Resilient Virtualized Network Functions for Data Centers and Decentralized Environments

Ghada Moualla

UNIVERSITÉ CÔTE D'AZUR

ÉCOLE DOCTORALE
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE DE DOCTORAT

Virtualisation Résiliente des Fonctions Réseau pour les Centres de Données et les Environnements Décentralisés

Ghada MOUALLA

Centre de recherche: INRIA (EPI DIANA)

Présentée en vue de l'obtention du grade de docteur en INFORMATIQUE
d'Université Côte d'Azur
Dirigée par: Thierry TURLETTI
Co-dirigée par: Damien SAUCEZ
Soutenue le: 23 September 2019

Devant le jury, composé de:
Guillaume Urvoy-Keller, Professeur, Université Cote'Azur
Hossam AFIFI, Professeur, Télécom SudParis
Benoît PARREIN, MC (HDR), Université de Nantes
Yassine HADJADJ-AOUL, MC, Université Rennes 1, IRISA

Inria
INVENTEURS DU MONDE NUMÉRIQUE

Acknowledgements

This thesis summarizes my research results during my Ph.D. study at the research team DIANA, Inria, France from 2015 to 2019.

I would like, firstly, to express my gratitude to my supervisors Dr. Thierry Turletti and Dr. Damien Saucez for their continued support during the whole Ph.D. period. They gave me the best guidance, help, and support to successfully go ahead in the Ph.D. and their support and advice on both research as well as on my career have been priceless. It was my great pleasure to be their student.

Also, all the thanks to the jury members for their presence, participation, and the time that they devoted to read my thesis. Their valuable comments and questions helped me to improve the quality of my dissertation.

I am extremely grateful to the members of DIANA team for their support and interesting discussion on the wide-range of topics, which helped me to broaden my perspectives and provide the perfect work environment. I specifically thank Dr. Walid Dabbous, the leader of our team, for all his support and guidance for all team members and to Christine Foggia, our team assistant, who helped me out with many administrative problems and contributed greatly to the smooth running of my Ph.D.

Last but not least, words cannot express how grateful I am to my mother and my father for all of the sacrifices that they have made on my behalf. Their prayer for me was what sustained me thus far. At the end, I would like to express appreciation to my beloved husband, who was always my support in the moments when there was no one to answer my queries. To my children, my little angels who increase my inspiration to achieve greatness. They have made me stronger and better.

Virtualisation Résiliente des Fonctions Réseau pour les Centres de Données et les Environnements Décentralisés

Résumé:

Les réseaux traditionnels reposent sur un grand nombre de fonctions réseaux très hétérogènes qui s'exécutent sur du matériel propriétaire déployé dans des boîtiers dédiés. Concevoir ces dispositifs spécifiques et les déployer est complexe, long et coûteux. De plus, comme les besoins des clients sont de plus en plus importants et hétérogènes, les fournisseurs de services sont contraints d'étendre ou de moderniser leur infrastructure régulièrement, ce qui augmente fortement les coûts d'investissement ($CAPEX$) et de maintenance ($OPEX$) de l'infrastructure. Ce paradigme traditionnel provoque une ossification du réseau et rend aussi plus complexe la gestion et la fourniture des fonctions réseau pour traiter les nouveaux cas d'utilisation.

La virtualisation des fonctions réseau (NFV) est une solution prometteuse pour relever de tels défis en dissociant les fonctions réseau du matériel sous-jacent et en les implémentant en logiciel avec des fonctions réseau virtuelles (VNFs) capables de fonctionner avec du matériel non spécifique peu coûteux. Ces VNFs peuvent être organisés et chaînés dans un ordre prédéfini, formant des chaînes de Services (SFC) afin de fournir des services de bout-en-bout aux clients. Cependant, même si l'approche NFV comporte de nombreux avantages, il reste à résoudre des problèmes difficiles comme le placement des fonctions réseau demandées par les utilisateurs sur le réseau physique de manière à offrir le même niveau de résilience que si une infrastructure dédiée était utilisée, les machines standards étant moins fiables que les dispositifs réseau spécifiques. Ce problème devient encore plus difficile lorsque les demandes de service nécessitent des décisions de placement à la volée.

Face à ces nouveaux défis, nous proposons de nouvelles solutions pour résoudre le problème du placement à la volée des VNFs tout en assurant la résilience des services instanciés face aux pannes physiques pouvant se produire dans différentes topologies de centres de données (DC). Bien qu'il existe des solutions de récupération, celles-ci nécessitent du temps pendant lequel les services affectés restent indisponibles. D'un autre côté, les décisions de placement intelligentes peuvent épargner le besoin de réagir aux pannes pouvant se produire dans les centres de données. Pour pallier ce problème, nous proposons tout d'abord une étude approfondie de la manière dont les choix de placement peuvent affecter la robustesse globale des services placés dans un centre de données. Sur la base de cette étude, nous proposons une solution déterministe applicable lorsque le fournisseur de services a une connaissance et un contrôle complets de l'infrastructure. Puis, nous passons de cette solution déterministe à une approche stochastique dans le cas où les SFCs sont demandées par des clients indépendamment du réseau physique du DC, où les utilisateurs n'ont qu'à fournir les SFC qu'ils veulent placer et le niveau de robustesse requis (e.g., les 5 neufs). Nous avons développé plusieurs algorithmes et les avons évaluées. Les résultats de nos simulations montrent l'efficacité de nos algorithmes et la faisabilité de nos propositions dans des topologies de centres de données à très grande échelle, ce qui rend leur utilisation possible dans un environnement de production.

Toutes ces solutions proposées fonctionnent de manière efficace dans un environnement de confiance, comme les centres de données, avec la présence d'une autorité centrale qui contrôle toute l'infrastructure. Cependant, elles ne s'appliquent pas à des scénarios décentralisés comme c'est le cas lorsque différentes entreprises ont besoin de collaborer pour exécuter les applications de leurs clients. Nous étudions cette problématique dans le cadre des applications MapReduce exécutées en présence de nœuds byzantins et de nœuds rationnels et en l'absence de tiers de confiance. Nous proposons un des premiers frameworks MapReduce qui soit adapté à ce type d'environnement et nos simulations montrent que l'intégrité des calculs est assuré avec un coût linéaire en fonction du nombre d'attaquants byzantins.

Mots-clés: Centre de Données, MapReduce, NFV, Réseaux Pair-à-Pair, Résilience, SFC, VNF

Resilient Virtualized Network Functions for Data Centers and Decentralized Environments

Abstract:

Traditional networks are based on an ever-growing variety of network functions that run on proprietary hardware devices called middleboxes. Designing these vendor-specific appliances and deploying them is very complex, costly and time-consuming. Moreover, with the ever-increasing and heterogeneous short-term services requirements, service providers have to scale up their physical infrastructure periodically, which results in high CAPEX and OPEX. This traditional paradigm leads to network ossification and high complexity in network management and services provisioning to address emerging use cases.

Network Function Virtualization (NFV) has attracted notable attention as a promising paradigm to tackle such challenges by decoupling network functions from the underlying proprietary hardware and implementing them as software, named Virtual Network Functions (*VNFs*), able to work on inexpensive commodity hardware. These VNFs can be arranged and chained together in a predefined order, the so-called Service Function chaining (*SFC*), to provide end-to-end services.

Despite all the benefits associated with the new paradigm, NFV comes with the challenge of how to place the functions of the users' requested services within the physical network while providing the same resiliency as if a dedicated infrastructure were used, given that commodity hardware is less reliable than the dedicated one. This problem becomes particularly challenging when service requests have to be fulfilled as soon as they arise (i.e., in an online manner).

In light of these new challenges, we propose new solutions to tackle the problem of online SFC placement while ensuring the robustness of the placed services against physical failures in data-center (*DC*) topologies. Although recovery solutions exist, they still require time in which the impacted services will be unavailable while taking smart placement decisions can help in avoiding the need for reacting against simple network failures.

First, we provide a comprehensive study on how the placement choices can affect the overall robustness of the placed services. Based on this study we propose a deterministic solution applicable when the service provider has a full knowledge and control on the infrastructure.

Thereafter, we move from this deterministic solution to a stochastic approach for the case where SFCs are requested by tenants oblivious to the physical DC network, where users only have to provide the SFC they want to place and the required availability level (e.g., 5 nines). We simulated several solutions and the evaluation results show the effectiveness of our algorithms and the feasibility of our propositions in very large scale data center topologies, which make it possible to use them in a productive environment.

All these solutions work well in trusted environments with a central authority that controls the infrastructure. However, in some cases, many enterprises need to collaborate together in order to run tenants' application, e.g., MapReduce applications. In such a scenario, we move to a completely untrusted decentralized environment with no trust guarantees in the presence of not only byzantine nodes but also rational nodes. We considered the case of MapReduce applications in such an environment and present an adapted MapReduce framework called MARS, which is able to work correctly in such a context without the need of any trusted third party. Our simulations show that MARS grants the execution integrity in MapReduce linearly with the number of byzantine nodes in the system.

Keywords: VNF, SFC, Data Centers, Robustness, Availability, MapReduce, P2P

Contents

List of Figures

List of Tables

List of Abbreviations

CAPEX	**Cap**ital **Ex**penditure
CDNs	**C**ontent **D**elivery **N**etworks
COTS	**C**ommercial **O**ff-**T**he-**S**helf
CPLPSS	**C**apacitated **P**lant **L**ocation **P**roblem with **S**ingle **S**ource
DC	**D**ata **C**enter
DLT	**D**istributed **L**edger **T**echnology
DRR	**D**ynamic **R**ound **R**obin
DS	**D**istributed **S**cheduling
ETSI	**E**uropean **T**elecommunications **S**tandards **I**nstitute
FFD	**F**irst **F**it **D**ecreasing
GA	**G**enetic **A**lgorithm
HA	**H**igh **A**vailability
ICN	**I**nformation **C**entric *N*etworking
IDS	**I**ntrusion **D**etection **S**ystems
IETF	**I**nternet **E**ngineering **T**ask **F**orce
ILP	**I**nteger **L**inear **P**rogram
IMS	**IP M**ultimedia **S**ubsystems
IoT	**I**nternet **o**f **T**hings
ISP	**I**nternet **S**ervice **P**rovider
IT	**I**nformation **T**echnology
MADs	**M**ultiple **A**dministrative **D**omains
MANO	**M**anagement and **O**rchestration
Mbps	**M**ega**b**it **p**er **s**econd
MILP	**M**ixed **I**nteger **L**inear **P**roblem
MTBF	**M**ean **T**ime **B**etween **F**ailures
MTTR	**M**ean **T**ime **T**o **R**epair
NF	**N**etwork **F**unction
NFV	**N**etwork **F**unction **V**irtualization
NFVI	**N**etwork **F**unction **V**irtualization **I**nfrastructure
NFV-RA	**NFV R**esource **A**llocation
OPEX	**Op**erational **Ex**penditure

OSPF	**O**pen **S**hortest **P**ath **F**irst
PM	**P**hysical **M**achine
POM	**P**roof **O**f **M**isbehavior
PRNG	**P**seudo **R**andom **N**umber **G**enerator
P2P	**P**eer **t**o **P**eer
RO	**R**obust **O**ptimization
SDN	**S**oftware **D**efined **N**etworking
SFC	**S**ervice **F**unction **C**hain
SFF	**S**ervice **F**unction **F**orwarder
SFP	**S**ervice **F**unction **P**ath
SGA	**S**imple **G**reedy **A**pproach
SID	**S**ubgraph **I**somorphism **D**etection
SLA	**S**ervice **L**evel **A**greement
ToR	**T**op **o**f **R**ack
TSP	**T**elecom **S**ervice **P**rovider
VM	**V**irtual **M**achine
VNE	**V**irtual **N**etwork **E**mbedding
VNER	**V**irtual **N**etwork **E**mbedding **R**equest
VNF	**V**irtual **N**etwork **F**unction
VNF-FG	**VNF F**orwarding **G**raph

1 Introduction

Contents

1.1 Motivation and Challenges

Traditionally, telecommunication services are based on network operators having physical proprietary appliances and equipment, namely *Middleboxes*, in order to bring new services into networks. These middleboxes, or what we called Network Functions *NFs*, perform a complex and varied set of functions ranging from security purposes (e.g., Firewalls and Intrusion Detection Systems *IDS*) to performance purposes (such as caches, proxies, and accelerators) [120].

However, network users increasingly ask for more varied and new short-term services, which means that operators must correspondingly and continually purchase, store and operate new and expensive hardware-based middleboxes. This does not only require high and rapidly changing technical skills to run and manage these devices and manually chain them to ensure the desired network service, but also leads to high capital expenditures for operators to buy, deploy and maintain this new equipment. Moreover, these ubiquitous middleboxes cannot easily be scaled up and down with changing demands, which leads to waste. Therefore, there is a real need for service providers to find alternative ways of building dynamic infrastructures so they can deliver new and innovative services to their tenants in short time [9, 188].

Network Function Virtualization (*NFV*) [125] has been proposed as a prominent way to alleviate these difficulties by leveraging the blooming virtualization technology [35] to shift middleboxes processing from hardware appliances to software running on low-cost commodity hardware (e.g. servers, switches and storage) located in data centers, network nodes and in end-user premises (as shown in Figure 1.1).

Virtualization technology separates the physical network devices from the network functions running on them, such that virtual network functions (*VNFs*) could be implemented through software and run on standard physical hardware. The VNFs may then be relocated and instantiated in different network locations, whenever needed, without the need of the purchase and deployment of new dedicated hardware. Hence,

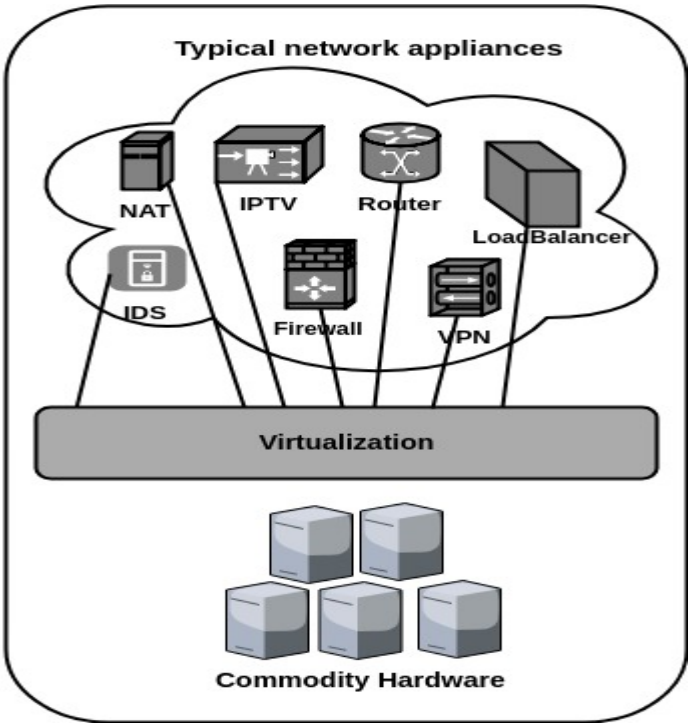


Figure 1.1: From dedicated hardware-based appliances for network services to software-based NFV solutions.

NFV gives the service providers the flexibility and elasticity when providing new services based on customers' needs, while reducing the time to market and decreasing the Capital Expenditure (*CAPEX*) and the Operational Expenses (*OPEX*).

As it is mentioned before, network functions are normally chained together in order to deliver the desired overall functionality or service. Similarly, VNFs could be arranged and chained together in a predefined order in what we called Service Function chaining (*SFC*) to provide end-to-end services [107]. Moreover, several industrial groups are developing standards for Service function chaining. For example, the Internet Engineering Task Force (*IETF*) has developed a service function chaining architecture [78] to determine how network flow classification could be used to route traffic between service functions. Further, the European Telecommunications Standards Institute (*ETSI*) [56] has proposed a service architecture that uses network forwarding graphs to forward traffic between service virtual network functions, called VNF Forwarding Graph (*VNF-FG*).

Despite all the promising advantages brought by NFV, there are still many essential challenges that need to be tackled in NFV-based SFC with more attention. For instance, virtualization may result in variable latency and unstable throughput even when the underlying infrastructure is under-provisioned [80,177]. Thus, maintaining network performance as good as with dedicated hardware will be one of the main difficulties in realizing NFV-based services. Furthermore, another major problem is to achieve fast and efficient resource allocation for VNFs. This problem is referred to as NFV resource allocation (*NFV-RA*) or SFC placement [17].

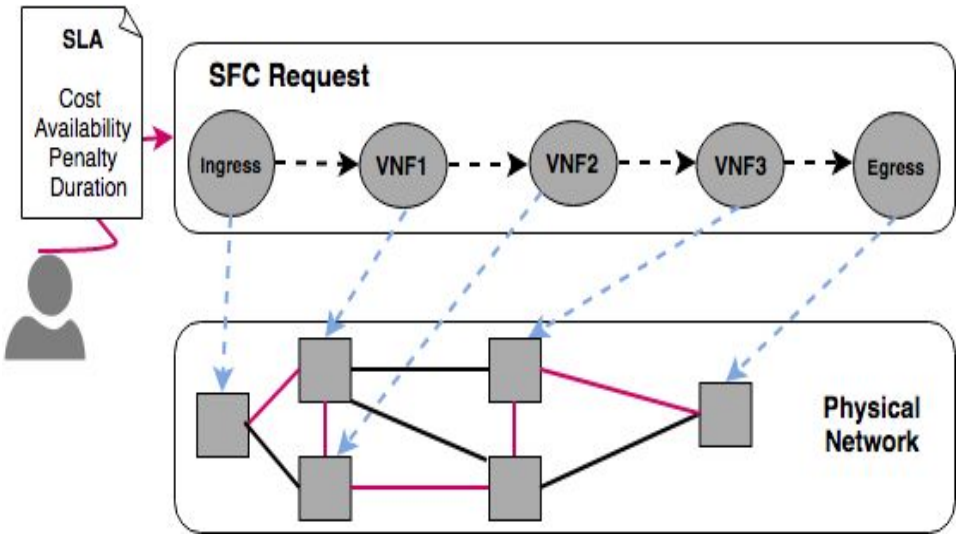


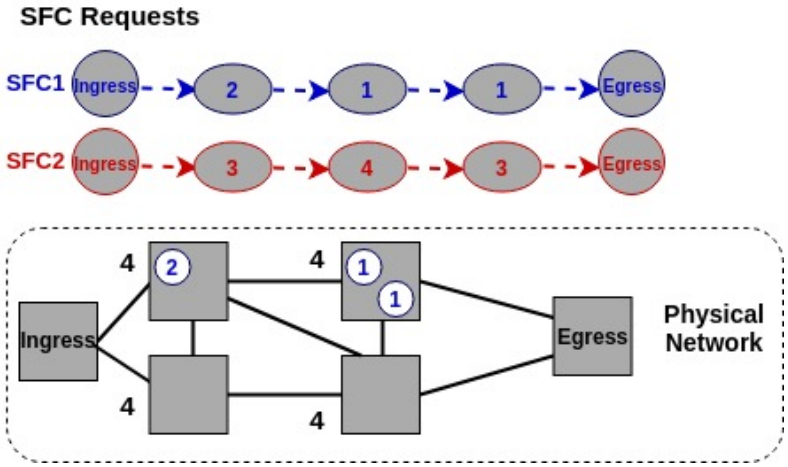
Figure 1.2: SFC Placement Problem.

SFC placement addresses the mapping of the service chains by finding the best locations and hosts for their VNFs. Then it steers the traffic across the placed network functions while respecting user requirements and maximizing provider revenue without violating the required Service Level Agreement (*SLA*) constraints (See Figure 1.2). The service provider's SFC placement decision can have a crucial effect on the service performance guarantees provided to the customers.

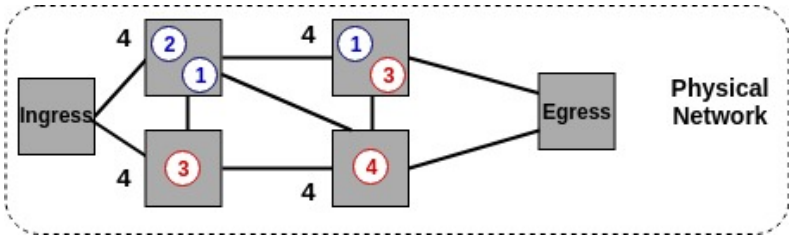
This placement is more challenging when a service chain requires online embedding upon its arrival without relying on future information [111, 174]. In this scenario, SFC requests arrive and leave at a certain probability, i.e., following some distribution functions, and cannot be accurately predicted, and the number of SFC requests needed to be processed cannot be obtained in advance. The main challenge in designing an effective online algorithm lies in the unknown nature of future resources requirements for each arrived service chain, where the service providers have to make the deployment decisions for the current SFC on the fly, while keeping the possibility to accept more tenants service requests in the future in order to increase their revenue. Therefore, we cannot place online SFC requests as easily as in the case of offline requests.

Figure 1.3 shows this problem with two service requests, namely SFC1 and SFC2, arrive in an online manner (i.e. the SFC1 arrives first and should be treated upon its arrival and then SFC2 arrives after some time). When the SFC1 request arrives to perform load balancing, the service provider could place this service as in the Figure 1.3a. Thus, when the SFC2 comes, there is no possibility for the service provider to place it. However, if the service provider had prior knowledge on the future request (i.e. has information about SFC1 and SFC2 resources requirements), he would be able to accept both services as shown in Figure 1.3b.

Moreover, VNF-based SFC brings more concerns about the service provided resiliency, which is defined as the ability of providing and maintaining an acceptable level of service against network failures [167]. Ensuring such resiliency, especially for



(a) Online Scenario



(b) Offline Scenario

Figure 1.3: SFC Request Placement with Online and Offline case, where the numbers refer to the required/available number of CPU cores

critical services with stringent requirements (e.g. e-health service [108], connected vehicles [83], or emergency calls), is difficult due to different reasons such as (*i*) the commodity hardware that host VNFs are more prone to errors and failures compared with dedicated hardware [79] and (*ii*) the software implementations of the SFCs' functions can be rather buggy and vulnerable to failures. Thus, NFV needs to build resiliency into software when moving to error-prone hardware platforms.

In light of these new challenges, we proposed new solutions to tackle the problem of online SFC placement while ensuring the robustness of the placed services in datacenter *DC* topologies. Even if many works had been done in the literature on VM placement, Virtual Network Embedding (*VNE*) and lately VNF placement, most of them aim to address one part of these challenges and a small number of them considers the resilient placement as they proposed solution for different optimization objective (such as energy saving, minimization latency and etc.). Ensuring robustness was left aside as they are many provided solutions in case of failure [92] (i.e. failure detection and recovery mechanisms). Although recovery solutions exist, they still take time in which the impacted services will be unavailable. Moreover, making smart placement decisions can help in avoiding the need to react against simple network failures.

In our work, we first start with a study on how the placement choices can affect the overall robustness. In this study, we advocate that accounting for robustness while

placing functions can significantly improve robustness of the chain without increasing the load of the VNF orchestrator which is a key component of the NFV architectural framework that performs service orchestration, as well as other functions (more details are provided in Chapter 2).

Besides on the new insights we got for the first study, we propose a new online algorithm that builds active-active chain replicas to deploy tenants SFC requests in a data-center topology (more details about replication mechanisms are available in Sec. 2.3.2). In this scenario, the service provider has full knowledge and control on this trusted environment. More precisely, we provide the formal model and solve it to get the optimal solution for each request upon its arrival. Then we propose a greedy solution to compare with the optimal solution results to understand the impact on the results.

Thereafter, we move from this deterministic solution where SFCs are directly deployed by the DC owner to a stochastic approach for the case where SFCs are requested by tenants oblivious to the physical DC network and that only have to provide the SFC they want to place and the required availability level. To evaluate our proposed algorithms, we implement discrete event simulators using Python. Python is used because it is simple and it supports simulation networking and plotting libraries (e.g., NetworkX [76] and matplotlib [84]). Our simulations are performed on a grid platform called Grid5000 [26].

Finally, we move to the decentralized environment where many contributors try to run tenants' applications, namely MapReduce applications, on their peer-to-peer network infrastructure. Under this condition, where we have no knowledge or control over this environment, trust and heterogeneity challenges arise. Thus, we have to work at the application level to answer the question of how to ensure the correctness of the result and the liveness property with the presence of byzantine nodes (i.e. nodes that can have an arbitrary behaviour where it could crash or even send incorrect messages) and rational partners that have self-interested behavior. Thus, we introduce a blockchain-based decentralized MapReduce framework eligible to work within untrusted peer-to-peer environments. To evaluate our proposal, we have adapted the MRSG MapReduce simulator [93] that is built on top of SimGrid [30].

1.2 Thesis Outline

In the following, we summarize the content of each chapter and the obtained results. In Chapter 2, we present the basic background used in this thesis. This chapter includes (*i*) Network Function Virtualization where we give details about its framework, benefits, and the raised challenges, (*ii*) Service function chaining architecture and SFC request models, and (*iii*) Availability notion with the service-level agreement and how it is calculated in a network system.

In Chapter 3, we present the state of the art related to the placement problem with different objectives and the works related to availability and robustness issues. Replication strategies and offline/online placement used in literature are also presented here. Moreover, the related work for Hadoop MapReduce framework and

BAR system model protocols are shown and compared to our proposed distributed MapReduce framework.

Chapter 4 studies the need for considering the robustness issue while taking the placement decisions for SFCs requested by tenants. In this study, we show that the placement choices can significantly improve robustness of the chain without any need for failure recovery solutions or having a service downtime. To that aim, we study a reference chain and demonstrate with an exhaustive study how its placement in the physical infrastructure can influence the overall robustness even when the virtual chain itself is supposed to be robust to failures.

In Chapter 5, taking into account the main points obtained from the previous study, we present an online two-step algorithm that determines the optimal number of VNF service instances (active-active replication mechanism) and their placement in the data-center network to guarantee the required robustness against R fail-stop node failures. This solution uses the k-resilient property per SFC request to place the requested SFC taking into account the available network resources and the resource requirements of the tenants' service requests, namely CPU and bandwidth. Three different data center topologies are used to evaluate the proposed solution. This deterministic solution is applicable when SFCs requests are directly deployed by the data center owner who has awareness about the network infrastructure.

Chapter 6 provides a stochastic approach for the case where SFCs are requested by tenants oblivious to the physical DC network and that only have to provide the SFC they want to place and the required availability. We present our iterative algorithm where an optimal solution with the objective of maximizing the obtained availability in each iteration until getting the tenant's requested availability. A large data center topology is used as a referenced topology in this chapter.

In Chapter 7, we consider the case when tenants request services for processing big data sets using MapReduce framework. Generally, the solutions that rely on the execution of MapReduce applications on data centers are inherently centralized where processing data is massively distributed but tasks scheduling is controlled by elected nodes (called master nodes). All nodes in this centralized system are assumed to be trustworthy and reliable.

However, in a cooperative scenario (represented in the Figure 1.4) when different partners or enterprises collaborate together to provide cooperative services based on peer-to-peer network systems, we move to a completely decentralized environment that is subject to new types of failures, namely byzantine failures and rational actors. In such a decentralized environment, there are no trust guarantees to the computation results as there is no central authority that controls the actions of all nodes, and the resources are not homogeneous anymore. Moreover, applications used in DC have been developed for trusted homogeneous environments and so the algorithms and protocols used in such applications are not adapted to untrusted heterogeneous environments [104], which implies that only changing at the network level is not sufficient.

Hence, there is a need to move to the application level where we should work on the workload itself and adapt the application to be able to work correctly under the

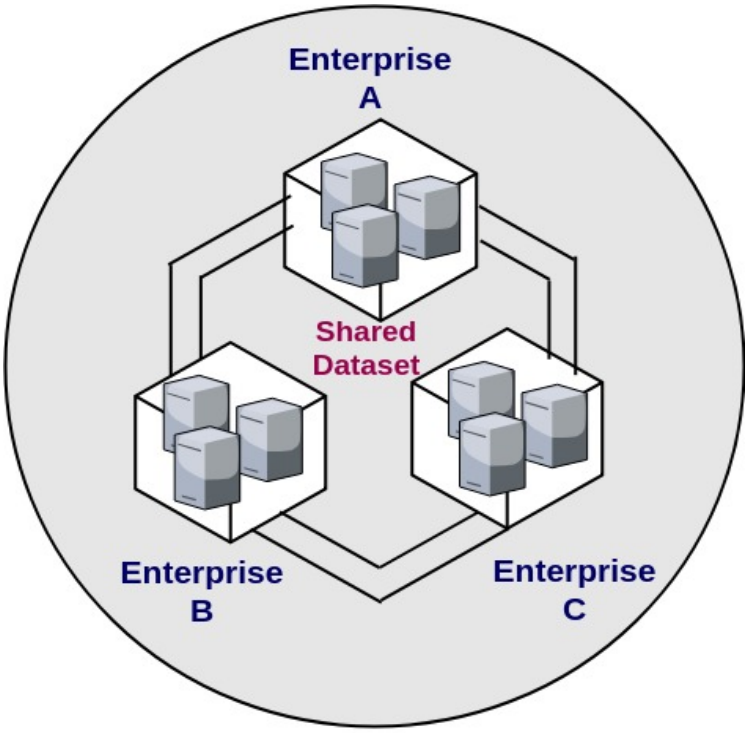


Figure 1.4: Collaboration between different enterprises.

existence of both byzantine and rational nodes in the system (called BAR system, more details are available in Sec 3.2) before solving the placement problem for data as where we place data affects the application performance. In Chapter 7, we present a new MapReduce framework, named MARS, conceived to be robust against byzantine and rational actors in untrusted peer-to-peer environments. This is an early work that has been done in collaboration with M. Alberto Zirondelli in the context of his Master research internship.

