Distributed and trustable SDN-NFV-enabled network emulation on testbeds and cloud infrastructures

Giuseppe Di Lena

▶ To cite this version:

Giuseppe Di Lena. Distributed and trustable SDN-NFV-enabled network emulation on testbeds and cloud infrastructures. Networking and Internet Architecture [cs.NI]. Université Côte d'Azur, 2021. English. NNT : 2021COAZ4028 . tel-03343533

HAL Id: tel-03343533

https://theses.hal.science/tel-03343533v1

Submitted on 14 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Emulation fiable et distribuée de réseaux virtualisés et programmables sur bancs de test et infrastructures cloud

Giuseppe DI LENA

Orange Labs - Inria Sophia Antipolis

Présentée en vue de l'obtention du grade de docteur en INFORMATIQUE
d'Université Côte d'Azur

Dirigée par : Thierry Turletti / Frédéric Giroire / Chidung Lac

Soutenue le : 22/03/2021

Devant le jury, composé de :

Président:
Guillaume Urvoy-Keller, *Professeur, Université Côte d'Azur*

Rapporteurs:
Marcelo Dias de Amorim, *Directeur de recherche, CNRS*
Stefano Secci, *Professeur des Universités, Cnam*

Examinateurs:
Mathieu Bouet, *Ingénieur, Thales*
Luigi Iannone, *Maître de conférence, Telecom Paris*

Directeur:
Thierry Turletti, *Directeur de recherche, Inria*

Co-Directeurs:
Frédéric Giroire, *Directeur de recherche, CNRS*
Chidung Lac, *Ingénieur-chercheur, Orange Labs*

i3S sophia antipolis

orange™

Inria

Acknowledgements

Firstly, I'd like to express my thanks to my patient and supportive supervisors Frédéric Giroire, Thierry Turletti, and Chidung Lac to guide me well throughout this research work.

I would like to acknowledge my colleagues and team leaders from Inria and Orange Labs, with a special mention to Damien Saucez, Andrea Tomassilli, Walid Dabbous, David Coudert, Jean-Philippe Luc, and Alain Henry.

I would like to also express acknowledgements to Marcelo Dias de Amorim and Stefano Secci, who both reviewed my thesis, thank you for your comments and suggestions.

I would also like to thank my committee members, Guillaume Urvoy-Keller, Mathieu Bouet, and Luigi Iannone.

A special thanks to my family, Caterina, Federica, Daniele, Rodolfo, and Alessandro.

Résumé

De nombreux progrès ont eu lieu ces dernières années dans les domaines de la virtualisation, l'informatique en nuage et la programmation des fonctions réseau. L'essor des concepts tels que Software Defined Networking (SDN) et Network Function Virtualization (NFV) a largement modifié la manière dont les fournisseurs de services Internet gèrent leurs offres. Parallèlement, au cours de la dernière décennie, les plateformes sécurisées de Cloud publiques telles que Amazon AWS ou Microsoft Azure sont devenues des acteurs incontournables de la scène. Ces nouveaux concepts permettent des réductions de coûts et une plus grande rapidité d'innovation, ce qui a conduit à l'adoption de ces paradigmes par l'industrie. Tous ces changements apportent également leur lot de nouveaux défis. Tout en étant devenus tentaculaires et complexes, ces réseaux offrent une plus grande diversité de services: les tester devient ainsi de plus en plus compliqué, tout en nécessitant beaucoup de ressources. Pour résoudre ce problème, nous proposons un nouvel outil qui combine les technologies d'émulation et les techniques d'optimisation afin de distribuer les simulations SDN/NFV dans des bancs de test privés et des plateformes de Cloud publiques. Par ailleurs, les fournisseurs de Cloud proposent en général aux utilisateurs des métriques spécifiques en termes de CPU et de ressources mémoire afin de caractériser leurs services, mais ont tendance à présenter une vue d'ensemble de haut niveau du délai maximum engendré par le réseau, sans aucune valeur spécifique. Ceci peut constituer un problème lorsqu'il s'agit de déployer des applications sensibles au délai dans le Cloud, car les utilisateurs n'ont pas de données précises sur ce sujet. Nous proposons un cadre de test pour surveiller le délai engendré par le réseau entre plusieurs centres de données des infrastructures Cloud. Enfin, dans le contexte des réseaux SDN/NFV, nous exploitons la logique centralisée SDN pour implémenter une stratégie optimale de routage en cas de défaillances multiples des liens dans le réseau. Un environnement de banc de test a également été créé afin de valider nos propositions pour différentes topologies de réseau.
−**Mots-clés**: *Emulation, SDN, NFV, Mininet, Virtualisation*

Abstract

In recent years, there have been multiple enhancements in virtualization technologies, cloud computing, and network programmability. The emergence of concepts like *Software Defined Networking (SDN)* and *Network Function Virtualization (NFV)* are changing the way the Internet Service Providers manage their services. In parallel, the last decade witnessed the rise of secure public cloud platforms like Amazon AWS and Microsoft Azure. These new concepts lead to cost reductions and fast innovation, driving the adoption of these paradigms by the industry. All these changes also bring new challenges. Networks have become huge and complex while providing different kinds of services. Testing them is increasingly complicated and resource-intensive. To tackle this issue, we propose a new tool that combines emulation technologies and optimization techniques to distribute SDN/NFV experiments in private test-beds and public cloud platforms. Cloud providers, in general, deliver specific metrics to the users in terms of CPU and memory resources for the services they propose, but they tend to give a high-level overview for the network delay, without any specific value. This is a problem when deploying a delay-sensitive application in the cloud, since the users do not have any precise data about the delay. We propose a testing framework to monitor the network delay between multiple datacenters in the cloud infrastructures. Finally, in the context of SDN/NFV networks, we exploit the SDN centralized logic to implement an optimal routing strategy in case of multiple link failures in the network. We also created a test-bed environment to validate our proposition in different network topologies.
−**Keywords**: *Emulation, SDN, NFV, Mininet, Virtualization*

Contents

1 Introduction **9**
1.1 Context . . . 9
1.2 Challenges . . . 11
1.2.1 Network Emulation . . . 11
1.2.2 Cloud Testing . . . 12
1.2.3 Failure Recovery . . . 12
1.3 Contributions . . . 12
1.4 Outline . . . 15

2 Background **17**
2.1 Virtualization . . . 17
2.1.1 Hypervisor-Based Virtualization . . . 17
2.1.2 Container Virtualization . . . 20
2.2 Network Function Virtualization (NFV) . . . 21
2.3 Software Defined Networking (SDN) . . . 22
2.3.1 SDN Architecture . . . 23
2.3.2 Openflow . . . 24
2.4 Service Function Chaining (SFC) . . . 25
2.5 Test-beds . . . 26
2.6 Cloud Environments . . . 27

3 Distrinet **31**
3.1 Introduction . . . 31
3.2 Related Work . . . 32
3.3 Distributed Mininet . . . 34
3.3.1 Multi-Host Mininet Implementation . . . 35
3.4 Distrinet Architecture . . . 38
3.5 Experiments . . . 39
3.5.1 Distrinet Core Performance Assessment . . . 39
3.5.2 Tools Comparison . . . 40
3.5.3 Experiments With High Load . . . 42
3.6 Conclusions . . . 44

4 Distributed Network Emulation **45**
4.1 Introduction . . . 45
4.2 Related Work . . . 47
4.3 Problem And Algorithms . . . 50
4.3.1 Problem Statement . . . 50
4.3.2 Algorithms . . . 51
4.3.3 Numerical Evaluation . . . 56
4.4 Evaluation Of The Placement Modules . . . 57
4.5 Overloading Experiment . . . 70
4.5.1 Bandwidth Intensive Experiments . . . 71
4.5.2 CPU Intensive Experiments . . . 74
4.5.3 Memory Intensive Experiments . . . 75
4.6 Conclusions . . . 77

5 Cloud Measurement **79**
5.1 Introduction . . . 79
5.2 Related Work . . . 80
5.3 Amazon Web Services Infrastructure . . . 82
5.4 Implementation . . . 84
5.5 Experiments . . . 87
5.6 Conclusions . . . 92

6 Failure Recovery **93**
6.1 Introduction . . . 93
6.2 Related Work . . . 94
6.3 Optimization Approaches . . . 96
6.3.1 Problem Statement And Notations . . . 96
6.3.2 A Layered Network Model . . . 99
6.3.3 Compact ILP Formulation . . . 100
6.3.4 Column Generation Approach . . . 101
6.3.5 Benders Decomposition Approach . . . 103
6.3.6 The Min-Overflow Problem . . . 104
6.4 Numerical Results . . . 109
6.5 Implementation Perspectives . . . 118
6.5.1 Implementation Options . . . 118
6.5.2 Experimental Setup . . . 119
6.5.3 Recovery Time . . . 120
6.5.4 Operational Trade-Offs . . . 124
6.6 Conclusion . . . 125

7 Conclusion And Future Work **127**
7.1 Summary Of Contributions . . . 127
7.2 Future Work . . . 128

Chapter 1

Introduction

1.1 Context

Networks are continuously growing and evolving, and the recent advances in virtualization and orchestration are accelerating this evolution. 5G is the perfect example of how Internet Service Providers (ISPs) are changing their offers and manage access to their infrastructures via SDN and NFV technologies. Another example of the rise of virtualization technologies is cloud computing. It is estimated that the global cloud computing market will grow from US $371.4 billion in 2020 to US $832.1 billion by 2025 Compound Annual Growth Rate of 17.5%. The major players are Amazon Web Services (AWS) and Microsoft Azure, followed by Google Cloud. The main advantage that the virtualization technology provides is flexibility. It is easy and just takes a few seconds to create a fully operational virtual machine, simply on demand. This flexibility can reduce for network operators and general IT companies the OPerating EXpense (OPEX) and CAPital EXpenditure (CAPEX) to deploy and operate around the globe.

Before deploying any network, there are studies and tests to be performed, to avoid congestion and not underestimate (or overestimate) the network capacity. Since the size and complexity of these networks grow, the number of resources needed to emulate them increases. A straightforward solution to solve this issue is vertically scale the emulating host in order to match the required resources. This solution cannot scale to infinity since a single machine will reach its limits at some point. The solution is to scale the emulation horizontally between multiple hosts. This is not an easy task, since most of the network emulators are designed to run on a single host. Another limitation is that NFV networks can have multiple services designed to run on different virtual environments. This means that for each service to deploy, the emulator requires a virtual environment.

To solve this issue, we can use Virtual Machines (VM) or containers. Our work aims to design a flexible SDN/NFV network emulator, easy to distribute in private clusters, test-beds, and public clouds, and compatible with the main

SDN emulator available in the market (i.e., Mininet). This project is divided into two different parts. The first part overviews the technical design decision made to create the emulator's architecture, describing the orchestration and virtualization technologies, while the second part focuses on the algorithmic problem raised when distributing a virtual network in a physical infrastructure. We investigated and compared our tool with the one available in the market, showing that our approach is the one that provides the most trusted results. The network emulator is designed to work in a controlled environment, like a test-bed, but it is also compatible with cloud environments.

Cloud providers sign a contract with their customers, called Service Level Agreement, that specify and ensure that a minimum service level is maintained. It guarantees levels of reliability, availability, and responsiveness to systems and applications. When describing the IaaS services offered to the customers, the providers are particularly specific in terms of virtual RAM, virtual CPU, and network bandwidth. Still, they tend to hide the value of the network delay in the infrastructure. The cloud providers usually indicate that the datacenters composing the infrastructure are connected via high speed, optical networks, but there are no specific values. The applications are usually developed with a monolithic architecture, i.e., built as a single unit with a database layer, a back end layer, and a client layer. With a monolithic strategy, it is difficult to scale such applications. This is why the enterprises are moving from monolithic to microservice applications. This can lead to synchronization problems since the application is not running on a single machine but on multiple machines. If the delay between the instances is too high, the application cannot work properly. Furthermore, the developers can decide to provide regional resiliency. This means that the application will be deployed in 2 different regions. If the provider does not share the network delay information, how can developers decide where to deploy the application if it is delay-sensitive ?

We propose a Command Line Interface (CLI) tool to test the delay stability in the cloud infrastructure. In particular, we analyze Amazon AWS. The tool is open-source and publicly available. It is written in Python and uses the latest libraries to create and manage virtual machines inside a cloud infrastructure. The idea behind the tool is to perform synchronized trace-routes between different instances in different regions and availability zones. It can perform experiments between two or more regions (multiregional scenario). Such tests are made to verify the delay stability between different regions. We expect in this case that the delay varies depending on the distance between the regions. The other tests are performed within a single region (regional experiment). The experiments monitor the delay in a specific region between two or more Availability Zones.

Even with the use of the best resources and equipment, networks may fail. In the context of SDN and NFV networks, efficient network algorithms considered too hard to be implemented in a legacy network now can exploit the benefits of a centralized controller with the SDN paradigm. The networks are failure-prone, and, usually, failures tend to be correlated. To design this correlation, we consider the network dimensioning problem with protection against the Shared Risk Link Group (SRLG). An SRLG is a set of links inside the network that can

simultaneously fail at any time. With SRLG, it is possible to model multiple link failure, single link failure (e.g., considering the set composed by a single link), or node failure (e.g., considering the set composed by the link connected to a specific switch). Let's imagine to design a solution that allows rerouting all the flows in the network optimally in each failure situation. This solution can lead to considerable savings in terms of CAPEX since the providers reduce the overprovision of the network infrastructure drastically in order to deal with failures. For this reason, we consider a protection technique called *unrestricted flow reconfiguration*, also known as *global rerouting*. In each of the possible failure situations (so for each SRLG), a new set of routes is computed for each demand. This makes the protection method *bandwidth-optimal*. However, this also means that each failure may result in a completely different routing for the demands. In standard networks, it is impossible to implement a solution that can potentially modify the route in each switch in the network due to the large number of rules to install on the network devices and hence signaling overhead. However, with the introduction of SDN, the logic of the solution can be computed in the centralized controller.

1.2 Challenges

1.2.1 Network Emulation

Emulating a network is often required before deploying it in a real environment. As networks are growing in terms of size and complexity, emulating them becomes more and more difficult. In the context of SDN, Mininet is the primary emulator used by the community. Since networks within the NFV paradigm require more resources, emulating a network in a single machine can become tricky. If the emulation tools do not provide a way to scale horizontally, the single physical machine can exceed the resource limit in terms of CPU or RAM, and the emulation can return results that are not aligned with the expected ones. Multiple works are proposed to share the emulation in a distributed infrastructure like MaxiNet and Mininet Cluster Edition, but these tools have some limitations. They do not consider, for instance, the physical topology and the capabilities of the physical infrastructure. We aim to address such limitations with our tool called Distrinet, and the new placement algorithms we propose which are natively implemented on it. The main challenge is to make the tool compatible with private clusters or public cloud platforms. The tool has to provide a virtual connection between two or more virtual nodes. This is a simple task to perform in private clusters since containerization technologies automatically manage overlay networks with multicast tunnels to connect the instances. All the cloud providers do not allow multicast addresses in their virtual network when using IaaS services. The emulation should be trustable, which means that the emulation results should be in line with the results in a real environment. The challenge is to distribute the emulation evenly in the testing infrastructure, since only a single host or a single physical link overloaded

can lead to a wrong result.

1.2.2 Cloud Testing

In recent years, there has been a transition between classical computing and cloud computing. The benefits of cloud computing convinced many IT companies to migrate their applications to, and build applications natively on, the cloud. Even ISPs are moving some of their services in the cloud, creating then a hybrid environment for their operations. Some applications are globally distributed to provide high availability. Distributing geographically the applications increases their robustness, since the services remain available in case of natural disasters or blackouts affecting a particular region. For delay-sensitive applications, the global distribution can be an issue since the instances are placed in two different regions. Even if the cloud provides a high-speed network, the delay can be in the order of milliseconds. Before starting the real deployment of services in the cloud, it is thus necessarily to know if the delay is stable enough to support the application requirements. The challenge is thus to build a simple tool to monitor the network's stability in the cloud infrastructure.

1.2.3 Failure Recovery

As introduced before, networks are not perfect, and they can experience failures, even multiple ones at the same moment. We focus on failure recovery strategies, and we consider a protection technique called *unrestricted flow reconfiguration* (also known as *global rerouting*). For each possible failure situation, a new set of backup routes is chosen to forward the traffic, considering the new status of the network. With this technique, it is possible to implement an optimal bandwidth-efficient protection method. As implementing global rerouting in legacy networks is challenging due to the massive amount of signaling events created by the system during failure situations, SDN eases the deployment of this approach. The centralized controller can be used to manage the network in case of failures in an optimal way. Another challenge is to compute the path of the demands in the network in an efficient way, since the problem is difficult to solve even for small instances.

1.3 Contributions

This work lies in the new and growing SDN/NFV and cloud computing paradigms of the recent years. We provide a high-level overview of the SDN framework and architecture, and we describe how it can be integrated with Network Function Virtualization (NFV) to procure Service Function Chaining (SFC), also known as Network Service. Our first contribution is *Distrinet*, a distributed network emulator that extends Mininet in order to run on multiple physical machines and provide higher isolation. We briefly explain the limitation of the emulators available on the market, then we run multiple tests showing that Distrinet

returns results aligned with the expected ones. The comparison is made on different architectural and tool choices in order to build a tool compatible with general Linux clusters (e.g., a Linux test-bed) and public cloud infrastructures (e.g., Amazon Web Services). Another factor that we consider and study is how to correctly distribute the experiment in a distributed environment. We propose three new algorithms and we compared them with the ones proposed in the main tools already available. We extensively compare the algorithms using more than 75,000 experiments over heterogeneous and homogeneous networks. The result shows that our algorithms return a feasible solution for almost all the network types. We also made an extensive study of the performance degradation due to an overcommitment of the physical machines [Len+21],[Di +19a], [Di +19b],[Di +19d] [Di +19e], [Di +21c], [Di +21b].

Since many organizations are moving their applications to the cloud, for our second contribution, we provide a tool that measures and analyzes the cloud network delay. This tool performs experiments for multiregional and regional environments. It automates all the processes from the creation of the cloud environment and the deployment of the virtual instances to the configuration of trace-route in the instances. The results are automatically retrieved and a map is created and plotted with all the experiment info [Di +21a].

Our last contribution is a failure recovery strategy for SDN/NFV embedded networks. Networks, in general, are failure-prone, and they can experience multiple link failures at the same time. These failures can be related to each other, and the network providers usually have all data enabling them to predict which links may fail. We propose a global rerouting strategy in which, for each failure situation, there is a new routing of all demands in the network using the SDN technology. We test these solutions on different networks, and the global rerouting solution consumes, on average, 40% less network bandwidth compared to a dedicated path protection strategy. We also provide an emulation framework with Mininet and OpenDayLight to validate the solution in a testing environment [Tom+19a], [Tom+19b], [Tom+21].

List Of Publications

International Journal

[Di +21c] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. “Distrinet: a Mininet Implementation for the Cloud”. In: *ACM Computer Communication Review* (2021) (cit. on p. 13).

[Tom+21] A. Tomassilli, G. Di Lena, F. Giroire, I. Tahiri, D. Saucez, S. Perennes, T. Turletti, R. Sadykov, F. Vanderbeck, and C. Lac. “Design of Robust Programmable Networks with Bandwidth-optimal Failure Recovery Scheme”. In: *Computer Networks* (2021). Ed. by Elsevier (cit. on p. 13).

Submitted to International Journals

[Di +21b] G. Di Lena, A. Tomassilli, F. Giroire, D. Saucez, T. Turletti, and C. Lac. “Placement Module for Distributed SDN/NFV Network Emulation”. Submitted to Computer Networks Journal. 2021 (cit. on p. 13).

International Conferences

[Di +19a] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac. “Mininet on steroids: exploiting the cloud for Mininet performance”. In: *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. 2019, pp. 1–3. DOI: 10.1109/CloudNet47604.2019.9064129 (cit. on p. 13).

[Len+21] Giuseppe Di Lena, Andrea Tomassilli, Frédéric Giroire, Damien Saucez, Thierry Turletti, and Chidung Lac. “A Right Placement Makes a Happy Emulator: a Placement Module for Distributed SDN/NFV Emulation”. Proceedings of IEEE International Conference on Communications (ICC). 2021 (cit. on p. 13).

[Tom+19b] A. Tomassilli, G. D. Lena, F. Giroire, I. Tahiri, D. Saucez, S. Perennes, T. Turletti, R. Sadykov, F. Vanderbeck, and C. Lac. “Bandwidth-optimal Failure Recovery Scheme for Robust Programmable Networks”. In: *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. 2019, pp. 1–6. DOI: 10.1109/CloudNet47604.2019.9064126 (cit. on p. 13).

National Conferences

[Di +19d] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. “Trust your SDN/NFV experiments with Distrinet”. In: *Journées Cloud* (2019) (cit. on p. 13).

Submitted to International Conferences

[Di +21a] G. Di Lena, F. Giroire, T. Turletti, and C. Lac. “CloudTrace Demo: Tracing Cloud Network Delay”. Submitted to IEEE International Conference on Network Softwarization (NetSoft), Demo session. 2021 (cit. on p. 13).

Demos, Posters, Extended Abstract

[Di +19b] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. "Demo Proposal - Distrinet: A Mininet Implementation for the Cloud". In: *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '19. Orlando, FL, USA: Association for Computing Machinery, 2019, pp. 82–83. ISBN: 9781450370066. DOI: 10.1145/3360468.3368186. URL: https://doi.org/10.1145/3360468.3368186 (cit. on p. 13).

[Di +19e] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, Chidung Lac, and Walid Dabbous. *Distributed Network Experiment Emulation*. GEFI 19 - Global Experimentation for Future Internet - Workshop. Nov. 2019. URL: https://hal.inria.fr/hal-02359801 (cit. on p. 13).

[Tom+19a] A. Tomassilli, G. Di Lena, F. Giroire, I. Tahiri, D. Saucez, S. Perennes, T. Turletti, R. Sadykov, F. Vanderbeck, and C. Lac. "Poster: design of survivable SDN/NFV-enabled networks with bandwidth-optimal failure recovery". In: *2019 IFIP Networking Conference (IFIP Networking)*. 2019, pp. 1–2 (cit. on p. 13).

1.4 Outline

The Thesis continues as follows. In Chapter 2, we provide the concepts, and background necessary for the rest of the work. In this chapter, we introduce the virtualization and containerization concepts and how they are related to NFV and SDN. We also provide an overview of the test-beds and the cloud environments, explain why they are important, and why more and more organizations are heavily investing on it.

In Chapter 3, we present our first contribution, introducing the main distributed network emulators present on the market, and comparing them with our proposition Distrinet. This chapter analyzes different technical aspects of Distrinet, which technologies are used to distribute the network experiments and what are the advantages that Distrinet has, compared to the other tools.

In Chapter 4, we continue our study of the network emulation tools by analyzing more in depth the algorithms used to share the emulation in a distributed environment. We introduce the Virtual Network Embedding Problem and the algorithms used in the main emulators, and we propose three different algorithms for Distrinet. A comparison with the algorithms proposed by the other tools shows that our algorithms outperform them in every scenario.

In Chapter 5, we focus on cloud computing and we discuss the advantages that it brings to the organizations. We present a tool for measuring the network delay in different deployment scenarios, in a single region or in a multiregional deployment. The tool is useful for all the persons who are planning to move delay-sensitive applications from on-premises environment to the cloud.

In Chapter 6, we focus on SDN/NFV embedded networks, and we propose in particular a global rerouting strategy to recover all the flows in the network in case of single or multiple link failures. The chapter deeply analyzes the optimization approaches used like ILP and Column Generation techniques. We finally experiment the global rerouting approach, comparing it with the classical dedicated path protection, and showing that global rerouting consumes on average 40% less bandwidth in multiple scenarios.

We conclude with Chapter 7 where we draw our conclusions, summarizing all the problems faced and the contributions we present. A short discussion is finally proposed on how the work can be improved and pursued.

Chapter 2

Background

2.1 Virtualization

Virtualization is one of the main building blocks of cloud computing and Service Function Chaining. Virtualization includes all the processes to be performed in order to create a software version of a physical environment, e.g., a virtual host or a virtual network. The main component of virtualization technologies is the *hypervisor*, also known as *Virtual Machine Monitor (VMM)*. With the new isolation functions present in the last versions of the Linux kernel, containerization solutions have been developed. Containers provide similar isolated environments without the hypervisor.

The hypervisor typically runs above the physical host and selects the available resources on the host for allocating them to the virtual environment. There are different types of virtualization. A first approach is to divide them into standard virtualization (hypervisor-based) and container-based virtualization.

2.1.1 Hypervisor-Based Virtualization

Multiple software packages can provide a virtualized environment (e.g., Virtual Box from Oracle, Hyper-V from Microsoft, etc.). The virtual environment running the guest Operating System (OS) is called *Virtual Machine (VM)*. There are multiple advantages for using VMs, the main one being that it is possible to use different OSs in a single machine simultaneously.

The hypervisor can set the maximum amount of resources that a VM can use, and it is also possible to connect multiple VMs in the same virtual network. The hypervisor is responsible for creating and configuring the virtual network interfaces and the virtual switches or the virtual bridges. The users can create a VM for each OS that they want to deploy. Fig. 2.2 showcases a classical example of one physical host running three different virtual machines. Each virtual machine can run a different OS (e.g., Windows 7, Ubuntu, Debian). Each OS can execute its own applications. The first difference between the different technologies stands for where the hypervisor is running. In Bare-Metal

virtualization (Fig. 2.1), the hypervisor is running directly on the physical host, while in hosted virtualization (Fig. 2.2) the hypervisor runs on top of the OS.

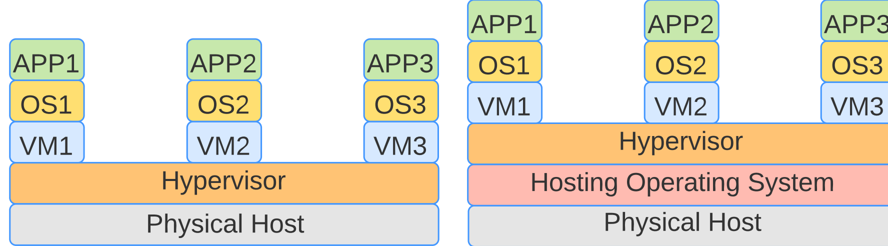


Figure 2.1: Bare-metal

Figure 2.2: Hosted

Three types of standard virtualization are possible.

Full Virtualization

In Full Virtualization, the guest OS is not aware that it is in a virtualized environment. Full Virtualization represents the first generation of solutions that provide a virtual environment. The first company that provided a server virtualization solution was IBM in 1966.

The host OS completely virtualizes the hardware. The guest OS can execute commands just as for a regular physical host, but all the hardware is created/simulated by the hypervisor. In this case, the binary files of the guest OS remain unchanged. The guest OS runs as a user level process, and does not have the same level of privileges as the OS that is hosting it on a physical machine.

If the guest OS is not modified, it can try to execute some privileged instructions available only in the hosting OS (Fig. 2.3). In this case, the hypervisor catches the privileged instruction and replaces the non-virtualizable instructions with a new sequence of instructions that have the intended effect on the virtual hardware, while the user-level application is directly executed on the physical machine to improve the performances (Fig. 2.4).

Paravirtualization

In Paravirtualization, the guest OS is aware that it is hosted in a virtual environment, so it has an installed driver specifically for the virtualization environment in which it is running. In this case, the source code of the guest OS is modified. This allows to perform optimizations in the virtualization process. The guest OS does not perform the privileged instructions like in Full Virtualization, but it executes the instructions provided by the hypervisor layer. The hypervisor layer can then access to the real resources in the underlying hardware, exploiting fully this capability (Fig. 2.5).

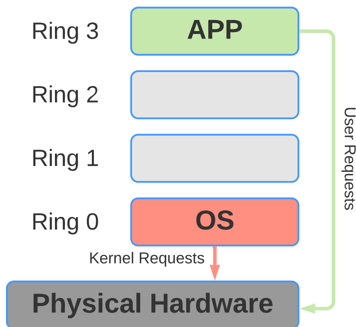


Figure 2.3: OS in physical host

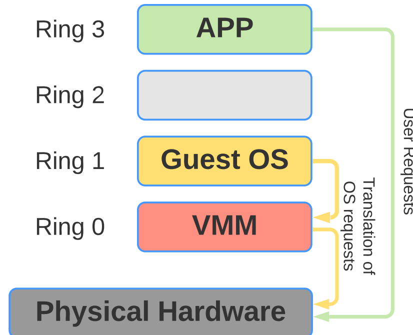


Figure 2.4: Full virtualization

Hardware Assisted Virtualization

While virtualization adoption was growing, CPU manufacturers developed new features to simplify the virtualization process. The Intel Virtualization Technology (VT-x) and the AMD-V Hardware-Assisted virtualization have extended the full virtualization technology. The microprocessor architecture has special instructions to assist the hardware virtualization. The guest OS can perform more actions natively. For example, instructions might allow a virtual context so that the guest can execute privileged instructions directly on the processor without affecting the host (Fig. 2.6).

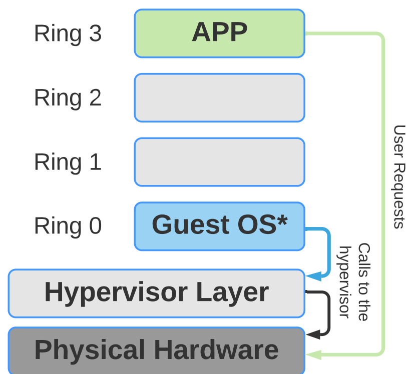


Figure 2.5: Paravirtualization

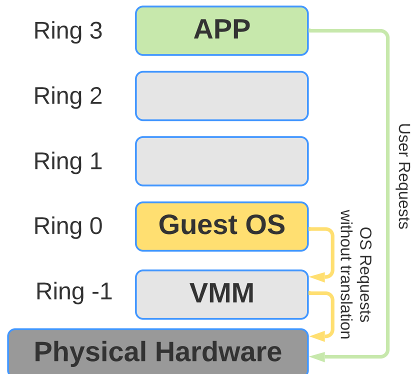


Figure 2.6: Hardware assisted virtualization

2.1.2 Container Virtualization

The container is often referred to as a light Virtual Machine, but there are significant differences between a container and a VM. The main difference is that a container does not have a hypervisor layer. Containers are built on top of three Linux Kernel features: *namespaces*, *Cgroups* and *chroot*.

The main advantage of containers is the absence of overhead in terms of performances, while with virtualization, there is the hypervisor overhead. One of the main disadvantages is that containers are available only for Linux OS. It is not possible to have containers natively on Windows or Mac OS. However, it is possible to install Docker on these systems, but the installation process creates a virtual machine with Linux kernel in the system. In such case, the containers run on top of a VM, and not directly on the system.

Since the containers are actually running Linux Kernel, it is not possible to virtualize containers with non-Linux OS. Namespaces is a Linux function that allows the kernel to assign to a set of processes an isolated environment. If a set of processes has a namespace assigned, they can only see the specified namespace's resource. In contrast, a different set of processes can access a different set of resources.

There are multiple types of namespaces:

- **User** - User namespace (UID) provides user isolation and identification across multiple sets of processes. This namespace is a critical security feature.
- **IPC** - Standing for Inter-Process Communications, this namespace isolates the system resources from a process. Processes in the same IPC namespace have visibility of each other, allowing interprocess communication between them.
- **UTS** - This namespace allows the processes to have different hosts and domain names in the same environment.
- **Mount** - With this namespace, the system can control the mount points that are visible to a container.
- **Network** - It virtualizes the entire network stack.
- **PID** - This namespace provides to the processes an independent set of process IDs.

Control groups (Cgroups) are often referred to as the seventh namespace, but instead of creating an isolated environment, they isolate the resources. Cgroups are used to limit the resource usage of a set of processes. There are many Cgroups available. For example, the *CPU* Cgroup allows to monitor the CPU usage of a group of processes: the *cpuset* forces the process to run in one or a set of dedicated CPU. There are also Cgroups for memory and network devices. For example, *memory* Cgroup makes sure that the memory usage assigned to

a set of processes does not increase over a certain limit, while *net_prio* Cgroup provides a system to prioritize the network traffic of different sets of processes.

The last function to analyze is the *chroot* command provided in the Linux environment. Chroot (contraction for change root) changes the apparent root directory for the current running process and its children. Figs. 2.7 and 2.8 present through a high-level overview the differences between VMs and containers. As we can see, the VM has to virtualize the entire guest OS stack, while the container has to create just the namespace and the Cgroup to separate the processes of each guest OS. The container images are lighter as there is no need to create the entire kernel since it is using the kernel running on the physical host. Each container image needs to have a small part of the OS in order to work in the host. As indicated earlier, it is only possible to create Linux containers since the other OSs do not have namespaces and Cgroups.

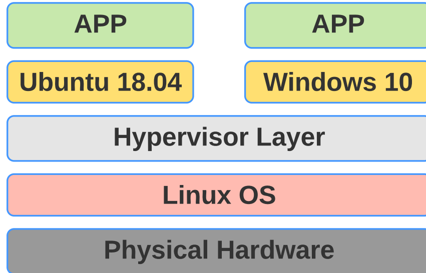


Figure 2.7: Virtual machine

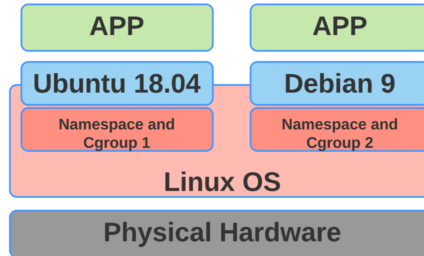


Figure 2.8: Container

2.2 Network Function Virtualization (NFV)

The virtualization concept can be used, not only for standard OS, but also for network equipments. Networks are made of different types of appliances (e.g., firewalls, routers, load balancers, intrusion detection systems, etc.). Each organization can use hundreds or thousands of these devices, and in general, there is a hardware device associated to a single function. These network devices usually run proprietary software, so it is nearly impossible to create additional functionality. Moreover, they need to be maintained by specialized technicians. It is also complicated to change the topology configuration when the network is already deployed since the network has to be physically reconfigured. This hardware is usually expensive to purchase and to maintain since it needs to be monitored, upgraded, and configured.

NFV aims to solve all these challenges, changing the way to deploy and manage the network. NFV applies the virtualization concept for all the network devices. Instead of having a physical device for each function, the hardware can be virtualized on top of COTS (Commercial Off-The-Shelf) equipment, with the use of specialized hardware to optimize the virtualization of some network

functions.

By decoupling network functions from their underlying hardware, NFV provides more flexible networking functionalities. Indeed, the virtualized functions can be created on-demand, without the installation overhead and they can be reconfigured remotely, without human intervention. Network operators can dynamically deploy network functions faster in the same underlying physical infrastructure. This approach also brings benefits in terms of cost reductions. Fig. 2.9 shows the idea behind the NFV concept. The functions can be virtualized with hypervisor-based or container-based virtualization.

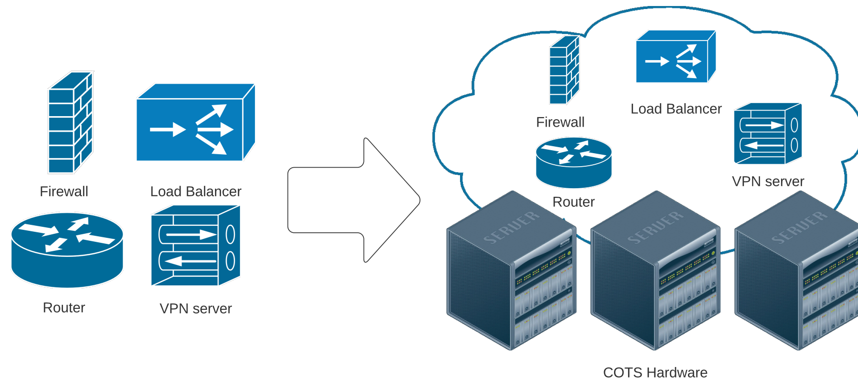


Figure 2.9: NFV

2.3 Software Defined Networking (SDN)

SDN is a network paradigm proposed to create networks in a more flexible way and which are easier to manage. In traditional network routing, the routers and the switches work on the control plane (the routing protocol identifies the best paths that satisfy the routing tables) and the forwarding plane (the router, looking at the routing table, understands which is the outgoing port to use for sending the packets received). This solution is very powerful and widely used over the years. In some cases, this approach shows its limits. For some scenarios, the shortest path is not the best since some paths could be overloaded. It would be better to take a routing decision based on all the switches and the status of the links in the network. Many mechanisms that have been introduced for forcing traffic on paths do not consider the number of hops as the only criteria of choice (i.e., MPLS). However, it is impossible to make decisions based on all the network status since a single switch or router just knows a part of the network. Fig. 2.10 shows a classical network with controller and data plane integrated into each switch, while in Fig. 2.11, we have represented the SDN approach. The idea is simple: separate the controller and the data plane, and

centralize the controller plane. With this approach, the controller has a global overview of the network and can perform routing decisions based on the full network topology and the different metrics in the switches and the links. For example, the controller can decide to reroute flows in a different path if there is an overloaded link.

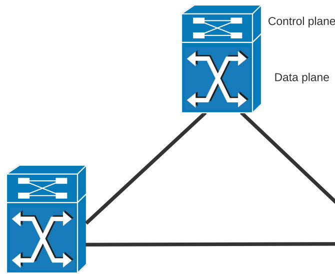


Figure 2.10: Standard network

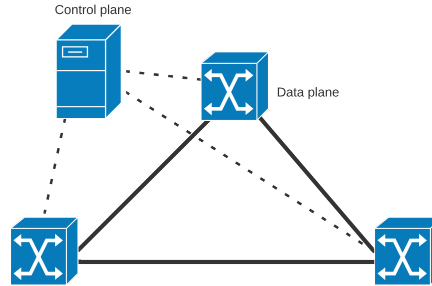


Figure 2.11: SDN network

2.3.1 SDN Architecture

An SDN architecture is composed of 3 main layers: *infrastructure layer*, *control layer*, and *application layer*.

The infrastructure layer is made of networking equipment that form the underlying network. The main objective of this layer is to forward network traffic. The architecture is formed of a set of different network switches and routers. These physical devices are designed with highly efficient and programmable packet forwarding mechanisms without any software to make autonomous decisions.

The control layer is the core of the logic in the SDN architecture. This layer fetches and maintains different network information, state details, topology, packet statistics, to decide where to route the traffic in the infrastructure layer. In general deployment, this layer is composed of a centralized controller. Since the SDN controller manages networks, it must have control logic for basic network use-cases like switching (i.e., routing, L2 VPN, L3 VPN, firewall security, etc.). Examples of centralized controllers are Pox, Ryu, OpenDayLight, etc. Several networking vendors and open-source communities are working on implementing these use-cases in their SDN controllers. The controller provides abstractions, services, and standard APIs to developers. Once they get implemented, these services expose their APIs (typically REST-based) to the application layer. Network administrators do not have to configure all the switches manually, since they can use applications on top of SDN controllers to configure, manage, and monitor the underlying network. The control layer lies in the middle, and it exposes two types of interfaces, Northbound and Southbound.

- **Northbound interface:** it is used for communication with the applica-

tion layer, and is usually realized through the REST APIs of SDN controllers.

- **Southbound interface:** it is used for the connection with the infrastructure layer. The network elements are designed to work with protocols like Openflow, Netconf, P4, etc.

The application layer includes applications that allow network operators to develop their high-level policies for the network. This layer is responsible for managing and analyzing all the information about network topology, network state, network statistics, etc. Multiple applications can be created to work in this layer. In general, there is no restriction on the type of language or tool that the developers can use to provide, for example, optimization techniques and failure recovery strategies, network automation, and security policies. Such SDN applications can provide various end-to-end solutions for real-world enterprise and datacenter networks.

Fig. 2.12 shows these different layers.

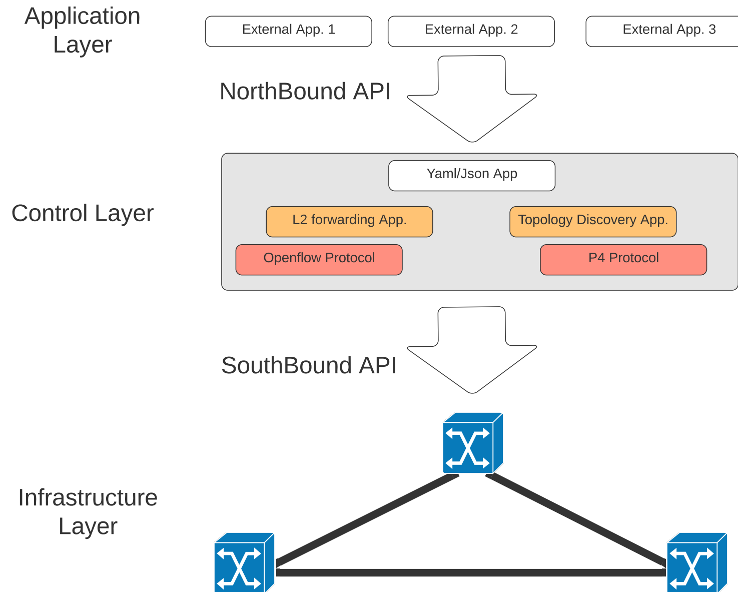


Figure 2.12: SDN architecture

2.3.2 Openflow

As introduced above, Openflow can be used for interfacing the SDN controller with programmable switches. Openflow describes a standard specification for

communication between the SDN controller and network equipment (typically switches). The specification is provided by the Open Networking Foundation (ONF) and is evolving with support for various requirements of current networking. It allows routing decisions to be taken by SDN controllers and lets forwarding rules, as well as security rules, being pushed on switches in the underlying network. SDN controllers and switches need to implement Openflow specifications in order to understand the common language of Openflow messages. The SDN controller pushes rules into switches to make decisions, so when traffic goes through the switches, they can apply the installed rules. Switches need to save these rules in the Openflow table. The protocol does not specify the maximum amount of tables that a switch is able to handle. Each vendor provides various switches with different memory sizes. The Openflow rules can also be referred to as *flows*, and they are stored in *flow tables*. Each flow contains four types of information:

- **Match fields:** they define the matching criteria (source/destination IP or MAC address, VLAN tag, protocol type, ARP fields, ICMP fields, etc.). A flow can have multiple matches.
- **Priority:** it is often referred inside the match field. When the switch has installed in the Openflow table two or more different rules that match a flow, the priority defines which rule should be applied. If the match's flows have the same priority, the protocol does not specify how to choose the rule. Each vendor can implement a different policy.
- **Actions:** they define what to do with a packet if it matches the flow. Actions can be DROP, forward on a specified output port, append, modify or remove a header on the packet (modify the destination IP, push/pop VLAN ID, push/pop MPLS label, etc.).
- **Counters:** they are used to keep track of the number of packets and the quantity of traffic which matched the flow.

