y axes shows the number of active links.

STREETE is very close to the optimum while the network is at low utilization. The only case when STREETE had to turn-*on* a lot of links is for the NSFNet network. To explain this case we can correlate with $x = 0$ on Figure 4.8c. We conclude that this case is due to the particularity of the traffic matrix used for validation: the demands are

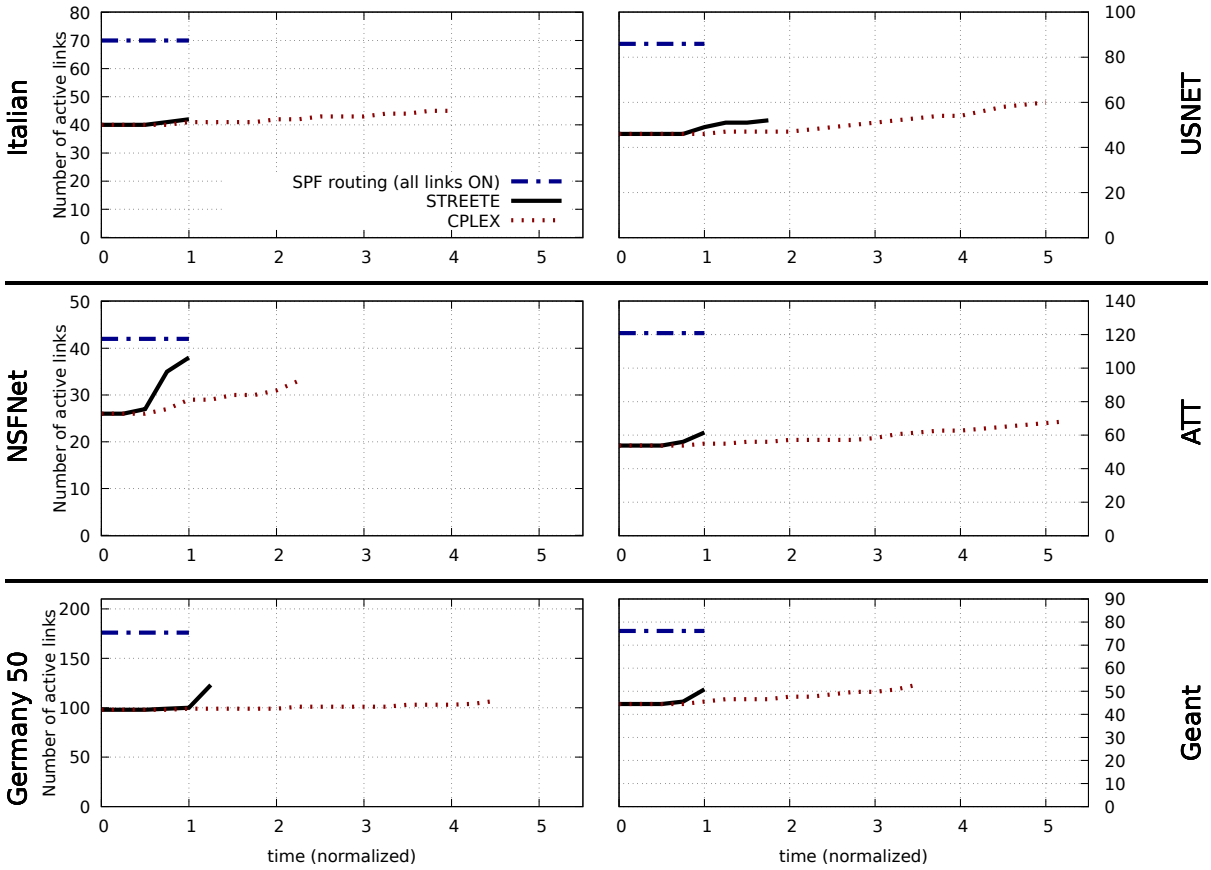


Figure 4.11: The number of links kept active in the network under continuous growth of network traffic. The x axes is normalized to the duration till congestion with shortest path routing.

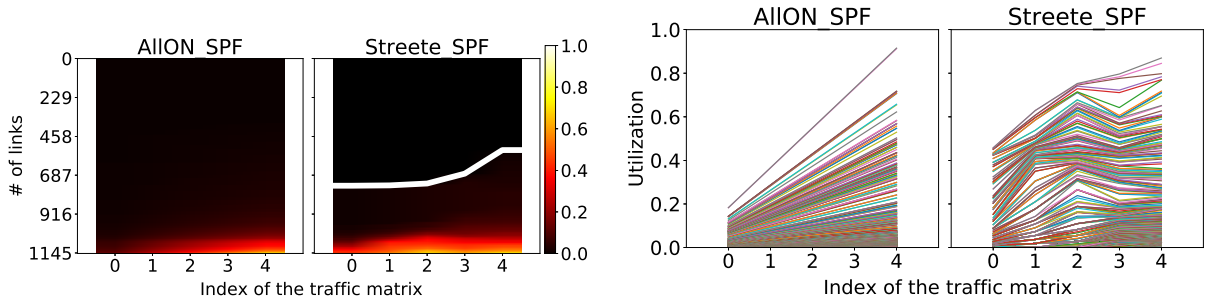
exceptionally high in this network, creating a high overall network load. It is perfectly normal that the algorithm turned *on* a lot of links to avoid congestion.

Something to be noted in Fig. 4.11 is the fact that STREETE is sometimes able to keep the network out of congestion longer than the shortest path routing. For example, this is the case in the USNET network, where STREETE is capable to route more than twice the amount of traffic of classical shortest path routing. The reason of this behavior was explained in the previous section, Fig. 4.10.

CPLEX is able to successfully route the demands in the network way beyond the point of congestion in the full network topology $\mathcal{G}$ with shortest path routing. Fig. 4.9 and 4.13 illustrate the reason behind this case on the example of the Generated200 network.

In the same scenario of constant traffic growth, both classical shortest path routing and STREETE congested the Generated200 network just after $tm = 4$. From the results provided on shortest path routing (left part of Fig. 4.12b), it becomes clear that the congestion occurs on one single link. Most of the links are not even close to having a high utilization. In this context, STREETE is even able to keep a lot of links *off* and reduce the energy consumption of the network (Fig. 4.12a, right).

Fig. 4.13 shows the distribution of the load in the network topology when STREETE



(a) Generated200, link state heatmap

(b) Generated200, link utilization

Figure 4.12: The heat-map (left) and per-link utilization (right) in the Generated200 network with increasing uniform all-to-all traffic

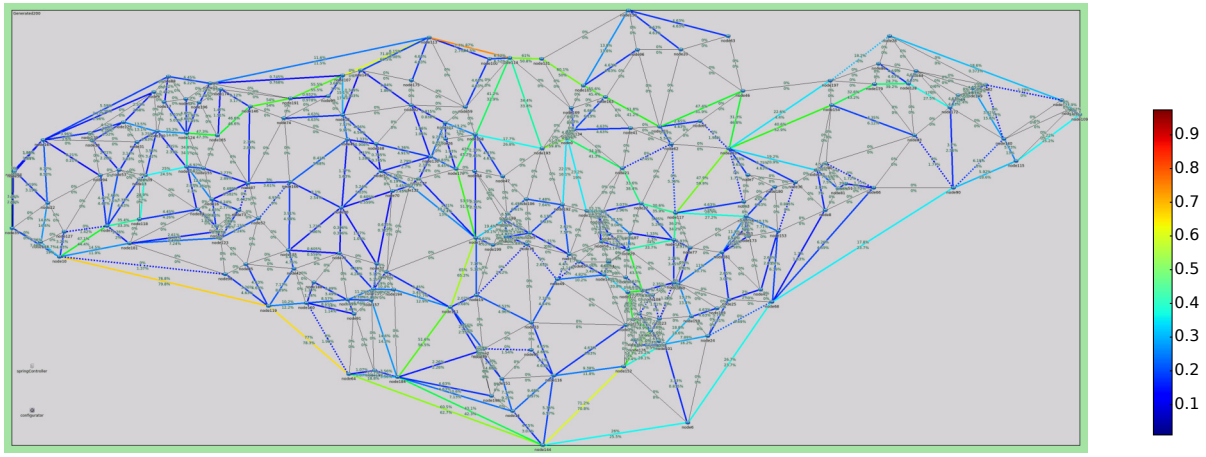


Figure 4.13: STREETE in the Generated200 network with the highest traffic before congestion ($tm = 4$ in Fig. 4.9)

is used. The highly loaded links from 4.12b can be easily identified by their orange/red color at the top-middle and on the bottom-left of the figure. The routing techniques based on shortest path routing tend to frequently use these links because they provide shortcuts to longer routes.

To make STREETE behave better under high network utilization, in the next chapter we propose a solution that allows it to reach the quality of CPLEX at the cost of increased computational complexity.

Computational time

One of our main goals was to create an online solution, capable to react fast to network changes. The computational time is very important, especially when it is necessary to turn link *on* to avoid an imminent congestion.

Table 4.2 indicates the maximum time taken by the algorithm on each of the analyzed networks. The most interesting results are provided by the Germany50, Coronet and Generated200 networks, because those are the biggest. The implementation with dynamic graph algorithm and a binary heap for Dijkstra computations allowed to scale

Network	STREETE	CPLEX
NSFNet	1ms	(1s)1827ms
Italian	3ms	(2s)2075ms
Geant	2ms	(3s)3491ms
USNET	4ms	(33s)33716ms
ATT	6ms	(21s)21488ms
Germany50	53ms	(2h*)7203220ms
CORONET	39ms	(**)-
Generated200	(8s)8311ms	(**)-

*CPLEX didn't find the optimal solution in the maximum allocated time (2h)
**CPLEX wasn't able to initialize the model

Table 4.2: Maximum time taken by STREETE and CPLEX on each of the analyzed networks

the algorithm to comfortably work on networks with 200 nodes.

Path stretch

Fig. 4.14 indicates how much the paths in the network increase due to consolidating the flows on a subset of network links. It is obvious that this action drastically extends some network paths. A minority are almost 15 times longer than the shortest path. A solution to this problem is to force some additional links to remain active. This decision has to be done at a topological level. We propose to use graph spanners [89] instead of spanning trees to guarantee that no unnecessarily long paths are created in the network as a result of turning links *off*. This case is left for future work.

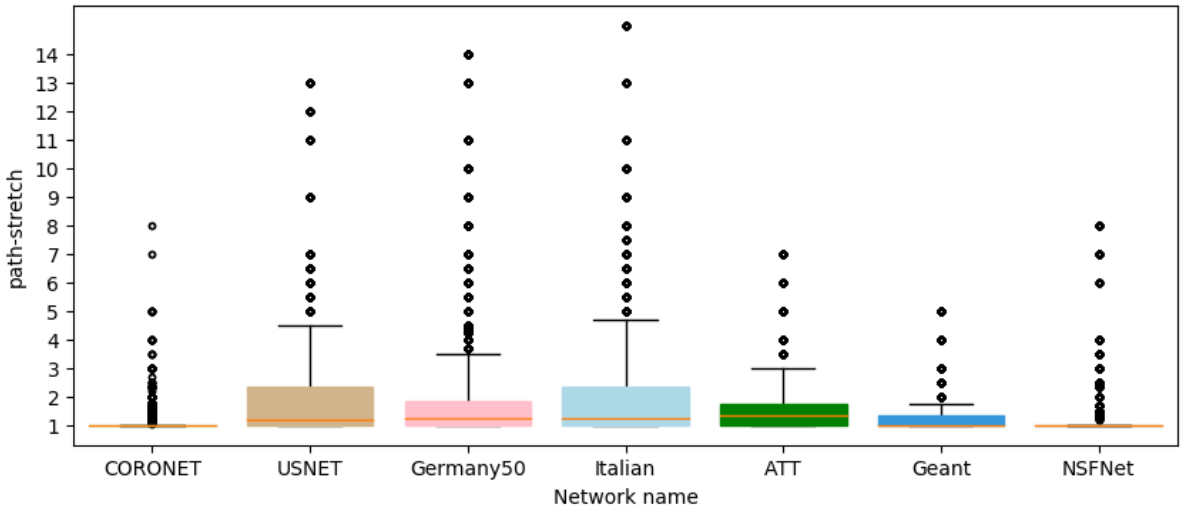


Figure 4.14: *path-stretch* in each of the evaluated networks using STREETE

To construct the boxplots we proceded as follows: for an evaluated traffic matrix we recorded the *path-stretch* of every origin-destination flow. That operation gave us $NB := |V| \cdot (|V| - 1)$ *path-stretch* values. The procedure was repeated for each evaluated traffic matrix. If 12 traffic matrices are evaluated, the boxplot shows the distribution of all the $12 \cdot NB$ recorded values.

4.4 Proof of concept and evaluation on a testbed

To evaluate our solution in real conditions, we built a small network test-bed relying on the ONOS SDN controller. A detailed description of this testbed along with results and lessons learned were previously analyzed in our works [J1][C2]. This section starts by presenting the platform and finishes with a discussion on the extracted results and the lessons learned during the evaluations.

The ONOS SDN controller

Open Network Operating System (ONOS) is an initiative to build an SDN [96] controller that relies on open-source software components. This SDN controller facilitates the work of developing network applications by providing northbound abstractions and southbound interfaces that transparently handle the forwarding devices, such as OpenFlow switches and other legacy network devices. In addition to a distributed core that enables control functions to be executed by a cluster of servers, ONOS provides two interesting northbound abstractions, namely the *Intent Framework* and the *Global Network View*.

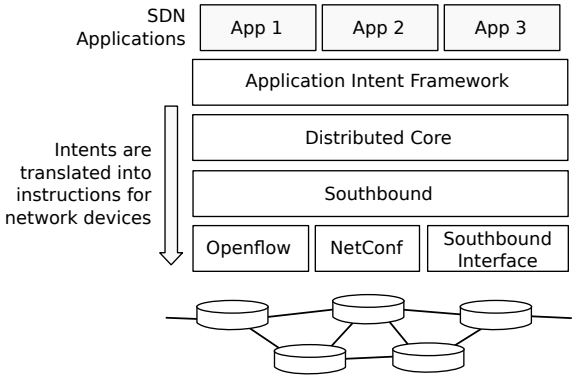


Figure 4.15: ONOS Intent Framework.

The intent framework, depicted in Figure 4.15, allows an application to request a network service without knowledge of how the service is performed. An intent manifested by an application is converted into a series of rules and actions that are applied to network devices. An example of intent is setting up a virtual circuit between two switches A and B. The global network view, as the name implies, provides an application with a view of the network and APIs to program it. The application may treat the view as a graph and perform several tasks that are crucial to traffic engineering, such as finding shortest paths.

The segment-routing based platform

Figure 4.16 illustrates our platform and its main components, depicting the deployment of switches, an SDN controller and applications. At smaller scale, the platform comprises components that are common to other infrastructures set up for networking research [97,

98, 99]. Moreover, we attempt to employ software used at the Grid5000 testbed [100][4] to which we intend to integrate the platform.

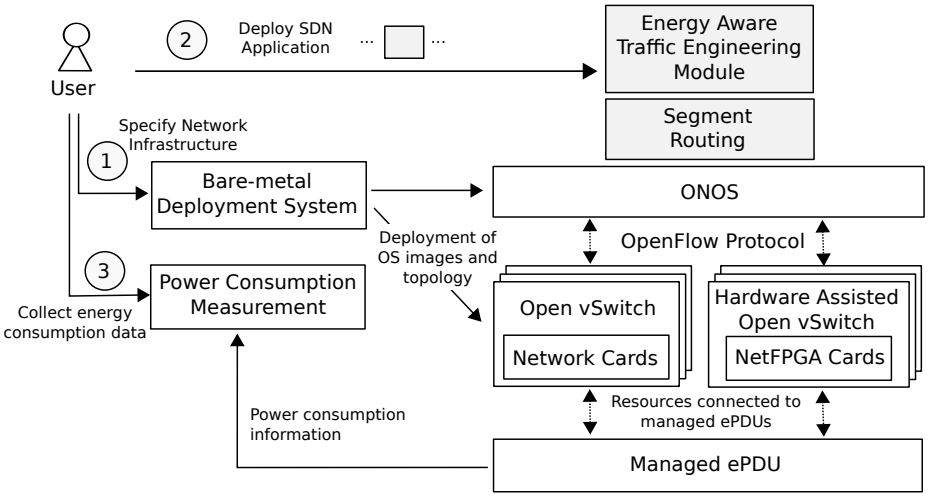


Figure 4.16: Overview of the network testbed.

To use the platform, a user requests: a slice or a set of cluster nodes to be used by an application as virtual switches or serving as traffic sources and sinks, an OS image to be deployed and a network topology to be used (step 1). We crafted several OS images so that nodes can be configured as SDN controllers and OpenFlow software switches, as discussed later. A bare-metal deployment system is used to copy the OS images to the respective nodes and configure them accordingly [101], whereas a Python application configures VLANs and interfaces of the virtual switches emulating point-to-point interconnects to create the user-specified network topology.

Once the nodes and the network topology are configured, the user deploys his or her application (step 2 in Figure 4.16). All cluster nodes are connected to enclosure Power Distribution Units (ePDUs)[5] that monitor the power consumption of individual sockets [102]. This information on power consumption may be used to evaluate the efficiency of an SDN technique (step 3).

The data plane comprises two types of OpenFlow switches, namely software-based and hardware-assisted. The former consists of a vanilla Open vSwitch (OVS) [103], whereas the latter OVS offloads certain OpenFlow functionalities to NetFPGA cards [104][6]. We use a custom OpenFlow implementation for NetFPGAs, initially provided by the Universität Paderborn (UPB) [105], that performs certain OpenFlow functions in the card, *e.g.* flow tables, packet matching against tables, and forwarding. Although the NetFPGA cards are by default programmed as custom OpenFlow switches, a user can reprogram them for different purposes by copying a bitstream file to their flash memories and rebooting the system.

The current testbed [7], depicted in Figure 4.17, comprises eight servers – five Dell R720 servers equipped with a 10Gbps Ethernet card with 2 SPF+ ports each and three

[4] `https://www.grid5000.fr`

[5] `http://www.eaton.com/Eaton/index.htm`

[6] `http://netfpga.org/site/#/systems/3netfpga-10g/details/`

[7] The testbed was financially supported by the chist-era SwiTching And tRansmission (STAR) project

HP Z800 servers with NetFPGA cards with 4 SPF+ ports each. All servers also have multiple 1Gbps Ethernet ports. The SPF+ ports have optical transceivers and are all interconnected by a Dell N4032F L3 switch whereas two 1Gbps Ethernet ports of each server are connected to a Dell N2024 Ethernet switch. This configuration enables testing multiple network topologies.

The infrastructure and the use of ONOS satisfy some requirements of energy-aware traffic engineering, namely providing actual hardware, allowing for traffic information to be gathered, using actual network protocols, enabling the overhead of control and management to be measured, and monitoring the power consumption of equipment. Some energy-optimisation mechanisms, however, are still emulated, such as switching off/on individual switch ports. Although the IP cores of the Ethernet hardware used in the NetFPGA cards enable changing the state of certain components, such as switching off transceivers, that would require a complete redesign of the employed OpenFlow implementation. It has therefore been left for future work.

Figure 4.17: Photo of the experimental platform

We implemented the algorithms as an ONOS OSGI module. The base of our solution was the "Segment Routing" module which is available in the ONOS distribution. As a result, we leverage features of the Segment Routing/SPRING for traffic engineering, such as the possibility to atomically change routes only on ingress devices without having to synchronize updates between devices. This flexibility comes at the cost of a small increase of the mean packet header size.

Segment-Routing application

ONOS provides an application implementing the SPRING(Segment Routing) protocol. We integrated our STREETE framework into this application.

We use a series of ONOS components, including its topology information, flow-rule services, and traffic flow objectives. As shown in Figure 4.18, a service *Manager* triggers the creation of remaining components when it is launched. Our energy-aware module, which comprises the proposed traffic-engineering algorithms, registers a flow-rule listener to measure flow traffic and link utilisation. The configuration component loads a file that specifies how switches are connected to local networks. This information is then augmented by a topology discovery process. Once the topology is updated, default shortest-path rules are created to guarantee that hosts from a network connected to a switch may reach hosts linked to another switch. A rule consists of a forwarding objective comprising a traffic selector and a treatment. Selectors and treatments result in sets of OpenFlow instructions that are passed to the switches. MPLS push/pop forwarding objectives are created for switches that do not have ports in the source and destination segments — *i.e.* are neither ingress nor egress switches — and normal IP forwarding objectives are built otherwise. While the service is running, our energy-aware module is notified about

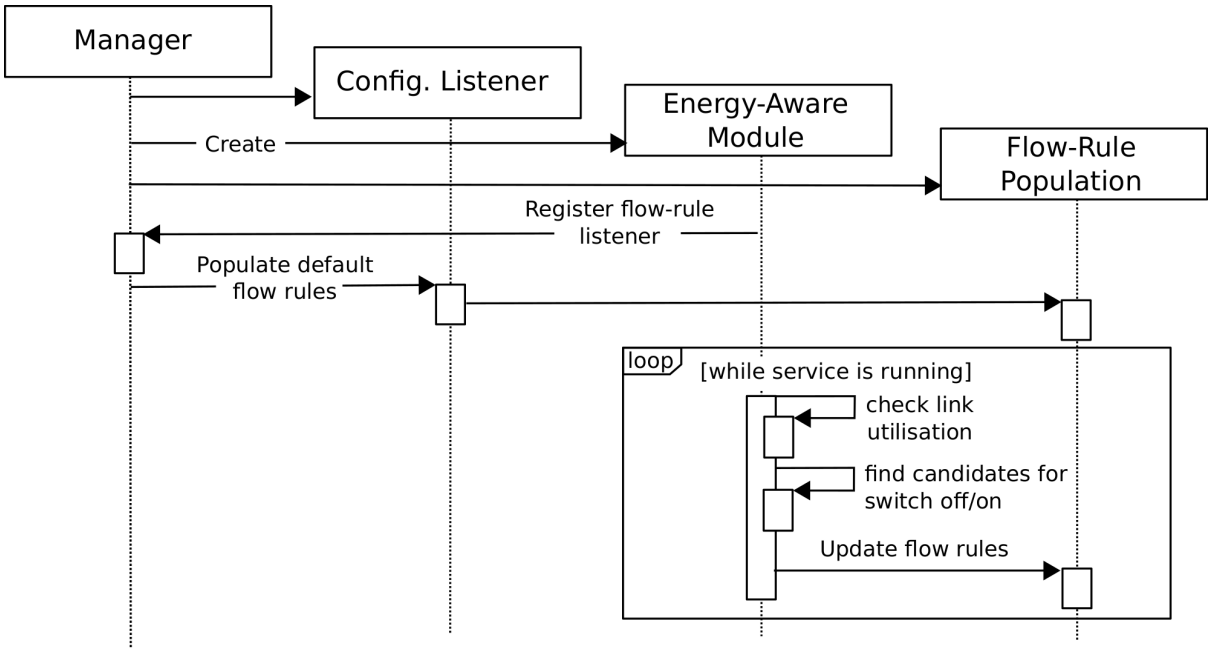


Figure 4.18: Start phase of the segment-routing application.

changes in topology as well as link utilisation, and periodically evaluates whether there are links to switch off/on. If changes in the link availability are required, the energy-aware module requests a flow-rule update to the Flow-Rule Population module.

The experimental validation of our solutions allowed us to discover some elements to be taken into consideration, which are presented in the next section.

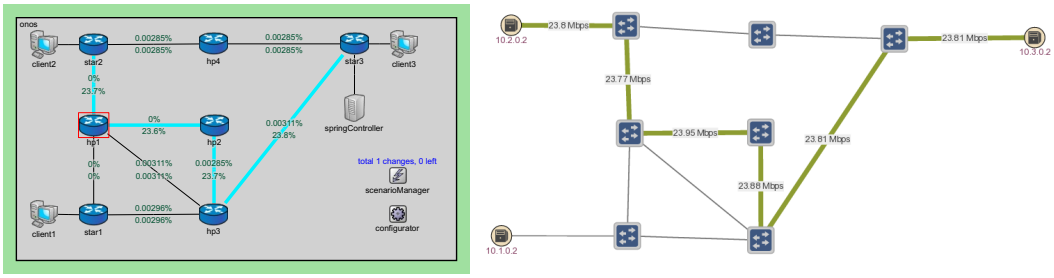
4.4.1 Lessons learned from experimental evaluation

We implemented our algorithms on the test-bed presented in the previous section, using the ONOS SDN controller and hybrid software/hardware(NetFPGA) openflow switches. The evaluated network contains 7 nodes with 10G NetFPGA cards. A modified Open vSwitch (OVS) v1.4 is used to offload some OpenFlow rules into the hardware. Rules which cannot be offloaded, are treated by the Open vSwitch software kernel stack.