Finally, in Chapter 8, we summarize the content of the thesis and discuss potential research directions.

1.3 Thesis Contributions

In this Section we briefly resume the major contributions of this thesis as follows:

- we provide a comprehensive study on the necessity of considering the resiliency while taking the placement decisions in the Chapter 4;
- we formalize the placement problem with the objective of ensuring robustness of the placed service functions against R node failures in the trusted environment in Chapter 5. We also present a new online placement algorithm for this problem. Moreover, our proposed solution was implemented and evaluated on large data center topologies (with up to 30,000 physical nodes) to emulate real

data-center topologies. Moreover, a comparison with a greedy solution, namely FFD, is provided to see how this choice could affect the results;

- a stochastic approach for the case in which SFCs are requested by tenants unaware of the physical infrastructure is considered in Chapter 6. For this case, an online availability-aware algorithm is proposed and evaluated on a 48-Fat-Tree data center topology;
- when tenants ask for running big data application, namely MapReduce applications, in a cooperative environment, new challenges arise as this unreliable environment. Thus, we present a new decentralized MapReduce framework to work in Peer-to-Peer networks, called MARS, which is robust against byzantine and rational actors in the Chapter 7. We also extended SimGrid simulator to support MARS.

1.4 Publications

The complete list of my publications is the following:

- Ghada Moualla, Thierry Turletti, Mathieu Bouet, and Damien Saucez. "On the Necessity of Accounting for Resiliency in SFC." In 2016 28th International Teletraffic Congress (ITC 28), vol. 2, pp. 13-15. IEEE, 2016. [131]
- Ghada Moualla, Thierry Turletti, and Damien Saucez. "An Availability-aware SFC placement Algorithm for Fat-Tree Data Centers." In 2018 IEEE 7th International Conference on Cloud Networking (CloudNet), pp. 1-4. IEEE, 2018. [132]
- Ghada Moualla, Thierry Turletti, and Damien Saucez. "Online Robust Placement of Service Chains for Large Data Center Topologies". In 2019 IEEE Access journal,7, pp.60150-60162, IEEE, 2019. [133]

2 Background

Contents

In this chapter, we introduce fundamental concepts that underpin the work presented in this thesis. We begin by presenting some background on NFV architectural framework and its main benefits, use cases and challenges in Sec. 2.1. Then we give some details related to service function chaining architecture and its request models.

Thereafter, Sec. 2.2 gives some background on SFC placement strategies, followed by Sec. 2.3 that covers the availability notion and how it is computed and insured is provided. Finally, we provide a general background relevant to Big Data processing and MapReduce framework in Sec. 2.5.

2.1 Network Function Virtualization

Network Function Virtualization (*NFV*) [56] transforms the way network operators and providers design, manage and deploy their network infrastructure by exploiting

the evolution of virtualization technologies. It enhances the delivery of network services to end users while reducing CAPEX and OPEX.

Previously, network functions, called *middleboxes* (such as firewalls, Network Address Translators *NAT*, Load Balancers *LBs*, etc.), were deployed on vendor-specific hardware and software, which incurs a heavy cost, huge production delays and refrains innovations to deploy new network services. The NFV paradigm has been proposed to decouple the hardware from its software and advocate the use of standard Commercial off-the-shelf (*COTS*) hardware located in the network [34]. Thus, new network services can be deployed rapidly, on an on-demand basis, providing benefits for both end users and network providers.

Moreover, NFV enables configuring hybrid scenarios where functions running on virtualized resources can co-exist with those running on physical resources [125]. NFV relies on traditional virtualization techniques. A virtualized network function may consist of one or more virtual machines running different software, on top of standard servers, switches and storage devices, instead of using dedicated hardware devices for each network function.

How and where to place network functions and how the traffic is routed through these functions are key challenges towards the deployment of NFV. Indeed, NFV is a fascinating use case and new rules placement solutions must be designed to deploy NFV.

Although NFV is a promising solution for service providers, it faces certain challenges that could degrade its performance and hinder its implementation in the telecommunications industry. Here we provide a short background on NFV including relevant aspects such as its architectural framework, its main benefits and the essential challenges involved.

2.1.1 NFV Framework

Virtualization technologies are at the core of the NFV paradigm [130]. Indeed, the success of virtualization technologies in IT operations in enterprises and for computing services (such as Amazon Elastic Compute Cloud *AWS EC2*) has motivated network operators to separate hardware and software using network functions.

To achieve the aforementioned objectives promised by NFV, each VNF should run on a framework that includes dynamic initiation and orchestration of VNF instances. The ETSI NFV Industry Standard Group (*ISG*) [56], which was initially founded by seven global network operators to promote the network function virtualization concept in late 2012, has defined the NFV architectural framework at the functional level using functional entities and reference points, without proposing a specific implementation. The functional entities of the architectural framework and the reference points are (See Figure 2.1):

- Virtualized Network Function: it is the software implementation of a network function that is deployed on virtual resources such as a virtual machine (*VM*). A single VNF may be composed of multiple internal components, thus it could be deployed over multiple VMs.

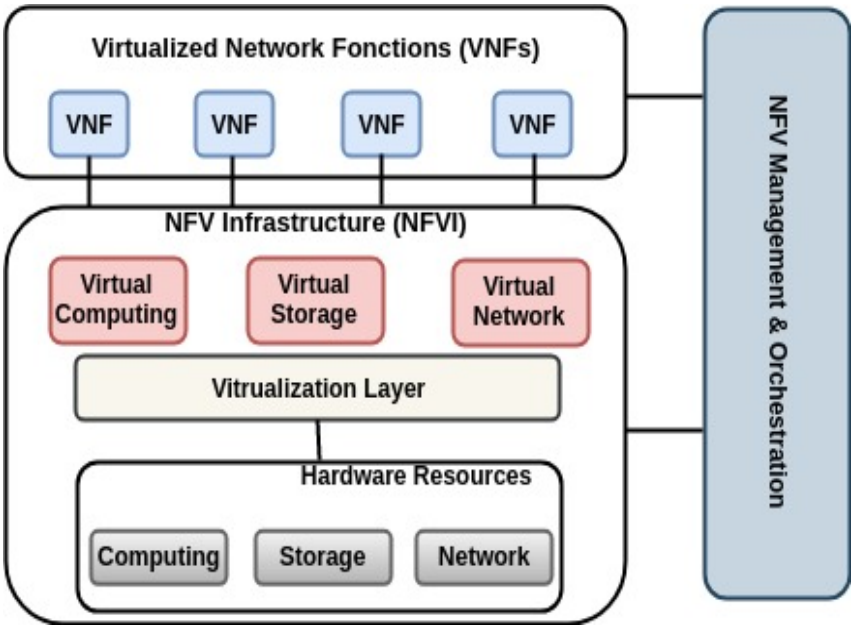


Figure 2.1: NFV reference architectural framework.

- NFV infrastructure (*NFVI*): it includes all hardware and software resources that make up the environment in which VNFs are deployed. The hardware resources typically include computing, storage and network resources providing processing, storage, and connectivity for VNFs through the virtualization layer (based on a hypervisor) that sits just above the hardware and abstracts the physical resources.
- NFV Management and Orchestration (*MANO*): it includes the Orchestrator, the virtual network function manager (*VNFM*), and the virtualized infrastructure manager (*VIM*) functional blocks. These blocks are responsible for the orchestration and lifecycle management of both physical and software resources that support the infrastructure virtualization, and the lifecycle management of VNFs, respectively.

2.1.2 Use case Scenarios of NFV

The NFV paradigm opens up many opportunities to transform network architecture and services. Hence, use cases for NFV are not limited to only existing network services, instead, NFV can deploy new network services, which were not previously feasible because of the high cost, the complex integration of technologies and the incompatible deployment environments.

NFV ISG selected a set of relevant use case scenarios [57], such as: (*i*) Network Functions Virtualization as a Service: where service providers can provide NFV infrastructure, platform and even a single VNF instance as a service, (*ii*) Virtualization of Mobile Core Network and IP Multimedia Subsystems (*IMS*): the mobile networks

and the IMS are used to be populated with a large variety of expensive and proprietary hardware appliances [142]. Thus, their costs and complexity can be reduced by exploiting NFV, (*iii*) Virtualization of Mobile base station: mobile operators can also tremendously benefit from NFV approach to reduce costs as well as improve and provide better service to their customers [46], (*iv*) Virtualization of Content Delivery Networks (*CDNs*): CDNs normally use cache nodes to increase the quality of multimedia services, but with lots of drawbacks where NFV can be used to mitigate them [155].

Moreover, many works try to exploit the NFV paradigm with other use cases such as in vehicular ad hoc networks (*VANETs*) to get rid of many VANETs bottlenecks [195].

2.1.3 Benefits of NFV

The NFV paradigm promises several benefits. Reducing cost is a top consideration for any operator or service provider these days. This aim is achieved with NFV via migration to software running on commodity hardware instead of using expensive dedicated network equipment.

Moreover, NFV provides rapid service innovation and flexibility, allowing service providers to launch, improve, and incrementally optimize their services using software updates. NFV can also leverage power management features of standard servers, switches and storage devices using workload optimization to reduce the energy consumption of the infrastructure by turning off the unused hardware devices. Furthermore, NFV is a scalable approach as it gives the operators the ability to scale their network architecture across multiple servers to adapt quickly to the changing needs of their customers.

2.1.4 Challenges for NFV

Despite all benefits provided by the NFV technology, this new environment arises certain challenges that could degrade its performance and hinder its implementation in the telecommunications industry.

One of the main challenges that NFV faces is the deployment of requested network functions in NFV-based network infrastructures while providing the same performance (regarding throughput and latency) as for traditional physical network functions. Portability is another challenge that operators encounter, as VNFs should be decoupled from any underlying hardware and software to be deployable on different virtual environments.

The placement decision of virtual appliances opens a new research challenge and future directions for NFV as it is crucial to the performance of offered services. Network operators should place VNFs where they will be used most effectively and in the least expensive way. Placement problems usually involve optimization through linear/integer programming or approximation algorithms.

NFV also brings more concerns for resiliency. Resiliency has been defined as the ability to provide and maintain an acceptable level of service in the presence of network failures and challenges to normal operation [167]. NFV ISG has identified various requirements, including resiliency requirements of VNFs [58]. Service Operators are concerned by the resiliency of their services as the unreliable services are likely to be discarded by users and the total costs of system failures can be enormous.

Ensuring such resiliency is difficult for VNFs due to different reasons [47, 79] such as (*i*): the commodity hardware that hosts VNFs is more prone to errors and failures compared with dedicated hardware that can directly affect the VNFs running on them, (*ii*) the software implementations of these VNFs (i.e., at various levels, such as host operating system, hypervisor, VM, or the VNF instance itself) can be rather buggy and are susceptible to failures, and (*iii*) Operation Faults such as incorrect configuration could also be a reason for failures in the NFV environment.

Furthermore, NFV brings new security concerns [63, 163] along with its benefits since software-based virtual functions in NFV can be configured or controlled by an external entity (e.g., third-party provider) and the virtual network functions might run in data centers that are not owned by the service provider. Thus, operators need to guarantee that the security features of their network will not be affected when deploying VNFs. However, the security challenges are out of our consideration in our work.

2.2 Service Function Chaining

A network service is an offer or a favor delivered by a network operator to network users by the use of network functions (such as Firewalls, Network Address Translators, Deep Packet Inspection, etc.) [17].

The main idea behind service chaining is that delivery of end-to-end services often requires various network functions. The definition and instantiation of an ordered set of network functions and subsequent steering of traffic through them is called Service Function Chaining (*SFC*).

The Internet Engineering Task Force (*IETF*) standardization organization has created the Service Function Chaining Working Group to work on function chaining. This group is aimed at producing architecture for service function chaining that includes the necessary protocols to convey the service function chain information to nodes that are involved in the implementation of SFCs, as well as mechanisms to steer traffic through service functions.

2.2.1 SFC Architecture

IETF has taken initiatives towards developing the formal architectures for SFC [78]. The core SFC Architecture is composed of the following logical architectural building blocks:

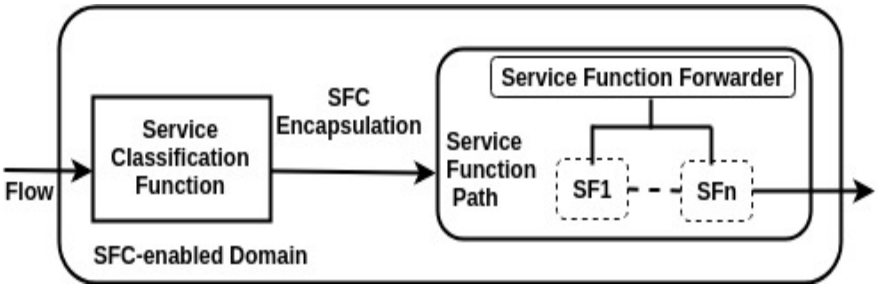
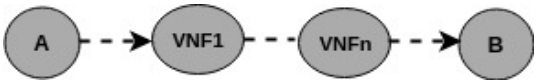


Figure 2.2: Service Function Chaining Architecture.

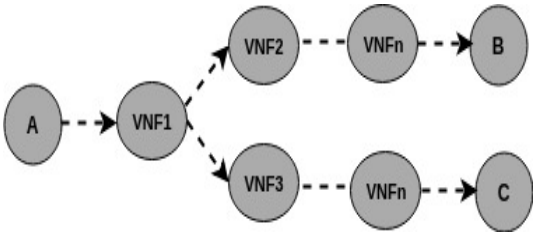
- Classifier: which performs classification on the traffic flows based on some policies. Traffic that satisfies classification rules is forwarded according to the matching rule to a specific path.
- Service Function Forwarder (*SFFs*): which forwards traffic to one or more connected service functions (*SF*) based on information maintained in the SFC encapsulation, handles traffic coming back from the SF and transports traffic to another SFF.
- Service Function: is a resource available for consumption as part of a service. SFs send/receive traffic to/from one or more SFFs. In order to provide a mechanism for such SFs to participate in the architecture, a logical SFC proxy function might be used. The SFC proxy acts as a gateway between the SFC encapsulation and SFs.
- SFC Proxy: which removes and inserts SFC encapsulation on behalf of a service function. It is used to enable the implementation of functions unaware of the SFC encapsulation.

These logical components are interconnected using the SFC encapsulation, which enables service function path selection. It also enables the sharing of metadata/context information when such metadata exchange is required. The Service Function Path (*SFP*) specifies where packets assigned to a certain service function path must go (i.e., specifies all SFF/SFs the packet will visit when it actually traverses the network, see Sec. 2.2.2).

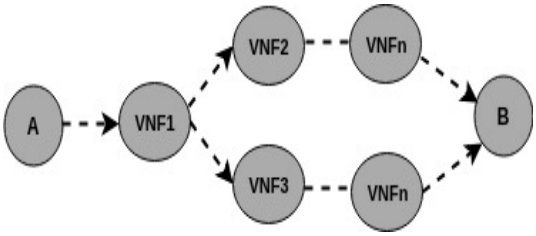
Fig. 2.2 represents the basic SFC architecture and explains the main procedure of flows that traverse the SFC-enabled domain (the SFC proxies are not shown in this figure). When a flow enters an SFC-enabled domain, first of all, it will be classified in order to decide which service function path the flow should traverse. Then, the flow will be encapsulated. The encapsulation step enables the SFP selection and the sharing of metadata information if necessary. Hence, flows from different services can simultaneously traverse the same VNF, but each has its own service function path. After that, the traffic flow will go through all the service functions defined by the encapsulation step. When the SFC is completed, the encapsulation will be removed, and the traffic flow will leave this SFC-enabled domain to continue its transmission in the network.



(a) line



(b) Furcated path with different endpoints



(c) Furcated path with a single endpoint

Figure 2.3: SFC Request Models

2.2.2 Service Function Path

At an abstract level, the service function chain is a conceptual view of a service that defines the set of required functions and the order in which they must be traversed. Service function chains start from the origin network node, or from any subsequent node in the graph. Moreover, a SF may become a branching node in the graph, with those SFs selecting edges that move traffic to one or more branches, depending on user service demands. Service function chains may have more than one termination point [78] as well. A service function path (SFP) is a mechanism used by a service chaining platform to express the result of applying more granular policy and constraints to the abstract requirements of a service function chain [148].

Based on this standardization, we can have three different SFC request models (See Figure 2.3). The simplest topological model is a line with two endpoints and one or more network functions (shown in Figure 2.3a). This model is suitable for handling flows between two endpoints that have to pass through a specific sequence of network functions. The second and third topological models are based on furcated paths. Network flows passing through furcated paths may finish with one endpoint or more.

For the flows with different endpoints (Figure 2.3b), the most basic component contains one source endpoint and two destination endpoints, where there is a network function (e.g., load balancer) that splits the traffic into different paths according to a certain policy. As for furcated paths with a single destination endpoint, we consider a scenario in which different fractions of traffic between two endpoints must be treated differently. For example, one part passes through a specific firewall, while the other

part, through an encryption function. However, more complex SFC requests may be created by freely combining these basic topological components.

2.2.3 SFC Placement Solution Strategy

In an NFV environment, network functions of the SFC are decoupled from the underlying hardware and implemented as VNFs and deployed in the network. To determine the positions for placing these VNFs such that the service requirements and quality can be satisfied is a critical problem. This VNF placement problem, for different objectives and scenarios tackled in the related works, is proved to be NP-hard [11, 22, 36, 42, 169, 174]. For example, Bari et al. [11] introduced the VNF orchestration problem which was equal to VNF placement problem and formulated it as an ILP model to minimize the OPEX and maximize the network utilization. Then, they have proved the NP-hardness of this problem by a reduction from the Capacitated Plant Location Problem with Single Source constraints (*(CPLPSS)*) [166].

In order to obtain the optimal solution for SFCs placement, the mathematical programming methods are generally used, such as Integer Linear Programming (*ILP*) and Mixed ILP (*MILP*). As it was difficult to find the optimal solution for VNFs placement especially in large scale network scenarios in a reasonable time, most of the works done in this area has focused on the design of heuristic or metaheuristic algorithms, such as greedy algorithms. Thus, in the following paragraphs, a few backgrounds on linear programming and greedy solutions are presented.

2.2.3.1 Linear Programming

Linear Programming, also called Linear Optimization, is a general strategy to model and achieve the best outcomes for many combinatorial problems [161, 187], such as the Knapsack problem [153]. In this problem, given a set of items where each item has a specific weight and a value, the goal is to find which items should be selected, such that the total weight is less than or equal to a given limit, and the total value is as large as possible. This problem often arises in resource allocation where there are many constraints and it is studied in many fields such as computer science.

Fundamentally, an LP is composed of a linear objective function, a set of linear inequality constraints formalized by variables representing the problem outputs and parameters (i.e., the problem inputs). The objective function represents the optimization goal and is written in terms of minimizing or maximizing, e.g., minimizing resources consumption and maximizing the provider profit. If the goal is just to find feasible solutions satisfying constraints, the objective function can be omitted. Normally, an LP is expressed in a canonical format as in Equation 2.1:

$$max \left\{ c^T x : Ax \leq b; x \geq 0 \right\}, \tag{2.1}$$

where x is the variables vector, b and c represent the coefficients vectors, while A is the coefficients matrix and $(.)^T$ is the matrix transpose. If all variables are integers, the LP will be called an Integer Linear Programming (*ILP*), as in the bin packing

problem. However, when only part of variables in x are integers, the LP is called a Mixed Integer Linear Programming (*MILP*).

Due to its wide utilization, many methods have been proposed to solve the LP problems, such as cutting plane [54], branch and cut [65], column generation [13]. These methods are usually implemented in LP solvers (e.g., CPLEX [86], Gurobi [138]), which can find exact and approximate solutions for the problem. In our study, we formalize the SFC placement problem as an ILP in Chapters 5 and 6 where we use the Gurobi solver, which is free for academic usage.

2.2.3.2 Greedy Algorithms

In general, most of the placement problems are known to be NP-hard [7], which means that there are no known polynomial-complexity algorithms to find the optimal solution, and using LP solvers for large problem instances (e.g., a large number of variables or constraints) is not practical because of the tremendous execution time.

To deal with this limitation, heuristics are used to find near optimal solutions [74]. Basically, an heuristic tries to improve the solution in each step and it stops when the result obtained is good enough. There are many popular heuristics, such as Greedy algorithms. A Greedy algorithm is one that makes choices based on what looks best at the moment. In other words, choices are locally optimal but not necessarily globally optimal. For example, to find solutions for the Knapsack problem mentioned before, one possible greedy approach is that in each step, a new item with the maximum value of (value/weight) is selected. In many problems, Greedy does not guarantee to find the global optimum, but it can approximate the global optimum in a reasonable execution time. For example, Greedy has been proven that it is 1/2-approximation for the Knapsack problem [161].

Considering the VNF placement problem, many greedy solutions have been proposed in the state of the art [2, 121]. In this thesis, the greedy algorithm considered is based on the First-Fit Decreasing (*FFD*) algorithm [53], to take placement decisions in Chapter 5. The key idea of this heuristic is to install virtual functions with the largest resources requirements in the first physical place in which it fits. When all of the functions of service are mapped to the physical network, the process is considered complete.

2.3 Availability

In this thesis, we define the availability as the ability of a service to be accessible and operational at a given time, i.e., the amount of time a service is actually operating to the total time it should be working [1].

The typical availability values are specified in decimal (such as 0.9998). In high availability applications, a metric, known as nines, is used. With this convention, "five nines" equals 0.99999 (or 99.999%) availability. The more "nines" means less

downtime of the service in minutes per year and higher availability. In Table 2.1 the service availability and its downtime are given [14].

Availability Value	Downtime Value
0.9	Down 5 weeks per year
0.99	Down 4 days per year
0.999	Down 9 hours per year
0.9999	Down 1 hour per year
0.99999	Down 5 minutes per year
0.999999	Down 30 seconds per year
0.9999999	Down 3 seconds per year

Table 2.1: Service availability and downtime values

The simplest representation for availability for a single component is the ratio of the expected value of the uptime of a system to the aggregate of the expected values of up and downtime. Availability is commonly defined through the following equation 2.2:

$$Availability = \frac{MTBF}{MTBF + MTTR}, \tag{2.2}$$

where Mean Time Between Failures ($MTBF$), is the average time interval (normally in hours) between two sequential component failures, while the Mean Time to Repair ($MTTR$), is the average time needed to detect the failure, repair the failed component and return it to its normal operations.

Herker et al. [81] have provided statistical data on element failures within data centers, namely (i) The MTTR and (ii) the MTBF, based on previous works in [68] and [88]. Based on their studies, we determine the availability values for each component in our DC network topology as in the Table 2.2.

DC component	MTBF (hours)
Server	0.99
Top of Rack/Aggregation switch	0.9999
Core switch	0.99999

Table 2.2: Availability values for different DC components

High availability (*HA*) [173] is a system or component characteristic that refers to the ability of this system to be operational and functional, namely *uptime*, for a long period of time. If the mean time between failures is very large compared to the mean time to repair, then a high availability will be reached. For example, data centers require high availability feature of their systems to perform daily activities. If a user cannot access a system, this system will be considered unavailable from the user's point of view and the term *downtime* is used to refer to these periods when the system is unavailable.

2.3.1 Availability with Service Level Agreement

Data center networks are error-prone as several network elements exhibit failures that can impact the normal operation. Under this condition, service providers cannot promise the delivery of persistent services to their tenants due to unavoidable network elements failures. Thus, the tenants need to know the percentage of time that the offered service will be available. The lowest acceptable availability by the tenants without incurring penalties for the provider has to be clearly defined in a specific agreement form called a Service Level Agreement (*SLA*) [126]. The SLA is a business contract that defines the commitments and requirements of providers and customers.

If the SLA requirements are not fulfilled, the service provider has to pay an agreed amount of money, which is called *SLA penalty*. The penalty plays a very important role, as it encourages the service providers to implement the requisite enhancement on their network and to do their best in order to maintain the service within the requirements. Furthermore, it gives to the tenants the opportunity to compensate for the service outage.

The SLA penalty may be defined in two different ways [70]. The first possibility is a binary model, where the penalty to be paid is 0$ if the delivered availability is above the agreed availability in the SLA contract, or a fixed amount of money if the requirements are not respected. Another option is a contract with an increasing penalty. In this option of contracts, the price of penalty increases after a certain accumulated downtime.

However, regardless of the SLA penalty models, violation of the SLA is not desirable especially in case of critical services such as e-health or autonomous transportation systems.

2.3.2 Fault Tolerance and Redundancy

High availability is often associated with fault-tolerant systems. The term *fault-tolerant* means that a system can operate in the presence of hardware component failures. For instance, a single component failure in a fault-tolerant system will not cause a system interruption because other alternate components will take over the task transparently.

The most common techniques to ensure resiliency of the provided service are: (*i*) redundancy and replication, and (*ii*) failure detection and failure recovery mechanisms. Furthermore, the proposed schemes for VNF replication can be classified in two main categories: (*i*) Active-Standby method, which is a simple solution where it is not required to have load balancing functions, but a mechanism to redirect traffic flow towards standby nodes in case of failure for the active ones, and (*ii*) Active-Active scheme, where all nodes are active and participating in providing service. With this solution, an additional load balancing function before the pool of replicated functions is used to distribute the traffic. In the case of failure, the traffic will be redirected to survived entities.

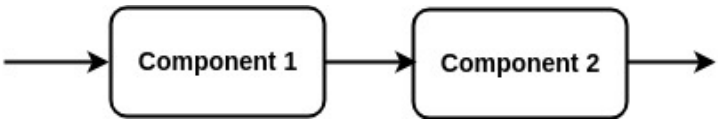
Moreover, there are different possibilities for each category. For example, using an active-standby architecture, the following options are possible:

- 1:1, a single active node is protected by a single standby node
- N:1, N active nodes protected by only one single standby
- N:M, N active nodes protected by M standby nodes.

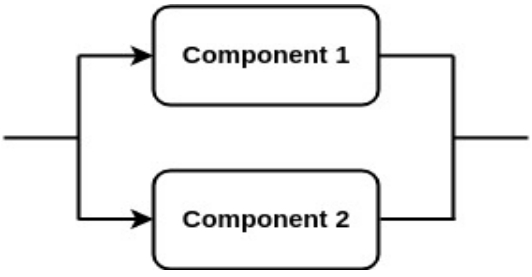
In this thesis, we have considered the active-active scheme with two options N+1, called *strict*, and N+M called *relax* in Chapters 5 and 6.

2.3.3 Availability in parallel and series systems

Typically, network systems are not only composed of a single component, on the contrary, it is a complex system comprising many components connected together in various ways (as in Figure 2.4). Thus, the system availability is calculated by modeling the system as an interconnection of parts in series and parallel manner. If one failure of a part makes another part inoperable, then two parts are considered to be operating in series. Otherwise, if the failure of a part did not affect other parts, the parts are considered to be operating in parallel.



(a) Serial connected components



(b) Parallel connected components

Figure 2.4: Two ways of combining system components

When a system is composed of two (or more) components that are connected in serial way (as in Figure 2.4a). All these components are required to be available in order for the system to be available [191]. If either one of those components fails, the system is considered to be unavailable (see Equation 2.3). However, in the parallel configuration [186] (as in Figure 2.4b), when one of the components is available, the system can survive. Components in this configuration are considered redundant and the availability is calculated as in Equation 2.4.

$$Availability_{serial} = \prod_{i}^{n} A\ (Component_i) \tag{2.3}$$

$$Availability_{parallel} = 1 - \prod_{i}^{n} [1 - A\ (Component_i)], \tag{2.4}$$

where n is the number of components in this system. When the system consists of multiple components, some of them are connected in a serial way and the others in parallel, to calculate the availability of such system, one may calculate any consecutive serial/parallel components and replace them with blocks with new availability in order to be able to complete the calculation.

2.4 Data Center Network Topologies

The data center is home of the computational power, storage, and applications necessary to support enterprises business to deliver various services to their customers. Good planning of the data center infrastructure design is critical, and issues such as performance, resiliency, and scalability should be considered carefully [38].