An Openflow connection is a setup between a switch and the controller. The controller listens by default on port 6653, and the connection runs on TCP. This connection is used to communicate with the switch and install, modify, or remove the rules, while the switches send to the controller the statistics of the flows received on the data port. If the controller runs and accepts correctly the connection on the default port, the switches can initiate the connection. The switch sends a connection request (a HELLO packet) to the controller. If the controller accepts the switch, a connection is initiated, and the SDN network is correctly set.

2.4 Service Function Chaining (SFC)

SFC can be described as a combination of NFV and SDN technologies. SFC is the instantiation of multiple network functions to form an end-to-end chain,

creating a Service Function Path (SFP). Services are built to satisfy a particular business needs and must satisfy policies that define operational characteristics and access control/security. A SFC defines the required functions and the corresponding order that must be applied to the packets belonging to a specific data flow. Thanks to the dynamic function provisioning of NFV and the centralized control of SDN, a SDN/NFV network is able to simplify the service chain deployment and provisioning by making the process easier and cheaper, enabling a flexible and dynamic deployment of network functions.

The European Telecommunications Standards Institute (ETSI) defines SFC as a **Network Service**, an offer provided by an operator that is delivered using one or more service functions. An example of SFC can be found in Fig. 2.13, where the traffic should follow the order firewall, load balancer and Intrusion Detection System (IDS). In traditional networks, all these functions run on dedicated physical boxes. The adoption of SFC simplifies this concept, since the functions can be created, destroyed or modified on demand on general purpose servers. The network intelligence can be easily managed by an SDN controller, and correctly interfaced with the underlying infrastructure architecture.

The SDN part is responsible to correctly forward the traffic to the correct VNFs (Virtual Network Functions), improving security and isolation of the network. The NFV technology is responsible for creating and managing the VNFs, monitoring and handling these VNFs in case of errors or failures, e.g., link failures. In this work, VNF is used in place of Service Function (SF)

Figure 2.13: Example of SFC

2.5 Test-beds

With the ever growing complexity of networks, researchers have to rely on test-beds to be able to fully assess the quality of their propositions. Emulation is an essential step in the evaluation cycle of network applications and protocols. It provides a fully controlled and duplicable environment to test real protocols/applications in a reproducible way, without any modification.

It is interesting for test-bed infrastructures to provide network emulation capabilities in their environments in order to run large-scale scenarios with realistic and accurate results. Test-beds, essentials for researchers, provide a controlled environment which avoids any interference from the external environment. In our work, we heavily rely on Grid'5000 for our experiments. Federated platforms such as Fed4Fire test-beds provide a uniform interface for running emulation scenarios, including hybrid ones that involve more than one test-bed.

Researchers need an experimental infrastructure which is extremely customizable and controlled. For example, they cannot rely on cloud infrastructure since the infrastructure is shared by different users, and they do not have control on the underlying layers of virtualization and, more importantly, they have no control on the physical servers and networking layer. Actually, experiments cannot be trustable in a shared and uncontrolled environment.

Test-beds are also the key components for reproducibility, since experiments can run on a huge amount of different types of resources, and the results can be affected by the different performances of the machines. Providing researchers a public environment where they can run the experiments brings the possibility to easily reproduce the results in different experiments. The researchers only need to create scripts or pieces of software that automatically deploy the resources they use for the experiment, and document these scripts with instructions to follow in order to reproduce their experiments, and possibly retrieve similar results.

2.6 Cloud Environments

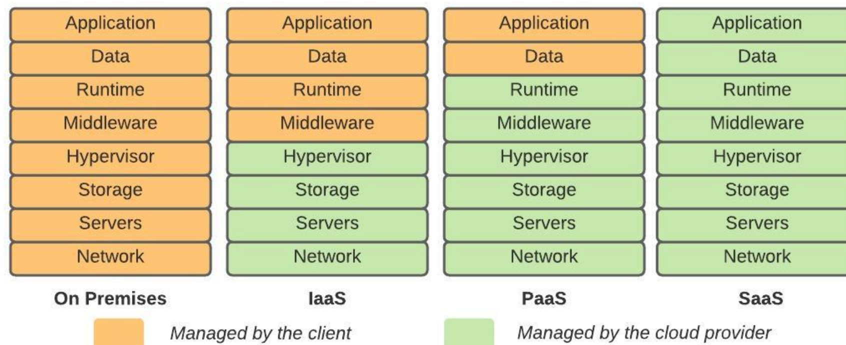


Figure 2.14: Cloud models

Cloud computing is based on the on-demand delivery of compute, database storage, networking capabilities, applications, and other IT resources through a cloud services platform via the Internet with pay-as-you-go pricing. It heavily relies on virtualization and network isolation technologies. Cloud computing was popularized with Amazon, via its Elastic Compute Cloud service proposed in 2006. There are a lot of reasons why cloud computing is going so popular. It reduces the CAPEX for variable workloads. Instead of having to invest heavily in datacenters and servers before knowing how to use them, the clients only pay when they consume computing resources, and only for the quantity of resources they consume. There are also benefits related to the massive economy of scale,

since all the services that the cloud provider offers are usually on-demand. The clients can stop guessing about the capacity needed, avoiding then to purchase too much or too few resources, i.e., wasting money or facing downtime due to infrastructure upgrade. The cloud can scale with their business needs, with no long term contracts. The use of virtual infrastructure thus increases speed and agility.

Cloud providers propose three different models for cloud computing (IaaS, PaaS, SaaS) which will described later. They also provide a new type of design called serverless architecture. It scales infinitely on demand. As the clients do not have to spend money running and maintaining datacenters, their organization can focus only on the problems to be solved, without having to manage the infrastructure which is maintained and updated by the cloud provider.

The last aspect to take into account is the global scaling. A few years ago, it was not possible for a small organization to provide a global service immediately, since an infrastructure has to be setup around the world with different datacenters, causing huge costs. With cloud computing, the clients can easily deploy their applications in multiple regions around the world through a few clicks. This allows the clients to provide low latency and a better experience for their customers at minimal costs.

Cloud computing provides convenient, on demand access to a potentially unlimited pool of computing resource. As mentioned previously, cloud providers offer three service models (Fig. 2.14):

- **On Premises**: the user is responsible for maintaining the entire infrastructure, the user has to manage the physical servers and the network (configuration and upgrade). The user is also responsible for designing failure recovery strategies in case of server or network failures, and backup of the data. All these management tasks have a huge cost, and if the user has to scale the infrastructure globally around the world, the only way to do it is to deploy new datacenters in the regions. For all these reasons, companies should carefully consider cloud platforms.

- **IaaS** (Infrastructure as a Service): the user can manage the virtual or physical servers inside the cloud provider infrastructure. As shown in the figure, the cloud provider is responsible to manage the physical infrastructure, and to provide a virtualization layer isolating each user environment. The provider is responsible to keep the data and execution of the virtual instances from access of unauthorized users. The figure also shows that the provider manages the physical servers and the network components of the infrastructure. The provider is also responsible for the maintenance of the OS and the hypervisor. The user has never access to the hypervisor layer, since resources are shared between the users. The client is responsible for all other layers in the application stack (e.g., Amazon EC2, Microsoft Azure), i.e., the client has to maintain the guest OS and the applications running on it. The user is also responsible of the security of the virtual network and the management of the virtual firewall, as well as the keys to access the instances.

- **PaaS** (Platform as a Service): the cloud provider takes care of all the layers that manage the IaaS, and it is also responsible for the guest OS and the platform hosting it (e.g., Google App Engine, GitHub). The client just uses the platform without worrying about the resources and policies needed to test its applications.

- **SaaS** (Software as a Service): the provider is responsible for the management of all layers of the application stack. The client uses the application on the application stack without any responsibility (e.g., in Google Docs).

There are also three types of cloud computing deployments:

- Public cloud: the cloud provider is public and everyone has access to its services (e.g., Amazon Web Services, Microsoft Azure, Google Cloud).

- Hybrid cloud: the client can decide to deploy its infrastructure partly on the public cloud, and partly on a private cloud.

- Private cloud: some big players in the market can decide to build their own cloud infrastructure. There are different choices for this type of scenario, e.g., using OpenStack or VMware solutions.

Chapter 3

Distrinet

3.1 Introduction

Modern networks became so complex and implementation-dependent that it is now impossible to solely rely on models or simulations to study them. On one hand, models are particularly interesting to determine the limits of a system, potentially at very large scale or to reason in an abstract way to conceive efficient networks. On the other hand, simulations are pretty handy for studying the general behavior of a network or for getting high confidence about the applicability of new concepts. However, these methods do not faithfully account for implementation details. To this end, emulation is more and more used to evaluate new networking ideas. The advantage of emulation is that the exact same code as the production one can be used and tested in rather realistic cases helping to understand fine-grained interactions between software and hardware. However, emulation is not the reality and it often needs to deal with scalability issues for large and resource-intensive experiments. As it relies on real software that implements all the details of the system to emulate, all three methods are used and complement each other.

When it comes to Software Defined Networking (SDN), `Mininet` [LHM10] is by far the most popular emulator. The success of `Mininet` comes from its ability to emulate potentially large networks on one machine, thanks to lightweight virtualization techniques and a simple yet powerful API. `Mininet` was designed to run on one single machine, which can be a limiting factor for experiments with heavy memory or processing capacity needs. A solution to tackle this issue is to distribute the emulation over multiple machines. However, `Mininet` single-machine design assumes that all resources are shared and directly accessible from each component of an experiment. Unfortunately, when multiple machines are used to run an experiment, this assumption does not hold anymore and the way `Mininet` is implemented has to be revised.

In this chapter, we present Distrinet [Di +19c], an extension of `Mininet` implemented to allow distributed `Mininet` experiments for leveraging resources

of multiple machines when needed. Mininet is used by a large community ranging from students to researchers and network professionals. This success of Mininet comes from the simplicity of the tool. It can work directly on a laptop and its installation is easy, its Python API is simple yet powerful, and it can be used for complex scenarios with the same ease as for basic tests. The challenge is to extend Mininet in such a way that these conditions still hold, while being distributed over multiple machines. Distrinet allows applications to run in isolated environments by using LXC to emulate virtual nodes and switches, avoiding the burden of virtual machine hypervisors. Distrinet also creates virtual links with bandwidth limits without any effort from the user.

Mininet and its associated API have proven to be the right tools for the community. This is why *Distrinet* is a direct extension of Mininet, whose implementation leverages the Mininet core in which experiments are defined with the Mininet idiom (e.g., API or CLI) and are transparently deployed on several machines when needed.

It is important to mention that Distrinet shares the same API as Mininet. Mininet programs can thus be reused with minimal - or even without any - changes in Distrinet, but with a higher degree of confidence on the results in case of resource intensive experiments. Our main contributions to reach this objective can be summarized as follows.

- **Compatibility with Mininet.** Mininet experiments are compatible with Distrinet, either using the Mininet API (with additional parameters, e.g., the IPs of the physical machines to use or the name of the images to deploy) or with the Mininet Command Line Interface (i.e., mn).

- **Architecture.** Distrinet is compatible with a large variety of infrastructures: it can be installed on a single computer, a Linux cluster, or the Amazon EC2 cloud. Distrinet relies on prominent open-source projects (e.g., Ansible and LXD) to set up the physical environment, manage experiments, and guarantee isolation.

- **Comparison with other tools.** Comparisons with the main emulators available for SDN/NFV networks, showing that our tool *handles more efficiently link bandwidth limitation* which is a *fundamental basic brick* of network emulation.

- **Flexibility**. Thanks to the usage of LXC (LinuX Containers), Distrinet allows to run VNFs or generic containers on the emulated topology composed of virtual switches and hosts. Each virtual node is properly isolated (with a proper init or support for services, daemons, Syslog, etc.) with LXC, so the user can treat them almost like a real Virtual Machine.

3.2 Related Work

One of the main problems, when it comes to testing new network protocols, is to find the right environment that provides researchers good flexibility and

results which are as closest as possible to the outcome in real conditions. Depending on the situation, three different approaches can be used: simulation, emulation or experimentation on test-beds. These approaches are complementary and, usually, several of them are required before the real deployment of a new solution.

Emulation allows to test the performances of real applications over a virtual network. A first frequently used tool to emulate networks is the Open vSwitch (OVS) software switch [ovs20]. To build a virtual network, virtual switches (vSwitches) can be connected with virtual interfaces, through GRE or VXLAN tunnels. To emulate virtual hosts (vHosts), one can use containerization tools (e.g., LXC [Can19] or Docker [Mer14]) or full virtualization tools (e.g., Virtual Box [Wat08]). An example of general purpose testbed is FITS (Future Internet Testbed with Security [Mor+14]), where the authors present the design and implementation of the facility for experimenting solutions for the next generation Internet. FITS is build using Xen [Xen21] and OpenFlow [Sga+13]. In [Zha+19] the authors evaluate the performances of state-of-the-art software switches: OVS-DPDK, snabb, BESS, FastClick, VPP and netmap VALE.

Graphical Network Simulator-3 (GNS3) [EA15] is a software emulating routers and switches in order to create virtual networks with a GUI. It can be used to emulate Cisco routers and supports a variety of virtualization tools such as QEMU, KVM, and Virtual Box to emulate the vHosts.

`Mininet` [LHM10] is the most common Software Defined Networking (SDN) emulator. It allows to emulate an SDN network composed of hundreds of vHosts and vSwitches on a single host. `Mininet` is easy to use and its installation is simple. As we show in Sec. 3.4, it is possible to create a network with dozens of vSwitches and vHosts in just a few seconds. `Mininet` is very efficient to emulate network topologies as long as the resources required for the experiments do not exceed the ones that a single machine can offer. If physical resources are exceeded, the results might be not aligned with the ones of a real scenario.

The tools closest to ours are `Maxinet` [Wet+14] and `Mininet Cluster Edition` (`Mininet CE`[Pro19]). They allow to distribute `Mininet` on a cluster of nodes. `Maxinet` creates different `Mininet` instances in each physical node of the cluster, and connects the vSwitches between different physical hosts with GRE tunnels. `Mininet CE` extends directly `Mininet` in order to distribute the vNodes and the vLinks in a set of machines via GRE or SSH tunnels. Containernet [PKR16] allows to extend `Mininet` to support Docker containers. By default, it is not able to distribute the emulation on different nodes, but it is possible to combine it with `Maxinet` or `Mininet CE` to support such an option and provide better vNodes isolation. One of the main problems is that Maxinet and Containernet do not consider the properties of the physical infrastructure (we will focus on this limitation in the next chapter) in which they are running, e.g., a machine or a link in the physical network may be overloaded during the emulation. With Distrinet, we combine the concepts of Maxinet and Containernet to provide a distributed Mininet implementation which takes into account the characteristics of the logical and physical topologies.

While the `Maxinet` approach makes it possible to increase the scalability

	Distrinet	Mininet CE	Maxinet
Mininet compatibility			
Runnable with mn command	✔	✔	✘
Mininet API	✔	✔	●
Tunneling technologies			
VXLAN Tunnels	✔	✘	✘
GRE Tunnels	●	✔	✔
Emulation features			
Unlimited vLink	✔	✔	✔
Limited vLink (No tunneling)	✔	●	●
Limited vLink (Tunneling)	✔	✘	●
vNode isolation	✔	●	●
Automatic cloud provision	✔	✘	✘

✔=Yes, ●=Partial, ✘=No

Table 3.1: Supported features in the different tools for distributed emulation.

of `Mininet` and offers a speed-up in terms of virtual network creation time for certain topologies, its main drawback is that it is not directly compatible with `Mininet`. Moreover, even though it is straightforward to setup networks with unlimited vLinks (i.e., vLinks without explicit bandwidth limit or delay), `Maxinet` does not fully support limited vLinks (i.e., vLinks with explicit bandwidth limits or delay). The `Mininet CE` approach offers a full compatibility with `Mininet`, but like `Maxinet`, it has some limitations when it comes to emulation of vLinks with limited bandwidth or delay. It is not possible to add limitations on the vLink if it is connected between two vNodes in different physical machines [Min16].

We believe that automatic cloud provision offered by Distrinet, its flexibility, and its compatibility with `Mininet` give our tool an important added value as `Mininet` is by far the most used tool to emulate SDN networks. Table 3.1 summarizes the main differences between the tools.

3.3 Distributed Mininet

Even though the general concepts and API of Mininet do not prevent experiments from running on multiple machines, the Mininet implementation has been thought and made to run on a single machine. However, in some circumstances, being able to run experiments on a set of machines would be helpful, for example when the memory of one machine would not suffice to support the whole experiment. With Distrinet, we extend the Mininet implementation to be able to distribute network experiments over multiple hosts by using the Mininet programmatic idioms, in order to remain fully compatible with Mininet.

Four key elements have to be considered in order to distribute `Mininet` experiments over multiple hosts.

First, emulated nodes must be *isolated* to ensure the correctness of the experiments, even when the hosts supporting the experiments are heterogeneous. To obtain these guarantees, virtualization techniques (full or container-based) have to be employed. Similarly, *traffic encapsulation* is needed such that the network of the experiment can run on any type of infrastructure.

To start and manage experiments, an *experimentation control plane* is necessary. This control plane allows to manage all the emulated nodes and links of the experiment, regardless of where they are physically hosted.

3.3.1 Multi-Host Mininet Implementation

In `Mininet`, network nodes are emulated as user-level processes isolated from each other by means of light virtualization. More precisely, a network node in `Mininet` is a shell subprocess spawned in a pseudo-tty and isolated from the rest by the means of Linux cgroups and network namespaces. Interactions between `Mininet` and the emulated nodes are then performed by writing bash commands to the standard input of the subprocess, and reading the content at the standard output and error of that process.

As `Mininet` runs on a single machine, every emulated node benefits from the same software and hardware environments (i.e., the one from the experimental hosts). This approach has proven to be adequate for single-machine experiments but cannot be directly applied when experiments are distributed, as it would push too much burden in preparing the different hosts involved in the experiments. As a consequence, we kept the principle of running a shell process, but instead of isolating it using cgroups and network namespaces, we isolated it within an LXC container [Can19]. Ultimately, LXC realizes isolation in the same way than using kernel cgroups and namespaces, but it provides an effective tool suite to set up any desired software environment within the container just by providing the desired image when launching the container. In this way, even when the machines used to run an experiment are set up differently, as long as they have LXC installed on them, it is possible to create identical software environments for all the network nodes, regardless of the machine that actually hosts them. In Distrinet, to start a network node, we first launch an LXC container and create a shell subprocess in that container. As `Mininet` runs on a single machine, the experiment orchestrator and the actual emulated nodes run on the same machine, which allows to directly read and write on the file descriptors of the bash process of the network nodes to control them.

In Distrinet, we allow to separate the node where the experiment orchestration is performed from the hosts where the network nodes are hosted, meaning that creating directly a process and interacting with its standard I/Os is not as straightforward as in `Mininet`.

Indeed, `Mininet` uses the standard `Popen` Python class to create the bash process at the basis of network nodes. Unfortunately, `Popen` is a low-level call in Python that is limited to launching processes on the local machine. In our case, we then have to rely on another mechanism. As we are dealing with remote machines and want to minimize the required software on the hosts involved in

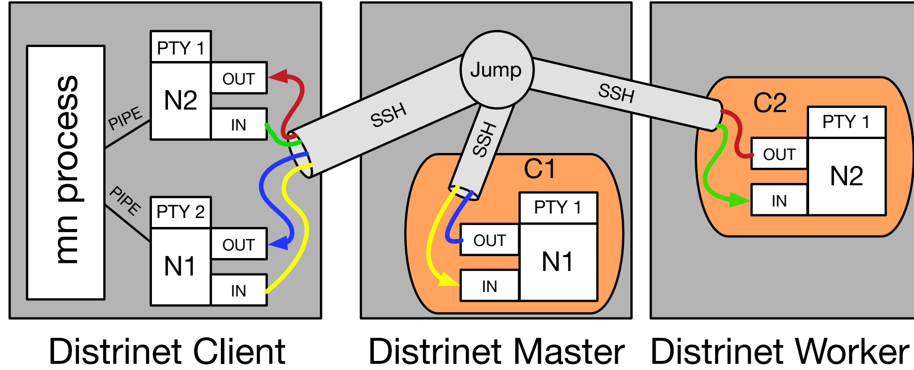


Figure 3.1: Distrinet processes interaction.

experiments, we use SSH as a means to interact between the orchestrator and the different hosts and network nodes. SSH is used to launch containers and once the container is launched, we directly connect through SSH to the containers and create shell processes via SSH calls.
In parallel, we open pseudo-terminals (PTYs) locally on the experiment orchestrator, one per network node, and attach the standard input and outputs of the created remote processes to the local PTYs. As a result, the orchestrator can interact with the virtual nodes in the very same way as `Mininet` does by reading and writing in the file descriptors of the network nodes' PTY.
This solution may look cumbersome and suboptimal, but it maximizes the `Mininet` code reuse, and ultimately guarantees compatibility with `Mininet`. Indeed, `Mininet` heavily relies on the possibility to read and write, via file descriptors, the standard input and outputs of the shell processes emulating the virtual nodes, and massively uses `select` and `poll` that are low-level Linux calls for local files and processes. Therefore, providing the ability to have local file descriptors for remote process standard input and outputs allowed us to directly use `Mininet` code as the only change needed was in the creation of the shell process (i.e., using an SSH process creation instead of `Popen`), with no impact on the rest of the `Mininet` implementation.
Solutions that would not offer low-level Linux calls compatibility to interact with the remote shell would cause to re-implement most of the `Node` classes of `Mininet`.

In `Mininet`, network nodes and links are created sequentially. The sequential approach is not an issue in `Mininet` where interactions are virtually instantaneous. However, a sequential approach is not appropriate in Distrinet, since nodes are deployed from LXC images and because every interaction with a node is subject to network delays. For this reason, in Distrinet, the node deployment and setup calls are made concurrent with the Asynchronous I/O

library of Python 3. However, as the compatibility with `Mininet` is a fundamental design choice, all calls are kept sequential by default and we added an optional flag parameter to specify the execution to run in concurrent mode. When the flag is set, the method launches the commands it is supposed to run and returns without waiting for them to terminate. The programmer then has to check if the command is actually finished when needed.

To this end, we have added a companion method to each method that has been adapted to be potentially non blocking. The role of the companion method is to block until the command calls made by the former are finished. This allows one to start a batch of long lasting commands (e.g., `startShell`) at once, then wait for all of them to finish. We have chosen to use this approach instead of relying on callback functions or multi-thread operations in order to keep the structure of the `Mininet` core implementation (Fig. 3.1).

To implement network links, `Mininet` uses virtual Ethernet interfaces and the traffic is contained within the virtual links thanks to network namespaces. When experiments are distributed, links may have to connect nodes located on different hosts, hence an additional mechanism is required. In Distrinet, we implement virtual links by using VXLAN tunnels (a prototype version with GRE tunnels also exists). The choice of VXLAN is guided by the need of transporting L2 traffic over the virtual links. In particular, we cannot rely on the default connection option provided directly with LXD. Indeed, the latter uses either multicast VXLAN tunnels or Fan networking [Can20] to interconnect containers hosted on different machines. However, cloud platforms such as Amazon EC2 do not allow the usage of multicast addresses and, in some scenarios, a single physical machine may have to host hundreds of containers.

Fan networking maps the addresses of a small network address space (e.g., /16 network) with a larger one (e.g., /8 network) and uses unicast tunnels to interconnect the different machines, but it does not allow to choose the IP addresses of the containers arbitrarily (the user cannot choose the IP of the virtual interfaces in the emulated network). In Distrinet, each link is implemented with a unicast VXLAN tunnel having its own virtual identifier. Also, since we are compatible with `Mininet`, to limit the capacity of the links, we simply use the `Mininet` implementation that relies on Linux Traffic Control (`tc`).

SSH is used to send commands and retrieve data from the network nodes in the experiments, and each virtual node is reachable with an IP address. To do so, a bridge, called *admin bridge*, is setup on every machine that hosts emulated nodes. An interface, called *admin interface*, is also created on each node and bridged to the admin bridge and is assigned a unique IP address picked up from the same subnet. All these admin bridges are connected to the admin bridge of the master node.

The machine running the script is then hooked with an SSH tunnel to the master host, and can then directly access any machine connected to the admin bridge. The general architecture of Distrinet is presented in Fig. 3.2. In Mininet, hosts are emulated as user-level processes isolated from each other by making use of the Linux cgroup and network namespaces. With Distrinet, we re-use the principle of containerisation but implement it with LXC and LXD to simplify the

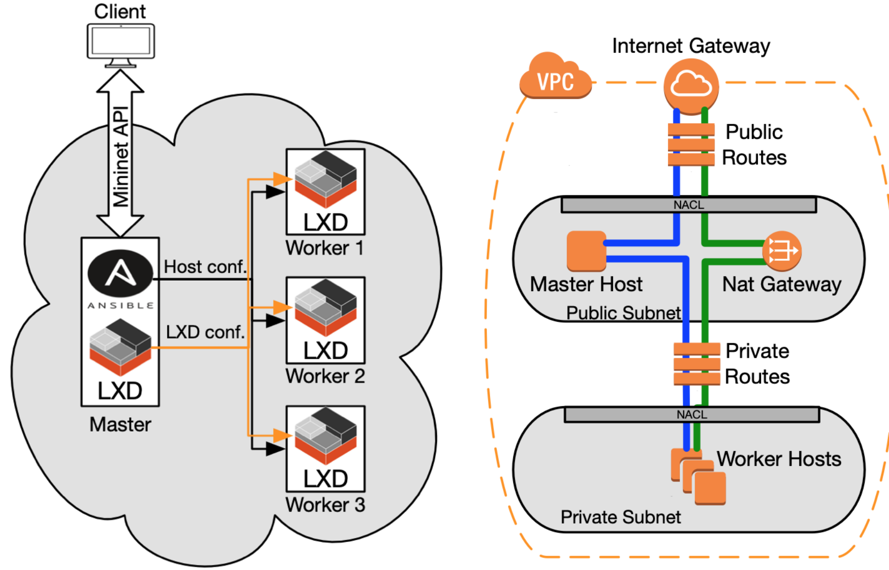


Figure 3.2: Distrinet general architecture.

Figure 3.3: Amazon VPC configuration.

management of complex scenarios. Mininet heavily relies on the `popen` function of Python to interact with the emulated nodes. In order to distribute Mininet, we implemented a new class, `RemotePopen`, that implements the same interface as the native `Popen` class of Python and extended the `Node` class of Mininet to use our implementation instead of using the one from Python. `RemotePopen` is implemented with SSH channels from the `Paramiko` library. To control emulated nodes via SSH, a control network interconnecting all nodes is deployed. Network links are emulated with virtual Ethernet when the two end points of the link are deployed on the same machine. If the end points are deployed on separate machines, then the connection is made over VxLAN. Link data rate control is directly inherited from Mininet.

3.4 Distrinet Architecture

Distrinet provides an infrastructure provisioning mechanism that uses Ansible to automatically install and configure LXD and SSH on each machine to be used during the experiment.

If the experimental infrastructure is Amazon EC2, Distrinet first instantiates a Virtual Private Cloud (VPC) configured as depicted in Fig. 3.3 in which the virtual instances running the experiment will be deployed. A NAT gateway is automatically created to provide Internet access to the Worker host. Access to the Worker nodes from the experimenter machine is ensured by a Master node acting as an SSH relay. The deployment on Amazon EC2 only requires an active

Amazon AWS account.

Distrinet environment (cloud or physical) includes the three following entities (Fig. 3.2):

- *Client*: host in which the Distrinet script is running and decides where to place the vNodes around the physical infrastructure (round-robin by default). The Client must be able to connect via SSH to the Master host.
- *Master*: host that acts as a relay to interconnect the Client with all the Worker hosts. It communicates with the Client and the different Workers via SSH. Note that the Master can also be configured as a Worker.
- *Worker(s)*: host(s) where all the vNodes (vSwitches and vHosts) are running. vNodes are managed by the Master and the Client, via the admin network.

Distrinet can then automatically install the remaining requirements. In particular, it installs and configures LXD/LXC and OpenVSwitch in the Master and Worker hosts. In the next step, Distrinet downloads two images: an Ubuntu:18.04 image to emulate the vHosts, and a modified version of that image with OVS installed in order to save time during the configuration process. A default configuration setup is provided, but the user – by following the tutorial we provide [Dis20] – can easily create a personalized image and distribute it in the environment using Ansible from the Master Node. After the configuration step, the user can start the emulation from the Distrinet Client.

3.5 Experiments

To evaluate Distrinet, we considered 3 types of experiments. The first one measures the overhead in term of execution time that the tools introduce to allow the distribution of the emulations.
The second experiment compares the network capabilities of `Mininet CE`, `Maxinet`, and Distrinet. The last experiment shows the behaviors of a resource intensive emulation inside a single physical host and in a distributed environment. For the evaluation, we used Amazon Web Service (AWS) [Ama20e] and Grid'5000 [Bal+13]. In Grid'5000, we used the *Gros* cluster where the hosts are all equipped with one Intel Xeon Gold 5220 (18 multi-threaded cores per CPU at 2.20 GHz, i.e., 36 vCores) and 96 GB of RAM.

3.5.1 Distrinet Core Performance Assessment

We first measure the overhead of distributing an experiment by timing the most fundamental operations of the tool in Grid'5000 and compare it with the other tools. The results are reported in Table 3.2. `Mininet` is running on a single host while the others are running on two hosts. We observe that the creation of a vNode or a vLink is much longer with Distrinet because it uses LXC containers that are slower to start, but are more isolated. This implies that setting up

Action	Mininet	Distrinet	Mininet CE	Maxinet
Link Creation	7.5 ± 0.19 ms	495 ± 18 ms	430 ± 840 ms	13 ± 0.3 ms
Link Deletion	48.4 ± 8.3 ms	495 ± 18 ms	40.4 ± 7.3 ms	—
Node Creation	13.2 ± 0.1 ms	2.09 ± 0.04 s	329.9 ± 161.4 ms	108.4 ± 5.4 ms
Running a Command	3.45 ± 0.1 ms	4.8 ± 0.2 ms	3.35 ± 0.07 ms	1.19 ± 0.1 ms
Tunnel Creation	—	—	—	184.8 ± 1.0ms
Topology Creation				
Linear ($n = 2$)	0.368 ± 0.01 s	13.44 ± 0.25 s	4.09 ± 0.05 s	1.60 ± 0.06 s
Linear ($n = 10$)	1.412 ± 0.03 s	37.5 ± 0.55 s	9.7 ± 0.08 s	5.22 ± 0.26 s
Binary Tree ($h = 4$)	2.051 ± 0.05 s	54.9 ± 0.69 s	13.51 ± 0.28s	6.36 ± 0.08 s
Fat Tree ($k = 4$)	2.605 ± 0.04 s	73.72 ± 0.48 s	59.685 ± 1.23 s	8.40 ± 0.27 s

Table 3.2: Time overhead ± standard deviation over 10 experiments

an experiment with Distrinet is slower. As examples, we provide the time to create different classic topologies with both tools. A Fat Tree 4 is built in 73.72 s with Distrinet to be compared to around 2.6 s with `Mininet`. `Mininet Cluster Edition` requires 59.68 s, while `Maxinet` requires around 8.4 s. This is because the *Link creation* implementation in Distrinet and `Mininet Cluster Edition` are similar, while in `Maxinet`, it is completely different. `Maxinet` first sets up the virtual sub-networks in the different physical machines without tunneling. After this step, it creates the tunnels between the vNodes placed in different hosts. For this reason, in Maxinet, there are different lines for *link creation* and *tunnel creation*, while in `Mininet CE` and Distrinet, the *tunnel creation* time is included in the *link creation*.
In Mininet, as explained before, vNodes are light but are not completely isolated. This can be problematic for some types of experiments. With our LXD/LXC approach, the vNode creation is slower, but the container provides a better isolation and the vNodes can be distributed.
The difference in setup time is significant with Maxinet, but for networks with a higher density of links (e.g., FatTree $k = 4$), the gap between Distrinet and Mininet CE is reduced. However, we believe that, for a large subset of experiments (e.g., running a 24 hour trace), the setup time represents a negligible part of the total experiment time. In any case, computation intensive experiments – such as the one presented in Sec. 3.5.3 – cannot be performed with a single physical machine. We thus believe that Distrinet is a useful tool in many different use cases.

3.5.2 Tools Comparison

In this section, we compare Distrinet, `Mininet`, `Mininet CE`, and `Maxinet` network capabilities. The experiments compare the maximum network throughput measured with iperf (TCP traffic, for one minute) between the first and the

	Single Host (Grid'5000)		Multiple Hosts (Grid'5000)				Amazon EC2
Topology	**Mininet**	**Distrinet**	**Mininet CE (SSH)**	**Mininet CE (GRE)**	**Maxinet**	**Distrinet**	**Distrinet**
Linear 2	957.0 ± 0.0	957.0 ± 0.0	731.2 ± 14.6	955.0 ± 0.0	953.0 ± 1.9	957.0 ± 0.0	947.2 ± 5.5
Linear 10	957.0 ± 0.0	957.0 ± 0.0	640.7 ± 8.8	955.0 ± 0.0	897.3 ± 8.3	957.0 ± 0.0	947.2 ± 4.8
Linear 20	957.0 ± 0.0	956.9 ± 0.3	582.3 ± 14.5	955.0 ± 0.0	841.9 ± 21.8	957.0 ± 0.0	938.3 ± 4.3
Linear 50	957.0 ± 0.0	955.0 ± 2.2	391.0 ± 12.3	955.0 ± 0.0	641.3 ± 44.0	957.0 ± 0.0	916.4 ± 7.0
Star 10	957.0 ± 0.0	957.0 ± 0.0	729.1 ± 15.4	955.0 ± 0.0	927.6 ± 2.76	957.0 ± 0.0	946.4 ± 5.3

Table 3.3: Maximum throughput (in Mbps) ± standard deviation over 10 experiments - the virtual links are bounded at 1 Gbps.

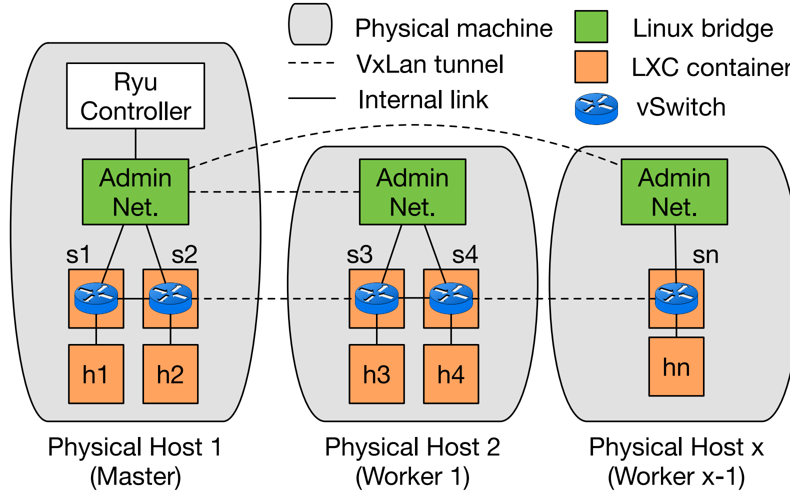


Figure 3.4: Distrinet setup example

last hosts for different linear topologies. For the star topology, there is an iperf connection between 5 pairs of hosts. Table 3.3 summarizes the results over 10 experiments for each virtual topology. The physical environment is composed of one or two machines completely dedicated to the experiments in Grid'5000. The vLink capacities are set to 1 Gbps. The second and the third columns compare `Mininet` and Distrinet while running on a single machine. We can see that the maximum throughput measured with iperf is very close between our tool and `Mininet`. The next set of experiments compares `Mininet CE`, `Maxinet`, and Distrinet while distributing the virtual network on two physical hosts - the Maximum Transmission Unit (MTU) for the physical interfaces has been set at 1600 bytes to avoid fragmentation. To be fair in the comparison, the distribution of the virtual nodes is done using the `Maxinet` placement algorithm with all the tools, since `Maxinet` is the only one that has restrictions on the placement (i.e., `Maxinet` does not allow to place a vSwitch and a vHost on different physical machines if they are connected through a vLink).
As mentioned in Sec. 3.1, `Mininet CE` does not support TCLinks for vLinks

connecting vNodes in different physical machines. Therefore, it is not possible to limit the capacity of the virtual link between two physical hosts. In Table 3.3, we observe that `Mininet CE` using SSH tunnels does not manage to run at `Mininet` speed even for a simple topology such as *Linear 2*. The explanation is that an SSH tunnel consumes lots of processing resources, which is confirmed by the fact that `Mininet CE` obtains results comparable to the ones of `Mininet` and Distrinet when lightweight GRE tunnels are used instead.
`Maxinet` obtains good results when the topology is not too large and when the traffic is not passing through many switches (e.g., *Linear 2* and *Star 10*). However, performances drop when topologies get larger. Distrinet results, obtained in a distributed environment, are comparable to the ones of `Mininet` and Distrinet on a single host. In the last column, we show Distrinet results on Amazon AWS using two instances - the instance type is *m5.8xlarge*, with 32 vCPU and 128 GB of RAM [Ama20e]. As displayed in the table, the performances are similar to the ones obtained in Grid'5000. The difference in terms of performances can be explained by the additional virtualization layer needed on the AWS instance and the network resources shared with other users' instances in the cloud.

3.5.3 Experiments With High Load

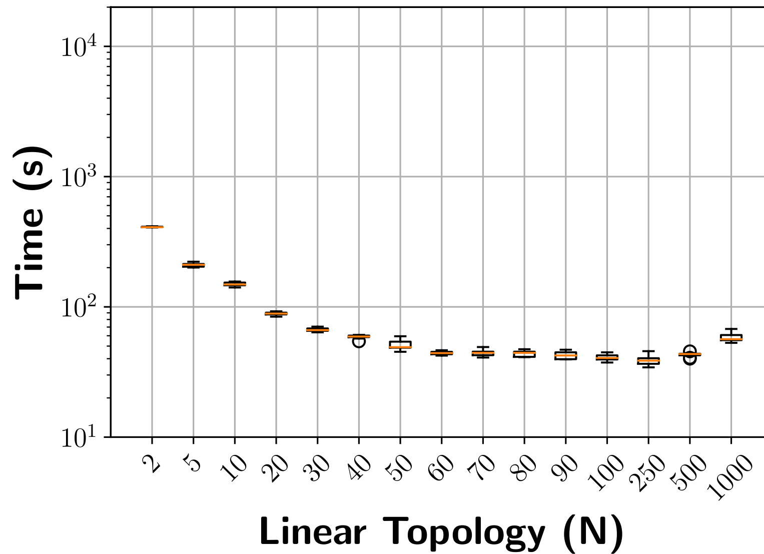


Figure 3.5: Distrinet running in multiple hosts, placing 10 vHosts and 10 vSwitches on each physical host.

To observe the behavior of Distrinet under high load, we set up an experiment running Hadoop Apache [Had20a]. We compare two sets of runs: one

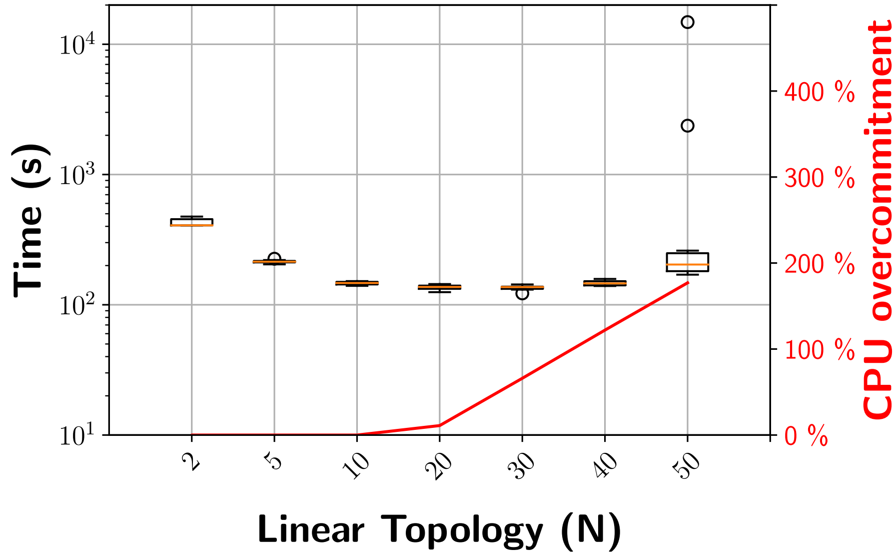


Figure 3.6: Distrinet running in a single host, with the CPU overcommitment percentage for each topology.

set distributed between one and 100 physical machines, the second set running different topologies on a single physical machine.

Distrinet Distributed Setup

Fig. 3.4 shows a simplified overview of the Distrinet setup. It is important to distinguish between the *Distrinet Master/Workers* and the *Hadoop Master/Workers*. Distrinet Master/Workers are the physical machines in which the environment emulation is running, whereas Hadoop Master/Workers are the virtual hosts created inside the physical machines. In the example of Fig. 3.4, physical Host 1 is the Distrinet Master, while the other physical hosts are the Distrinet Workers. *Physical Host 1* deploys 2 virtual hosts: *h1* (Hadoop Master) and *h2* (Hadoop Worker).
The remaining Hadoop Workers are virtualized inside the different Distrinet Workers.