Due to the small network size, we consider that the results in terms of heuristic quality and speed are not representative. Even the MILP solver is able to provide an optimum solution in negligible time on such a small network. That is why, we concentrate on the control traffic overhead and traffic matrix reconstruction. Similar work already exists in the literature [106], but the analysed scenarios are not applicable to our case. In particular, we are interested in an online, fast response to traffic burst with lots of flow modifications each time a link is turned on. The literature usually assumes that flow modification are spread in time.

Although switching off underused links may be effective from energy efficiency perspective, sudden bursts in traffic may lead to congestion, hence requiring links to be made available. Performance evaluation using discrete-event simulation and UDP-like traffic has shown that the approach successfully reacts to traffic bursts without incurring considerable packet loss. It is assumed, however, that the SDN controller gathers the in-

formation about link utilisation from switches every second and that a decision to power a given link back on may be taken and enforced quickly.



(a) Interface of OMNeT++ (b) Interface of the ONOS SDN controller

Figure 4.19: Interfaces of the simulator and the testbed showing a data flow avoiding the shortest path

We performed a simple test and measured the time needed for a controller to decide whether a link should be switched on. A small network topology was considered, as depicted in Figure 4.19. The Figure also shows the ONOS graphical interface and a data flow (green lines). The network starts with a minimal number of links turned on, forming a spanning tree, and with a TCP flow that nearly exceeds the utilisation threshold, above which the controller decides to turn on more links to handle congestion. A second flow is then injected, thus exceeding the threshold and forcing the controller to switch links on; we measure the time from flow injection to a switch-on decision. In the simulation, the decision takes on average 1.075 seconds, with most of the time spent gathering information on link utilisation. In the testbed, the time is on average 20% higher than in simulation.

We notice that the difference in results between simulation and real testbeds are generally due to simulations assuming zero delay at multiple parts of the processing pipeline and the manner network events are handled. While a single delay simplification would have marginal impact on the results, multiple delays along the packet processing pipeline can account for up to 30% difference in the time to react to changes. Examples of delay simplifications during discrete-event simulations include: instantaneous insertion of forwarding rules into the data path, immediate update of routing tables, fast propagation of flow counters from the simulated hardware ports to the software of the SDN switch, and instantaneous processing of IP UDP/TCP packets. Generally, the only delay properly handled by a simulator is packet queueing time.

Existing work has already shown that updating the data path forwarding rules is slow in current commercial SDN switches [77]. Google employees report[78] that their SDN-based WAN had an outage due to this issue on propagation of forwarding rules. Improvements can be made in the simulation software to account for some delay, and in the hardware design itself to reduce the time to propagate rules.

Other issues that we investigated concern the stability of the algorithms and the impact of traffic re-routing on TCP flows. Unlike traditional networks where changes in link availability are sporadic, under energy efficient traffic engineering, frequent changes may be the rule. Re-routing TCP flows, however, may lead to serious performance degradation due to segments arriving out of order, which in turn result in multiple duplicate ACKs and hence triggering the TCP congestion algorithms at source. Even though the algo-

rithms in the simulator mimic the behaviour of their corresponding theoretical models, they differ from the actual network software implementations provided by certain operating systems. In our in-depth analysis, which we present in the chapter 6 of this thesis, we observed that the Linux kernel, for instance, includes several non-standard optimisations [107]. While simulations highlighted that re-routing TCP flows severely impacts the throughput of the transported TCP flows, empirical evaluation on the testbed demonstrated almost no impact under the same conditions. We believe that existing work that wraps real network software stack into simulators[8] may help minimise this issue.

4.5 Conclusion

Solutions for improving the energy efficiency of wired computer networks propose to turn the links *off* during periods of low network utilization. This chapter follows this trend and proposes a reactive Software-Defined Network (SDN)-based framework which dynamically monitors the network status and turns the links *on* or *off* to provide the capacity needed for routing the network flows. Compared to the related work in this field, we focus on an online solution capable to instantly react to unpredictable network events and keep the network out of congestion while maintaining a low number of links active, thus reducing the energy consumption of the network.

To maintain a low computational overhead, our algorithm relies on an innovative idea which maintains two different views of the network topology in parallel. This allows to improve the quality of the results, especially for finding the correct links to ignite, field which is mostly overlooked by the research community. Moreover, it allows to reduce the impact of historical decisions on the execution of the turn-*off* algorithm.

We implemented our solution using state of art dynamic shortest path algorithms to further increase the execution speed. Moreover, the fact of entirely relying on shortest path computations, together with the use of the SPRING source routing protocol, allows to keep the control traffic overhead at a minimum.

We evaluate the proposed algorithms using real backbone network topologies with real traffic matrices and conclude that good results are obtained at low traffic load both in number of turned-*off* links and in its ability to avoid network congestion. We also implemented a proof of concept on a testbed using the ONOS SDN controller in order to evaluate the framework on real hardware.

This extensive validation allowed to detect that there is room for improving our solution at high load, leading to the work which we'll present in the next chapter. Moreover, the validation on the testbed revealed some instabilities due to the bad inter-operation between STREETE and the TCP congestion control, leading to the analysis performed in the chapter 6 of this thesis.

[8] `http://www.wand.net.nz/~stj2/nsc`

Chapter 5

Consume Better: traffic engineering for optimizing network usage

In the previous chapter, we presented a framework for reducing the energy consumption of IPoWDM networks by turning links *on* and *off* depending on the network load. However, despite the energy saving potential and small footprint of the presented solution, the results highlighted a potential shortcoming: a lot of resources can remain underutilized due to the minimum hop shortest path routing. In this chapter, we intend to address this weakness by including a load balancing technique in the STREETE framework to enable better utilization of the active links.

We start the chapter with a description of our first attempt to design such a load balancing technique and justify the reason why we did not explore this direction further. In the second part of the chapter, we present another, much more elegant and efficient, attempt to design and integrate into STREETE an Software-Defined Network (SDN) based online traffic engineering solution based on a state-of-art approximate multi-commodity flow algorithm to keep the computational complexity as small as possible. This solution also leverages the power of SPRING source routing to reduce the complexity and cost of centralized management.

5.1 CF: Searching for the perfect cost function

5.1.1 General overview

Our first attempt to avoid the shortcomings of the minimum hop shortest path routing still relies on the Dijkstra algorithm, but with a cost of traversing links that depends on their utilization. Instead of always routing flows via the smallest number of hops, we select the least utilized route by using a carefully crafted cost function which makes the cost of traversing a link dependent on its utilization. To describe the change more formally, we revisit the cost function $l : E \rightarrow \mathbb{R}^+$ which we introduced in chapter 3. Until now, we implicitly assumed that l is a constant. In this section, we force $l = f(u(e))$, with $f : [0,1] \rightarrow \mathbb{R}^+$.

An example of such functions is presented in Fig 5.1. For example, a link utilized at 10% of its capacity will have a cost of 1.88, while a utilization of 80% will increase the cost of the link to 3.27. The intuition may tell us that routing flows in the network over

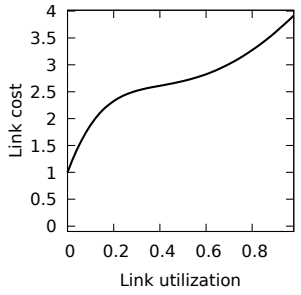


Figure 5.1: Example of link cost function

shortest paths with this cost function will naturally diverge the traffic from the highly utilized links by choosing longer paths regarding hop counts, but a lesser cost.

The computational complexity of such a solution may be quite significant. A trivial algorithm will compute the shortest path for each origin-destination pair, resulting in $|V|^2$ shortest path computations. To reduce the complexity, we leverage a particularity of the sizes of network flows. More precisely, different authors [108] [109] [110] analyzed real traffic traces and concluded that most network traffic is transported by a low number of elephant network flows. This result was also confirmed in the case of aggregated origin-destination demands [111]. Our solution starts by breaking the demands from the traffic matrix into two groups. The first one contains the requirements of mice flows, while the second one the demands of elephant flows. For this purpose, we use a classification algorithm designed for heavy-tailed distributions [112].

The algorithm will route all the elephant demands one by one using a dedicated shortest path computation per demand. However, to keep a low computational overhead, the mice demands will be routed per-source: all the demands from the same source will be routed using our custom modified shortest path algorithm which greedily adapts the costs of links at the same time as computing the shortest path.

5.1.2 Algorithms

The only non-classical algorithm used by our solution is a modified Dijkstra created for routing the mice flows. It is a greedy algorithm which computes a tree for routing the flows originating from a given source node $v_{root} \in V$ while dynamically updating the link costs in conformance with the cost provided by the function $l(e)$. In particular, during the execution of the algorithm, the costs $l(e)$ are not static, they change as consequence of previous routing decisions and influence the selection of network paths for the subsequent flows. Nevertheless, it remains a greedy algorithm: at each iteration, the previous choices are never re-evaluated and only influence the subsequent ones.

Algorithm 6 Constructs the tree $\mathcal{T}$ which gives the routes for the demands originating from the root node v_{root} towards all other nodes in the network while greedily respecting the link costs defined by the function $l(e)$

```
1: function RouteInTree(T(V, E_T), v_root ∈ V, d_{v_root,*})
2:     for all v ∈ V do
3:         distance[v] ← ∞
4:         parent[v] ← undefined
5:     end for
6:     treatedList ← []                                   ▷ Empty list of nodes
7:     visitedList ← [v_root]                             ▷ List of nodes containing one element (v_root)
8:     distance[v_root] ← 0
9:     while visitedList.notEmpty() do
10:        a ← visitedList[0]
11:        lowestDist ← ∞
12:        for all v ∈ visitedList[1 :] do                ▷ all elements in the list except the first one
13:            distance[v] ← ∞
14:            for all u ∈ V s.t. uv ∈ E_G and u ∈ treatedList do
15:                if distance[v] > distance[u] + l(uv) then
16:                    distance[v] ← distance[u] + l(uv)
17:                    parent[v] ← u
18:                end if
19:            end for
20:            if lowestDist > distance[v] then
21:                lowestDist ← distance[v]
22:                a ← v
23:            end if
24:        end for
25:
26:        visitedList.delete(a)
27:        treatedList.append(a)
28:        if a ≠ v_root then
29:            E_T ← E_T ∪ {(parent[a],a)}
30:        end if
31:
32:        v ← a
33:        while v ≠ v_root do
34:            f_(parent[v],v) ← f_(parent[v],v) + d_{v_root,v}
35:            v ← parent[v]
36:        end while
37:        for v ∈ treatedList[1 :] do                    ▷ all elements in the list except the first one (v_root)
38:            distance[v] ← distance[parent[v]] + l((parent[v],v))
39:        end for
40:
41:        for all w ∈ V s.t. aw ∈ E_G and w ∉ treatedList ∪ visitedList do
42:            visitedList.append(w)                      ▷ non-visited neighbors of a
43:            parent[w] ← a
44:        end for
45:    end while
46: end function
```

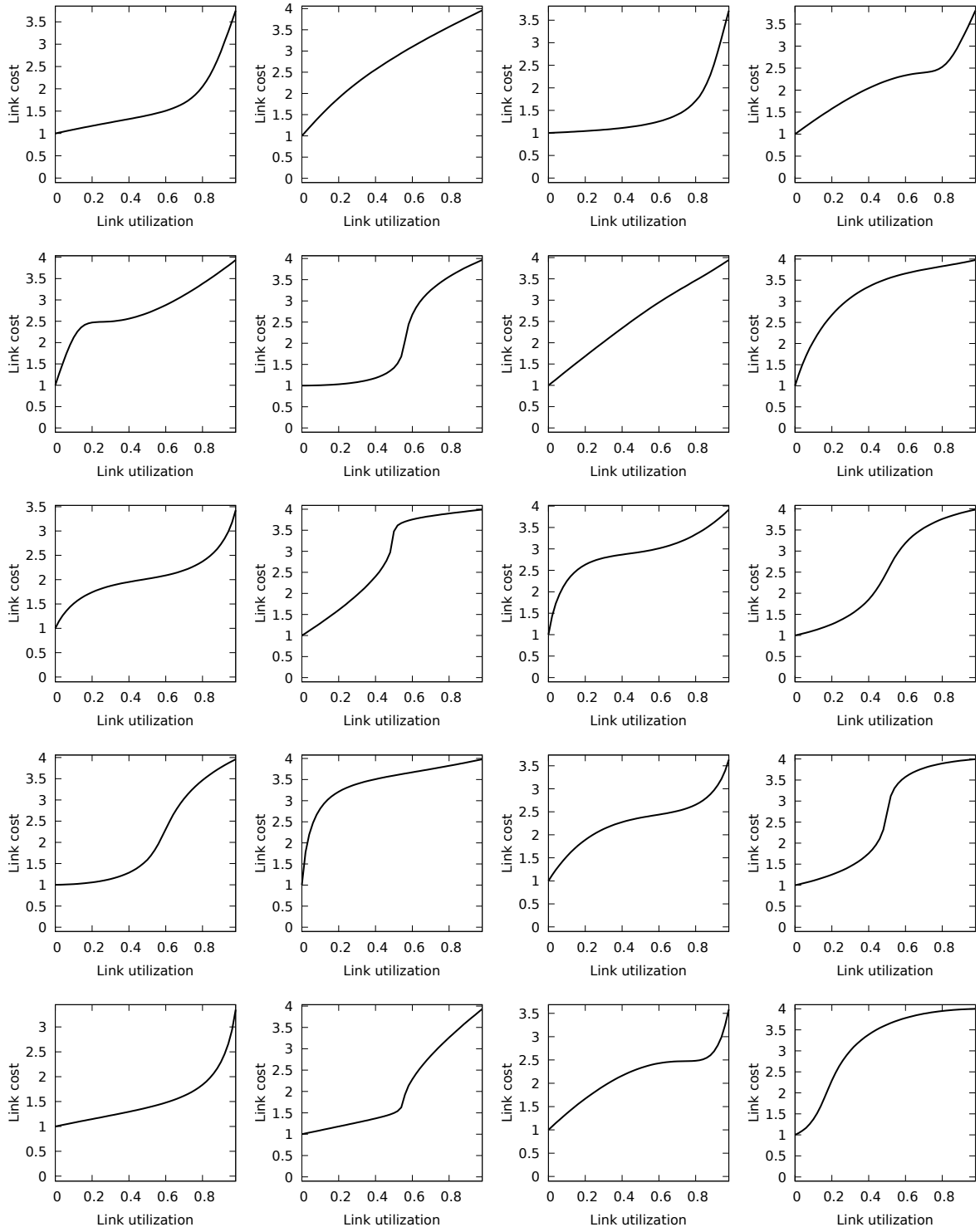


Figure 5.2: 1+3·cubic_bezier(random parameters)

The algorithm proposed by us (Alg. 6) acts similarly to Dijkstra's. At each step, it selects the node $a \in V$ that is the closest to v_{root} under the length function l and was not yet connected to the routing tree $\mathcal{T}$. This operation is performed at lines 10-24. After that, the selected node is marked as treated and is connected to the tree being constructed (lines 26 - 30).

At lines 32-36, the demand $d_{v_{root},a}$ is placed into the network using the unique path from the root node v_{root} to a in this tree $\mathcal{T}$. This action will probably change the cost of passing through the corresponding edges. As a consequence, the distance of nodes from the root v_{root} may also change. At lines 36-39, the distances are updated to reflect the new costs of links.

Finally, the last *for* loop ensures that the neighbor nodes of a that were not yet encountered by the algorithm will be considered in the next iterations of the outer *while* loop.

5.1.3 Evaluation methodology and first results

Evaluation parameters

In the previous chapter, we highlighted that STREETE without load balancing is capable of delivering good results when the utilization of the network is inferior or equal to the utilization achieved by the classical shortest path routing. The bad performance starts to be noticeable at higher utilization. That is why we focus our evaluation with scenarios using high traffic load.

Similarly to the previous chapter, we evaluate the time the algorithm can keep the all-*on* network out of congestion. The traffic demands are additively increased at each step until the network becomes congested. We also analyze the utilization of the most loaded link. This metric is directly related to the capacity of the algorithm to keep the system out of congestion if the load increases. The problem of minimizing the utilization of the most loaded link is known under the name of "minimum bottleneck utilization routing problem" and is equivalent to the maximum concurrent flow problem [88].

A critical part of the proposed algorithm is in selecting the correct link length function. Before investigating this problem further, we decided to test the viability of the proposed solution using a large number of randomly generated cost functions. We automatically generated cost functions using the cubic bezier parametric curve with random parameters. Figure 5.2 illustrates multiple examples of such functions. We tested the algorithm with each one and selected the best routing, i.e. that results in the smallest maximum link utilization.

CPLEX model used as baseline

The optimal baseline was computed using the following linear program, which corresponds to the standard node-arc formulation of the "minimum bottleneck utilization routing problem".

Considering the variables $f_e = \sum_{(i,j) \in V^2 : d_{i,j} > 0} f_e^{i,j}$, the model objective is to minimize the utilization of the most loaded link:

$$\textbf{Minimize: } max(\tfrac{f_e}{c(e)})$$

Subject to the flow conservation and edge capacity constraints:

$$\sum_{e \in w_v^-} f_e^{i,j} - \sum_{e \in w_v^+} f_e^{i,j} = \begin{cases} -d_{i,j}, & \text{if } v = i \\ d_{i,j}, & \text{if } v = j \\ 0, & \text{otherwise} \end{cases} \qquad (5.1)$$

$$\forall v \in V, \forall (i,j) \in V^2 : d_{i,j} > 0$$

$$f_e \leqslant c_e \qquad \forall e \in E \qquad (5.2)$$

Maximum link utilization

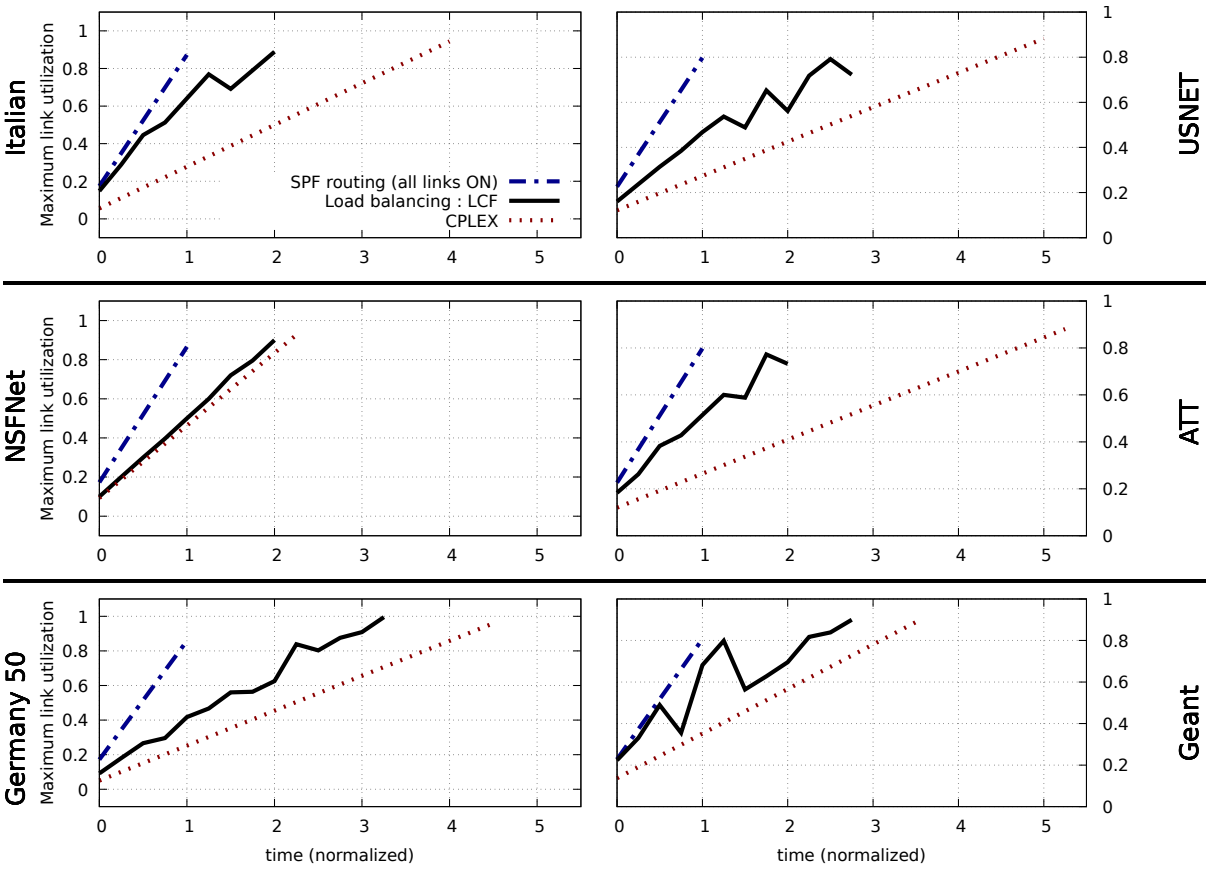


Figure 5.3: Utilization of the most loaded link under continuous growth of network traffic. x axes are normalized to the moment of congestion with classical shortest path routing.

The results are shown in Fig. 5.3. We slightly improved the distribution of the load on the network and kept it out of congestion for up to 3 times longer than with shortest path routing. However, the quality of the result is volatile: the maximum link load varies a lot. Our tests showed that even tiny variation in the shape of the cost function could lead to extreme changes in the quality of the final solution.

Moreover, the additive nature of the cost of network paths does not suit the problem. An unfavorable case that frequently occurs in practice is depicted in Fig. 5.4. Two paths are possible between the node a and c: i)(ad,dc); and ii) (ab,bc). The first path contains two links with a utilization of 50%. The second path passes through two links, with a usage of 90% and 5%. At the same time, due to the additive nature of the lengths of the path, both paths have the same length of 4 and are considered equally good for routing the flow ac. This behavior is not desirable in a load balancing technique. The first path should probably be preferred to avoid increasing the utilization of the link ab even further.

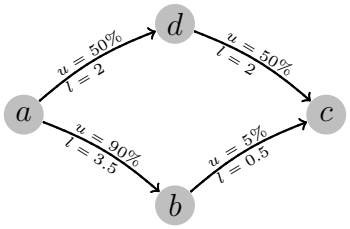


Figure 5.4: Two paths of equal cost between the node a and c

Our preliminary tests showed particularly bad results in the context of the heterogeneous links: the flows were not discouraged from passing through low speed links, leading to their congestion very fast.

We believe that there may exist techniques to improve the stability of the presented approach, but we abandoned this direction in favor of the solution proposed in the next section.

5.2 LB: maximum concurrent flow for load balancing with SDN

In this section, we present our second attempt to include a load balancing technique into STREETE. Similarly to the solution presented in the previous section, the central optimization phase of the current solution relies on assigning costs to network links. During the execution of the algorithm, the cost of each link increase when demand is routed through the link. Moreover, the algorithm also uses shortest path computations and relies on these costs to prioritize the paths passing through the less loaded links. However, there is a huge difference between the two approaches. The reason why the forthcoming algorithms obtain much better results is in the fact of repeatedly routing tiny fractions of the traffic demands into the network. Moreover, at each iteration, the costs of the links increase by a minimal factor. For example, thanks to the very slow variations of costs of the links, the unfavorable case presented in Fig. 5.4 will be avoided.