The data center network design is based on a *layered* approach. This layered approach is the main principle of the data center design that searches for improving scalability, flexibility, resiliency, performance, and maintenance. Most of the network topologies for data centers share a multi-tier architecture, especially three-tier architecture. These three layers of the data center design are briefly described as follows:

- Core layer: which provides the high-speed packet switching backplane for all flows going in and out of the data center. It also provides connectivity to multiple aggregation switches. The core layer runs an interior routing protocol, such as Open Shortest Path First ($OSPF$), and load balances traffic between the campus core and aggregation layers. Core layer switches are also responsible for connecting the data center to the Internet.
- Aggregation layer: It is also called a distribution layer aggregates the uplinks from the access layer to the DC core layer. This layer is a crucial point for control and application services. Security and application service devices such as load balancing, firewalls, and IPS middleboxes are often deployed as modules in this layer.

- Access layer: Network servers, also called hosts, physically attached to the network in this layer, that are also called Edge layer. The access-layer is typically built with high-performance, low-latency Layer 2 switches that provide connectivity to the aggregation layer switches.

Moreover, typical DC architectures can be categorized in switch-only or switch-centric topologies such as 2-/3-tier tree and Fat-Tree or server-centric topologies such as BCube and DCell. Here, we present the data center (DC) topologies that are considered as targeted environments for the evaluation in this thesis.

2.4.1 Tree Data Center Topology

Tree networks architecture is attractive because of its simplicity of wiring and reduced economic costs. The traditional basic tree network topology is usually built with two or three tiers, which are edge tier, aggregation tier, and core tier (shown in Figure 2.5). The leaves in this tree are network servers which are connected to top of rack (ToR) switches in the access layer. These ToR switches are then connected to switches in the aggregate layer, which are successively connected to core layer switches [179].

The core layer interconnects the aggregation switches together and controls the traffic flows into and out of the DC, while the aggregation tier provides the domain service and load balancing. This design approach results in a serious bandwidth oversubscription towards the network core tier that occurs when the overall switching bandwidth of a switch is less than the total bandwidth of all ingress switch ports. Tree topologies suffer from scalability bottlenecks. Thus, to solve this problem, a Fat-Tree topology was proposed (more details are presented in Sec. 2.4.2).

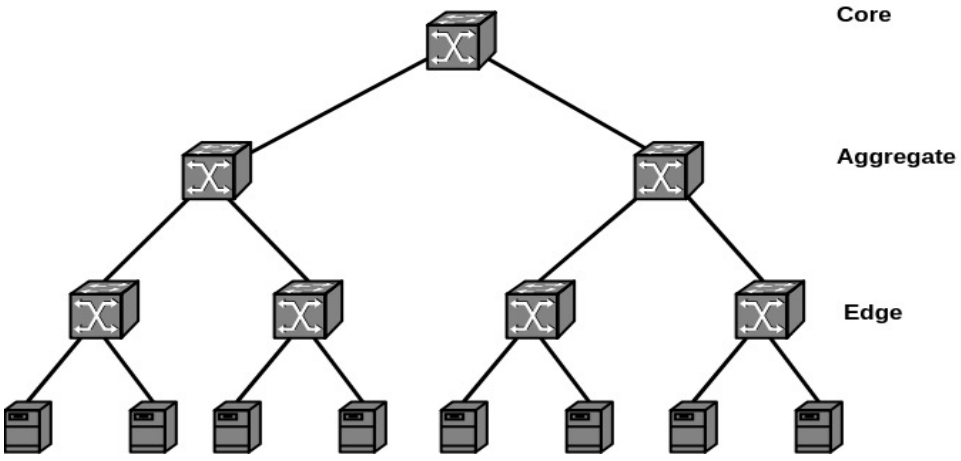


Figure 2.5: Traditional 3-tier Tree Data Center Topology.

2.4.2 Fat-Tree Data Center Topology

Fat-tree DC topology is a three-tier Clos network built in the form of a multi-rooted tree. It was firstly introduced by Al-Fares et al. to construct the data center network [5].

This architecture provides a non-blocking structure, which provides an oversubscription ratio of 1:1 to all servers. The Fat-Tree network topology is built with n pods, where n is the degree of the Fat-Tree topology. The pod is defined as a collection of edge and aggregation switches connected in a complete bipartite graph. Each pod contains $(n/2)^2$ servers and 2 layers of $(n/2)$ n-port switches. It supports $(n^3/4)$ servers in total. For $n = 4$, the fat-tree topology is shown in Figure 2.6 and supports up to 16 physical hosts connected to 8 edge switches at the bottom.

The main advantage of this topology is that $(n^2/4)$ redundant paths are available to route the traffic between any two physical servers. Thus, this topology allows for high bisection bandwidth using a large number of less expensive switches allowing support for a large number of servers at much less cost.

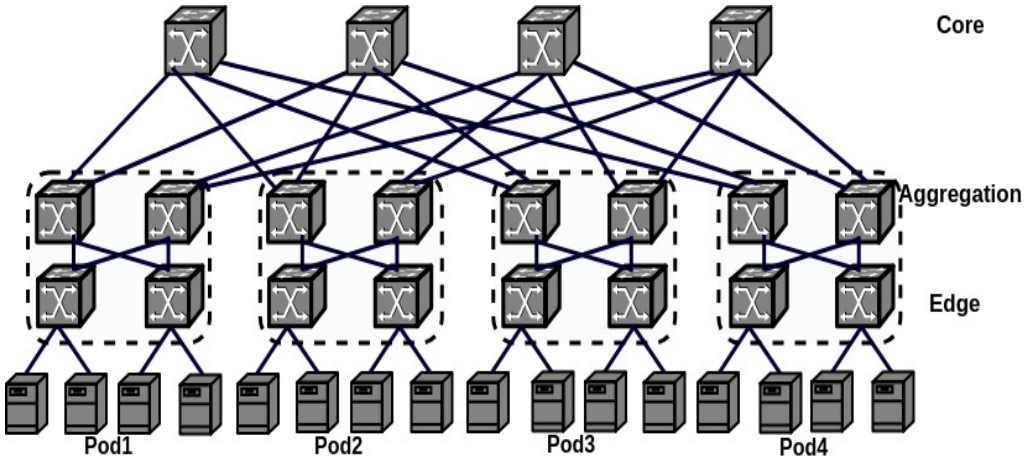


Figure 2.6: 4-Fat-Tree Data Center Topology.

2.4.3 Spine-and-Leaf Data Center Topology

Spine-and-Leaf network topology [10,39] is a two-tier Clos network architecture where each ToR switch in the leaf layer is directly connected to servers and to all Spine switches, the backbone of the network, in the upper layer in a full-mesh topology (shown in Figure 2.7).

The physical links that interconnect the servers to leaf switches may have a different capacity from the links that connecting leaf switches to the spine switches. The path between any two servers is randomly chosen so that the traffic load is distributed among the top-tier switches. If one of the top tier switches failed, it would only slightly degrade performance throughout the data center.

Spine-and-Leaf topology makes it easier to extend the data center capacity to cope with an oversubscription of a link by adding more spine or leaf switches. However, in case high bisection bandwidth is intended, scaling to more than hundreds of servers in a cluster can lead to increased costs due to the need for spine switches with many high capacity ports.

With the spine-and-leaf architecture, it does not matter to which leaf switch a specific server is connected as its generated traffic has always to cross the same number of devices to reach another server (unless the other server is connected to the same leaf switch). This approach ensures a predictable level of latency as the traffic only has to hop to a spine switch and another leaf switch to reach the destination server.

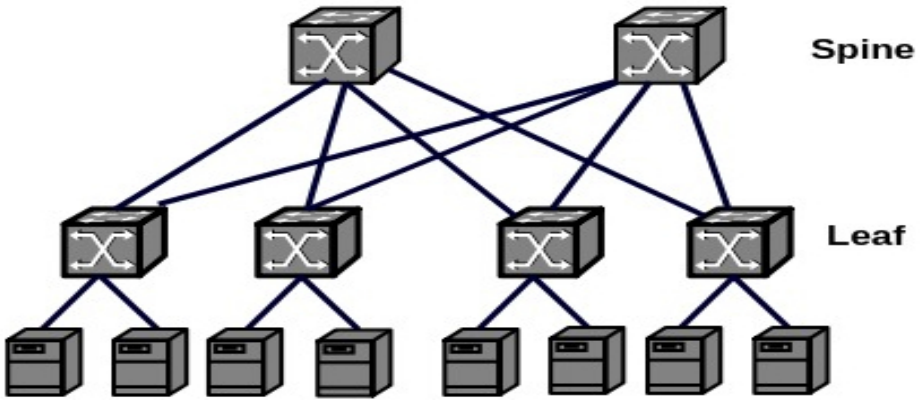


Figure 2.7: Two-tier Spine-and-Leaf Data Center Topology.

2.5 Big Data Processing

Under the explosive increase of global data, the term *Big Data* is mainly used to describe unstructured massive data sets that could not be perceived, managed, and processed by traditional IT and software/hardware tools within a tolerable time [32]. The challenge is not only to store and manage such these datasets, but also to process and extract meaningful value from it, where it requires a data-intensive application to be deployed on large scale systems.

To meet these challenges, several parallel execution models on distributed architectures have been proposed. The most popular method to process huge data sets adopts the *MapReduce* programming model [49] and its open source implementation Hadoop, as it solves the data volume problem successfully (by doing the computation in a distributed manner) and provides resiliency against machine failures. In this section, we present the MapReduce programming model, its widely used implementation Hadoop [19] to process certain unstructured data.

2.5.1 MapReduce Model

MapReduce is a software framework popularized by Google to support Big Data distributed computing on clusters of computers. It runs on top of the Google File System (*GFS*) where data is loaded, partitioned into chunks, and each chunk replicated across multiple machines. The user of MapReduce programming model expresses computation as two functions: (*i*) the Map function that processes each data chunk into key/value pairs to generate a set of intermediate key/value pairs (Equation 2.5):

$$map(key1, value1) \rightarrow list(key2, value2) \tag{2.5}$$

(*ii*) the reduce function that merges all the intermediate values associated with the same intermediate key to compute the final result:

$$reduce(key2, list(value2)) \rightarrow list(value2) \tag{2.6}$$

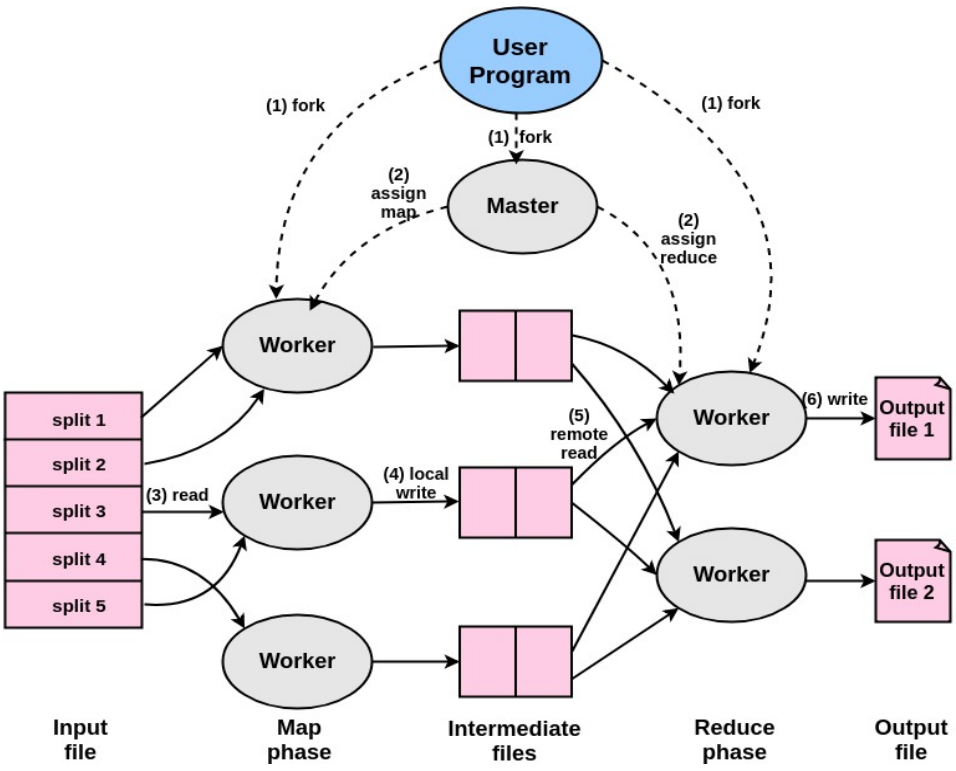


Figure 2.8: Map Reduce Execution Overview.

When the user program calls the MapReduce function, the following sequence of actions occurs (as shown in Figure 2.8). The MapReduce library in the user program first splits the input files into M pieces of typically 16 to 64 megabytes per piece. It then launches many replicas of the program on a cluster. One is the "master" and the rest are "workers". The master is responsible for assigning the *map* and *reduce* tasks to the workers and monitoring the task progress. Thus, there is a single point of failure. When map tasks arise, the master assigns the task to an idle worker,

taking into account data locality. A worker reads the content and emits key/value pairs to the user-defined Map function.

Thereafter, the intermediate key/value pairs are buffered in the memory and periodically written to a local disk, partitioned into R sets. The master passes the location of these stored pairs to the reducer which reads the buffered data using remote procedure calls, sorts and gathers the intermediate values associated with the same key. For each key, the Reduce function is applied to generate the output in R output files (i.e., one per reduce task).

2.5.2 Hadoop Framework

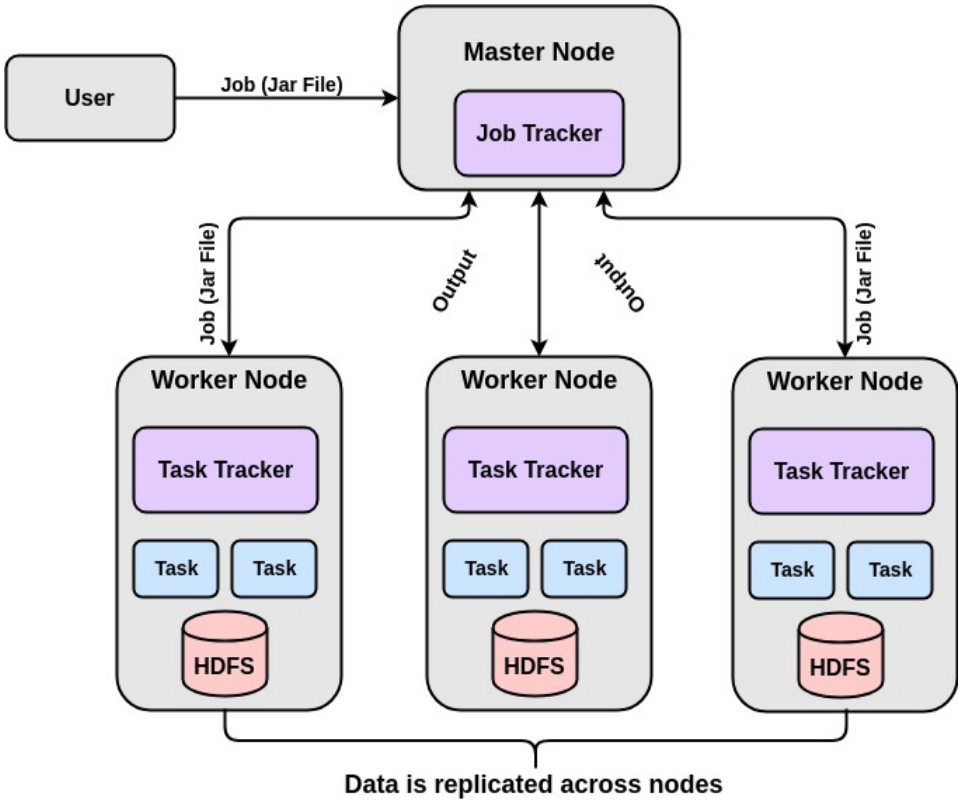


Figure 2.9: Hadoop Architecture.

Apache Hadoop [19] is a Java open-source implementation of MapReduce sponsored by Yahoo!. The two fundamental sub-projects are the Hadoop MapReduce framework that provides the compute layer and the Hadoop Distributed File System (*HDFS*) [165], which provides the data layer (See Figure 2.9).

Hadoop Distributed File System is designed to run on clusters of commodity machines. Each HDFS cluster consists of a single NameNode, a master server that manages the file system namespace and regulates access to files by clients, in addition to a number of DataNodes, which are the actual store of the data blocks, usually one per node in the cluster. They manage storage attached to the nodes they run on.

The NameNode splits the files into blocks that are stored in a set of DataNodes. It also executes file system namespace operations like opening, closing and renaming files and directories. It determines the mapping of blocks to DataNodes. DataNodes are responsible for serving read and write requests from the file systems clients and perform block creation, deletion, and replication upon instruction from the NameNode.

The MapReduce framework is a software framework for distributed processing of large data sets on compute clusters. It has a master, called JobTracker, responsible for: (*i*) querying the NameNode for the block locations, (*ii*) scheduling the tasks on the worker (or it could be called slave), which is hosting the tasks blocks, and (*iii*) monitoring the successes and failures of the tasks. The workers, called TaskTracker, execute the tasks as directed by the master.

3 State of the Art

Contents

In this chapter, we first review the state of the art regarding the problem of placement, starting from VM placement to service function chain placement. The related works had considered a large number of objectives. However, in this chapter, we provide summaries of these works with a focus on the approaches that had taken the resiliency as the main objective based on redundancy solutions. Furthermore, the works relevant to big data processing and different MapReduce frameworks, with emphasis on works has been done in adapting MapReduce to perform well in the presence of failures, are discussed here to help us in providing a clear position of our contributions compared to the literature.

3.1 Placement in Virtualized Environment

With the virtualization technology, data centers can consolidate their services onto a lesser number of physical servers than originally required with the use of virtual machines. Mapping virtual machines (VMs) to physical machines (PMs) is called the VM placement. In other words, VM placement is the process of selecting the appropriate physical host (or it could be called a physical node) for the given VM [96]. To use the physical resources effectively, VM should be placed on a convenient host. Since the usual size of data centers is large in the cloud computing environment, selecting a correct physical host to place the VM is a very challenging task during the virtual machine placement.

Many virtual machine placement algorithms with different goals have been proposed in the literature for the cloud computing environment [20, 25, 89, 99]. The placement goals can be related to performance (e.g., CPU capacity, link bandwidth), energy-efficiency [103, 156, 184] (e.g., power usage), reliability [89, 113], or other parameters [99].

Rodrigo et al. [25], proposed a heuristic for the mapping between VM and PM. The aim of this approach is to balance the CPU utilization on each PM. In their

heuristic, named Hosting Migration-Networking (*HMN*) heuristic, a list of PMs is built in descending order of CPU capacity and the first acceptable physical host is chosen. Moreover, a migration step is performed after the initial mapping to increase the system load-balance. Jayasinghe et al. [89] also present a placement algorithm with the objective of improving the performance and availability of cloud services. Their solution takes as input an application description with constraints and a data center description and translates them into tree models. Then the algorithm places VMs on physical machines by grouping VMs, placing VMs groups on server clusters, and then checking if the individual VM requirements are met.

The same problem is also considered in [144] where the authors propose a network-aware VM placement and migration schema which maps the VMs to servers, taking into account the network conditions to minimize the data transfer time consumption and retain the application performance. Furthermore, Li et al. in [99] proposed a new method for online VMs placement requests with the objective of minimizing the job completion time. Their heuristic has two scenarios, namely Direct placement and Migration-based placement. In the direct placement, the algorithm estimates the completion time by placing the current VM over the candidate PMs and then chose the physical machine that ensures the shortest total completion time and has sufficient resources. However, in the migration-based placement, the algorithm chooses the PM that provides the lowest total completion time, whether it has enough resources to host the VM or not. When the available resources are not sufficient, the algorithm migrates one of the already placed VMs on this PM to another PM in a way to place the new VM.

Energy consumption is one of the most critical problems in the cloud environment and cannot be ignored. This important objective is considered by Lin et al in [103]. In this work, the authors proposed two different algorithms for energy-effective virtual machine provisioning and consolidation. The first algorithm, called Dynamic Round-Robin (*DRR*) is an extension of the original Round-Robin method [77], while the second algorithm is the Hybrid, which combines DRR and First-Fit method [185]. Sampaio et al. [156] also take this problem with the same objective into account. To save energy and provide fault tolerance, virtual machine consolidation or migration has been used to put idle physical machines in sleep mode based on two proposed heuristic algorithms. The proposed algorithms consolidate VMs when needed, taking into account the reliability of the nodes, to minimize energy consumption. The resource assignment to VMs is updated dynamically along time. Furthermore, virtual machine placement with the same objective is considered in [184] based on a genetic algorithm (*GA*) where the authors considered energy consumption in communication networks as well as in PMs in the proposed algorithm.

Network Virtualization has become more important where it could be used for network simulation [15, 37] or to provide customized end-to-end services over the same physical network [9, 62]. To build the virtual network, Virtual Network Embedding (*VNE*) is introduced. Several algorithms to solve the problem of VNE have been discussed in the literature with various goals [6, 22, 33, 36, 105, 150, 196]. The VNE problem addresses the efficient mapping of a set of Virtual Network Requests (*VNERs*) to physical nodes and links [64]. Each VNER is composed of virtual nodes that must be mapped to a set of physical nodes with sufficient resources to satisfy

the requirements, and a set of virtual links to be mapped to a set of physical paths in the network.

In [174,196], authors have proposed heuristic greedy algorithms to maintain low and balanced stress among all physical hosts and links during the VN assignment process. The aim of such greedy algorithms is to assign virtual nodes to substrate nodes with maximum available resources. The VNE studies either assume that the VNE requests are known in advance, namely offline, such as [22, 109, 150, 196], while other works tackle the problem of the online VNE where the VNRs arrive dynamically and they are not known in advance, such as [6,36,105,174]. Botero et al. [22] have tackled the offline VNE problem and solved it with mixed integer program (*MIP*) but with the goal of minimizing the energy consumption. Lu et al. [109] also handled the offline variant for only a single virtual network with a backbone-star topology with the objective of minimizing the cost. The authors have assumed that only bandwidth constraints are imposed and the network resources are infinite.

The online scenario is considered in [174]. The authors in this work have proposed to queue a group of incoming VNRs (ordered by revenue) during a time window and try to efficiently place them. If it is not possible to accept the request, the latter will be sent to the queue and wait for resources release in the network until its timeout finishes, then it will be canceled. In [105], Lischka et al. have proposed another online heuristic algorithm approach based on the Subgraph Isomorphism Detection (*SID*) problem with backtracking. In this approach, the authors try to find an isomorphic subgraph representing the VNER inside the network.

Moreover, by virtualizing network functions, such as firewalls and load balancers, that were formerly carried out by vendor-based specialized hardware devices, and migrating them to software-based appliances, a new challenge of these virtual functions deployment arises with the presence of resource scarcity (e.g., computation, storage, and bandwidth) on VMs. VNF-based SFC and the corresponding resource optimization problems have been widely studied. The works of resource allocation in SFC are closely related to the previous VNE problem and can always be formulated as an optimization problem. Some of these works have considered offline scenario [110, 122, 128, 154], while other works have considered online SFC requests [33, 124, 129, 169, 180].

Moens et al. in [128], have formulated the placement problem as an ILP model with the objective of allocating the SFC requests within NFV environments while minimizing the total number of servers used. Mehraghdam et al. [122] have also proposed a VNF placement algorithm but with different optimization goal. Their approach constructs a VNF forwarding graph to be mapped to the physical resources, assuming limited network resources and functions with specific requirements and possibly shared. In their evaluation, they maximize the link available bandwidth and minimize latency and number of host nodes but do not account for robustness.

The same problem is solved in a DC topology by Cohen et al. [42] to minimize the total system operational cost (OPEX). However, the proposed LP-relaxation solution has the flaw of violating physical resource capacities as NFs are shared between the clients.

Sahhaf et al [154] also treated the problem of service function chaining through an ILP model and a heuristic-based algorithm composed of two phases: a decomposition selection with backtracking phase and a mapping phase, leading consequently to sub-optimal solutions. Luizelli et al. [110] formulated the ILP model targeting on minimizing the end-to-end delay. They have proposed a binary search heuristic to cope with large infrastructures, to jointly place VNFs and map service chains onto them without any ordering constraints. The objective was to minimize the number of virtual network function instances mapped on top of the NFVI. Gupta et al [73] also formulated the problem as an ILP to minimize bandwidth consumption.

Bari et al. [12] studied the SFC placement problem and solve it for determining the optimal number of VNFs required and their placement with the objective of minimizing the OPEX caused by the allocation to the service provider while guaranteeing the service delay bounds. They formulate the problem using an ILP and present a heuristic that maps nodes for each request on a single physical host.

Mijumbi et. al [124] formulated an online VNF mapping and scheduling problem and propose a set of greedy algorithms and a Tabu Search meta heuristic [69] for solving it with the objective of reducing the flow execution time and the embedding cost in Telecom Service Provider (*TSP*) environment. Wang et al. [180] also considered the online placement to determine the optimal number of VNF instances and their optimal placement in DCs, which minimizes the operational cost and resource utilization over the long run. Their algorithm takes scaling decisions based on current traffic and assumes infinite inter-servers bandwidth.

Moreover, Mohammadkhan et al. [129] proposed a MILP formulation to determine the placement of online VNFs requests with the objective of reducing latency by minimizing the link bandwidth and the number of used cores. They propose a heuristic to solve the problem incrementally but do not consider resiliency against failures. An ILP-based model is also proposed by Sun et al. [169] to minimize the total deployment cost while increasing the service providers' revenue by increasing the probability of accepting SFC requests.

Since solving the placement problem is shown to be hard. The mathematical proposals suffered from a scalability weakness, because of the execution time grows as the network size increases [11,169]. For example, Bari et al. [11] spent about 26 minutes using CPLEX to solve their ILP model with a small network scenario with only 23 nodes and 43 links.

Being aware of this problem, many works have proposed heuristics algorithms right after solving their optimal models [11,12,18,66,99,124,143,189]. For instance, Pham et al. [143] have proposed a heuristic solution for energy saving objective based on Markov mechanism to solve the VNF placement problem. Precisely, the algorithm started with an arbitrary chosen feasible VNF placement solution, and moved to another feasible one based on the network states. This algorithm converged when the Markov chain reached the steady-state.

Algorithms like the Simple Greedy Approach (*SGA*) [2,18,60] and heuristics such as First Fit Decreasing (*FFD*) have been widely studied and proposed in the literature for the VM placement problem to reduce the time needed to get a reasonable solution.

Many authors compared their own solutions to one of SGAs such as [18,103,168]. In FFD approach, VNFs are organized in a decreasing order of resource requirements and each VNF is then placed into the first physical server available with sufficient remaining resources.

3.1.1 Resiliency

Many studies showed that hardware and software failures are common [68, 72, 146] and with NFV-based environment, where low reliable commodity hardware is used, the chance of failures is even increased [56]. The failure detection and recovery time depends on the type of failure and may take seconds or more for hardware failures such as link and node failures [68]. Thus, ensuring high availability (*HA*) to maintain critical NFV-based services is an important design feature that will help the adoption of virtual network functions in production networks, as it is important for critical services to avoid outages.

To guarantee the continuity of service, VNFs must be able to preserve the related state information which can be used to protect customers from disruptive events and to recover services from disasters quickly [58]. In addition, to make sure that essential services (e.g., the voice call service for emergency events) are still available when failures happen, network operators need to give more attention to the placement decisions. Although network operators cannot guarantee the functionality of all the services when a large scale network disaster occurs, they can at least guarantee service continuity of some essential services when simple physical failures happen through redundancy solutions and by avoiding a single point of failures.

Some works considered this problem and introduced solutions for failure detection and consistent failover mechanisms. Kulkarni et al. [94] presented a resiliency framework to deal with all different kinds of software and hardware failures where they replicate state to standby NFs while enforcing NF state correctness. Resiliency is also considered by Schöller et al. [159]. In their work, they describe a deployment function that takes an abstract service description including placement and resiliency requirements as input and focusing on the delay between redundant instances. This deployment function is based on OpenStack [152] using availability zones. Marotta et al. [117] described a different robust placement algorithm to cope with variations on resources required for VNFs. They leveraged the Robust Optimization (*RO*) [16] theory to reduce energy consumption by minimizing the number of hosts used.

Robustness is considered with the works related to VM placement problem [20, 113, 193]. Machida et al. [113] and Bin et al. [20] both addressed the problem of making virtual machines (VMs) resilient to k physical host failures. They define a high-availability property so that if VMs are marked as k-resilient, they can be safely migrated to other hosts when there are less than k host failures. The k-fault tolerance is considered by Zhou et al. [193]. To improve the reliability of cloud services, a network-topology aware redundant VM solution is proposed to minimize the consumption of network resources, under the k VM failure using backup VMs.

Fan et al. [59] proposed an approximation online algorithm to map SFC requests with HA requirements with the objective to maximize the acceptance ratio while

reducing the resources used. They assumed that VNFs are heterogeneous in terms of functional and resource requirements but they consider several DCs and assume the presence of protection schemes in the DC so that the deployed VNFs always have 100% availability.