Distrinet Single Host Setup

The incomplete isolation of Mininet nodes prevents Hadoop from running properly. Hence, we use Distrinet in a single host, since the network performances are very similar with respect to the Mininet's ones (Tab. 3.3).

Experiments

The experiments consist in running a standard Hadoop benchmark function (`hadoop-mapreduce-examples.jar`) [MVE14] in a virtual linear topology. π is calculated using a quasi-Monte Carlo method. We used 400 maps and 400 samples for each map. Experiments are run for 2 to 1,000 Hadoop hosts. Execution times of the experiments are reported in Fig. 3.5, and Fig. 3.6, each experiment being repeated 10 times for each topology size.
The expected behavior is that the execution time decreases when Hadoop nodes can be executed on an increasing pool of physical resources (cores and memory). Adding then Hadoop nodes should not change the execution time. We set each vHost with 2 vCores and 6 GB of RAM, while a vSwitch requires 1 vCPU and 3.5 GB of RAM. With these parameters, a single physical host is able to virtualize without **CPU overcommitment** 10 vHosts and 10 vSwitches.
We call CPU overcommitment the estimation (in %) of how much CPU is assigned in excess to a physical node, i.e., if a physical machine has 36 cores and the vNodes assigned in the machine require 54 vCores at full speed, the machine has 50% overcommitment.
The behavior of the experiments using Distrinet with each host running 10 vHosts and 10 vSwitches (except for linear 2 and linear 5) is the one expected from Hadoop (Fig. 3.5). The computation time decreases while the number of vHosts increases, until adding new workers does not decrease anymore the completion time (in this case between 50 and 60 vHosts). When the linear topology is composed of 1,000 vHosts, there is a slight increment in terms of the execution time, which probably depends on two factors: (i) the distance between the Hadoop Master host and the last Hadoop Worker (the connection has to cross 1,000 vSwitches), (ii) and the large amount of Hadoop Workers that the single Hadoop Master has to manage.
We observe that the behavior using a single machine is hardly predictable (Fig. 3.6) when the resources in the single machine are not sufficient to satisfy the requests of the virtual instances. For this reason, we need to distribute the load of the emulation in different hosts.
Comparing Fig. 3.5 to Fig. 3.6, we can observe in the single host emulation scenario that we obtain the same results as in the multiple hosts one when experiments do not exceed the physical resources of the Host (until linear 10). When the emulation requires additional resources, we can observe that the single host emulation needs more time to complete the execution. The red line is an estimation of the CPU overcommitment inside the physical host. Virtualizing the linear 50 network on a single host requires more time than for a linear 10 network. With linear 50, we experienced some abnormal behaviors in two experiments (one run required 2,370 s and another one 14,797 s). This is due to an increasing overcommitment of the physical machine.

3.6 Conclusions

To overcome the limitations of resource-intensive experiments being run on a single machine, we proposed Distrinet that extends `Mininet` capabilities to make it able to distribute experiments on an arbitrary number of hosts.
We have shown the cost to adapt an application designed to run on a single machine to run in multiple hosts, while remaining compatible with it. Our implementation is flexible and experiments can be run on a single Linux host, a cluster of Linux hosts, or even in the Amazon EC2 cloud. Distrinet automatically provisions hosts and launches Amazon instances such that experimenters do not need to know the details of the infrastructure used for their experiment. It is compatible with `Mininet` and its source code is available on `https://distrinet-emu.github.io`. In the next chapter; we study how to correctly distribute the emulation in a physical environment.

Chapter 4

Distributed Network Emulation

4.1 Introduction

The complexity of networks has greatly increased in the last years. Networks currently rely massively on software frameworks and virtualization, and their performances become implementation dependent.
Using a single machine for a rapid emulation is thus limiting to handle resource intensive experiments, e.g., needing heavy memory, processing, input/output, or specific hardware to emulate, for instance, networks with virtual network functions or artificial intelligence algorithms.
To tackle this issue, distributed emulation tools were proposed: Maxinet [Wet+14], Mininet Cluster Edition [Pro19], Distrinet [Dis19; Di +19c]. These tools were presented in the previous chapter. They allow to run experiments of various types on a large number of machines with different hardware configurations. In this chapter, we focus more specifically on the methodology used to distribute the experiments with each tool.

Carrying distributed emulation rises several challenges. First, when facing an experiment, is there a need to distribute it? In other words, how to know if the experiment exceeds the capacity of a single node? Then, if yes, with how many nodes and on which nodes should it be distributed? If it has to be distributed onto m machines, how should the experiments be executed on these machines?
Actually, a networking experiment can be seen as a virtual network or a graph with node and link demands in terms of CPU, memory, network capacity, etc. A fundamental problem that arises in this context is how to map virtual nodes and links to a physical network topology, while minimizing a certain objective function without exceeding the available resources.
Existing tools have placement modules answering partially these questions. `Mininet Cluster Edition` implements three simple algorithms (Round Robin,

Random, and Switch Bin [Pro19]), while `Maxinet` uses an algorithm from `METIS` [met20], a library for graphs partitioning. However, these placement methods have several important limitations.
Firstly, they do not take into account the nodes' resources and the links' capacities. This means that they do not verify if nodes or links capacities are exceeded. Consequently, experiments may run with congested links and overloaded nodes, leading to unreliable results.
Secondly, they do not minimize the number of machines required as they use all machines at their disposal. This is especially important for public clusters, where physical resources are shared, as policy rules might lead to a large waiting time to obtain the needed resources.

To solve these limitations, we studied placement algorithms to map an experiment onto a set of physical machines (e.g., in private test-beds or clusters). The experimental infrastructure topology is taken into account. The goal is to provide a mapping such that the physical resources of the nodes (i.e., processing and memory) and links (i.e., capacity) are not exceeded. This is important to ensure a trusty emulation. The combination of node and link constraints makes finding a feasible solution a difficult task. Indeed, by using a small number of physical nodes, we might exceed their physical resources, while by using too many physical nodes, we may exceed the available rate of the physical links if the virtual nodes are not placed correctly. Our objective consists in minimizing the number of reserved machines to run an experiment, motivated by the fact that scientific clusters such as *Grid'5000* [Gri20] require to reserve a group of machines before running an experiment [Vic+13] and an excess in these terms may lead to usage policy violations, or to a large waiting time to obtain the needed resources.
The problem can be seen as a variant of a Virtual Network Embedding (VNE) problem. However, only exact methods based on linear programming were proposed to deal with it in the literature, and such solutions do not scale well and have long execution times for large networks which constitute our targets in this work. This motivates the need for fast algorithms that can provide near optimal solutions.

We propose new tailored placement algorithms and compare them with the ones used in existing tools. We built a *placement module* for distributed emulators to solve efficiently this problem in practice. This module first decides if the experiment has to be distributed. Then, given a pool of available machines, it computes the deployment using the minimum number of machines to run the experiments in such a way that physical resources are not exceeded. The placement module can be used with any emulator. However, to test it in the wild, we integrated it in `Distrinet`. Through this approach, the experiment is automatically distributed over several nodes using the optimal allocation.

To summarize, our contributions are as follows:

- We study placement algorithms to distribute an experiment onto the machines of a test-bed. We proposed several efficient algorithms to deal with the problem.

- We build a placement module for distributed emulators with all the algorithms implemented. The placement module can currently be used in `Distrinet`, but the algorithms may potentially be integrated in any tool.
- We compare our algorithms with the ones implemented in existing tools using extensive simulations. We show that they succeed in ensuring that no link or node capacity is exceeded, while the same experiments running with other tools would lead to resource overload.
- We then carry experimentation in a private cluster with the goal of evaluating the impact of such resource overload on the emulation. We show that overloading a link, the CPU, or the memory may lead to respectively important drops of measured bandwidth, the increase of execution time, and emulation crashes.

The rest of this chapter is organized as follows. In Sec. 4.2, we review the related works on placement methods to carry out distributed emulation. We then formally state the problem and propose algorithms to deal with it in Sec. 4.3. We evaluate the algorithms against existing placement modules with extensive simulations in Sec. 4.4 and with experimentation in Sec. 4.5. Last, we conclude and present future work on placement algorithms in Sec. 4.6.

4.2 Related Work

Placement For Distributed Emulations

Existing tools for distributed large-scale emulations adopt different strategies to map the virtual topology to the physical one.

`Mininet Cluster Edition` provides three different placement algorithms [Pro19]:
- *SwitchBinPlacer* first distributes the virtual switches (vSwitches) (and the controllers if some are assigned) around the infrastructure, such that each physical host (also called server) has the same amount of vSwitches assigned. It then places the virtual hosts (vHosts) on the server to which its connected vSwitches are assigned.
- *RoundRobinPlacer* is the implementation of the classic RoundRobin algorithm that assigns a vSwitch or a vHost, choosing each time the next physical host in the list.
- *RandomPlacer* is the simplest placer: for each vHost or vSwitch to be assigned, it chooses a random physical node.

`Maxinet` uses the Multilevel Recursive Bisectioning algorithm [KK98] to partition the virtual switches and the virtual hosts into the physical machines. In `Maxinet`, there is no notion of the physical infrastructure (hosts resources or network topology). This means that the partition will not change if we deploy the virtual network in different physical topologies (i.e., spanning tree, clique, etc.). The virtual network, given as an input to the partitioning algorithm, does not have notion of virtual CPU (vCPU) or virtual RAM (vRAM), i.e., a virtual node (vNode) requiring 1 vCPU is treated like a vNode requiring 10 vCPUs.

The partitioning algorithm is not directly implemented in `Maxinet`. `Maxinet` uses `METIS` [met20], a set of tools for partitioning graphs. The goal of the algorithm is to find a partitioning of the nodes such that the sum of the nodes weights (e.g., workload) in each partition is balanced and the sum of all the edges in the cuts are minimized.

During the emulation with Maxinet, three fundamental steps can be found:

- **Network partition.** Maxinet takes as input the virtual network and uses Metis to partition it. In Maxinet, if a vHost and a vSwitch are directly connected, they cannot run on different physical machines. For this reason, the input graph representing the virtual network is reduced as follows. Let's assume that we want to emulate a tree topology with 7 switches and 4 hosts (Fig. 4.1). All the vSwitches are represented by a node in the new graph with weight equal to 1, while the weight on the edges represents the capacity of the link. If a vHost is directly connected to a switch, we increase the weight of the node representing the vSwitch by 1. The input graph is the one in Fig. 4.2. The algorithm runs with Metis'

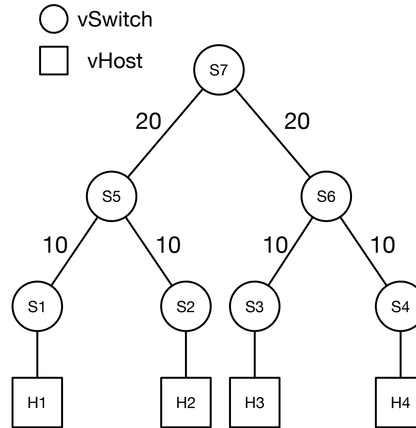


Figure 4.1: Virtual topology

  Multilevel Recursive Bisectioning. In the example (Fig. 4.3), Metis splits the network in two different hosts. After the split, the initial network can be easly reconstructed.

- **Building Mininet networks.** After the partitioning, Maxinet creates a Mininet experiment on each physical node. The network to emulate on each node is provided in the partition.

- **Connecting the remote switches.** After creating different Mininet networks on all the nodes chosen by the partition, Maxinet connects the vSwitches that have a link in the original topology via GRE tunnels.

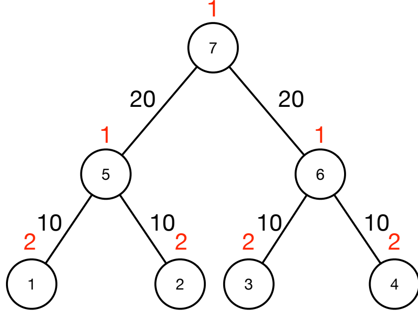


Figure 4.2: Input

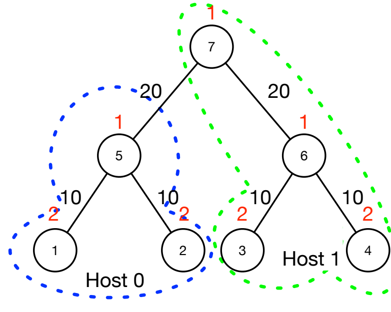


Figure 4.3: Output

In all these algorithms, the physical infrastructure is not taken into account. This means that a *physical link or a physical machine can be overloaded and become a bottleneck* for the emulation *without the user being notified.* In fact, we show that the existing placement solutions behave well when the physical infrastructure is an homogeneous environment. However, when the physical environment is heterogeneous (different types of machines or a complex physical network), they often return solutions with overloaded resources.

Moreover, the existing solutions do not evaluate the minimum number of machines needed to run the experiment (and in particular if the experiment has to be distributed). These solutions use all the machines put at their disposal by the user. On the contrary, our placement module provides to the user the smallest number of physical hosts in order to run the experiment without any overloaded physical resource.

Virtual Network Embedding (VNE) Problem

The solution we propose is based on the investigation of a VNE problem. Such problems have been widely studied in the literature. We refer to [Fis+13] for a comprehensive survey of the existing works. Many different settings have been considered. Minimization of the resource allocation cost [CRB09], of the energy consumption [Bot+12], of the maximum load [CRB12] or revenue maximization [Yu+08] are just few examples. In our settings, we aim at finding a mapping which uses the smallest number of substrate nodes. Thus, our objective can be seen as a variant of the energy-aware VNE problem in which we aim to minimize the number of activated substrate nodes.
Exact solutions which provide optimal techniques to solve small instances have been proposed (see, e.g., [Mel+11; Hou+11]). They are mainly based on exact approaches such as Integer Linear Programming (ILP) and formulate the VNE problem as virtual nodes and links mapping. These approaches are not suitable in our use case, as runtime is a crucial factor and the delay in the embedding of a

virtual request should be minimized. A heuristic approach to find an acceptable solution in a short execution time is to be preferred. We thus propose heuristic approaches able to provide near-optimal solutions in a reasonable computation time. In addition, beyond the general case, we study specific settings often encountered when carrying out emulations in real cluster environments. They are characterized by a homogeneous computing environment or by a spanning tree routing protocol, and they have not been fully addressed in the literature.

4.3 Problem And Algorithms

When emulating large datacenter networks with hundreds or thousand of nodes, it is necessary to distribute the emulation over multiple physical machines in order not to overload the physical resources.

We study here the problem of mapping virtual nodes and links to a physical network topology, while minimizing the number of used machines and without exceeding the available physical resources (CPU core, memory, and link rate). We first define formally the optimization problem which is considered. We then propose algorithms to deal with it. In order to evaluate the performances of the algorithms, we run numerical evaluations and compare their solutions with the optimal one found using an ILP.

We consider several specific settings often found in real cluster environments: homogeneous topology or physical network arranged as a tree.

4.3.1 Problem Statement

The VNE problem can be formally stated as follows.
Substrate Network. We are given a substrate network modeled as an undirected multigraph $G^S = (N^S, L^S)$ where N^S and L^S refer to the set of nodes and links, respectively. G^S is a multigraph as there may be multiple links between a pair of nodes. Each node $n^S \in N^S$ is associated with the CPU capacity (expressed in terms of CPU cores) and memory capacity denoted by $c(n^S)$ and $m(c^S)$, respectively. Also, each link $e^S(i, j) \in L^S$ between two substrate nodes i and j is associated with the bandwidth capacity value $b(e^S)$ denoting the total amount of rate that can be supported.
Virtual Network Request. We use an undirected graph $G^V = (N^V, L^V)$ to denote a virtual network, where N^V is the set of virtual nodes and L^V the set of virtual links. Requirements on virtual nodes and virtual links are expressed in terms of the attributes of the nodes (i.e., CPU cores $c(n^V)$ and memory $m(c^V)$) and links (i.e., the rate to be supported $b(e^V)$) of the substrate network. If there are no sufficient substrate resources available, the virtual network request should be rejected or postponed. When the virtual network expires, the allocated substrate resources are released.

The problem consists in mapping the virtual network requests to the substrate network, while respecting the resource constraints of the substrate network. The problem can be decomposed into two major components: (1) node assignment in which each virtual node is assigned to a substrate node, and (2)

link assignment in which each virtual link is mapped to a substrate path.
The combination of node and link constraints makes the problem extremely hard for finding a feasible solution. Indeed, if on one hand, by using a small amount of substrate nodes, we may exceed physical resources capacities, such as CPU and memory, on the other hand, by using too many nodes, we may exceed the available rate of the substrate links. Rost et al. [RS18] show that the problem of finding a feasible embedding is NP-complete, even for a single request.

4.3.2 Algorithms

We propose three algorithms to tackle this problem. Our algorithms are able to provide near-optimal solutions in a reasonable computation time. Solutions are compared with the optimal ones computed using an ILP approach.

The first two algorithms, K-BALANCED and DIVIDESWAP, have two phases. Firstly, virtual nodes are mapped into the physical topology and, secondly, physical paths are found to map virtual links. The third proposed algorithm, GREEDYPARTITION, mixes both nodes and links mapping.

Homogeneous Case

If the substrate nodes within the cluster are homogeneous in terms of physical resources (or if there is a subset of homogeneous nodes from the entire cluster), an assignment strategy may consist in carrying out a partition of the tasks to be done by the physical machines while minimizing the network tasks that would be necessary to be done. We refer to this algorithm as K-BALANCED. Similarly as in [Gir+19], we use as a subroutine an algorithm for the k-balanced partitioning problem. Given an edge-capacitated graph and an integer $k \geq 2$, the goal is to partition the graph vertices into k parts of equal size, so as to minimize the total capacity of the cut edges (i.e., edges from different partitions). The problem is NP-hard even for $k = 2$ [GJS74]. K-BALANCED solves a k-partitioning problem for $k = 1, \ldots, \min(|N^S|, |N^V|)$, and tests the feasibility of the computed mapping $m : N^V \rightarrow N^S$ of virtual nodes on the substrate network. The smallest k for which a feasible k-partitioning exists will be the output of the algorithm. The corresponding pseudo-code is given in Algorithm 1.
The best known approximation factor for the k-balanced partitioning problem is due to Krauthgamer et al. [KNS09] and achieves an approximation factor of $O(\sqrt{\log n \log k})$, with n being the number of nodes in the virtual network. Nevertheless, as their algorithm is based on semi-definite programming and would lead to long execution time, to deal with the problem, we use the $O(\log n)$ approximation algorithm described in [ST97]. The main idea consists in solving recursively a Minimum Bisection Problem. To this end, we use the Kernighan and Lin heuristic [KL70].

Algorithm 1 K-BALANCED

1: **Input:** Virtual network G^V, Substrate network G^S.
2: **Output:** a mapping of virtual nodes to substrate nodes $m : N^V \rightarrow N^S$.
3: **for** $k = 1, 2, \ldots, \min(|N^S|, |N^V|)$ **do**
4: sol $\leftarrow$ Compute an approximate solution of the k-balanced partitioning problem for G^V.
5: **if** sol is feasible (see Sec. 4.3.2) **then return** sol
6: **end if**
7: **end for**
8: **return** $\emptyset$

Algorithm 2 DIVIDESWAP

1: **Input:** Virtual network G^V, Substrate network G^S.
2: **Output:** a mapping of virtual nodes to substrate nodes $m : N^V \rightarrow N^S$.
3: **for** $k = 1, 2, \ldots, \min(|N^S|, |N^V|)$ **do**
4: Divide the nodes from N^S in k balanced subsets V_1, V_2, ..., V_k.
5: Take a random sample of k physical nodes P_1, P_2, ..., P_k
6: Assign nodes in V_j to P_j for $j = 1, ..., k$.
7: **for** $i = 1, 2, \ldots, N_SWAPS$ **do**
8: Choose at random two nodes $u, v \in N^V$ assigned to two distinct physical nodes.
9: If by swapping u and v the cut weight decreases, swap them in sol and update the cut weight
10: **end for**
11: **if** sol is feasible (see Sec. 4.3.2) **then return** sol.
12: **end if**
13: **end for**
14: **return** $\emptyset$

K-BALANCED has theoretical guarantees of efficiency for its node mapping phase, but only when the number of parts takes some specific values (powers of 2). Indeed, the procedure is based on merging two small partitions until the number of partitions is greater than the desired one (e.g., if $k = 3$ and the algorithm computes 4 partitions, then the result is unbalanced as 2 will be merged).

We thus propose a new algorithm, DIVIDESWAP, efficient for any value of k. The global idea of DIVIDESWAP is to first build an arbitrary balanced partition dividing randomly the nodes in balanced sets, and then swapping pairs of nodes to reduce the cut weight (or required rate for the communications). Its pseudo-code is given in Algorithm 2.

Algorithm 3 GreedyPartition

1: **Input:** Virtual network G^V, Substrate network G^S.
2: **Output:** a mapping of virtual nodes to substrate nodes $m : N^V \rightarrow N^S$.
3: T $\leftarrow$ Compute a bisection tree of G^V.
4: **for** $j = 1, 2, \ldots, |N^S|$ **do**
5: Select the j most powerful machines, $P_1, \ldots, P_j$
6: Perform a BFS on the bisection tree T.
7: **for each** Node v of T **do**
8: **if** $\exists\ P \in P_1, \ldots, P_j$ with enough resource to host the virtual nodes in v **then**
9: v is assigned to P
10: Remove v and the subtree rooted at v from T
11: **end if**
12: **end for**
13: **if** T is empty **then return** sol
14: **end if**
15: **end for**
16: **return** $\emptyset$

General Case

When the substrate nodes are associated with different combinations of CPU, memory, and networking resources, k-balanced and DivideSwap may have difficulties in finding a good assignment of virtual nodes to substrate nodes which respects the different capacities. To prevent this, we define a general procedure referred to as GreedyPartition. Again, we first build a bisection tree. We compute it by recursively applying the Kernighan-Lin bisection algorithm (see the discussion above on bisection algorithm). We then test the use of an increasing number of physical machines, from 1 to N. We take the j most powerful ones, with powerful defined as a combination of CPU and memory. Next, we perform a Breadth First Search (BFS) visit on the bisection tree. For each considered node, we find a physical node such that the resources are enough and the communication can be performed considering the already placed virtual nodes. If the conditions hold for a node, then this node is removed from the tree with all its subtrees. If not, we consider the next node of the bisection tree. If, at any point, the tree is empty, we return the solution.

Link Mapping

DivideSwap and k-balanced are based on the assignment of virtual nodes into physical nodes, then the feasibility checking of the problem by trying to allocate all virtual links between virtual nodes assigned in two different substrate nodes to a substrate path. This problem is solved differently according to the structure of the physical substrate network.

Tree Topologies

Even if the substrate network is assumed to be a tree, there are still decisions to be made in terms of how the network interfaces of a substrate compute node should be used by the virtual nodes. Indeed, if we allow the traffic associated to a single virtual link to be sent using more than one network interface (e.g., 50% on `eth0` and 50% on `eth1`), then multiple links between a compute node and a switch can be considered as a single link with an associated rate which corresponds to the sum of rates on all the node network interfaces. As a consequence, once the mapping between virtual and substrate nodes has been selected, checking if a substrate compute node has enough resources on its interfaces to send or receive a given set of rates can be done *exactly* in polynomial time with a BFS or Depth First Search (DFS) visit in time $O(|N^S| + |L^S|)$, as there exists only one path between each source and destination pair. Conversely, if the virtual link rate to be supported can be mapped on only a single network interface, then the situation can be reduced to the *Bin Packing Problem* and is thus NP-hard. To deal with it, we use the First-fit decreasing heuristic which has been shown to be $\frac{11}{9}$-approximated for this problem [Joh+74] (i.e., it guarantees an allocation using at most $\frac{11}{9}$OPT + 1 bins, with OPT being the optimal number of bins).

General Topologies

In this case, we also need to distinguish two situations according to the desired strategy for mapping a virtual link of the virtual network to a physical path in the substrate network. If path splitting is supported by the substrate network, then the problem can be solved in polynomial time by using a multi-commodity flow algorithm [Yu+08]. On the other hand, if path splitting is not supported, then the situation can be reduced to the *Unsplittable Flow Problem*, which is NP-hard [KS97]. In such case, we use the following approach. We consider the virtual links in non-increasing order. Given the remaining capacities, we find the shortest path in the residual network in which we remove all links with an available rate smaller than the rate to be mapped. If we succeed to find a physical path for all the virtual links to be mapped (i.e., between nodes assigned to distinct physical machines), then the problem is considered feasible.

ILP Approach

As previously mentioned, we compare our approximation algorithms with the optimal solution computed using an ILP approach. The goal of the ILP is to minimize the number of host used for the mapping. We already defined with $G^V = (N^V, E^V)$ and $G^S = (N^S, E^S)$ the virtual network and the substrate (physical) network. The tuple $(u, v) \in E^V$, represents the virtual link between $u, v \in N^V$; while $(i, j, d) \in E^S$ represents the physical link between $i, j \in N^S$ via the device d. We define with c_j and m_j the virtual cores and the memory required by the virtual node $j \in N^V$; while C_i and M_i represent the maximum

cores and memory available in the physical node $i \in N^S$. For the links, r_{uv} represents the bandwidth required by the virtual link $(u, v) \in E^V$, while R_{ijd} represents the maximum bandwidth for the physical link $(i, j, d) \in E^S$. We indicate with $\delta(i)$ the set of neighbours of the node $i \in N^V$ and with $\alpha(i, j)$ the set of devices for the tuple (i, j) with $i, j \in N^S$.
Variables:

- $x_i \in \{0, 1\}$: boolean variable equal to 1 iff there is at least one virtual node assigned to the physical node $i \in N^S$, 0 otherwise
- $y_{ji} \in \{0, 1\}$: boolean variable equal to 1 iff the virtual node $j \in N^V$ is assigned to the physical node $i \in N^S$, 0 otherwise
- $l_{uvijd} \in \{0, 1\}$: boolean variable equal to 1 if the physical link $(i, j, d) \in E^S$ is used by the virtual link $(u, v) \in E^V$, 0 otherwise

Objective (4.1): minimization of the physical nodes needed in order to embed the virtual network.

$$\min \sum_{i \in N^S} x_i \tag{4.1}$$

A machine is used if at least a virtual node is mapped on it (4.2)

$$x_i \geq y_{ji} \qquad \forall i \in N^S, \forall j \in N^V \tag{4.2}$$

Assignment of virtual nodes to physical nodes (4.3)

$$\sum_{i \in N^S} y_{ji} = 1 \qquad \forall j \in N^V \tag{4.3}$$

Cpu (4.4) and memory (4.5) limit constraints:

$$\sum_{j \in N^V} c_j * y_{ji} \leq C_i \qquad \forall i \in N^S \tag{4.4}$$

$$\sum_{j \in N^V} m_j * y_{ji} \leq M_i \qquad \forall i \in N^S \tag{4.5}$$

Bandwidth conservation constraints 4.6:

$$\sum_{j \in \delta(i)} \sum_{d \in \alpha(i,j)} (l_{uvijd} - l_{uvjid}) = y_{ui} - y_{vi} \qquad \forall (u, v) \in E^V, \forall i \in N^S \tag{4.6}$$

Link capacity constraints (4.7):

$$\sum_{(u,v) \in E^V} (r_{uv} * l_{uvijd} + r_{uv} * l_{uvjid}) \leq R_{ijd} \qquad \forall (i, j, d) \in E^S \tag{4.7}$$

Given a virtual link a physical machine, the rate that goes out from the physical machine to a device (4.8) or that comes in to the physical machine from a device (4.9) is at most 1 :

$$\sum_{j \in \delta(i)} \sum_{d \in \alpha(i,j)} l_{uvijd} \leq 1 \qquad \forall (u, v) \in E^V, \forall i \in N^S \tag{4.8}$$

$$\sum_{j\in\delta(i)} \sum_{d\in\alpha(i,j)} l_{uvjid} \leq 1 \qquad \forall (u,v) \in E^V, \forall i \in N^S \tag{4.9}$$

A link can be used only in a direction (4.10):

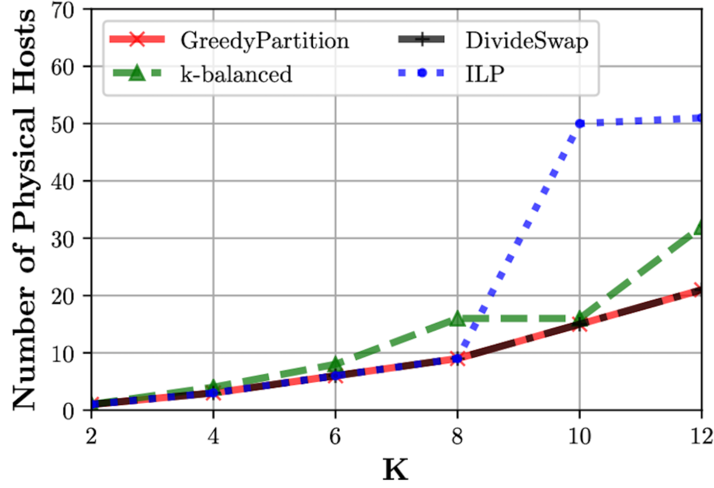
$$l_{uvijd} + l_{uvjid} \leq 1 \qquad \forall (u,v) \in E^V, \forall (i,j,d) \in E^S \tag{4.10}$$

4.3.3 Numerical Evaluation

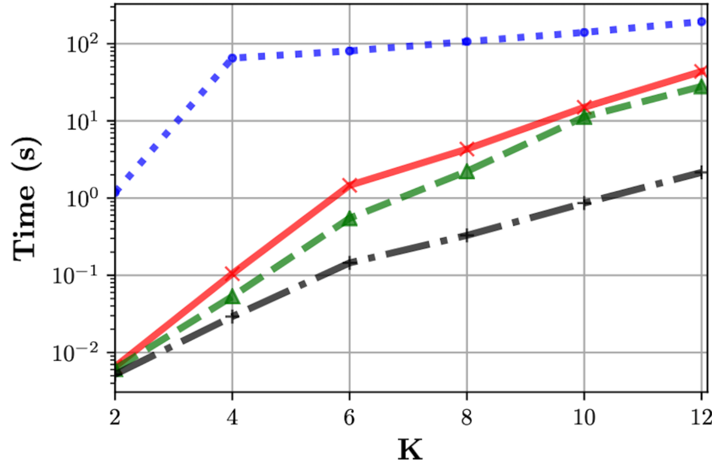
The experiment consists in mapping datacenter servers interconnected by a K-Fat Tree onto the Nancy node of the academic cluster Grid'5000 [Gri20]. Each logical switch and server require 2 cores and 8 GB of memory, and each server is sending 0.2 Gbps of traffic. The physical nodes are machines with 32 cores and 132 GB of RAM. The physical links are Ethernet links of capacity 10 Gbps. The goal is to minimize the number of machines of the cluster used for the experiment for different datacenter sizes. In Fig. 4.4, we compare the solutions given by three heuristics with the ones computed using an ILP approach (with a running time limit set to 2 minutes).

The experiments are run on an Intel Core i5 2.9 GHz with 16 GB of memory. For the ILP approach, we use CPLEX 12.8 as a solver. When the time limit is reached, we report the best solution found so far. First, we verify that our solutions map a Fat Tree with K=2 onto a single machine as the requirements in terms of cores and memory are low enough. However, this is no longer the case for 4-Fat Trees and larger topologies. Second, we see that the best proposed algorithms, DivideSwap and GreedyPartition, can compute optimal or near-optimal solutions within a few seconds, showing that they can be used for fast experimental deployment.

We see that DivideSwap and GreedyPartition distribute the experiment very efficiently for large Fat Trees. Indeed, they obtain solutions using the same number of machines as the ILP when it gives optimal solutions (Fat Trees with $K \leq 8$). The ILP does not succeed to find efficient solutions for Fat Trees with K=10 and K=12 in less than 2 minutes, while our algorithms succeed. For the 10-Fat Tree, our algorithms return a solution using 16 physical machines (when the solution of the ILP is using 50). Note that it corresponds to a case (powers of 2) for which k-balanced is guaranteed to use near optimal partitions of the experiment graph. This confirms the efficiency of DivideSwap and GreedyPartition which obtain the same result in this case. While our algorithms are fast and can map large Fat Trees in few seconds, DivideSwap is the fastest. However, GreedyPartition better handles more complex scenarios as it is shown in the next sections. To sum up, we see that the best proposed algorithms, DivideSwap and GreedyPartition, can compute optimal or near-optimal solutions within a few seconds, showing they can be used for fast experimental deployment.



(a) Number of hosts.



(b) Time

Figure 4.4: Performances of the heuristics and ILP solver for a K-Fat Tree - value of the solution found (a) and time needed to find the solution (b).

4.4 Evaluation Of The Placement Modules

In this section, we compare our placement algorithms with the ones used by `Mininet Cluster Edition` and `Maxinet`. In order to make the comparison as meaningful as possible and to understand the advantages and disadvantages of each algorithm, we considered different scenarios with *homogeneous* or *heteroge-*

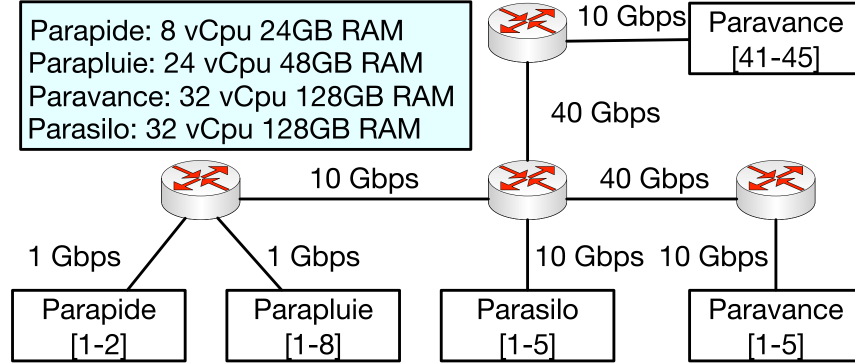


Figure 4.5: *Rennes* topology cluster

neous virtual and physical topologies. Scenarios with homogeneous virtual and physical infrastructures are the most favorable for simple placement algorithms. Scenarios with homogeneous physical infrastructures should be the most favorable ones for the placement modules of the existing tools, as they do not take into account the physical infrastructure. We show that our algorithms outperform them even in this scenario. The heterogeneous scenario represents a more complex case to show the importance of taking into consideration the capacities of the physical infrastructure.

Physical Topologies

The first one is a simple star topology corresponding to the one of the *Gros* cluster in Grid'5000 [Bal+13]. We use 20 physical machines, each equipped with an Intel Gold 5220 (Cascade Lake-SP, 2.20 GHz, 1 CPU/node, 18 cores/CPU) and 96 GB of RAM. The machines are connected by a single switch with 25 Gbps links. The second is represented in Fig. 4.5: this infrastructure is made of a subset of 25 hosts of the *Rennes* cluster in Grid'5000. There are four types of servers with different numbers of cores and memory sizes: Parapide (2 servers with 8 cores and 24 GB of RAM); Parapluie (8 servers with 24 cores and 48 GB of RAM); Parasilo (5 servers with 32 cores and 128 GB of RAM); Paravence (10 servers with 32 cores and 128 GB of RAM) for a total of 25 machines. The servers are interconnected using a small network with 4 switches and links with capacities of 1 , 10 or 40 Gbps.

Virtual Topologies

We use two different families of virtual topologies: Fat Trees and Random. We choose the first one as it is a traditional family of datacenter topologies: this corresponds to the homogeneous scenario. Indeed, Fat Trees present symmetries and all servers are usually similar.

Algorithm Cluster	 vTopo	GreedyP	k-balanced	DivideSwap	Metis	Random	RoundRobin	SwitchBin	Intersection
Gros	vFT	100.0%	91.72%	97.76%	85.02%	72.01%	98.81%	83.18%	68.59%
	vRD	100.0%	83.31%	96.64%	58.31%	52.64%	93.15%	37.86%	32.09%
Rennes	vFT	100.0%	83.90%	92.09%	65.81%	43.22%	68.36%	44.49%	34.18%
	vRD	99.98%	73.51%	74.41%	32.75%	20.37%	34.67%	28.40%	11.47%

Table 4.1: Percentage of solutions found using different algorithms, virtual topologies, and different clusters.

Fat Trees. We test Fat Trees with different parameters:

- K: number of ports included in each switch (2, 4, 6, 8 or 10);
- *Number of CPU cores*: number of virtual cores to assign to each vSwitch or vHost (1, 2, 4, 6, 8 or 10);
- *Memory*: amount of RAM required by each vSwitch or vHost (100, 1,000, 2,000, 4,000, 8,000 or 16,000 MB);
- *Links' rates*: rate associated to each virtual link (1, 10, 50, 100, 200, 500 or 1,000 Mbps).

This corresponds to 3,074 different Fat-Tree networks.

Random Topologies. We use a generator of random topologies which takes as input the number of vSwitches and the link density between them. Half of the vSwitches are chosen to be the core network (meaning that no host is attached to them). The other half are the edge switches (vHosts are connected to them). The generator then chooses a random graph to connect the vSwitches, making sure that all of them are connected. The random graph is obtained by generating Erdös-Renyi graphs using the classical `networkx` Python library till we obtain a connected one that can be used as vNetwork. After setting up the switch topology, a vHost is connected to a single edge switch selected uniformly at random. The capacity of the vLinks (1, 10, 50, 100, or 200 Mbps), the number of virtual cores (vCores) (1, 2, 3, 4, 5, 6, 7, or 8) and the RAM (1,000, 2,000, 4,000, 6,000, 8,000, or 16,000 MB) required for each vNode are then selected uniformly at random. For our experiments, for each pair (N, Density) with N $\in$ (10, 15, 20, 25, 30, 35, 40, 45, 50) and with density $\in$ (0.1, 0.2, 0.3, 0.4, 0.5), we generated 100 random networks.

A fundamental difference between the two families of networks is that, while for Fat Trees all the virtual networks have homogeneous resource requirements (i.e., all nodes and links have the same physical requirements among them), for

Random Networks the requirements associated to the virtual nodes and virtual links may be different.

In this section, we extensively study the performances of the different placement algorithms. To this end, we considered more than 70,000 test instances, corresponding to the mapping of each generated virtual topology on the two physical topologies presented above.

While the existing placement algorithms always return solutions as they do not take into account node and link capacities constraints, it is not the case of our algorithms, as they make sure that resources are not overloaded. To assess the impact of such a difference, we first analyze the cases where feasible solutions are found. We then study the cases where physical constraints are not respected. Finally, we discuss how the algorithmic choices are translated in the number of physical hosts needed to run the experiments.

Finding A Feasible Solution

When comparing the results of the placement algorithms, we only consider the virtual instances for which at least one of them was able to find a feasible solution. In total, we report the experiments made for more than 5,000 instances.

Table 4.1 shows the percentage of instances solved by each algorithm (over the set of feasible instances). We provide the percentage for each family of virtual topologies: *vFT* (virtual Fat Tree) and *vRD* (virtual Random) topologies. For each family of virtual topologies, the tests have been performed on both physical topologies. In particular, the number of feasible solutions analyzed are 761 for *Gros* vFT (Homogeneous-Homogeneous), 4,500 for *Gros* vRD (Homogeneous-Heterogeneous), 708 for *Rennes* vFT (Heterogeneous-Homogeneous), and 4,436 *Rennes* vRD (Heterogeneous-Heterogeneous). We indicate in the last column of the table (Intersection) the percentage of virtual instances for which all algorithms return a feasible solution. First, we observe that a large number of instances cannot be solved by all the algorithms. Second, the results confirm that heterogeneous (whether virtual or physical) topologies are a lot harder to solve (in particular for the algorithms of the existing tools). Note that only 11.7% of the vRD were solved by all algorithms on the *Rennes* cluster.

Two of the proposed algorithms reach the higher success ratio in terms of number of solved instances. In particular, GreedyPartition succeeds to find a feasible solution for almost all the feasible virtual networks when mapped to the *Gros* cluster, and vFT when mapped to the *Rennes* cluster, while it finds a feasible solution for 99.98% of the instances in the vRD case. The second best algorithm is DivideSwap which solved more than 90% of the instances in the *Gros* cluster and in the *Rennes* Cluster for the vFT topology. Note that k-balanced has a lower percentage (76.7%). This is expected as this algorithm is efficient when the solution is mapped on specific numbers of physical hosts (powers of 2), a case for which it has some theoretical guarantees. But it is behaving as well for other values.