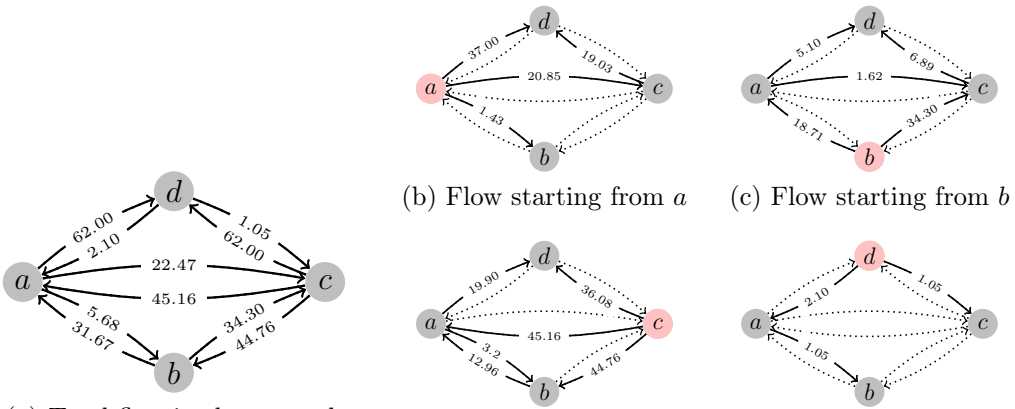
As a consequence of iteratively routing small fractions of demands at each iteration of the algorithm, the integral demands will be split into multiple paths. The originality of our contribution consists in proposing a way to efficiently achieve the multipath forwarding by combining centralized and distributed optimization. This way we also reduce the control traffic overhead, because the SDN controller does not have to orchestrate the data-planes of SDN switches explicitly. The SDN controller transmits only a set of constraints to each switch. These constraints are used to re-compute the network paths and update the forwarding tables locally. We rely on the source routing protocol SPRING to be able to atomically change the paths of the flows by only updating the forwarding database of ingress network devices.

The solution that we propose combines i) the advantage of making global routing decision in the SDN controller; with ii) the offloading of some computations and the multipath forwarding rule construction to the SDN switches.

5.2.1 General overview

In this section, we give a general overview of our solution without emphasis on details. The algorithms will be presented in the next section.

The SDN controller performs the first step of our solution. It relies on Karakostas'[5] work on solving the approximate maximum concurrent flow problem via the primal/dual method. It constructs a solution to the dual problem, i.e. the length(cost) of the edges, by making slow variations in the values of the variables of the primal i.e. the amount of flow per path. During the execution of this algorithm, the centralized controller keeps track of the total flow of each ingress node. In particular, for each source node, the controller computes the amount of total flow passing through the links of the network.



(a) Total flow in the network

(b) Flow starting from a

(c) Flow starting from b

(d) Flow starting from c

(e) Flow starting from d

Figure 5.5: The total flow in the network (5.5a), and the decomposition of this flow per source node

Fig. 5.5 shows an example of output computed by this algorithm in a small network with hypothetical all-to-all communications. In particular, we are interested in the following output: i) the best possible routing of the flow in the network which minimize the utilization of the most loaded link (Fig.5.5a); ii) the decomposition of this multicommodity flow per source node (Fig. 5.5b - 5.5e).

The second step of our solution corresponds to sending the output from the previous algorithm to the corresponding SDN switches. For example, it sends the following list to the node a: $((ab, 1.43), (cd, 19.03), (ad, 37.00), (ac, 20.85))$. In particular, SDN controller informs each SDN switch of how the data it originates must be transmitted through the network and provides a way to achieve a global optimization of the network traffic. On the other hand, the data transmitted to the SDN switches does not contain any explicit information about the routes to be taken by each origin-destination flow. The communication overhead decreases, but the ingress SDN switches must now locally compute the forwarding paths in a way to respect the constraints given by the centralized controller. For example, the switch a will try to locally route the traffic to match the flow from Fig.5.5b.

The last step of our solution is to locally compute the forwarding rules on SDN switches while matching the flow transmitted by the SDN controller. We insist on the fact that the flow sent by the SDN controller is, by construction, a valid flow, but is not necessarily a minimum-cost one. As a consequence, it may contain cycles which we would like to reduce to simplify the task of building the forwarding rules. An example of cycle is shown in Fig. 5.5d: $a \to b \to a$. The SDN switch will start with applying the cycle-canceling algorithm[113] on the received flow. This operation has the advantage of outputting a Directed Acyclic Graph (DAG). The SDN switch can now easily compute the forwarding paths in the network by leveraging the properties of the DAGs. In particular, the fact that it can be topologically sorted. Moreover, the switch has the visibility into the composition of the flow at a granularity which cannot be transmitted to the SDN controller. It can leverage this knowledge to split the origin-destination flows into sub-flows and send each sub-flow over a separate path to match the constraints imposed by the controller correctly.

5.2.2 Step 1: maximum concurrent flow computed by the SDN controller

The core of the centralized optimization part of our solution relies on the algorithm by Karakostas[5] for computing an approximate solution to the maximum concurrent flow problem.

Notations

Maximize λ under the constraints:

$$\sum_{P \in \mathcal{P}_{i,j}} f(P) = \lambda \cdot d_{i,j} \qquad \forall (i,j) \in V^2 : d_{i,j} > 0$$
$$\sum_{P \in \mathcal{P} : e \in P} f(P) \leqslant c(e) \qquad \forall e \in E$$
$$f(P) \geqslant 0 \qquad \forall P \in \mathcal{P}$$

(a) Primal

Minimize $\sum_{e \in E} l(e)c(e)$ under the constraints:

$$\sum_{e \in P} l(e) \geqslant l(SP_{i,j}) \qquad \forall (i,j) \in V^2 : d_{i,j} > 0 \quad \forall P \in \mathcal{P}_{i,j}$$
$$\sum_{(i,j) \in V^2 : d_{i,j} > 0} l(SP_{i,j}) d_{i,j} \geqslant 1$$
$$l(e) \geqslant 0 \qquad e \in E$$

(b) Dual

Figure 5.6: The primal and dual of the maximum concurrent flow problem in a path-flow formulation

For convenience, we summarize the notations. Fig. 5.6 shows the edge-path formulation of the maximum concurrent flow problem that was introduced in a previous chapter. We also recall that P is a path in the network, $\mathcal{P}$ represents the set of all the paths and $\mathcal{P}_{i,j}$ is the set of all the paths starting at node $i \in V$ and finishing at node $j \in V$. Moreover, $f(P)$ is the amount of flow passing through the path P, $d_{i,j}$ is the demand from node i to node j. The function $l : E \rightarrow \mathbb{R}^+$ and $c : E \rightarrow \mathbb{R}^+$ give, respectively, the cost of passing through a link $e \in E$ and the capacity of this link. Finally $SP_{i,j}$ is the shortest path among all the paths from i to j: implicitly assumed under the length function l. To simplify the notations, let $D(l) = \sum_{e \in E} l(e)c(e)$ be the value of the dual objective function; $D(l^*)$ be the dual optimum; and define $a(l) = \sum_{(i,j) \in V^2 : d_{i,j} > 0} l(SP_{i,j})d_{i,j}$ to be the expression from the last constraint of the dual.

Algorithm

The Alg. 7 presents the algorithm as implemented and used by our solution. It closely follows the original algorithm [5]. Nevertheless, the structure was reorganized for easier implementation and possibility to compute the per-source flow. Moreover, we incorporated the details of some non-trivial building blocks used as primitives by the Karakostas.

The algorithm starts with the trivial valid solution $f = 0$ for the primal LP and a constant length function $l(e) = \frac{\delta}{c(e)}$ which represents an invalid solution for the dual LP. After that, the algorithm proceeds in multiple phases to construct the optimum solution for the dual LP. Each step corresponds to an iteration of the loop at line 4. δ is a very small constant which is crucial to the proof of the algorithm and is specifically chosen for this purpose. More details on this constant will be provided in the next section.

In each phase, for each source node *src*, the algorithm uses the shortest path tree $\mathcal{T}_{src}$ rooted at this source which is computed by the Dijkstra's algorithm. This allows to simultaneously route the demands towards all the destinations (line 10). The details of the *RouteInTree* procedure was provided in the previous chapter (section 4.2.3). The algorithm then increases the length/cost of the links which have just been used exponentially to the amount of data which transited through the link (line 17). From an LP perspective, this action corresponds to adjusting those variables of the dual which violate the most its constraints. This is done while keeping a relationship between the variables of the primal and those of the dual.

A more intuitive interpretation can also be given: the algorithm repeatedly routes the demands over the least loaded links and increases the cost of these links exponentially to the added load. This way, the most loaded links will become more costly and have less probability to be used in the next steps of the algorithm.

By construction, the computed flow respects the flow conservation constraints and non-negativity constraints but may violate the link capacity constraints. At the end, the flow must be scaled down to obtain the minimum utilization routing used in the next steps of our solution.

Proof of the algorithm

This section unifies the proofs from the four works [88][114][115][5], as none of them provides a unified proof with all the necessary details.

Algorithm 7 Approximate maximum fractional concurrent flow

procedure FRACTIONALMCFP($\mathcal{G}$)
  initialize $l(e) \leftarrow \delta/c(e), \forall e$
  $D(l) \leftarrow m\delta$
  while $D(l) < 1$ **do**
    for all $v_{src} \in V$ **do**
      Initialize $d'_{v_{src},v_{dst}} \leftarrow d_{v_{src},v_{dst}}, \forall v_{dst} \in V$
      moreTraffic $\leftarrow \exists v_{dst}$ s.t. $d'_{v_{src},v_{dst}} > 0$
      while $D(l) < 1$ and moreTraffic $= true$ **do**
        $\mathcal{T}_{src} \leftarrow$ DIJKSTRA(v_{src})
        $f_{(*,*)} \leftarrow$ ROUTEINTREE($\mathcal{T}_{src}, v_{src}, d'_{v_{src},*}$)
        $\sigma \leftarrow \max(1, \max_{uv \in E_{\mathcal{T}}: f_{(u,v)} \in f_{(*,*)}}(f_{(u,v)}/c(uv)))$
        for all $f_{(u,v)} \in f_{(*,*)}$ **do**
          $f_{(u,v)} \leftarrow f_{(u,v)}/\sigma$
          $f^{v_{src}}_{(u,v)} \leftarrow f^{v_{src}}_{(u,v)} + f_{(u,v)}$
          $d'_{(u,v)} \leftarrow d'_{(u,v)} - f_{(u,v)}$
          $D(l) \leftarrow D(l) + l(uv) * \epsilon \frac{f_{(u,v)}}{\cancel{c(uv)}} \cancel{c(uv)}$
          $l(uv) \leftarrow l(uv) + l(uv) * \epsilon \frac{f_{(u,v)}}{c(uv)})$
          if $d'_{(u,v)} > 0$ **then**
            moreTraffic$\leftarrow true$
          end if
        end for
      end while
    end for
  end while
  for all $v \in V$ **do**
    $f^{v}_{(u,v)} \leftarrow f^{v}_{(u,v)}/log_{1+\epsilon}\frac{1+\epsilon}{\delta}$
  end for
end procedure

The proof that the algorithm efficiently computes a solution to the maximum concurrent flow with a precision ϵ is done in three steps:

- provide an upper bound for the dual optimum $D(l^*)$ at the end of the algorithm.
- provide a lower bound for the primal optimum λ^* at the end of the algorithm.
- show that the dual to primal ratio $\frac{D(l^*)}{\lambda^*} \leqslant 1 + \epsilon$ (in an optimal solution, the ratio $\frac{D(l^*)}{\lambda^*} = 1$. Moreover, by the weak duality property, $D(l) \geqslant \lambda$ for any $D(l)$ and λ. This way, we prove that the algorithm computes an ϵ-approximation of the optimal solution.)

i) To provide an upper bound on the resulted dual solution, we assume that $D(l^*) \geqslant 1$. If $D(l^*) < 1$, it is possible to scale down all the demands $d_{i,j}$ to ensure that $D(l^*) \geqslant 1$. In fact, having $D(l^*) < 1$, corresponds to the case when the network capacity is not enough to route all the demands. A possible scaling procedure will consist in routing the demands into the network by ignoring the link capacity constraints. This will allow to

detect the maximum violation of a link capacity: $r = \max(\frac{f_{uv}}{c(uv)} | uv \in E)$. Scaling all the flows down by r will ensure that $D(l^*) \geqslant 1$.

Let l_i be the length function l at the end of phase i (i.e., of the iteration i of the loop at line 4). The operation at line 16 increases $D(l)$ by at most $\epsilon \cdot a(l_i)$ during the phase i. In fact, during the iterations of the outer loop, the increment added by each execution of the line 16 ($\epsilon \cdot l(uv) \cdot f_{(u,v)}$) gradually computes the same sum as $\epsilon \cdot a(l_i)$ (i.e. $\epsilon \sum_{(i,j) \in V^2 : d_{i,j} > 0} l(SP_{i,j}) d_{i,j}$), but with lower link lengths. Considering that the lengths monotonically increase at line 17, at the end of the phase i, $a(l_i)$ will be bigger than the same sum computed with lower link lengths.

As a result, at the end of each phase i,

$$D(l_i) \leqslant D(l_{i-1}) + \epsilon \cdot a(l_i)$$

Consider the relation $\frac{D(l)}{a(l)} \geqslant D(l^*)$ which was proven true for any length function l in the chapter 3 (lemma 3.12). It allows us to substitute $a(l_i)$ in the previous equation and obtain $D(l_i) \leqslant D(l_{i-1}) + \epsilon \frac{D(l_i)}{D(l^*)}$. By re-organizing the therms, we get:

$$D(l_i) \leqslant \frac{D(l_{i-1})}{1 - \epsilon / D(l^*)}$$

The previous recursive relation, together with the fact that $D(l_0) \leftarrow m\delta$ at line 3 implies

$$\begin{aligned} D(l_i) \leqslant & \frac{m\delta}{(1 - \epsilon / D(l^*))^i} \\ = & \frac{m\delta}{1 - \epsilon / D(l^*)} (1 + \frac{\epsilon}{D(l^*) - \epsilon})^{i-1} \\ \leqslant & \frac{m\delta}{1 - \epsilon / D(l^*)} e^{\frac{\epsilon \cdot (i-1)}{D(l^*) - \epsilon}} \end{aligned}$$

Under the assumption that $D(l^*) \geqslant 1$:

$$D(l_i) \leqslant \frac{m\delta}{1 - \epsilon} e^{\frac{\epsilon \cdot (i-1)}{(1-\epsilon) \cdot D(l^*)}}$$

Taking into consideration that the algorithm finishes at the first iteration t in which $D(l) \geqslant 1$, we obtain $\frac{m\delta}{1-\epsilon} e^{\frac{\epsilon \cdot (t-1)}{(1-\epsilon) \cdot D(l^*)}} \geqslant 1$. As a result, at the end of the algorithm:

$$D(l^*) \leqslant \frac{\epsilon \cdot (t-1)}{(1-\epsilon) ln \frac{1-\epsilon}{m\delta}} \tag{5.3}$$

ii) To provide a lower bound on the value of the primal solution λ^*, we observe that each phase of the algorithm (except the last one) routes exactly $d_{src,dst}$ units of flow from each source node src to each destination node dst. As a result, if the algorithm finishes during the iteration t of the main loop, the algorithm has routed at least $(t-1) \cdot d_{src,dst}$ units of each flow.

Another observation is that, at the end of any phase i except the last one, the length $l_i(e)$ of each edge e is shorter than $1/c(e)$ (otherwise $D(l) = \sum_{e \in E} l(e)c(e) \geqslant 1$ and terminates

the algorithm). Furthermore, this property is also maintained during the last phase by the condition $D(l) < 1$ in the loop 8. As a result, when the algorithm ends, this property is violated at most once per edge by the line 17. The length of any edge e at the end of the algorithm is thus at most $l(e) < (1+\epsilon)/c(e)$. Knowing that the lengths are initialized to $\delta/c(e)$ at the start of the algorithm (line 2), each edge finishes with a length which is at most $\frac{(1+\epsilon)/c(e)}{\delta/c(e)} = \frac{1+\epsilon}{\delta}$ times its initial length.

Moreover, the operation at line 13 ensures that each $c(e)$ units of flow routed through an edge e increases its length by at least $1+\epsilon$. As a result, the capacity of any edge is violated by at most $log_{1+\epsilon}\frac{1+\epsilon}{\delta}$ times its capacity.

Finally, all these observations imply that the algorithm routes at least $t-1$ units of each commodity, while violating the capacity of any edge by at most $log_{1+\epsilon}\frac{1+\epsilon}{\delta}$ times its capacity. Scaling the flow down by $log_{1+\epsilon}\frac{1+\epsilon}{\delta}$ gives us a feasible flow for the primal LP. As a result, at the end of the algorithm, a lower bound for λ^* is given by:

$$\lambda^* \geqslant \frac{t-1}{log_{1+\epsilon}\frac{1+\epsilon}{\delta}} \tag{5.4}$$

iii) From equations 5.3 and 5.4, we obtain:

$$\begin{aligned}\frac{D(l)^*}{\lambda^*} \leqslant \frac{\frac{\epsilon(t-1)}{(1-\epsilon)ln\frac{1-\epsilon}{m\delta}}}{\frac{t-1}{log_{1+\epsilon}\frac{1+\epsilon}{\delta}}} = \frac{\epsilon \cdot log_{1+\epsilon}\frac{1+\epsilon}{\delta}}{(1-\epsilon)ln\frac{1-\epsilon}{m\delta}} = \frac{1}{1-\epsilon}\cdot\frac{\epsilon}{ln(1+\epsilon)}\cdot\frac{ln\frac{1+\epsilon}{\delta}}{ln\frac{1-\epsilon}{m\delta}} \leqslant \\ \leqslant \frac{1}{1-\epsilon}\cdot\frac{\epsilon}{\epsilon-\epsilon^2/2}\cdot\frac{ln\frac{1+\epsilon}{\delta}}{ln\frac{1-\epsilon}{m\delta}} \leqslant \frac{1}{(1-\epsilon)^2}\cdot\frac{ln\frac{1+\epsilon}{\delta}}{ln\frac{1-\epsilon}{m\delta}}\end{aligned} \tag{5.5}$$

By setting

$$\delta = \frac{1}{(1+\epsilon)^{\frac{1-\epsilon}{\epsilon}}}\cdot(\frac{1-\epsilon}{m})^{\frac{1}{\epsilon}}$$

the term $\frac{ln\frac{1+\epsilon}{\delta}}{ln\frac{1-\epsilon}{m\delta}}$ becomes equal to $\frac{1}{1-\epsilon}$. and $\frac{D(l^*)}{\lambda^*}$ becomes smaller than $(1-\epsilon)^{-3}$.

It is possible to pick ϵ in a way such that the ratio is less than $1+w$ for any $w > 0$. As a result, the presented algorithm computes an $1+\epsilon$ approximation of the maximum concurrent flow problem.

Discussions on choosing the precision

The asymptotic bound of the complexity proven by Karakostas [5] is $\tilde{O}(\epsilon^{-2}|E|^2)$, where the notation $\tilde{O}$ hides the logarithmic factor $log(|E|)log(|V|)$. In practice, the algorithm is very dependent on the chosen precision ϵ for more than one reason.

Evidently, ϵ defines the execution time. However, the quality of the solutions is much more affected than the speed. For example, setting a precision of $\epsilon = 0.3$ may be tempting, as the algorithm will provide a result almost ten times faster compared to a precision $\epsilon = 0.1$. However, the obtained dual to primal ratio $\frac{D(l^*)}{\lambda^*} = (1-03)^{-3} = 2.91545$ resulted in almost useless outputs during our preliminary tests.

Another, less obvious, dependence on ϵ is hidden in the initialization of the algorithm at line 2. δ is a very small constant which depends on ϵ and the number of links. Setting

the precision too low, like 0.01, not only drastically increases the execution time, but also induces numerical issues due to the limited precision of floating point numbers.

To avoid these two issues, we have empirically chosen the precision to be $\epsilon = 0.1$, which provides a dual to primal ratio of 1.37174. Even in this case, we had to use the "long double" type for our variables to avoid numerical issues on the biggest of our evaluated networks: Generated200.

5.2.3 Step 2: transmitting the constraints to the SDN switches

In this section, we discuss the communication overhead of the proposed solution. After execution of the centralized optimization phase, the SDN controller sends separately to each switch the flow he must achieve. At most, the control traffic overhead grows asymptotically with the number of links in the network: $O(|E|)$. However, in practice, only a fraction of the links will be used.

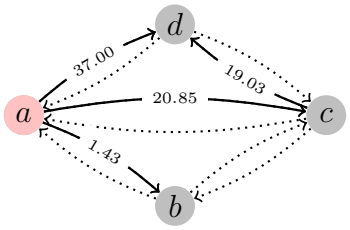


Figure 5.7: Flow originating from the node a

Fig. 5.7 retakes the same example that was presented in the previous section (5.2.1) and illustrates the flow originating from the node a. In this case, the information transmitted by the network controller to the node a will consist of the list $((ab, 1.43), (cd, 19.03), (ad, 37.00), (ac, 20.85))$.

We evaluate the control traffic overhead in the *Generated*200 network. Even if we assume the worst case scenario when each of the nodes uses all the links, the total network traffic overhead is $O(|V| \cdot |E|)$. We also consider an efficient encoding of data when transferring through the network, were only $32bytes$ are used to uniquely identify a link and represent the amount of flow passing through it. In this case, the SDN controller will have to send into the network a total of $200 * 1148 * 32bytes \simeq 8Mbytes$ of control traffic. We consider that this is a reasonable small control traffic overhead for a network with 200 nodes.

In practice, nodes do not use all the links for forwarding. For example, in 5.7, the nodes use less than half of network links. The communication overhead will be smaller than the estimated quantity.

5.2.4 Step 3: computing paths by SDN switches

The goal of this step is to compute the explicit paths used for data forwarding. The information transmitted by the SDN controller gives a good indication of the routes which must be preferred by the SDN switch but does not explicitly indicate these routes. An additional computation must be done to build the forwarding paths and update the

forwarding database of the network device. For example, the node a from Fig. 5.7 will construct the following routing:

- The demand from a to b will be integrally sent over the path (ab)
- The demand from a to c will be integrally sent over the path (ac)
- The demand from a to d will be split over two paths:
  - 66% ($\frac{37}{37+19.03}$) over the path (ad)
  - 34% ($\frac{19.03}{37+19.03}$) over the path (ac, cd)

This section presents the algorithms proposed for this purpose. These algorithms rely on two hypothesis: i) use of a source routing forwarding protocol, and ii) the origin-destination demands have a high aggregation level.

The first assumption is respected by the use of the SPRING protocol. It enables the ingress nodes to locally change the network routes without having to update the forwarding tables of midpoint devices. The second is true in backbone networks, making it is possible to split the demands at a TCP/UDP flow granularity: out of thousands of TCP/UDP connections which make the aggregated origin-destination flows, some connections will be forwarded over one path, while the others will take another path. Without a large aggregation of flows, the efficiency of the proposed solution may decrease, because it will be difficult to split the demands into multiple paths.

Transform the flow into a topologically sorted DAG

The per-source flow computed at the SDN controller is not necessarily a minimum-cost flow. When the ingress node receives the flow from the SDN controller, it will start by deleting the cycles from the flow graph of the source node src ($\mathcal{G}_{src}$) to obtain a minimum-cost flow. The corresponding algorithm is trivial: it starts by transforming the flow in the network topology graph into the corresponding residual flow. Afterward, it executes the classical cycle canceling algorithm [113]. As the resulting directed graph does not have any cycles, we topologically sort this DAG using the unmodified, classical, sorting algorithm [116].

At this stage, the SDN switch has all the pre-requisites needed to quickly compute the forwarding paths for the network flows and assign the traffic to each route using the procedure that we present in the next section.

Computing the routes for flows

To define the explicit paths in the network, we propose a procedure that considers that the DAG has a topological order.

For each node $v \in V$, a data structure is used to keep track of the explicit paths from the root v_{root} towards this node. This data structure is a set of pairs, defined as $\mathcal{R}_{v_{root},v} := \{(P_1, r_1), (P_2, r_2), ...)\}$. Each pair represents a path P_i and the associated ratio $0 \leqslant r_i \leqslant 1$, which corresponds to the ratio of demand $d_{v_{root},v}$ which must be sent over the path P_i. The sum of all ratios is $\sum_i r_i = 1$. Alg. 8 describes the proposed solution.