Furthermore, Oechsner and Ripke discussed the topic of VM placement in the context of NFV deployment [136]. They utilize a placement mechanism with a resilience pattern mapped to OpenStack to provide an automatic deployment of resilient components in cloud environments. The considered use-case is to place a redundant active-backup pair of VMs with the requirement of placing instances close enough to ensure the end-to-end delay, but far enough apart to guarantee a certain level of availability. Their heuristic takes as an input, the availability of the components of the physical infrastructure, the delay between them, the maximum delay between the redundant instances and the minimum availability of the joint component.

All these previous works ensure robustness against physical nodes failure. However, some works tried to guarantee the robustness in the presence of physical link failures [6, 150, 170]. Rahman et al. [150] took into consideration that network infrastructure does not remain always operational. Thus, to solve the VNE problem in such situation, the authors proposed a proactive and a hybrid policy heuristic based on a speedy re-routing strategy and utilizes a pre-reserved portion for backup on each physical link. Tomassilli et al. [170] presented two models for dedicated and shared path protection against a single link failure in elastic optical networks (with 24 nodes, 43 links). Khan et al. in [6] also presented a multi-path link embedding mechanism based on the path splitting to achieve VNE survivability against one physical link failure while solving the VNE problem.

Replications have been studied in a virtualization environment [27,28]. Knowing that VNF replications may help in the network load balancing, Carpio et al. [28] tackled the problem of VNF placement using replications. To that aim, they provided a linear programming model for small networks, and two heuristics for large networks to reduce the required computation time for the placement and the replication of VNFs. The authors in another work [27] proposed a new linear programming model, to solve the optimal placement of VNFs to minimize the network cost and balance the utilization of all links in mobile core networks. Hmaity et al. [82] also showed that to ensure the resiliency against single-node failures, it was required to duplicate the amount of resources to survive against0 a single node or link failure.

3.1.2 Availability-aware placement

One essential issue in NFV deployment is to ensure the availability of the provided services. Compared to traditional IT services with availability of the order of two to three nines, telecom services ask for higher availability requirement [79] (i.e., five or six nines).

Multiple works tackled the problem of robust placement taking into account the availability of the physical network elements [81, 89, 123, 190]. Zhang et al. [190] and Sampaio et al. [156] considered the MTBF of DC components to propose high availability placements of virtual functions in DCs. However, none of these works

consider the benefits of using redundancy to ensure reliability. Rabbani et al. [149] solved the problem of availability-aware Virtual Data-Centers (VDC) embedding by taking into account components' failure rates when planning the number and the place of redundant virtual nodes but they do not consider the case of service chains.

Fan et al. [60] explored the problem of online mapping service function chains with guaranteed availability. They assumed that only VMs would experience failures and ignore others, such as switches and links. Ding et al. in [51] introduced a heuristic scheme to efficiently place primary and backup VNFs taking into account the availability of backup deployments. The authors assign a reliable SFC by computing the cost of all physical resources within that SFC.

In Herker et al. work [81], SFC requests are mapped to the physical network to build a primary chain, and backup chains are decided based on that primary chain, while Qu et al. [147] proposed multi-path backup schemes to maximize the reliability of SFCs in Data center networks while minimizing end-to-end service delay. The proposed LP model also replicated the traffic between two end-points in order to support immediate recovery after a failure. While applying this solution does not prevent service outage, redundant network and server resources are necessary to ensure the fast recovery. Moreover, Engelmann et al. [55] proposed to split service flows into multiple parallel smaller sub-flows sharing the load and providing only one backup flow for reliability guarantee.

The work of [61] proposes a different online backup selection mechanism. Precisely, first, VNF chains are mapped onto the network's substrate. Then, backup VNFs are selected on the fly if the performed mappings violate the reliability requirements the authors tended to backup the most unreliable VNFs to improve the total reliability of SFCs.

3.1.3 Conclusion

In the light of the current research regarding the placement problem (presented in Sec. 3.1), we found that each work of the state of the art contributions only addresses a single problem: either placement with some optimization goals (e.g., placement of VMs, VNFs or SFCs), or online/offline placement while some works have considered the resilient placement.

To explain this clearly, we provide a taxonomy for these efforts. In this taxonomy we classify the related works with regard to these metrics: (*i*) The targeted environment (Data center or Mobile networks), (*ii*) the type of the placement problem (i.e., online or offline problem), (*iii*) The problem area (i.e., VM, VNE or VNF), (*iv*) The used solution (i.e., exact or heuristic one) and (*v*) the main objective (i.e., energy, latency, cost, resiliency, etc.). The taxonomy for the main contributions done in the literature is shown in Figure 3.1.

Based on this taxonomy, we are the first to propose an approach that considers the online placement of SFCs requests in data center topologies while taking resiliency requirements into account as ensuring the continuity of the provided services has been one of the main challenges in NFV environment. Motivated by this fact, we

first present a general discussion on the necessity of considering resiliency while placing virtual functions where show that the choice of the placement may have a dramatic impact on the ability of the system to be robust to failures (See Chapter 4).

Thereafter, we provide an algorithm (See Chapter 5) for users SFC placement considering that VNF-based SFC requests come on an online manner, similar to the assumptions considered by Wang et al. [180], but differently from them, we consider that each SFC is dedicated to only one tenant. We solve this placement problem using the k-resilient property to guarantee a requested robustness level as in [20, 113, 193] but per-SFC request instead of per VM. We provide a solution based on a priori VNFs replication [27, 28] to avoid the outage of critical services upon failures. Moreover, the FFD greedy solution [2, 18, 60]is considered in our thesis to compare with the optimal solution results to understand the impact on the results.

After that, influenced by works [55, 81, 147] in considering the reliability of the network, we proposed a stochastic solution (in Chapter 6) to place SFCs that are requested by tenants that only provide the SFC they want to deploy along with the required availability level. However, differently from these works, we consider active-active replication approach for SFC placement such that resources are not wasted for backup and we propose a different placement strategy for them to satisfy the required availability.

In our contributions we consider only physical nodes failure as our targeted network topologies provide more redundancy to physical links comparing with nodes motivated by the related works [20, 113, 193]. However, unlike all previous works, we evaluate our proposed solution on very large topologies, considering real data center topology sizes.

3.2 Big Data Applications

In our information-driven society, data is constantly produced and consumed. It is estimated that by 2020, for every person on earth, 1,7MB of data will be created every second [52]. The term *Big Data* describes innovative techniques and technologies to capture, store and analyze large-size data sets with high velocity and variety. The MapReduce framework [49] proposed for processing big data is composed of a programming model and a runtime environment. The input is typically huge and divided in files called splits. In the first phase, each split is processed by the map function that generates key-value pairs. Then, these outputs are shuffled according to their keys and passed to the reduce tasks that process them again, producing a final result for each key.

Apache Hadoop [75] is an open-source implementation of MapReduce sponsored by Yahoo. The two fundamental sub-projects are the Hadoop MapReduce framework that provides the compute layer and the Hadoop Distributed File System *(HDFS)* [165], which provides the data layer. The architecture is master-slave, composed by the worker's nodes that execute the tasks on the data, and the master that maintains the state of the job, schedules the tasks on the workers and maintains the state of the cluster; it was designed to be crash-fault tolerant. However, in

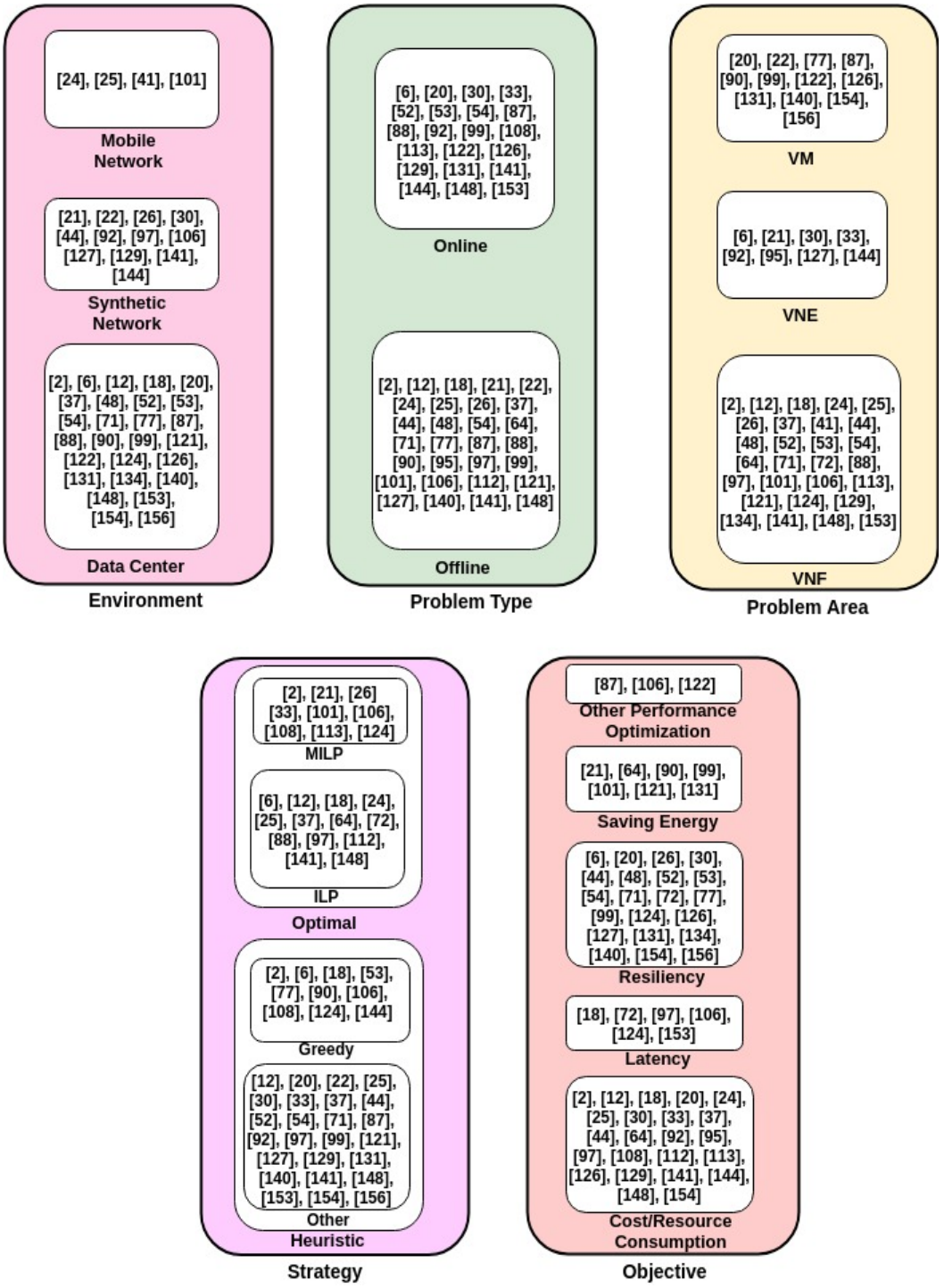


Figure 3.1: Taxonomy of placement efforts founded in the literature.

distributed systems we have to deal with a more complex class of faults such as the Byzantine faults introduced by Lamport [97]. A protocol that tolerates f byzantine attackers without compromising the system correctness is called Byzantine Fault Tolerant *(BFT)*.

Castro et al. [31] described a new replication algorithm, called BFT, which allows systems to tolerate byzantine faults. Their BFT algorithm can be used to implement real services in asynchronous environments such as the Internet. It combines mechanisms to defend against Byzantine-faulty users by proactively recovering replicas. This replication algorithm needs at least $3f + 1$ replicas to tolerate f faulty users [97, 158].

Costa et al. [44] also proposed a replication-based Hadoop MapReduce, namely BFT, which tolerates accidental Byzantines faults, where $f+1$ replicas for map and reduce tasks. However, voluntary malicious attackers in the system are not tolerated. They achieve performance close to the double of the standard Hadoop in normal conditions, without faults. Also, the storage can be made BFT by applying the same principles where a BFT HDFS solution has been proposed in [40].

In consequence, a Byzantine fault-tolerant version of Hadoop is presented [43]. This proposed byzantine fault tolerant MapReduce system tolerates arbitrary faults by executing each task more than once and comparing the outputs. The challenge in this work was to do this efficiently, with $f + 1$ replicas of all map tasks that must complete successfully for reduce tasks to be launched, without the need of running $3f + 1$ replicas to tolerate at most f faulty as in [31, 175].

In recent years, Peer-To-Peer (*P2P*) systems have received more attention from industry and academia. Totally different from the standard client-server architecture, every peer in a P2P system participates in a virtual overlay network while acting as a client and a server. Moreover, each peer in a P2P network is responsible for providing and retrieving data and services to and from other peers in the overlay network without the need for a central server.

However, the applicability of P2P networks for distributed computing can be a challenge as the data is distributed to nodes which are, unfortunately, untrustworthy and unreliable. Thus, we have to operate collaborative services in environment where we have not only byzantine nodes but also rational nodes that only aim at increasing their benefit and potentially deviate from the program. This environment is called BAR (*Byzantine, Altruistic, Rational*) [4].

Aiyer et al. [4] proposed a BAR Tolerant *BART* State Machine Replica *(SMR)* that can tolerate a limited number of byzantine nodes plus an unlimited number of rational nodes. To denounce misbehaving nodes to other correct nodes, they use the concept of proofs of misbehaviour (*POM*). Li et al. also proposed a BART gossip protocol [98] to make a peer-to-peer data streaming application that guarantees predictable throughput and low latency in the BAR model.

Many MapReduce frameworks have been proposed for master-slave opportunistic environments [104, 127] and P2P networks [118, 119]. The MOON system proposed by Lin et al. [104] is designed to support MapReduce jobs on opportunistic environments. It extends the standard Hadoop with adaptive task scheduling algorithms to offer reliable MapReduce services on a hybrid resource infrastructure, where volunteer computing systems are provided by a small set of dedicated nodes. These adaptive task scheduling algorithms distinguish between different types of node outages in order to place tasks and data on both dedicated and volatile nodes.

Furthermore, the work of Fabrizio et al. [118] focus on improving MapReduce implementations for distributed platforms such as Grid or P2P. Knowing that failures are likely to happen since peers join and leave the network at an unpredictable rate. The authors tackled the problems of interrupted peer participation, master failure and job recovery issues of the MapReduce applications. Where, in case of the master failure, the backup master is promoted to the master by the election mechanism of a pool of backup masters.

3.2.1 Conclusion

Byzantine agreement [31, 97] and Byzantine fault tolerant state machine replication have been studied in the literature [4, 31, 158]. Moreover, few works in the literature address the problem of proposing fault tolerance mechanisms on Hadoop MapReduce framework to tolerate system faults that corrupt the results of computation of tasks [40, 43, 45].

Moreover, when running collaborative services in MAD distributed systems, we have not only byzantine nodes but also rational nodes that only aim at increasing their benefit and potentially deviate from the program. Thus, a BAR model is introduced in the literature to address this challenge [4, 41, 98]. However, to the best of our knowledge, there is no collaborative MapReduce framework considering the BAR model in the literature.

4 Accounting for Resiliency in SFC

Contents

When deploying network service function chains, the focus is usually given on metrics such as the cost, the latency, or the energy and it is assumed that the underlying cloud infrastructure provides resiliency mechanisms to handle with the disruptions occurring in the physical infrastructure. In this chapter, we advocate that while usual performance metrics are essential to decide on the deployment of network service function chains, the notion of resiliency should not be neglected as the choice of virtual-to-physical placement may dramatically improve the ability of the service chains to handle with failures of the infrastructure without requiring complex resiliency mechanisms.

4.1 Introduction

To provide the services demanded by their customers, network providers must continually purchase, deploy, and reconfigure middle-boxes, which results in high CAPEX and OPEX and leads to long product cycles to provide these services with a strong dependence on specialized hardware. To provision more rapidly new services while reducing costs, *Network Function Virtualization* (NFV) was proposed [56] and extended by *Service Function Chaining* (SFC) that combines multiple network functions in specific orders. NFV and SFC leverage virtualization to deploy services on commodity hardware hence reducing costs but raising the new challenge of how to provide efficiency and resiliency into the software with shared and error-prone hardware resources [58].

To realize service chains, the placement of the virtual functions onto the physical infrastructure becomes critical as it plays a major role in the performance and robustness of the chain. However, while tremendous efforts have been realized on placing functions to reduce costs and improve performances [102], robustness is often neglected under the cover that the orchestrator can cope with failures.

With this chapter, we advocate that accounting for robustness when placing functions can significantly improve robustness of the chain without increasing the load of the orchestrator. To that aim, we study a reference chain and demonstrate with an exhaustive study how placement in the physical infrastructure can influence the overall robustness of the system even when the virtual chain itself is supposed to be robust to failures.

4.2 Service function chain robustness

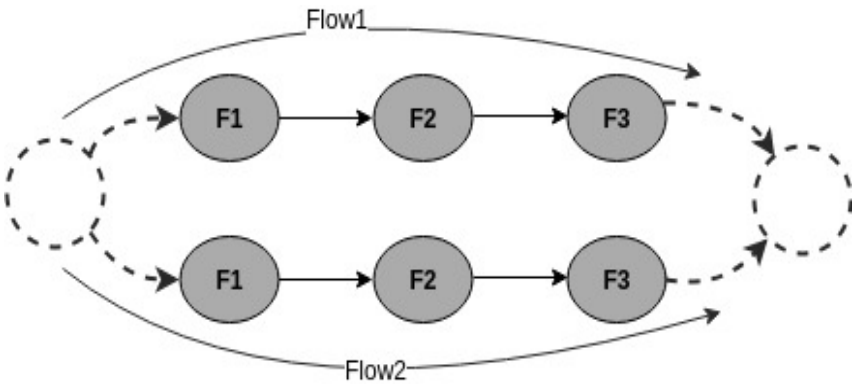
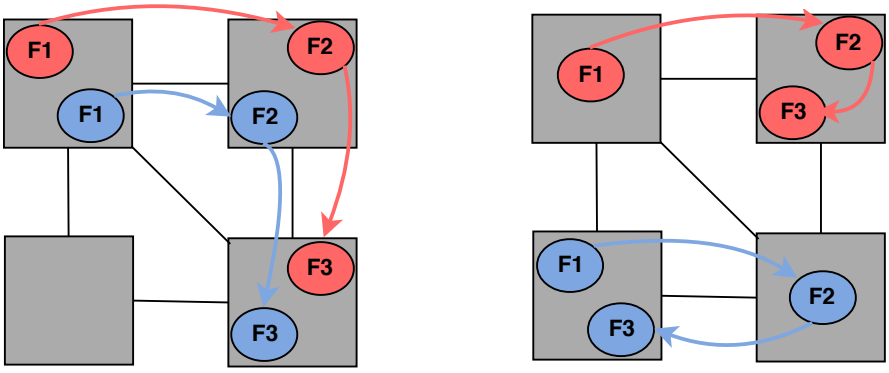


Figure 4.1: Reference chain.



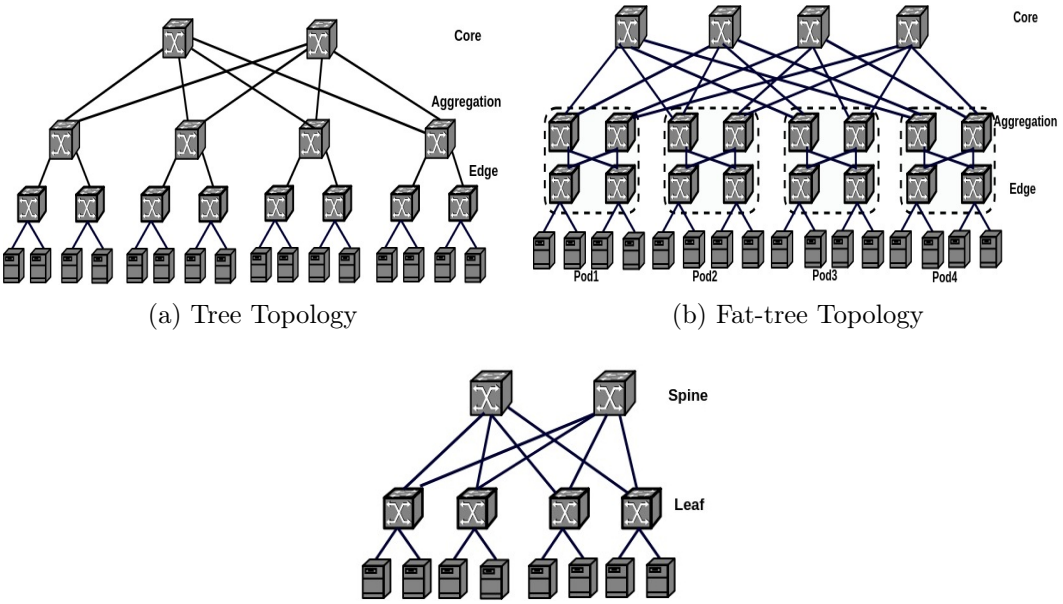
(a) Not robust (b) Robust

Figure 4.2: Examples of placements and their robustness to one failure.

To illustrate the impact of the network functions placement on the resiliency of a chain, we consider the reference chain presented in Fig. 4.1 and composed of two equal sub-chains of 3 functions such that when a flow must be processed by the chain, it can be processed by either sub-chain. The sub-chain that processes a flow is selected by the cloud infrastructure (e.g., load balancer).

This reference chain is general as it is at the same time robust and fragile. Robust as processing can be done by any of sub-chain and fragile as if an element in one sub-chain fails, the whole sub-chain is disrupted. This situation is common when the flow state is mandatory. We assume that the system that decides the sub-chain to

be used is able to determine if a sub-chain is working properly or not while selecting the sub-chain of a flow and that it functions properly.



(a) Tree Topology

(b) Fat-tree Topology

(c) Spine-and-Leaf Topology

Figure 4.3: Reference topologies.

Fig. 4.2 shows an example where the placement of the chain in the physical infrastructure can impact the robustness of the chain to one single failure (node or link).

4.2.1 Simulation environment

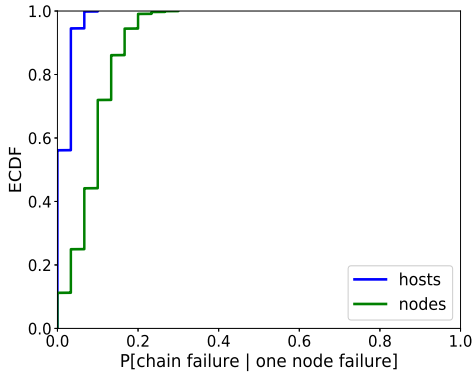
To study the impact of the placement of functions onto the physical infrastructure, we consider all placements of the reference chain presented in Fig. 4.1 onto the following network topologies: (*i*) Tree network topology presented in Fig. 4.3a, (*ii*) Fat-Tree topology with 36 nodes (16 hosts and 20 switch nodes) presented in Fig. 4.3b and (*iii*) Spine-and-Leaf topology with 22 nodes (16 host nodes and 6 switch nodes) presented in Fig. 4.3c [1].

These topologies are intentionally simple, and very common as data-center topologies (more details about these DC topologies are provided in Sec. 2.4.2), to clearly outline the impact of placement on the service robustness. As the number of potential placements is combinatorial, we randomly sampled the set of all solutions to take a total of 16,718 different placements[2].

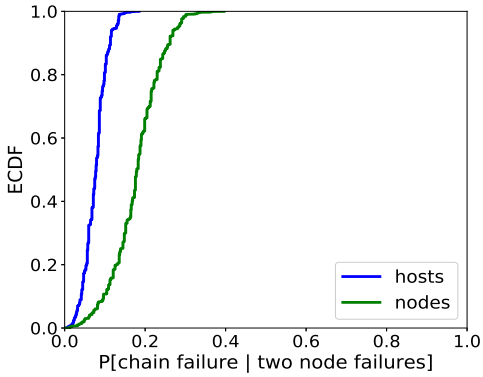
To assess the topological properties of the different placements, we consider three usual metrics: (*i*) the *number of physical hosts* and (*ii*) the *number of Top-of-the-Rack (ToR) switches* of the physical infrastructure involved in the placement, which

[1] As the purpose is to motivate the robustness problem in service function chaining, we do not apply any constraint (e.g., bandwidth) on the placement.

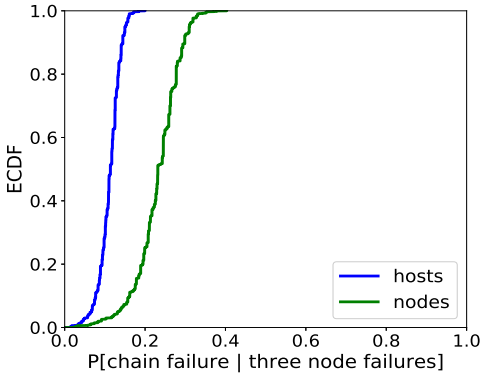
[2] All the code and placement algorithms used in this chapter are available at `https://team.inria.fr/diana/files/2016/05/procon16.zip`



(a) 1 failure



(b) 2 simultaneous failures



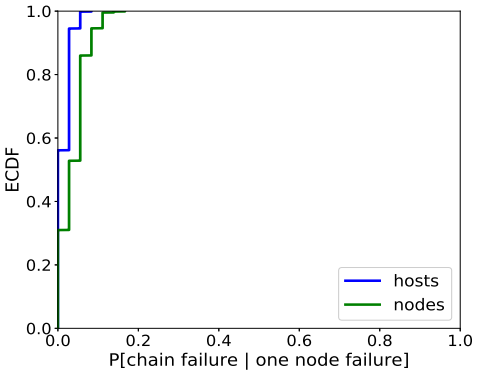
(c) 3 simultaneous failures

Figure 4.4: Probability of chain disruption in case of node failure within Tree network topology.

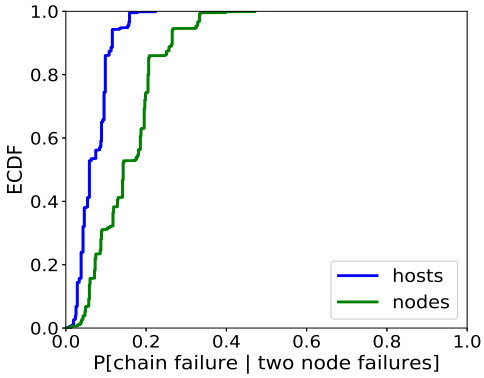
indicates how the placement shares the load and how it is distributed in the infrastructure, and (*iii*) the *number of virtual functions per host* that indicates how processing virtualization is leveraged by the placement.

Considering the topology as a graph, we can identify two categories of failures: the failure of a node (i.e., a computing host or a switch) or the failure of an edge (i.e., a link). To assess the robustness of the placement against failures, we independently considered these two categories and exhaustively tested every combination of n-failures of elements of the category (i.e., n nodes or n links).

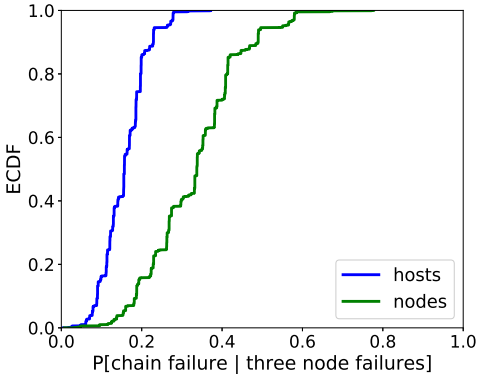
A placement is considered as n-robust if a flow can be processed by the service chain for any failure of n physical elements of the network. However, by the construction of our chain, there exists always at least one case where the chain is not robust: if several failures happen simultaneously (i.e., $n \geq 2$). For this reason, we rather consider the probability of the chain to fail in case of n simultaneous failures when failures are independent and equiprobable.



(a) 1 failure



(b) 2 simultaneous failures



(c) 3 simultaneous failures

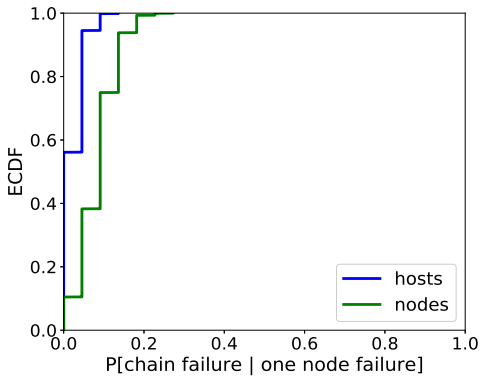
Figure 4.5: Probability of chain disruption in case of node failure within Fat-Tree network topology.