Then, the algorithm used by `Maxinet` (`METIS`) finds 85.02% of the solutions for *Gros* vFT. As expected, the algorithm drastically changes its performances

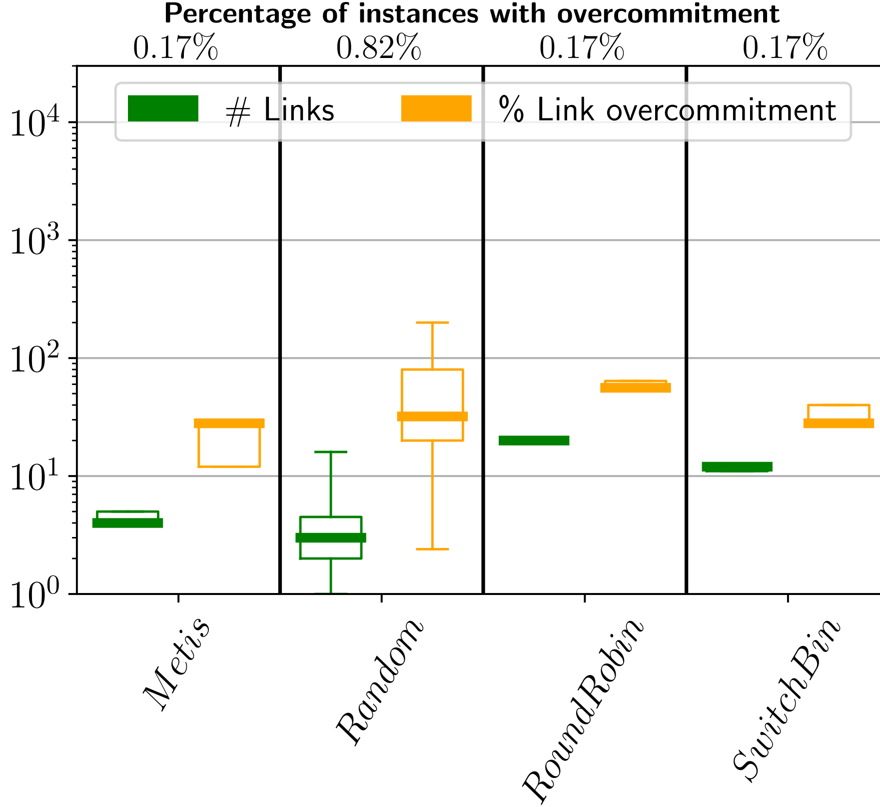


Figure 4.6: *Gros* link overcommitment

when the physical environment is non heterogeneous (i.e., *Rennes* vFT), or when the network to emulate has different vNodes requirements or vLinks requirements (i.e., virtualizing a random network in *Gros*). The worst scenario that we have with `METIS` is in the case of homogeneous infrastructure virtualizing a random topology (i.e., *Rennes* vRD) where only 32.75% of the returned solutions are feasible. We tested the 3 other algorithms that are directly implemented in `Mininet Cluster Edition`. `Random` solves 72% of the instances in *Gros* vFT, while the performances drop drastically in the homogeneous case and when virtualizing a random topology. The same behavior can be observed for `Round Robin` and `Switch Bin`.

Analysis of Solutions not Respecting Capacities.

Even though the existing algorithms do not always succeed to find solutions respecting the physical capacity constraints, they still return solutions. Here,

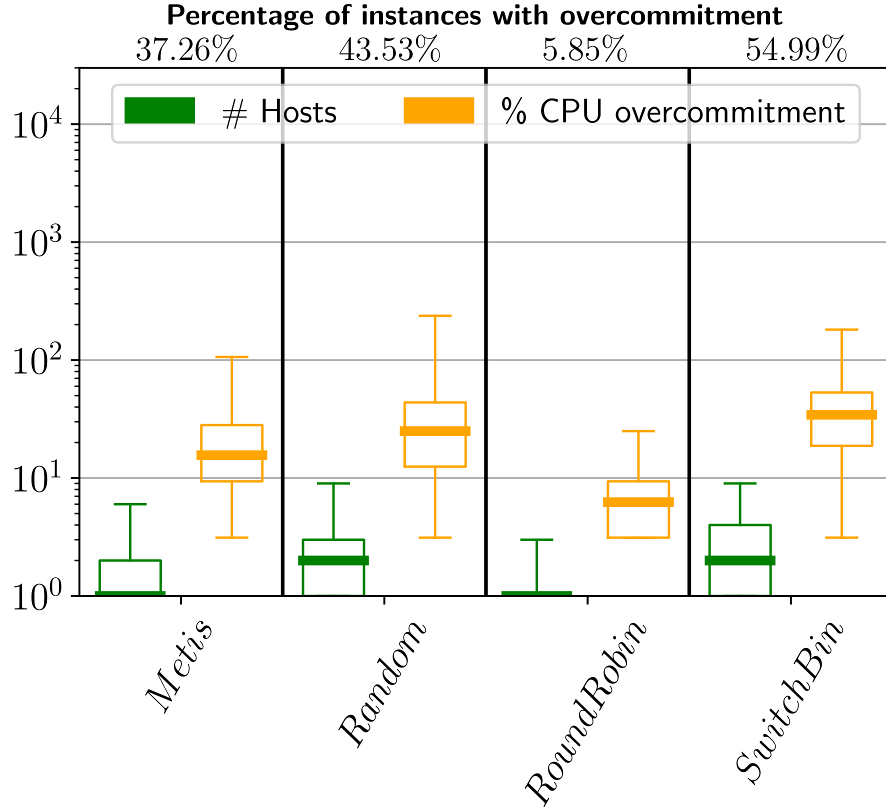


Figure 4.7: *Gros* CPU overcommitment

we study how severely overloaded links and nodes can belong to the computed solutions.

Figs. 4.6–4.11 take into account the virtual instances for which an overloaded solution is returned (in terms of CPU, memory, or link overcommitment) for all the Mininet Cluster and Maxinet algorithms, using the different clusters.

The *overcommitment* is the estimation (in %) of how much the CPU, memory, or bandwidth is assigned in excess to a physical node or a physical link. From the percentages in the plots, we can observe a strong difference in terms of feasible solutions returned for the two different clusters.

If we focus our attention on the link overcommitment (Figs. 4.6 and 4.9), we can observe that less than 1% of the returned solutions lead to links overloaded in the *Gros* cluster. This is expected as the physical topology is a simple star. Conversely, in the *Rennes* cluster, a high percentage of the solutions lead to links overloaded: from 21.7% for Metis to almost 70% for Random, and 51% and 57% for SwitchBin and RoundRobin, respectively. Moreover, for such instances, the overcommitment is important: 4 or 5 links are 100% overloaded in

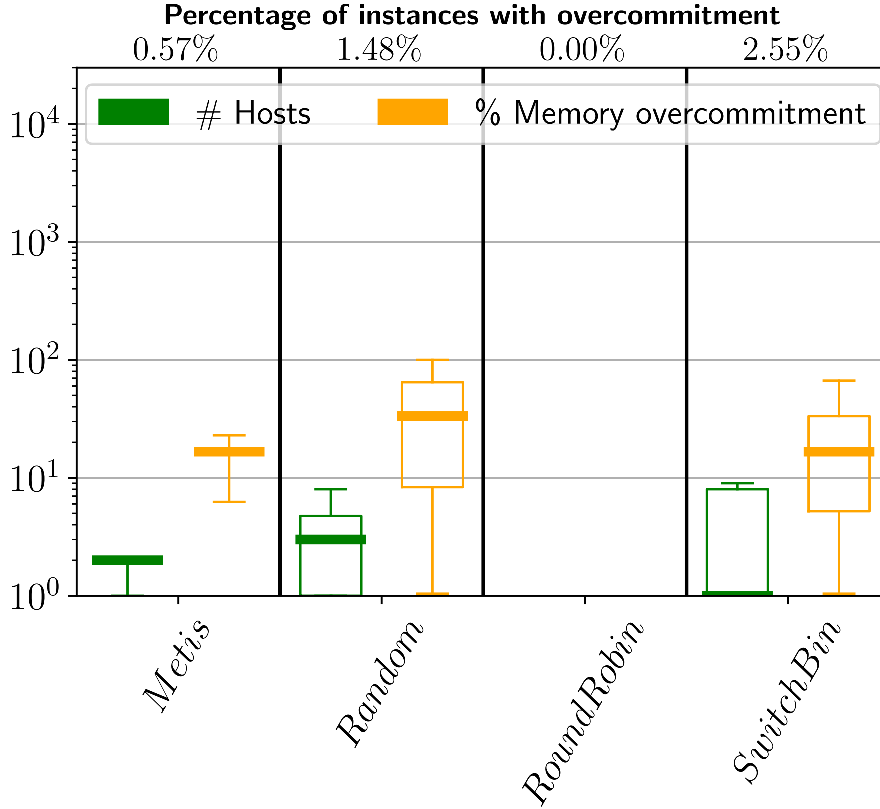


Figure 4.8: *Gros* memory overcommitment

average for Random, SwitchBin, and RoundRobin. The overload factor may reach several hundred percents and even more than 1,000% for some instances with Random. The overcommitment is lower for Metis, the median case being 2 links with a 40% overload. The explanation of this better behavior of the latter algorithm is that it is using a partitioning graph algorithm minimizing the cuts between partitions placed in different machines.

Considering now CPU overcommitment (Figs. 4.7 and 4.10), we see that it is frequent, both for the *Gros* and *Rennes* clusters. RoundRobin is the algorithm handling the best CPU resources: only 5.85% of its returned solutions are overloaded compared to 35%, 43%, and 55% for Metis, Random, and SwitchBin, respectively. The explanation is that RoundRobin tries to distribute the load evenly to the physical hosts. In *Rennes*, Metis behaves a little bit better than Random and SwitchBin with fewer overloaded nodes. Memory overcommitment rarely happens in *Gros* (see Figs. 4.8 and 4.11). In *Rennes*, from 18% to 35% of the instances are overloaded. Metis is the best algorithm in this case. Focusing on Metis, we can see that it returns a solution with more link overload-

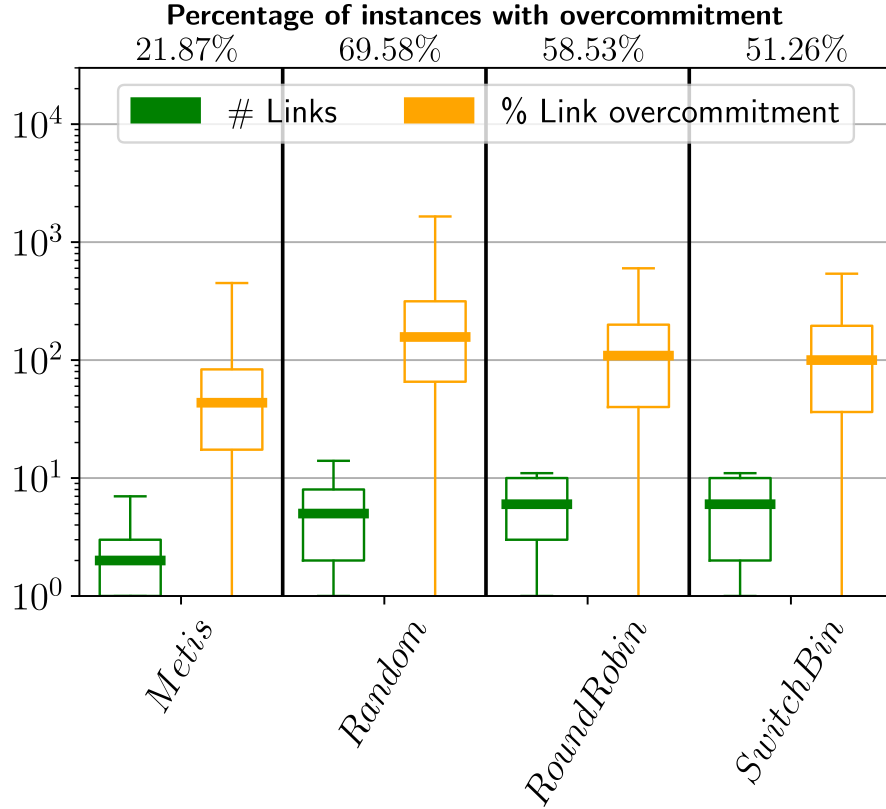


Figure 4.9: *Rennes* link overcommitment

ing when the physical infrastructure is heterogeneous (Rennes cluster), whereas there are not significant differences for the CPU and memory overcommitment. For the algorithms returned by the Mininet Cluster edition algorithms, there are not significant differences in the different physical infrastructures.

Number of Needed Physical Hosts

An important additional advantage of the algorithms we propose is that they minimize the number of physical hosts needed to emulate the experiments. This helps reducing the use of test-bed resources and even making feasible some large experiments that would not be able to run without optimizing hardware usage, as we show below.

We report in Figs. 4.12 and 4.13 the distributions of the number of hosts used by the algorithms over all the virtual topologies for which all algorithms found a feasible solution (INTERSECTION subset). Note that this subset of experiments

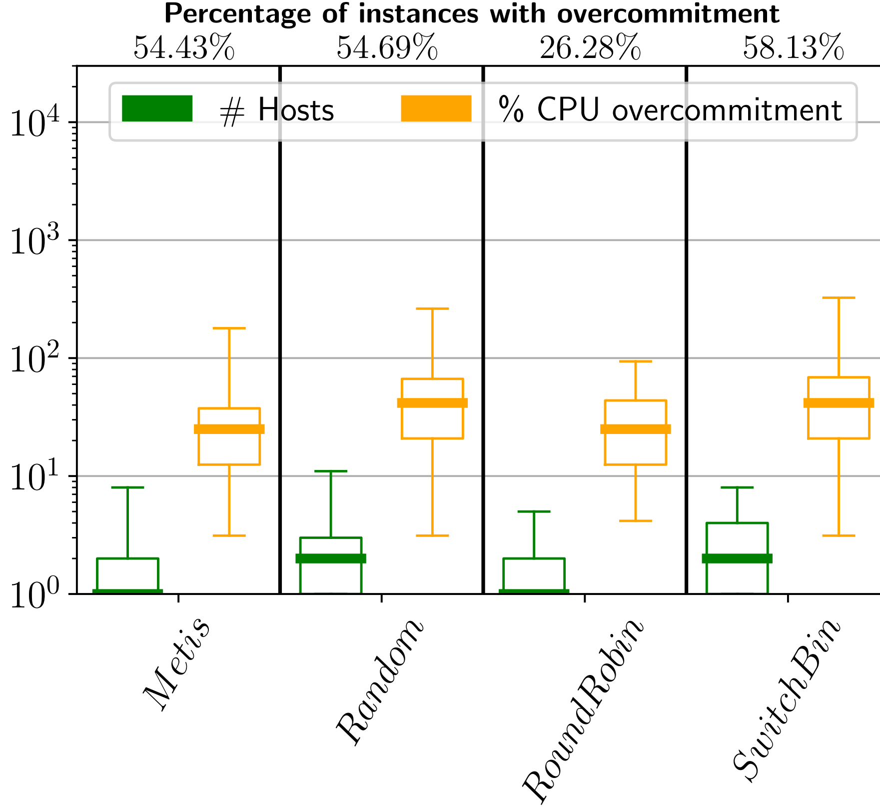


Figure 4.10: *Rennes* CPU overcommitment

does not contain many large topologies, as the less efficient placement algorithms were not able to find solutions for them.

As expected, the proposed algorithms use much fewer physical hosts. For the *Gros* cluster (Fig. 4.12), the general tendency is that GreedyPartition uses between 1 and 13 hosts (median is 3) in case it is emulating a vFT, and between 2 and 11 hosts (median is 5) in case it emulates a vRD. `METIS` uses a minimum of 4 instances with a maximum of 20 instances (medians are 4 and 12). `Round Robin` (medians are 13 and 14), `Random` (medians are 7 and 20), and `Switch Bin` (medians are 5 and 15) use in general more hosts than Metis.

The differences are even more important for the heterogeneous topology, *Rennes*, especially in terms of maximum number of hosts used (Fig. 4.13). The numbers of hosts used by our algorithms are in general lower (e.g., between 1 and 7 hosts for GreedyPartition), while the ones for the existing algorithms are higher (e.g., between 4 and 24 hosts for `METIS`, and between 7 and 26 for `Round Robin`). Indeed, the same virtual topology is harder to solve on a heterogeneous physical topology than on a homogeneous one. So, the set of topologies for which

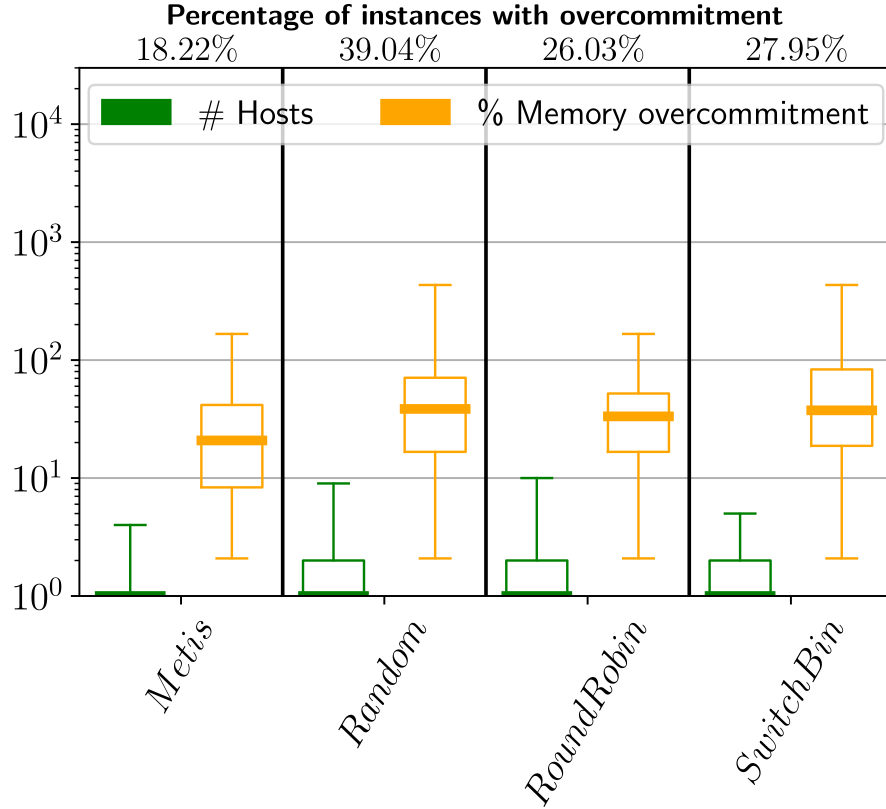


Figure 4.11: *Rennes* memory overcommitment

all algorithms find a solution is smaller for the *Rennes* topology and contains smaller virtual topologies on average. Our algorithms thus find solutions using a lower number of hosts, while other algorithms have difficulty to map them efficiently on the heterogeneous physical topology. Again, GreedyPartition is the best algorithm in terms of resource usage.

Limiting The Number Of Physical Hosts

The algorithms adopted by `Maxinet` and `Mininet Cluster Edition` do not optimize the number of physical hosts, but tend to use all the available hosts. However, the fact that they are using more hosts does not mean that they would not be able to use fewer of them. To check this, we carried out another study. We first computed the minimum number of hosts needed to run the experiment (as given by GreedyPartition). We then put this number of hosts at the disposal of the algorithms used by `Maxinet` and `Mininet Cluster Edition`. We check (*i*) if the algorithm finds a solution and (*ii*) if this solution overloads

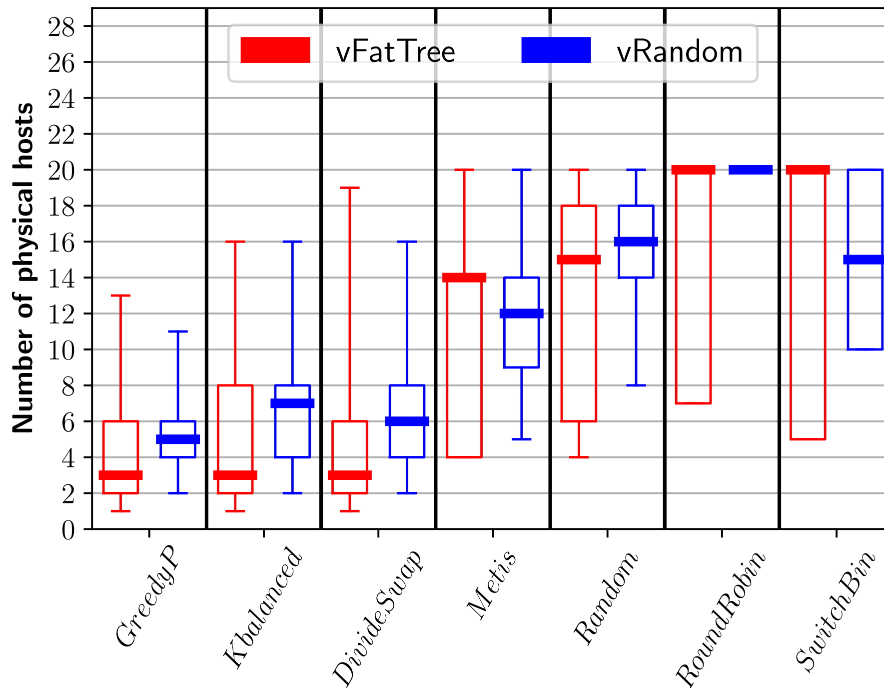


Figure 4.12: Number of physical hosts used by the placement algorithms - homogeneous *Gros* cluster.

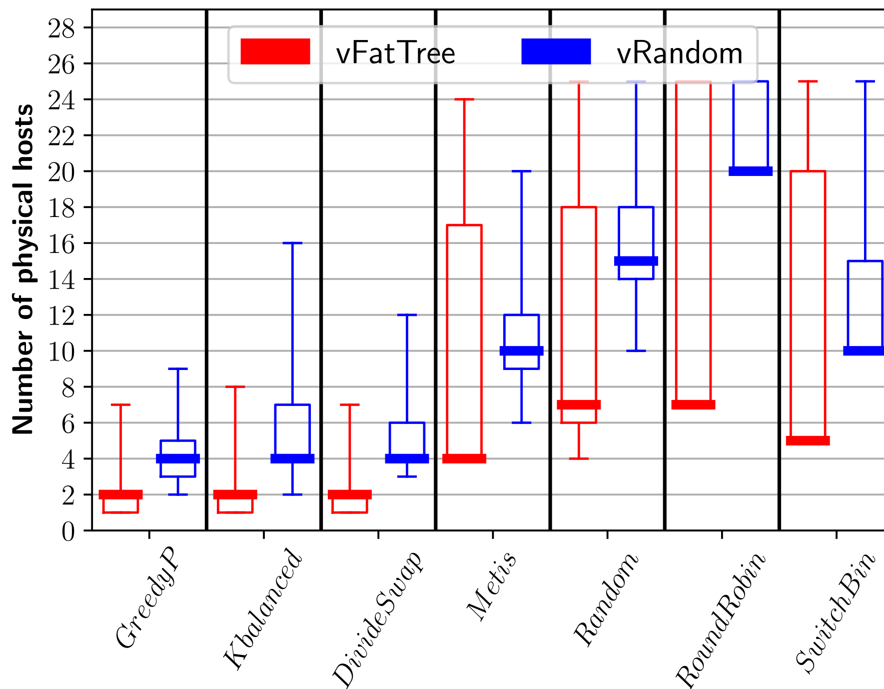


Figure 4.13: Number of physical hosts used by the placement algorithms - heterogeneous *Rennes* cluster.

Alg.	METIS	RANDOM	ROUNDROBIN	SWITCHBIN
vFT	59.42%	33.52%	96.38%	32.0%
vRD	5.51%	2.71%	11.82%	1.40%

Table 4.2: Percentage of feasible solutions found by the algorithms used in `Maxinet` and `Mininet Cluster Edition` when the number of physical hosts to use is set to the minimum one found by GREEDYPARTITION.

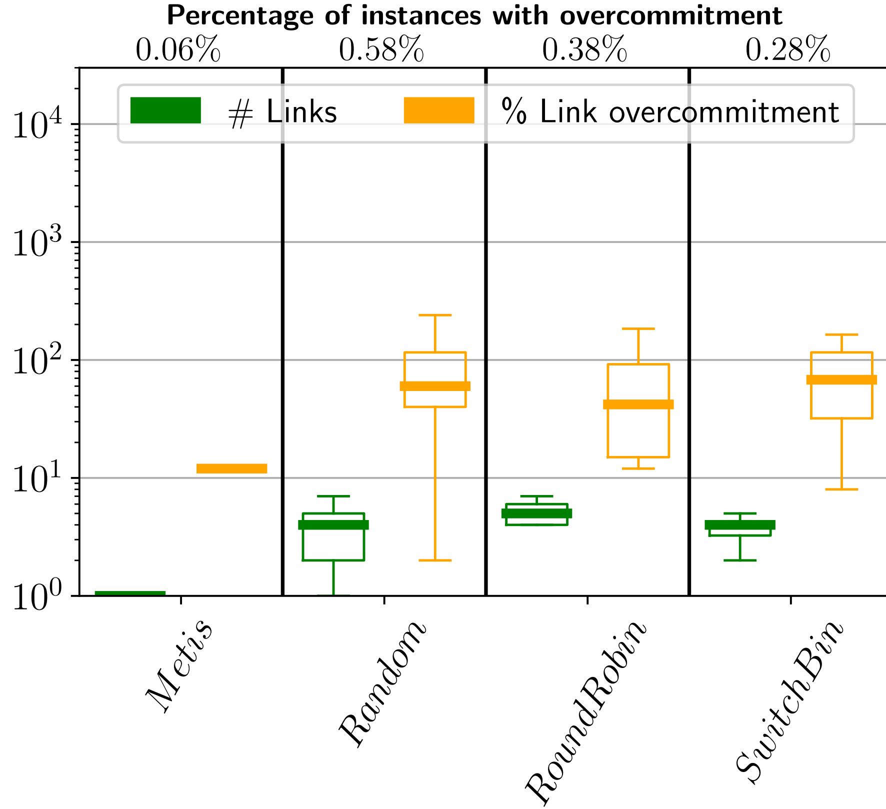


Figure 4.14: *Gros* link overcommitment, with forced hosts.

physical hosts or links. The results are reported in Table 4.2 for vFT and vRD topologies for the *Gros* cluster and in Figs. 4.14, 4.15, and 4.16. We tested 525 different vFT instances and 4,500 vRD topologies. For the vFT, the most favorable scenario for the existing algorithms (homogeneous virtual and physical topologies), the first observation is that `METIS`, `Round Robin`, `Random`, and `Switch Bin` are only able to find feasible (with no node or link overcommitment) solutions with the same number of physical hosts for 59.42%, 33.52%, 96.38%, and 32.0% of the instances, respectively. The percentage is low for `METIS` and

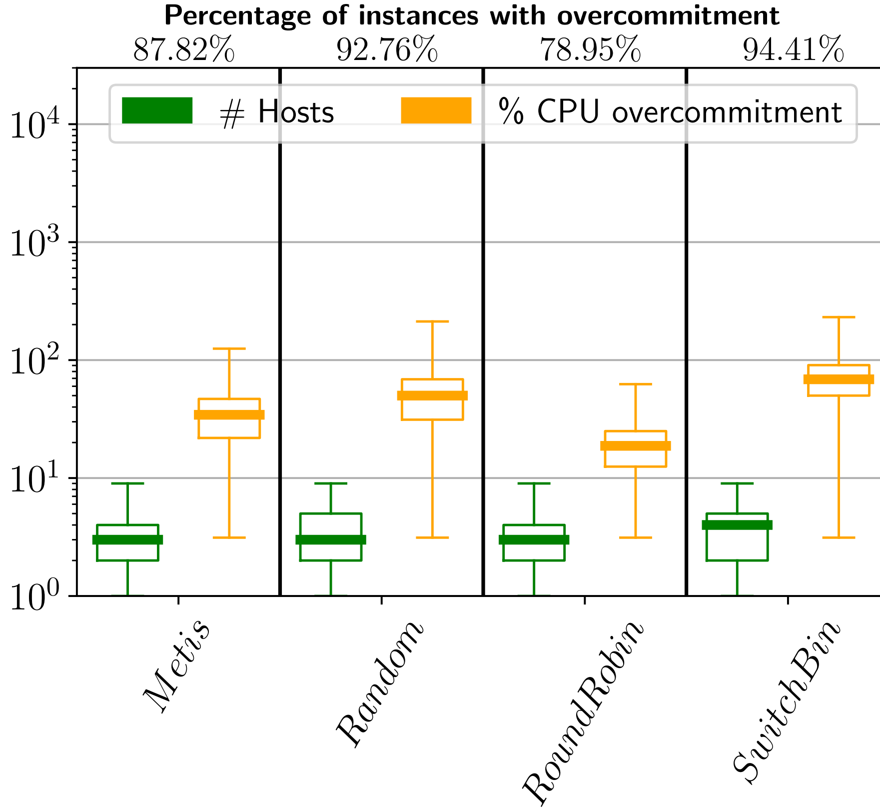


Figure 4.15: *Gros* CPU overcommitment, with forced hosts.

very low for `Random` and `Switch Bin`. For `Round Robin`, the results are good for this scenario because each physical machine has the same capabilities and each vNode has the same requests. In homogeneous cases and when considering the number of hosts found by our algorithm, if the Round Robin strategy returns an unfeasible solution, it is only due to link overcommitment. As the link capacities are very high in *Gros* (25 Gbps), it explains why `Round Robin` is able to solve 96.38% of instances. If we consider now the more complex scenario with vRT, the percentage of solved instances by the existing algorithms drops to values between 1.4% and 11.82%. This shows the efficiency of GreedyPartition to find efficient solutions even in hard cases. Note that this advantage would be even more important when doing experiments on heterogeneous clusters.

We analyze now the solutions returned when no solution satisfying node and link constraints could be found. The goal is to see if such solutions are "close" to be feasible, in the sense that only a node or link is a little bit overloaded. Link, CPU, and memory overcommitments are reported in Figs. 4.14, 4.15, and 4.16, respectively. `Random`, `Round Robin`, and `Switch Bin` do not re-

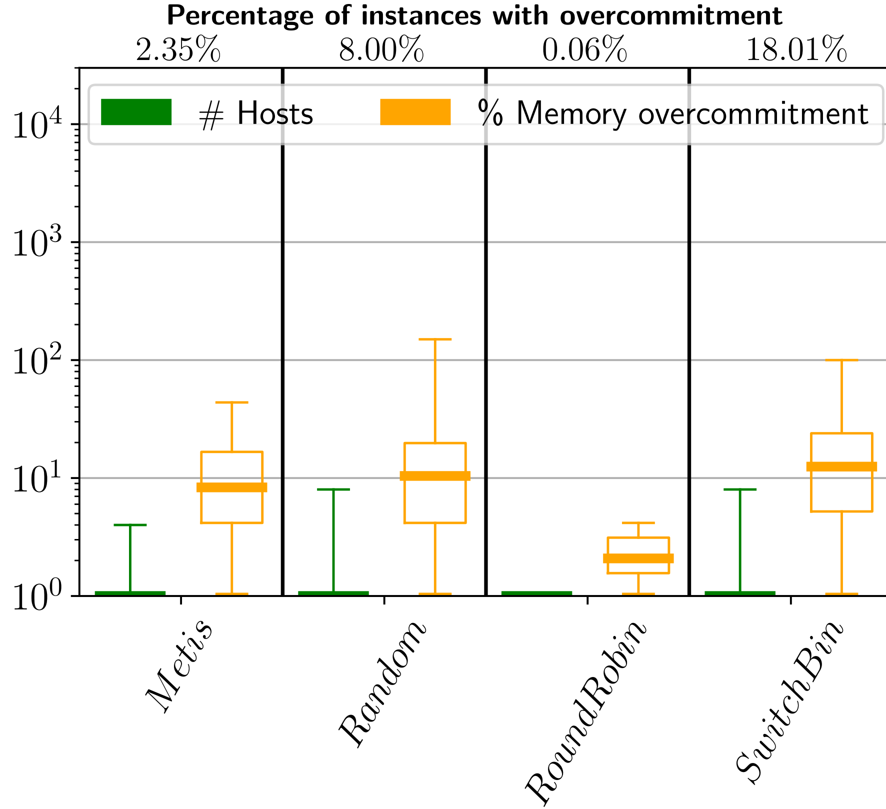


Figure 4.16: *Gros* memory overcommitment, with forced hosts.

turn close solutions when they have some overloaded links. Indeed, they have multiple overloaded links (with a median of 4 and 5) with an overcommitment between 2% and 200%, and median values of 60%, 42%, and 68%, respectively. For METIS, the link overcommitment is lower and involves a single link overloaded at around 10%. The overcommitment in terms of CPU tends to be similar within the algorithms. The number of overloaded hosts is between 1 and 10 (with a median of 3 or 4) and the median overload is between 20% and 90%. The maximum overload reaches more than 100% for three of the four algorithms.

4.5 Overloading Experiment

To show the importance of placement for distributed emulations and to evaluate the impact of violating the limitations in terms of physical resources, we carried out experiments using a placement module implemented in `Distrinet` [Dis19].

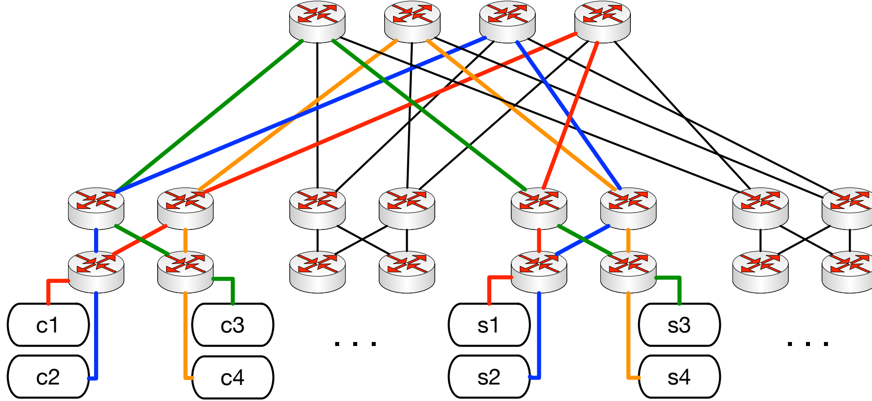


Figure 4.17: vFatTree K=4, bandwidth experiment.

They were performed using Grid'5000 [Bal+13] on two different clusters (*Gros* and *Rennes*). We perform three kinds of experiments: bandwidth, CPU, and memory intensive experiments - discussed in Secs. 4.5.1, 4.5.2, and 4.5.3, respectively. We show the very strong impact of placement on bandwidth usage and, thus, on emulation reliability, as well as the one on crashes and increased emulation completion time due to lack of memory or because of CPU overloading.

4.5.1 Bandwidth Intensive Experiments

The first experiment shows the performances of our placement module in a network intensive scenario. The networks we emulate are virtual Fat Trees with K=4 and K=6. They are composed of vHosts and vSwitches requiring 1 vCore and 1 GB of RAM, while all the vLinks are set to 500 Mbps. Half of the vHosts are clients and the other ones are servers. The experiment consists in running TCP `iperf` [Tir99] between each pair of client/server. The total aggregated demand to be served is 4 Gbps and 13.5 Gbps, while the total network traffic generated is 24 Gbps and 81 Gbps, for K=4 and K=6, respectively. The traffic is forwarded in a way guaranteeing that each client is theoretically able to send at full speed (see Fig. 4.17 for an example). This is possible as a Fat Tree is a permutation network. Each experiment is performed 10 times, i.e., enough to obtain reliable results as the variability is small.
To avoid the installation's impact of the forwarding rules by the controller for the measurements, these rules have been installed manually on each switch before starting sending the traffic. The first half of vHosts acts as a client, while the second half corresponds to the servers. Each vLink is set to 500 Mbps, so in theory each client should be able to send 500 Mbps to its corresponding virtual server, if the routes in the switch are correctly set. To check what the

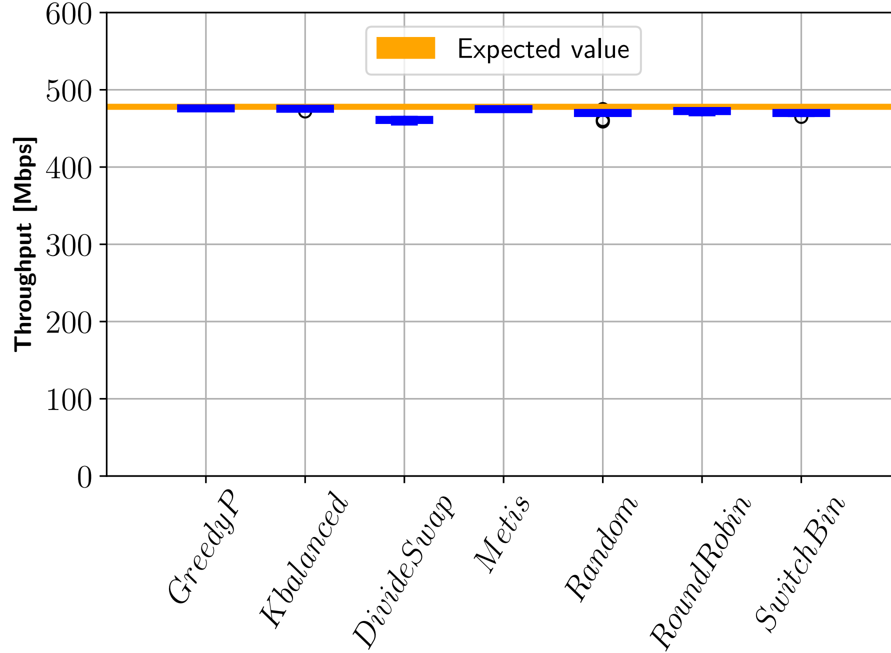


Figure 4.18: Network experiment, *Gros* cluster, vFatTree **K**=6.

expected traffic is, we set up a simple experiment with `Mininet`. We create a simple topology with one switch and two vHosts, and we measure the traffic with `iperf` for 60 s. The result with `Mininet` is 478 Mbps, so we will use this value as a baseline for the distributed experiments.

Homogeneous Case

The first experiment shows the bandwidth measured in the *Gros* cluster (see Sec. 4.4) with 20 physical nodes, virtualizing a vFT with K=6. The physical network in this cluster is a simple star topology with 25 Gbps links connecting all physical nodes to the central nodes. The achieved throughput values for all placement solutions are summarized in Fig. 4.18 by means of boxplots representing the distributions over all client/server pairs. Since each link has a capacity of 500 Mbps, the maximum `iperf` speed is slightly lower than this value (as for Mininet). We observe that in this simple case, all the algorithms return a solution that is not overloading any physical link, nor any physical machine. The emulation is working as expected for each of the studied placement solutions.

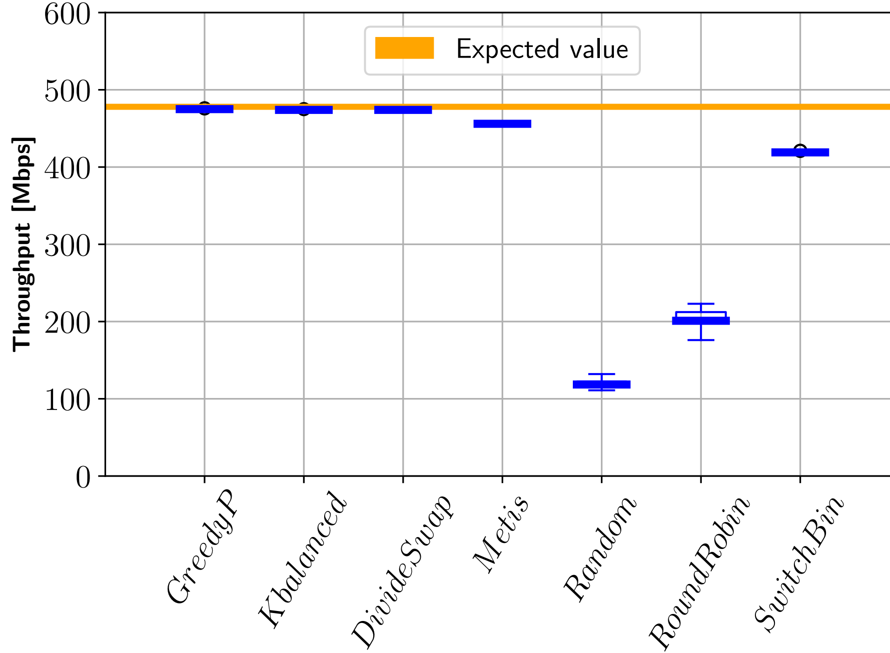


Figure 4.19: Network experiment, *Rennes* cluster, vFatTree **K**=4.

Heterogeneous Case

We now consider the heterogeneous physical (yet simple) physical topology of the *Rennes* cluster (Fig. 4.5). Results using this experimental platform are illustrated in Figs. 4.19 and 4.20. The experiments show how the emulation can return unexpected results when the placement of the virtual nodes does not take into account links' rates. As this topology is not homogeneous like the *Gros* cluster, finding a good placement is significantly harder, as discussed in Sec. 4.4.

The first test in the *Rennes* cluster consists in emulating a vFT topology K=4 using all the algorithms. In this case, the emulation creates a vFT with 16 vHosts and 20 vSwitches (Fig. 4.17). Fig. 4.19 reports the bandwidth performance of a vFT (with K=4) obtained for the different placement algorithms when the emulation is performed in the *Rennes* cluster. As we can observe, the bandwidth results using GreedyPartition, k-balanced, DivideSwap, and Metis are the ones expected from the emulation, while the results obtained by other algorithms, Random and RoundRobin, are far from the expected ones. Indeed, some of the links are overloaded in the placement returned by the latter algorithms. This means that the paths of two demands are using the same links. For these links, the throughput drops to 120 Mbps and 200 Mbps, respectively. When running a larger emulation for a vFT with K=6, we can

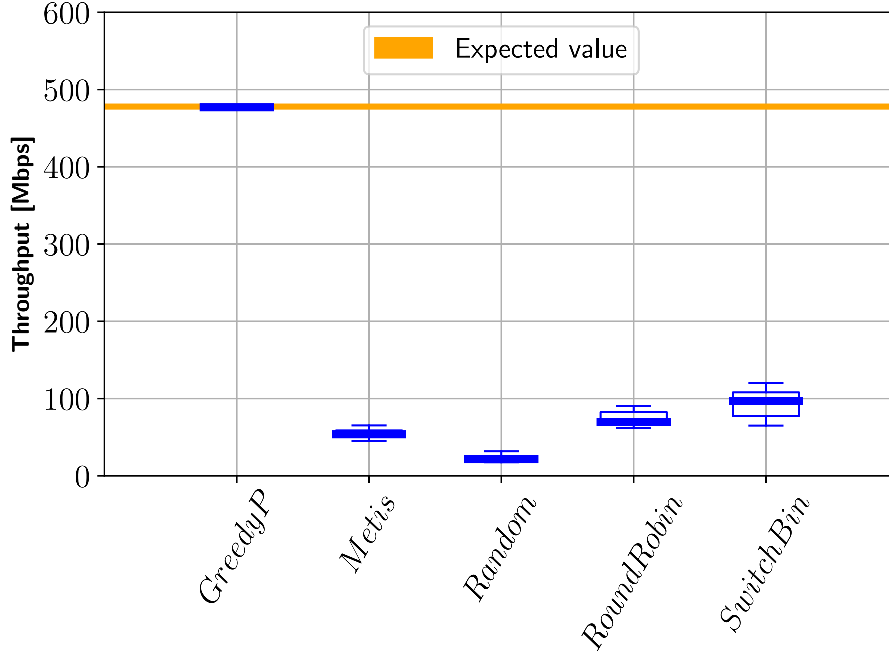


Figure 4.20: Network experiment, *Rennes* cluster, vFatTree **K**=6.

observe in Fig. 4.20 that k-balanced, DivideSwap do not find a feasible solution and the results returned by Metis and SwitchBin are not trustworthy anymore. The measured throughput on some overloaded links has fallen to very low values between 20 and 100 Mbps, to be compared with the expected 478 Mbps. On the contrary, the emulation distributed with GreedyPartition returns exactly the expected results, showing the efficiency and reliability of the proposed algorithm.