Algorithm 8 Compute the explicit routes for the flow from v_{root} towards all nodes $* \in V$

```
1: procedure computeRoutesInDAG(D(V, E_D), v_root ∈ V, d_{v_root,*}, sortedV)
2:     initialize ratio_uv ← 0, ∀uv ∈ E_D
3:     for all v ∈ V do
4:         totalInTraf ← 0
5:         for all (u,v) ∈ E_D do
6:             totalInTraf ← totalInTraf + f_uv
7:         end for
8:         for all (u,v) ∈ E_D do
9:             ratio_uv ← f_uv/totalInTraf
10:        end for
11:    end for
12:    R_{v_root,v_root} ← ((),1)
13:    for all v ∈ sortedV[1 :] do
14:        R_{v_root,v} ← ∅
15:        for all (u,v) ∈ E_D do
16:            X ← {(R.P + uv, R.r · ratio_uv)|∀R ∈ R_{v_root,u}}
17:            R_{v_root,v} ← R_{v_root,v} ∪ X
18:        end for
19:    end for
20: end procedure
```

Before constructing the explicit paths, we do a preliminary computation: for each node $v \in V$, we compute the fraction of flow entering the node through each of its inbound links (2 - 11).

Afterward, the DAG is traversed following the topological order of nodes, which was computed in the previous section and stored in the list $sortedV$. Thanks to this order, when a node v is encountered, the paths used to reach its parents are already computed. At this moment, the set of paths allowing to reach v can be computed as the union of all the paths towards the parent nodes, extended by the corresponding inbound links from the parent (lines 16-17). The split ratio of flows over this path is the product of the split ratio to reach the parent multiplied by the ratio of inbound flow on the link. It is a number between 0 and 1. By construction, the computed split ration per path represents the fraction of the demand $d_{v_{root},v}$ which must be sent over this path.

Once the paths and the corresponding ratios computed, the problem of assigning the sub-flows of the origin-destination demand $d_{v_{root},v}$ into each of the computed paths corresponds to the "Generalized assignment problem"[117], which is NP-hard. Nevertheless, under the hypothesis of a high aggregation level, we assume that the integer constraint on the flows can be relaxed. As a result, the sub-flows of $d_{v_{root},v}$ can be packed in these paths using a simple heuristics and obtain a solution which does not deviate much from the optimal fractional packing computed by the algorithm 8. An example of such heuristic may be the "first fit decreasing" algorithm. Without a large aggregation of flows, the efficiency of the proposed solution may decrease, because the efficiency of the "first fit decreasing" can be limited. The description of this heuristic is out of the scope of this work.

Avoiding long routes

The previous procedure has the drawback that it can generate a lot of long routes for transporting average ratios of the origin-destination flow. This applies in particular to big networks. However, it is possible to put a limit on the split level to avoid routes which are too long and provide little benefit. For this purpose, at line 16, whenever a small split ratio is detected, we should not add a new path to the set of paths. Instead, the corresponding fraction of demand $d_{v_{root},v}$ should be redistributed over existing paths. In big networks, we recommend to use this approximation and prune all the paths with a small split ratio. Due to lack of time, we did not implement this optimization in the programs used to generate the results from the next section.

5.2.5 Evaluation

The algorithms were implemented in C++ and evaluated using the same methodology as presented in the previous chapter (section 4.3).

Maximum link utilization

Fig. 5.8 shows how well the algorithm keeps the network out of congestion and the maximum recorded link utilization. Once again, the x axes represent the time normalized to the moment just before a congestion in the all-on network. The y axes represent the utilization of the most loaded link in the network. For example, in the Italian network, our load balancing (LB) technique allowed to keep the network out of congestion 4 times longer (x axes) than the shortest path routing. In the meantime, the maximum link utilization was always close to the one provided by the LP solver (CPLEX). Our solution closely follows the optimum solution on all the networks. It is even difficult to distinguish between the result of our algorithms and the one provided by CPLEX. Our algorithm becomes unable to keep up with the optimal solution when very close to congestion. However, the deviation from the optimum is small.

Fig. 5.9 shows the state of network links in the Italian network just before congestion. With LB, the network can route four times more traffic thanks to efficiently distributing the load. The shortest path routing performs very poorly in this case. The node R19 (top middle of the figure) is the destination of a vast number of elephant flows. Most of these flows pass through the same link. The figure on the right illustrates the Italian network using the LB routing technique. LB efficiently distributed the load over the four inbound links of the node R19. A small under-optimality can still be seen, as the two links at the left of R19 are less utilized compared to the two ones on the right. This is exactly the reason behind the sub-optimality which is visible in Fig. 5.8.

Fig. 5.10 provides additional insights on the load of all the links in the eight evaluated networks. It separately shows the utilization of each network link. In fact, maximum value on the y axes on Fig. 5.10 corresponds to the value shown on the previous Fig. 5.8. The load is more uniformly distributed over the links. For example, in the case of the CORONET network (Fig. 5.10g), SPF routing results in a couple of highly loaded links and a lot of links with utilization close to zero. In the meantime, with LB, there are no links with utilization close to zero.

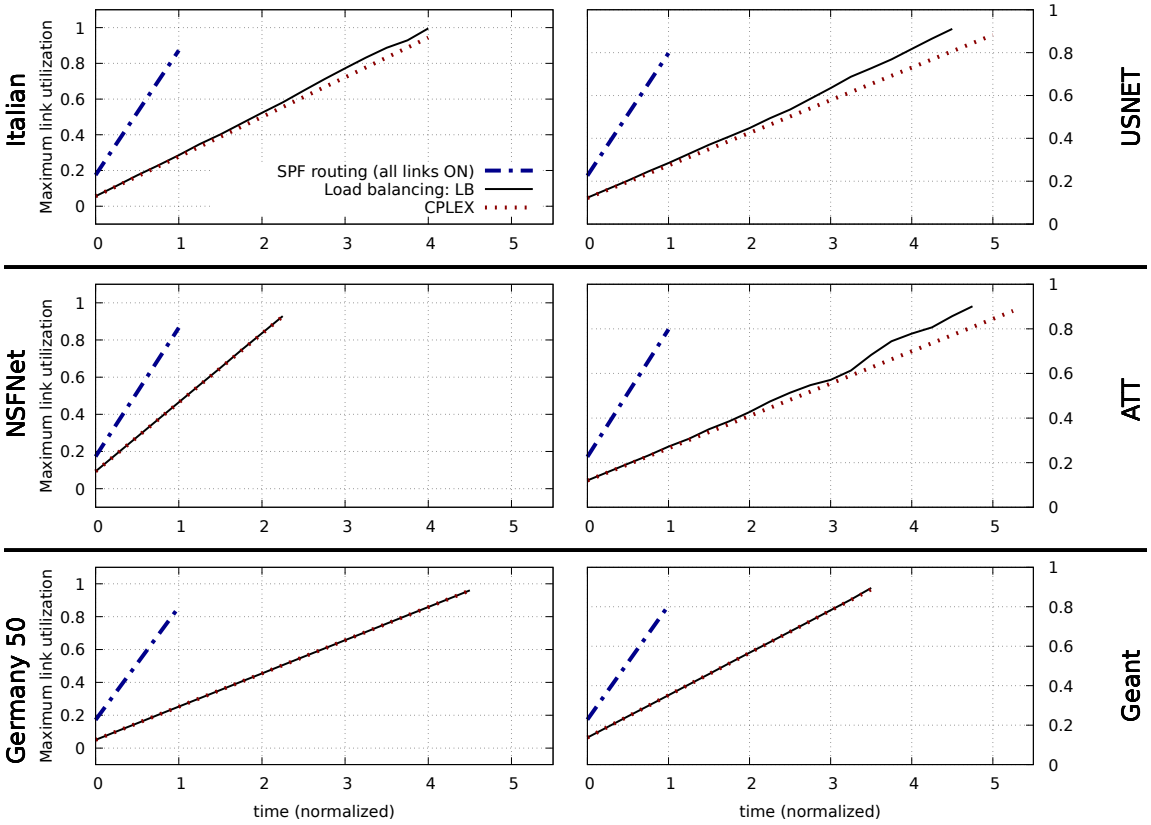


Figure 5.8: Utilization of the most loaded link under continuous growth of network traffic. x axes are normalized to the moment of congestion with classical shortest path routing.

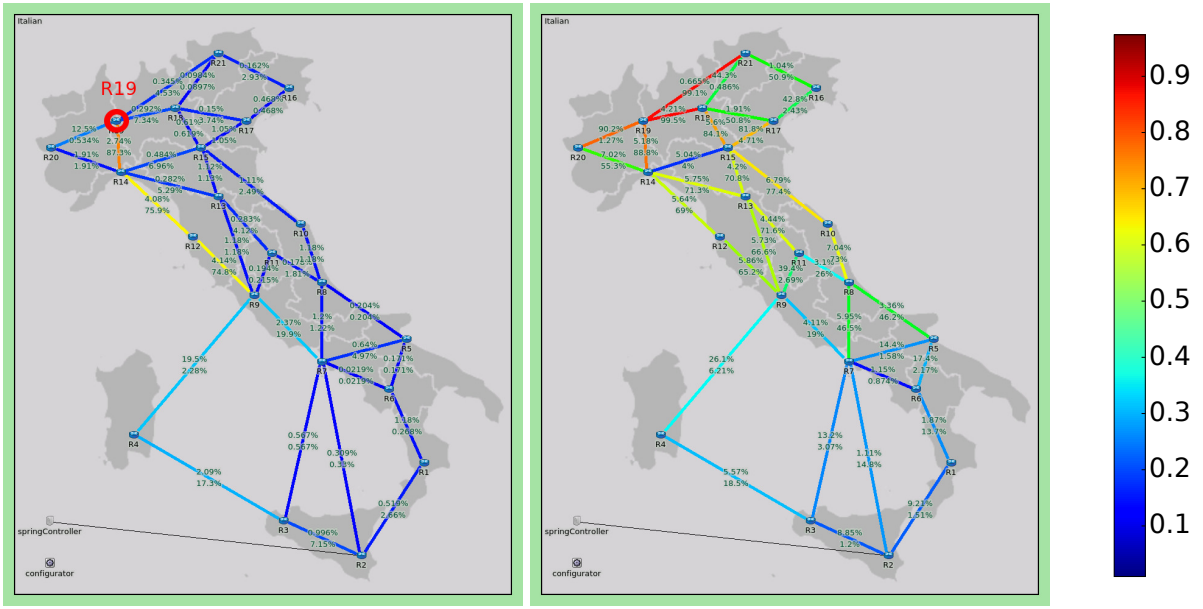
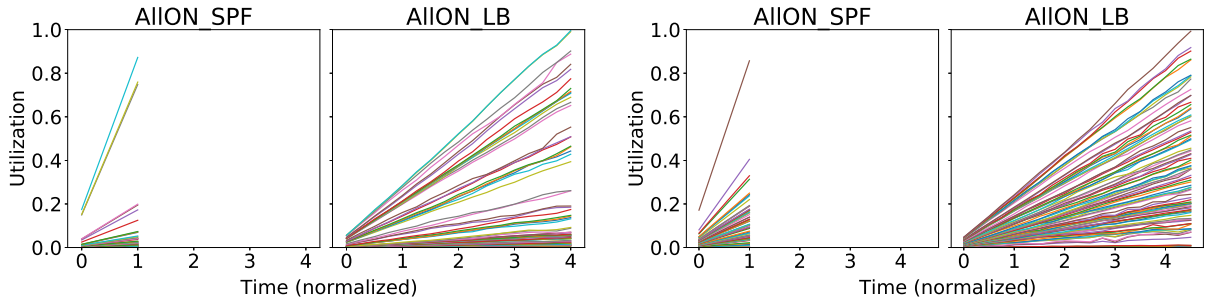
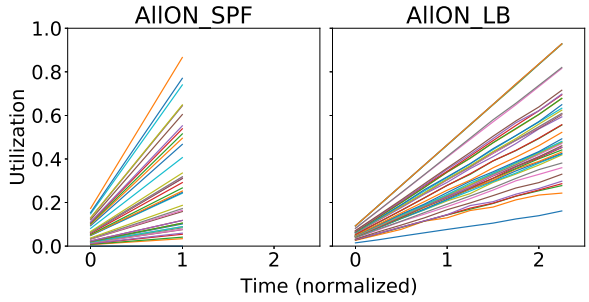


Figure 5.9: Italian network with the highest traffic before congestion. At the left: shortest path routing at $t = 1$ on figure 5.8. At the right: LB with 4 times more traffic than the shortest path routing ($t = 4$ on figure 5.8).

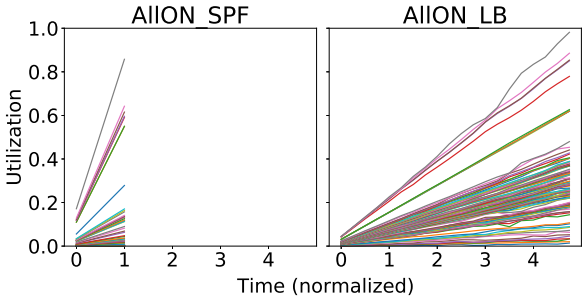


(a) Italian

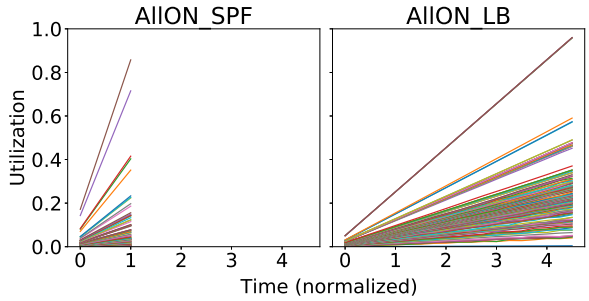
(b) USNET



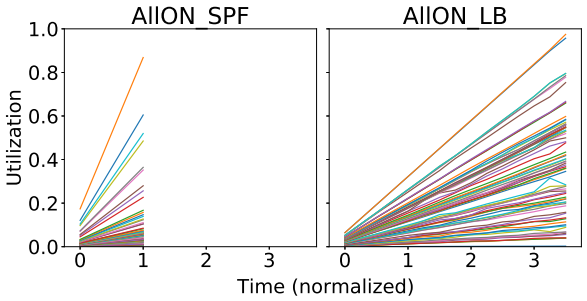
(c) NSFNet



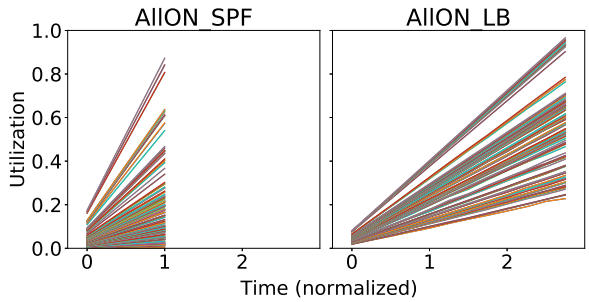
(d) AT&T



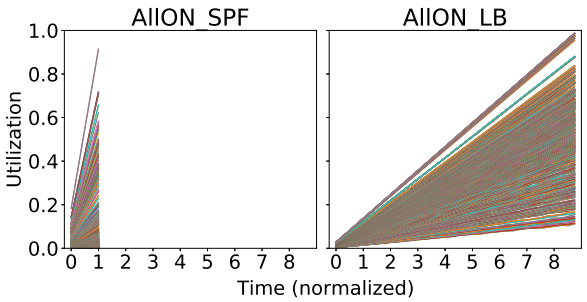
(e) Germany50



(f) Geant



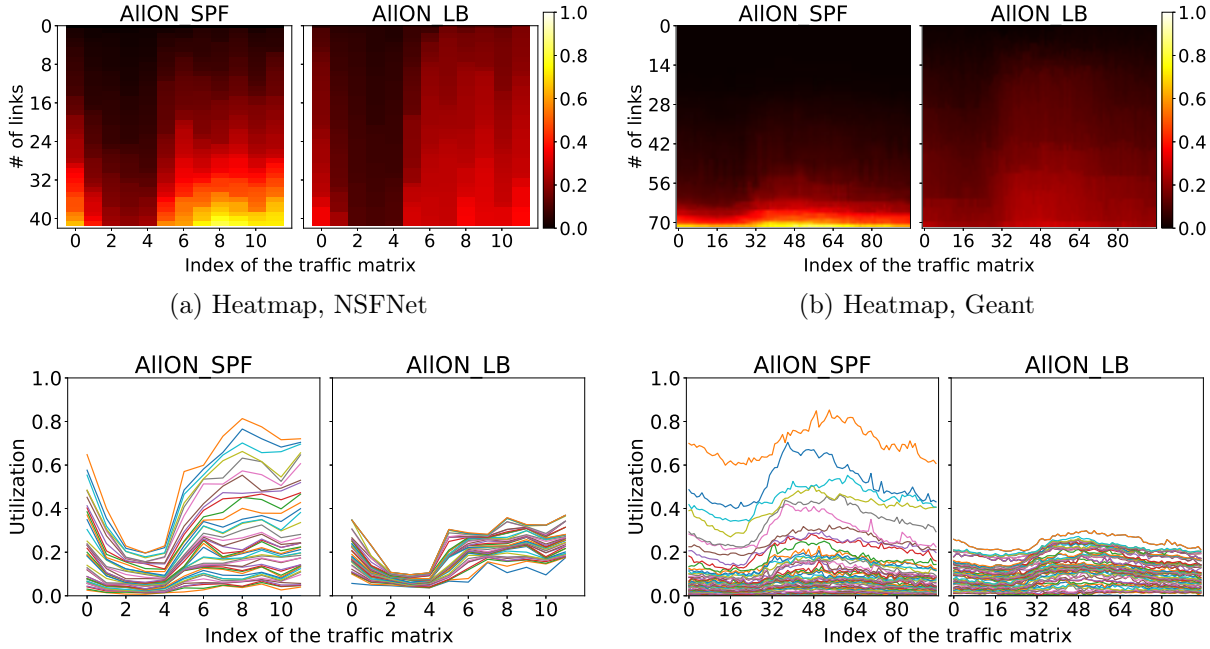
(g) CORONET



(h) Generated200

Figure 5.10: The utilization of each link in the evaluated networks

Fig. 5.11 shows the heatmap and the link load in the NSFNet and Geant networks with daily traffic, i.e. the real traffic matrices introduced in section 4.3.1. As previously, the load is effectively and uniformly distributed over the links of the network. The results on other networks do not provide any additional useful information.



(a) Heatmap, NSFNet

(b) Heatmap, Geant

(c) Link utilization, NSFNet

(d) Link utilization, Geant

Figure 5.11: The heat-map and the link utilization in the NSFNet and Geant Networks with the daily traffic

All these results and other can be obtained by repeating the simulations. Please follow the instructions from the end of the first chapter of this thesis.

Computational time

Tab. 5.1 resumes the computational time taken by the algorithms. In the case of LB, the numbers include both the time taken by the centralized optimization phase presented in Section 5.2.2 and by the explicit computation of the forwarding paths presented in section 5.2.4. The precision used for the Karakosta's algorithm is $\epsilon = 0.1$.

For completeness, we also provide the computational time taken by our first attempt to load balance (CF) presented in Section 5.1. We conclude that LB delivers better results while being, in most cases, faster than CF. This is particularly the case on big networks (Generated200, CORONET).

On small networks, CPLEX may seem to be an excellent alternative to the proposed heuristics, because it provides a proven optimal solution and its computational time is comparable to the heuristics. However, the output of the LP optimization is not a routing which can be directly applied to the network. Additional post-treatment must be done to build the routes for the flows. Alternatively, the linear programming (LP) model solved by the CPLEX solver could be updated to generate a viable routing directly. However,

Network	CF	LB	CPLEX
NSFNet	225ms	166ms	458ms
Italian	383ms	546ms	(1s)1126ms
Geant	829ms	248ms	948ms
USNET	544ms	737ms	(2s)2539ms
ATT	823ms	794ms	(2s)2830ms
Germany50	(2s)2385ms	(2s)2024ms	(8s)8333ms
CORONET	(24s)24358ms	(3s)3688ms	(*)-
Generated200	(8m)537820ms	(1m)107526ms	(*)-

*CPLEX was not able to initialize the model

Table 5.1: Maximum computational time

this change will lead to a further increase of the computational time, making it less appealing than LB.

Path length increase

Shifting the traffic out of the congestion zone comes at the cost of probably taking longer routes and thus increasing the end-to-end delay of the network traffic. In this section, we evaluate how much the path lengths increased during the evaluated scenarios compared to the shortest path routing. For this purpose, we use the parameter *path-stretch* introduced in the previous chapter.

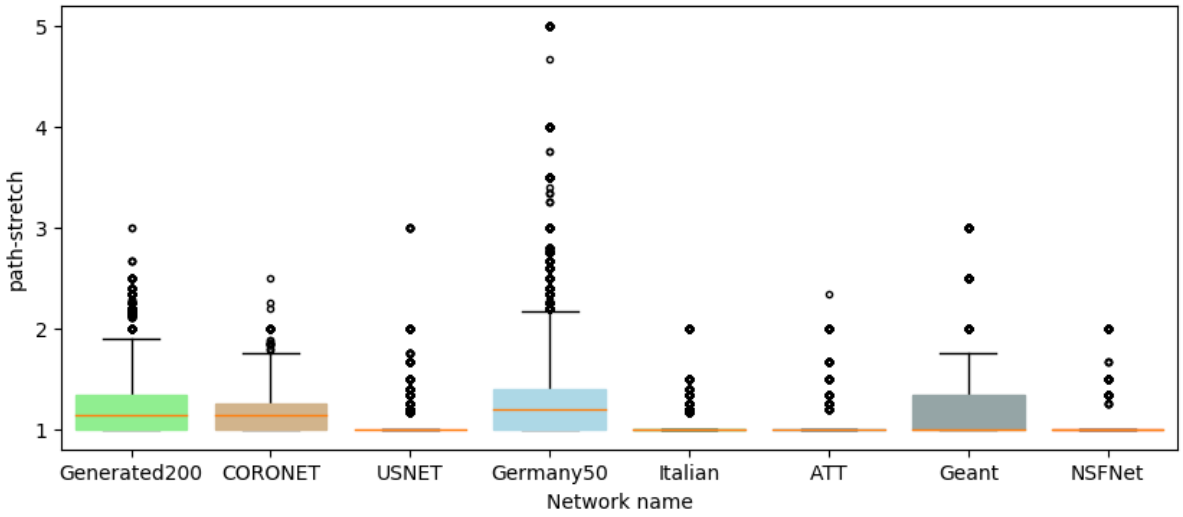


Figure 5.12: *path-stretch* in each of the evaluated networks

Fig. 5.12 summarizes the results for all the network topologies. To construct the boxplots we proceeded as follows: for an evaluated traffic matrix we recorded the *path-stretch* of every origin-destination flow. That operation gave us $NB := |V| \cdot (|V| - 1)$ *path-stretch* values. The procedure was repeated for each evaluated traffic matrix. For instance, if 12 traffic matrices are evaluated, the boxplot shows the distribution of all the $12 \cdot NB$ recorded values.

The results show that only a few flows are routed over long routes. The path length of each origin-destination flow rarely exceeds the double of the shortest path (SP) route. The median, represented by the red line in the middle of the box-plots, is close to one. This allows us to conclude that the majority of paths remain almost as short as with SP routing.

5.3 Conclusion

In this chapter, we proposed two solutions for load balancing the network load. The first one relies on a function that changes the cost of a link depending on its utilization. The preliminary validation revealed that minimal variations in the shape of the cost function could lead to a bad performance of the algorithm. In the search for a solution to this problem, we proposed an alternative load balancing technique.