4.2.2 Robustness analysis

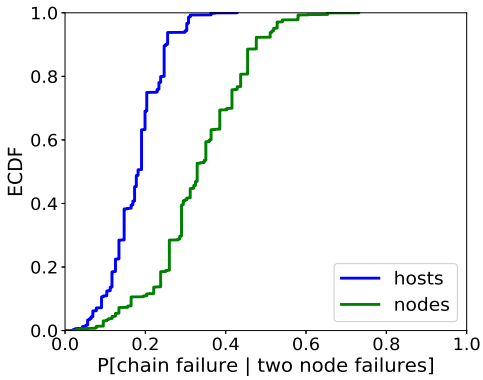
Overall, no strong linear correlation exists between topological and placement properties in terms of robustness. Nevertheless, the probability that the chain fails is weakly correlated with the number of functions deployed on one physical host (correlation is 0.68 for Tree topology) as it is sufficient that at least one function of each sub-chain is deployed on the same host to break the chain in case of one single failure.

This conclusion is confirmed by Figures 4.4, 4.5 and 4.6, where x-axis is the probability of chain being disrupted while y-axis is the empirical cumulative distribution function (*ECDF*) of having a chain disruption in case of node failures. The ECDF represents the proportion of observations that are less or equal to the corresponding the probability of chain disruption value on the x-axis.

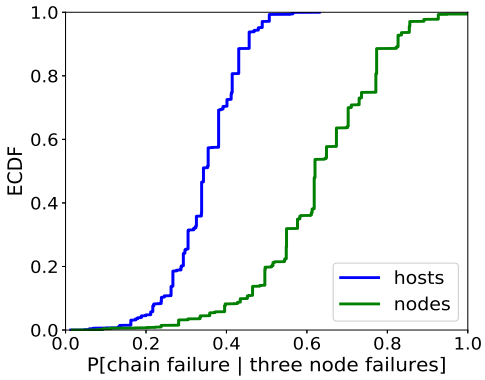
Here, we consider the failure of a node in the topology in general (i.e., including switches and computing hosts) and the cases accounting only for the failure of a computing host. All these figures indicate that the robustness of the chain is less sensitive to the failure of computing hosts than to the failure of switches. This is



(a) 1 failure



(b) 2 simultaneous failures



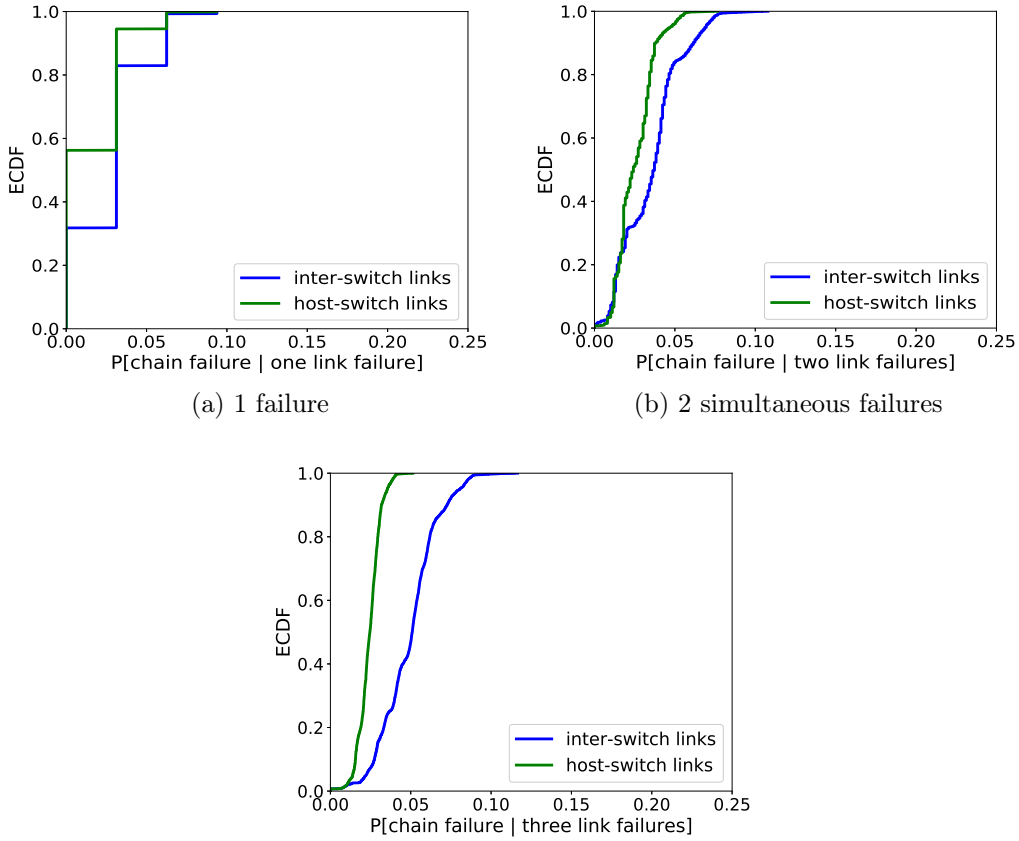
(c) 3 simultaneous failures

Figure 4.6: Probability of chain disruption in case of node failure within Leaf-and-Spine network topology.

because each host is connected to the network without redundancy via a ToR/leaf switch. Therefore, the failure of one single ToR/leaf switch breaks the entire chain even if it is deployed on different physical hosts. On the contrary, to disrupt the chain with the failure of a host, it is necessary that at least one function of each sub-chain is deployed on the same physical host.

Moreover, one can note the behavior of the Fat-Tree network is roughly similar to one obtained from the Tree topology, even though for only one single node failure, Fat-Tree topology shows less sensitivity than the Tree topology as there is more redundancy in the edge and aggregation layers in the Fat-Tree network (See Figures 4.5a and 4.4a). However, the two-tier Spine-and-Leaf topology shows more vulnerability to failure than the three-tier network topologies especially when the number of simultaneous failures increases as shown in the Figure 4.6c.

Similar conclusions can be drawn while considering link failures instead of node failures (see Figures 4.7, 4.8 and 4.9) where we distinguish failures of inter-switch links and host-switch links. We can observe that backbone link failures impact the robustness of the chain, particularly the failure of inter-switch links as actually losing



(a) 1 failure

(b) 2 simultaneous failures

(c) 3 simultaneous failures

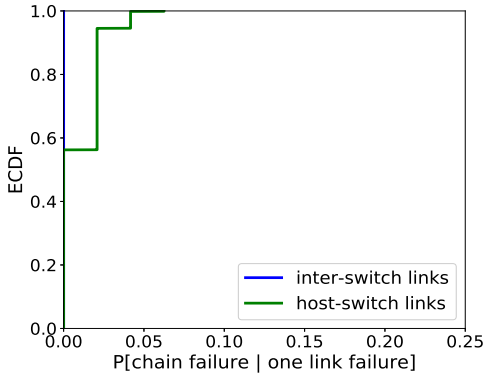
Figure 4.7: Probability of chain disruption in case of link failure within Tree network topology.

the connectivity of ToR switches may impair the chain, regardless of the computing redundancy.

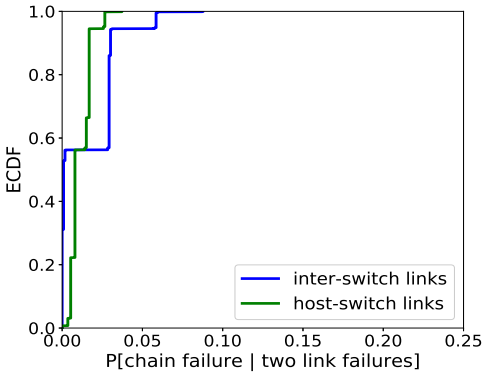
As illustrated by the all figures, the overall robustness behavior is the same regardless of the number of simultaneous failures except that the likelihood of having a chain disruption increases with the number of failures. However, we can notice that link failures have less impact on the service continuity comparing with the physical node failures as we have more links redundancy in the reference topologies.

4.3 Discussion

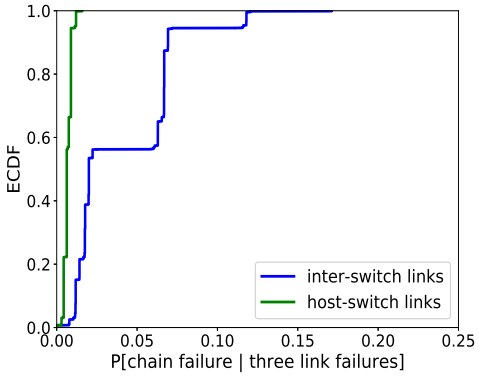
In this chapter, we advocate that the choice of the placement of virtual functions constituting chains onto a physical infrastructure should not neglect the robustness of the chain as while some placement may offer good performance, they may be very fragile to failures of a physical component. In Sec. 4.2.2 we exhaustively studied the impact on the robustness of the placements of a reference chain within three different data center topologies. We have seen that even though a chain is logically robust,



(a) 1 failure



(b) 2 simultaneous failures

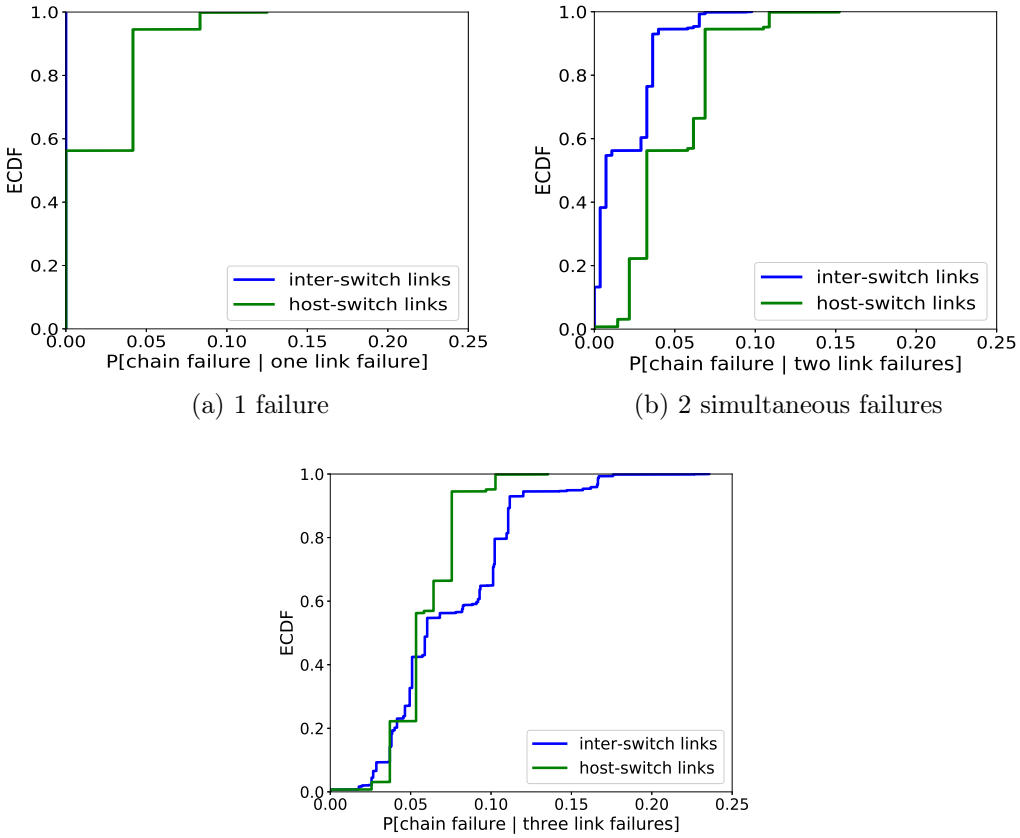


(c) 3 simultaneous failures

Figure 4.8: Probability of chain disruption in case of link failure within Fat-Tree network topology.

if robustness is not accounted while deciding the placement it may be broken by a single failure.

One may argue that the failure of physical infrastructure elements is not an issue as recovery mechanisms are implemented by the orchestrator. What we answer is that if these mechanisms are mandatory to make the system truly resistant, they remain slow as they require VM migrations. On the contrary, a wise selection of placement permits to be robust to usual simple failures of the physical infrastructure without having to trigger migrations and thus reducing the stress on the orchestrator that would then be used only for complex situations.



(a) 1 failure

(b) 2 simultaneous failures

(c) 3 simultaneous failures

Figure 4.9: Probability of chain disruption in case of link failure within Leaf-and-Spine network topology.

5 Online Robust Placement of Service Chains for Large Data Center Topologies

Contents

The trend today is to deploy applications and more generally SFCs in public clouds, instead of using dedicated infrastructures where software, hardware, and network components are managed by the same entity. However, by moving their services to the cloud, the users lose their control on the infrastructure and hence on the robustness, which makes challenging to guarantee the robustness as with dedicated infrastructures.

In the previous chapter, we showed that the choice of the placement of service functions onto a physical infrastructure should not ignore the robustness of the chain as some placement decisions may be very fragile to failures of physical components. Based on the proposed study on the robustness of the placements of a reference chain in a tree topology, we have seen that even though a chain is logically robust, if placement choice was not smart enough, this chain may be broken by a single failure.

Inspired by the results from the study presented in the previous study, in this chapter, we provide an online algorithm for robust placement of service chains in data centers. Our placement algorithm determines the required number of replicas for each function of the chain and their placement in the data center. Our simulations on large data-center topologies with up to 30,528 nodes show that our algorithm is fast enough such that one can consider robust chain placements in real time even in a very large data center and without the need of prior knowledge on the demand distribution.

5.1 Introduction

Digital services and applications are nowadays deployed in public virtualized environments instead of dedicated infrastructures. This change of paradigm results in reduced costs and increased flexibility as the usage of the hardware resources can be optimized in a dynamic way, and allows one to build the so-called *Service Function Chains* (SFCs) [78].

Conceptually a "cloud" provides one general-purpose infrastructure to support multiple independent services in an elastic way. To that aim, cloud operators deploy large-scale data centers built with commercial off-the-shelf (COTS) hardware and make them accessible to their customers. Compared to dedicated infrastructures, this approach significantly reduces costs for the operators and the customers. However, COTS hardware is less reliable than specific hardware [125] and its integration with software cannot be extensively tested, resulting in more reliability issues than in well-designed dedicated infrastructures. This concern is accentuated in public clouds where resources are shared between independent tenants, imposing the use of complex isolation mechanisms. As a result, *moving Service Function Chains to data centers calls for a rethinking of the deployment model to guarantee high robustness levels.*

In the context of exogenous independent service chains requests in a very large data center (DC), it is particularly complex for an operator to dynamically place the different virtual functions constituting the chains in their data center, while guaranteeing at the same time robustness to their customers' services and maximization of the number of services that can be deployed in their infrastructure. The reason is that operators have no view on the future requests they will receive and how long services deployed at a given moment will last, as the demand is elastic by nature. *The key challenge is to design placement algorithms in large data centers given the unknown nature of the future service chain requests and the need to make the placement decisions on the fly.*

In this chapter, we provide an online optimization algorithm that builds active-active chain replicas placements such that in case of fail-stop errors breaking down some function instances, the surviving replicas can be used to support the traffic normally carried by the failed functions. Our algorithm computes the number of replicas needed to be robust to R arbitrary fail-stop node failures and where to place them in the underlying data center.

The salient contributions of this chapter are the following:

- *Fast approximation online algorithm (§5.3)* We propose an online two-step approximation algorithm for very large data centers that determines the optimal number of service VNF instances and their placement in DCs, based on the available network resources and the resource requirements of the tenants service requests, namely CPU and bandwidth.
- *Evaluation at very large scale (§5.4)* We provide a comprehensive evaluation of the proposed algorithm using three large DC topologies: (*i*) a 48-Fat-Tree topology with 30,528 nodes, (*ii*) a Spine-and-Leaf topology with 24,648 nodes, and (*iii*) a generic two-layer topology with 27,729 nodes. To the best of our knowledge we are the first to demonstrate that robust placement algorithms can be used in practice in very large networks.

The chapter is organized as follows. Sec. 5.2 states the problem addressed with this chapter, our approach, and our assumptions. Sec. 5.3 details our algorithm to deploy SFC with robustness guarantees on DC topologies and Sec. 5.4 assesses its performance on very large-scale networks with simulations. Finally, Sec. 5.5 concludes the chapter.

5.2 Problem Statement

This chapter aims at providing a mechanism to deploy Service Function Chains (SFCs) in large public cloud data centers in a way that guarantees that the deployed SFCs cannot be interrupted upon node failures. In the context of public cloud data centers, the infrastructure operator does not control the workload and the placement must be oblivious to the future workload as it is unknown. When a tenant requests the placement of a chain in a data center, it provides its requirements in terms of VMs (e.g., VM flavor in OpenStack) and its desired availability SLA (see Sec. 5.2.4).

To address the so-called *robust SFC placement in large data centers*, we propose to develop an online optimization algorithm that builds active-active chain replicas placements. The placement must be such that up to R arbitrary fail-stop errors no deployed service would be interrupted or degraded.

The target of our algorithm is to maximize the overall workload that a data center can accept such that service requests are always very likely to be accepted, even though they are unknown in advance. In other words, we aim at optimizing the SFC request acceptance ratio.

As our algorithm aims to be used in an online manner, its resolution time must be kept fast. Namely the resolution of a SFC placement must be done in a time no larger than the one required to instantiate the SFC functions in the infrastructure (i.e., the order of a few tens of seconds) even for large data center topologies (more than 30,000 physical nodes).

We develop a two-step approximation algorithm that first computes the optimal placement of functions on the DC nodes regardless of the link constraints. It then

computes the routing table for the traffic carried by the SFC, using a feasible shortest path between functions.

5.2.1 Assumption

In the following of this section we detail the assumption we took to address the problem of robust SFC placement in large data centers.

5.2.1.1 Environment: Data Center Topologies with Fault Domains

In this chapter, we consider the common case of multi-tier DC topologies [38] decomposable in fault domains such as Fat-Tree or Spine-and-Leaf topologies (more details about DC topologies are provided in Sec. 2.4.2).

Fat Tree (see Figure 5.1) is a common bigraph based three-tier topology for data centers [5]. The elementary block in this topology is called *pod* and is a collection of access and aggregation switches connected in a complete bigraph. Each pod is connected to all core switches. Fat Trees are clos topologies relying on high redundancy of links and switches.

Spine and Leaf [10, 39] (see Figure 5.2) are common two-tier topologies in data centers, where each lower-tier switch, called *leaf* switch, is connected to each of the top-tier switches, named *spine* switches, in a full-mesh topology. In Spine-and-Leaf networks groups of servers are connected to the leaves.

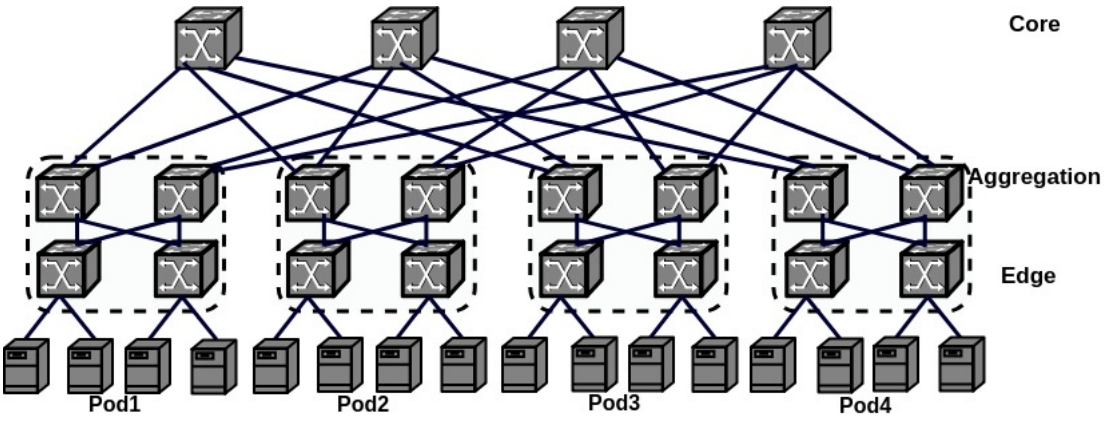


Figure 5.1: Fat-tree Topology.

When network reliability and availability are considered at the early network design phases, topologies are built with multiple *network fault domains*. A fault domain is said to be a single point of failure. It represents a group of machines that share a common power source and a network switch and it is defined based on the arrangement of the hardware. A machine, rack or pod can be a fault domain.

In tree-based, switch-centric DC network topology such as Fat Tree and Spine and Leaf [10], we can define the fault domains easily. In Fat-Tree topologies, each pod is

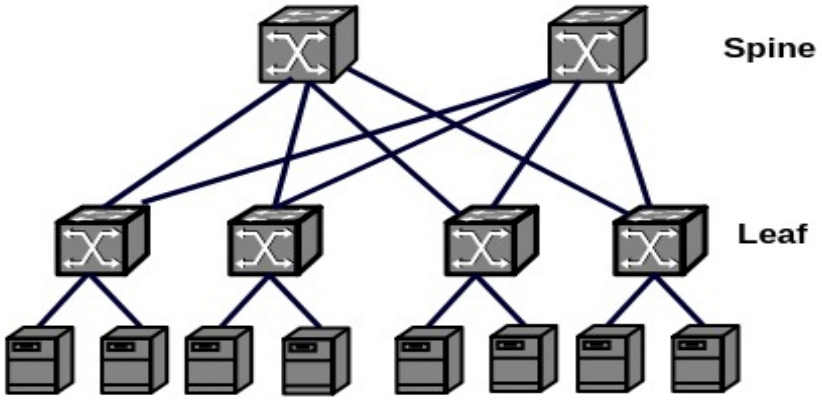


Figure 5.2: Spine-and-Leaf Topology

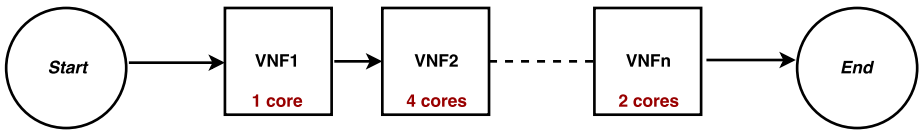


Figure 5.3: SFC Request Topology.

considered as one fault domain. In Spine-and-Leaf topologies, each leaf switch and the hosts connected to it form a fault domain.

On the contrary, it is not possible to define fault domains in server-centric DCs – such as Dcell and Bcube [183]. We therefore do not consider such topologies in our work.

5.2.2 Service Function Chains independence and workload

Public cloud DCs share the same physical infrastructure with heterogeneous services operated by multiple tenants. In this chapter, we consider the case of tenants willing to deploy SFCs in the DC. An SFC is a group of virtual functions ordered in sequences to provide a service to an end user. Each function uses the output of the previous one in the chain as an input [78].

An SFC can be represented as a directed graph. Each node represents a virtual function annotated with its resource requirements (e.g., CPU, memory, etc.) while each edge represents a virtual link *vLink* annotated with its requirements (e.g., bandwidth). A virtual link logically represents the flow of traffic between two functions where the destination node (i.e., function) consumes the traffic generated by the origin node (i.e., function). If no traffic is directly exchanged between two functions, no vLink is defined. While in the general case SFCs can be arbitrary directed graphs, we restrict our work to the common case of directed acyclic graphs [110].

In this work, each function is dedicated to only one SFC, and an SFC is under the sole control of a single tenant. This assumption holds in case of public clouds, where

tenants are independent actors and the DC operator considers functions as opaque virtual machines. If a function is implemented by using multiple instances of the same VM (e.g., because of processing limitations of a single host), we assume that the load is equally and instantaneously balanced between all the function instances, e.g., through LBaaS in OpenStack.

To preserve performance while sharing the same physical hosts between many tenants, the total amount of the physical host resources is always larger than the sum of the used resources by various VMs deployed on that host.

As we do not consider the deployment phase of SFCs and given that we consider Fat-Tree and Spine-and-Leaf topologies in this chapter, we can safely assume that the network provides infinite bandwidth w.r.t. SFCs demands.

5.2.3 Online Placement

In some specific private cloud deployments, one can control the workload and thus apply offline optimization techniques to decide on the placement of virtual service chain functions in the data center. However, in the general case of a public cloud, the workload and the management of the infrastructure are handled by different independent entities (i.e., tenants and the cloud provider). As a result, the placement of SFCs must be determined in an online manner that is oblivious to future demands.

5.2.4 Robustness and Failure Model

We target the placement of SFCs with robustness guarantees, where the k robustness level stands for the ability of an SFC to remain fully operational upon the failure of k entities of the infrastructure and without having to re-deploy or migrate virtual machines upon failures in order to guarantee zero downtime.

When a tenant requests the placement of a service function, it provides the service function graph with its required resources – the VM flavor for each chain function – and the SLA commitment for the chain (e.g., five nines).

Assuming a strict *fail-stop failure model* [157] with uncorrelated events and given the knowledge of its infrastructure (MTBF and MTTR of the physical equipment), the SFC graph and subscribed duration, and the requested SLA commitment [132], the data center operator can determine the maximum number of concomitant physical node failures that the chain may encounter during its lifetime.

5.3 SFC Placement with Robustness

In this section, we propose a two-phase algorithm to place SFCs in a DC such that whenever a chain is deployed, it offers robustness guarantees. To avoid downtime upon failures in the physical infrastructure, we cannot rely on a reactive approach that would redeploy functions after failure detection [117]. Instead, we propose to

account in advance for the potential fail-stop node failures that the chains may encounter during their life cycle.

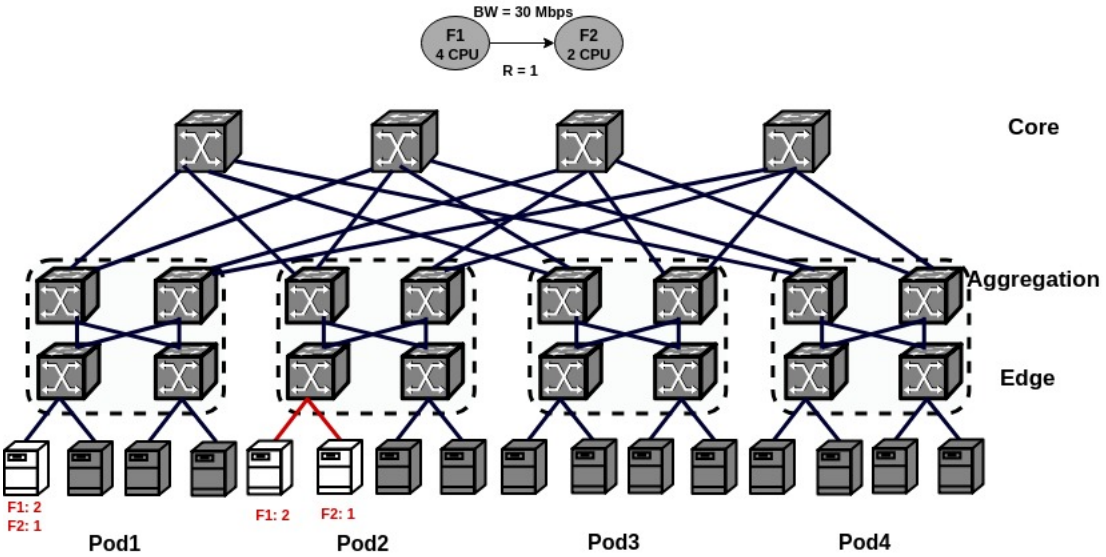


Figure 5.4: Initial placement. Link capacity: 20 Mbps, Core per hosts:3, where (Fx: y) → (*Function name* : *Required CPU Cores*)

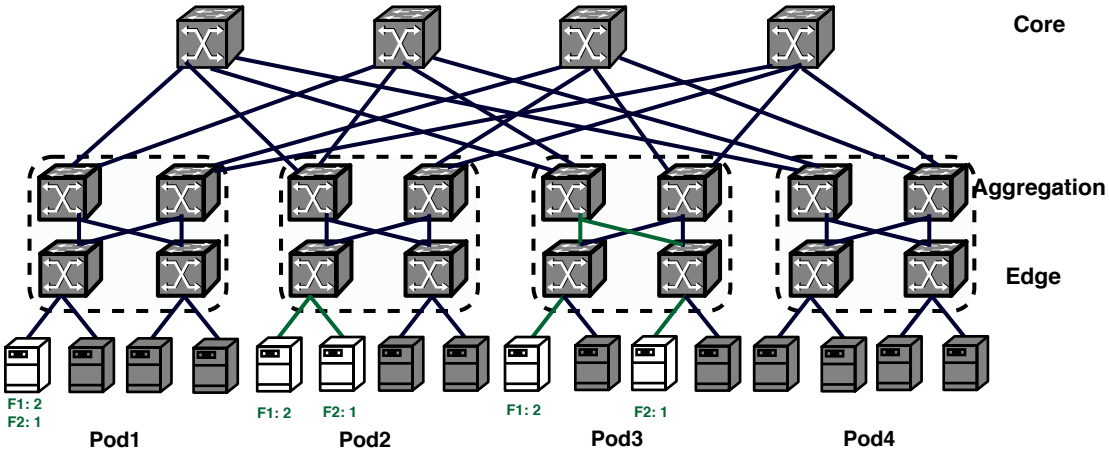


Figure 5.5: Final placement. Link capacity: 20 Mbps, Core per hosts:3, where (Fx: y) → (*Function name* : *Required CPU Cores*)

To that aim, our algorithm *replicates* multiple times the chain and *scales down* each replica such that each replica has an equal fraction of the total load of the initial chain. In the remaining of this chapter, we refer to such scaled down replicas with the term *scaled replica*. Our algorithm is called each time a request to install an SFC is received. Specifically, for a *robustness level* R, the algorithm determines how many scaled replicas to create for that SFC and where to deploy them within the data center such that the chain will be robust to at least R simultaneous fail-stop node failures without impairing the robustness guarantees of the chains already deployed. In other words, even if R nodes fail, every chain deployed in the data center will keep working at its nominal regime.