4.5.2 CPU Intensive Experiments

We now study the behavior of emulations for which a placement algorithm returns a solution with CPU overcommitment. We evaluate the impact on emulation execution time.

We built a CPU intensive scenario using Hadoop Apache [Had20a]. The cluster used for this test is *Gros*, and the virtual topology is again a vFT with K=4. One vHost in the vFT is the Master, while the other vHosts are the Workers. The experiment consists in running a classical Hadoop benchmarking test. This test computes π using a quasi-Monte Carlo method and MapReduce, with 2,000 maps and 1,000 samples for each map. Each vHost requires 24 vCPUs and 32 GB of RAM, while the vSwitches only require 1 vCPU and 1 GB of RAM. Each machine of the cluster provides 32 vCores and 96 GB of RAM.

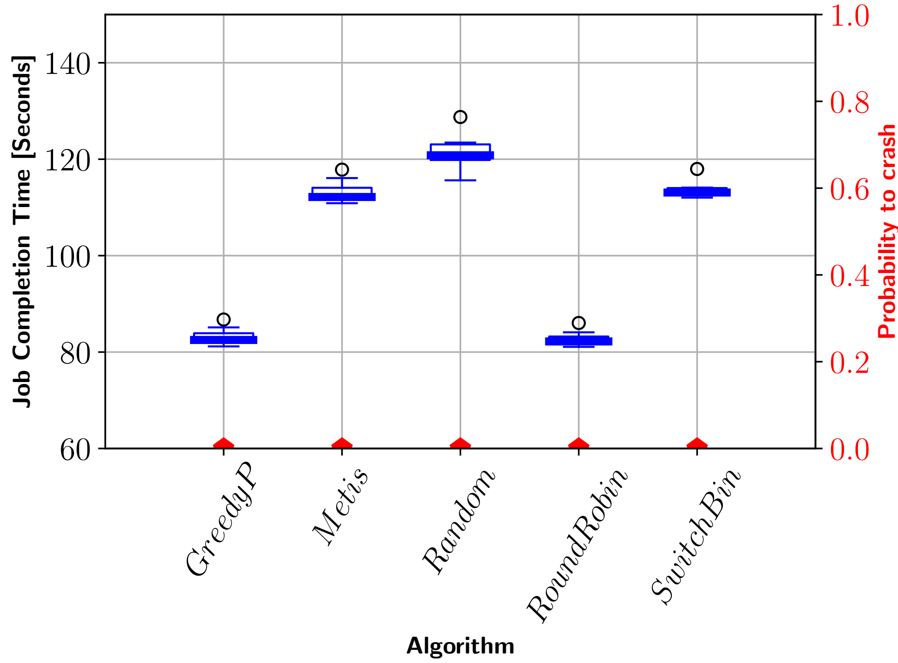


Figure 4.21: CPU experiment, *Gros* cluster, vFatTree **K**=4.

Fig. 4.21 reports the Hadoop Job Completion times for each placement algorithm. The boxplots provide their distribution over 10 experiments. The best performances are the ones of GreedyPartition and RoundRobin. The reason is that their solutions are not overloading any physical machine, nor link. On the contrary, Metis and SwitchBin return the same placement overloading 8 physical machines in terms of CPU by 50% (48 vCores are assigned to 8 hosts with 32 cores available). The impact of this overcommitment on job completion time is an increase of around 40% as seen in the figure. Random returns a different placement for each experiment. For most of them, at least one physical machine hosts 3 virtual hosts leading to a CPU overcommitment of 100%, explaining why Random has the highest job completion time.

4.5.3 Memory Intensive Experiments

For the memory intensive tests, we create an experiment in which the nodes run at the same time an in-memory file storage and the Hadoop benchmark computing π used in the previous section. The first application is is memory-hungry, while the second application is used to assess the impact of memory overload on execution time in case the experiment does not crash due to a lack of memory. We choose the parameter of the experiments to ensure that each algorithm may return solutions with only memory overcommitment, but not

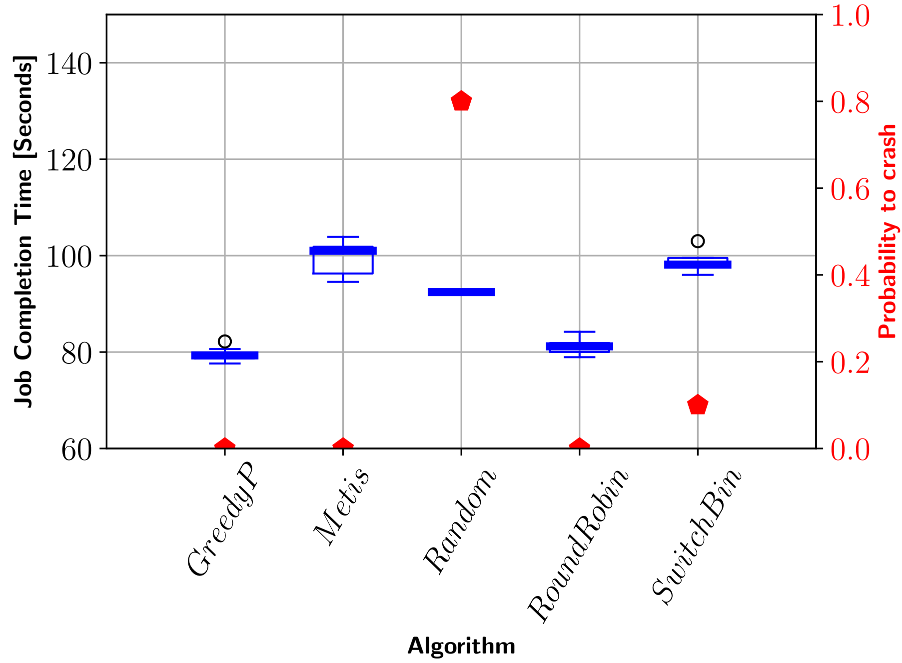


Figure 4.22: Memory (swap) experiment, *Gros* cluster, vFatTree **K**=4.

CPU or network ones. We test a scenario with low memory overcommitments, as large ones would induce direct crashes. We consider two scenarios for the impact of memory overcommitment: swap memory enabled or disabled in the physical machines. Note that `Distrinet` is dynamically allocating virtual memory. So, if the vHost is not using it, it is available for other vHosts or for other tasks in the hosting machine. The experiment creates a vFT with K=4, in which each vHost requires 12 vCPU and 50 GB of RAM. A physical machine has 32 cores and 96 GB of RAM. Hence, if we assign 2 vHosts that use all 50 GB of RAM, the physical machine is overloaded in terms of memory by 4.2%. In these tests, just like for the CPU overcommitment ones, one vHost is the Master, while the other vHosts are the Workers.

Swap Enabled Case

The physical machines also provide 4 GB of swap memory. So, if the RAM is completely full (depending on the swappiness parameter in the kernel, we use the default value for the tests), the machine starts to use the swap memory (*Gros* cluster uses SSDs for the storage).

Fig. 4.22 presents results using the different placement algorithms. In this case, we observe that overcommitment slows down the job completion time using METIS. We also notice that a large fraction of runs crashed, especially

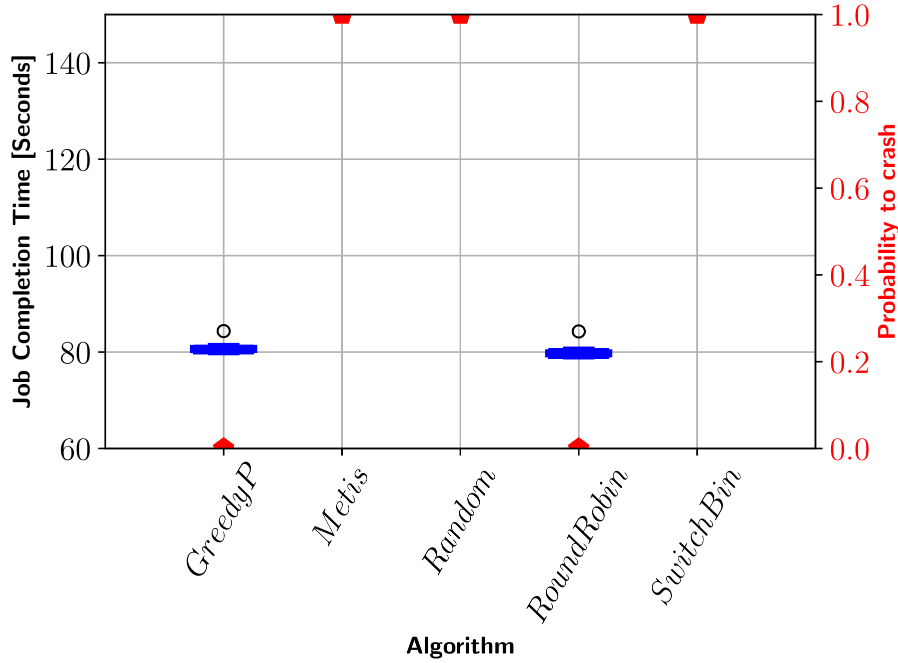


Figure 4.23: Memory (no swap) experiment, *Gros* cluster, vFatTree **K**=4.

using RANDOM (80% of the experiments did not succeed). This is due to a bad assignment of resources made by the algorithm (often 3 vHosts were assigned to the same physical host, leading to a memory overcommitment of 56%).

Swap Disabled Case

When the swap memory is disabled, if the assignment overloads the memory, the machine cannot rely on the SSD memory. In this case, we see in Fig. 4.23 that, with only 4% of memory overcommitment, the emulation cannot run with METIS or SWITCHBIN. We observed during the same experiment different behaviors. Sometimes, `Distrinet` containers simply crashed, while other times the in-memory files generated were corrupted during the emulations. In both cases, the emulation is considered crashed. Similar to the experiments with swap enabled, GREEDYPARTITION and ROUNDROBIN do not overcommit the physical machines and manage to complete the task without issues.

4.6 Conclusions

In this chapter, we propose a placement module for tools enabling distributed network emulation. Indeed, large scale or resource intensive emulations have to

be distributed over several physical hosts to avoid overloading hosts carrying out the emulation. A network experiment can be seen as a virtual network with resources needed for nodes (e.g., CPU or memory) and links (e.g., bandwidth). This network has to be mapped to the physical clusters on which the emulation is done. As we show previously, a bad mapping may lead to overloaded physical resources leading to untrustworthy experiments. This is the reason why we propose and evaluate placement algorithms that provide trustworthy mapping, ensuring that resources are never overloaded. As a bonus, our algorithms also minimize the number of physical machines needed. We show that they outdo existing solutions in all aspects. With the ever-growing need of resources for experiments, we are convinced that having such a fast, efficient, and trustworthy placement module is essential for the community.

In this work, we consider the case of private test-beds for which we know and control the infrastructure. However, not all emulations are done on such platforms. Experiments may be done in private test-beds for which the infrastructure is known but the control is not total (e.g., the routing cannot be chosen), or in public clouds (e.g., Amazon EC2 or Microsoft Azure) in which the characteristics of available machines are known, but not the network interconnecting them. It would be interesting to study how to adapt the methods and algorithms developed in this work to scenarios in which the knowledge of, and the control over, the experimental infrastructure is partial.

Chapter 5

Cloud Measurement

5.1 Introduction

Cloud computing provides convenient on-demand access to a potentially unlimited pool of computing resources. In recent years, cloud computing resources have become cheaper and more powerful, with the growing interest of companies around the world. Cloud computing brings several advantages: a small company has little or no capital expenditure (CAPEX) when launching a new product, instead of having to invest in datacenters and servers before knowing how much these resources are going to be used. Cloud computing is flexible and on-demand. This means that the users can consume computing resources and only pay for the amount of resources they actually consumed. There is no necessity to spend money on running and maintaining a datacenter since only cloud providers have access to the physical equipment.

On a high level, cloud providers offer three service models:

- **IaaS** (Infrastructure as a Service): the cloud provider is responsible for managing the physical network, the servers and providing a virtualization layer to isolate the environment of each user. Responsible for all the other layers in the application stack, the user needs to make sure that the guest OSs are updated, and check that the applications are correctly configured. The user is also responsible of the management of the (virtual) network used by the virtual instances. Examples of IaaS services are Amazon EC2 or Microsoft Azure.

- **PaaS** (Platform as a Service): the cloud provider is in charge of all the layers managed in the IaaS service model, and it is responsible for the Operating System layers, and the development platform which is offered. Examples of PaaS services are Fargate from AWS (which provides an environment for docker containers completely managed by Amazon), or Amazon Simple Queue Service (which provides a simple way to manage and scale multiple thousands of requests).

- **SaaS** (Software as a Service): the provider is responsible for managing all the layers of the application stack, while the user utilizes the application without any responsibility on the application stack. Examples are Google docs, or Microsoft One Drive.

In this work, we are focusing on IaaS provisioning, in particular we provide a brief overview of the main component of AWS that is responsible for this cloud model which is Amazon Elastic Cloud Computing (EC2). We describe all the virtual devices involved in the creation of a virtual infrastructure, and we apply the best practice strategies to deploy our experiments in a regional and multiregional environment.

Amazon AWS and, in general, all the major cloud providers offer resources for each instance flavor (e.g., t3.medium, m5.large, etc.), describing how many vCores and how much RAM are assigned. Regarding networking, AWS provides the maximum throughput that a flavor is able to generate (from 500 Mbps to 100 Gbps). AWS does not provide the maximum delay inside the infrastructure. It only mentions that all the regions and the availability zones are connected with a high performance optical fiber network.

Some applications are delay-sensitive, and for such a type of scenarios, the user should be able to check if the application can be deployed in two different regions, or two different datacenters geographically separated.

The goal of this work is to provide CloudTrace, a Command Line Interface (CLI) that can measure the network delay between two, or more, different networks in AWS using different strategies.

Since there are more than 10 different regions, we present the experiment results running only on 3 distinct regions (London, Dublin and Frankfurt). Interested readers can easily perform their experiments in other regions following the instructions at `https://github.com/Giuseppe1992/CloudTrace`

5.2 Related Work

Since AWS launches EC2 service [Ama20d], multiple approaches have been proposed to measure computing or network performances in virtualized environments and cloud infrastructures. Most of them focus on the impact of a virtualized environment with different hypervisors [RR14], virtualization using Virtual Machines and containers [Li+17], or the network performances variability in virtualized instances [GBU19]. One of the most convincing measurement tools for the cloud environments is Cloudbench (CBTOOL) from IBM [IBM20]. It is a tool that automates testing of Cloud providers. The tool is able to automate experiments for a large range of workloads (e.g., Hadoop [Had20b], Redis [Red20]).

In [JVU19], the authors performed large-scale traceroute measurements over the global AWS network infrastructure. The traceroutes were performed between 15 regions, and the goal was to study the complexity of the AWS infrastructure. The work is done using traceroute with different protocols (TCP,

UDP, and ICMP) using different source ports and the default destination port. The study does not take into account how the delay and the path are changing at different times of the day like in our study. The work in [Arn+20] covers more generally the major cloud providers and the main Tier-1 ISPs: this paper shows how Internet traffic is more and more generated and transmitted in the private interconnection between the cloud providers, thus bypassing the Tier-1 ISPs.

Migration of applications to the cloud is non-trivial. One of the first works trying to automate the process is CloudGenius in 2012 [MR12]. CloudGenius automates the decision-making process using models for web server migration to the cloud, taking into consideration various constraints of the application, like QoS, image types, costs, etc. Since 2012, the number of services and complexity have exploded, and the cloud providers tend not to disclose the details on network performance. The customers thus suffer from highly variable and unpredictable network performance [MP12], [Pal+19]. [Gar07] evaluates AWS computing and networking virtual resources' performance, by analyzing its limitations and the APIs. The work evaluates the AWS security features, the DNS, Simple Queue Service (SQS) [Ama20g] , and EC2 services.

In [Ber+13], the authors evaluate the AWS network performance through the passive analysis of network traffic collected from a single university campus and three large Points of Presence (PoP) of an Italian national-wide ISP for 60 days. The main services tested are EC2 , S3 [Ama20f], and Cloudfront [Ama20b], the Content Distribution Network that Amazon built to provide services closer to its customers.

In [Per+15a] and [Per+15b], the authors analyzed the cloud performance in Amazon AWS and Microsoft Azure, with more than 800 hours of experimentation. In particular, they studied what is the maximum network throughput achievable between two VMs deployed on Microsoft Azure and how it changes over time, across different scenarios.

Another problem is to define what is the network performance metric to be considered when evaluating cloud infrastructures. Such research is done in [MP12], where the authors review the proposals for providing performance guarantees within cloud networks.

A metric to be considered when evaluating cloud performances is the effective bandwidth assigned by the cloud provider. The authors in [HX18] experimented public clouds with the Available Bandwidth Estimation Tools (ABETs). They prove that the tools do not correctly estimate the available bandwidth of public clouds. The main reasons are the rate limitations of network virtualization, the packet sizes of the traffic generated, and the overhead of VM scheduling when virtual instances are deployed in the physical servers.

An interesting scenario where it is critical to have low delay is cloud gaming. The users do not need a powerful machine or a console to play as the providers use cloud processing to run the game. In this case, the network delay cannot exceed few milliseconds in order to be playable. While playing, the users are constantly sending input commands to the game that is running far from the user's location. The gaming service has to send back the output of the game in

real time. In [Cho+12], simulations demonstrate that end-users geographically distant from the datacenter running the game experience unacceptable latencies. This work was done in 2012 and with the rising of edge computing now, cloud gaming becomes a reality, with services like Microsoft project xCloud [Mic20] or Google Stadia [Goo20].

Finally, in [UTK16], the authors propose a benchmarking methodology based on latency measurements collected via active probing of cloud resources. The measurements are done at multiple layers of the network protocol stack. They focus on latency as it is among key performance parameters for the vast majority of applications. They do not provide any tool to automatize the task in case a user wants to personalize an experiment, and the project was closed in 2017.

Unfortunately, all these propositions do not provide a simple way to automatically deploy a testing infrastructure in a regional or multiregional environment for studying the network delay's behavior.

5.3 Amazon Web Services Infrastructure

The AWS global infrastructure is composed of 2 different entities: **Region** and **Availability Zone (AZ)**.

The Availability Zone is a set of one or more datacenters connected by redundant high-speed networks and power in order to guarantee high availability to the users.

The region represents a set of multiple AZs clustered in a specific area around the world, within 100 km from each other. The AZs in a same region are interconnected with high-bandwidth, low-latency, high-throughput networks over fully redundant, dedicated metro fiber. To guarantee privacy, all traffic between AZs is encrypted. Services like Simple Service Storage (S3) require replication between different AZs in order to guarantee 99.999999999% durability and 99.99% availability of objects over a given year [Ama20f]. To achieve such requirements, S3 performs synchronous replication between AZs of a single region via a high-speed network. AZs make partitioning applications for high availability easy. If an application is partitioned across AZs, companies are better isolated and protected from issues such as power outages or natural disasters [Ama20a].

AWS provides hundreds of different services. We describe briefly the main services and tools provided for IaaS. The first layer of an IaaS in AWS is the **Virtual Private Cloud (VPC)** that creates a virtual network defined by the user. All the virtual instances and the virtual devices (i.e., Virtual Gateway or route tables) should be created within a VPC.

The VPC is composed of one or more **subnets**. The user managing the subnets can create as many subnets as desired in a single VPC till the address space in the VPC becomes empty. To control the network flows in the network subnet, the user can create **route tables** associated with them.

If the instances are in the same VPC, it is possible to connect them directly using the private IP if they are in the same subnet. If they are in a different

subnet, the routing table should be configured before the connection (Fig. 5.1).

By default, the VPC does not provide any way to connect the instances to the external network. The **Virtual Internet Gateway** is a highly scalable virtual device managed entirely by AWS that allows traffic to go from the VPC to the outside world, and vice-versa. Once created, the routing table should be updated to redirect traffic to the subnets.

Two other main services are **Network Access Control List (NACL)** and **Security Group (SG)**. NACL is a stateless firewall associated with a subnet. By default, the rule is to DENY the traffic. In order to enable the traffic in the subnet, the ALLOW rule associated with a specific flow should be appended to the NACL. SG is similar to NACL. It is a stateful firewall associated with an interface: it can be a virtual interface of an instance, or another virtual device that has an IP created in the VPC, i.e., virtual load balancer.

There are three different type of addresses in AWS: private, public, and **Elastic IP**. Elastic IP is a public address that can be conveniently attached to, and detached from, a virtual interface at any time. Once created and attached, Elastic IP is not released, even if the user deletes the instance or the interface. The address should be explicitly released by the user when it is not needed anymore.

In some scenarios, the user needs to connect instances or services between different VPCs. This can be achieved, of course, using the public IPs of the interfaces. This solution comes with various side effects. First of all, the traffic is going through the public Internet, and it is a risk in terms of security and privacy. The other issue of this solution is that if an application doesn't need a public IP but has to communicate with another service or instance in another VPC, then it is forced to have a public IP. To solve this issue, AWS proposes the **VPC peering** service [Ama20h]. With VPC peering, it is possible to connect two or multiple VPCs. The instances in the peering can connect to each other via private IP, like they are in the same network. Amazon ensures that the traffic going via the peering never leaves the AWS network, increasing security and performances.

The last important concept to describe before presenting CloudTrace is **Elastic Compute (EC2)**. EC2 is a service that provides secure and resizable compute capacity in the cloud [Ama20d]. EC2 provides a large number of instance types. The user can choose the instance type that fits better the targeted scenario. Some instances are compute-optimized, with high-performance processors ideally used for scientific modeling and High-Performance Computing (HPC). The memory-optimized instances are able to process large data sets in memory: this type of instances is well suited for the in-memory database (i.e., Redis), GPU-optimized for machine learning workloads and many other types of instances [Ama20c]. For our experiment, we use two different architectures (Figs. 5.1 and 5.2). Fig. 5.1 describes a regional infrastructure composed of two subnets created in two different AZs within a single VPC, i.e., in a single region. As explained before, the Internet Gateway is responsible to allow connectivity with the external Internet. An instance is created on each subnet, and the flow table is stored in a single routing table, serving the two subnets. The instances

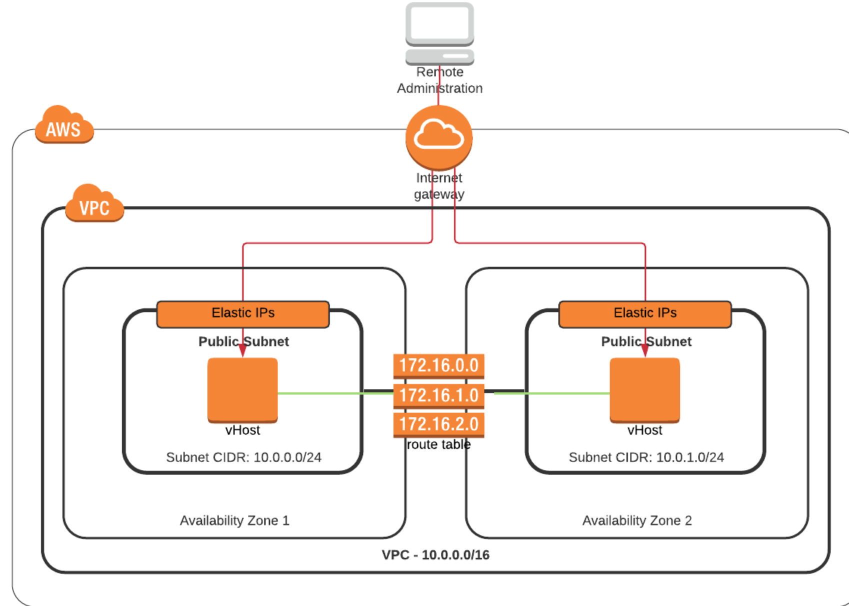


Figure 5.1: Architecture in a regional deployment

can connect to each other via the public IPs (an elastic IP is attached on each interface of the instance), or via the private IP since they are on the same VPC.

In Fig 5.2, the infrastructure is shared between two different regions. The architecture is similar, and the instances can still use the public IP to connect to each other like in the regional environment, but they cannot use by default the private IPs. In this case, a VPC peering connection between the two VPCs needs to be created to allow the instances to use the private IP to connect them as if they were on the same network.

CloudTrace is generating the infrastructure in both cases, with two or more AZs in case of regional experiments, and two or more regions in case of multi-regional experiments.

Our implementation choices are discussed in the next section.

5.4 Implementation

CloudTrace is open-source, and it is compatible with Linux and MAC OSX, while it is possible to install it on Windows via Docker. The tool is built using LiteSQL, Ansible, Paris-Traceroute, and the Boto3 API for automatic deployment in Amazon AWS. The basic idea is to have a CLI that takes as input the regions and the type of experiment that the user wants to run, and creates an environment performing multiple traceroutes between the virtual instances.

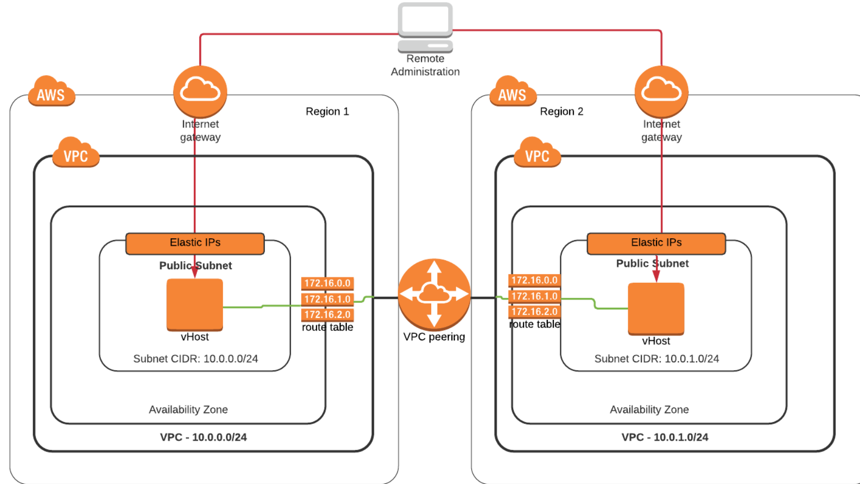


Figure 5.2: Architecture in a multiregional deployment, with VPC peering

First of all, CloudTrace creates a unique ID for the experiment (a string composed of 8 alphanumeric random characters). Then, depending on the type of experiment, it performs the following steps.

Regional Experiment. When creating a regional experiment, the user has to specify only one region. CloudTrace first checks if there is at least one VPC slot available in the region, then creates the VPC with subnet *10.0.0.0/16*. Finally, it creates a *10.0.X.0/24* subnet for each Availability Zone (AZ) available in the region. After setting up the subnets, it creates an Internet gateway and attaches it to the VPC. It then creates a routing table and appends all the routes necessary to allow external connectivity to the future instances created within the subnet. To allow Internet connectivity, adding the route is not enough. CloudTrace also adds the rules in the NACL. After configuring all the subnets and the NACL rules, the user creates an instance in each subnet. For each instance, it creates a Service Group that allows connectivity to the external networks. Finally, the tool waits that all the instances are in a RUNNING state and returns all the information. All these interactions are performed using the Boto3 API, and all the information returned by AWS are saved in the LiteSQL database of the client where the tool is running.

Multiregional Experiment. The setup of the infrastructure is similar to the regional one. First of all, CloudTrace connects with each region specified by the user and checks if there is at least one VPC slot for each region. If yes, it creates a VPC, a subnet and an Internet gateway in each region in the default AZ, and configures them like in the regional experiment. The VPC and the virtual subnet have the same subnet *10.0.X.0/24*, because in case of VPC peering, the subnet cannot overlap.

If the user selects the option to use VPC peering, a peering connection between all the VPCs are created, and all the route tables are updated, in order to forward the traffic via the private IP between remote VPCs (Fig. 5.2).

All the information needed is saved in the client database (i.e., public IPs, VPC ID, regions, instances ID, etc.). CloudTrace parses all this information and creates for each experiment an Ansible **inventory** (a file that describes the login info and the addresses of the instances). For each experiment, Ansible runs a standard **playbook**, a YAML file that lists all the requirements the instances have to install and all the commands that each instance has to execute before running the experiment, e.g., update the kernel. All public IPs of the instances are saved in the client database. CloudTrace also creates a host configuration file for Ansible.

The process described above summarizes the creation process of the experiment, but the experiment does not start by default. Once all the requirements are installed, the environment is ready for the experiment. The user can easily start the traceroute experiment with the *start* command followed by the ID of the experiment. Once the user decides to start the experiment, the tool creates a traceroute script for each instance. This script runs Paris-traceroute on the default source and destination ports of all the other instances in the experiment. For example, if the user creates a multiregional experiment with 4 regions listed, each instance has to perform 3 traceroutes in parallel (one for each other instance). If the user decides to run the experiment via the public IP, the traceroute file uses the public IPs of the instances. Otherwise, it uses the private ones. The scripts are copied on the virtual instances on AWS. Ansible uses SSH to connect to the instances, and the login configurations are saved in the inventory file of the experiment.

After copying the files in the remote instances, Ansible configures *crontab* to run the script each minute and saves the file in a specified directory in the machine. The process is run in parallel using the Ansible *–fork* option. At any moment, the user can retrieve the traceroutes from the instances, using the *retrieve_data* command. This command will not stop the experiment in the instances, so the user can retrieve the new data without interrupting the experiment. Ansible compresses the data on the remote instances before sending them back to the client, in order to decrease the retrieving time (it is also made in parallel with the fork command).

Once the data are retrieved in the client, the user can analyze it manually or can use CloudTrace for a simple analysis of the data.

CloudTrace can also plot a simple and interactive network map that describes how the delay varies around the globe. The map is created using Plotly, Folium, and Dash Python libraries. Fig. 5.3 is an example of the delays obtained between 3 regions.

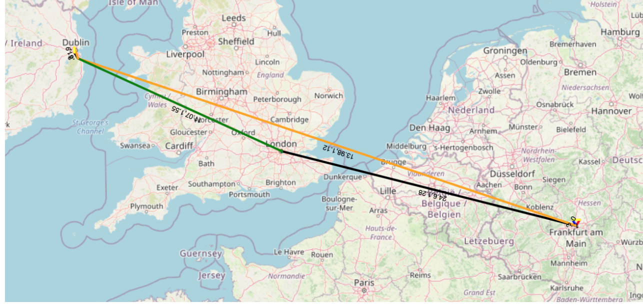


Figure 5.3: Map example for a multiregional experiment

5.5 Experiments

We present multiple experiments, tracing the network performances during one month in Amazon AWS. The tests are made on 3 AWS regions: Frankfurt (eu-central-1), Dublin (eu-west-1) and London (eu-west-2). The experiment ran from 04/11/2020 till 04/12/2020, and we collected in total more than two millions of traceroutes.

Regional Experiments. We did these first experiments in all the Availability Zones of the three regions chosen above. The regional experiment creates a virtual instance in each Availability Zone and performs traceroutes between them. The first region we take into consideration is Frankfurt, with the traffic going from eu-central-1a to eu-central-1b (Fig. 5.4) and eu-central-1b to eu-central-1c (Fig. 5.5). In this first experiment, we present the result using the private IPs to perform the traceroute.

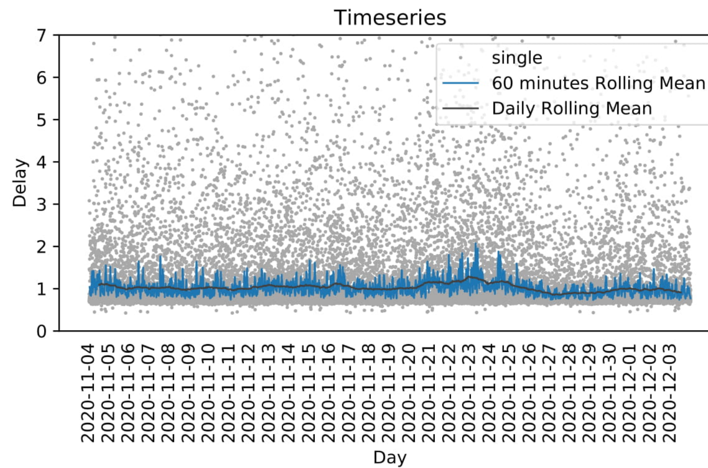


Figure 5.4: Eu-central-1a to eu-central-1b trace (private IP)

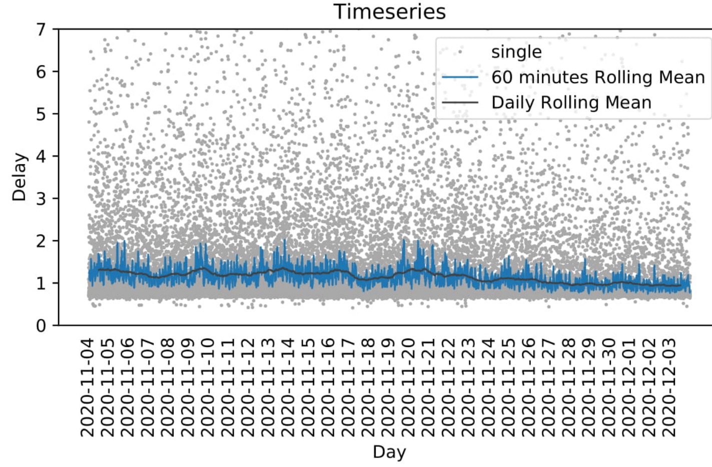


Figure 5.5: Eu-central-1b to eu-central-1c trace (private IP)

As we can see in Fig. 5.4, the average delay during the day is stable. The black line indicates the daily rolling mean, while the blue line shows the hourly rolling mean. The delay is regular in both Availability Zones. In the first plot, we can see that the average daily delay is increasing in the week from 20 November to 26 November. The reason could be that the Availability Zone was more used in this period. Unfortunately, we don't have access to the AWS data to see how crowded was the region during this period. What we will see in the latest experiment is how to link the delay with the AWS Spot instances prices.

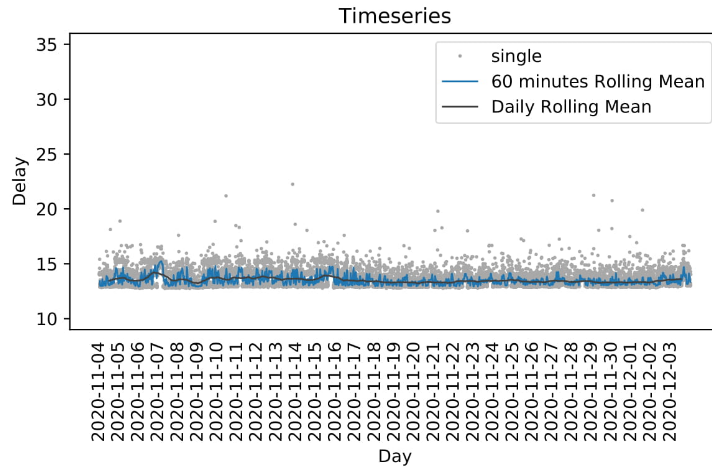


Figure 5.6: Eu-central-1a to eu-west-2a trace (public IP)

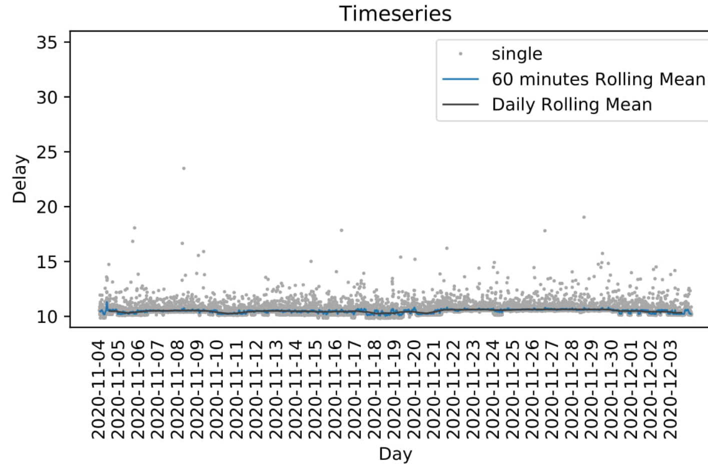


Figure 5.7: Eu-west-2a to eu-west-1a trace (public IP)

Multiregional Experiment		Confidence Interval Delay [ms]	
Source	Destination	Public IPs	Private IPs
eu-central-1a	eu-west-1a	24.614 ± 1.125	24.596 ± 1.281
eu-west-1a	eu-central-1a	24.264 ± 1.095	24.789 ± 1.333
eu-central-1a	eu-west-2a	13.495 ± 0.618	13.98 ± 1.12
eu-west-2a	eu-central-1a	13.385 ± 0.689	14.042 ± 1.05
eu-west-1a	eu-west-2a	10.366 ± 0.275	11.071 ± 1.554
eu-west-2a	eu-west-1a	10.484 ± 0.305	11.155 ± 1.302

Table 5.1: Multiregional delay with confidence intervals (99%) with public and private IPs

Multiregional Experiments. We are performing here the same experiment as before, but we are using Availability Zones in different regions and the public IP - a comparison between public and private IPs will be made in the next experiments. Multiregional experiments mean longer distances since the traffic has to go from Frankfurt to Dublin and London, i.e., higher delays are expected. Fig. 5.6 shows the delay between Frankfurt and London, and Fig. 5.7 the delay between London and Dublin. As expected, the delay in Fig. 5.6 is higher since the physical distance between the two regions is longer, so in this case, if the user wants to deploy a delay-sensitive application with multiregional high availability, it is recommended to deploy it in London and Dublin. Both measurements are stable over time.

Public vs. Private. One surprising behavior that we faced in the experiment was the difference in delay using the public and private IPs. As explained before, AWS makes it possible to create a VPC peering between two or more different regions. This is done to connect two instances deployed in separated

Regional Experiment		Confidence Interval Delay [ms]	
Source	Destination	Public IPs	Private IPs
eu-central-1a	eu-central-1b	0.835 ± 0.697	1.027 ± 1.04
eu-central-1b	eu-central-1a	0.864 ± 0.839	1.147 ± 1.202
eu-central-1a	eu-central-1c	1.066 ± 0.7	1.301 ± 1.496
eu-central-1c	eu-central-1a	1.094 ± 0.599	1.188 ± 1.369
eu-central-1b	eu-central-1c	0.915 ± 0.928	1.122 ± 1.505
eu-central-1c	eu-central-1b	0.943 ± 0.961	1.482 ± 2.149

Table 5.2: Frankfurt delay with confidence interval for public and private IPs

regions with the private IP assigned by AWS. Table 5.1 shows the confidence intervals (at 99%) for all the multiregional experiments performed in the entire month. As we can see, the delay using private IPs is higher, and also less stable, than for public IPs. This behavior is also repeated in all the regional experiments performed. For example, Table 5.2 shows the delay with confidence interval in the Frankfurt region. We think that these differences are due to the additional layer added inside the AWS network and the control that each packet has to pass in order to circulate inside the AWS network. For example, to prevent IP spoofing, the AWS network does not allow a virtual instance to send a packet inside a network if the packet's source address is different from the one assigned to the instance. These types of control can slow down the traffic and increase the delay.

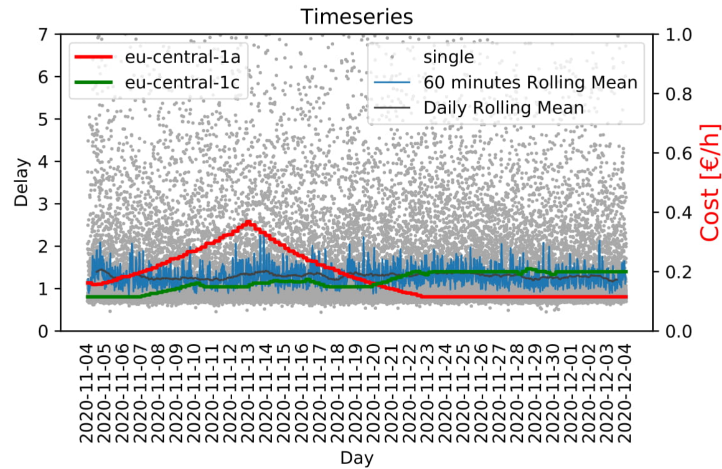


Figure 5.8: Eu-central-1a to eu-central-1c trace (Private IP) with spot prices

Delay vs. spot instances prices. We first need to explain the concept of *spot instance*. The AWS clusters are not always full. There are periods when the

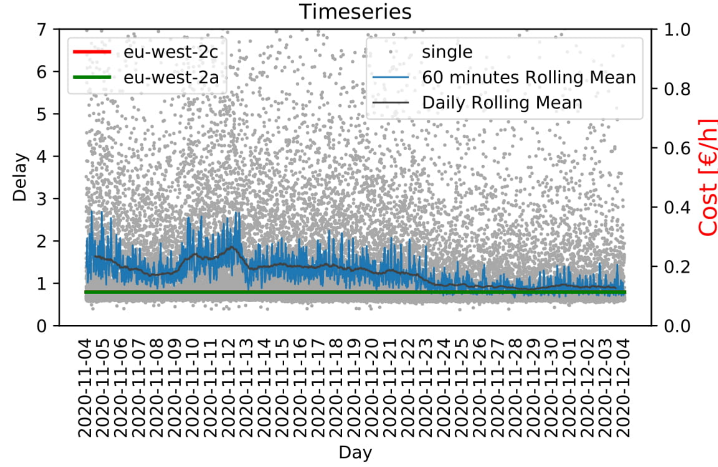


Figure 5.9: Eu-west-2c to eu-west-2a trace (Private IP) with spot prices

datacenters are barely used, so Amazon offers instances at discounted prices. It thus depends on how much the customer is willing to pay for general instances. If the spot instance's actual price is costs less than the amount the customer is willing to pay, the customer can use it. The downside is that the price is always changing, and if the spot instance price costs more than the amount set by the customer, the instances of the customer are terminated within five minutes. The spot instances are useful for applications that are not suffering immediate terminations. The spot prices allow the customer to save (a lot of) money since these prices are between 80% and 60% lower than on-demand prices. As the spot instances' prices increase when the datacenter is loaded, our intuition is to use the spot prices to check if the delay is changing with the prices.