Our second attempt provides a complete solution which tries to keep a low overhead of centralized management. For this purpose, the construction of the solution is split into two parts. First, a centralized optimization phase uses a state-of-the-art approximate maximum concurrent flow algorithm to find the near optimal distribution of the flow from each source. Similarly to the first load balancing technique, the central optimization phase of our second solution relies on assigning costs to network links. During the execution of the algorithm, the cost of each link increase when demand is routed through the link. Moreover, the algorithm also uses shortest path computations and relies on these costs to prioritize the paths passing through the less loaded links. However, thanks to making tiny steps during the execution of the algorithm, it is possible to achieve a testable near-optimal solution.

Unfortunately, the algorithm delivers a solution which splits the traffic into multiple paths. Our contribution is in providing a way to effectively apply the previously computed routing into the network switches and change the paths of network flows. We avoid using explicit routes and keep the communication overhead reasonable low even on the biggest of the evaluated networks. This is achieved by deporting part of the computation to a network device. The second part of our solutions consists in locally computing the explicit paths for the network flows on ingress devices, which also allows updating the forwarding database locally. This solution would not be possible without the use of a source routing protocol for data forwarding. We propose to use SPRING for this purpose. It enables the ingress nodes to locally change the network routes without having to update the forwarding tables of midpoint devices.

The presented technique allows achieving a global, network-wise, traffic optimization. It was evaluated against a linear programming model and was shown to compute a near-optimal routing in all the evaluated networks. The utilization of the most loaded link is always kept close to the value provided by the LP model. Moreover, the lengths of network paths are kept close to the lengths with shortest path routing. At the same time, the computational times do not surpass a couple of seconds in any of the real backbone networks used for evaluation.

Chapter 6

Consume less and better: evaluation and impact on transported protocols

In the previous two chapters, we presented two methods to increase the energy efficiency of backbone networks. The first one, which we called SegmenT Routing based Energy Efficient Traffic Engineering (STREETE), acts by putting unused resources to sleep whenever the network is at low utilization. In the second, we proposed a means to achieve a near-optimal distribution of the load in the network while keeping a low computational and communication overhead. The latter solution can be used separately, to push the limits of the network and avoid premature updates to higher data-rates that consume more power. Furthermore, this load balancing technique can also be incorporated into our STREETE framework to obtain a solution that acts both by minimizing the number of active resources and by optimizing the traffic within these active resources.

We start this chapter by evaluating the combination of the previously proposed techniques: STREETE combined with load balancing (LB). The proposed solution, which we call STREETE-LB, can compete with linear programming models both regarding turned-*off* links and in its ability to keep the network out of congestion.

Unfortunately, achieving such performance comes at the cost of frequent changes of network paths. Not only the algorithm has to reroute the flows each time a link is turned *on* or *off*, but even a small variation in network conditions may induce a reorganization of the paths due to the inclusion of the load balancing technique. To estimate the impact of such frequent route changes, we conduct a detailed evaluation on our network testbed. In particular, we evaluate how the change in the network delay due to moving a flow between two paths impacts the congestion control mechanism of TCP.

6.1 Consume less and better: STREETE-LB

In this section, we evaluate the combination of previously proposed techniques that we call STREETE-LB: STREETE with load balancing technique LB.

The combination of the two algorithms is straightforward. The STREETE algorithm (Alg. 2) presented in Chapter 4, (Section 4.2.3) remains unchanged. Only the logic behind the steps which simulate the routing of the traffic into the network must be modified. Unfortunately, this change is non-negligible because it drastically impacts the computational complexity on the critical path of the STREETE framework.

To reduce the influence of this computation, we use two different precisions for the Karakosta's algorithm. A low precision of $\epsilon = 0.2$ is used during each iteration of the STREETE framework, i.e. when a link extinction/ignition is simulated. A higher precision, of $\epsilon = 0.1$ is used to compute the final routing, which will be sent to the SDN switches. Performing less accurate computation during the iterations of STREETE allows for drastically improving the computational time because the complexity of the algorithm increases quadratically with the precision.

We use the same MILP model as in Chapter 4 (Sec. 4.3.2) for this evaluation.

6.1.1 Computational time

Taking into consideration that we introduced a complex computation into the critical path of the STREETE framework, the first parameter evaluated during our evaluations is the computational time. Tab. 6.1 provides this information for each of the analyzed network topologies.

Network	**STREETE-LB**	**CPLEX**
NSFNet	216ms	(1s)1827ms
Italian	496ms	(2s)2075ms
Geant	104ms	(3s)3491ms
USNET	637ms	(33s)33716ms
ATT	(2s)2009ms	(21s)21488ms
Germany50	(14s)14478ms	(2h*)7203220ms
CORONET	(15s)15384ms	(**)-
Generated200	(20m)1204510ms	(**)-

*CPLEX didn't find the optimal solution in the maximum allocated time (2h)
**CPLEX wasn't able to initialize the model

Table 6.1: Maximum computational time

In our opinion, STREETE-LB is not fast enough in the big, 200-node network. In the worst case, it took a maximum of 20 minutes to compute a solution. Nevertheless, we believe that the 15 seconds on the 75-node CORONET network topology are promising. In fact, anticipating some conclusions detailed later, we affirm that very frequent reroutings may be dangerous for the network stability. More details on this matter are given in the second part of this chapter.

STREETE-LB is thus capable of computing a solution on any of the evaluated real backbone networks in adequate time.

6.1.2 Performance at high network load

The biggest problem of the STREETE framework presented in Chapter 4 was the bad performance under high network load. Fig. 6.1 shows the behavior of the STREETE-LB mixed solution in this case. We do the same evaluation as previously where the network load increases continuously until congestion is detected.

The mixed solution can avoid congestion for almost the same time as CPLEX. Some notable exceptions can be seen in the cases of AT&T and USNET networks. We analyzed

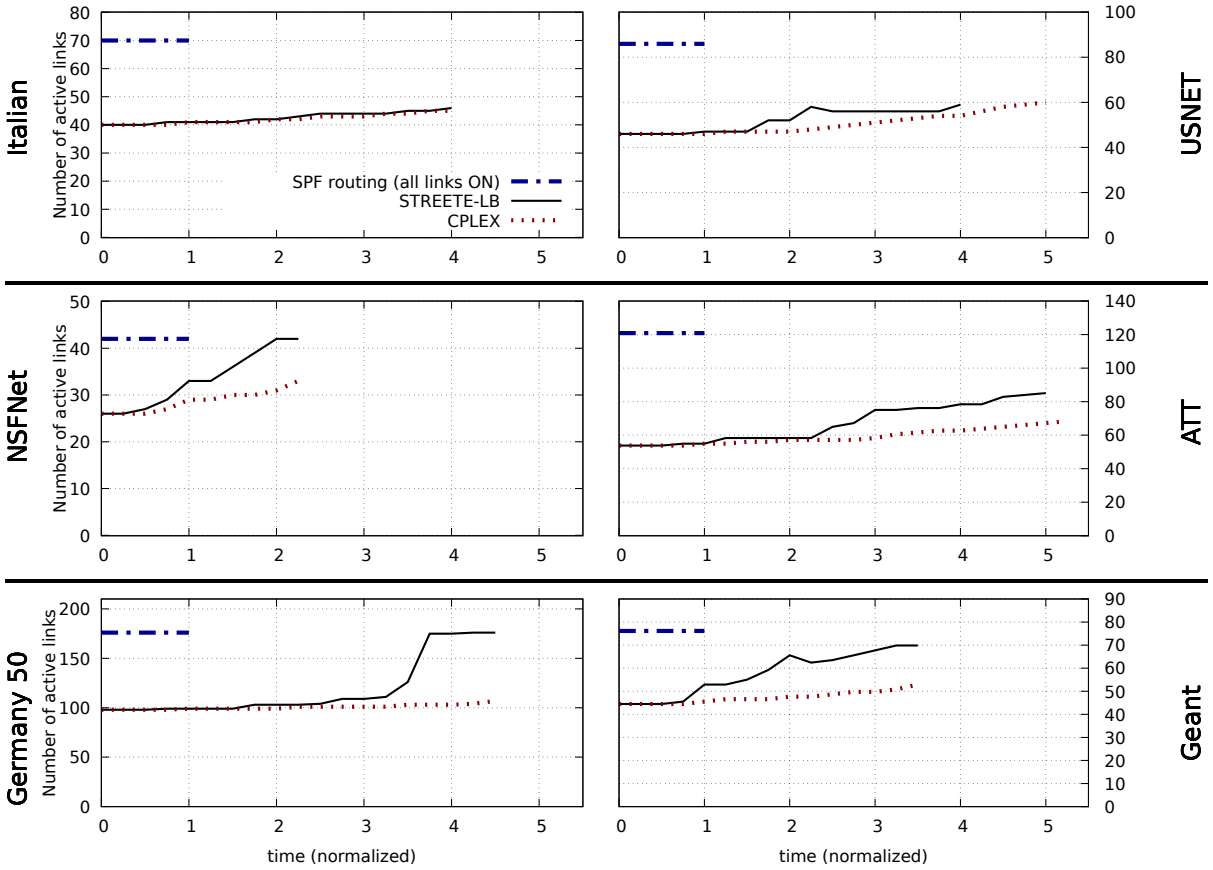


Figure 6.1: Maximum link utilization in the analyzed networks using the STREETE-LB mixed solution

these scenarios in detail and concluded that it is due to the approximate nature of the Karakosta's algorithm and the low precision of 0.2 used by STREETE. Increasing the precision of the algorithm leads to better results at the cost of longer computations. It is a trade-off to be considered. It may be worth increasing the accuracy on small networks if sub-second computational time is not needed.

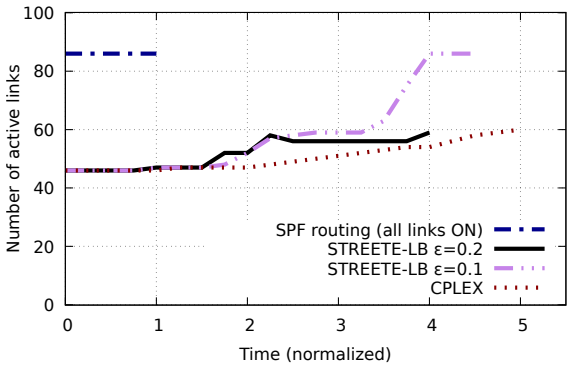


Figure 6.2: Influence of the precision ϵ on the number of active links and on the congestion avoidance. USNET network

For example, Fig. 6.2 illustrates the performance on the USNET network with a precision of $\epsilon = 0.1$. The additional precision enables improving the quality of the results at a drastic increase in the computational complexity. The maximum time taken by STREETE-LB in this case was of approximately 18 seconds. For comparison, in the previous section, we showed that STREETE-LB takes approximately 0.7s with a precision $\epsilon = 0.2$.

From Fig. 6.1 and 6.2, it may look like STREETE-LB behaves much worse than CPLEX in terms of active links. We observe links being switched *on* in bursts. For example, the biggest such burst is seen on the Germany50 network at around $t = 3.5$. Analyzing this behavior revealed that it is due to the much more conservative nature of STREETE compared to the CPLEX model used for validation. In particular, our solution uses a double threshold (60% and 80%) to decide if it is worth turning links *off* / *on* and to avoid link flapping [1].

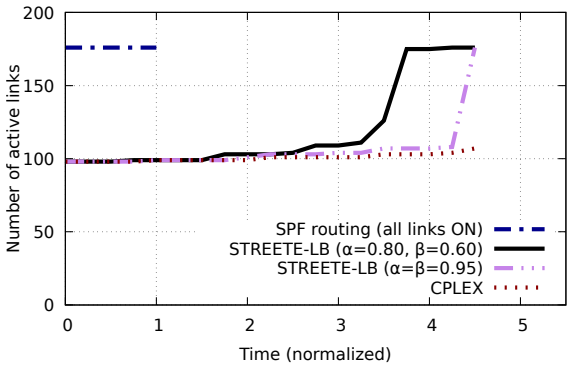


Figure 6.3: Influence of the thresholds α and β on the number of active links in the Germany50 network

Fig. 6.3 shows that setting the two thresholds close to the link congestion makes STREETE-LB behave similarly to CPLEX for a longer period. However, using a double threshold allows to improve the stability of the framework and avoid frequent change in the paths of network flows.

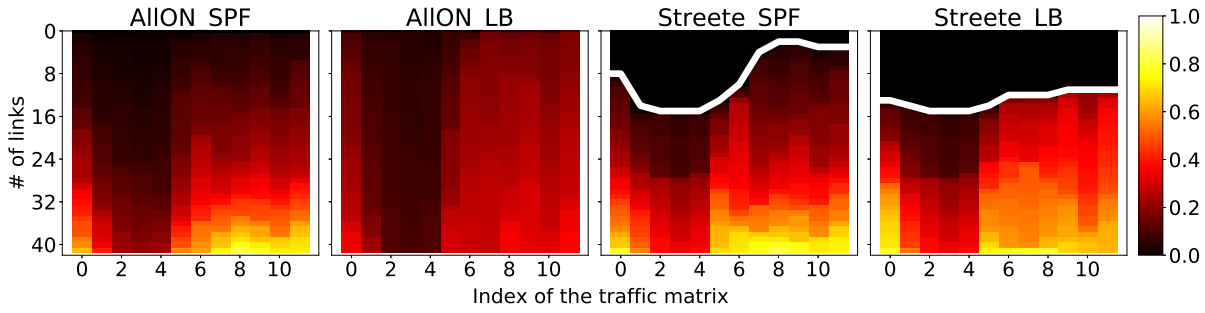
6.1.3 Performance with realistic traffic matrices

STREETE alone showed real energy savings when tested with realistic traffic matrices. Nevertheless, we may expect to have better savings if the framework includes a load balancing technique. Surprisingly for us, the evaluation did not confirm this intuition.

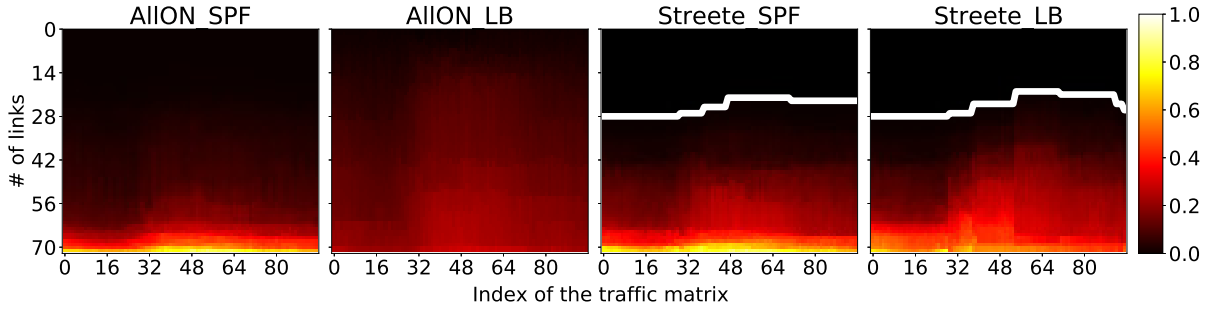
Fig. 6.4 shows the results in 3 representative cases. First, is represented the distribution of load in the NSFNet network. In this case, STREETE-LB behaves better than its shortest-path based equivalent (presented in chapter 4) because the overall network load is initially high compared to other networks. In this case, load balancing techniques came in handy and allowed to turn *off* more links. This is visible especially in the second half of the day (between $tm = 6$ and $tm = 12$)

A second case is seen on the Geant network. STREETE-LB behaves worse than its shortest path based equivalent. More links are kept active in the network. Nevertheless,

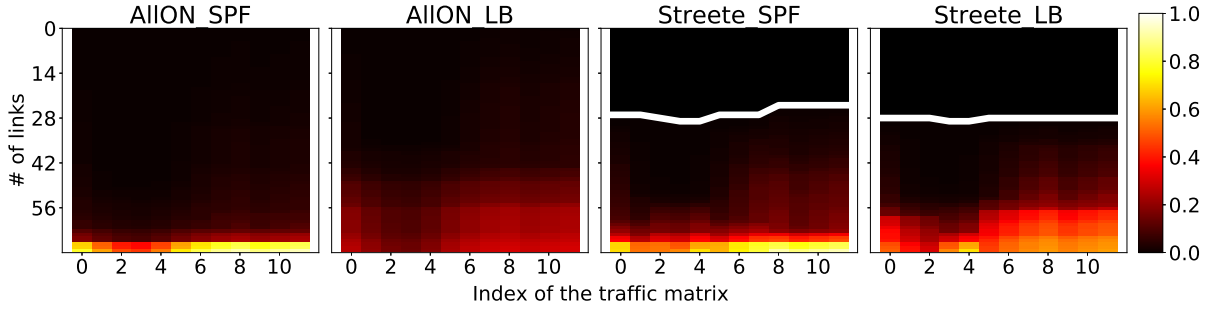
[1] frequent change of the state of links (*off* / *on*) due to small variations in network traffic



(a) NSFNet



(b) Geant



(c) Italian

Figure 6.4: The heat-map of network links with real traffic matrices

this is not necessarily a bad behavior. The additional links were ignited to reduce the load on a couple of highly utilized links, which were close to congestion. The shortest path routing was unable to handle this case and avoid network congestion efficiently.

The latest case, illustrated on the Italian network, shows a distinct advantage of STREETE-LB. In this case, the algorithm was able to both slightly reduce the number of active links and to keep the network at relatively low utilization.

6.2 Impact of frequent route changes on TCP flows

Throughout this manuscript, we have presented our framework for increasing the energy efficiency of IPoWDM backbone networks by employing frequent network re-optimizations. The paths of network flows constantly change for this purpose. However, we completely ignored that rerouting flows can have a severe impact on the transported protocols. In particular, a drop in throughput can occur if TCP is the transport protocol.

During the validation of the STREETE framework on our network testbed (presented in section 4.4), we observed uncontrolled oscillations of link states: *on*<->*off*. These oscillations occurred in rare cases when: i) TCP was used, and ii) the delay of the network paths differed in two or more orders of magnitude.

The reason of this oscillation was that TCP reduced its throughput when shifted to a new route. STREETE quickly detected this reduction of network traffic as an opportunity to switch some links *off* and save energy. Nevertheless, the drop in the speed of the TCP flow was only temporary. TCP quickly recovered from the drop and forced STREETE to turn-*on* links to provide enough capacity for the additional demand. Turning the link *on* has shifted flows towards new network paths, which produced a new drop in network bandwidth.

In fact, the use of double thresholds by STREETE as triggers to turn links *on* and *off* helped to avoid this case to a certain extent. This is the reason why the oscillation only occurred in extreme cases: when the delays of paths differed in orders of magnitude.

In this section, we use our networking testbed to perform an experimental analysis of this case. In particular, we evaluate how the default implementation of the TCP protocol in the Linux kernel reacts to the fact of frequently rerouting the network flows. Our goal is to find a rerouting frequency which is safe and avoids the case above.

6.2.1 Background

Rerouting and congestion control

The TCP's congestion control algorithm, defined in the RFC 5681, is an important, and one of the most complex, features of modern TCP implementations. This algorithm tries to split the network capacity fairly among all the flows traversing it. Under such algorithm, the sender keeps a congestion window that is dynamically modified depending on the network conditions. The source cannot send more data than what fits in this window during a Round-Trip Time (RTT). The maximum instantaneous bandwidth, B, of the TCP connection parameters, is thus limited by the size of the congestion window: $B = W \cdot MSS/RTT$, where W is the size of the congestion window in segments, MSS is the maximum segment size in bytes.

TCP was initially designed for standard IP routing assuming that all packets follow the same route towards a destination. Packet reordering and route changes were considered rare. However, in an SDN network, the controller may frequently shift a TCP flow to an alternative path to optimize the overall network throughput. The route change can impact the throughput of a TCP flow in two different ways:

- When the new route has higher RTT, the sender increases the size of the congestion window to maintain the same sending rate. For example, if the RTT doubles, the

bandwidth will be halved and will gradually increase with the growth of the TCP congestion window; a direct application of the equation $B = W \cdot MSS/RTT$.

- The receiver sees packets arriving out of order if the new route has a lower RTT. TCP's congestion control algorithms assume the worst-case scenario and view this as an indication of packet loss due to congestion. Figure 6.5 illustrates the problem considering a sample backbone network. The link ab becomes available for transmission and hence packets 2 and 3 take a route shorter than packet 1. The packets arrive out of order at the receiver who uses duplicate ACKs to notify the sender about a problem. The sender reduces the size of the sending window to avoid congestion, hence decreasing the transmission rate.

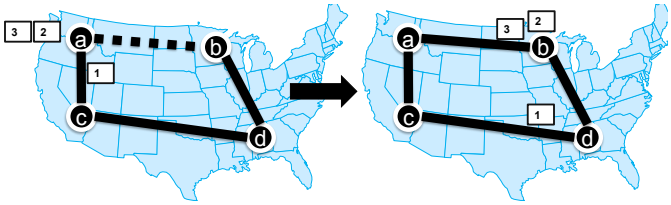


Figure 6.5: Rerouting towards a lower RTT ⇒ packet reordering.

Given that a large part of network traffic is often carried by a small number of large, long-lived flows [118], rerouting these flows may significantly reduce the overall network throughput.

Congestion Control Algorithms

The analysis focuses on the impact of route changes on the default TCP congestion-control mechanism in the Linux kernel. As of writing, the default algorithm used in all the tested devices is Cubic, with a fallback to Reno, including devices using kernel versions 3.2, 3.13, 4.4 and 4.7.

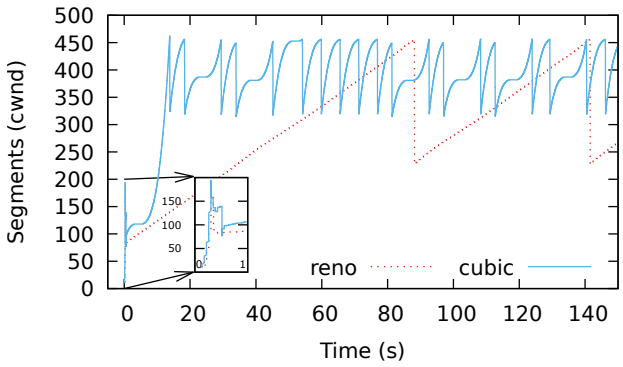


Figure 6.6: TCP congestion control; RTT = 50ms; 100mbps link.

Figure 6.6 shows how the congestion window of a single flow evolves when using the Reno and Cubic congestion control algorithms.

RENO The standard TCP congestion control mechanism is Reno, which is still the fallback algorithm in the Linux kernel. This algorithm is straightforward and comprises two phases: 1) the slow start phase, where the window size doubles each RTT; and 2) the congestion control phase where the window size increases by one each RTT or is divided by two under packet loss.

The slow start phase allows for a fast increase in the window size at the beginning of communication. It corresponds to the spike at time 0 in Figure 6.6. In this work, we ignore the slow start.

Although TCP Reno behaves well under small Bandwidth x Delay Product (BDP), it severely underutilizes the channel on long fat networks – *i.e.* networks with high data rate and high delay – because it linearly grows the windows by one every RTT when recovering from packet loss. In Figure 6.6, Reno takes more than a minute to grow the congestion window and fill a 100Mbps link with an RTT of 50ms.

CUBIC The Cubic TCP algorithm [119] aims to avoid the shortcomings of Reno and achieve high data rates in networks with large BDP. At the same time, when Cubic detects a network with small BDP, it tries to mimic Reno's behavior emulated by a mathematical model. This happens, particularly in local high-speed, low-delay, networks.

As its name suggests, the algorithm grows the congestion window by using a cubic function of the elapsed time from the last loss event $y = (\Delta t)^3$. More precisely, using a shifted and scaled version $y = 0.4 \cdot (\Delta t - RTT - K)^3 + W_{max}$. In this equation, Δt is the time passed from the last loss event, W_{max} is the size of the congestion window just before the loss event. $K = (W_{max}/2)^{1/3}$ is a parameter that depends on W_{max}.

Cubic is also less aggressive in reducing the congestion window at a loss event. Compared to Reno, which halves the window, Cubic reduces it by 20%.