To guarantee the isolation between scaled replicas of a chain, each replica of a chain is deployed in a different fault domain [100]. Also, as we assume a fail-stop failure model, at least $R+1$ scaled replicas are needed to be robust to R failures. At first, the algorithm creates $R+1$ scaled replicas[1] and tries to find $R+1$ fault domains able to host the scaled replicas. If no solution exists, the algorithm is repeated for $R+2$, $R+3$,..., *max iteration* replicas until a solution is found. If a solution is found, the scaled replicas can effectively be deployed in the data center.

To determine whether a placement is possible for a given robustness level, the algorithm considers the *normal committed resources*, i.e., the minimum resources (e.g., cores, memory) that a compute node must dedicate to guarantee proper functioning under normal conditions (i.e., no failures) and the *worst-case committed resources*, i.e., the minimum number of resources required on compute nodes to guarantee proper functioning upon R simultaneous compute node failures impacting the chain.

Figure 5.4 and Figure 5.5 illustrate the behavior of the algorithm with the deployment of a chain composed of two functions: the first function requires 4 cores, while the second one requires 2 cores and the flow between these two functions requires 30 Mbps to properly work, with the target of being robust to one node failure. Figure 5.4 shows the placement of the chain in a Fat-Tree data center where each node in the DC has 3 cores. Each replica receives 50% of the expected load, shown in red (e.g., Function F1 in each replica needs 2 CPU cores instead of 4 CPU cores). After checking the robustness, the algorithm decides to split the chain since the first placement does not meet the robustness requirements because the worst-case commitment is not respected, as one failure would result in the need of 4 cores on the remaining hosts.

Moreover, the physical network links cannot support more than 20 Mbps while in the worst-case the link requirement is 30 Mbps. The placement in Figure 5.5, depicted in green, meets the robustness level as when a node fails, one of the scaled replica will fail but the other replicas will be able to temporarily support the whole load of the failed one as in the worst case each link needs to hold 5 Mbps more.

In order to find such a placement, we propose a two-step algorithm, listed in Algorithm 1. In the first step, `SolveCPU(`G`,` C`,` R`,` M`)` solves the problem of placing the service function chain C on the DC topology G taking into account the required robustness level R and the functions CPU requirements (see Sec. 5.3.1). If the solution is empty, this means that no function placement can be found and the SFC request will be rejected. Otherwise, the result of this step corresponds to the set of mappings associating replica functions and the compute nodes on which they have to be deployed. In the second step, the obtained solution will be used as an input of Algorithm `SolveBW(`G`,` C`,` *CPU-Placement*`)` in which each *vLink* is mapped to one or more physical link(s), called *path(s)*, according to the bandwidth requirements, see Sec. 5.3.3. If all vLinks can be mapped, the service will be accepted and deployed on the DC network. Else, the service request will be rejected as the bandwidth requirements cannot be satisfied.

[1] Each scaled replica is in charge of $\frac{1}{R+1}$ chain load.

Algorithm 1: Robust placement algorithm

Input: Physical network Graph: G
C $\in$ Chains
Robustness level: R
Maximum number of replicas: max_iterations

M = max_iterations
CPU_Placement= `SolveCPU(`G, C, R, M`)`
if *CPU_Placement*=ϕ **then**
  `Error(`*Impossible to place the chain NFs*`)`
else
  BW_Placement= `SolveBW(`G, C, *CPU_Placement*`)`
  if *BW_Placement*=ϕ **then**
    `Error(`*Impossible to place the chain vLinks*`)`
  else
    `deploy(`G, *CPU_Placement*, *BW_Placement*`)`

Algorithm 2: *SolveCPU* algorithm

Input: Physical network Graph: G
C $\in$ Chains
Scaled chain replica graph: SC
Robustness level: R
Maximum number of replicas: M

n = R + 1
CPU_Placement = ϕ
while *CPU_Placement* = ϕ *and* $M > 0$ **do**
  SC = `scale_down(`C, n`)`
  CPU_Placement = `solve_placement(`SC, G, n`)`
  n = n + 1
  M = M -1
`return(`*CPU_Placement*`)`

5.3.1 Node placement

In the function placement step (see Algorithm 2), the `solve_placement(`SC`,` G`,` n`)` function considers two graphs: the DC topology graph G and the scaled replica graph SC where the `scale_down(`C`,` n`)` function computes the scaled replica scheme, i.e., an annotated graph representing the scaled down chain, for a chain C if it is equally distributed over n scaled replicas (see Sec. 5.3.2). The goal of the function `solve_placement(`SC`,` G`,` n`)` is to project n function replicas of the scaled replica graph S on the topology graph G with respect to the physical and chain node constraints.

Algorithm 3: ***SolveBW*** algorithm

Input: Physical network Graph: G
C $\in$ Chains
CPU_Placement: Placement of SFCs nodes

BW_Placement = ϕ
foreach *replica_placement* $\in$ *CPU_Placement* **do**
  foreach *vLink* **do**
    paths = `all_shortest_paths(`G, SC, D`)`
    path = `valid_path(`*paths*`)`
    if *path* $\neq \phi$ **then**
      BW_Placement $\uplus$ path
    else
      BW_Placement = ϕ
      break
`return(`$BW\_Placement$`)`

For each fault domain, `solve_placement(`SC`,` G`,` n`)` tries to find a solution for the linear problem defined in Sec. 5.3.1.1, which aims at finding a placement for the scale replica graph in the fault domain while respecting VNFs requirements. If there are at least n fault domains with a solution to the problem, then any n of them is a solution to our robust placement problem. Otherwise, no solution is found and an empty set is returned.

5.3.1.1 ILP Approach

The online robust placement problem can be formulated as an Integer Linear Programming (ILP).

Given the physical network undirected graph $G = (V, E)$ and the service function chain directed graph $C = (V^{'}, E^{'})$. Table 5.1 summarizes all the variables that define the problem and other variables used in our model formulation to place one particular service chain.

To solve the placement problem, we introduce two binary decision variables of different types:

(1) Bin used variables. $u(h)$ indicates whether physical host h is used.

(2) Item assignment variables. $m_{f,h}$ indicates whether function f is mapped to physical host h.

Parameter	Description
G	G = (V, E) Undirected graph that represents the physical network
V	Set of physical nodes V= S ∪ H, where S represents the switch nodes for routing and H stands for the nodes with computational resources used to host service functions
E	Set of physical links with available bandwidth resources
C	C = (V', E') Directed graph that represents the SFC requested by tenants
SC	Scaled graph, similar to C graph but with scaled resources
V′	Set of nodes representing virtual functions with computational resources requirements
E′	Set of virtual links with bandwidth requirements
H	Set of compute hosts h
F	Set of virtual functions f of the SFC to place
A	Set of start/end points for SFC requests
$CPU(h)$	Number of available CPU cores on the physical host node $h \in H$: $CPU(h) \neq 0$
$CPU(f)$	Number of CPU cores required by the chain function $f \in F$: $CPU(f) \neq 0$, while $\forall a \in A : CPU(a) = 0$
$CPU_R(h)$	Number of remaining CPU cores on the host node h after placement
$u(h)$	Binary variable for physical host node assignment: $\forall h \in H, u(h) = 1$ if host h is used and u(h) = 0 otherwise
$m_{f,h}$	Binary variable for chain function f to host node h mapping $\forall h \in H, \forall f \in F, m_{f,h} = 1$ if function f mapped to h and $m_{f,h} = 0$ otherwise.
T_{IA}	Mean inter-arrival time of chain placement requests
S	Mean service time in which chain remains in the system
R	Required robustness level (i.e., maximal number of simultaneous physical failures allowed in the system)

Table 5.1: Notations used in the chapter

5.3.1.2 ILP Formulation

Objective:

$$\max \min_{\forall h \in H} (CPU_R(h)) \tag{5.1}$$

Subject to:

Assignment constraints:

$$\forall f \in F, \qquad \sum_{h \in H} m_{f,h} = 1 \tag{5.2}$$

$$\forall h \in H, \quad u(h) = 1 \text{ if } \sum_{f \in F} m_{f,h} \geq 1 \tag{5.3}$$

Capacity constraints: $\forall h \in H$,

$$\sum_{f \in F} m_{f,h} \,.\, CPU(f) \leq CPU(h) \tag{5.4}$$

$$CPU_R(h) = CPU(h) - \sum_{f \in F} m_{f,h} \,.\, CPU(f) \tag{5.5}$$

5.3.1.3 ILP Explanation

Normally, to implement their policies, operators must define their objective function; for example, service providers may want to reduce the placement cost or the energy consumption by minimizing the number of used hosts involved in the placement.

For our model, the optimization objective presented in Equation 5.1 aims at maximizing the minimum remaining CPU resources on each physical host in the network. This objective corresponds to spreading the load over all the hosts in the DC.

Constraint (5.2) guarantees that each virtual function is assigned only once while Constraint (5.3) accounts for the used hosts. Constraints (5.4) and (5.5) ensure that hosts are not over-committed and account for their usage, where $CPU(h)$ is the amount of available CPU cores of the physical host (h) and $CPU(f)$ is the number of CPU cores required by function (f).

5.3.2 Replication Model

When a new SC request is received, in order to fulfill its required robustness level, the chain is replicated in additional chains; each one is called a *scaled replica*. The idea behind replication is to exactly replicate the functionality of a chain such that the load can be balanced equally among all replicas. Each replica requires only a fraction of the initial required resources. More precisely, each scaled replica requires $\frac{1}{n}$ of the resources of the main chain if the chain has been replicated n times.

The `scale_down(`C`,` n`)` function computes an annotated graph representing the same graph as C but where the resources associated to each node and link have been scaled down by a factor n. It is worth noting that some resources are discrete or cannot go below some threshold, meaning that the function may not be linear. For example, if the unit of core reservation is 1 core, then scaling down 3 times a resource that requires 2 cores will result in requiring 1 core on each replica.

5.3.3 vLink placement

The *BW_problem* (see algorithm 3) represents the last step in our placement process. Its objective is to map virtual links to actual network paths, based on the placement of virtual network functions obtained from the *CPU_placement* step.

For each virtual link between two functions in each service scaled replica, it retrieves all the shortest paths between the source and the destination physical servers that host these two functions (i.e., the traffic traversing a vLink may cross several physical links). Among these shortest paths, the `valid_path(`*paths*`)` function tests the shortest paths randomly in order to find one path that can hold the required traffic. Thus, for each vLink it tries to find one valid shortest path.

If none exists, it returns an empty set, which means that the placement will be rejected. Else, this accepted path will be appended to the list of accepted paths. The set of vLinks placement (*BW_Placement*) is returned so that the chain can ultimately be deployed by using the `Deploy(G, CPU_Placement, BW_Placement)` function (i.e., virtual functions are instantiated and network routes are installed in the switches).

5.3.4 Discussion

Defining the optimal of an online problem is always a challenge as it potentially requires solving at any time t a problem whose optimal depends on time $t' > t$ for which the knowledge is incomplete or absent.

In this chapter we aim at finding, in an online manner, placements for SFCs in large data centers that guarantee robustness and with the objective of maximizing the SFC request acceptance ratio. Our problem is a variation of the online job shop scheduling problem with multiple operations (i.e., functions) per job (i.e., SFCs) for a number of resources > 2 (i.e., servers and links), with penalties and unknown jobs arrival

and duration. This particular problem is reputed to be NP-complete [67,90,172]. To the best of our knowledge, no bounded heuristic is known for this problem.

We approximate this problem with a two-step algorithm to be executed at each SFC request arrival. The first step finds an optimal feasible placement for the different constituting functions of the service function chain within one fault domain. A feasible placement is a placement for which there is no over-commitment of CPU cores on the server (i.e., a function never shares a core with another function and the number of consumed cores on a server does not exceed the number of cores of the server) and an optimal placement is a placement for which each server maximizes its number of available cores for future potential function placements.

Even though this is a variation of the Knapsack problem, which optimization is NP-hard, in practice as chains are small and as fault domains do not face high contention situations, finding the optimal is feasible in short time (see Sec. 5.4 for practical examples on very large data centers). Once the placement of functions is decided at the first step, regardless of the network situation, a feasible path is decided in the second step of the algorithm in polynomial time using shortest path computation exploration.

It is worth it to mention that our approximation algorithm does not guarantee to maximize the acceptance ratio of SFC requests. However, it approximates it by ensuring that after each placement, each server will offer the maximum number of free CPU cores. In tight scenarios with high contention, this would be far from optimal. However, in practical cases with limited resource contention, this approach offers both good acceptance ratios and acceptable computation times, as demonstrated in Sec. 5.4.

5.4 Evaluation

In the following we evaluate the robust SFC placement algorithm introduced in Sec. 5.3.

5.4.1 Simulation Environment

We have implemented a discrete event simulator in Python.[2] In the evaluation, requests to deploy a chain are independent and follow an exponential distribution of mean T_{IA}, where T_{IA} is the mean inter-arrival time of chain placement requests (measured in arbitrary time unit). Service function chains have a service time of S time units, i.e., the time the chain remains in the system is randomly selected following an exponential distribution of mean S. An SFC that cannot be deployed in the topology is lost, i.e., there is no further request for the rejected chain. In total, our synthetic workload for the simulations contains 1,000 service request arrivals made of 20 arbitrary chains.

[2]All the data and scripts used in this chapter are available on https://team.inria.fr/diana/IEEEAccess/.

In the simulations, every SFC forms a linear chain of functions put in sequence. Each chain has one single starting point and one single destination point. The number of functions between the two endpoints is selected uniformly between 2 and 5, based on typical use cases of networks chains [107], and the requirements of each function in terms of cores is 1, 2, 4, or 8 inspired by the most common Amazon EC2 instance types [8]. Each vLink consumes 250 Mbps.

Simulations are performed on the three following topologies: (i) 48-*Fat-Tree topology*, with 48 pods, each of them having 576 hosts for a total of 27,648 hosts; (ii) *Spine-and-Leaf topology*, a network with 48 leaf switches directly connected to 512 hosts for a total of 24,576 hosts, and *generic topology*, which is built from 54 switches connected to each other and each one of them is connected to 512 host nodes. Each switch represents one fault domain with a total number of 27,648 hosts in this topology. The three topologies are representative of today's data centers and are directly comparable (they have either the same number of fault domains, or the same number of hosts and cores). Resources are homogeneous in the topologies: all hosts have the same number of cores (4 cores per host); all links between aggregation and core switches in the Fat Tree and between leaf and spine switches are 10 Gbps links; and hosts are connected to their ToR/leaf switch through a 1 Gbps link.

To ensure that we are not studying transient results with the workload, we verified that the whole system is in steady state before running a workload of 1,000 service requests. We fixed T_{IA} to the value 0.01 such that in the ideal case, the *Fat-Tree* topology would be loaded at about 90%. Because of space limitations, we fixed R to be equal for each chain in a run, however the algorithm allows using a different value of R for each chain. Our simulations have been performed in Grid'5000.[3] In addition, all the following experiments were repeated 10 times using ten different workloads with the same parameters.

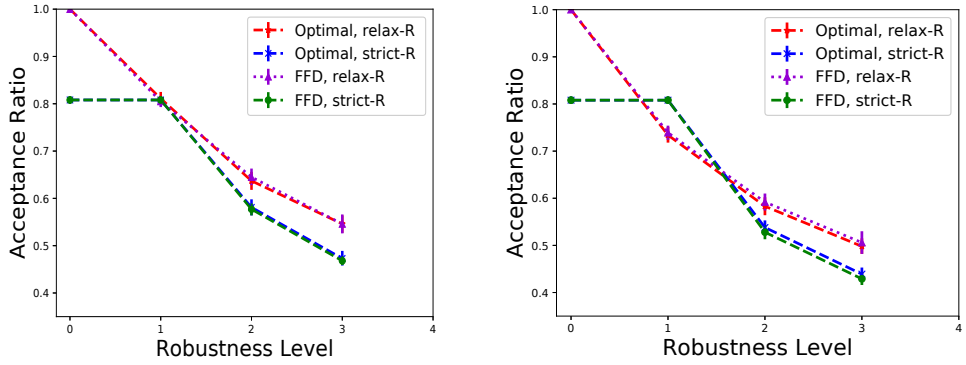
5.4.2 Acceptance Ratio

In this section we study the impact of required robustness level R on the ability to satisfy SFCs placement requests. To that aim, we use the *acceptance ratio* defined as the number of accepted requests over the total number of requests.

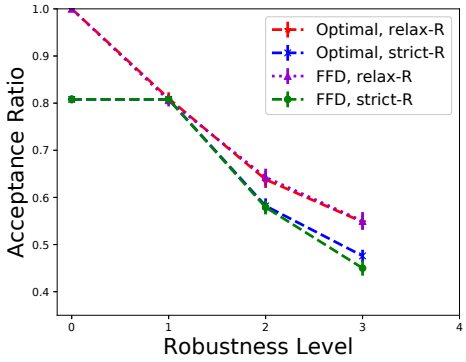
Figure 5.6 shows the evolution of the acceptance ratio with the 3 different large data-center topologies described above (i.e., Fat Tree, Spine and Leaf, and Generic) w.r.t. the robustness level. The particular choice of topologies permits to evaluate the impact of the number of fault domains and the number of core resources on the acceptance ratio. Here we distinguish between two configurations for our placement algorithm: in ***strict*** we impose the number of scaled replicas to be exactly $R+1$ while in ***relax*** the number of scaled replicas can be any integer value between $R+1$ and $(R+1)\cdot 2$.

Moreover, we consider two different function placement algorithms: (*i*) *Optimal* solves the optimization problem specified in Sec. 5.3.1.1 and (*ii*) *FFD* uses the well-known *First-Fit Decreasing* (FFD) greedy heuristic [18, 189].

[3] We ran the experiments on the site located in Rennes, https://www.grid5000.fr/.



(a) 48-*Fat-Tree* Topology (b) *Spine-and-Leaf* Topology



(c) *Generic* Topology

Figure 5.6: Comparing acceptance ratio of the Optimal solution and Greedy FFD for 3 different topologies with 3 robustness levels.

In general we can expect that the acceptance ratio decreases when the robustness level increases as increasing robustness means dedicating more resource to each function. This trend is confirmed by Figure 5.6. One can also expect to have better acceptance ratio with *Optimal* than with *FFD* but even if it is true, in practice the difference is negligible as shown in Figure 5.6.

While the impact of R and the impact of using FFD instead of the optimal are evident to forecast, it is much harder to speculate on the impact of being strict in the number of scaled replicas or not (i.e., *strict* versus *relax*). On the one hand being strict reduces the amount of resources used for each deployed function and should thus give more space for other functions. On the other hand not being strict allows splitting chains further such that the replicas can be "squeezed" in servers with less available resources. This duality is clearly visible in Figure 5.6.

For $R = 0$ we observe that by being strict, only around 81% of the requests can be satisfied while allowing more than $R+1$ scaled replicas allows to satisfy all demands. The difference between the two scenarios can be explained by the fact that we intentionally made the workload such that in 19% of the demands at least one function in the chain requires 8 cores. As the servers only have 4 cores, it is then impossible to install them unless we allow using multiple replicas (which is the case for *relax* with

R \ Topology	*Fat-Tree*	*Spine&Leaf*	*Generic*
0	0.808	0.808	0.808
1	0.838	0.797	0.836
2	0.649	0.615	0.647
3	0.570	0.542	0.578

Table 5.2: Similarity Index between the *strict* and *relax* configuration for the three different topologies.

$R = 0$ case but not for *strict* with $R = 0$ case). This first observation confirms that allowing more scaled replicas gives more flexibility in finding a placement solution.

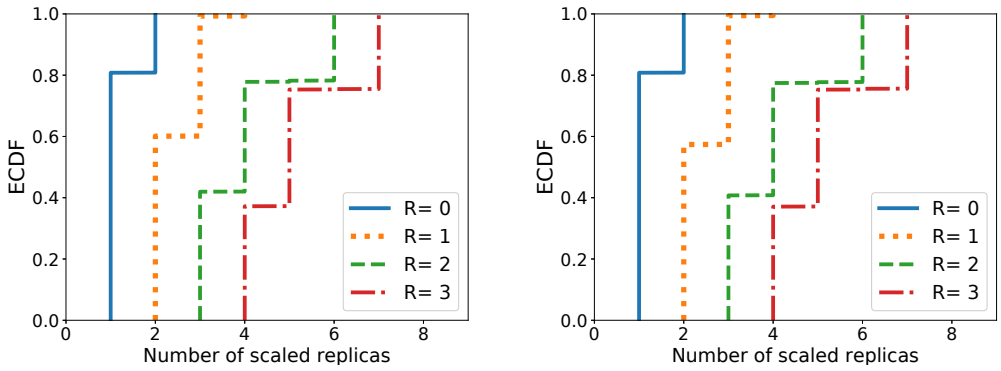
This trend is clearly visible, except for $R = 1$ where we can see that in the *Spine-and-Leaf* topology (see Figure 5.6b) *strict* outperforms *relax*. The reason of this difference lays in the fact that in the *strict* case it is still impossible to install the 19% of requests with at least one function requiring 8 cores. Indeed, in case of failure the only remaining replica would still require 8 cores, while under normal operations each of the two replicas only needs 4 cores. As these chains are not installed, they leave enough room for the others to be installed. On the contrary, with the *relax* case, these requests can be satisfied but consume a substantial amount of resources; they need at least 3 scaled replicas to be deployed, which prevents other chains to be installed, hence reducing the overall acceptance ratio.

It is worth mentioning that if the acceptance ratios for $R = 1$ seem to be identical for both cases in the *Fat-Tree* and the *Generic* topologies, they are actually slightly different and the similitude is only an artifact of the workloads and topologies that we used. Indeed, even though the acceptance ratios are very close, the placements are largely different as shown by the Jaccard similarity coefficient [87] of only 0.84 (see Table 5.2). In general, the dissimilarity of placements increases with R. For example, the Jaccard similarity coefficient is as low as 0.54 in the *Spine-and-Leaf* topology when $R = 3$.

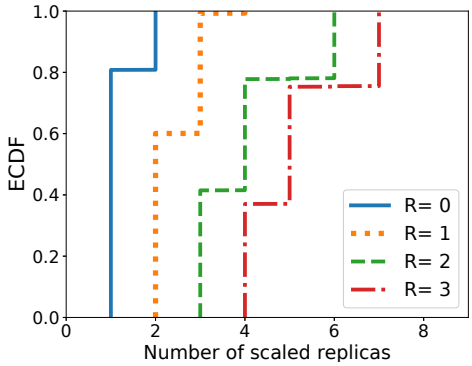
Keeping in mind that the *Fat-Tree* topology has the same number of fault domains as the *Spine-and-Leaf* topology but has more cores in total, and that the *Generic* topology has the same number of cores as the *Fat-Tree* topology but with more fault domains, the comparison between the 3 topologies leads us to conclude that as long as the number of fault domains is larger than *max_ iterations*, the number of cores is what influences the most the acceptance ratio.

To complement the acceptance ratio study, Figure 5.7 and Figure 5.8 provide the empirical cumulative distribution functions of the number of scaled replicas created when placing SFCs while guaranteeing different robustness levels for the three different topologies with the *relax* configuration. As we consider highly loaded topologies, most of the time $R + 1$ or $R + 2$ replicas are enough to ensure robustness level of R and we seldom reach the $(R+1) \cdot 2$ limit, as most resources are consumed by replicas of other chains.

Moreover, if we take a careful look at the number of scaled replicas for $R = 0$, about 80% of services are placed with only 1 replica which is the same value of the acceptance ratio for $R = 0$ with the *strict* configuration in Figure 5.6 – and about 20% with two scaled replicas. This extra replica leads to an increase in the acceptance ratio where the acceptance ratio reaches 1 when we relax the replication (in Figure 5.6).



(a) 48-*Fat-Tree* Topology (b) *Spine-and-Leaf* Topology



(c) *Generic* Topology

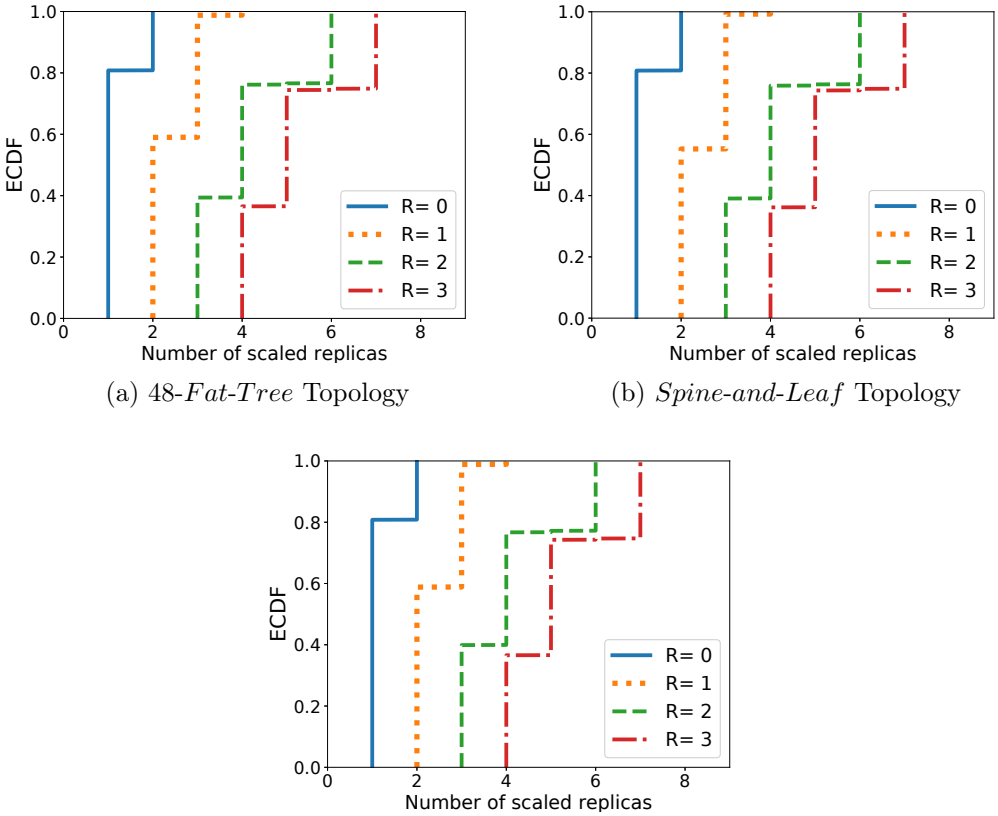
Figure 5.7: The ECDF for the number of created replicas with 3 robustness levels with 3 different topologies for the optimal algorithm.

If we study the probability of a chain to be accepted as a function of its requested number of cores (see Figure 5.9 and Figure 5.10), we see that our algorithm favors the installation of small chains over large ones, particularly for large values of R.[4]

5.4.3 Acceptance ratio in case of network congestion

In Sec. 5.4.2 when a request is rejected, the reason is always that the placement algorithm was not able to find hosts with enough free cores, and never because of the network capacity. This is because each host is connected to the network with a

[4]We can explain that the figures do not show smooth decreasing lines by the fact that we only used 20 different chain types, which is not enough to cover all potential cases.



(a) 48-*Fat-Tree* Topology

(b) *Spine-and-Leaf* Topology

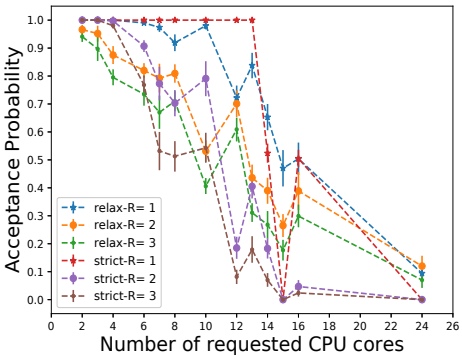
(c) *Generic* Topology

Figure 5.8: The ECDF for the number of created replicas with 3 robustness levels with 3 different topologies for the FFD algorithm.

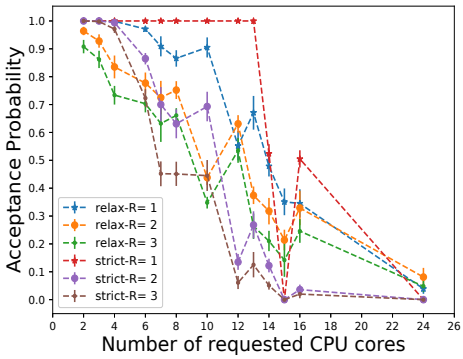
1 Gbps link and has 4 cores. As our algorithm cannot overcommit hosts we know that a host will never run more than 4 functions simultaneously. Therefore, as each vLink requests 250 Mbps, the traffic to or from a host never exceeds 1 Gbps which is not enough to overload the host links and as we use clos topologies, it means that the backbone network also cannot be overloaded.