Fig. 5.8 shows the delay between eu-central-1c and eu-central-1a Availability Zones. We can see that from 5 November to 13 November, the spot price increases and became almost twice expensive. This means that in this period, the Availability Zone was more crowded than usual and returned at the standard price around 23 November. In this period, we can see that the delay was not affected by the increasing demand from the customer. On the opposite, in Fig. 5.9, we can see that in the London region (between eu-west-2c and eu-west-2a), the average delay is higher from November 10 to November 13, but the spot prices are not changing. This means that there is no correlation between the spot prices and the delay measured in the Availability Zone.

5.6 Conclusions

In this chapter, we analyzed and presented the main components of the AWS IaaS service. AWS provides hundreds of services to its customers. We focused on the main EC2 components (like VPCs, subnets, Virtual Internet gateway, etc.) that companies and organizations use to move their infrastructures to the cloud.

We focused mainly on the networking performances of the global infrastructure. We studied the network delay stability in multiple deployment scenarios. Companies that are moving delay-sensitive applications on AWS are interested in response and delay time of the network for a high availability (multiregional) configuration or a regional configuration with multiple Availability Zones. We implemented a simple yet powerful tool for the community to measure the networking performances and plot automatically statistics, together with a map of the analyzed network.

CloudTrace is publicly accessible at `https://github.com/Giuseppe1992/CloudTrace`.

We performed multiple experiments on 3 different AWS regions using the basic EC2 instances, and we monitored the performance of the network for one month. Interested readers can easily download and install their applications, and test the networking performances in other regions. The deployment in the cloud and the experiment part are separated into different modules; CloudTrace can be easily extended to be compatible with Microsoft Azure and Google Cloud. These extensions can be helpful to differentiate the performances in the various cloud providers.

Chapter 6

Failure Recovery

6.1 Introduction

More than ever, data networks have demonstrated their central role in the world economy, but also in the well-being of humanity that needs fast and reliable networks. In parallel, with the emergence of Network Function Virtualization (NFV) and Software Defined Networking (SDN), efficient network algorithms considered too hard to be put in practice in the past now have a second chance to be considered again. In this context, as new networks will be deployed and current ones get significant upgrades, it is thus time to rethink the network dimensioning problem with protection against Shared Risk Link Group (SRLG) failures.

In this chapter, we consider a path-based protection scheme with a global rerouting strategy in which, for each failure situation, there may be a new routing of all the demands. Our optimization task is to minimize the needed amount of bandwidth. After discussing the hardness of the problem, we develop two scalable mathematical models that we handle using both Column Generation and Benders Decomposition techniques. Through extensive simulations on real-world IP network topologies and on randomly generated instances, we show the effectiveness of our methods: they lead to savings of 40% to 48% of the bandwidth to be installed in a network to protect against failures compared to traditional schemes. Finally, our implementation in OpenDaylight demonstrates the feasibility of the approach, and its evaluation with Mininet shows that technical implementation choices may have a dramatic impact on the time needed to reestablish the flows after a failure takes place.

As introduced before, NFV is an emerging approach in which network functions such as firewall, load balancing, and content filtering are no longer executed by proprietary hardware appliances but, instead, can run on generic-purpose servers located in small cloud nodes and can be instantiated on demand [Han+15]. Network flows are often required to be processed by an ordered sequence of network functions. Moreover, different customers may have

different requirements in terms of the sequence of network functions to be performed. This concept is referred to as Service Function Chaining (SFC) [QN15], also presented earlier.
Network failures such as cable cuts, natural disasters, faulty interfaces, or human errors are the daily routines of a network [Mar+14]. Such events clearly need to be taken into account when allocating resources to the network. Faults in the IP and optical layers tend to be correlated between them [KKV05]. Indeed, the failure of a component located in a common router, such as a linecard, or in the underlying optical infrastructure, such as a common fiber, may result in the consequential failure of multiple entities at the IP layer. To model this correlation, the concept of *Shared Risk Link Groups* (SRLGs) has been proposed [Pap01]. SRLGs allow to express easily a risk relationship, and can also represent different types of failures, such as single and multiple, node and link failures.

We consider in this work a protection technique called *unrestricted flow reconfiguration*, also known as *global rerouting* [PM04]. In each of the possible failure situations, a new set of backup paths is defined, one for each demand. This makes the protection method *bandwidth-optimal*. However, this also means that each failure may result in a completely different routing for the demands. In legacy networks, it is impractical to implement this technique due to the large number of rules to install on the network devices and hence signaling overhead. However, the introduction of SDN may change the game.

With SDN, the network control is decoupled from the packet forwarding data plane. Network intelligence is centralized in the controller that maintains a global view of the network [McK+08]. SDN offers to network operators better ways to manage and to configure their networks. Indeed, the introduction of logically centralized controllers reduces the control plane protocols complexity [KF13]. Routing decisions are taken in a single location, *the controller*, with a complete knowledge of the network state, instead of resulting from a distributed algorithm.

6.2 Related Work

The problem of providing network protection against failures has been widely investigated in the last decades [FV00]. With the intoduction of NFV and SFCs, an additional challenge is to map network functions to nodes and to guarantee that the execution order of the network functions is respected in both primary and backup paths. The problem of guaranteeing service continuity in Service Function Chain scenario has started to be investigated recently. Both *restoration* [Sou+17; LM15] and *protection* [Hma+17; Ye+16; BBS] techniques have been investigated. [Tom+18] proposes a scalable exact decomposition model to provide reliable service function chaining. Column generation techniques have been shown to be effective in dealing with both Service Function Chaining [HJG17; Hui+18] and failure protection [AV14].

With the advent of SDN/NFV, there are opportunities to create, deploy,

and manage networks more efficiently. Indeed, with SDN and its control–data planes decoupling, routing decisions can be done using a logically centralized approach. This paves the way for a broadening of perspective in terms of fault management [FM17].

Chu et al. [Chu+15] consider a hybrid SDN network and propose a method to design the network in such a way that fast failure recovery from any single link failure is achieved. Their proposal consists in redirecting the traffic of the failed link from the routers to SDN switches through pre-configured IP tunnels. Next hops are pre-configured before the failures take place, and the set of candidate recovery paths for different affected destinations is chosen by the SDN controller in such a way that the maximal link utilization after redirecting the recovery traffic through these paths is minimized. In [Qiu+17] they build a high-performance control plane for path computation using multiple controllers; while in [Qiu+19] the authors uses fast SDN rerouting reducing the time to compute new paths in case of link failures, minimizing delay of the new paths.

Suchara et al. [Suc+11] propose a joint architecture for both failure recovery and traffic engineering. Their architecture uses multiple pre-configured paths between each pair of edge routers. In the event of a failure, the failover is made on the least congested path that ensures connectivity. Besides, Sgambelluri et al. [Sga+13] propose a controller–based fault recovery solution that uses OpenFlow's Fast Failover Group Tables to quickly select a pre-configured backup path in case of link failure.

The idea of using a set of pre-configured multiple backup network configurations is not new. For instance, in [Kva+06; KCG07], the authors propose a pre-configured proactive IP recovery scheme that makes use of multiple routing backup configurations as a method for fast recovery. The main idea is to create a small set of backup routing configurations to be used in the case of a single link or node failure. Since the backup configurations are kept in the routers, it is necessary to reduce their number to avoid requiring the routers to store a significant amount of state information.

In [Vas+18; VNT20; Vas+20], the authors use SRLG for modelling in order to enhance the preparedness of a given network to natural disasters, or regional failures. For example, a regional SRLG aims to characterize a failure damaging the network only in a bounded geographical area. The authors in [Ass+20] propose a Mixed Integer Linear Program (MILP) for an elastic optical network protection with SRLG group and dedicated protection mechanism.

Herein, we take to the extreme the idea of multiple routing configurations by allowing a completely different routing in response to an SRLG failure situation. Unlike the above studies, our aim is to provide a bandwidth-optimal mechanism to design a reliable network. Our work revisits optimization techniques and protection schemes introduced to design survivable optical networks in a large corpus of studies of the 90s - see for example [GMS95; KM05; Sto06] for surveys or reference books. We generalize the results to layered graphs. Indeed, besides guaranteeing the recovery, our proposed approach also takes into consideration the Service Function Chain (SFC) requirement of the flows. Moreover, we show with the use of experiments that global rerouting, which was studied in the

past just as a lower bound, becomes today a viable and very efficient protection solution thanks to SDN control. It allows to study effectively what is the right number of NFVI-enabled nodes, in terms of costs and acceptable QoS levels.

Table 6.1 summarizes the most important works to be compared to our proposal. In particular, we check if the works consider optimal (bandwidth) dimensioning for protection against SRLGs, handle the VNFs of the network services, implement fast rerouting upon failures using SDN, propose scalable solutions through decomposition methods, or approximation algorithms.

Context and problem solved	[Chu+15]	[Suc+11]	[Ass+20]	[Taj+19]	[Sga+13]	[Kva+06]	[KCG07]	[VNT20]	[BSY18]	Our proposal
Survivable network design	✔	✔	✔	✔	✔	✔	✔	✔	✘	✔
Fast rerouting with SDN	✔	✘	✘	✔	✔	✘	✘	✘	✔	✔
Network services and NFV	✘	✘	✘	✔	✘	✘	✘	✘	✘	✔
SRLG	✘	✔	✔	✘	✘	✘	✘	✔	✘	✔
Theoretical methods and results										
Optimal bandwidth	✔	✔	✔	✘	✘	✘	✘	✘	✔	✔
Decomposition methods	✘	✘	✘	✘	✘	✘	✘	✘	✔	✔
Approximation algorithms	✔	✔	✘	✔	✘	✔	✔	✔	✘	✔
Implementation results										
Implementation via emulation	✘	✘	✘	✘	✔	✘	✘	✘	✘	✔

✔=Yes, ✘=No

Table 6.1: Summary of related work. The table analyzes the context and the methods used in the different works.

6.3 Optimization Approaches

In this section, we first rigorously define the bandwidth-optimal failure recovery problem and prove the hardness and inapproximability results for the Global Rerouting problem (Sec. 6.3.1). We provide a compact ILP formulation (Sec. 6.3.3) which will be used as a baseline for evaluation. We then propose a scalable[1] decomposition model that relies on the Column Generation technique (Sec. 6.3.4). Both formulations are based on a layered network model described in Sec. 6.3.2. We next introduce a second scalable model using Benders Decomposition approach (Sec. 6.3.5). The models led us to study the complexity of a subproblem, the Min-Overflow Problem, and to propose approximation algorithms to solve it (Sec. 6.3.6).

6.3.1 Problem Statement And Notations

We model the network as an undirected graph $G = (V, E)$, where V represents the set of nodes and E the set of links. We are given a set of SRLG events $\mathcal{R}$ that can incur link failures.

Each $r \in \mathcal{R}$ consists of a set of links that share a common physical resource. We denote by $\mathcal{D}$ the set of demands (e.g., traffic between two locations). As we

[1] The term scalable has to be understood here in the context of exact optimization methods. The proposed optimization methods are said *scalable* in the sense that they allow to solve much larger instances than the classic compact ILP formulation solving the same problem.

Symbol	**Description**
G	Undirected graph
V	Set of nodes in the graph G
E	Set of links in the graph G
$r \in \mathcal{R}$	Set of links that share a common physical resource
$\mathcal{R}$	Set of SRLGs
$d \in \mathcal{D}$	Single demand composed by (s_d, t_d, bw_d, C_d)
$\mathcal{D}$	Set of demands
s_d	Source of the traffic
t_d	Destination of the traffic
bw_d	Required units of bandwidth G
C_d	Ordered sequence of network functions
$\ell(d)$	Length of the SFC for a demand d
$V^{\text{VNF}} \subseteq V$	Set of NFVI nodes
$G^L(d)$	Layered graph for demand d
π	Service path
Π_d^r	Service function paths for a demand d
a_{uv}^{π}	Number of times link (u, v) is used in the service path π
$y_{\pi}^{d,r}$	Boolean value to check if demand d uses path π as a service path in the SRLG failure event r
x_{uv}	Bandwidth allocated to link (u, v)
$\varphi_{(ui,vj)}$	Boolean value to check if the flow is forwarded on link $((u, i), (v, j))$ of $G^L(d)$
α_{ω}^{sd}, β_{uv}^{r}	Dual values relative to constraints (6.8) and (6.9)
$\lambda_d(\mu)$	Length of the shortest path for demand d, with respect to link metrics μ

Table 6.2: Notation table.

are solving a dimensioning problem, we assume prior full knowledge of traffic demands, i.e., traffic matrices are known beforehand. A demand $d \in \mathcal{D}$ is modeled by a quadruple (s_d, t_d, bw_d, C_d) with s_d the source, t_d the destination, C_d the ordered sequence of network functions that need to be performed to all the packets belonging to the flow of the demand, and bw_d the required units of bandwidth. We denote by $\ell(d)$ the length of the SFC for a demand d.

Network functions need to be executed on the so-called NFV Infrastructure (NFVI) nodes. Not all the network nodes are enabled to run virtualized func-

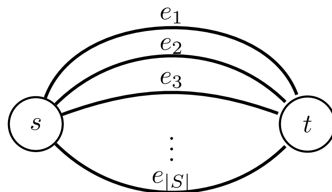


Figure 6.1: The multigraph resulting from the reduction. $G = (V, E)$ is the multigraph with $V = \{s, t\}$ and $E = \{e_i,\ i = 1, ..., |S|\}$. All the edges have s and t as endpoints.

tions. We denote by $V^{\text{VNF}} \subseteq V$ the set of NFVI nodes. Moreover, we assume that an NFVI node can only run a subset of the network functions, as there may be constraints on their location in the network (e.g., geography or regulatory constraints and anti-affinity rules).
Given the network topology and the traffic rate of the demands to be supported, the purpose of the design problem is to precompute a set of paths to guarantee the recovery of all the demands in the event of an SRLG failure, while satisfying their SFC requirements.
The considered optimization task is to *minimize the required bandwidth in the network*. We refer to this problem as the Global Rerouting problem. For each demand $d \in \mathcal{D}$, we have to find a primary path and a protection one for each SRLG failure situation $r \in \mathcal{R}$, such that the total amount of bandwidth needed to guarantee the recovery in all the failure situations is minimized.

We denote by $u(i)$ the copy of node u in layer i. The paths for demand D^c_{sd} starts from node $v_s(0)$ in layer 0 and ends at node $v_d(1)$ in layer 1. Using link $(u(i), v(i))$ on G^L implies using link (u, v) on G. On the other hand, using link $(u(0), u(1))$ implies using node u to perform the required network function(s).

We begin the section by proving hardness and inapproximability results for the Global Rerouting problem. Then, we propose a scalable decomposition model which relies on the Column Generation technique which is based on a layered network model.

Proposition 1. *The* Global Rerouting *problem is NP-hard even for a single demand, and cannot be approximated within*
$(1 - \epsilon)\ln(|R|)$ *for any* $\epsilon > 0$ *unless P=NP, where* $|R|$ *denotes the number of failing scenarios.*

We use a reduction from the Hitting Set Problem, which is defined as follows. We are given a collection C of subsets of a finite set S and the problem consists in finding a hitting set for C, i.e., a subset $S' \subseteq S$ such that S' contains at least one element from each subset in C of minimum cardinality. Given an instance $\mathcal{I} = (S, C)$ of Hitting Set, we can build an instance $\mathcal{I}' = (G, \mathcal{D}, \mathcal{R})$ of Global Rerouting in the following way. $G = (V, E)$ is a multigraph with $V = \{s, t\}$ and $E = \{e_i,\ i = 1, ..., |S|\}$. All the edges have s and t as endpoints

(see Fig 6.1 and Example 1 for an example). For each $C' \subseteq C$, we add a failing scenario $r_{C'} = E \setminus C'$ to R, corresponding to edges that cannot be used in the failure situation r. Finally, we add to $\mathcal{D}$, a demand d with s and t as source and destination respectively, and with charge equal to 1. The goal now consists in finding a path for each of the failure scenarios $r \in \mathcal{R}$ minimizing the needed capacity to deploy. The total capacity needed to satisfy d in each of the failure situations does not exceed $c \iff$ there exists a hitting set of cardinality not greater than c. The proposition follows immediately from the fact that Hitting Set is NP-hard [ADP80] and cannot be approximated within a factor of $\ln |S|$ [DS14], unless P=NP.

Example 1 (Reduction)**.** Consider the following instance of the Hitting Set Problem: $S = \{1, 2, 3, 4, 5\}$ and $C = \{C_1, C_2, C_3, C_4, C_5\}$ with $C_1 = \{1, 2, 3, 4\}$, $C_2 = \{1, 4, 5\}$, $C_3 = \{2, 5\}$, $C_4 = \{2, 3, 5\}$, and $C_5 = \{1, 4\}$. The multigraph resulting from the reduction of the instance is a multigraph with 5 edges (noted $1, 2, 3, 4$, and 5) linking two nodes s and t corresponding to the elements in S. The corresponding instance of the Global Rerouting problem has a single demand d with source s and destination t, an empty SFC, and requiring a single unit of bandwidth to be sent. It has 5 SRLG events $\mathcal{R} = \{r_1, r_2, r_3, r_4, r_5\}$, as many as the number of subsets in C, with $r_1 = E \setminus C_1 = \{5\}$, $r_2 = E \setminus C_2 = \{2, 3\}$, $r_3 = E \setminus C_3 = \{1, 3, 4\}$, $r_4 = E \setminus C_4 = \{1, 4\}$, and $r_5 = E \setminus C_5 = \{2, 3, 5\}$. Solving the Global Rerouting problem boils down to finding a set of edges (of minimum cardinality) for which at least an edge will be available over all failure scenarios: $\{4, 5\}$. If we install a capacity of 1 for these two edges, we obtain a survivable network of minimum cost to protect against the 5 considered SRLG events. Note that this is equivalent to finding a minimum cardinality Hitting Set for the original instance, as the sets C_1, C_2, C_3, C_4, C_5 correspond to the edges available in each failure scenario.

6.3.2 A Layered Network Model

In order to support any service types and service function chains, the traffic associated with each demand must be processed by an ordered sequence of network functions defined in advance. Similarly to [HJG18], we use a layered graph to model this constraint.

Let $G = (V, E)$ be a graph. We associate to each demand $d \in \mathcal{D}$ a layered graph $G^L(d) = (V', E')$. $G^L(d)$ is defined as follows. For each $u \in V$, V' contains the vertices $(u, 0), (u, 1), ..., (u, \ell(d))$. An edge $((u, i), (v, j))$ belongs to E' if and only if (1) $(u, v) \in E$ and $i = j$, or (2) u is a NFVI node, $u = v$, $j = i + 1$, and the jth function of C_d is installed on u.

Given a demand d, let s_d and t_d be the source and the destination node, respectively. A path starting at vertex $(s_d, 0)$ and finishing at vertex $(t_d, \ell(d))$ of $G^L(d)$ defines (a) which edges of G are used to route the flow associated to the demand; and (b) on which NFVI nodes the traffic is processed by each of the requested network functions. We refer to a path in $G^L(d) = (V', E')$ as a Service Function Path (SFP) - see Fig. 6.2 for an example.

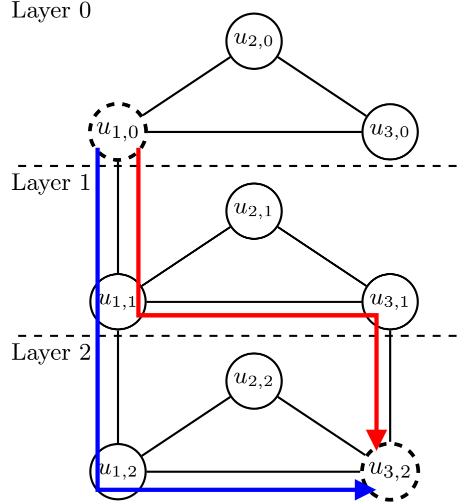


Figure 6.2: The layered network $G^L(d)$ associated with a demand d such that $s_d = u_1$, $t_d = u_3$, and $C_d = f_1, f_2$, with $G = (V, E)$ being a triangle network. We assume f_1 installed on Node u_1 and f_2 installed on Nodes u_1 and u_3. Source and destination nodes of $G^L(d)$ are $u_{1,0}$ and $u_{3,2}$, respectively. They are drawn with dashed lines. Two possible service paths that satisfy d are drawn in red and blue.

6.3.3 Compact ILP Formulation

A straightforward way to model our problem consists in using an ILP. The goal of the ILP is to find for each demand $d \in \mathcal{D}$ a service path on the layered graph $G^L(d)$ for each SRLG event such that the total bandwidth required in the network is minimized. In order to take into account the scenario without failure, we add an SRLG associated with an empty set of links to $\mathcal{R}$. Thus, the SRLGs set is extended to $\mathcal{R} \cup \emptyset$.
Variables:

- $\varphi^{d,r}_{(ui,vj)} \in \{0, 1\}$, with $\varphi^{d,r}_{(ui,vj)} = 1$ if demand d uses link $((u, i), (v, j))$ of $G^L(d)$ in the SRLG failure event r.

- $x^r_{uv} \geq 0$ is the amount of bandwidth allocated to link (u, v) of G in the SRLG failure event r.

Objective (6.1): minimization of the bandwidth needed in the network in order to guarantee the recovery.

$$\min \sum_{(u,v) \in E} \max_{r \in \mathcal{R}} x^r_{uv} \tag{6.1}$$

For each link, the needed capacity to be installed is $\max_{r \in \mathcal{R}} x^r_{uv}$ in order to guarantee the recovery for every failure scenario or SRLG in $\mathcal{R}$. We thus mininimize the sum over all links in (6.1).
Flow Conservation constraints (6.2) (6.3) (6.4): for each demand d, SRLG set $r \in \mathcal{R}$,

$$\sum_{(v,j) \in \omega((s_d,0))} \varphi^{d,r}_{(s_d 0, vj)} = \sum_{(v,j) \in \omega((t_d,\ell(d))} \varphi^{d,r}_{(t_d \ell(d), vj)} = 1 \tag{6.2}$$

$$\sum_{(v,j) \in \omega((u,i))} \varphi^{d,r}_{(ui,vj)} \leq 2 \quad v \in V \setminus \{(s_d,0),(t_d,\ell(d))\} \tag{6.3}$$

$$\sum_{(v',j') \in \omega((u,i)) \setminus \{(v,j)\}} \varphi^{d,r}_{(ui,v'j')} \geq \varphi^{d,r}_{(ui,vj)}$$
$$v \in V \setminus \{(s_d,0),(t_d,\ell(d))\}, \ell \in \omega((u,i)) \tag{6.4}$$

Equation (6.2) ensures that the path of demand d starts at the source node $(s_d, 0)$ in layer 0 and ends at the destination node $(t_d, \ell(d))$ in layer $\ell(d)$, with $\ell(d)$ the length of the SFC of the demand d. Equation (6.3) forces the paths to not use twice the same link of the layered graph to avoid loops. The flow conservation of each path (i.e., a path arriving at an intermediate node has to leave this node) is a result of Equation (6.4). Remark that the constraint is an inequality. However, we have equality for an optimal solution as a product of the minimization objective.

Unavailable links in an SRLG failure event (6.5): for each $r \in \mathcal{R}$,

$$\sum_{d \in \mathcal{D}} \sum_{(u,v) \in r} \sum_{k=0}^{\ell(d)} \varphi^{d,r}_{(uk,vk)} = 0. \tag{6.5}$$

Equation (6.5) ensures that a path cannot use a link involved in SRLG r when considering the failure scenario r.

Bandwidth utilization in an SRLG failure event (6.6): for each SRLG set $r \in \mathcal{R}$, link $(u,v) \in E$,

$$\sum_{d \in \mathcal{D}} bw_d \cdot \sum_{k=0}^{\ell(d)} \varphi^{d,r}_{(uk,vk)} \leq x^r_{uv}. \tag{6.6}$$

Equation 6.6 guarantees that the bandwidth usage on each link (which is the sum of the bandwidth usage over all layers on the link) is lower than or equal to the link capacity.

6.3.4 Column Generation Approach

One can apply the Dantzig-Wolfe decomposition [DW60; CC+83] to the ILP formulation to exploit its block structure per demand $d \in \mathcal{D}$. The resulting model takes the form of a path flow formulation. In order to model the ordered

sequence of network functions by which the traffic associated to a demand must be processed, we use a layered graph, similarly as in [HJG18]. Let $G = (V, E)$ be a graph. We associate to each demand $d \in \mathcal{D}$ a layered graph $G^L(d) = (V', E')$. $G^L(d)$ is defined as follows. For each $u \in V$, V' contains the vertices $(u, 0), (u, 1), ..., (u, \ell(d))$. An edge $((u, i), (v, j))$ belongs to E' if and only if (1) $(u, v) \in E$ and $i = j$, or (2) u is an NFVI node, $u = v$, $j = i + 1$, and the jth function of C_d is installed on u. We refer to a path in $G^L(d) = (V', E')$ as a Service Function Path (SFP).
We denote by Π^r_d the set of service function paths for a demand d in the SRLG failure situation r. Each service path π is associated to an integer value $a^\pi_{uv} \geq 0$ telling the number of times link (u, v) is used in the service path π.
Variables:

- $y^{d,r}_\pi \geq 0$, where $y^{d,r}_\pi = 1$ if demand d uses path π as a service path in the SRLG failure event $r \in \mathcal{R}$.
- $x_{uv} \geq 0$, bandwidth allocated to link $(u, v) \in E$.

Objective (6.7): minimization of the required bandwidth

$$\min \sum_{(u,v) \in E} x_{uv} \tag{6.7}$$

One service path for each demand and SRLG failure event (6.8): for all $d \in \mathcal{D}$, $r \in \mathcal{R}$

$$\sum_{\pi \in \Pi^r_d} y^{d,r}_\pi \geq 1. \tag{6.8}$$

Inequality (6.8) ensures that at least one path is selected for each pair demand/SRLG event. Note that a single path is selected for optimal solutions due to the minimization objective.

Bandwidth utilization (6.9): for all $(u, v) \in E$, $r \in \mathcal{R}$

$$x_{uv} \geq \sum_{d \in \mathcal{D}} \sum_{\pi \in \Pi^r_d} bw_d \cdot a^\pi_{uv} \cdot y^{d,r}_\pi. \tag{6.9}$$

Given its very large number of variables, Column Generation is an efficient technique to handle the above linear integer programming model. One starts with a limited set of variables in a so-called Restricted Master Problem (RMP). At each iteration, the RMP is solved. The dual values associated to the constraints are used to generate new paths with negative reduced cost and the associated variables are added to the RMP that may enable to improve the current solution. This process is repeated until no more columns can be added to the RMP, i.e., no more columns with negative reduced cost exist. We refer to [DDS06] for more details regarding this technique.

Pricing Problem

The pricing subproblem is solved independently for each demand d and SRLG failure event r and it returns a service path π. It consists in finding

a minimum cost service path in the layered graph where the weight of a link is defined according to the dual values of the associated constraint.
Variables:

- $\varphi_{(ui,vj)} \in \{0,1\}$, where $\varphi^{d,r}_{(ui,vj)} = 1$ if the flow is forwarded on link $((u,i),(v,j))$ of $G^L(d)$.

Let $\alpha^{sd}_{\omega} \geq 0$ and $\beta^r_{uv} \geq 0$ be the dual values relative to constraints (6.8) and (6.9), respectively. The service path reduced cost for a given demand d and an SRLG r can be written as:

$$\min -\alpha^d_r + bw_d \cdot \sum_{(u,v)\in E} \beta^r_{uv} \cdot \sum_{k=0}^{\ell(d)} \varphi_{(uk,vk)}$$

The first term is a constant for each request, and the second term corresponds to a summation over the links of the network. Therefore, the objective function (6.10) becomes:

$$\min \sum_{(u,v)\in E} \beta^r_{uv} \cdot \sum_{k=0}^{\ell(d)} \varphi_{(uk,vk)}. \tag{6.10}$$

Thus, for each request and for each failure situation, the pricing subproblem corresponds to a weighted shortest-path problem in the layered graph. In a given SRLG failure situation r and for all the demands $d \in D$, the weight of a link $((u,i),(v,j))$ of $G^L(d)$ is defined to be β^r_{uv} if $i = j$, 0 otherwise. Either one of these paths leads to a negative reduced cost column, or the current master solution is optimal for the unrestricted program. In the former case, the new configurations found are then added iteratively to the RMP. In the second case, the solution of the linear relaxation of the RMP z^*_{LP} is optimal. Convergence of the basic Column Generation procedure suffers from dual oscillations as the number of constraints (6.9) is large. To improve the convergence and reduce the fluctuations in the dual variables, we use the piecewise linear penalty function stabilization described in [Pes+18]. Associated to the optimal solution of the linear relaxation of the RMP, for each demand d and SRLG failure situation r, there is a set of service paths identified by all the variables $y^{d,r}_{\pi}$ with value greater than 0. These service paths guarantee the minimum cost in terms of required bandwidth to deploy for ensuring the recovery in the splittable flow case. However, if we restrict our attention to the unsplittable flow case, we have to select only one service path for each demand and SRLG failure situation. The problem now consists in making this choice by reducing the *overflow* introduced in the network. One possible way consists in changing the domain of the variables in the last RMP from continuous to integer and use an ILP solver. We refer to this strategy as MasterILP.

6.3.5 Benders Decomposition Approach

Applying Benders Decomposition technique [Ben62] to our compact model consists in splitting the original problem variables into first stage link capacity

assignments on one hand, and second stage routing decisions on the other hand. The master problem is in terms of the x_{uv} variables. It takes the following form.

Objective (6.11): minimization of the bandwidth used in the network

$$\min \sum_{(u,v)\in E} x_{uv}. \tag{6.11}$$

Metric inequalities (6.12):

$$\sum_{(u,v)\in E} \mu_{uv} \cdot \delta_{uv,r} \cdot x_{uv} \geq \sum_{d\in\mathcal{D}} \lambda_d(\mu) \cdot bw_d \quad \forall\ \mu \in \mathbb{R}+^E \tag{6.12}$$

where the latter constraints are known as metric inequalities [CCG09]. They can be separated in polynomial time by solving an LP. Hence, they can be handled in a lazy way by a dynamic generation, which allows to solve the problem using the cutting plane algorithm. These cuts are iteratively added to the master problem until the difference between the lower bound, corresponding to the solution of the master problem, and the upper bound, corresponding to the solution of the subproblems, falls under a fixed value ϵ.

Benders separation subproblem is solved given the link bandwidth vector x. This capacity assignment is globally feasible (for the splittable problem) if and only if for each vector $\mu = \{\mu_{uv} \geq 0 : (u,v) \in E\}$ and for each SRLG failure situation $r \in \mathcal{R}$, the inequality

$$\sum_{(u,v)\in E} \mu_{uv} \cdot \delta_{uv,r} \cdot x_{uv} \geq \sum_{d\in\mathcal{D}} \lambda_d(\mu) \cdot bw_d \tag{6.13}$$

holds, where $\delta_{uv,r} \in [0,1]$ is the available portion of link (u,v) under scenario r, and $\lambda_d(\mu)$ is the length of the shortest path for demand d with respect to link metrics μ.

Associated to the optimal solution of the master problem, we have the optimal link capacities in the splittable flow case, as in the Column Generation case. The main difference lies in the fact that we do not have the selected paths. We thus have to find a path for each demand and failure situations trying to minimize the *overflow*, with respect to the solution found in the splittable flow case.

6.3.6 The Min-Overflow Problem

The efficiency of decomposition methods such as Column Generation and Benders technique is based on two main principles: (i) decomposing the problem into subproblems which can be solved efficiently (ii) first solving fractional versions of the problems. Indeed, solving a fractional Linear Program can be done in polynomial time, when their mixed integral version often is NP-complete (as it is the case for the problems considered here). An important step of the methods is to obtain good integral solutions from the optimum fractional solutions found. The Min Overflow Problem appears in this context.

Algorithm 4 Iterative ILP (IterILP)

1: Solve the linear relaxation of the general problem
2: $\tilde{c} \leftarrow c^*$
3: **for each** $r \in \mathcal{R}$ **do**
4: (a) route the demands on $G' = (V, E \setminus r, \tilde{c})$ solving MinOverflowILP, an integral multicommodity flow problem minimizing the overflow
5: (c) update $\tilde{c}$ with the introduced overflow (if any)
6: **end for**
7: **return** $\tilde{c}$

As it is costly to solve (exactly) the integer version of the master program (the MasterILP strategy discussed above at the end of Sec. 6.3.4), to obtain a "good" integer solution, we could use another approach. That is, we may start by computing efficiently a fractional solution to the linear relaxation of the problem (i.e., when flows are splittable) using either the Column Generation algorithm or the Benders Decomposition technique. We then try to obtain a *good* integer solution to the problem (i.e., when flows are unsplittable) by minimizing the cost to pay in terms of additional capacity (i.e., the *overflow*) over all the scenarios when using a single path for each demand.

IterILP Algorithm

We define overflow as the total amount of additional bandwidth to be allocated to the network in order to satisfy all the demands. To this end, one possible strategy considers each scenario one at a time, and formulates a multicommodity flow problem as an ILP. The objective function consists in minimizing the overflow to be allocated to the network. We refer to this strategy as IterILP - see Algorithm 4 for its pseudo-code. The used ILP, denoted as MinOverflowILP, is given below.

MinOverflowILP:

Input a graph $G = (V, E)$, an SRLG set $r \in \mathcal{R}$, and a capacity function $\tilde{c}$.

Objective (6.14): minimization of the overflow (additional bandwidth) needed in the network in order to guarantee the recovery for the scenario corresponding to SLRG r.

$$\min \sum_{(u,v) \in E} o^r_{uv} \tag{6.14}$$

Remark: if the objective function is equal to $|E|$, all demands can be routed with the capacity function $\tilde{c}$. All the overflow ratios are equal to 0 and no additional capacity is needed for the scenario of the SLRG r.

Flow conservation constraints (6.15) (6.16) (6.17): for each demand d,

$$\sum_{(v,j)\in\omega((s_d,0))} \varphi^{d,r}_{(s_d0,vj)} = \sum_{(v,j)\in\omega((t_d,\ell(d))} \varphi^{d,r}_{(t_d\ell(d),vj)} = 1 \quad (6.15)$$

$$\sum_{(v,j)\in\omega((u,i))} \varphi^{d,r}_{(ui,vj)} \leq 2 \quad v \in V \setminus \{(s_d,0),(t_d,\ell(d))\} \quad (6.16)$$

$$\sum_{(v',j')\in\omega((u,i))\setminus\{(v,j)\}} \varphi^{d,r}_{(ui,v'j')} \geq \varphi^{d,r}_{(ui,vj)}$$
$$v \in V \setminus \{(s_d,0),(t_d,\ell(d))\}, \ell \in \omega((u,i)) \quad (6.17)$$

Bandwidth utilization for the SRLG set r (6.18): for each link $(u,v) \in E$,

$$\sum_{d\in\mathcal{D}} bw_d \cdot \sum_{k=0}^{\ell(d)} \varphi^{d,r}_{(uk,vk)} \leq x^r_{uv}. \quad (6.18)$$

Equation 6.18 guarantees that the bandwidth usage on each link is lower than, or equal to, the link capacity.

Definition of the overflow: the overflow ratio of the link $uv \in E$, o^r_{uv}, is defined as 1 if $x^r_{uv} \leq \tilde{c}_{uv}$ and $(x^r_{uv} - \tilde{c}_{uv})/\tilde{c}_{uv}$ otherwise, i.e., $o^r_{uv} = \max((x^r_{uv} - \tilde{c}_{uv})/\tilde{c}_{uv}, 1)$. It thus gives inequalities (6.19) and (6.20), for each link $(u,v) \in E$,

$$o^r_{uv} \geq \frac{x^r_{uv} - \tilde{c}_{uv}}{\tilde{c}_{uv}} \quad (6.19)$$

$$o^r_{uv} \geq 1 \quad (6.20)$$

Note that the minimization of the objective function will imply the equality to the maximum for an optimal solution.

If on one hand, this strategy leads to good results, on the other hand, it may not scale well, since we have to solve an ILP for each SRLG failure scenario.

Algorithmic Complexity and Randomized Rounding Strategy

Another strategy consists in using an algorithm to route the demands while minimizing the overflow. The problem to be solved for an SRLG failure scenario which we refer to as MIN OVERFLOW PROBLEM can be stated as follows.
Input: A graph $G = (V,E)$, a collection $\mathcal{D}$ of demands, each associated with a source, a destination and the units of flows to be routed. Also, each demand is associated with a set of paths, corresponding to the fractional solution of the splittable flow version of the problem. Lastly, a capacity function $c^* : (u,v) \rightarrow c^*_{uv}$, according to the optimal capacities found solving the linear relaxation of the general problem.
Output: A path for each demand.
Objective: Minimize the overflow, i.e., minimize $\sum_{(u,v)\in E} \frac{\tilde{c}(u,v)}{c^*_{uv}}$ with $\tilde{c}(u,v)$ defined as the maximum between c^*_{uv} and the capacity of the link (u,v) after having selected one path per demand.

Note that, contrary to the classical version of the problem, we do not have hard capacity constraints to respect while computing an integer routing. Herein, the goal is to route all the demands reducing the increase in terms of capacity over each of the links (i.e., the overflow) with respect to the *free given capacities* already available in the network.