6.2.2 Estimating the Behaviour of TCP Cubic under Route Changes

Recovering from a Loss

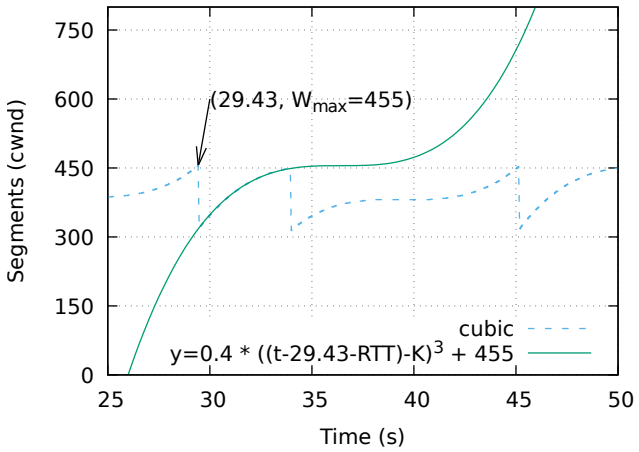


Figure 6.7: Function used by the Cubic algorithm; RTT = 50ms.

Figure 6.7 illustrates the cubic function when applied to part of the trace from Figure 6.6. At time $t = 29.43$, a loss is detected when reaching the bottleneck capacity of a network path. Cubic hence reduces the congestion window to 80% of W_{max}.

The growth function, y, used to increase the congestion window depends mainly on W_{max}, which makes it easy to estimate the time taken by TCP cubic to restore its congestion window after a loss. To do that, we ignore the insignificant dependency on RTT in the Cubic function and solve $W_{max} = y = 0.4 \cdot (\Delta t - K)^3 + W_{max}$. We obtain $\Delta t = K = (W_{max}/2)^{1/3}$.

Moreover, the RTT and the bottleneck bandwidth are used to determine the maximum window size. Table 6.2 estimates the size of the congestion window needed to transmit at 100Mbps with 1448 byte segments. It also uses the equation $\Delta t = K$ to assess the time taken by the Cubic algorithm to grow the window back to W_{max} after a loss.

RTT (ms)	10	50	180	500
W_{max} (segment)	87	431	1550	4316.3*
Time to recover (s)	3.5	6	9.2	13*

* the default maximum size of the kernel buffer is too small to allow full speed communications with $RTT = 500ms$. In this case, the transmission speed is limited by the size of the kernel buffer; approximately 2100 segments.

Table 6.2: Estimated W_{max} and time needed to recover after a loss, 100Mbps bottleneck link.

This estimation is valid only if no further loss is detected. Figure 6.7 contains a counter-example where TCP would reach W_{max} at approximately $t = 36s$, but a loss happened at ∼34s and forced the congestion control algorithm to reduce the size of the congestion window.

While particularly interested in the time needed to recover from a loss, we note that route changes may generate false loss events when the TCP flow shifts towards a shorter route. At such time, the size of the congestion window decreases, and cubic tries to grow it back. Intuitively, the bandwidth of a TCP flow may drop significantly if the SDN controller attempts a second re-optimization in the meantime.

Relevant Linux Implementation Details

The TCP implementation in the Linux kernel incorporates a lot of standard and non-standard optimizations [107]. One of the non-standard features, which is very important for our work, is the possibility to undo an adjustment to the congestion window. This implementation tries to distinguish between packet reordering and loss by using the "Timestamp" TCP option. When the sender detects that a past loss event was a false positive due to packet reordering, the algorithm reverts the window size to the value used before the reduction. As a result, packet reordering may have much less impact compared to standard TCP.

Multiple TCP Flows on a Bottleneck Link

As mentioned earlier, TCP cannot send more data than the size of the congestion window per RTT. As a result, when the sender's window W is smaller than W_{max}, TCP spends part of the time waiting for acknowledgments without sending any data.

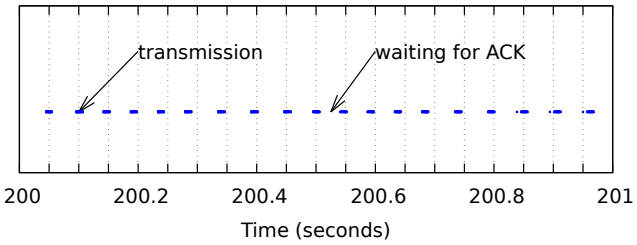


Figure 6.8: Alternated sending and waiting phases.

Figure 6.8 illustrates this case, summarizing data collected using the *tcpdump* application to capture the packets of a TCP flow over a second. The figure shows the times when packets were captured. Approximately 2/3 of the time, there was no packet passing through the network because the congestion window is too small (only 150 segments). Under such conditions, a congestion window of 431 segments was needed to fill the bottleneck link.

If a route change happens when a flow is waiting for acknowledgments, there is no reordering problem since no packet is sent. Hence, the probability that a TCP destination will receive packets out of order increases as W approaches W_{max}. Respectively, when TCP backs off and lowers its sending rate by reducing W, it also reduces the probability to be impacted by the rerouting.

If a large number of TCP flows share the same bottleneck link, the flows spend a lot of time waiting for the ACKs. Hence, rerouting a large bunch of TCP flows at the same time may have less impact on the overall network bandwidth than rerouting a single TCP flow.

This TCP behavior may change in the future. The benefit of smoothing the transmission over time by employing traffic pacing was shown to have a beneficial effect on network performance [120] and state-of-art congestion control algorithms, like BBR[121], use this technique to avoid the bufferbloat problem.

6.2.3 Experimental Evaluation

Experimental Setup

We consider backbone, highly over-provisioned, networks where resource capacity is not a limiting factor. The transmission speed is either bounded by a congested link in the aggregation, or by a very high propagation delay, *i.e.* by the size of the maximum allowed congestion window. No congestion ever occurs on the backbone links.

We construct the topology presented in Figure 6.9. The examined scenario consists of multiple TCP sources sending data to remote clients. The transfers pass through a backbone network employing traffic engineering: the two parallel 10Gbps links. The traffic is routed through two alternative paths within the backbone network, emulated

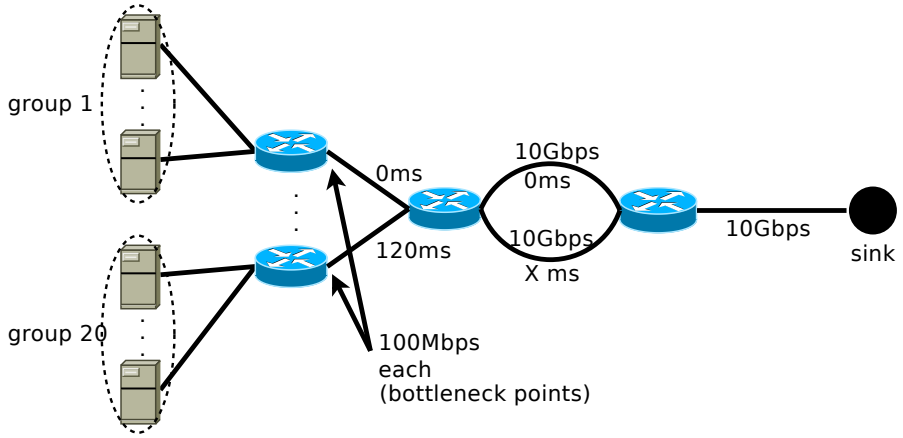


Figure 6.9: Testbed overview

by these two links. The paths have different delays. One of the backbone paths has no additional delay (0ms), while the second path induces an additional delay of X ms to simulate a longer route. We use *tc-netem* for this purpose. Each of the two paths (links) has sufficient capacity to transfer the flows. The bottleneck capacity is outside the backbone network. Congestion never happens on these two links.

The sources are aggregated in groups. The flows from 2 different groups never share a bottleneck point. However, two flows of the same group compete for the bandwidth allocated to the group. Each group has a different path RTT, uniformly distributed in the interval [0ms, 120ms], which adds to the delay of the backbone paths. The goal is to recreate the scenario where flows with different end-to-end delay coexist in the backbone network.

Concerning the physical infrastructure: nodes in the topology correspond to Ubuntu 14.04 Linux servers running Open vSwitch[2] v2.4 (manually compiled for Ubuntu 14.04). The network is controlled by the ONOS SDN controller (v1.7) via an out-of-band control connection. We wrote an ONOS SDN application to reroute the flows between the two backbone links.

The backbone links use 10G optical Ethernet interconnects. The connections to the TCP sources use 1G Ethernet ports. In the physical topology, both 10G and 1G network interconnects pass through Dell Ethernet switches. The point-to-point links are emulated using VLANs on these hardware switches. This permits to reconfigure the network topology remotely. Unfortunately, the queuing disciplines on these switches are opaque and cannot be fine tuned. To avoid perturbations, we use *tc-tbf* (Token Bucket Filter) to limit the speed of each group to 100Mbps. In such a way, the data passing through a hardware switch is always much below the link capacity. Moreover, this allows avoiding perturbations due to the Ethernet flow control mechanisms: PAUSE frame. We use a drop-tail bottleneck queue having the size 0.1 * BDP.

[2] `http://openvswitch.org`

Rerouting Independent Flows

This section analyses the impact of rerouting multiple independent TCP flows and how the total backbone throughput changes as a consequence of rerouting.

We generate 20 unsynchronised flows, one flow per group, that start at random times within a 30-second interval. We run the transfer for 400 seconds to allow TCP flows to converge to a steady state. After that, we reroute the traffic every 200 seconds and measure how the aggregated throughput changes at during rerouting.

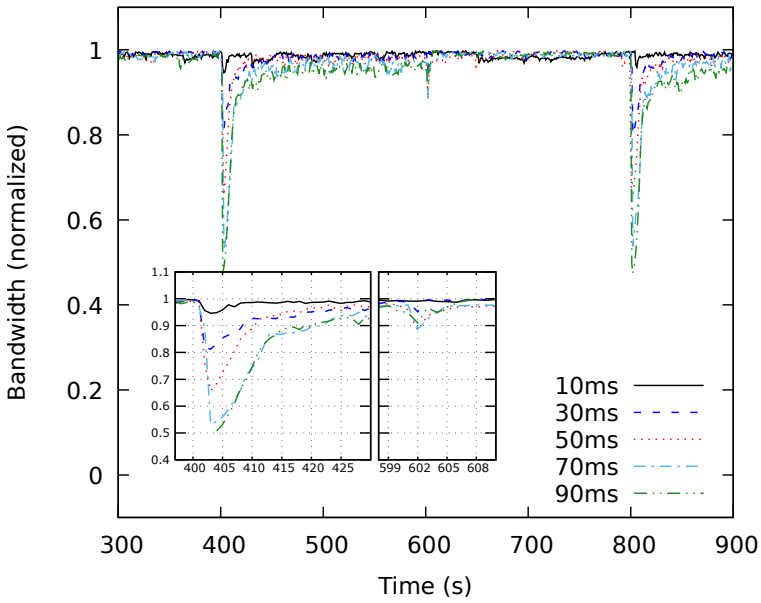


Figure 6.10: Bandwidth drop when rerouting 20 independent cubic flows

Figure 6.10 shows how the aggregated bandwidth varies when the second backbone path induces an additional delay of 10ms, 30ms, 50ms, 70ms or 90ms.

At $t = 600$, the flows are rerouted towards the path with lower delay. Respectively, at $t = 400$ and $t = 800$, the flows are rerouted towards the longer path. This experiment confirms our expectations, that *rerouting the flows impacts their throughput both when moving towards a longer route and when moving towards a shorter one.*

Nevertheless, packet reordering, which happens at $t = 600$, has a less pronounced impact than expected. Theoretically, Cubic would reduce the transmission window to 80% of its size when a loss is detected. Then it would gradually increase the window back to the size before the loss. However, the Linux TCP can detect packet reordering in some cases. In particular, this happens when the difference of delay between two routes is small compared to the delay of the fastest route. Hence, a slight variation of delay, like 10ms, has a marginal impact on the total throughput of the flows. The next section gives more insights on this optimization.

Rerouting towards the link with higher RTT substantially reduces the aggregate bandwidth of the 20 flows ($t = 400$ and $t = 800$). The larger the delay difference between the two routes, the bigger the drop. In this case, rerouting is transparent to the congestion control algorithm. The drop comes from the unexpected increase of the delay. As a consequence, the size of the transmission window must be increased to allow transmission at

the same speed.

Rerouting Flows Sharing a Bottleneck Point

To evaluate the impact of sharing a bottleneck point where flows inside a group compete for bandwidth, we fix the RTT of the second path to $50ms$ and vary the number of flows per group. With one flow per group, 20 flows traverse the network whereas, under 9 flows per group, there is a total of 180 flows.

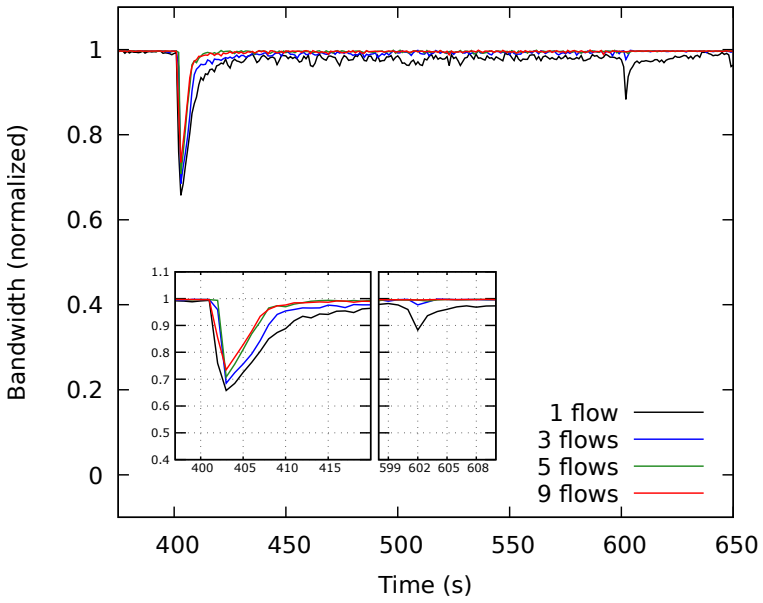


Figure 6.11: Impact of the number of flows in each of 20 groups.

Figure 6.11 shows how competing flows impact the recovering speed after rerouting. The bandwidth drop caused by rerouting towards a shorter route becomes quickly insignificant ($t = 600s$) in contrast to rerouting towards a longer path ($t = 400s$). With the increase in the number of flows sharing a bottleneck link, each of these flows gets a smaller proportion of the total bottleneck bandwidth. The flows hence spend more time waiting for acknowledgments than actually sending data. At rerouting, fewer flows see their packets arrive out-of-order. Moreover, the flows that do not experience out-of-order arrivals can increase their sending rate by using the capacity released by flows that have just reduced the sending window. As a consequence, the total aggregated throughput on the backbone links is almost not impacted.

When rerouting flows towards a longer path, the aggregate bandwidth recovered slightly quicker when the number of competing flows per group increases. It is due to nonlinear growth function used by Cubic. Having more flows increases the probability that some of them will be in the "aggressive growth" phase of the cubic function.

One TCP Flow under Frequent Rerouting

The previous sections showed that packet reordering has negligible impact compared to the increase of RTT. In this section, we assess whether very frequent route changes may

reorder enough packets to perturb the congestion control algorithm even if the delay difference between the two considered routes is small compared to the RTT.

While focusing on analyzing a single TCP flow, we use *iperf* to transfer 2Gbytes of data and measure its mean bandwidth (on testbed from Figure 6.9). A big enough transfer size was chosen to reduce the impact of the slow start phase on the mean throughput. Considering a maximum bottleneck throughput of 100Mbps, each transfer takes approximately 3 minutes if transferred at full speed. The flow is frequently rerouted between the two paths. We tested with periods of rerouting going from once every 15 seconds, to as low as once every 0.1s.

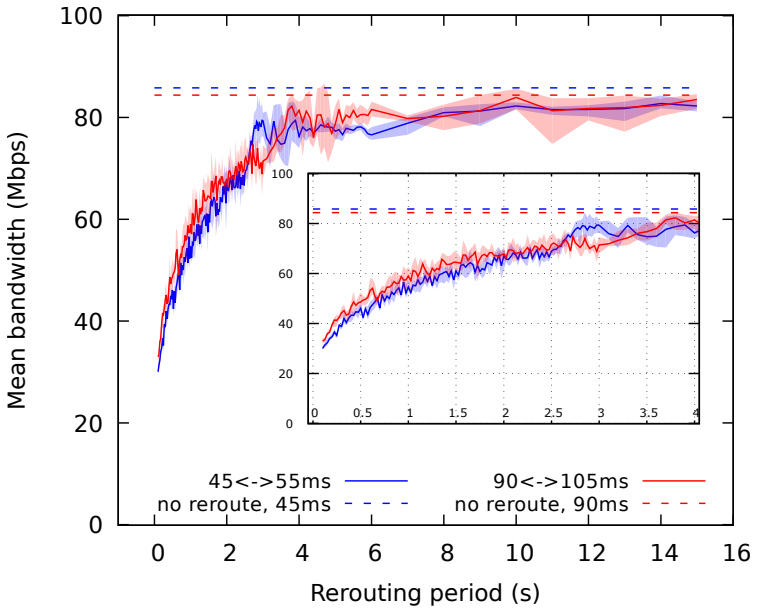


Figure 6.12: Throughput of a TCP flow. Periodic rerouting between 2 routes with different RTT; Low RTT.

Figure 6.12 summarises the results. Considering five experiment runs, each data point in the graph corresponds to mean throughput, *i.e.* the amount of transferred data divided by the total transmission time. The error intervals show the absolute minimum and maximum values recorded. The baseline without rerouting shows the average throughput of a TCP flow when it always passes through the same path. The baseline is the mean of 20 experiments. The inner plot zooms in on the region with frequent route changes.

Frequent route changes have a large impact on the performance of the congestion control algorithm. When the traffic is rerouted once every 0.1s, the throughput is as low as 35% of the throughput without rerouting. We can also observe that with a route change every 2 seconds, the throughput is around 85% compared to the throughput without rerouting. Although from a practical point of view we believe that these frequencies of rerouting are extreme (and not very realistic), they were included here to evaluate the limits on traffic rerouting. To re-optimise the network at such frequencies, a lot of control messages must be transmitted between the SDN controller and the switches, creating a flood of control traffic. Moreover, the network optimization problems are usually computationally intensive and take the time to find a good solution.

It is worth noting that the biggest drop in throughput is observed when a second rerouting happens before Cubic recovers from the first rerouting. Previously, we gave an estimation of the recovery time in Table 6.2.

Figure 6.13 shows two interesting results for higher RTT:

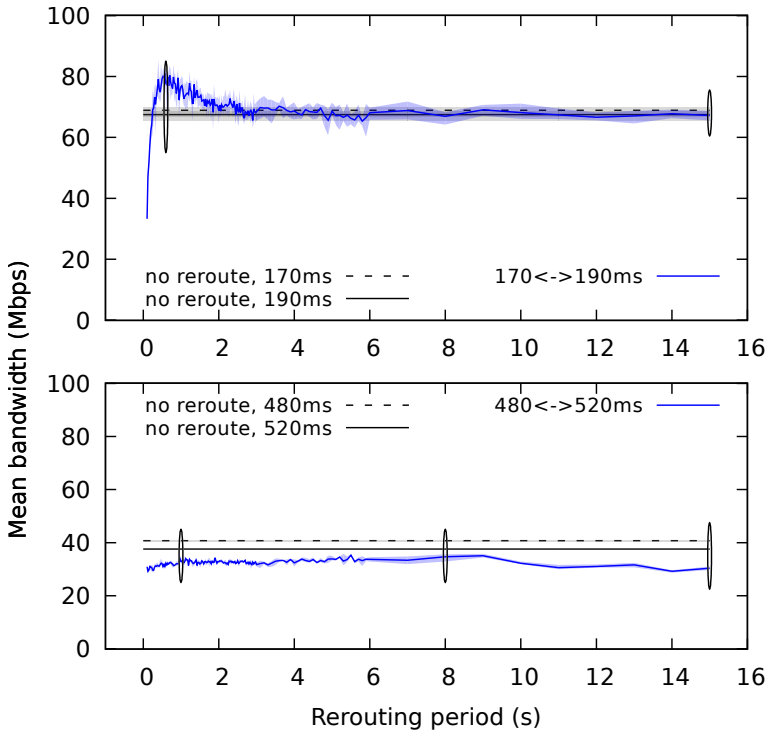


Figure 6.13: Throughput of a TCP flow. Periodic rerouting between 2 routes with different RTT; High RTT.

1. Frequent rerouting has a beneficial effect under an RTT of 170, observed at $period = 0.6s$ (the first ellipse) where the mean bandwidth of a flow rerouted every 0.6s is higher than the average bandwidth of a flow that is always transmitted over the same path.

2. Under RTT=480, the bandwidth of the flows is also less impacted when rerouted frequently.

These results are consistent among the multiple transfers. The error intervals are very tight and difficult to see in the figure. To explain this behavior, we analyze in detail the evolution of the congestion window and packets traversing the network. We choose to concentrate on the points marked with ellipses in Figure 6.13.

Inspecting the beneficial effect at RTT=170 Figure 6.14 gives more insights on this case. The dashed line shows the evolution of the congestion window without rerouting. At times $t = 23$, $t = 43$, etc, the congestion control algorithm does not behave as

expected. Instead of reducing the congestion window by 20% as dictated by the Cubic algorithm, the drop is much higher. It is because Cubic claims bandwidth too aggressively. Hence a huge number of packets are lost when reaching the bottleneck capacity.

Moreover, Linux TCP does not employ a *multiplicative decrease* technique of reducing the congestion window. Instead, it reduces it additively at every second duplicate ACK received. These two causes together imply a bad performance at RTT=170.

Rerouting the flow in the meantime perturbs the Cubic protocol and avoids this unwanted behavior. It is due to the optimization in the Linux kernel which detects out-of-order deliveries thus reducing the impact of the rerouting. The evolution of the congestion window in the case when the route changes every 0.6s can be seen on the continuous, blue, line of Figure 6.14. For example, at $t = 8$, a duplicate ACK is detected as being due to reordering and not to a loss, the congestion window is restored to the size before the reduction.

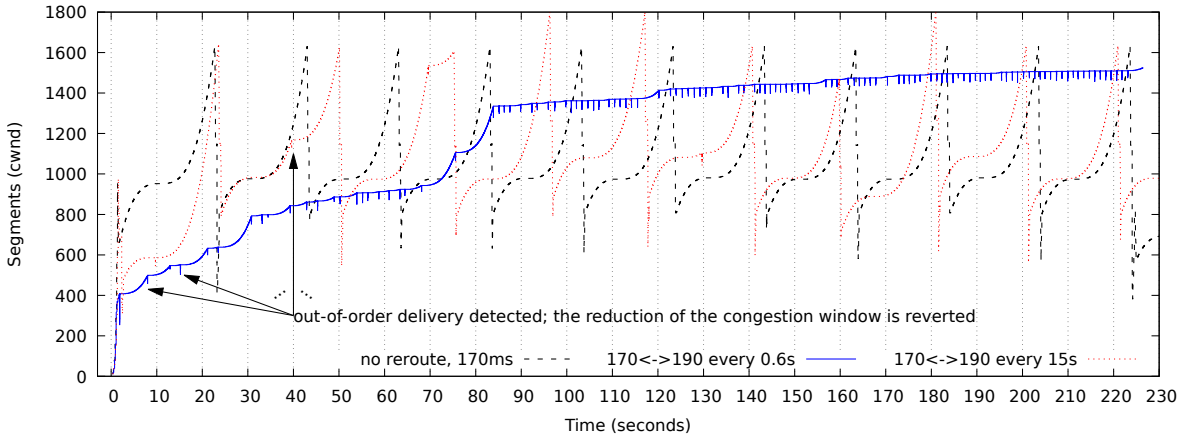


Figure 6.14: Evolution of the congestion window at $RTT = 170ms$.

We mentioned in an earlier section that the TCP sender might "miss" a rerouting because the sender alternates between sending packets and waiting for ACKs. If the route change happens when the sender waits for ACKs, no packets will arrive out-of-order. The blue line illustrates this case. At small window size, between $t = 0$ and $t = 30$, the sender spends a lot of time waiting. As a result, the probability to be impacted by a rerouting is small. The bigger the congestion window, the higher the probability to be impacted by a rerouting. Starting with $t = 120$, an equilibrium is created. The sender struggles to grow the window any further: any rerouting has a high probability of reordering packets.