In this section, we aim at stressing the network as well as the hosts. To that aim we keep the same workload as in Sec. 5.4.2 but vLinks request 500 Mbps instead of 250 Mbps, which may result in network congestion.

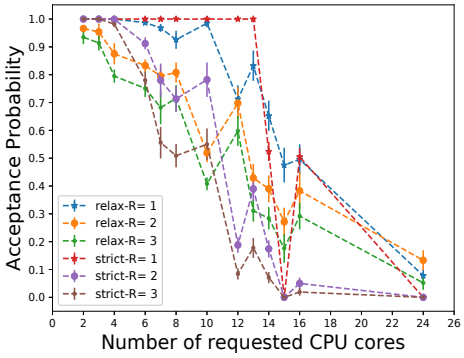
Figure 5.11 shows the acceptance ratio for this new scenario (labeled *w/ congestion* and compares it to previous results (labeled as *w/o congestion*). For $R = 0$, the acceptance ratio drops by 50% or more because the network cannot handle the load. Even though the drop is important in both cases, as the ***relax*** option allows to create multiple replicas, it outperforms the ***strict*** option. However, as soon as $R \geq 1$, we obtain the same results than in Sec. 5.4.2 as we fall back in a case with no network congestion because when every function uses at least 2 scaled replicas, the network demand for a host will not exceed 1 Gbps.



(a) 48-*Fat-Tree* Topology



(b) *Spine-and-Leaf* Topology



(c) *Generic* Topology

Figure 5.9: Probability of service chain being accepted based on the number of requested CPU cores for different robustness levels with 3 different topologies using the Optimal algorithm.

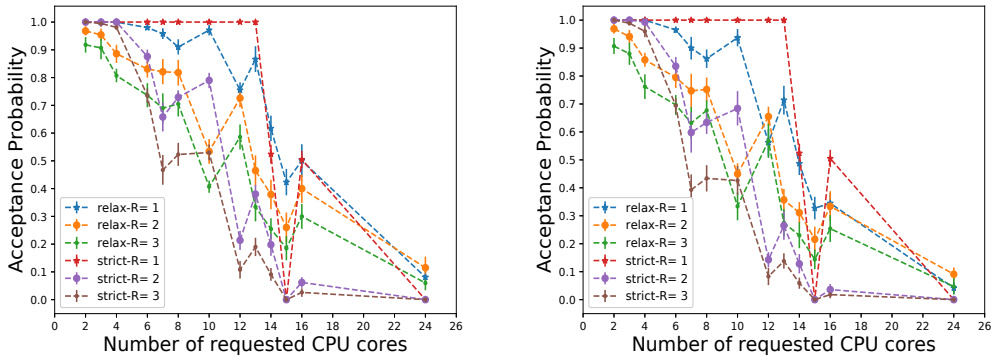
5.4.4 SFC Request Placement Time

To be acceptable, the time spent on finding a placement must be at most of the same order of magnitude as the deployment of the VMs themselves in order not to impact the deployment time of a service.

Figure 5.12 shows the whisker plot of all computation times of Algorithm 1 for the harder instance of the problem, namely the *Fat-Tree* topology with the ***relax*** scheme for both the optimal and FFD. The simulations were performed in Grid5000 [26] on the Rennes site in fall 2018.

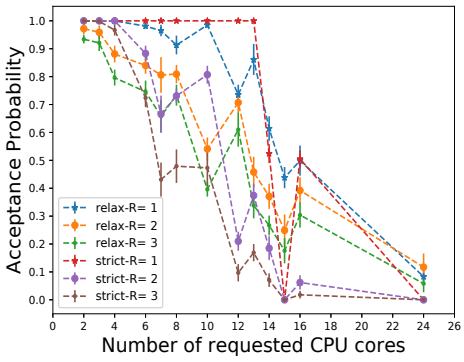
We make the distinction between the time elapsed when requests result in an effective placement (*Accepted Services*) in Figure 5.12b and when they do not (*Rejected Services*) in Figure 5.12c, while Figure 5.12a (*All Services*) aggregates computation time for all requests, regardless of the outcome.

The computation time increases rather linearly with the robustness level and never



(a) 48-*Fat-Tree* Topology

(b) *Spine-and-Leaf* Topology



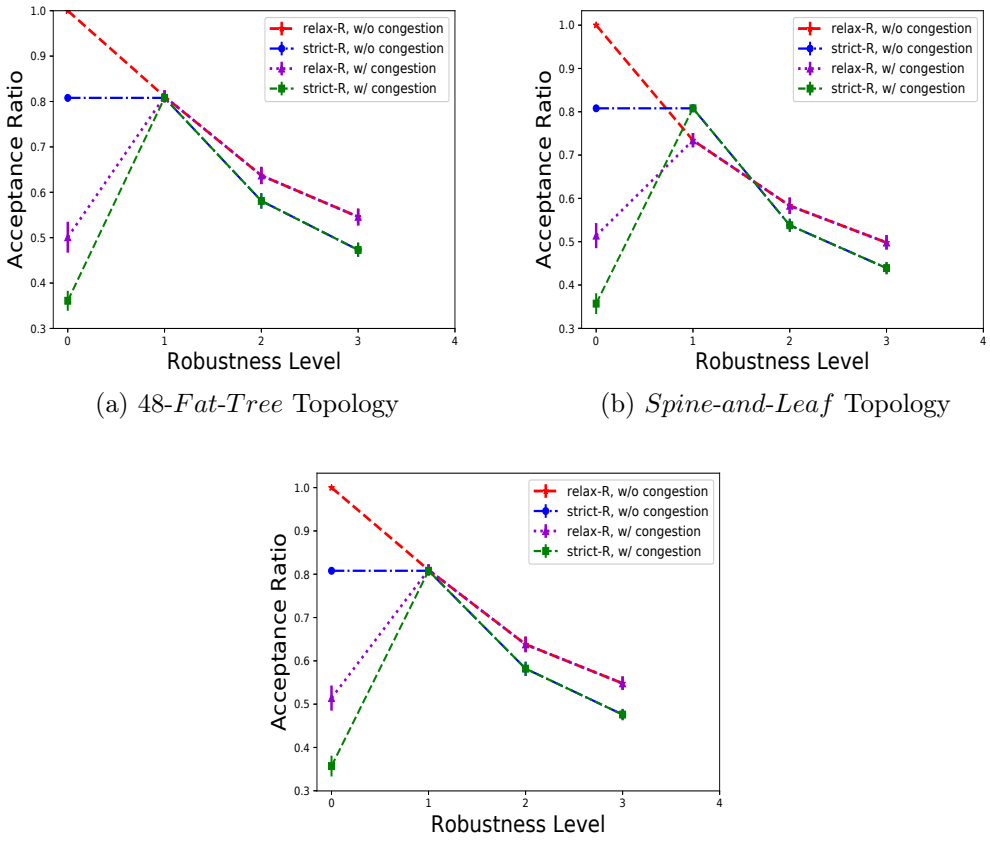
(c) *Generic* Topology

Figure 5.10: Probability of service chain being accepted based on the number of requested CPU cores for different robustness levels with 3 different topologies using the FFD algorithm.

exceeds a few seconds, which is negligible compared to the typical time necessary to deploy and boot virtual functions in data centers [116]. This rather linear increase is because an increase of R incurs a proportional increase of the number of iterations (*max_iteration*) and the number of required fault domains (n in `solve_placement(`S`, `G`, `n`)`) but does not change the size of the `solve-placement` problem (see Sec. 5.3) as the size of the fault domain is not impacted by R.

Furthermore, for both figures, the computation time is longer when requests are rejected than when they are accepted as the rejection of a service request can only be decided after having tested all the allowed number of replicas (i.e., **max_iteration**). Note that all demands are accepted for the *relaxed* case when $R = 0$ which explains the absence of observations for $R = 0$ in Figure 5.12c.

Regarding accepted services, (e.g., for $R = 1$ in Figure 5.12b), the spread between median and upper quartile is smaller than the spread between median and lower quartile as most of placements require $R + 1$ or $R + 2$ replicas only. However, in some scenarios, the algorithm is iterated until the maximum allowed iterations reached in order to find this valid placement, which explains having the outliers in the Figure 5.12.



(a) 48-*Fat-Tree* Topology

(b) *Spine-and-Leaf* Topology

(c) *Generic* Topology

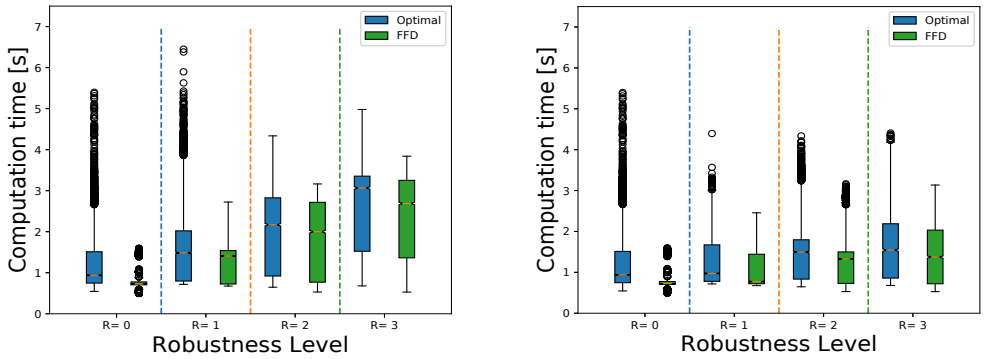
Figure 5.11: Comparing acceptance ratio of the Optimal solution for the different topologies with 3 robustness levels for the two workloads.

Interestingly, even though the execution time is shorter when *FFD* is used, it remains of the same order of magnitude as when the optimal placement is used instead.

5.5 Conclusion

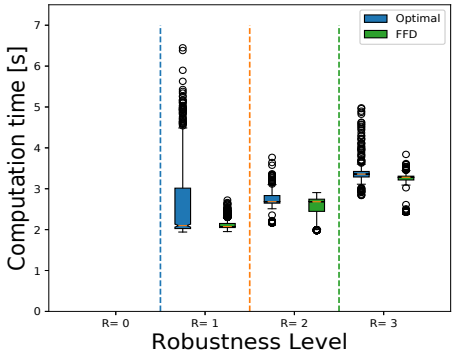
In this chapter we proposed a solution to deploy tenants' service function chains in public cloud data centers with guarantees that chains are robust to R independent fail-stop physical node failures. The idea is to replicate the chain in multiple independent locations in the data center and to balance the load between these replicas based on their availability in order to prevent downtime upon failures in the physical infrastructure.

To that aim, we proposed an online two-phase algorithm that determines the number of replicas and where to place them to guarantee some robustness level based on an ILP solution or its approximation. Moreover, we extensively evaluated this algorithm on very large data center networks – up to 30,528 nodes – to assess the



(a) *All* Services

(b) *Accepted* Services



(c) *Rejected* Services

Figure 5.12: Algorithm computation time with different robustness levels for the Fat-Tree topology with relax configuration.

feasibility of our proposition in very large scale data-centers. We showed that approximating the solution with the widely used FFD technique was not mandatory as optimal placement of independent replicas was feasible in acceptable time, which allows placement decisions to be made on-demand and without prior knowledge on the DC workload.

We studied the impact of the choice of the topology and the expected robustness level on the acceptance ratio and on the placement computation time. It shows that when the data center is sufficiently provisioned, our algorithm is able to provide a robust placement for all the chains. On the contrary, when the DC lacks resources, the algorithm tends to favor shorter chains as they consume less resources, giving them more placement options.

However, the deterministic solution provided in this chapter works well when the requested services are deployed by a service provider that has a full control and knowledge of the underlying infrastructure. Thus, in the next chapter we will present a stochastic approach to deploy the services requested by tenants oblivious to the underlying physical network.

6 An Availability-aware SFC placement Algorithm for Fat-Tree Data Centers

Contents

As mentioned before, complex inter-connections of virtual functions form the so-called Service Function Chains (SFCs) deployed in the Cloud. Such service chains are used for critical services like e-health or autonomous transportation systems and thus require high availability. Respecting some availability level is hard in general, but it becomes even harder if the operator of the service is not aware of the physical infrastructure that will support the service, which is the case when SFCs are deployed in multi-tenant data centers. In the previous chapter, we considered the situation in which the service provider has a full control and knowledge of the underlying infrastructure.

In this chapter, we propose an algorithm to solve the placement of topology-oblivious SFC demands such that placed SFCs respect availability constraints imposed by the tenants. The algorithm leverages Fat-Tree properties to be computationally doable in an online manner. The simulation results show that it is able to satisfy as many demands as possible by spreading the load between the replicas and enhancing the network resources utilization.

6.1 Introduction

As mentioned before, in NFV environments, multiple VNFs share the same physical resources of the underlying infrastructure; even a single failure in the underlying network can affect a large number of the services of the operators. Therefore, ensuring the required availability level is an important feature in virtualized environments.

Replication mechanisms have been proposed in the literature (e.g., [27, 59, 113]) to improve the required service availability based on VNF redundancy, which allow configurations in Active-Backup or Active-Active modes. However, some propositions [29] focus on replicating the SFCs in multi-tenant data centers where the tenant demands are oblivious to the actual physical infrastructure of the Data Center. Such an environment is particularly challenging as the demand is not known in advance and cannot be controlled. For the data center operator, it is therefore important to limit the number of replications to its minimum, yet respecting the level of service agreed with its tenants.

Differently from the solution presented in the previous chapter, in this chapter, we propose a placement algorithm for topology-oblivious SFCs in Data Centers relying on Fat-Tree topologies. The algorithm is run by the network hypervisor and guarantees that Service Level Agreements (*SLA*) with the tenants are respected, given the availability properties of the hardware deployed in the data center. Our proposition is based on an iterative linear program that solves the placement of SFCs in an online manner without prior knowledge on placement demand distribution. The algorithm is made computationally doable by leveraging symmetry properties of Fat-Tree topologies. Our evaluation on a very large simulated network topology (i.e., 27,648 servers and 2,880 switches) shows that the algorithm is fast enough for being used in production environments.

The rest of the chapter is organized as follows. Section 6.2 describes the problem statement. Section 6.3 formulates the optimization model. Finally, Section 6.5 evaluates the performance of our solution and Section 6.6 concludes this chapter.

6.2 Problem Statement

This section defines the problem of placing SFCs in Data Centers under availability constraints.

Without any loss of generality, and inspired by works ([68, 81]), we only consider server and switch failures and ignore link failures. We also consider that all equipment of a same type has the same level of availability (more details are provided in Section 6.5.1).

6.2.1 Detailed description on the problem

This work develops an availability-oriented algorithm for resilient placement of VNF service chains in Fat-Tree based DCs where component failures are common [81].

The Fat-Tree topology is modeled as a graph where the vertices represent switching nodes and servers, while the edges represent the network links between them. Furthermore, SFC provides a chain of network functions with a traffic flow that need to traverse them in a specific order. We only consider acyclic SFCs. As we are in a multi-tenant scenario, functions are deployed independently and cannot be aggregated (i.e., function instances are not shared between SFC instances or tenants).

Each function is considered as a single point of failure. Thus, to guarantee the availability of a chain we use *scaled replicas*: we replicate the chain multiple times and equally spread the load between the replicas.

Upon independent failures, the total ***availability*** for the whole placed SFC replicas will be computed using Eq. (6.1) (availability for parallel systems).

$$availability = 1 - \prod_{i \in n\_replicas} (1 - ava_{sc_i}), \tag{6.1}$$

where ava_{sc_i} is the availability of replica i of service chain sc and $n\_replicas$ is the number of scaled replicas for this service chain. The availability of each service chain replica ava_{sc_i} is defined by

$$ava_{sc_i} = \prod_{f \in F} A_f, \forall i \in n\_replicas \tag{6.2}$$

where A_f is the availability of a service chain function f, which corresponds to the availability of the physical node that hosts this function ([29, 59]).

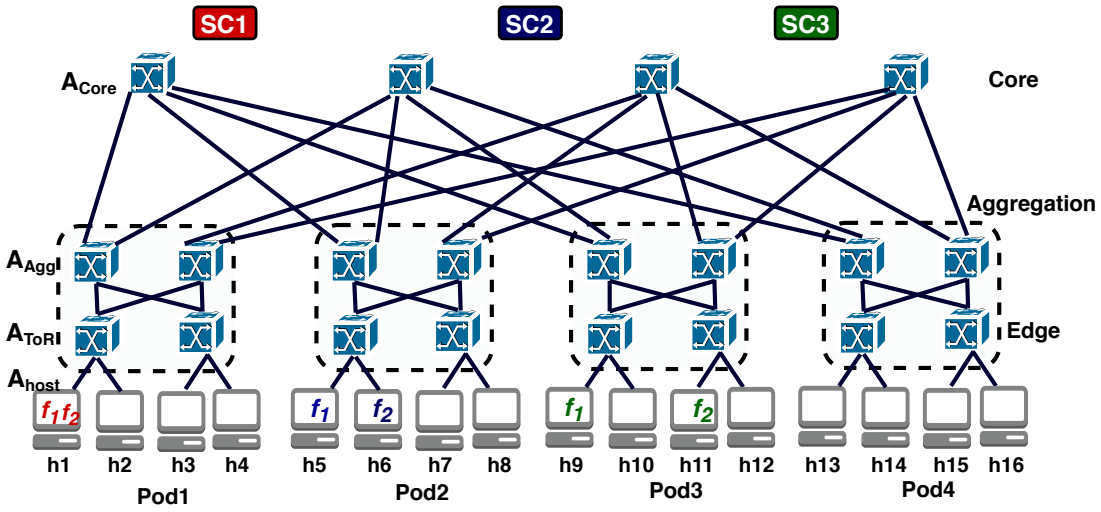


Figure 6.1: Fat-Tree topology example.

In Fat-Tree network topologies (Fig. 6.1), the availability of a system composed of multiple functions depends on its placement in the topology. For example, the availability of a SFC for the three scenarios for SFCs placement presented in Fig. 6.1 is calculated as follows: (i) `Scenario 1`, where service chain (`SC1`) is mapped to only one physical host, the availability of `SC1` is equal to the availability of its host node (h_1) while (ii) in `Scenario 2`, `SC2` is placed on 2 different hosts under the same ToR

switch, so the availability of this service is equal to the availability of all participating host nodes in addition to the availability of their parent ToR switch as follows:

$$A_{SC2} = A_{h5}.A_{h6}.A_{ToR}. \tag{6.3}$$

However, in `Scenario3` where the chain `SC3` is placed on different hosts connected to different ToR switches within the same pod, the availability is given by:

$$A_{SC3} = A_{h9}.A_{h11}.\left(A_{ToR}\right)^2.\left[1-\left(1-A_{Agg}\right)^2\right]. \tag{6.4}$$

In the general case of a k-ary Fat-Tree, each host is connected to one ToR switch (edge switch) which is directly connected to $k/2$ switch in aggregation layer and each switch in the aggregation layer is then connected to $k/2$ switches in the core layer. The generalized availability equation becomes:

$$A_{SC3} = (A_{host})^{n\_h}.(A_{ToR})^{n\_ToRs}.\left[1-\left(1-A_{Agg}\right)^{k/2}\right], \tag{6.5}$$

where n_h refers to the total number of used hosts and n_ToRs refers to the total number of ToRs involved in the placement.

6.3 Model Description and Formalization

6.3.1 Model Variables

We present here the variables used in our model formulation to place one particular service chain:

- $p \in P$: pod Id, $t \in ToRs$: ToR Id, $h \in Hosts$: host Id and F is a sequence of NFs id f of the SFC;
- π_p: binary variable, equals to 1 if the pod p is used, and 0 otherwise;
- τ_α: availability for the mapped service chain;
- $R \in [0,1]$: SFC requested availability in the SLA;
- $\vartheta_{p,t,h,f}$: binary variable, equals to 1 if the function f is mapped to a specific physical host identified by its pod p, its ToR t, and its Id h, and 0 otherwise;
- $\xi_{h,p,t}$: binary variable, equals to 1 if the host h under the ToR t of the pod p is used, and 0 otherwise;
- $\rho_{t,p}$: binary variable, equals to 1 if the ToR t on the pod p is used, and 0 otherwise;
- $\delta_{f,p}$: a binary variable that equals 1 if the function f is mapped to the pod p, and 0 otherwise;
- ε_p: total number of used hosts under pod p;
- σ_p: total number of used ToRs under pod p;

- C_f: required CPU resources for function f;
- C_h: total available CPU resources on host h;
- A_h, A_s: availability of host h and switch node s, respectively.

6.3.2 Model Formulation

The optimization objective is to minimize the number of scaled replicas (i.e. number of used pods as each replica is placed in a different pod). This translates into:

$$Obj : Min \sum_{\mathbf{p} \in \mathbf{P}} \pi_{\mathbf{p}}. \tag{6.6}$$

Subject to the following constraints:

$$\tau_\alpha >= R \tag{6.7}$$

$$\forall_{h \in H, p \in P, t \in T} : \xi_{h,p,t} = 1 \text{ if} \sum_{f \in F} \vartheta_{p,t,h,f} \geq 1 \tag{6.8}$$

$$\forall_{t \in T, p \in P} : \rho_{t,p} = 1 \text{ if} \sum_{h \in H} \sum_{f \in F} \vartheta_{p,t,h,f} \geq 1 \tag{6.9}$$

$$\forall_{f \in F, p \in P} : \delta_{f,p} = 1 \text{ if} \sum_{t \in T} \sum_{h \in H} \vartheta_{p,t,h,f} \geq 1 \tag{6.10}$$

$$\forall_{p \in P} : \pi_p = 1 \text{ if} \sum_{t \in T} \sum_{h \in H} \sum_{f \in F} \vartheta_{p,t,h,f} \geq 1 \tag{6.11}$$

$$\forall_{p \in P}, \ \forall_{f \in F} \ \forall_{f' \in F, f' > f} : \delta_{f,p} \leq \delta_{f',p} \tag{6.12}$$

$$\forall_{f \in F}, \forall_{p \in P} : \sum_{t \in T} \sum_{h \in H} \vartheta_{p,t,h,f} \leq 1 \tag{6.13}$$

$$\forall_{p \in P} : \ \varepsilon_p = \sum_{t \in T} \sum_{h \in H} \xi_{h,p,t} \tag{6.14}$$

$$\forall_{p \in P} : \ \sigma_p = \sum_{t \in T} \rho_{t,p} \tag{6.15}$$

$$\forall_{p \in P} : \alpha_p = \begin{cases} 0, & \text{if } \varepsilon_p = 0 \\ A_h, & \text{if } \varepsilon_p = 1 \\ A_h^{\varepsilon_p}.A_s, & \text{if } \varepsilon_p > 1 \text{ and } \sigma_p = 1 \\ A_h^{\varepsilon_p}.A_s^{\sigma_p}.(1-(1-A_s)^{\frac{k}{2}}), & \text{if } \sigma_p > 1 \end{cases} \tag{6.16}$$

$$\tau_\alpha = 1 - \prod_{p \in P} (1 - \alpha_p) \tag{6.17}$$

$$\forall_{h \in H} \sum_{p \in P} \sum_{t \in T} \sum_{f \in F} \vartheta_{p,t,h,f} \, C_f \leq C_h \tag{6.18}$$

Constraint (6.7) ensures that the placement offers an availability at least as high as the one required in the SLA. Constraint (6.8) (resp. (6.9)) marks that the host (resp. ToR) is used if at least one NF is deployed on it (resp. on a host connected to it). Similarly, Constraint (6.11) indicates whether or not a specific pod is used.

Constraint (6.10) indicates that a function f is placed on a specific pod p.

Constraint (6.12) ensures that if one function of a scaled service replica is placed in one pod p then all other functions of this replica are placed in this same pod. Constraint (6.13) ensures that two replicas of the same function are never placed in the same pod.

Equation (6.14) refers to the total number of used hosts under each pod p, while Equation (6.15) indicates to the total number of used ToRs under each pod p in order to use them in Equation (6.16) to compute the availability of each scaled SFC replica. The latter takes one of four possible values based on the values we get from Equations (6.14) and (6.15). Finally, Constraint (6.18) ensures that the functions placed on a physical host node cannot use more CPU resources than its host resource capacity.

Constraints (6.16) and (6.17) are non-linear; we show how to linearize them in Appendix A.1. In our evaluation section, we used the linearized version of the model.

6.4 SFC Placement with Robustness

Directly solving the model of Sec. 6.3 for a large DC topology is impractical. Instead we apply the model on a (small) subset of the topology, more precisely, only in one fault domain (pod), for the new model: Equations (6.7, 6.17) will be removed, and the new objective is to maximize the placement availability over that fault domain :

$$Obj : Max(\alpha_p). \tag{6.19}$$

Our algorithm is called each time a request to install an SFC is received. Specifically, for a *required availability* R, the algorithm determines how many scaled replicas to create for that SFC and where to deploy them; taking into account the availability of network elements (servers and switches) without impairing the availability guarantees of the chains already deployed. To guarantee the isolation between scaled replicas, each replica of a chain is deployed in a different fault domain.

Algorithm 4 presents the pseudo-code of our algorithm where `scale_down(`C`, `n`)` is a function that computes the scaled replica scheme, i.e., an annotated graph representing the scaled down chain, for a chain C if it is equally distributed over n scaled replicas and where `solve_placement(`S`, `G`, `n`)` solves the problem of placing n replicas S on the network topology G. The solution of a placement is a set of mappings associating replica functions and the compute nodes on which they have to be deployed. The solution is empty if no placement can be found.

Our algorithm starts with one replica of a service request and first checks that no function is requesting more resources than what the pod can offer.

In the case it is not possible to find a placement with one replica, the algorithm scales down the chain S by adding one more replica and tries to find a placement for each one of these replicas in different fault domains. Otherwise, the algorithm tries to find a placement for it under one fault domain of the network (the fault domain is chosen randomly to spread the load over the entire DC) using `solve_placement(`S`, `G`, `n`)` function; if no placement is found in the current fault domain, we check the another fault domain, otherwise we compute the total availability for the current placement.

This strategy continues until a termination condition is met: (*i*) if the requested availability is reached then the service can be deployed with (`deploy(`*placement)*); (*ii*) if the maximum acceptable time for finding a placement is reached then no solution is found; (*iii*) if the number of created scaled replicas reached the maximum number of replicas (i.e., maximum number of fault domains), then no solution is found. The `compute_ava` function computes the availability of a chain placement according to the formulas presented in Sec. 6.2.

6.4.1 Scale down function

When multiple replicas are used, each one gets a fraction of the load and their individual resource requirements is lower than the one needed if there is only one chain instance. The resources depend on the availability of the other replicas and can be upper-bounded by:

$$\hat{R}_{f,n} = \left\lceil \frac{R_f}{n} + (n-1) \cdot \frac{C_f}{n} \cdot (1 - Ava_f) \right\rceil, \tag{6.20}$$

where $\hat{R}_{f,n}$ is an upper-bound on the average amount of resources that would require a replica of a function f if it is replicated n times while R_f is the number of resources required by f in case it is not replicated at all, and Ava_f is the availability of the least available replica among the n replicas.

6.4.2 Solve placement function

The `solve_placement(`S`, `G`, `n`)` function considers two graphs: the DC topology graph G and the scale replica graph S. The purpose of this function is to project

Algorithm 4: Availability-aware placement algorithm

Input: Physical network Graph: G
$chain \in$ Chains
Scaled chain replica graph: $replica\_scheme$
Required availability: R

$T = max\_time$; $n = 1$; $tot\_ava = 0$; $tot\_time = 0$;
$placement = \phi$
$replica\_scheme = chain$
while $tot\_ava < R$ ***and*** $tot\_time < T$ ***and*** $n < max\_n$ **do**
    if $max\_req > max\_ava$ **then**
        $n = n + 1$
        $replica\_scheme =$ `scale_down(`$chain, n$`)`
    else
        $placement =$ `solve_placement(` $replica\_scheme, G, n$`)`
        if ***not*** $placement$ **then**
            $n = n + 1$
            $replica\_scheme =$ `scale_down(`$chain, n$`)`
        else
            $tot\_ava =$ `compute_ava(`$placement$`)`
            $n = n + 1$
    tot_time.update()
if $tot\_ava \geq R$ **then**
    `deploy` ($placement$)
else if $tot\_time \leq T$ **then**
    Error ("max time is reached !")
else
    Error ("max number of replicas is reached !")

the scaled replica graph S on the topology graph G with respect to the physical and chain constraints.

For each fault domain in the network, `solve_placement(`S`,` G`,` n`)` tries to find a solution for the linear problem defined earlier that aims at finding a placement for the scale replica graph in one fault domain while maximizing the availability of the replica placement.

6.5 Evaluation

In the following we evaluate the Availability-aware placement algorithm introduced in the previous section.