Proposition 2. *The* MIN OVERFLOW PROBLEM *is APX-hard (and so is NP-hard) and cannot be approximated within a factor of* $1 + \frac{3}{320}$*, unless P=NP.*

Proof. We use a reduction from MAX 3-SAT. Let $\mathcal{I}$ be an instance of MAX 3-SAT with n variables $V_i, 1 \leq i \leq n$ and m clauses $C_j, 1 \leq j \leq m$. We associate each boolean variable V_i to a demand d_i asking for one unit of flow from a source s_{d_i} to a destination t_{d_i} connected by two paths $P_0(V_i)$ and $P_1(V_i)$. Selecting $P_1(V_i)$ (respectively $P_0(V_i)$) corresponds to assign to V_i the true (respectively false) value.
We associate each clause C to an edge (u_C, v_C) and we build the paths in the following way. For each variable V_i, we consider all the set $C(V_i)$ with all the clauses in which V_i appears as positive literal. $C(V_i) = C_{i_1}, C_{i_2}, ..., C_{i_m}$ with $i_1 \leq i_2 \leq ... \leq i_m$. Then, $P_1(V_i) = s_{d_i}, (u_{i_1}, v_{i_1}), (u_{i_2}, v_{i_2}), ..., (u_{i_m}, v_{i_m}), t_{d_i}$.
In a similar way, we consider now all the clauses in which V_i appears as negative literal. $C(\overline{V_i}) = C_{\bar{i}_1}, C_{\bar{i}_2}, ..., C_{\bar{i}_m}$ with $\bar{i}_1 \leq \bar{i}_2 \leq ... \leq \bar{i}_m$. $P_1(V_i)$ is defined as $s_{d_i}, (u_{\bar{i}_1}, v_{\bar{i}_1}), (u_{\bar{i}_2}, v_{\bar{i}_2}), ..., (u_{\bar{i}_m}, v_{\bar{i}_m}), t_{d_i}$.
As we build paths in this way, the load of an edge (u_C, v_C) is equal to the number of literals in the clause C assigned to the false value. There are $\sum_{i=1}^{n}(2i_m + 1)(2\bar{i}_m + 1) = 6m + 2n$ edges in the construction, as $\sum_{i=1}^{n} |C(V_i)| + |C(\overline{V_i})| = 3m$, the number of literals in the formula. We now assign to each edge a capacity 2. A fractional routing always exists. Indeed, routing one half of the charge of each demand d_i on $P_0(V_i)$ and the other half on $P_1(V_i)$ is feasible, since after identification, an arc receives at most $3 \times \frac{1}{2} \leq 2$. The case of an integral flow is quite different, since, in such a case, only one between $P_0(x)$ or $P_1(x)$ can be chosen. Since the capacity of the edges is 2, the cost will be 2 on each identified edge $\iff$ the formula is satisfiable. This proves that the problem is NP-complete (as 3-SAT is NP-complete). Then, we derive an inapproximability result using the fact that it is NP-hard to satisfy more than $\frac{7}{8}$ of the clauses (even if the formula is satisfiable) [Hås01]. So, we may have to pay 3 on $m/8$ edges (even though the optimal is 2 on all edges). Since the initial cost is less than 2 times the number of edges, it is less than $2 \times (6m + 2n) = 12m + 4n$. We have $n \leq \frac{m}{3}$. So, it is NP-hard to decide if the cost is 1 or $\frac{(12+\frac{4}{3})m + \frac{m}{8}}{(12+\frac{4}{3})m} = 1 + \frac{3}{320}$. □

Proposition 3. *The* MIN OVERFLOW PROBLEM *can be approximated with high probability within a factor of* $(1 + \frac{1}{e}) + \epsilon$*, for any* $\epsilon > 0$*.*

Let c^*_{uv} be the optimal capacity of an edge (u, v) in the splittable flow case. After having computed a fractional flow, we have associated to each demand $d \in \mathcal{D}$ a set consisting of $n(d) \geq 1$ paths $\mathcal{P}_d = \{P_{d,i} : i = 1, ..., n(d)\}$. Each path $P_{d,i}$ is associated to a multiplier $0 \leq \lambda_{d,i} \leq 1$ such that $\sum_{i=1}^{n(d)} \lambda_{d,i} = 1$ which gives the amount of flow $\lambda_{d,i} \cdot bw_d$ routed on $P_{d,i}$. Let $\lambda_{d,i}(uv)$ be the fraction of

flow routed on the edge (u, v) by a demand d. Note that for each edge (u, v), we have $\sum_{d \in \mathcal{D}} \sum_{i=1}^{n(d)} bw_d \cdot \lambda_{d,i}(uv) \leq c^*_{uv}$ since by hypothesis these capacities are feasible for the splittable flow case. In order to find an unsplittable solution, we use a rounding–based heuristic referred to as RANDOMIZED ROUNDING, which assigns to a demand d a path $P_{d,i}$ with probability $\lambda_{d,i}$. We consider now the impact in terms of load on an edge (u, v). Let f_{uv} be the flow on (u, v) at the end of the rounding procedure. Clearly, for each edge (u, v), $\mathbb{E}(f_{uv}) \leq c^*_{uv}$ holds. Let O_{uv} be the overflow on the edge (u, v) defined as $\max(0, f_{uv} - c^*_{uv})$. We denote by $\mathsf{P}_0(uv) = \mathsf{P}[f_{uv} = 0]$ the probability that the edge (u, v) is not used.

$$\begin{aligned}\mathbb{E}[O_{uv}] &= \mathsf{P}_0(uv) \cdot 0 + (1 - \mathsf{P}_0(uv))\,\mathbb{E}[f_{uv}|f_{uv} > 0] - c^*_{uv} \\ &= (1 - \mathsf{P}_0(uv))\,\mathbb{E}[f_{uv}|f_{uv} > 0] - c^*_{uv}(1 - \mathsf{P}_0(uv))\end{aligned}$$

Moreover,

$$\mathbb{E}[f_{uv}] = \mathsf{P}_0(uv) \cdot 0 + (1 - \mathsf{P}_0(uv))\,\mathbb{E}(f_{uv}|f_{uv} > 0)$$

$$\mathbb{E}[f_{uv}|f_{uv} > 0] = \frac{\mathbb{E}[f_{uv}]}{1 - \mathsf{P}_0(uv)}$$

We can therefore bound the expected overflow of a link (u, v).

$$\begin{aligned}\mathbb{E}[O_{uv}] &= \mathbb{E}[f_{uv}] - c^*_{uv}(1 - \mathsf{P}_0(uv)) \\ &= \mathsf{P}_0(uv)c^*_{uv} - (c^*_{uv} - \mathbb{E}[f_{uv}]) \leq \mathsf{P}_0(uv)c^*_{uv}\end{aligned}$$

Let us now consider the probability $\mathsf{P}_0(uv)$ that an edge is not used after the randomized rounding. Given an edge (u, v), we define $\mathcal{P}_{uv}$ to be the paths that contain (u, v) as an edge.

$$\mathsf{P}_0(uv) = \prod_{P_{d,i} \in \mathcal{P}_{uv}} (1 - \lambda_{d,i})$$

The probability for an edge not to be selected is maximized when all $\lambda_{d,i}$ are equal (i.e., $\lambda_{d,i} = \frac{1}{|\mathcal{P}_{uv}|}\ \forall\ \lambda_{d,i} \in \mathcal{P}_{uv}$). Thus,

$$\mathsf{P}_0(uv) = (1 - \frac{1}{\rho|\mathcal{P}_{uv}|})^{|\mathcal{P}_{uv}|}$$

where ρ is defined to be $\frac{\mathbb{E}[f_{uv}]}{c^*_{uv}}$. This gives an upper bound for the possible value of $\mathsf{P}_0(uv)$. Indeed,

$$\mathsf{P}_0(uv) \leq \lim_{n \to \infty} (1 - \frac{1}{\rho n})^n = \frac{1}{\mathrm{e}^\rho}.$$

The function is minimized with $\rho = 1$. We thus get

$$\begin{aligned}\mathbb{E}[O_{uv}] &\leq \frac{1}{\mathrm{e}^\rho} c^*_{uv} - (c^*_{uv} - \mathbb{E}[f_{uv}]) \\ &\leq c^*_{uv}(\frac{1}{\mathrm{e}^\rho} - (1 - \rho)) \leq \frac{1}{\mathrm{e}} c^*_{uv} \approx 0.37 c^*_{uv}.\end{aligned}$$

Algorithm 5 Iterative Randomized Rounding

1: Solve the linear relaxation of the general problem
2: $\tilde{c} \leftarrow c^*$
3: **for each** $r \in \mathcal{R}$ **do**
4: (a) route the demands on $G' = (V, E \setminus r, \tilde{c})$ solving a fractional multicommodity flow problem
5: (b) use Randomized Rounding to find a $(1+\frac{1}{e}+\epsilon)$–approximate integer routing
6: (c) update $\tilde{c}$ with the introduced overflow (if any)
7: **end for**
8: **return** $\tilde{c}$

Finally, the expected cost of the solution provided is

$$\mathbb{E}\left[\frac{\sum_{(u,v)\in E} O_{uv}}{\sum_{(u,v)\in E} c^*_{uv}}\right] = \frac{\sum_{(u,v)\in E} \mathbb{E}[O_{uv}]}{\sum_{(u,v)\in E} c^*_{uv}} \leq \frac{1}{\mathrm{e}} \approx 0.37.$$

By using the Markov inequality, the probability that the obtained solution has a cost larger than $1.37(1+\epsilon)$ is at most $\frac{1}{1+\epsilon}$. The overflow resulting from the execution of the randomized rounding can be checked in polynomial time. If the overflow exceeds the factor of $(1+\frac{1}{e})+\epsilon$, another trial may be necessary in order to find a solution below this value. The number of trials depends on the chosen value for ϵ. For instance, if we set $\epsilon = \frac{1}{10}$, we need an average of 10 trials in order to find a solution with cost not greater than 1.507 ($= 1.37 + 0.137$) times the optimal fractional one. As we have just shown, the problem of minimizing the overflow can be approximated efficiently for a single scenario. The proposed scheme consists in a randomized rounding to be performed according to the value of the splittable flow solution. We may extend Randomized Rounding to the case of multiple scenarios by simply solving the scenarios in an iterative fashion. At each iteration, an SRLG $r \in \mathcal{R}$ is considered. First, a fractional capacitated multicommodity flow is solved. Then, a $(1+\frac{1}{e}+\epsilon)$–approximated integer solution is found using the Randomized Rounding procedure described in Algorithm 6. The overflow introduced (if any) by the procedure is then added. We refer to this method as Iterative Randomized Rounding - see Algorithm 5 for the pseudo-code of our proposed algorithm.

6.4 Numerical Results

In this section, we evaluate the performances of our proposed algorithms on both real and synthetic network topologies and workloads. The compared methods are MasterILP, in which the last RMP is solved as an ILP by setting the domain of the path variables from fractional to binary. IterILP, in which each scenario is solved independently with an ILP that has, as a goal, the minimization of the overflow and ITERATIVE RANDOMIZED ROUNDING for which, instead

Algorithm 6 RANDOMIZED ROUNDING

Require: A network with a capacity function $\tilde{c}$ and a fractional routing for a set of demands $\mathcal{D}$
Ensure: A $(1 + \frac{1}{\mathrm{e}} + \epsilon)$–approximate integer routing
1: **repeat**
2: **for each** $d \in \mathcal{D}$ **do** ▷ Build an integral routing
3: select a path $P_{d,i}$ among all fractional paths p_i with probability $\lambda_{d,i}(uv)$.
4: **end for**
5: $c_i \leftarrow$ compute the capacities of the drawn integral routing
6: overflow $= 0$ ▷ Compute the overflow
7: **for each** $e \in E$ **do**
8: overflow $+= \frac{\max(c_i, \tilde{c})}{\tilde{c}}$
9: **end for**
10: **until** overflow $\leq (1+\frac{1}{\mathrm{e}}+\epsilon)|E|$ **return** c_i, the capacities of the drawn integral routing

of using an ILP to minimize the overflow, we use a $(1 + \frac{1}{\mathrm{e}} + \epsilon)$–approximation algorithm. We show the effectiveness

- of our algorithms in terms of *scalability*. Indeed, we are able to solve instances with much larger numbers of demands, network nodes and links than the classic compact ILP model, and
- of GLOBAL REROUTING in terms of *bandwidth usage*, i.e., the bandwidth needed to be provisioned by an operator to handle failures is minimum.

Data Sets

We conduct experiments on three real-world topologies from SNDlib [Orl+10]: `polska`, (12 nodes, 18 links, and 66 demands), `pdh` (11 nodes, 34 links, and 24 demands) and `nobel-germany` (17 nodes, 26 links, and 121 demands). For these networks, we use the traffic matrices provided with the data set. No information is available about the SRLGs for these networks. Thus, the collection of network failures $\mathcal{R}$ for these instances contains single edge failures.
We also conduct experiments on randomly generated instances of different sizes and with different SRLGs. We build our synthetic instances using a similar method to the one in [KKV05]. We generate two networks in which we place nodes in a unit square. In each of them, we add links according to the Waxman model [Wax88]. The probability of having a link (u, v) is defined as $\alpha \exp^{\frac{-\mathrm{dist}(u,v)}{\beta L}}$ where $\mathrm{dist}(u, v)$ is the Euclidean distance from node u to node v, L is the maximum distance between two nodes and α, β are real parameters in the range $[0, 1]$. One of the two networks represents the logical IP network, i.e., IP routers and IP links while the other represents the underlying optical

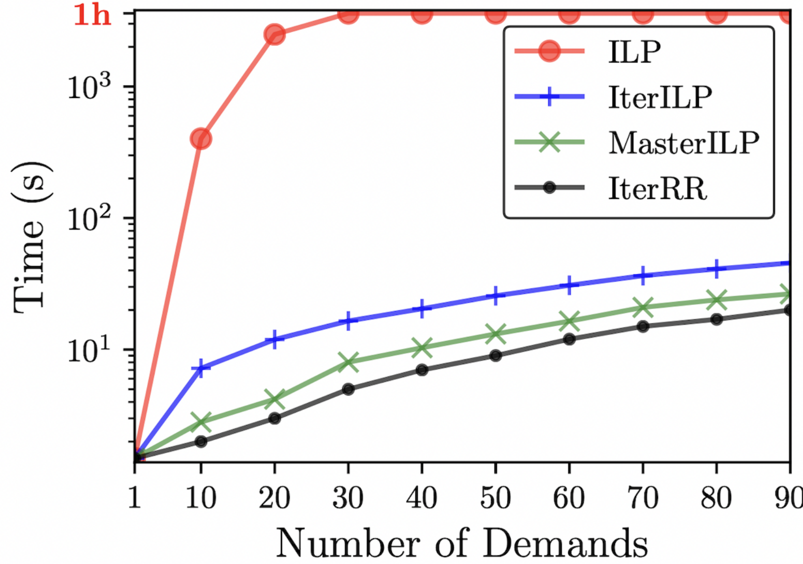


Figure 6.3: Time of the solution found by the ILP and by our proposed methods as a function of the number of demands.

network, i.e., cross-connect and fibers. Each IP node is mapped to the closest optical cross-connect and each IP link (u, v) is mapped onto the shortest path between u and v in the physical network.
All IP links using the same physical link are associated to an SRLG. In addition, we add an SRLG for each undirected link. Demands are generated using the model described in [FT02]. The model considers the distance factor $\exp^{\frac{-\text{dist}(u,v)}{2L}}$ between two nodes u and v. As a result, the load of the demands between close pairs of nodes is higher with respect to pairs of nodes far apart.
Finally, the chain of each demand is composed of 3 to 6 functions uniformly chosen at random from a set of 10 functions. Each NFVI node can run up to 6 network functions. Indeed, a node may not be allowed to run all the network functions. Similarly as in [HJG18], locations are chosen according to their betweenness centrality, an index of the importance of a node in the network: it is the fraction of all shortest paths between any two nodes that pass through a given node. Experiments have been conducted on an Intel Xeon E5520 with 24 GB of RAM.

Limits Of An ILP-based Approach

To study the limits in terms of computing time of an ILP-based approach, we tested our optimization models on a small random topology with 10 nodes, 16 links, and 26 SRLGs. In Fig. 6.4 and Fig. 6.3, we show the impact of the number

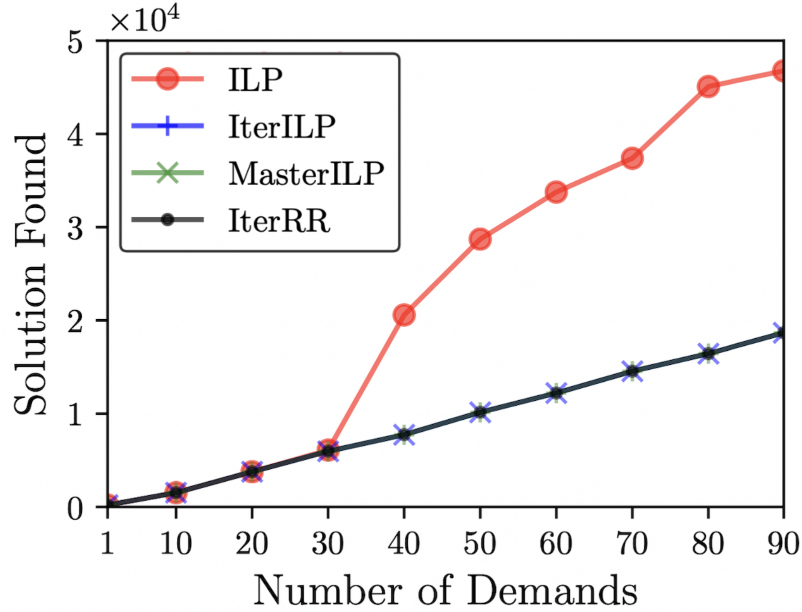


Figure 6.4: Value of the solution found by the ILP and by our proposed methods as a function of the number of demands.

of demands on the execution time. We compare the time necessary to find an optimal solution (Fig. 6.4) and the value of the solution found (Fig. 6.3) by the ILP and by our proposed methods. For each experiment, we set a maximum time limit of one hour. If the time limit is exceeded, the solution reported represents the best solution found so far.
For just 30 demands, the time needed by IBM ILOG Cplex 12.8 to find an exact solution exceeds 1 hour and for larger instances, no optimal solution can be found using an ILP approach in a reasonable amount of time. On the contrary, the algorithms we propose can compute solutions for larger instances with a fair efficiency. Indeed, our algorithms only take 1 minute to solve the problem for 90 demands. As the considered network is small, the computed values by the three proposed methods are very close between them. We compare them in the following on larger networks.

Performances Of The Optimization Models

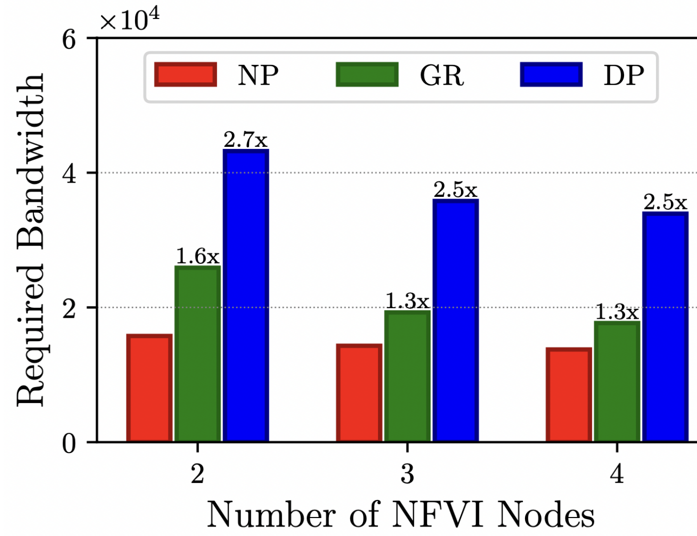
Table 6.3 summarizes the results of our proposed methods for the three real networks and for four Waxman random networks. Networks are identified as `wxm_N` with `N` being the number of nodes.
The number of demands is set to be $50, 100, 150$, and 200 for the $10, 20, 30$, and 40 nodes networks, respectively. Moreover, the number of resulting SRLGs for

the Waxman random networks are $22, 40, 53$, and 70, respectively. The first column compares the Column Generation (ColGen) and the Benders Decomposition [Ben62] (Benders) techniques to find a fractional solution based on which the heuristics find an integer solution. The Column Generation technique appears to be faster in finding the optimal solution z^*_{LP}: on the largest considered network, `wxm40` only takes 22 minutes to find an optimal solution, while Benders would require more than one hour. The remaining three columns refer to our optimization methods. For each method, we present both the time needed to find a solution $\tilde{z}_{ILP}$, as well as the ratio $\epsilon = \frac{\tilde{z}_{ILP} - z^*_{LP}}{z^*_{LP}}$ with respect to the optimal fractional solution z^*_{LP}. ϵ, called *accuracy* in the following, gives an upper bound on the maximum overflow to pay in excess with respect to the optimal integer solution z^*_{ILP}, since the optimal integer solution may be larger than the fractional one.
Both MasterILP and IterILP allow to find near–optimal solutions. As the size of the network increases, we begin to observe the limits of the IterILP approach, as it solves an ILP for each scenario. Although MasterILP demonstrates a better scalability and a very high accuracy, for larger networks we have a tradeoff between the time to find the solution and the quality of the solution found. Indeed, for `wxm40`, IterRR only takes 2 minutes to find a good solution with an accuracy of about 9%, while MasterILP requires 27 minutes to find a solution with an accuracy of 2.2%.

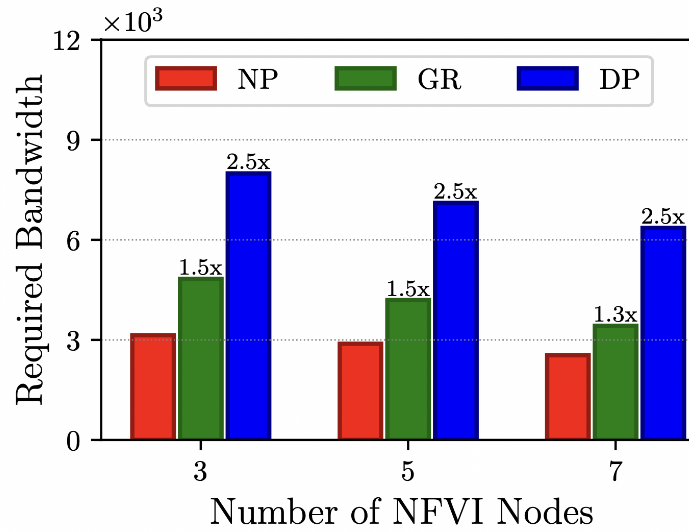
Varying The Number Of NFVI Nodes

NFVI nodes are expensive to both purchase and maintain (e.g., hardware, software licenses, energy consumption, and maintenance). If, on one hand, over-provisioning corresponds to undue extra costs, on the other hand, under-provisioning may result in poor service to user and in Service Level Agreement (SLA) violations. It is thus necessary to find the right trade-off in terms of NFVI nodes in the network design phase.

Bandwidth overhead. In Fig. 6.5, we compare the overhead in terms of bandwidth needed in the network by the global rerouting scheme and Dedicated Path Protection with respect to the bandwidth needed in the unprotected case. For Dedicated Path Protection, we compute for each demand two SRLG-disjoint paths, i.e., two paths such that no link on one path has a common risk with any link on the other path. By doing so, we set the bandwidth minimization as an optimization task. With an increasing number of NFVI nodes in the network, the required bandwidth decreases. However, the overhead with respect to the unprotected case tends to remain constant. Indeed, if with global rerouting we only need from 30 to 60% more bandwidth, with dedicated path protection we may need almost three times more bandwidth to guarantee the recovery.



(a) Pdh.



(b) Nobel-germany.

Figure 6.5: Bandwidth overhead comparison of the Global Rerouting (GR) and Dedicated Path Protection (DP) schemes with respect to the No-Protection scenario (NP) for `pdh` and `Nobel-germany` networks. Labels on top of the bars indicate the overhead with respect to the unprotected case.

Network	z^*_{LP}		MasterILP		IterILP		IterRR	
	ColGen	Benders	time	ϵ	time	ϵ	time	ϵ
pdh	22s	32s	11mn	4%	1mn	4.82%	40s	12.7%
polska	15s	18s	40s	0.22%	1mn	0.1%	20s	1.4%
nb-germany	35s	1mn	40s	0.17%	4mn	0.06%	30s	3.2%
wxm10	10s	5s	50s	0.3%	40s	1%	10s	5.5%
wxm20	40s	2mn	1mn	0.6%	4mn	0.6%	30s	2.7%
wxm30	3mn	16mn	6mn	0.2%	21mn	0.9%	1mn	4.5%
wxm40	22mn	>1h	27mn	2.2%	>1h	-	2mn	9.2%

Table 6.3: Numerical results for the proposed optimization models. First column refers to the time needed to find the optimal fractional solution z^*_{LP}. We set a maximum time limit of 1h. The other columns refer to the proposed methods to obtain an integer solution $\tilde{z}_{ILP}$. For each method, we show the additional time needed and the quality of the solution found, expressed as the ratio $\epsilon = \frac{\tilde{z}_{ILP} - z^*_{LP}}{z^*_{LP}}$.

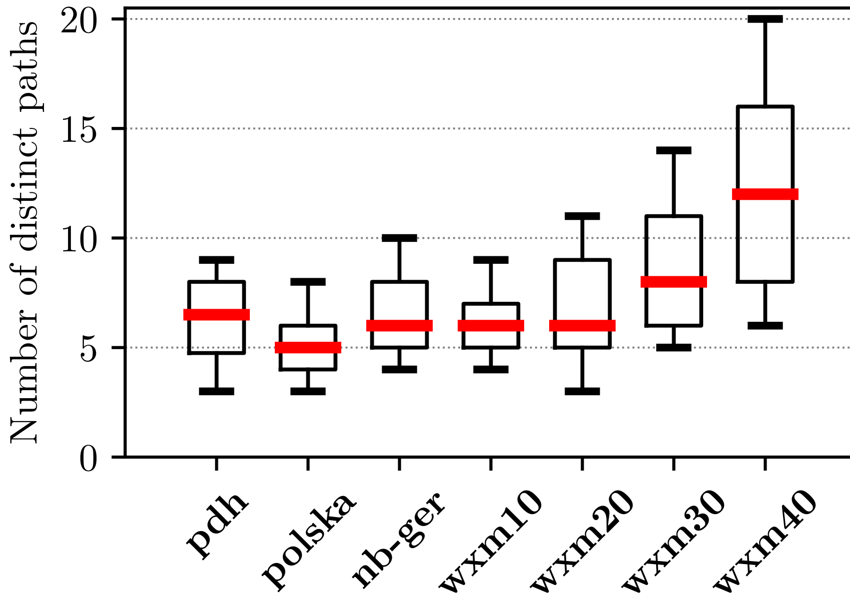
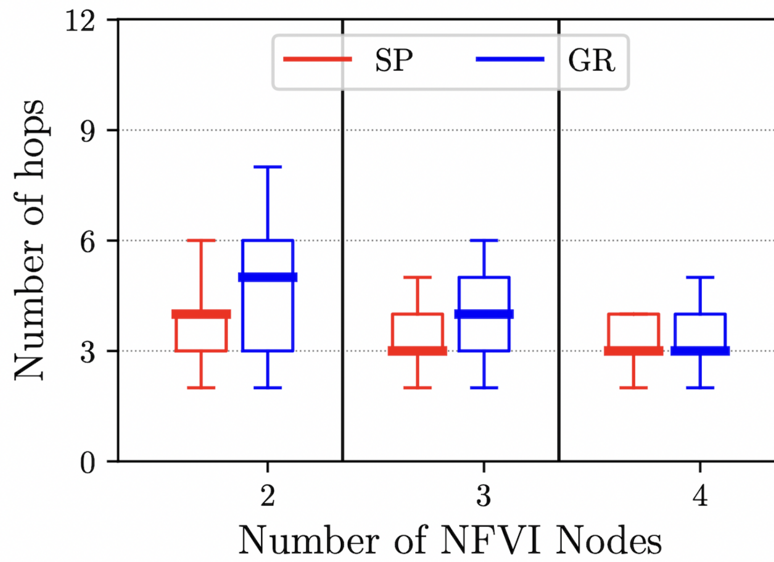


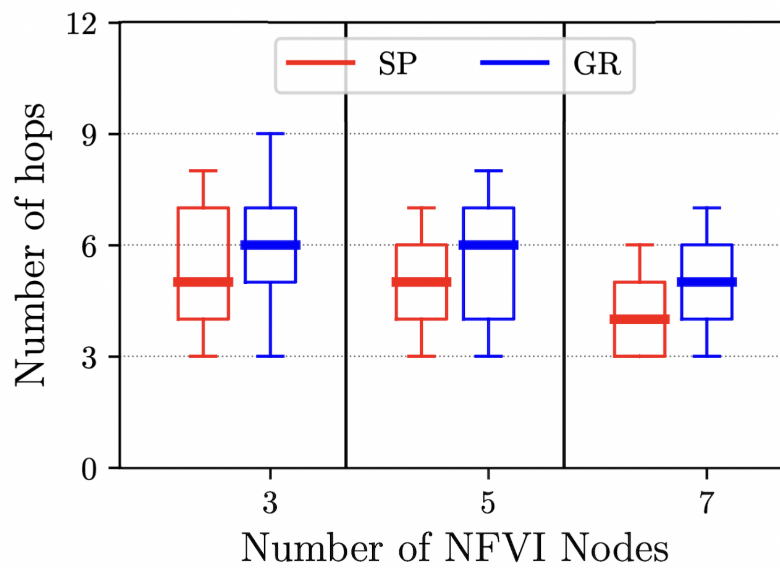
Figure 6.6: Distribution for the number of distinct paths for each demand. Boxes are defined by the first and third quartiles. Ends of the whiskers correspond to the first and ninth deciles. The median value is drawn in red.

Paths' delays. In Fig. 6.7, we show the impact of the number of NFVI nodes on the path latency distribution and compare them with the ones calculated using shortest paths on the layered network. As expected, we see that the number of hops decreases as the number of NFVI nodes increases.
Actually, more there are NFVI-nodes in a network, higher is the opportunity to find easily closer NFVI-nodes which can perform some of the required network functions. Another result is that the paths computed using our method are almost as short (in terms of number of hops) as the shortest paths.

Number of paths. In our considered protection scheme, a demand may be rerouted on a different path in each of the possible SRLG failure situations. Even though our optimization models do not impose constraints on the number of distinct paths for a demand, the experimental results indicate that their number tends to be small in practice.
In Fig. 6.6, we show the distribution for the number of distinct paths of the demands for our considered networks. The number of distinct paths increases with the size of the network and tends to stay within the range $(5, 10)$ for most of them. For instance, for `wxm40` we may have potentially 71 distinct paths to be used for a demand, one for each of the possible SRLG failure scenarios plus one for the case without failure. However, as the results show, in such a case, 50% of the demands would use no more than 12 distinct paths.



(a) Pdh.



(b) Nobel-germany.

Figure 6.7: Hops distribution of the backup paths computed by the global rerouting scheme (GR) compared with the shortest paths (SP) for `pdh` and `nobel-germany` networks.

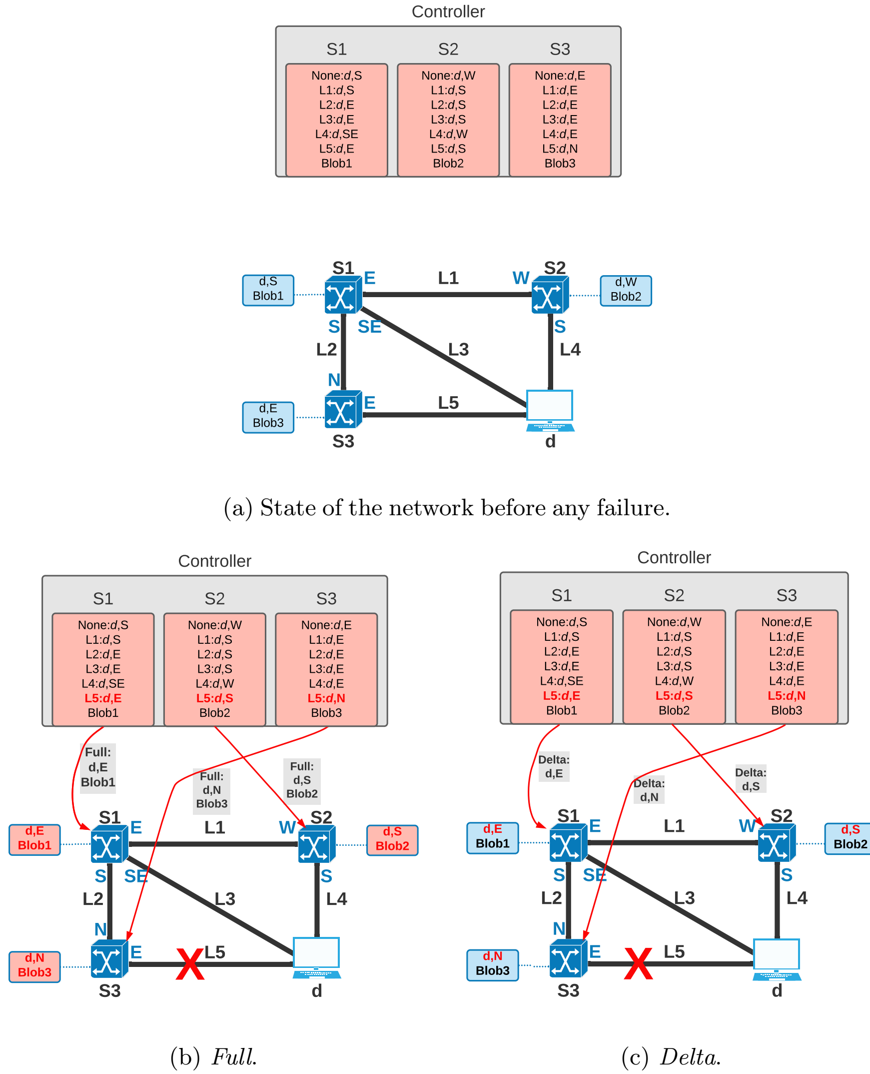
6.5 Implementation Perspectives

In the previous sections, we followed a theoretical approach to design a bandwidth-optimal failure recovery scheme relying on the hypothesis that programmable networks should be able to implement such a scheme. In this section, we verify this assumption by implementing the scheme on production SDN appliances, namely Open vSwitch and OpenDayLight, and evaluate different trade-offs with Mininet. Our evaluation shows that implementation choices have a significant impact on the recovery time of protection mechanisms.

6.5.1 Implementation Options

A first option to implement the protection scheme in OpenFlow is to let the OpenFlow controller fully update the flow tables on the switches upon failure. When the controller detects a failure, it sends the new flow tables to the impacted switches. This approach minimizes the memory usage on the switches, but incurs high signaling overhead between the controller and the switches, and imposes the latter to install a full flow table at every network change. We refer to this option as *full*. A variation of this option is to only send the changes to be performed on the flow tables to the switches to reduce the signaling load and the number of flow table updates on the switches. We name this option *delta*. Another option is to leverage the Multiple Flow Tables capability introduced in OpenFlow 1.3 to pre-install the flow tables for each SRLG failure scenario in the switches. When the controller sends a failure notification to a switch, the switch activates the appropriate flow table in only one operation (using goto). This approach minimizes the signaling load and flow table changes, but consumes more memory on the switches than the other options. This option is referred to as *notification*. In the rest of this chapter, we study the impact of the technical choices on the recovery time in realistic operational scenarios.

Fig. 6.8 illustrates how the `full` and `delta` implementation options presented above operate. The network is composed of three switches (`S1`, `S2`, `S3`), one destination for traffic (`d`), and one controller. The controller has the full knowledge of the network and pre-computed the alternative flow tables for each potential failure. To save space, we only show the information related to link failures for host `d`, the rest (e.g., node failures, SRLGs, other destinations. . .) of the table is hidden behind the *blob* term. Figs. 6.8b and 6.8c show the messages sent by the controller to the switches to recover from the failure of link `L5` and the resulting flow tables on the switch. Fig. 6.9 is the counterpart of Fig. 6.8 for the `notification` implementation option. In this case, the controller does not store flow tables in memory. Instead, it stores the goto operations to be performed in case of failures. This implementation minimizes the controller's computational effort in case of failure because every possible rule is precomputed and installed in the switches. In case of failure the controller sends a single message to each switch, with a single rule update (the goto rule). Fig. 6.9b shows the goto instructions sent to the switches to recover from the failure of link `L5` and how it changes the forwarding decisions on the switches.



(a) State of the network before any failure.

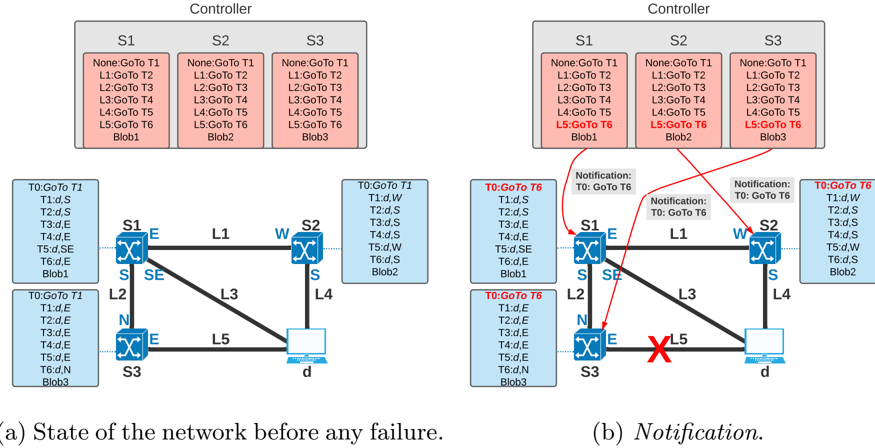
(b) *Full*.

(c) *Delta*.

Figure 6.8: Switch update operations for the `full` and `delta` protection scheme implementation options in SDN networks upon one link failure. In the `full` protection scheme, the table inside the switches are completely overwritten by the new tables. In the `delta` protection scheme, the controller only modifies the rules that are changing in the failure scenario.

6.5.2 Experimental Setup

Our experimental platform is a dual Intel Xeon E5-2630 CPU server with 128 GB of RAM running Mininet 2.2.2 [LHM10] and the controller *OpenDaylight*



(a) State of the network before any failure.

(b) *Notification*.

Figure 6.9: Switch update operations for the `notification` protection scheme implementation option in SDN networks upon one link failure. The rules are preinstalled in the switches in a different table for each failure scenario, and `table 0` is pointing to the `no failure table` in the normal state. In a failure situation, the controller updates the pointer in `table 0` to use the table assigned for the failure scenario.

Oxygen [Med+15] with OpenFlow 1.3.
The routing logic is implemented as a network application orchestrator that communicates with the controller with the HTTP OpenDaylight Northbound API. This approach is recommended as it decouples the implementation of the logic from the implementation of the controller.

We also made an *ideal* implementation to assess the best possible performance one could have. It is equivalent to the *notification* option, but is implemented directly in Mininet with Open vSwitch commands. In this case, the switches are programmed directly, without the controller overhead. Mininet emulation is centralized, so we are able to synchronize all failure notifications to the switches just after the failure occurs, bypassing thus the controller.

Due to the limited number of CPU cores on our emulation server, we could only evaluate the `wxm10` and the `polska` networks.

6.5.3 Recovery Time

The *recovery time* is the span of time between a failure event and the moment in which all switches are updated to be in a state that circumvents the failure. To measure the recovery time, we continuously probe the end-to-end paths with UDP datagrams. Fig. 6.10 shows the recovery time for our three OpenDaylight implementation options and the ideal one. It compares our Global Rerouting protection scheme to the Dedicated Path protection scheme. The figure highlights the importance of implementation choices on the recovery time: the

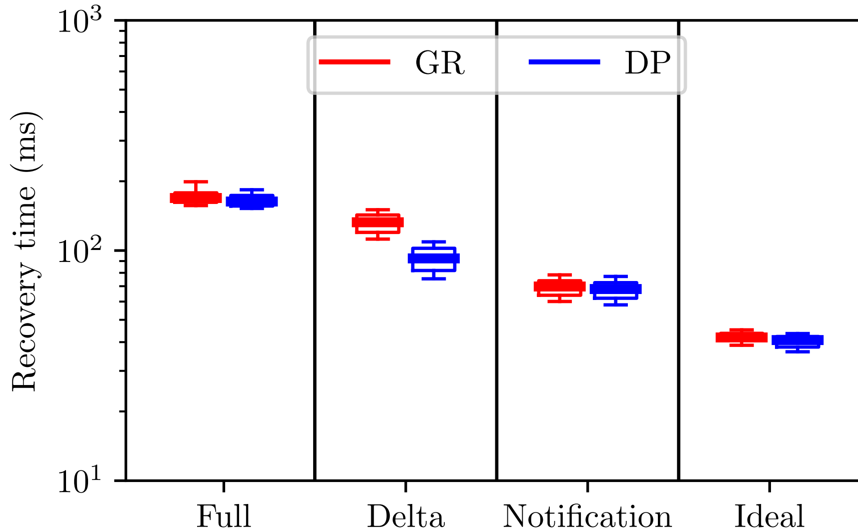


Figure 6.10: Recovery time comparison of various implementation options for Global Rerouting (GR) and Dedicated Path Protection (DP) for the `polska` network.

notification option significantly outperforms the other options. The ideal implementation also shows that the tools used to implement the protection scheme have a significant impact on the recovery time as, all things considered, our ideal is just a way of implementing the notification option without a controller. Actually, a significant fraction of the recovery time in OpenDaylight implementations is caused by the usage of the Northbound API. All implementation options offer sub-second recovery time for the considered network.

Figs. 6.10 and 6.14 show that there is a direct link between the number of changes to be performed on the switches and the recovery time in Polska network. The same observation applies to the wx10 network as highlighted by Figs. 6.11 and 6.15. Figs. 6.14 and 6.15 report, for each switch, the maximum number of flow table changes observed expressed in number of flow entries for the three OpenDaylight implementation options. Dedicated path protection has similar recovery time than global rerouting when the full implementation is used. This is because the number of flows to install on switches is similar between dedicated path protection and global rerouting. Similarly, no performance difference can be observed when the notifications based implementation is used as the controller only has to update one rule per switch. On the contrary, if updates are performed with the delta implementation, then dedicated path protection converges faster than global rerouting, as it requires much fewer path changes.

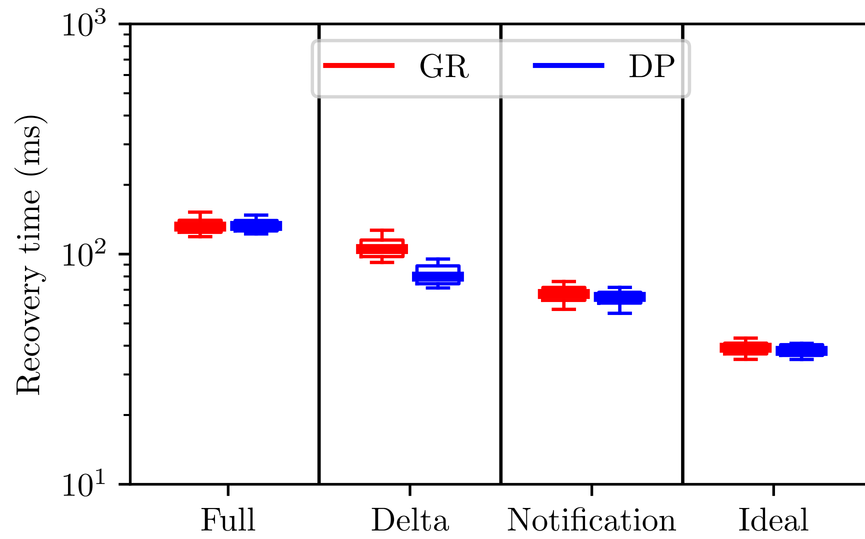


Figure 6.11: Recovery time comparison of various implementation options for Global Rerouting (GR) and Dedicated Path Protection (DP) for the `wxm10` network.

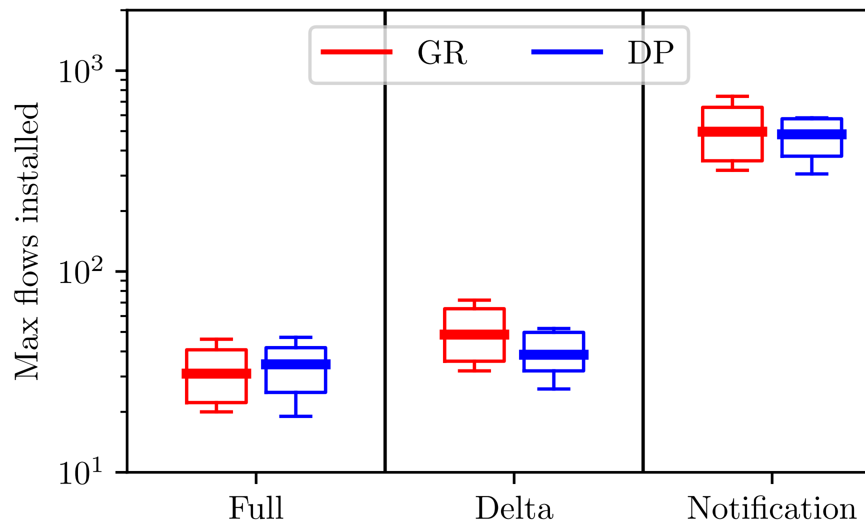


Figure 6.12: Comparison of the flow table sizes of various implementation options for Global Rerouting (GR) and Dedicated Path Protection (DP) for the `polska` network.