Inspecting the unexpected behavior at RTT=480 Figure 6.15, which analyses the evolution of the congestion windows, shows that the window remains constant if the routes do not change. This behavior is expected as the transmission is limited by the size of the congestion window, which in turn is constrained by the size of the in-kernel buffer reserved for the TCP connection.

If we compare the evolution of congestion window in case of rerouting, we observe that the Linux Kernel's optimization, which detects out-of-order packets, is more susceptible to work as expected at frequent route changes. This unexpected behavior is consistent among multiple transfers.

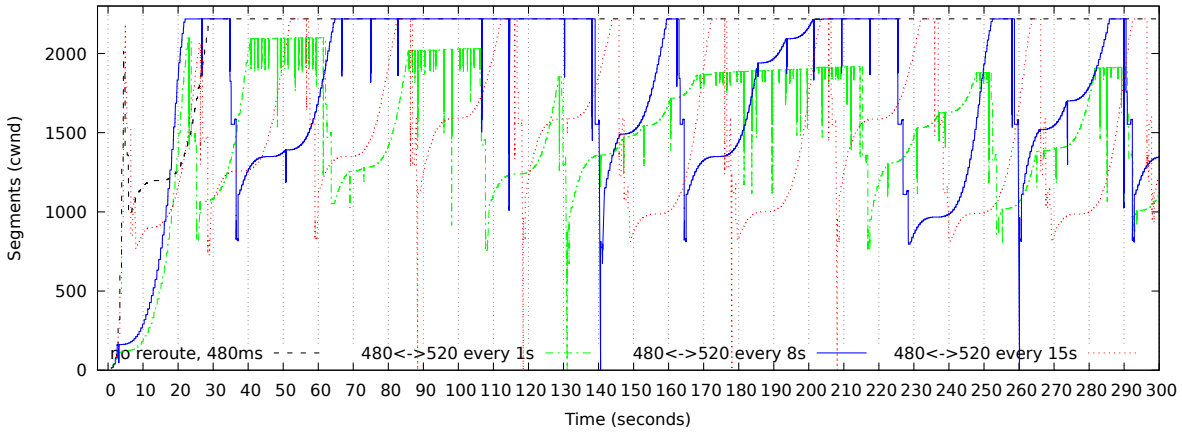


Figure 6.15: Evolution of the congestion window at $RTT = 480ms$.

However, rerouting once every 15 second allows the flow to converge towards the maximum window size. Any bigger rerouting period (20s, 30s, etc.) will have less impact on the average throughput. The worst case scenario occurs at a period of 14s.

6.2.4 Conclusion

In this chapter, we started by evaluating the combination of the techniques presented in the previous chapter, which we call STREETE-LB. We showed that it could achieve near-optimal solutions both regarding active links and in its capacity to keep the network out of congestion. We managed to avoid the drastic increase of the computational complexity by using a variable precision during the execution of the algorithms.

After that, we analyzed the impact of frequent route changes on the Cubic TCP flows. We performed an extensive analysis of the behavior of cubic TCP on a real testbed to find a safe rerouting frequency. Our study confirmed that at a low aggregation level when a limited number of TCP flows share a path, the rerouting reduces the speed of the TCP flow. Nevertheless, the reason for this behavior is opposite to the reason that is usually evoked by the networking community.

It is considered that packet reordering will make TCP falsely assume a congestion in the network and reduce its speed. In reality, the engineers implemented a set of non-standard optimizations that detect this case and reduce the impact of reordering to a minimum. Moreover, the impact of reordering decreases as the aggregation level increases. This is because the probability of affecting a particular TCP flow decreases.

At the same time, we recorded a bandwidth drop when the flows are rerouted towards a longer path. The increase of delay artificially limits the speed of TCP until the congestion window "catches up" with the new network delay. The time needed by TCP to converge to a steady state depends on the difference of network delay between the two routes and the congestion control mechanism used by TCP. For example, in the case of the Cubic congestion control, approximately 20 seconds are enough, even with a very high difference in network delay, due to its fast growth of congestion window.

In any case, the traffic engineering SDN applications must be made aware of the short drop of network throughput following a rerouting and must limit the frequency of

network re-optimisations. Otherwise, this throughput drop caused by the reaction of the TCP congestion control mechanisms may be incorrectly interpreted as a need for further optimization of network flows, which can cause uncontrolled oscillations of traffic between network paths.

Chapter 7

Conclusion and perspectives

7.1 Conclusion

Information and communication technologies (ICT) are at the core of the digital economies and increasingly important part of all of our personal and professional lives. With this drastic rise in communicating applications, the network operators are in need to deploy additional capacity to their network to support predicted and unpredicted traffic growth. The resulted increase of the energy consumption has not only negative environmental consequences but also an economic impact. Major ICT players, including Google, Microsoft, and Facebook, are already deploying solutions to increase the efficiency of their networks. Nevertheless, these solutions rely on full control of the communicating applications and are limited to private deployments. They are suitable for cases where the communicating applications are not under the network operator's control. This thesis investigated the problem of improving the energy efficiency of operator backbone networks by changing the paths of network flows transparently to the communicating applications.

The first contribution of this thesis is the SegmenT Routing based Energy Efficient Traffic Engineering (STREETE) framework for aggregating the flows over a subset of links and turning the other links *off*. It is an online solution that uses two different views of the network to speed-up the execution. This is especially handy when there is a need for rapidly reacting to an unexpected burst of network traffic; a case that is mostly overlooked by the research community. To further increase the speed of our algorithms, we implemented them using state of the art dynamic shortest path algorithms. Moreover, relying on shortest path computations, together with the SPRING source routing protocol, allows keeping the control traffic overhead at a minimum. The evaluation of STREETE on real backbone network topologies with real traffic matrices enabled us to conclude that good results are obtained at low traffic load both in the number of turned-*off* links and in its ability to avoid network congestion.

To improve the quality of STREETE under heavy load, we searched for a SDN-based traffic engineering technique for optimizing the network utilization. We first presented an unsuccessful attempt to build such a solution using a link cost function. Later we presented a second solution, which we name "LB". It enables near-optimal load balancing of traffic on the network thanks to a careful combination of centralized and distributed computations.

The centralized computation performed by LB at the SDN controller consists of a state

of the art algorithm for approximately solving the maximum concurrent flow problem. It can achieve near-optimal load balancing of traffic in the network. Our contribution is in proposing a means to apply the result of this centralized optimization into the network devices. Our solution keeps a low communication overhead between the SDN controller and the SDN switches. To achieve this goal, part of the optimization is made on the network devices. This allows for locally computing the paths of the flows and update the forwarding databases without the need of centralized orchestration. Similarly to STREETE, we rely on the source routing provided by the SPRING protocol to update the network paths atomically on a single device: the ingress SDN switch.

As a next step, we combined STREETE with the LB load balancing technique and proposed the STREETE-LB framework. Overall, this combination revealed excellent results both concerning the number of turned-*off* links and regarding load distribution in the network. Nevertheless, to avoid an explosion of the computational complexity, we had to reduce the precision of the algorithms used by the load balancing technique, leading to sometimes sub-optimal, but still promising results.

We took the challenge of implementing and evaluating the STREETE framework on a network testbed using real SDN hardware and the ONOS SDN controller. This action allowed us to discover an oscillation due to the elasticity of TCP flows which did behave well under the frequent route changes made by STREETE. To find a safe rerouting frequency, we performed an in-depth evaluation of the behavior of TCP flows under frequent route changes. Contrary to popular belief, the experiments showed that packet reordering incurred by rerouting towards a path with lower RTT has a marginal impact on the bandwidth of TCP flows. This result is partially due to non-standard optimizations introduced by the Linux kernel developers. A degradation of throughput was only observed under very high rerouting frequencies. However, even this effect decreases with the increase of the flow aggregation level. In the meantime, shifting the traffic towards a path with longer RTT has an adverse influence on the bandwidth of TCP flows. We conclude that SDN-based traffic engineering techniques must be constrained to avoid significant variations in the end-to-end delay when rerouting the flows. Alternatively, if the increase of delay is unavoidable, the traffic engineering SDN applications must be made aware of the short temporary of network throughput which follows the rerouting. Otherwise, this short drop of throughput may be wrongly interpreted as a sign to perform a network re-optimization.

In addition to the scientific contributions, we took special care to enable the reproducibility of the results presented in chapters 4 and 5. All the results, except the ones based on the network tested, can be reproduced by following a couple of simple steps on any Linux computer. Moreover, an Android application was developed to allow interactive interaction with the proposed algorithms.

7.2 Perspectives

At critical times in the course of this thesis, some research directions have been preferred over others, leaving entire territories unexplored. Here we describe some of the areas that can be explored in the future.

Continuing the evaluation and implementation As a short term perspective, we would like to continue the implementation and the evaluation of the proposed solutions. The proposed load balancing technique and STREETE-LB were not validated on the real testbed. Some technical difficulties in the implementation may have to be solved to split the demands into multiple paths.

Moreover, we did not evaluate the overhead introduced by the explicit paths into the headers of network packets due to the use of the SPRING/Segment Routing. A huge advantage of SPRING over other source routing protocols is its ability to compress the paths stocked in the header, meaning that there is no need to list the intermediate nodes explicitly. Finding a good and fast technique for path compression, which respects the needs of the SPRING protocol, is a research direction to be considered.

The analysis of the behavior of TCP flows presented in the last chapter was done considering only the *cubic* congestion control algorithm. For the completeness of the study, we should consider testing the behavior of the *compound* algorithm in Windows and of the *newreno* algorithm in FreeBSD. This would cover the absolute majority of the big, long-lived, flows in the internet. In particular, it would be interesting to verify if the implementations in FreeBSD and in Windows are also robust to packet reordering.

Extending the STREETE framework Medium term perspectives include the extension of the STREETE framework to handle additional constraints.

We believe that the most important improvement is the inclusion of various link speeds. At this point, the decision of which link to shut down or to turn on does not take into consideration that it may be better to switch *off* multiple slow links instead of a single, high data-rate one. The load balancing technique presented in Chapter 5 is already capable of working with multiple link speeds. As a result, the main objective would be to find a more energy-aware metric for turning *off* compared to the number of links.

Another relevant problem of the STREETE framework is due to the extensive growth of network path as a result of rerouting towards longer routes. Future research may consider this aspect. We already mentioned that it is possible to use "graph spanners" for this purpose, but did not yet validate this solution.

Link failure is also an aspect that we did not take into consideration. In particular, due to the reduction of the number of active links, a fiber cut can disconnect the network. To avoid this problem, protection links may be computed in the network. We believe that link failure, similarly to the issue of path extension, is better treated at a topological level, by forcing some links to be always active. This would slightly increase the energy consumption but provide a much better resiliency. Moreover, we believe that these two problems may be solved together, the computed graph spanner for the previous solution may already provide a satisfactory level of redundancy.

Focusing on aggregation networks Another perspective would be to study the applicability of STREETE outside the backbone networks.

Devices of backbone networks consume substantial amounts of energy. However, the number of such devices is relatively small. An additional way to reduce the energy consumption would be to move closer to the access networks. Doing small energy savings on a large number of devices is a good way to cut off the overall consumption. However,

STREETE is not suitable for access networks. It relies on the availability of multiple alternative paths between the network devices. In access networks, this is usually not true. That is why protocols similar to 802.3az reign at this level. However, in between the access and the core, lay the aggregation and metro networks. STREETE may be a viable solution at this level. However, due to a relatively large number of network devices, it may be needed to virtually partition these systems in smaller subsets to reduce the computational complexity.

Deadline-driven network optimization Finally, as a long-term research perspective, it would be interesting to study the deadline-driven network optimization techniques: given a certain deadline, find the best possible solution achievable in the provided time. This field is particularly interesting in the context of SDN networks, where meeting a given deadline may be more important than the accuracy of the solution. We may use Chapter 5 as an entry point towards this research direction.

Publications

International Journals:
[J1] M. D. Assunção, R. Carpa, L. Lefevre, O. Glück, P. Borylo, A. Lason, A. Szymanski and M. Rzepka "Designing and Building SDN Testbeds for Energy-Aware Traffic Engineering Services". *Photonic Network Communications(PNET)*, 2017, to appear.
[J2] R. Carpa, O. Glück, L. Lefevre and J.-C. Mignot. "Improving the Energy Efficiency of Software Defined Backbone Networks". *Photonic Network Communications(PNET)*, 2015, vol. 30, nr. 3, pp. 337-347.

International Conferences:
[C1] R. Carpa, M. D. Assunção, O. Glück, L. Lefevre, and J.-C. Mignot. "Responsive Algorithms for Handling Load Surges and Switching Links On in Green Networks." *2016 IEEE International Conference on Communications(ICC16)*, Kuala Lumpur, Malaysia, May 2016.
[C2] M. D. Assunção, R. Carpa, L. Lefèvre and O. Glück. "On Designing SDN Services for Energy-Aware Traffic Engineering." *11th EAI International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities(TRIDENTCOM2016)*, Hangzhou, China, June 2016.
[C3] R. Carpa, O. Glück and L. Lefevre. "Segment routing based traffic engineering for energy efficient backbone networks," *2014 IEEE International Conference on Advanced Networks and Telecommuncations Systems (ANTS)*, New Delhi, India, Dec 2014, pp. 1-6.

National Conferences:
[NC1] R. Carpa, O. Glück, L. Lefevre and J.-C. Mignot. "STREETE : Une ingénierie de trafic pour des réseaux de cœur énergétiquement efficaces." *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS2015).* Lille, France, Jun 2015.

Ongoing work:
[UC1] (submitted to CNSM 2017) R. Carpa, M. D. Assunção, O. Glück, L. Lefevre, and J.-C. Mignot. "Evaluating the Impact of Frequent Route Changes on the Performance of Cubic TCP Flows".
[UJ1] (to be submitted) R. Carpa et al. "A Congestion Avoidance Solution to Reprovision Link Capacity in Green Networks".

Bibliography

[1] Dan Kilper. *Energy challenges in access and aggregation networks*. Symposium: Communication networks beyond the capacity crunch. Accessed: sep/2015. The Royal Society, London, May 2015. URL: https://royalsociety.org/events/2015/05/communication-networks/.

[2] 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". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 3–14. URL: http://doi.acm.org/10.1145/2486001.2486019.

[3] *Geant network looking glass and usage map*. https://tools.geant.net/portal/. Accessed: sep/2015.

[4] "GreenTouch Final Results from Green Meter Research Study". In: *GreenTouch White Paper* (). URL: https://s3-us-west-2.amazonaws.com/belllabs-microsite-greentouch/index.php@page=greentouch-green-meter-research-study.html.

[5] George Karakostas. "Faster Approximation Schemes for Fractional Multicommodity Flow Problems". In: *ACM Trans. Algorithms* 4.1 (Mar. 2008), 13:1–13:17. URL: http://doi.acm.org/10.1145/1328911.1328924.

[6] *Nokia 1830 Photonic Services Switch Datasheet*. https://tools.ext.nokia.com/asset/194070.

[7] Ward Van Heddeghem, Filip Idzikowski, Willem Vereecken, Didier Colle, Mario Pickavet, and Piet Demeester. "Power consumption modeling in optical multilayer networks". English. In: *Photonic Network Communications* 24.2 (2012), pp. 86–102.

[8] *ITU-T G.709 : Interfaces for the optical transport network*. http://www.itu.int/rec/T-REC-G.709/en.

[9] Maruti Gupta and Suresh Singh. "Greening of the Internet". In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '03. Karlsruhe, Germany: ACM, 2003, pp. 19–26.

[10] Xiaowen Dong, T.E.H. El-Gorashi, and J.M.H. Elmirghani. "On the Energy Efficiency of Physical Topology Design for IP Over WDM Networks". In: *Lightwave Technology, Journal of* 30.12 (June 2012), pp. 1931–1942.

[11] J. Chabarek, J. Sommers, P. Barford, C. Estan, D. Tsiang, and S. Wright. "Power Awareness in Network Design and Routing". In: *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. Apr. 2008.

[12] Giuseppe Rizzelli, Annalisa Morea, Massimo Tornatore, and Olivier Rival. "Energy efficient Traffic-Aware design of on–off Multi-Layer translucent optical networks". In: *Computer Networks* 56.10 (2012). Green communication networks, pp. 2443–2455. URL: http://www.sciencedirect.com/science/article/pii/S1389128612001065.

[13] A. Coiro, M. Listanti, and A. Valenti. "Impact of energy-aware topology design and adaptive routing at different layers in IP over WDM networks". In: *2012 15th International Telecommunications Network Strategy and Planning Symposium (NETWORKS)*. Oct. 2012, pp. 1–6.

[14] *Scaling the Netflix Global CDN, lessons learned from Terabit Zero, @scale 2015*. https://www.youtube.com/watch?v=tbqcsHg-Q_o.

[15] D. Kilper, K. Guan, K. Hinton, and R. Ayre. "Energy Challenges in Current and Future Optical Transmission Networks". In: *Proceedings of the IEEE* 100.5 (May 2012), pp. 1168–1187.

[16] Xiaowen Dong, Ahmed Q. Lawey, Taisir E. H. El-Gorashi, and Jaafar M. H. Elmirghani. "Energy-efficient core networks". In: *2012 16th International Conference on Optical Network Design and Modelling (ONDM)* (2012), pp. 1–9.

[17] Frédéric Giroire, Joanna Moulierac, Truong Khoa Phan, and Frédéric Roudaut. "Minimization of network power consumption with redundancy elimination". In: *Computer Communications* 59 (2015), pp. 98–105. URL: http://www.sciencedirect.com/science/article/pii/S0140366414003673.

[18] Mohammed El Mehdi Diouri. "Energy efficiency in very high-performance computing : application to fault tolerance and data broadcasting". Theses. Ecole normale supérieure de lyon - ENS LYON, Sept. 2013. URL: https://tel.archives-ouvertes.fr/tel-00881094.

[19] R. Bolla, R. Bruschi, A. Carrega, F. Davoli, and P. Lago. "A Closed-Form Model for the IEEE 802.3az Network and Power Performance". In: *Selected Areas in Communications, IEEE Journal on* 32.1 (Jan. 2014), pp. 16–27.

[20] R. Bolla, R. Bruschi, and P. Lago. "The hidden cost of network low power idle". In: *2013 IEEE International Conference on Communications (ICC)*. June 2013, pp. 4148–4153.

[21] V. Sivaraman, P. Reviriego, Z. Zhao, A. Sánchez-Macián, A. Vishwanath, J.A. Maestro, and C. Russell. "An experimental power profile of Energy Efficient Ethernet switches". In: *Computer Communications* 50 (2014). Green Networking, pp. 110–118. URL: http://www.sciencedirect.com/science/article/pii/S0140366414000723.

[22] R. Bolla, R. Bruschi, P. Donadio, and G. Parladori. "Energy efficiency in optical networks". In: *Telecommunications Network Strategy and Planning Symposium, 2012 XVth International*. Oct. 2012, pp. 1–6.

[23] CFP MSA Group. *C form-factor pluggable (CFP) Specification*. 2013.

[24] SFF Committee. *SFF-8431 Specification for SFP+*. 2013.

[25] T. Miyazaki, I. Popescuy, M. Chino, X. Wang, K. Ashizawa, S. Okamotoz, M. Veeraraghavan, and N. Yamanaka. "High speed 100GE adaptive link rate switching for energy consumption reduction". In: *2015 International Conference on Optical Network Design and Modeling (ONDM)*. May 2015, pp. 227–232.

[26] A. Morea, O. Rival, N. Brochier, and E. Le Rouzic. "Datarate Adaptation for Night-Time Energy Savings in Core Networks". In: *Lightwave Technology, Journal of* 31.5 (Mar. 2013), pp. 779–785.

[27] "SwiTching And tRansmission (STAR)". In: *Confidential CHIST-ERA Project Periodic Report n° 2* ().

[28] J. Comellas, R. Martinez, J. Prat, V. Sales, and G. Junyent. "Integrated IP/WDM routing in GMPLS-based optical networks". In: *Network, IEEE* 17.2 (Mar. 2003), pp. 22–27.

[29] Chankyun Lee and J.-K.K. Rhee. "Traffic grooming for IP-Over-WDM networks: Energy and delay perspectives". In: *Optical Communications and Networking, IEEE/OSA Journal of* 6.2 (Feb. 2014), pp. 96–103.

[30] W. Van Heddeghem, B. Lannoo, D. Colle, M. Pickavet, F. Musumeci, A. Pattavina, and F. Idzikowski. "Power consumption evaluation of circuit-switched versus packet-switched optical backbone networks". In: *2013 IEEE Online Conference on Green Communications (OnlineGreenComm)*. Oct. 2013, pp. 56–63.

[31] F. Musumeci, D. Siracusa, G. Rizzelli, M. Tornatore, R. Fiandra, and A. Pattavina. "On the energy consumption of IP-over-WDM architectures". In: *Communications (ICC), 2012 IEEE International Conference on*. June 2012, pp. 3004–3008.

[32] F. Idzikowski, S. Orlowski, C. Raack, H. Woesner, and A. Wolisz. "Saving energy in IP-over-WDM networks by switching off line cards in low-demand scenarios". In: *Optical Network Design and Modeling (ONDM), 2010 14th Conference on*. Feb. 2010, pp. 1–6.

[33] Filip Idzikowski, Sebastian Orlowski, Christian Raack, Hagen Woesner, and Adam Wolisz. "Dynamic Routing at Different Layers in IP-over-WDM Networks - Maximizing Energy Savings". In: *Opt. Switch. Netw.* 8.3 (July 2011), pp. 181–200. URL: http://dx.doi.org/10.1016/j.osn.2011.03.007.

[34] R. S. Tucker, R. Parthiban, J. Baliga, K. Hinton, R. W. A. Ayre, and W. V. Sorin. "Evolution of WDM Optical IP Networks: A Cost and Energy Perspective". In: *Journal of Lightwave Technology* 27.3 (Feb. 2009), pp. 243–252.

[35] Annalisa Morea, Jordi Perelló, Salvatore Spadaro, Dominique Verchère, and Martin Vigoureux. "Protocol Enhancements for "Greening" Optical Networks." In: *Bell Labs Technical Journal* 18.3 (2013), pp. 211–230. URL: http://dblp.uni-trier.de/db/journals/bell/bell18.html#MoreaPSVV13.

[36] Xin Chen and C. Phillips. "Virtual router migration and infrastructure sleeping for energy management of IP over WDM networks". In: *Telecommunications and Multimedia (TEMU), 2012 International Conference on*. July 2012, pp. 31–36.

[37] V. Eramo, M. Listanti, A. Cianfrani, and E. Miucci. "Evaluation of power saving in an MPLS/IP network hosting a virtual router layer of a single service provider". In: *ICT Convergence (ICTC), 2013 International Conference on*. Oct. 2013, pp. 12–17.

[38] V. Eramo, S. Testa, and E. Miucci. "Evaluation of Power Saving and Feasibility Study of Migrations Solutions in a Virtual Router Network". In: *Journal of Electrical and Computer Engineering* (2014).

[39] John Moy. *OSPF Version 2*. RFC 2328. Apr. 1998. URL: https://rfc-editor.org/rfc/rfc2328.txt.

[40] D. Oran. *OSI IS-IS Intra-domain Routing Protocol*. RFC 1142 (Informational). Internet Engineering Task Force, Feb. 1990. URL: http://www.ietf.org/rfc/rfc1142.txt.

[41] M. Goyal, M. Soperi, E. Baccelli, G. Choudhury, A. Shaikh, H. Hosseini, and K. Trivedi. "Improving Convergence Speed and Scalability in OSPF: A Survey". In: *IEEE Communications Surveys Tutorials* 14.2 (Second 2012), pp. 443–463.

[42] Stefano Salsano, Alessio Botta, Paola Iovanna, Marco Intermite, and Andrea Polidoro. "Traffic Engineering with OSPF-TE and RSVP-TE: Flooding Reduction Techniques and Evaluation of Processing Cost". In: *Comput. Commun.* 29.11 (July 2006), pp. 2034–2045.