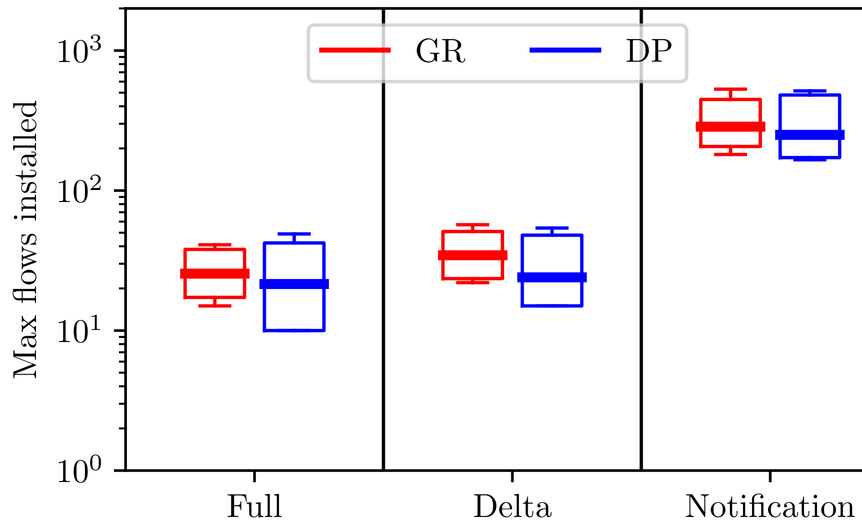


Figure 6.13: Comparison of the flow table sizes of various implementation options for Global Rerouting (GR) and Dedicated Path Protection (DP) for the `wxm10` network.

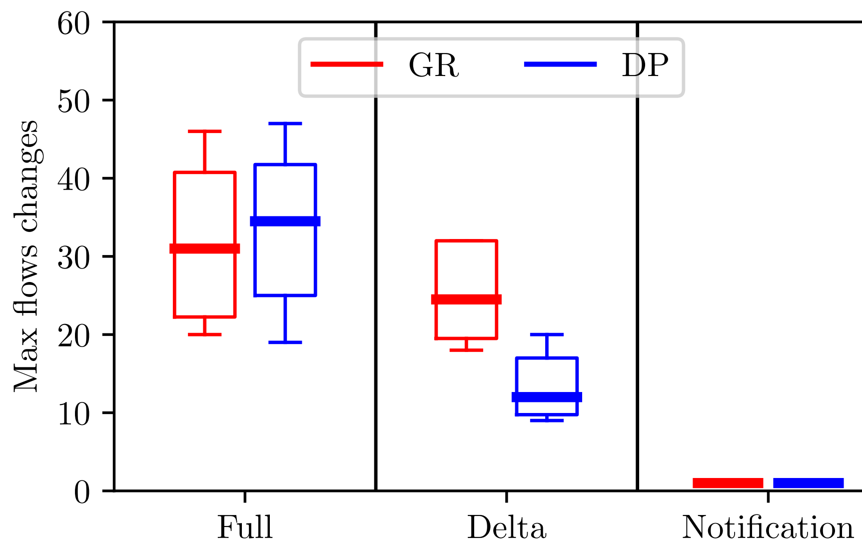


Figure 6.14: Comparison of the number of flow table changes of various implementation options for Global Rerouting (GR) and Dedicated Path Protection (DP) for the `polska` network.

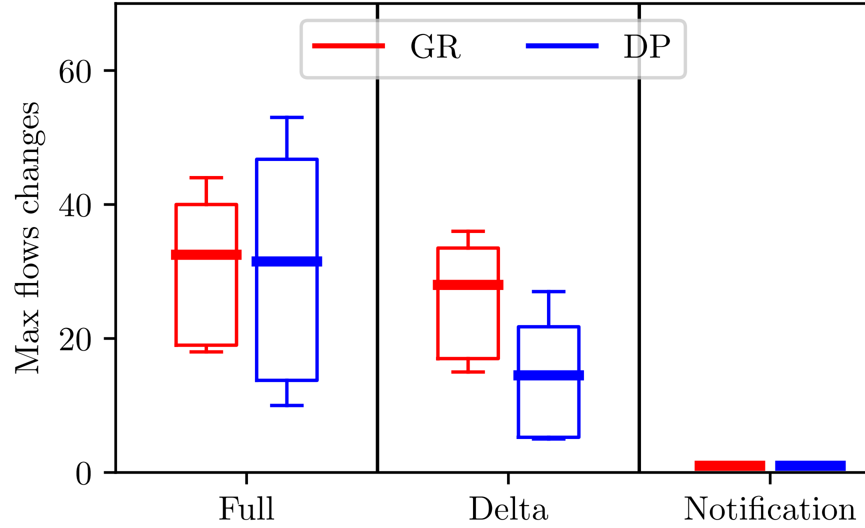


Figure 6.15: Comparison of the number of flow table changes of various implementation options for Global Rerouting (GR) and Dedicated Path Protection (DP) for the `wxm10` network.

6.5.4 Operational Trade-Offs

Based on the recovery time, one would recommend deploying the notification option. However, the reduction of the recovery time comes at the cost of increasing flow table sizes on switches as shown in Fig. 6.12 and Fig. 6.13. Actually, these figures report, for each switch, the maximum observed flow table size expressed in number of flow entries for the three OpenDaylight implementation options in both networks. The full option minimizes the number of entries, as it only requires to have the flow table for the current routing case. The delta option consumes slightly more space than the full one, as the flow table always contains the "no-failure" scenario flow table and the additional flow entries needed to circumvent the current failure. Finally, the notification option has significantly larger flow tables (one order of magnitude more), as flow tables always contain all the potential failure scenarios, which may prevent it to be used in practice on low end switches or for large networks.
As the robustness of the controller is an orthogonal problem that must be treated by all SDN solutions and because it is already largely studied [Zha+18], it was not considered here.

6.6 Conclusion

In this chapter, we studied the network dimensioning problem with protection against a Shared Risk Link Group (SRLG) failure in the light of network virtualization. We considered a path-protection method based on a global rerouting strategy, which makes the protection method optimal in terms of bandwidth. We proposed algorithms to compute the backup paths for the demands which rely on the Column Generation and Benders Decomposition techniques. We validated them with simulations on real-world and on randomly generated network topologies and workloads. Simulations show that our algorithms can reach near-optimal solutions in a short time, even for large instances. Finally, we proposed a real implementation of our proposition in OpenDaylight and show the applicability of the global rerouting protection method when SDN is used. The experimental results in Mininet show that our solution provides sub-second recovery times, but the way it is implemented may greatly impact the amount of signaling traffic exchanged. In our evaluations, the recovery phase requires only a few tens of milliseconds for the fastest implementation, compared to a few hundreds of milliseconds for the slowest one.

Chapter 7

Conclusion And Future Work

7.1 Summary Of Contributions

Cloud computing is the next norm. Companies are investing a massive amount of resources for migrating to the cloud. Software Defined Networking and Network Function Virtualization are paving the way how Internet Service Providers manage and distribute their services to their clients.

In Chapter 3 and Chapter 4, we have extensively analyzed Distrinet, a distributed network emulator for SDN and NFV networks. Chapter 3 focused on the technical choices made to build the tool. We analyzed the main tools available (MaxiNet and Mininet Cluster Edition), showing their limitations and how Distrinet helps to resolve these limitations. The experiments are done using the same allocation algorithm for all the tools. We ran Iperf on different virtual topologies in a single host with Mininet and on two hosts with the distributed tools. We showed that Distrinet is the one that returns the closest results to Mininet. We also analyzed the creation and removal time for each element of the virtual networks, demonstrating that the network creation time in Distrinet is slower, due to a better container isolation and the compatibility with the Mininet API.

In Chapter 4, we analyzed the optimization problem in the case of distributed network emulation. As the challenge is a Network Embedding Problem, we proposed three different algorithms and compared them with the algorithms implemented in the other network emulation tools. We tested more than 75,000 scenarios in heterogeneous and homogeneous networks. The algorithms proposed to find a feasible solution for almost all the tests, while the algorithms used in the other tools often return that overload occurs in a physical host or a physical link. To justify the need for algorithms returning a solution that is not overloaded, we conducted experiments showing the impact of the overloading (for CPU, RAM, and link). These experiments proved that a wrong allocation

of the virtual network could lead to results that are not trustable, or a complete crash of the test-bed.

In Chapter 5, we considered the cloud computing topic. Organizations that are planning to move their applications into the cloud should first check that the cloud network will support these applications. On premises datacenter, the delay is usually low, while cloud infrastructures do not provide a specific value for the delay in a region or in between two different regions. To monitor the delay within the cloud infrastructure, we implemented CloudTrace, a simple tool that automatizes the deployment and the configuration of multiregional and regional Virtual Private Clouds. We first showed the main services proposed by Amazon Web Services to provide Infrastructure as a Service deployment. After introducing the single components, we described how to merge them to provide a multiregional and regional environment. CloudTrace automatically creates the environments on the regions specified by the user, then manages all the virtual instances, updating them and installing all the requirements. The user can run the experiment after the deployment. CloudTrace will synchronize Paris-Traceroute to run each minute. Results can be retrieved by the user at any moment, without stopping the experiments. CloudTrace is finally able to analyze the data received and build a map with all the data collected.

In our last contribution (Chapter 6), we investigated different problems and proposed new solutions in order to minimize the bandwidth used in the network. The challenge is to design a network, considering a path-based protection scheme with a global rerouting strategy. In the worst case, using a global rerouting strategy, we may have a new routing of all the flows. This means that the controller has to send many updates to the switches. We extensively experimented the new approach, comparing it with the dedicated path protection one. For the experiment test-bed, we used Mininet to emulate the networks and OpenDaylight as a controller. We demonstrated that there is not a significant increase in the update rule time in both strategies. However, the global rerouting consumes on average 40% less bandwidth with respect to the dedicated path protection. The experiments ran on real-world network topologies and randomly generated instances. We also presented three different implementation strategies. We showed that the technical implementation choices greatly impact the time needed to reestablish the flows after a failure occurs.

7.2 Future Work

There are multiple ways to continue our research. For distributed network emulation and Distrinet, it could be interesting to study the variation and the correctness of the results in cloud environments. Investigating which scenarios and experiments are suitable for the use of public clouds to perform emulation helps to identify the scenarios where it is not possible to use clouds due to their shared resources (including networking). In such cases, it is more convenient to use private test-beds in a controlled environment.

Another challenge is to see how much the virtualization layer of public and

private cloud infrastructures is affecting the emulation results. It is also possible to extend Distrinet to use Docker containers by default, and measure if the overhead added by LXC is different from Docker overhead when emulating a network. There is also another problem when distributing the emulation, such as synchronization of the virtual nodes on different physical hosts.

The proposed optimization algorithms used in Distrinet consider general physical infrastructure and general virtual networks for emulation. One enhancement is to build an algorithm that optimizes the distribution in fixed networks topologies used in datacenter or test-beds, such as fat-tree topology. Distrinet also does not automatically program the physical switches in the network. It could be interesting to create an SDN feature to allow Distrinet to automatically manage private test-beds' physical infrastructure, for example, Grid'5000.

Regarding the CloudTrace tool, it can be extended to work with Microsoft Azure and Google cloud in order to compare the network delay stability of the different cloud providers. It could also be interesting to improve it to deploy hybrid cloud environments, to see what are the delays between different cloud providers, and check their stability during the day.

Lastly, to continue the failure recovery strategy, it is worth studying the behavior of network recovery using segment routing techniques in the edge routers (e.g., using MPLS). It can also be studied in-depth if the different implementation options return similar results with physical routers.

Bibliography

[ADP80] Giorgio Ausiello, Alessandro D'Atri, and Marco Protasi. "Structure preserving reductions among convex optimization problems". In: *Journal of Computer and System Sciences* 21.1 (1980), pp. 136–153 (cit. on p. 99).

[Ama20a] Amazon. *AWS Global Infrastructure documentation*. https://aws.amazon.com/about-aws/global-infrastructure/regions_az/. 2020 (cit. on p. 82).

[Ama20b] Amazon. *Cloudfront documentation*. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html. 2020 (cit. on p. 81).

[Ama20c] Amazon. *EC2 instance types*. https://aws.amazon.com/ec2/instance-types/. 2020 (cit. on p. 83).

[Ama20d] Amazon. *EC2 service documentation*. https://aws.amazon.com/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc. 2020 (cit. on pp. 80, 83).

[Ama20e] Amazon. *M5 instances AWS*. https://aws.amazon.com/ec2/pricing/on-demand/?nc1=h_ls. 2020 (cit. on pp. 39, 42).

[Ama20f] Amazon. *Simple Service Storage documentation*. https://aws.amazon.com/s3/reduced-redundancy/. 2020 (cit. on pp. 81, 82).

[Ama20g] Amazon. *SQS documentation*. https://aws.amazon.com/sqs/. 2020 (cit. on p. 81).

[Ama20h] Amazon. *VPC Peering AWS documentation*. https://docs.aws.amazon.com/vpc/latest/peering/what-is-vpc-peering.html. 2020 (cit. on p. 83).

[Arn+20] Todd Arnold, Jia He, Weifan Jiang, Matt Calder, Italo Cunha, Vasileios Giotsas, and Ethan Katz-Bassett. "Cloud Provider Connectivity in the Flat Internet". In: *Proceedings of the ACM Internet Measurement Conference*. 2020, pp. 230–246 (cit. on p. 81).

[Ass+20] K. D. R. Assis, R. C. Almeida, H. Waldman, M. J. Reed, B. Jaumard, and D. Simeonidou. “Linear formulation for the design of elastic optical networks with squeezing protection and shared risk link group: Invited Paper”. In: *2020 22nd International Conference on Transparent Optical Networks (ICTON)*. 2020, pp. 1–5. DOI: `10.1109/ICTON51198.2020.9203486` (cit. on pp. 95, 96).

[AV14] YK Agarwal and Prahalad Venkateshan. “Survivable network design with shared-protection routing”. In: *European Journal of Operational Research* 238.3 (2014), pp. 836–845 (cit. on p. 94).

[Bal+13] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. “Adding Virtualization Capabilities to the Grid’5000 Testbed”. In: *Cloud Computing and Services Science*. Ed. by Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: `10.1007/978-3-319-04519-1_1` (cit. on pp. 39, 58, 71).

[BBS] Michael Till Beck, Juan Felipe Botero, and Kai Samelin. “Resilient allocation of service Function chains”. In: *IEEE NFV-SDN2016*. IEEE (cit. on p. 94).

[Ben62] Jacques F Benders. “Partitioning procedures for solving mixed-variables programming problems”. In: *Numerische mathematik* 4.1 (1962) (cit. on pp. 103, 113).

[Ber+13] I. Bermudez, S. Traverso, M. Mellia, and M. Munafò. “Exploring the cloud from passive measurements: The Amazon AWS case”. In: *2013 Proceedings IEEE INFOCOM*. 2013, pp. 230–234. DOI: `10.1109/INFCOM.2013.6566769` (cit. on p. 81).

[Bot+12] Juan Felipe Botero, Xavier Hesselbach, Michael Duelli, Daniel Schlosser, Andreas Fischer, and Hermann De Meer. “Energy efficient virtual network embedding”. In: *IEEE Communications Letters* 16.5 (2012), pp. 756–759 (cit. on p. 49).

[BSY18] Mostafa Bastam, Masoud Sabaei, and Ruhollah Yousefpour. “A scalable traffic engineering technique in an SDN-based data center network”. In: *Transactions on Emerging Telecommunications Technologies* 29.2 (2018), e3268 (cit. on p. 96).

[Can19] Canonical. *Linux Containers*. `https://linuxcontainers.org`. Accessed: 2019-05-10. 2019 (cit. on pp. 33, 35).

[Can20] Canonical. *Ubuntu Fan Networking*. `https://wiki.ubuntu.com/FanNetworking`. 2020 (cit. on p. 37).

[CC+83] Vasek Chvatal, Vaclav Chvatal, et al. *Linear programming*. Macmillan, 1983 (cit. on p. 101).

[CCG09] Alysson M Costa, Jean-François Cordeau, and Bernard Gendron. "Benders, metric and cutset inequalities for multicommodity capacitated network design". In: *Computational Optimization and Applications* 42.3 (2009), pp. 371–392 (cit. on p. 104).

[Cho+12] Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. "The brewing storm in cloud gaming: A measurement study on cloud to end-user latency". In: *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*. IEEE. 2012, pp. 1–6 (cit. on p. 82).

[Chu+15] Cing-Yu Chu, Kang Xi, Min Luo, and H Jonathan Chao. "Congestion-aware single link failure recovery in hybrid SDN networks". In: *Proceedings of IEEE INFOCOM, 2015*. 2015 (cit. on pp. 95, 96).

[CRB09] NM Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. "Virtual network embedding with coordinated node and link mapping". In: *IEEE INFOCOM 2009*. IEEE. 2009, pp. 783–791 (cit. on p. 49).

[CRB12] Mosharaf Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. "Vineyard: Virtual network embedding algorithms with coordinated node and link mapping". In: *IEEE/ACM Transactions on Networking (TON)* 20.1 (2012), pp. 206–219 (cit. on p. 49).

[DDS06] Guy Desaulniers, Jacques Desrosiers, and Marius M Solomon. *Column generation*. Vol. 5. Springer Science & Business Media, 2006 (cit. on p. 102).

[Di +19a] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac. "Mininet on steroids: exploiting the cloud for Mininet performance". In: *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. 2019, pp. 1–3. DOI: 10.1109/CloudNet47604.2019.9064129 (cit. on p. 13).

[Di +19b] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. "Demo Proposal - Distrinet: A Mininet Implementation for the Cloud". In: *Proceedings of the 15th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '19. Orlando, FL, USA: Association for Computing Machinery, 2019, pp. 82–83. ISBN: 9781450370066. DOI: 10.1145/3360468.3368186. URL: https://doi.org/10.1145/3360468.3368186 (cit. on p. 13).

[Di +19c] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. "Mininet on steroids: exploiting the cloud for Mininet performance". In: *IEEE CloudNet*. 2019 (cit. on pp. 31, 45).

[Di +19d] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. "Trust your SDN/NFV experiments with Distrinet". In: *Journées Cloud* (2019) (cit. on p. 13).

[Di +19e] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, Chidung Lac, and Walid Dabbous. *Distributed Network Experiment Emulation*. GEFI 19 - Global Experimentation for Future Internet - Workshop. Nov. 2019. URL: `https://hal.inria.fr/hal-02359801` (cit. on p. 13).

[Di +21a] G. Di Lena, F. Giroire, T. Turletti, and C. Lac. "CloudTrace Demo: Tracing Cloud Network Delay". Submitted to IEEE International Conference on Network Softwarization (NetSoft), Demo session. 2021 (cit. on p. 13).

[Di +21b] G. Di Lena, A. Tomassilli, F. Giroire, D. Saucez, T. Turletti, and C. Lac. "Placement Module for Distributed SDN/NFV Network Emulation". Submitted to Computer Networks Journal. 2021 (cit. on p. 13).

[Di +21c] Giuseppe Di Lena, Andrea Tomassilli, Damien Saucez, Frédéric Giroire, Thierry Turletti, and Chidung Lac. "Distrinet: a Mininet Implementation for the Cloud". In: *ACM Computer Communication Review* (2021) (cit. on p. 13).

[Dis19] Distrinet. *Distrinet website*. `https://distrinet-emu.github.io`. 2019 (cit. on pp. 45, 70).

[Dis20] Distrinet. *Create a personalized host image*. `https://distrinet-emu.github.io/personalize_vhost.html`. 2020 (cit. on p. 39).

[DS14] Irit Dinur and David Steurer. "Analytical Approach to Parallel Repetition". In: *Proceedings ACM STOC 2014*. New York, New York, 2014. ISBN: 978-1-4503-2710-7 (cit. on p. 99).

[DW60] George B Dantzig and Philip Wolfe. "Decomposition principle for linear programs". In: *Operations research* 8.1 (1960), pp. 101–111 (cit. on p. 101).

[EA15] Rodrigo Emiliano and Mário Antunes. "Automatic network configuration in virtualized environment using gns3". In: *2015 10th International Conference on Computer Science & Education (ICCSE)*. IEEE. 2015, pp. 25–30 (cit. on p. 33).

[Fis+13] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann De Meer, and Xavier Hesselbach. "Virtual network embedding: A survey". In: *IEEE Communications Surveys & Tutorials* 15.4 (2013), pp. 1888–1906 (cit. on p. 49).

[FM17] Paulo Fonseca and Edjard Mota. "A survey on fault management in software-defined networks". In: *IEEE Communications Surveys & Tutorials* (2017) (cit. on p. 95).

[FT02] Bernard Fortz and Mikkel Thorup. "Optimizing OSPF/IS-IS weights in a changing world". In: *IEEE journal on selected areas in communications* 20.4 (2002), pp. 756–767 (cit. on p. 111).

[FV00] Andrea Fumagalli and Luca Valcarenghi. "IP restoration vs. WDM protection: Is there an optimal choice?" In: *IEEE network* 14.6 (2000) (cit. on p. 94).

[Gar07] Simson Garfinkel. "An evaluation of Amazon's grid computing services: EC2, S3, and SQS". In: (2007) (cit. on p. 81).

[GBU19] Karyna Gogunska, Chadi Barakat, and Guillaume Urvoy-Keller. "Tuning optimal traffic measurement parameters in virtual networks with machine learning". In: *2019 IEEE 8th International Conference on Cloud Networking, CloudNet 2019, Coimbra, Portugal, November 4-6, 2019.* IEEE, 2019, pp. 1–3. DOI: `10.1109/CloudNet47604.2019.9064132`. URL: `https://doi.org/10.1109/CloudNet47604.2019.9064132` (cit. on p. 80).

[Gir+19] Frédéric Giroire, Nicolas Huin, Andrea Tomassilli, and Stéphane Pérennes. "When Network Matters: Data Center Scheduling with Network Tasks". In: *IEEE International Conference on Computer Communications (INFOCOM).* Paris, France, Apr. 2019, pp. 2278–2286. DOI: `10.1109/INFOCOM.2019.8737415` (cit. on p. 51).

[GJS74] Michael R Garey, David S Johnson, and Larry Stockmeyer. "Some simplified NP-complete problems". In: *Proceedings of the sixth annual ACM symposium on Theory of computing.* ACM. 1974, pp. 47–63 (cit. on p. 51).

[GMS95] Martin Grötschel, Clyde L Monma, and Mechthild Stoer. "Design of survivable networks". In: *Handbooks in operations research and management science* 7 (1995), pp. 617–672 (cit. on p. 95).

[Goo20] Google. *Google Stadia website.* `https://store.google.com/product/stadia`. 2020 (cit. on p. 82).

[Gri20] Grid5000. *Grid5000.* `https://www.grid5000.fr/w/Grid5000:Home`. 2020 (cit. on pp. 46, 56).

[Had20a] Hadoop. *Apache Hadoop documentation.* `http://hadoop.apache.org`. 2020 (cit. on pp. 42, 74).

[Had20b] Hadoop. *Hadoop docs.* `https://hadoop.apache.org/docs/stable/`. 2020 (cit. on p. 80).

[Han+15] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. "Network function virtualization: Challenges and opportunities for innovations". In: *IEEE Communications Magazine* 53.2 (2015), pp. 90–97 (cit. on p. 93).

[Hås01] Johan Håstad. "Some optimal inapproximability results". In: *Journal of the ACM (JACM)* 48.4 (2001), pp. 798–859 (cit. on p. 107).

[HJG17] Nicolas Huin, Brigitte Jaumard, and Frédéric Giroire. "Optimization of Network Service Chain Provisioning". In: *IEEE International Conference on Communications (ICC).* Paris, France, May 2017 (cit. on p. 94).

[HJG18] Nicolas Huin, Brigitte Jaumard, and Frédéric Giroire. "Optimal Network Service Chain Provisioning". In: *IEEE/ACM Transactions on Networking* (2018) (cit. on pp. 99, 102, 111).

[Hma+17] Ali Hmaity, Marco Savi, Francesco Musumeci, Massimo Tornatore, and Achille Pattavina. "Protection strategies for virtual network functions placement and service chains provisioning". In: *Networks* (2017), pp. 1–15 (cit. on p. 94).

[Hou+11] Ines Houidi, Wajdi Louati, Walid Ben Ameur, and Djamal Zeghlache. "Virtual network provisioning across multiple substrate networks". In: *Computer Networks* 55.4 (2011), pp. 1011–1023 (cit. on p. 49).

[Hui+18] Nicolas Huin, Andrea Tomassilli, Frédéric Giroire, and Brigitte Jaumard. "Energy-Efficient Service Function Chain Provisioning". In: *IEEE/OSA Journal of Optical Communications and Networking* 10.2 (2018) (cit. on p. 94).

[HX18] Phuong Ha and Lisong Xu. "Available bandwidth estimation in public clouds". In: *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS).* IEEE. 2018, pp. 238–243 (cit. on p. 81).

[IBM20] IBM. *IBM Cloudbench Documentation.* `https://developer.ibm.com/depmodels/cloud/projects/cloudbench-cbtool/`. 2020 (cit. on p. 80).

[Joh+74] David S. Johnson, Alan Demers, Jeffrey D. Ullman, Michael R Garey, and Ronald L. Graham. "Worst-case performance bounds for simple one-dimensional packing algorithms". In: *SIAM Journal on computing* 3.4 (1974), pp. 299–325 (cit. on p. 54).

[JVU19] Quentin Jacquemart, Alessandro Baldi Vitali, and Guillaume Urvoy-Keller. "Measuring the Amazon Web Services (AWS) WAN Infrastructure". In: *CoRes 2019.* Saint Laurent de la Cabrerisse, France, 2019. URL: `https://hal.archives-ouvertes.fr/hal-02128052` (cit. on p. 80).

[KCG07] Amund Kvalbein, Tarik Cicic, and Stein Gjessing. "Post-failure routing performance with multiple routing configurations". In: *Proceedings of IEEE INFOCOM.* 2007 (cit. on pp. 95, 96).

[KF13] Hyojoon Kim and Nick Feamster. "Improving network management with software defined networking". In: *IEEE Communications Magazine* 51.2 (2013), pp. 114–119 (cit. on p. 94).

[KK98] George Karypis and Vipin Kumar. "Multilevel algorithms for multi-constraint graph partitioning". In: *SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. IEEE. 1998, pp. 28–28 (cit. on p. 47).

[KKV05] Srikanth Kandula, Dina Katabi, and Jean-Philippe Vasseur. "Shrink: A tool for failure diagnosis in IP networks". In: *Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*. ACM. 2005, pp. 173–178 (cit. on pp. 94, 110).

[KL70] Brian W Kernighan and Shen Lin. "An efficient heuristic procedure for partitioning graphs". In: *Bell system technical journal* 49.2 (1970), pp. 291–307 (cit. on p. 51).

[KM05] Hervé Kerivin and A Ridha Mahjoub. "Design of survivable networks: A survey". In: *Networks: An International Journal* 46.1 (2005), pp. 1–21 (cit. on p. 95).

[KNS09] Robert Krauthgamer, Joseph Naor, and Roy Schwartz. "Partitioning graphs into balanced components". In: *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*. SIAM. 2009, pp. 942–949 (cit. on p. 51).

[KS97] Stavros G Kolliopoulos and Clifford Stein. "Improved approximation algorithms for unsplittable flow problems". In: *Proceedings 38th Annual Symposium on Foundations of Computer Science*. IEEE. 1997, pp. 426–436 (cit. on p. 54).

[Kva+06] Amund Kvalbein, Audun Fosselie Hansen, Stein Gjessing, and Olav Lysne. "Fast IP network recovery using multiple routing configurations". In: *Proceedings of IEEE INFOCOM*. 2006 (cit. on pp. 95, 96).

[Len+21] Giuseppe Di Lena, Andrea Tomassilli, Frédéric Giroire, Damien Saucez, Thierry Turletti, and Chidung Lac. "A Right Placement Makes a Happy Emulator: a Placement Module for Distributed SDN/NFV Emulation". Proceedings of IEEE International Conference on Communications (ICC). 2021 (cit. on p. 13).

[LHM10] Bob Lantz, Brandon Heller, and Nick McKeown. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010. DOI: 10.1145/1868447.1868466 (cit. on pp. 31, 33, 119).

[Li+17] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson. "Performance Overhead Comparison between Hypervisor and Container Based Virtualization". In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. 2017, pp. 955–962. DOI: 10.1109/AINA.2017.79 (cit. on p. 80).

[LM15] S.-I. Lee and Myung M.-K. Shin. "A self-recovery scheme for service function chaining". In: *International Conference on Information and Communication Technology Convergence (ICTC)*. 2015, pp. 108–112 (cit. on p. 94).

[Mar+14] Athina Markopoulou, Gianluca Iannaccone, Supratik Bhattacharyya, Chen-Nee Chuah, and Christophe Diot. "Characterization of failures in an IP backbone". In: *Proceedings of IEEE INFOCOM, 2004*. 2014 (cit. on p. 94).

[McK+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74 (cit. on p. 94).

[Med+15] J. Medved, R. Varga, A. Tkacik, and K. Gray. "OpenDaylight: Towards a Model-Driven SDN Controller architecture". In: *Proceedings of IEEE WoWMoM 2014*. 2015 (cit. on p. 120).

[Mel+11] Márcio Melo, Jorge Carapinha, Susana Sargento, Luis Torres, Phuong Nga Tran, Ulrich Killat, and Andreas Timm-Giel. "Virtual network mapping–an optimization problem". In: *Springer MONAMI*. 2011 (cit. on p. 49).

[Mer14] Dirk Merkel. "Docker: lightweight linux containers for consistent development and deployment". In: vol. 2014. 239. 2014, p. 2 (cit. on p. 33).

[met20] metis. *Metis documentation*. `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`. 2020 (cit. on pp. 46, 48).

[Mic20] Microsoft. *Project xCloud website*. `https://www.xbox.com/en-GB/xbox-game-streaming/project-xcloud`. 2020 (cit. on p. 82).

[Min16] Mininet. *Mininet Cluster Edition TCLink discussion*. `https://mailman.stanford.edu/pipermail/mininet-discuss/2016-July/007005.html`. 2016 (cit. on p. 34).

[Mor+14] Igor M. Moraes, Diogo M.F. Mattos, Lyno Henrique G. Ferraz, Miguel Elias M. Campista, Marcelo G. Rubinstein, Luís Henrique M.K. Costa, Marcelo D. de Amorim, Pedro B. Velloso, Otto Carlos M.B. Duarte, and Guy Pujolle. "FITS: A flexible virtual network testbed architecture". In: *Computer Networks* 63 (2014). Special issue on Future Internet Testbeds Part II, pp. 221–237. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.bjp.2014.01.002`. URL: `http://www.sciencedirect.com/science/article/pii/S1389128614000036` (cit. on p. 33).

[MP12] Jeffrey C Mogul and Lucian Popa. "What we talk about when we talk about cloud network performance". In: *ACM SIGCOMM Computer Communication Review* 42.5 (2012), pp. 44–48 (cit. on p. 81).

[MR12] Michael Menzel and Rajiv Ranjan. "CloudGenius: decision support for web server cloud migration". In: *Proceedings of the 21st international conference on World Wide Web*. 2012, pp. 979–988 (cit. on p. 81).

[MVE14] Arun C Murthy, Vinod Kumar Vavilapalli, and Doug Eadline. *Apache Hadoop YARN: moving beyond MapReduce and batch processing with Apache Hadoop 2*. Pearson Education, 2014 (cit. on p. 43).

[Orl+10] Sebastian Orlowski, Roland Wessäly, Michal Pióro, and Artur Tomaszewski. "SNDlib 1.0—Survivable network design library". In: *Networks* 55.3 (2010) (cit. on p. 110).

[ovs20] ovs. *Open vSwitch*. `https://www.openvswitch.org`. 2020 (cit. on p. 33).

[Pal+19] Fabio Palumbo, Giuseppe Aceto, Alessio Botta, Domenico Ciuonzo, Valerio Persico, and Antonio Pescapé. "Characterizing Cloud-to-user Latency as perceived by AWS and Azure Users spread over the Globe". In: *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2019, pp. 1–6 (cit. on p. 81).

[Pap01] Dimitri Papadimitriou. "Inference of shared risk link groups". In: *Internet-draft: draft-many-inference-srlg-02. txt* (2001) (cit. on p. 94).

[Per+15a] Valerio Persico, Pietro Marchetta, Alessio Botta, and Antonio Pescapé. "On network throughput variability in microsoft azure cloud". In: *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2015, pp. 1–6 (cit. on p. 81).

[Per+15b] Valerio Persico, Pietro Marchetta, Alessio Botta, and Antonio Pescapè. "Measuring network throughput in the cloud: The case of Amazon EC2". In: *Computer Networks* 93 (2015), pp. 408–422 (cit. on p. 81).

[Pes+18] Artur Pessoa, Ruslan Sadykov, Eduardo Uchoa, and François Vanderbeck. "Automation and combination of linear-programming based stabilization techniques in column generation". In: *INFORMS Journal on Computing* (2018) (cit. on p. 103).

[PKR16] M. Peuster, H. Karl, and S. van Rossem. "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments". In: *IEEE NFV-SDN*. Nov. 2016, pp. 148–153. DOI: `10.1109/NFV-SDN.2016.7919490` (cit. on p. 33).

[PM04] Michal Pióro and Deep Medhi. *Routing, flow, and capacity design in communication and computer networks*. Elsevier, 2004 (cit. on p. 94).

[Pro19] The Mininet Project. *Cluster Edition Prototype*. `https://github.com/mininet/mininet/wiki/Cluster-Edition-Prototype`. Accessed: 2019-01-02. 2019 (cit. on pp. 33, 45–47).

[Qiu+17] K. Qiu, S. Huang, Q. Xu, J. Zhao, X. Wang, and S. Secci. "ParaCon: A Parallel Control Plane for Scaling Up Path Computation in SDN". In: *IEEE Transactions on Network and Service Management* 14.4 (2017), pp. 978–990. DOI: 10.1109/TNSM.2017.2761777 (cit. on p. 95).

[Qiu+19] K. Qiu, J. Zhao, X. Wang, X. Fu, and S. Secci. "Efficient Recovery Path Computation for Fast Reroute in Large-Scale Software-Defined Networks". In: *IEEE Journal on Selected Areas in Communications* 37.8 (2019), pp. 1755–1768. DOI: 10.1109/JSAC.2019.2927098 (cit. on p. 95).

[QN15] Paul Quinn and Tom Nadeau. "Problem statement for service function chaining". In: (2015) (cit. on p. 94).

[Red20] Redis. *Redis docs*. https://redis.io/topics/cluster-tutorial. 2020 (cit. on p. 80).

[RR14] P Vijaya Vardhan Reddy and Lakshmi Rajamani. "Evaluation of different hypervisors performance in the private cloud with SIGAR framework". In: *International Journal of Advanced Computer Science and Applications* 5.2 (2014) (cit. on p. 80).

[RS18] Matthias Rost and Stefan Schmid. "Charting the Complexity Landscape of Virtual Network Embeddings". In: *Proc. IFIP Networking*. 2018 (cit. on p. 51).

[Sga+13] Andrea Sgambelluri, Alessio Giorgetti, Filippo Cugini, Francesco Paolucci, and Piero Castoldi. "OpenFlow-based segment protection in Ethernet networks". In: *Journal of Optical Communications and Networking* 5.9 (2013) (cit. on pp. 33, 95, 96).

[Sou+17] O. Soualah, Marouen Mechtri, Chaima Ghribi, and Djamal Zeghlache. "A link failure recovery algorithm for Virtual Network Function chaining". In: *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. 2017 (cit. on p. 94).

[ST97] Horst D Simon and Shang-Hua Teng. "How good is recursive bisection?" In: *SIAM Journal on Scientific Computing* 18.5 (1997), pp. 1436–1445 (cit. on p. 51).

[Sto06] Mechthild Stoer. *Design of survivable networks*. Springer, 2006 (cit. on p. 95).

[Suc+11] Martin Suchara, Dahai Xu, Robert Doverspike, David Johnson, and Jennifer Rexford. "Network architecture for joint failure recovery and traffic engineering". In: *Proceedings of ACM SIGMETRICS 2011*. ACM. 2011 (cit. on pp. 95, 96).

[Taj+19] Mohammad M Tajiki, Mohammad Shojafar, Behzad Akbari, Stefano Salsano, Mauro Conti, and Mukesh Singhal. "Joint failure recovery, fault prevention, and energy-efficient resource management for real-time SFC in fog-supported SDN". In: *Computer Networks* 162 (2019), p. 106850 (cit. on p. 96).

[Tir99] Ajay Tirumala. "Iperf: The TCP/UDP bandwidth measurement tool". In: *http://dast. nlanr. net/Projects/Iperf/* (1999) (cit. on p. 71).

[Tom+18] A. Tomassilli, N. Huin, F. Giroire, and B. Jaumard. "Resource Requirements for Reliable Service Function Chaining". In: *2018 IEEE International Conference on Communications (ICC)*. May 2018, pp. 1–7 (cit. on p. 94).

[Tom+19a] A. Tomassilli, G. Di Lena, F. Giroire, I. Tahiri, D. Saucez, S. Perennes, T. Turletti, R. Sadykov, F. Vanderbeck, and C. Lac. "Poster: design of survivable SDN/NFV-enabled networks with bandwidth-optimal failure recovery". In: *2019 IFIP Networking Conference (IFIP Networking)*. 2019, pp. 1–2 (cit. on p. 13).

[Tom+19b] A. Tomassilli, G. D. Lena, F. Giroire, I. Tahiri, D. Saucez, S. Perennes, T. Turletti, R. Sadykov, F. Vanderbeck, and C. Lac. "Bandwidth-optimal Failure Recovery Scheme for Robust Programmable Networks". In: *2019 IEEE 8th International Conference on Cloud Networking (CloudNet)*. 2019, pp. 1–6. DOI: 10.1109/CloudNet47604.2019.9064126 (cit. on p. 13).

[Tom+21] A. Tomassilli, G. Di Lena, F. Giroire, I. Tahiri, D. Saucez, S. Perennes, T. Turletti, R. Sadykov, F. Vanderbeck, and C. Lac. "Design of Robust Programmable Networks with Bandwidth-optimal Failure Recovery Scheme". In: *Computer Networks* (2021). Ed. by Elsevier (cit. on p. 13).

[UTK16] Vojtech Uhlir, Ondrej Tomanek, and Lukas Kencl. "Latency-based benchmarking of cloud service providers". In: *Proceedings of the 9th International Conference on Utility and Cloud Computing*. 2016, pp. 263–268 (cit. on p. 82).

[Vas+18] Balázs Vass, László Németh, Martin Zachariasen, Amaro De Sousa, and János Tapolcai. "Vulnerable regions of networks on sphere". In: *2018 10th International Workshop on Resilient Networks Design and Modeling (RNDM)*. IEEE. 2018, pp. 1–8 (cit. on p. 95).

[Vas+20] Balázs Vass, János Tapolcai, David Hay, Jorik Oostenbrink, and Fernando Kuipers. "How to model and enumerate geographically correlated failure events in communication networks". In: *Guide to Disaster-Resilient Communication Networks*. Springer, 2020, pp. 87–115 (cit. on p. 95).

[Vic+13] Pascale Vicat-Blanc, Brice Goglin, Romaric Guillier, and Sebastien Soudan. *Computing networks: from cluster to cloud computing*. John Wiley & Sons, 2013 (cit. on p. 46).

[VNT20] Balázs Vass, László Németh, and János Tapolcai. "The Earth is nearly flat: Precise and approximate algorithms for detecting vulnerable regions of networks in the plane and on the sphere". In: *Networks* 75.4 (2020), pp. 340–355 (cit. on pp. 95, 96).

[Wat08] Jon Watson. "Virtualbox: bits and bytes masquerading as machines". In: vol. 2008. 166. Belltown Media, 2008, p. 1 (cit. on p. 33).

[Wax88] Bernard M Waxman. "Routing of multipoint connections". In: *IEEE journal on selected areas in communications* 6.9 (1988), pp. 1617–1622 (cit. on p. 110).

[Wet+14] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl. "MaxiNet: Distributed emulation of software-defined networks". In: *2014 IFIP Networking Conference*. June 2014, pp. 1–9. DOI: 10.1109/IFIPNetworking.2014.6857078 (cit. on pp. 33, 45).

[Xen21] Xen. *Xen Project*. https://xenproject.org/developers/teams/xen-hypervisor/. 2021 (cit. on p. 33).

[Ye+16] Zilong Ye, Xiaojun Cao, Jianping Wang, Hongfang Yu, and Chunming Qiao. "Joint topology design and mapping of service function chains for efficient, scalable, and reliable network functions virtualization". In: *IEEE Network* 30.3 (2016) (cit. on p. 94).

[Yu+08] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. "Rethinking virtual network embedding: substrate support for path splitting and migration". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 17–29 (cit. on pp. 49, 54).

[Zha+18] Yuan Zhang, Lin Cui, Wei Wang, and Yuxiang Zhang. "A Survey on Software Defined Networking with Multiple Controllers". In: *J. Netw. Comput. Appl.* 103.C (Feb. 2018), pp. 101–118. ISSN: 1084-8045. DOI: 10.1016/j.jnca.2017.11.015 (cit. on p. 124).

[Zha+19] T. Zhang, L. Linguaglossa, J. Roberts, L. Iannone, M. Gallo, and P. Giaccone. "A benchmarking methodology for evaluating software switch performance for NFV". In: *2019 IEEE Conference on Network Softwarization (NetSoft)*. 2019, pp. 251–253. DOI: 10.1109/NETSOFT.2019.8806695 (cit. on p. 33).