[43] X. Liu, S. Mohanraj, M. Pióro, and D. Medhi. "Multipath Routing from a Traffic Engineering Perspective: How Beneficial Is It?" In: *2014 IEEE 22nd International Conference on Network Protocols*. Oct. 2014, pp. 143–154.

[44] Inc. Juniper Networks. *PTX Series Packet Transport Routers Datasheet*. 2014. URL: http://www.juniper.net/us/en/local/pdf/datasheets/1000364-en.pdf.

[45] A. Farrel, A. Ayyangar, and JP. Vasseur. *Inter-Domain MPLS and GMPLS Traffic Engineering – Resource Reservation Protocol-Traffic Engineering (RSVP-TE) Extensions*. RFC 5151 (Proposed Standard). Internet Engineering Task Force, Feb. 2008. URL: http://www.ietf.org/rfc/rfc5151.txt.

[46] Evangelos Haleplidis, Kostas Pentikousis, Spyros Denazis, Jamal Hadi Salim, David Meyer, and Odysseas Koufopavlou. *Software-Defined Networking (SDN): Layers and Architecture Terminology*. RFC 7426. Jan. 2015. URL: https://rfc-editor.org/rfc/rfc7426.txt.

[47] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. "Achieving High Utilization with Software-driven WAN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 15–26. URL: http://doi.acm.org/10.1145/2486001.2486012.

[48] A.P. Bianzino, L. Chiaraviglio, and M. Mellia. "GRiDA: A green distributed algorithm for backbone networks". In: *Online Conference on Green Communications (GreenCom), 2011 IEEE*. Sept. 2011, pp. 113–119.

[49] A. P. Bianzino, L. Chiaraviglio, and M. Mellia. "Distributed algorithms for green IP networks". In: *2012 Proceedings IEEE INFOCOM Workshops*. Mar. 2012, pp. 121–126.

[50] A. Cianfrani, V. Eramo, M. Listanti, M. Polverini, and A. V. Vasilakos. "An OSPF-Integrated Routing Strategy for QoS-Aware Energy Saving in IP Backbone Networks". In: *IEEE Transactions on Network and Service Management* 9.3 (Sept. 2012), pp. 254–267.

[51] M. Rifai, D. Lopez Pacheco, and G. Urvoy-Keller. "Towards enabling green routing services in real networks". In: *2014 IEEE Online Conference on Green Communications (OnlineGreenComm)*. Nov. 2014, pp. 1–7.

[52] Meng Shen, Hongying Liu, Ke Xu, Ning Wang, and Yifeng Zhong. "Routing On Demand: Toward the Energy-Aware Traffic Engineering with OSPF". In: *NETWORKING 2012: 11th International IFIP TC 6 Networking Conference, Prague, Czech Republic, May 21-25, 2012, Proceedings, Part I*. Ed. by Robert Bestak, Lukas Kencl, Li Erran Li, Joerg Widmer, and Hao Yin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 232–246. URL: http://dx.doi.org/10.1007/978-3-642-30045-5_18.

[53] Mingui Zhang, Cheng Yi, Bin Liu, and Beichuan Zhang. "GreenTE: Power-aware traffic engineering". In: *Network Protocols (ICNP), 2010 18th IEEE International Conference on*. Oct. 2010, pp. 21–30.

[54] F. Idzikowski, L. Chiaraviglio, A. Cianfrani, J. López Vizcaíno, M. Polverini, and Y. Ye. "A Survey on Energy-Aware Design and Operation of Core Networks". In: *IEEE Communications Surveys Tutorials* 18.2 (Secondquarter 2016), pp. 1453–1499.

[55] B. Addis, A. Capone, G. Carello, L.G. Gianoli, and B. Sansò. "On the energy cost of robustness and resiliency in IP networks". In: *Computer Networks* 75 (2014), pp. 239–259. URL: http://www.sciencedirect.com/science/article/pii/S1389128614003594.

[56] Luca Chiaraviglio, Antonio Cianfrani, Esther Le Rouzic, and Marco Polverini. "Sleep modes effectiveness in backbone networks with limited configurations". In: *Computer Networks* 57.15 (2013), pp. 2931–2948. URL: http://www.sciencedirect.com/science/article/pii/S1389128613002028.

[57] Joanna Moulierac and Truong Khoa Phan. "Optimizing IGP Link Weights for Energy-efficiency in Multi-period Traffic Matrices". In: *Comput. Commun.* 61.C (May 2015), pp. 79–89. URL: http://dx.doi.org/10.1016/j.comcom.2015.01.004.

[58] R. Wang, Z. Jiang, S. Gao, W. Yang, Y. Xia, and M. Zhu. "Energy-aware routing algorithms in Software-Defined Networks". In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. June 2014, pp. 1–6.

[59] H. Yonezu, K. Kikuta, D. Ishii, S. Okamoto, E. Oki, and N. Yamanaka. "QoS aware energy optimal network topology design and dynamic link power management". In: *36th European Conference and Exhibition on Optical Communication*. Sept. 2010, pp. 1–3.

[60] Mohamad Khattar Awad, Mohammed El-Shafei, Tassos Dimitriou, Yousef Rafique, Mohammed Baidas, and Ammar Alhusaini. "Power-efficient routing for SDN with discrete link rates and size-limited flow tables: A tree-based particle swarm optimization approach". In: *International Journal of Network Management* (2017). e1972 nem.1972, e1972–n/a. URL: http://dx.doi.org/10.1002/nem.1972.

[61] F. Giroire, J. Moulierac, and T. K. Phan. "Optimizing rule placement in software-defined networks for energy-aware routing". In: *2014 IEEE Global Communications Conference*. Dec. 2014, pp. 2523–2529.

[62] Luca Davoli, Luca Veltri, Pier Luigi Ventre, Giuseppe Siracusano, and Stefano Salsano. "Traffic Engineering with Segment Routing: SDN-Based Architectural Design and Open Source Implementation". In: *Proceedings of the 2015 Fourth European Workshop on Software Defined Networks*. EWSDN '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 111–112. URL: http://dx.doi.org/10.1109/EWSDN.2015.73.

[63] C. Thaenchaikun, G. Jakllari, B. Paillassa, and W. Panichpattanakul. "Mitigate the load sharing of segment routing for SDN green traffic engineering". In: *2016 International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*. Oct. 2016, pp. 1–6.

[64] R. Wang, Z. Jiang, S. Gao, W. Yang, Y. Xia, and M. Zhu. "Energy-aware routing algorithms in Software-Defined Networks". In: *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*. June 2014, pp. 1–6.

[65] François Baccelli and Dohy Hong. "Interaction of TCP Flows As Billiards". In: *IEEE/ACM Trans. Netw.* 13.4 (Aug. 2005), pp. 841–853. URL: http://dx.doi.org/10.1109/TNET.2005.852883.

[66] Jon CR Bennett, Craig Partridge, and Nicholas Shectman. "Packet reordering is not pathological network behavior". In: *IEEE/ACM Transactions on Networking (TON)* 7.6 (1999), pp. 789–798.

[67] J. He, M. Bresler, M. Chiang, and J. Rexford. "Towards Robust Multi-Layer Traffic Engineering: Optimization of Congestion Control and Routing". In: *IEEE J.Sel. A. Commun.* 25.5 (June 2007), pp. 868–880. URL: http://dx.doi.org/10.1109/JSAC.2007.070602.

[68] M. Laor and L. Gendel. "The effect of packet reordering in a backbone link on application throughput". In: *IEEE Network* 16.5 (Sept. 2002), pp. 28–36.

[69] K. c. Leung, V. O. k. Li, and D. Yang. "An Overview of Packet Reordering in Transmission Control Protocol (TCP): Problems, Solutions, and Challenges". In: *IEEE Transactions on Parallel and Distributed Systems* 18.4 (Apr. 2007), pp. 522–535.

[70] Jie Feng, Zhipeng Ouyang, Lisong Xu, and Byrav Ramamurthy. "Packet reordering in high-speed networks and its impact on high-speed {TCP} variants". In: *Computer Communications* 32.1 (2009), pp. 62–68. URL: http://www.sciencedirect.com/science/article/pii/S0140366408005100.

[71] Ethan Blanton and Mark Allman. "On Making TCP More Robust to Packet Reordering". In: *SIGCOMM Comput. Commun. Rev.* 32.1 (Jan. 2002), pp. 20–30. URL: http://doi.acm.org/10.1145/510726.510728.

[72] Reiner Ludwig and Randy H. Katz. "The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions". In: *SIGCOMM Comput. Commun. Rev.* 30.1 (Jan. 2000), pp. 30–36. URL: http://doi.acm.org/10.1145/505688.505692.

[73] J. Karlsson, P. Hurtig, A. Brunstrom, A. Kassler, and G. Di Stasi. "Impact of multi-path routing on TCP performance". In: *2012 IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. June 2012, pp. 1–3.

[74] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. "On the impact of packet spraying in data center networks". In: *2013 Proceedings IEEE INFOCOM*. Apr. 2013, pp. 2130–2138.

[75] G. Swallow, S. Bryant, and L. Andersson. *Avoiding Equal Cost Multipath Treatment in MPLS Networks*. RFC 4928 (Best Current Practice). RFC. Updated by RFC 7274. Fremont, CA, USA: RFC Editor, June 2007. URL: https://www.rfc-editor.org/rfc/rfc4928.txt.

[76] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. "HULA: Scalable Load Balancing Using Programmable Data Planes". In: *Proceedings of the Symposium on SDN Research*. SOSR '16. Santa Clara, CA, USA: ACM, 2016, 10:1–10:12. URL: http://doi.acm.org/10.1145/2890955.2890968.

[77] Keqiang He, Junaid Khalid, Aaron Gember-Jacobson, Sourav Das, Chaithan Prakash, Aditya Akella, Li Erran Li, and Marina Thottan. "Measuring Control Plane Latency in SDN-enabled Switches". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. Santa Clara, California: ACM, 2015, 25:1–25:6. URL: http://doi.acm.org/10.1145/2774993.2775069.

[78] *Lessons learned from B4, Google's SDN WAN*. https://atscaleconference.com/videos/lessons-learned-from-b4-googles-sdn-wan/. 2015.

[79] M. Kamola and P. Arabas. "Shortest path green routing and the importance of traffic matrix knowledge". In: *Digital Communications - Green ICT (TIWDC), 2013 24th Tyrrhenian International Workshop on*. Sept. 2013, pp. 1–6.

[80] Y. Ohsita, T. Miyamura, S. Arakawa, S. Ata, E. Oki, K. Shiomoto, and M. Murata. "Gradually Reconfiguring Virtual Network Topologies Based on Estimated Traffic Matrices". In: *IEEE/ACM Transactions on Networking* 18.1 (Feb. 2010), pp. 177–189.

[81] Anders Gunnar, Mikael Johansson, and Thomas Telkamp. "Traffic Matrix Estimation on a Large IP Backbone: A Comparison on Real Data". In: *Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement*. IMC '04. Taormina, Sicily, Italy: ACM, 2004, pp. 149–160.

[82] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. "FlowSense: Monitoring Network Utilization with Zero Measurement Cost". In: *Passive and Active Measurement: 14th International Conference, PAM 2013, Hong Kong, China, March 18-19, 2013. Proceedings*. Ed. by Matthew Roughan and Rocky Chang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 31–41. URL: http://dx.doi.org/10.1007/978-3-642-36516-4_4.

[83] C. Filsfils, S. Previdi, A. Bashandy, B. Decraene, S. Litkowski, and R. Shakir. *Segment Routing Architecture", draft-ietf-spring-segment-routing-06 (work in progress)*. Oct. 2015.

[84] *IPv6 segment routing in linux kernel*. https://lwn.net/Articles/722804/. 2017.

[85] *Cisco Segment Routing Overview with Clarence Filsfils*. https://www.youtube.com/watch?v=ZpU 2016.

[86] S.P. Bradley, A.C. Hax, and T.L. Magnanti. *Applied Mathematical Programming*. Addison-Wesley Publishing Company, 1977. URL: https://books.google.fr/books?id=MSWdWv3Gn5cC.

[87] Wilbert E. Wilhelm. "A Technical Review of Column Generation in Integer Programming". In: *Optimization and Engineering* 2.2 (2001), pp. 159–200. URL: http://dx.doi.org/10.1023/A:1013141227104.

[88] Farhad Shahrokhi and D. W. Matula. "The Maximum Concurrent Flow Problem". In: *J. ACM* 37.2 (Apr. 1990), pp. 318–334. URL: http://doi.acm.org/10.1145/77600.77620.

[89] David Peleg and Alejandro A Schäffer. "Graph spanners". In: *Journal of graph theory* 13.1 (1989), pp. 99–116.

[90] Frédéric Giroire, Stephane Perennes, and Issam Tahiri. "Grid spanners with low forwarding index for energy efficient networks ". In: *International Network Optimization Conference (INOC)*. Electronic Notes in Discrete Mathematics. Warsaw, Poland, May 2015. URL: https://hal.inria.fr/hal-01218411.

[91] Camil Demetrescu and Giuseppe F. Italiano. "Experimental Analysis of Dynamic All Pairs Shortest Path Algorithms". In: *ACM Trans. Algorithms* 2.4 (Oct. 2006), pp. 578–601. URL: http://doi.acm.org/10.1145/1198513.1198519.

[92] G. Ramalingam and Thomas Reps. "An Incremental Algorithm for a Generalization of the Shortest-path Problem". In: *J. Algorithms* 21.2 (Sept. 1996), pp. 267–305. URL: http://dx.doi.org/10.1006/jagm.1996.0046.

[93] Korotky S. "Projections of IP Traffic to 2020 Based on Regression Analyses of Historical Trends and Nearer Term Forecasts". In: *GreenTouch Confidential Report* ().

[94] Hinton K. "Traffic modelling for the core network". In: *GreenTouch Confidential Report* ().

[95] *OMNeT++ Discrete Event Simulator*. `https://omnetpp.org/`.

[96] *Introducing ONOS: A SDN Network Operating System for Service Providers*. Whitepaper. Open Networking Lab ON.Lab, Nov. 2014. URL: `http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf`.

[97] JongWon Kim, ByungRae Cha, Jongryool Kim, Namgon Lucas Kim, Gyeongsoo Noh, Youngwan Jang, Hyeong Geun An, Hongsik Park, JiHoon Hong, DongSeok Jang, TaeWan Ko, Wang-Cheol Song, Seokhong Min, Jaeyong Lee, Byungchul Kim, Ilkwon Cho, Hyong-Soon Kim, and Sun-Moo Kang. "Proceedings of the Asia-Pacific Advanced Network". In: *OF@TEIN: An OpenFlow-enabled SDN Testbed over International SmartX Rack Sites* 36 (2013), pp. 17–22.

[98] N.B. Melazzi, A. Detti, G. Mazza, G. Morabito, S. Salsano, and L. Veltri. "An OpenFlow-based Testbed for Information Centric Networking". In: *Future Network Mobile Summit (FutureNetw 2012)*. July 2012, pp. 1–9.

[99] Sebastia Sallent, Antonio Abelém, Iara Machado, Leonardo Bergesio, Serge Fdida, Jose Rezende, Siamak Azodolmolky, Marcos Salvador, Leandro Ciuffo, and Leandros Tassiulas. "FIBRE Project: Brazil and Europe Unite Forces and Testbeds for the Internet of the Future". In: *Testbeds and Research Infrastructure, Development of Networks and Communities*. Ed. by Thanasis Korakis, Michael Zink, and Maximilian Ott. Vol. 44. LNICST. Springer Berlin Heidelberg, 2012, pp. 372–372.

[100] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lantéri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Touche Iréa. "Grid'5000: a large scale and highly reconfigurable experimental Grid testbed". In: *Int. Journal of High Performance Computing Applications* 20.4 (Nov. 2006), pp. 481–494.

[101] Emmanuel Jeanvoine, Luc Sarzyniec, and Lucas Nussbaum. "Kadeploy3: Efficient and Scalable Operating System Provisioning". In: *USENIX ;login:* 38.1 (Feb. 2013), pp. 38–44.

[102] Francois Rossigneux, Jean-Patrick Gelas, Laurent Lefevre, and Marcos Dias de Assuncao. "A Generic and Extensible Framework for Monitoring Energy Consumption of OpenStack Clouds". In: *SustainCom 2014*. Sydney, Australia, Dec. 2014, pp. 696–702.

[103] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. "The Design and Implementation of Open vSwitch". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2015)*. 2015.

[104] *NetFPGA 10G*. `http://netfpga.org`.

[105] *The NetFPGA-10G UPB OpenFlow Switch*. `https://github.com/pc2/NetFPGA-10G-UPB-OpenFlow`.

[106] Shih-Chun Lin, Pu Wang, and Min Luo. "Control traffic balancing in software defined networks". In: *Computer Networks* 106 (2016), pp. 260–271. URL: http://www.sciencedirect.com/science/article/pii/S1389128615002571.

[107] Pasi Sarolahti and Alexey Kuznetsov. "Congestion Control in Linux TCP". In: *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2002, pp. 49–62. URL: http://dl.acm.org/citation.cfm?id=647056.715932.

[108] A. Hassidim, D. Raz, M. Segalov, and A. Shaqed. "Network utilization: The flow view". In: *INFOCOM, 2013 Proceedings IEEE*. Apr. 2013, pp. 1429–1437.

[109] I. Juva, R. Susitaival, M. Peuhkuri, and S. Aalto. "Traffic characterization for traffic engineering purposes: analysis of Funet data". In: *Next Generation Internet Networks, 2005*. Apr. 2005, pp. 404–411.

[110] S. Uhlig, B. Quoitin, S. Balon, and J. Lepropre. "Providing public intradomain traffic matrices to the research community". In: *ACM SIGCOMM Computer Communication Review* 36.1 (Jan. 2006).

[111] R.O. De Schmidt, R. Sadre, and A. Pras. "Gaussian traffic revisited". In: *IFIP Networking Conference, 2013*. May 2013, pp. 1–9.

[112] B. Jiang. "Head/tail Breaks: A New Classification Scheme for Data with a Heavy-tailed Distribution". In: *ArXiv e-prints* (Sept. 2012). arXiv: 1209.2801 [physics.data-an].

[113] Morton Klein. "A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems". In: *Management Science* 14.3 (1967), pp. 205–220.

[114] Lisa K. Fleischer and Kevin D. Wayne. "Fast and simple approximation schemes for generalized flow". In: *Mathematical Programming* 91.2 (2002), pp. 215–238. URL: http://dx.doi.org/10.1007/s101070100238.

[115] Naveen Garg and Jochen Könemann. "Faster and Simpler Algorithms for Multicommodity Flow and Other Fractional Packing Problems". In: *SIAM Journal on Computing* 37.2 (2007), pp. 630–652. eprint: https://doi.org/10.1137/S0097539704446232. URL: https://doi.org/10.1137/S0097539704446232.

[116] A. B. Kahn. "Topological Sorting of Large Networks". In: *Commun. ACM* 5.11 (Nov. 1962), pp. 558–562. URL: http://doi.acm.org/10.1145/368996.369025.

[117] Reuven Cohen, Liran Katzir, and Danny Raz. "An efficient approximation for the Generalized Assignment Problem". In: *Information Processing Letters* 100.4 (2006), pp. 162–166. URL: http://www.sciencedirect.com/science/article/pii/S0020019006001931.

[118] L. Qian and B. E. Carpenter. "A flow-based performance analysis of TCP and TCP applications". In: *2012 18th IEEE International Conference on Networks (ICON)*. Dec. 2012, pp. 41–45.

[119] Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: A New TCP-friendly High-speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. URL: http://doi.acm.org/10.1145/1400097.1400105.

[120] H. H. Gharakheili, A. Vishwanath, and V. Sivaraman. "Edge versus host pacing of TCP traffic in small buffer networks". In: *2013 IFIP Networking Conference*. May 2013, pp. 1–9.

[121] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. "BBR: Congestion-Based Congestion Control". In: *ACM Queue* 14, September-October (2016), pp. 20–53. URL: `http://queue.acm.org/detail.cfm?id=3022184`.

Appendices

Using the android application

The android application can be installed by scanning the QR code from Fig. 1. It allows to interactively visualize the execution of the algorithms proposed in chapters 4 and 5 of this thesis.

Figure 1: Access app using Google Play
`https://play.google.com/store/apps/details?id=me.carpa.streete`

At launch, a network view is opened (Fig. 2). Its main components are:

- **(1)** Network selection. Sliding to the left/right reveals more available network topologies. To select a network, click on it.
- **(2)** Algorithm selection. By default, a shortest path routing based on the Dijkstra's algorithm is used. Checking "Consume Less", "Consume Better", or both, enables respectively STREETE, LB, or STREETE-LB.
- **(3)** Network state. It shows the link utilization when the algorithm selected by the previous check-boxes is applied into the network. On start, an uniform all-to-all traffic is injected into the network.
- **(4)** The slider allows to increase / reduce the traffic in the network by a multiplicative factor.

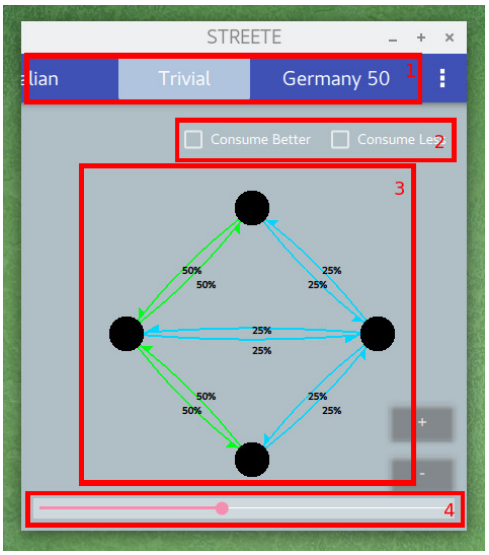
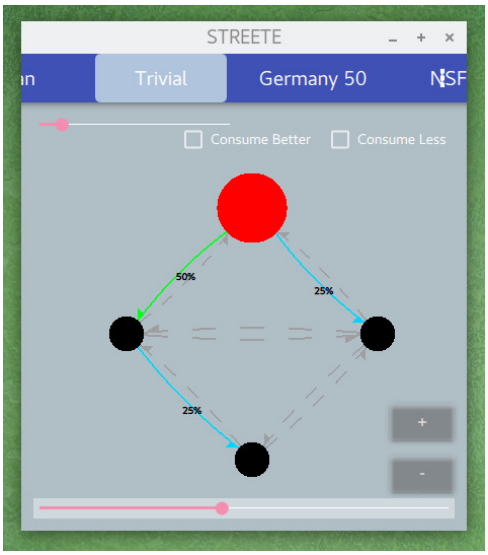


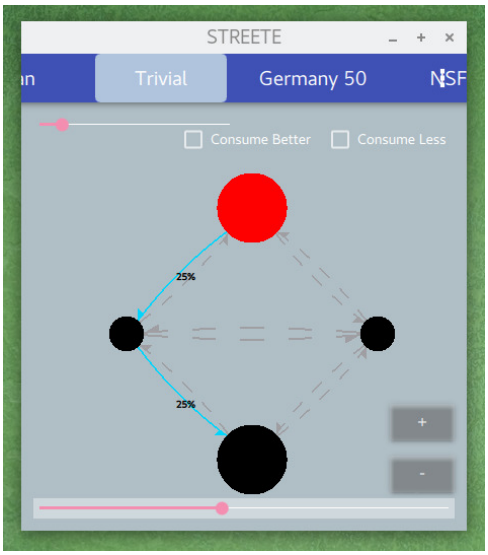
Figure 2: Main interface of the application

It is possible to click on a node (Fig. 3a) to select it. Doing so, filters on the the total flow originating from the selected node towards all other nodes in the network. Moreover, a new slider appears at the top-left corner. It allows to increase/reduce this flow. However, this functionality is in an early stage of development: the changes are applied into the network, but the slider does not keep the state.

To deselect the node, it is sufficient to click on it a second time.



(a) Clicking on a node



(b) Clicking on a second node

Figure 3: Interacting with the network

While having a node selected, it is possible to click on a second node to visualize the paths taken by the flows between these two nodes and the associated network flow (Fig. 3b).