6.5.1 Simulation Environment

We have implemented a discrete event simulator in Python interfaced with the Gurobi Optimizer 8.0 solver [138]. All simulations have been run on a Intel i7-4800MQ CPU at 2.70GHz and 32GB of RAM running GNU/Linux Fedora core 21.[1]

In the evaluation, requests to deploy SFCs are independent and follow an exponential distribution of mean inter-arrival time T_{IA} (measured in arbitrary time units). SFCs have a service time of S time units, i.e., the time the SFC remains in the system is randomly selected following an exponential distribution of mean S. If an SFC cannot be deployed in the network, it will be rejected. In total, our synthetic workload for the simulations contains 2,000 SFC request arrivals made of 20 random SFCs. As we are only interested in the steady state of the system, the servers are preloaded with service chains. All experiments presented here were repeated 5 times (5 different workloads of 2,000 SFC requests).

Furthermore, all SFCs are linear, i.e., they are formed of functions put in sequence between exactly one start point and one destination point. The number of NFs between the two endpoints is selected uniformly between 2 and 5, based on typical use cases of networks chains [107], and the requirements of each function in terms of cores is 1, 2, 4, or 8 inspired by the common Amazon EC2 instance types [8]. Simulations are performed on a 48-Fat-Tree topology with 48 pods having each 576 hosts for a total of 27,648 hosts where all hosts have the same number of cores (4 cores per host). The availability of the physical devices in the Data Center are assigned accordingly to the statistical study of these works ([68, 81]), namely 0.99 for servers, 0.9999 for ToR and aggregation switches, 0.99999 for core switches, and 1.0 for links.

Because of space limitations, we fixed the SLA to be equal for each SFC in a run (we consider the 0.95, two, three, four and five nines for SLA), however the algorithm allows using a different SLA value for each SFC.

To avoid impacting the deployments of services, finding their placement must be computed in reasonable time. Therefore, we limited the computation time to at most 6s per request. If no solution is found within this time, the request is rejected (the acceptable time for finding a placement must be at most of the same order of magnitude as the deployment of the VMs themselves to not impact the deployment time of a service).

6.5.2 Acceptance Ratio

The required availability level has an impact on the ability of a network to accept or not SFC requests. To study this impact, we consider the *acceptance ratio* defined as the number of accepted SFC requests over the total number of requests.

Figure 6.2 shows the evolution of the acceptance ratio w.r.t. the 5 different SLA levels. We can notice that the acceptance ratio decreases when the required availability

[1] All the data and scripts used in this chapter are available on https://team.inria.fr/diana/robstdc/.

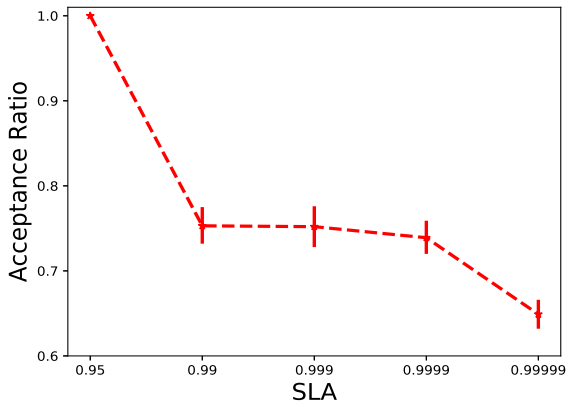


Figure 6.2: Comparing acceptance ratio for 5 different SLA values with 48-Fat-Tree topology.

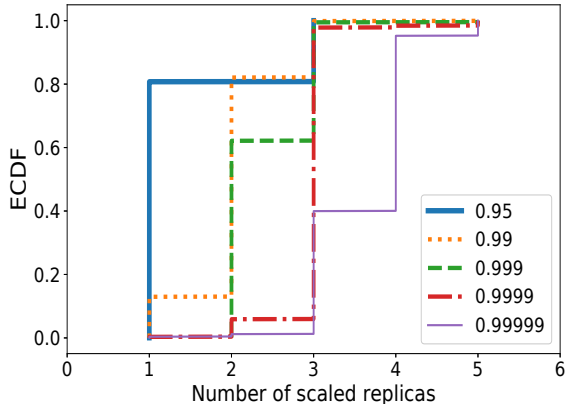


Figure 6.3: The ECDF for the number of created replicas with 5 different SLA with 48-Fat-Tree topology

level increases as each chain must reserve more resources than for lower availability levels as the physical topology is kept untouched. This can be explained by the fact that when increasing the required availability of a chain, it is necessary to replicate it further and then to consume more resources as at least one core is attributed to each function, replicated or not.

6.5.3 Level of Replication

To complete the acceptance ratio study, Figure 6.3 provides the Empirical Cumulative Distribution Function (ECDF) of the number of scaled replicas created for accepted SFCs for the different studied SLAs. It is clear that for the lowest required availability (0.95), 80% of SFCs were satisfied with exactly one replica as the availability of network elements are higher than this SLA level. However, when a SFC request needs more resources than the available resources in the network, it is split

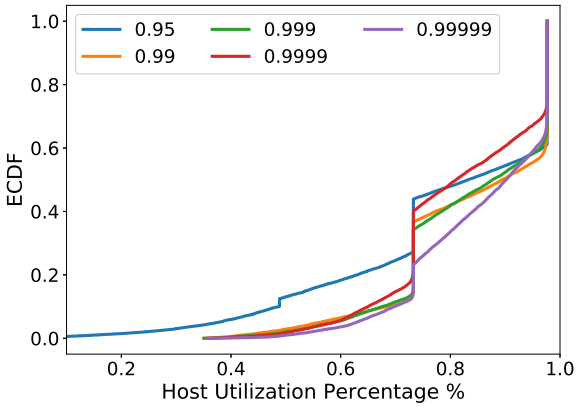


Figure 6.4: The ECDF for the host core utilization for different SLAs with 48-Fat-Tree topology

(20% of services were split for SLA=0.95) and, as the required availability increases, the required number of replicas are increased to satisfy the SFC SLA.

Interestingly, as in practice the availability of the infrastructure is high we can observe than even for an aggressive SLA of 0.9999, 90% of SFC requests can be satisfied with no more than 3 replicas. Figure 6.3 shows that the number of replicas tops to 5 even though in theory it would be possible to observe up to 48 replicas in a 48-Fat-Tree topology as there are 48 pods. We can explain this, as the computation time of our optimization is restricted to be less than 6 seconds.

Nevertheless, we can observe that a general increase of availability requirement increases the required number of replicas, which explains why the acceptance ratio decreases when the availability requirements increase.

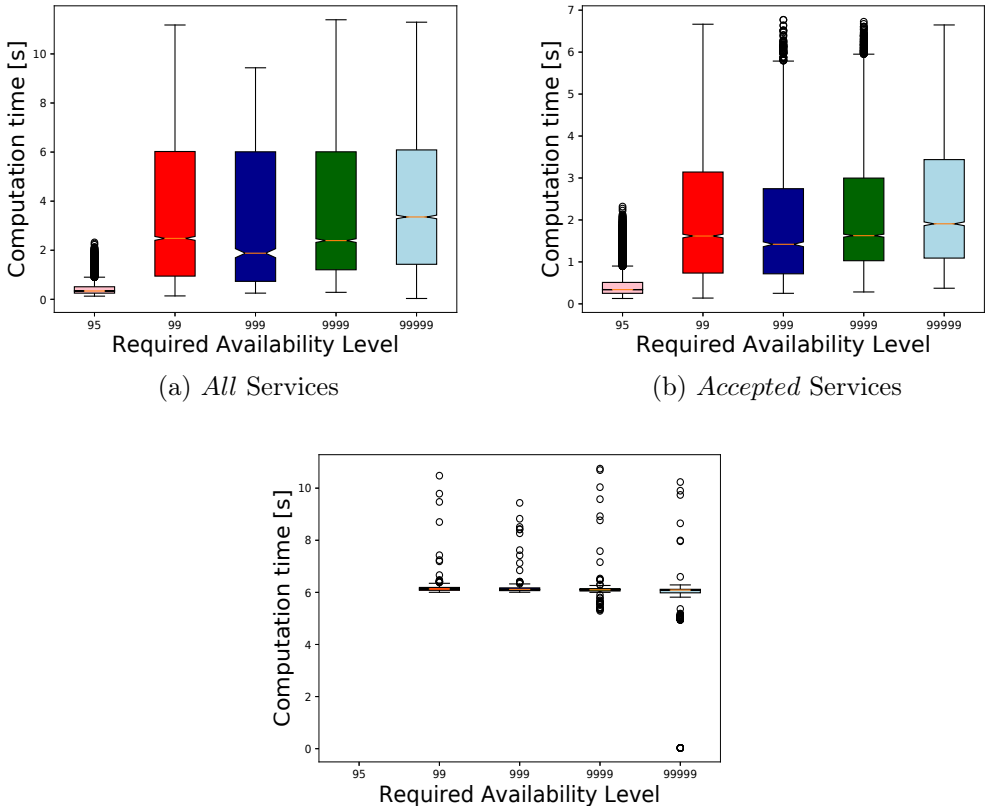
6.5.4 Servers utilization

Figure 6.4 shows the ECDF of the server core utilization where the host utilization is the ratio between the total consumed CPU time and the total CPU time offered by the server. For example, for an experiment that lasts 2 units of time, if a server has 4 cores, the total CPU time offered by the server is 8. If during the experiment 3 functions are installed on the server and each function lasts 1.1 units of time and requires 2 cores, the total consumed CPU time is $3 \cdot 2 \cdot 1.1 = 6.6$, which means that the server is utilized at 82.5% of its capacity ($\frac{6.6}{8} = 82.5\%$).

We can see that more than 40% of the servers are fully occupied, in SLA scenarios. However as the required level increases, more servers CPU resources are used which explains why when the required availability increases, the overall load of the servers increases. When the required availability is as high as 0.99999, more than 80% of servers are more than 80% occupied. These results show that even when the infrastructure is close to be saturated, our algorithm is able to efficiently allocate resources in order to satisfy as much demands as possible.

6.5.5 SFC Request Placement Time

To avoid impacting the deployments of services, finding their placement must be computed in reasonable time, this is why we limited the computation time to at most 6s per request [116]. If no solution is found within this time the request is rejected.



(a) *All* Services

(b) *Accepted* Services

(c) *Rejected* Services

Figure 6.5: Algorithm computation time with different SLAs with the Fat-Tree topology.

Figure 6.5 shows the whisker plot for the placement computation time. We make the distinction between the time elapsed when requests are accepted (*Accepted Services*) in Figure 6.5b and when it does not (*Rejected Services*) in Figure 6.5c, while Figure 6.5a (*All Services*) aggregates computation time for all requests, regardless of the outcome.

The computation time increases rather linearly with the availability level and never exceeds a few seconds, which is negligible compared to the typical time necessary to deploy and boot virtual functions in data centers *some-ref*. The reason of the higher computation time is that the required SLA incurs a proportional increase of the number of iterations to fulfill this required availability.

Furthermore, the computation time is longer when requests are rejected than when they are accepted as the rejection of a service request can only be decided when the algorithm iterates over available fault domains, trying to satisfy the requested availability, until a termination condition is met. Furthermore, the spread between the median and upper quartile is smaller than the spread between the median and lower quartile 6.5b as most of placements require more replicas.

6.6 Conclusion

In this chapter, we propose an online algorithm for SFC placement in data centers that leverages the Fat-Tree properties and respects the SFC availability constraints dictated by the tenant, taking into account the network components availability. The simulation results show that our algorithm is fast enough for being used in production environments and is able to satisfy as many demands as possible by spreading the load between the replicas while improving the network servers CPU utilization at the same time.

This proposed solution could be extended to consider other data center topologies, such as Leaf-and-Spine and BCube.

In this chapter, the service provider has full control over the environment with only one kind of failures, namely *fail-stop* nodes failures, are considered. However, when moving to a collaborative environment, other failures should be taken into account. In the next chapter, we explore this problem and propose a new protocol to ensure the correctness of the users' applications against different types of failures (i.e., byzantine and rational nodes).

7 MARS: Map-reduce in BAR Systems

Contents

All solutions presented in the previous chapters are feasible and applicable in the centralized environment, namely Data Center networks, with a main authority that controls the underlying infrastructure. In such an environment, users requested services are deployed and need to be robust in the presence of only simple fail-stop physical failures.

However, in some scenarios, there is a need for a collaboration between many contributors in order to run tenants' applications, specifically MapReduce applications. In such cooperative scenarios, when different partners or enterprises collaborate together to provide cooperative services based on peer-to-peer network systems, we move to a completely decentralized environment that is subject to not only byzantine failures, but also rational actors that have self-interested behavior. However, these applications used in DC have been developed for trusted homogeneous environments and so the algorithms and protocols used in such applications are not adapted to untrusted heterogeneous environments.

Therefore, we need to move from the network level to the application one where the MapReduce application should be changed and adapted to be able to work correctly under the existence of both byzantine and rational nodes in the system before solving the placement problem. In this chapter, we present a new MapReduce framework,

named MARS, with a new scheduling mechanism conceived to be robust against byzantine and rational actors in untrusted peer-to-peer environments. The work presented here been done in collaboration with M. Alberto Zirondelli in the context of a Master research internship.

7.1 Introduction

We live in the data age as a result of the rapid development of the Internet that has led to large volumes and variety of data becoming available. Additionally, the rate at which data is generated is increasing enormously [162]. For example, it is reported that Facebook collects approximately 500 terabytes per day [165]. These characteristics are known to be the fundamental of Big Data [52]. Collecting, managing, and analyzing such data within a tolerable elapsed time require massive storage and intensive computational power.

Big Data analytics is the center of attention nowadays but surprisingly even though these data are scattered around the world from heterogeneous source [134], most data analytics solutions rely on the execution of analytics on data centers that are massively centralized [140]. The MapReduce programming model [49], presented by Google in 2004, with its popular implementation *Hadoop framework* is often used for data analytics.

Hadoop MapReduce is based on a master/slave architecture. It processes data in a massively parallelized way by dividing a user's MapReduce program, referred to as a *MapReduce job*, into a set of independent Tasks. The MapReduce job is the execution of a Mapper or Reducer across a set of data [49]. However, its tasks' scheduling is usually controlled by elected nodes [178], potentially replicated [93,139] to cope with failures, and it is assumed that nodes are trustworthy and reliable to some extent [50, 181] (See Sec. 2.5 for more details about MapReduce model and Hadoop framework).

However, with the advent of the sharing economy businesses and the globalization of the information, a collaboration between different actors becomes common to provide the so-called *Collaborative Services* or *Cooperative Services* (e.g., collaborative editing systems [112] and Wikipedia [24]).

Within the Data analytics area, collaboration among different companies to collect data in order to analyze and extract information to gain insights and take decisions automatically becomes required [134] as the analysis of this large amount of data is typically expensive, both in terms of bandwidth, storage, and computational resources. Thus, with the advent of peer-to-peer business built on peer-to-peer (*P2P*) network systems [160] (e.g., most crypto-currencies [21]), one can avoid relying on centralized entities that could control the system. In this case, the analysis of data will be distributed to the nodes constituting the P2P system called *peers*. Unfortunately, such systems are inherently untrustworthy and unreliable as the final outcome depends on all peers [194].

In this scenario, peers from multiple administrative domains collaborate in order to provide some services that benefits each participant nodes in the system but with the absence of any central authority to control the nodes' behaviors. Thus, no one can fully trust the other participant nodes where nodes in such environment may deviate from following the protocol for two different reasons. Firstly, nodes may be broken arbitrarily because of component failure or malicious attacks. Secondly, nodes might be selfish and follow their interest in order to increase their utility [3, 4, 41].

This complex environment in which we have to operate, typical of collaborative services, is defined as BAR, i.e., *Byzantine, Altruistic, Rational*, by Aiyer et al. in [4]. In this BAR model, nodes are classified into three categories.

- *Byzantine nodes:* They deviate arbitrarily from the suggested protocol for any reason. They may be broken (e.g., misconfigured, hacked, or malfunctioning) or may just be optimizing for an unknown utility function that differs from the suggested utility function.
- *Altruistic nodes:* that follow the suggested protocol exactly. Intuitively, altruistic nodes correspond to correct nodes in the fault-tolerance literature.
- *Rational nodes:* which are self-interested nodes aim at maximizing their benefit according to a known utility function. The utility function accounts for a node's costs (e.g., CPU cycles, storage, network bandwidth, or power consumption) and benefits. Rational nodes will deviate from following the required protocol if, and only if, doing so increases their net utility from participating in the system.

Under BAR model, the goal is to provide a protocol that guarantees the correct behavior with the presence of both byzantine and rational actors, which is called *BAR Tolerant* (BART) protocol. The BART protocols must satisfy these following properties: (*i*) *Safety*, i.e., only correct results are accepted as final solutions or failure will be received, and (*ii*) *Liveness*, i.e., actions are eventually finished [4].

To analyze a large amount of data, distributed across different actors data-stores, a new kind of MapReduce processing framework needs to be designed. To address this need, we present in this chapter *MARS*, a decentralized *MapReduce for BAR Systems*. MARS is a blockchain-based BART extension to MapReduce framework eligible to work within untrusted peer-to-peer environments.

The rest of this chapter is organized as follows. In Sec. 7.2, we present our system concepts and the main requirements. The details of our protocol methodology and algorithms are provided in Sec. 7.3 . Next, in Sec. 7.4 we evaluate our proposition and compare it to the state of the art. Finally, we conclude this work in Sec. 7.5.

7.2 MARS Concepts

Typically, Hadoop MapReduce relies on a master node that maintains the state of the job, schedules the *map* and *reduce* tasks on worker nodes and reschedules failed tasks. Moreover, the Hadoop distributed file system *HFDS* has a master Namenode,

which maintains the system namespace metadata to keep track of where data is stored in the cluster of slave nodes (more details in Sec. 2.5).

This traditional approach works well in trusted environments but is a severe limitation under the BAR model. The collaboration is limited by the trust problem as these cooperated enterprises could be competing against each other in the same market. Thus, each participant company will try to gain the maximum dividend from the analysis of the big data while minimizing their resources consumption.

For these reasons, there is a need to find a new way to perform distributed big data computations across multiple collaborated actors, but still have the trust with the computation result. To that aim, we propose MARS, a new MapReduce framework that is able to work in such a cooperative environment and tolerate both f malicious attackers and unbounded rational nodes. In MARS, we replace the centralized entity by a fully distributed mechanism robust to both byzantine and rational actors (i.e., no master node is available). To that aim, MARS is based on a decentralized deterministic, yet unpredictable (i.e., the scheduling of MapReduce tasks is unknown by the actors before starting the computation) MapReduce task assignment by the use of the Blockchain technology.

In MARS system, many participant enterprises are collaborating and each enterprise is represented with its own controlled computational resources, connected using privileged peer-to-peer mechanism. Each enterprise has nodes that collaborate to analyze a shared dataset which is considered secure against tampering. The group of enterprises is known and considered static; it does not evolve over time. Thus, once a group of enterprises establishes a collaboration, no one else will join it. Moreover, each enterprise's node has a unique identity correlated to a secure cryptographic key, issued by a known certification authority and linked to a participant enterprise. This key is used to sign messages exchanged in the network.

Byzantine and Rational nodes may return incorrect results for the tasks they are assigned to. However, the system doesn't know which nodes have such behavior and do not know the correct result of a task. Thus, to tolerate f byzantine nodes and an unbounded number of rational nodes, MARS uses the replication and voting mechanism, for which to have a correct result, each task is assigned up to $(2f + 1)$ nodes [97, 141]. Each task is considered completed when *consensus* is reached with $(f + 1)$ similar result [45, 106, 137].

MARS is targeting untrusted distributed systems. Thus, it cannot rely on the traditional MapReduce data storage systems that are usually managed by a reliable NameNode. Instead, MARS uses the Blockchain assisted distributed storage system proposed by Kumar and Rahman [95]. With this solution, the NameNode of MapReduce becomes distributed and replicated to make it BAR tolerant. The nodes that constitute the MARS system are then used as storage units connected in a peer-to-peer way and accessible by every participated node while all storage related meta-data are managed with the Blockchain maintained by all nodes.

7.2.1 MARS Assumptions

To make MARS model feasible, the following assumptions are considered. First, a set of different enterprises are collaborating to perform data analytics. We assumed that the participating enterprises in MARS are known before starting the computations and each enterprise's node has a unique identity correlated to a secure cryptographic key, issued by a known certification authority and linked to a participant enterprise. This key is used to sign messages exchanged in the network.

Moreover, each enterprise has the same number of nodes, so no enterprise is over-represented or under-represented in the voting process to reach the consensus. These nodes are connected using privileged P2P network [160] by low-latency high available bandwidth links.

Furthermore, we assume that there are not any other jobs to be executed on the nodes dedicated to MapReduce jobs. This assumption is made because all participating nodes should respect the agreement made which requires them to use their resources to execute the MapReduce tasks and give the result in a reasonable time. Making external jobs on the same resources would break this promise as rational nodes could exploit that by pretending to use their resources to do other jobs in order to avoid doing the required MapReduce tasks.

Similarly to Aiyer et al. [4], we assume that the rational nodes [164] when they have long-term benefits from participating in the computations as the computation final result will serve all the participants. The rational nodes r will deviate from following the standard execution only if doing this will lead to maximizing their utility, e.g., getting the maximum benefits while reducing costs (e.g., computation cycles, storage, network bandwidth and etc.) [114] otherwise they will continue to follow the protocol. However, rational nodes will not collude together.

On the contrary, byzantine nodes [31,97,115] may deviate arbitrarily from the specific protocol for any reason. They can be random fault during the processing or malicious nodes that can modify data, delete data or even corrupt the results. To reach an agreement among a group of n peers, among which up to f byzantine nodes can depart from the protocol arbitrarily. Abraham et al. in [151] have shown that byzantine agreement requires that ($f < n/3$), but can be solved if ($f < n/2$) [91]. In MARS we consider the assumption that we can tolerate up to ($f < n/2$). With this assumption, we guarantee that an agreement for each executed task will be reached eventually. Moreover, both byzantine and rational nodes are accountable for the decision of non-executing or a task or for their misbehaving, that would constitute a Proof of Misbehavior (PoM) proposed by Aiyer et al. [4].

Although the byzantine nodes can tamper the result of MapReduce tasks, the dataset that needs to be analyzed is considered secure against tampering and modifications of the stored information.

7.2.2 MARS Design Requirements

This section presents the main requirements around MARS protocol design.

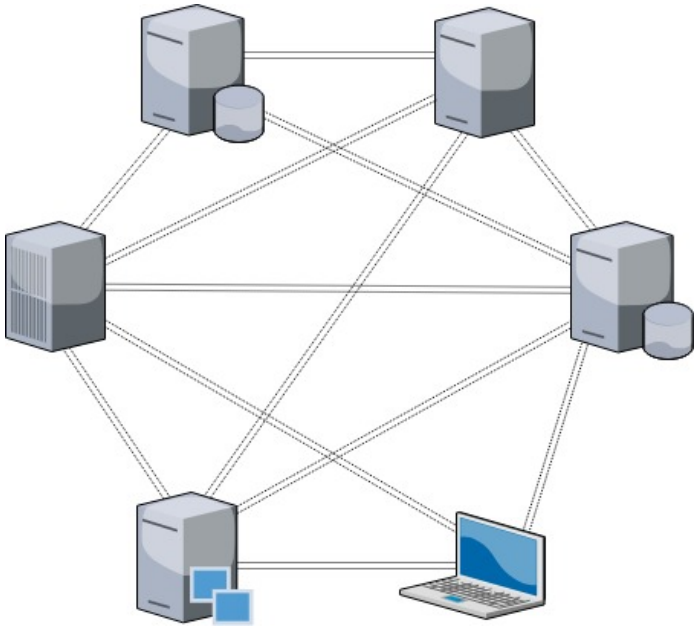


Figure 7.1: MARS P2P Environment

7.2.2.1 Underlying P2P Environment

To achieve our proposed collaborative MapReduce, we consider a cloud-based scenario composed of nodes (also called peers) that belong to a set of different enterprises. These nodes are connected using privileged P2P network by low-latency high available bandwidth links [160] (See Figure 7.1). These peers make a portion of their resources, such as processing power, disk storage or network bandwidth, directly available to other network participants, without the need for central coordination by servers or stable hosts.

In this distributed environment, the participated nodes need to communicate together to agree on the tasks assignments, to exchange the MapReduce intermediate results, and to reach the consensus on the final results.

This underlying P2P network supports the broadcast messaging. When a node sends a broadcast message, this message is given to the P2P network to be delivered. It is assumed that the underlying network will eventually transfer the message with the same content as sent, to all connected nodes (yet this message may take arbitrarily long to arrive). When a message is transferred to these nodes, it is said that the message is received by them.

However, with fault-tolerant distributed systems, where nodes can fail during a task execution, a reliable broadcast is needed to ensure the safe transfer of a message [71]. A best-effort broadcast is the weakest broadcast type. It is a form of a broadcast that guarantees reliability only if the sender is correct (i.e., the sender does not crash while broadcasting).

The best-effort broadcast says that every correct node will deliver the message being broadcasted. A correct node here means the node that does not crash and runs as expected. The best-effort broadcast also guarantees that there is no duplicate

message and no message being delivered without being sent before. Thus, the best-effort broadcast just uses the reliable links for broadcasting without any additional feature.

However, in MARS, the number of participating nodes is assumed to be bigger enough comparing with the number of byzantine nodes ($f << n$) and thus there is no need for stronger broadcast (such as reliable broadcast, uniform reliable broadcast, etc.) to ensure that a consensus will be reached.

7.2.2.2 Blockchain Technology

A blockchain is a distributed database of records or public ledger of all transactions or events that have been executed and shared among different participating parties [192]. Each transaction in this ledger is verified and confirmed by the consensus of a majority of the system's participants. Whenever information is registered, it cannot be erased. The blockchain contains a specific record for every single transaction ever made.

Blockchain technology has specific applications in both financial and non-financial sectors. In 2008, Satoshi Nakamoto published a paper that solved the double spending problem for distributed cryptocurrencies [135], which implementation gave life to Bitcoin. Bitcoin is the most common example that uses blockchain technology. It uses cryptographic proof instead of trusting a third party for a transaction execution by two parties over the Internet. Each transaction is protected by a signature. The transactions are verified and broadcasted through the P2P network to all involved peers.

The blockchain technology has introduced new possibilities to create collaborative services among different enterprises. It has the potential to revolutionize the digital world by enabling a distributed consensus where every online transaction, past and present, could be verified at any time in the future. This could be done without compromising the privacy of the peers involved. The *distributed consensus* and *immutability* are the most important characteristics of the blockchain technology.

Blockchain can prevent the double spending problem. When a peer registers a transaction, this transaction will be broadcasted to the entire network of peers and asks them to determine whether the transaction is legitimate. If they collectively decide that the transaction is in order, then this transaction will be accepted and everyone will update their blockchain. Thus, if this peer tries to spend this request multiple times, other peers on the network will notice, and the transaction will not go through.

All the blocks of the blockchain are linked together by the use of two hash values, current hash computed from the current block and previous hash from the previous block. The previous hash links each block with the next one, which makes the entire blockchain immutable. The data that is stored in the blockchain database is not stored in one single location; instead, it is distributed on every peer in the P2P network.

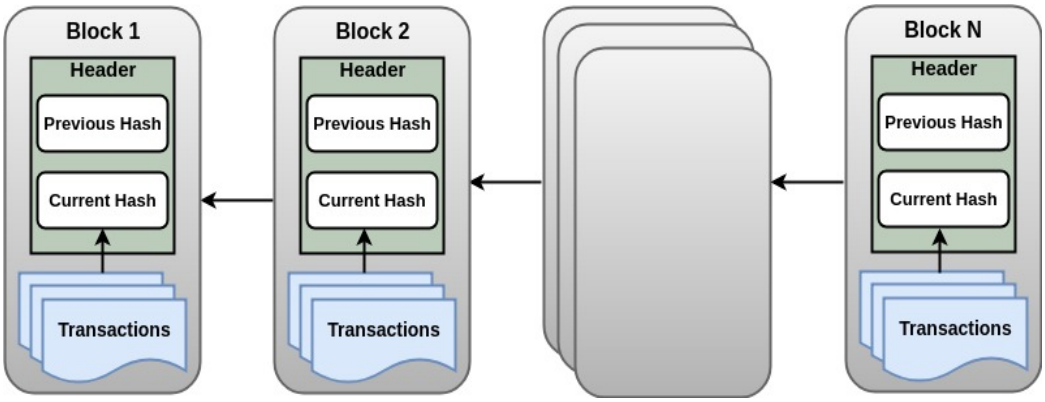


Figure 7.2: Simplified Blockchain Structure

One essential emerging use case of the blockchain technology is the *smart contracts*. Smart contracts are basically computer programs that can automatically fulfill the terms of a contract. They can be executed consistently by a network of distrusting nodes, without the need of a trusted authority. Because of their resilience to tampering, smart contracts are appealing in many scenarios. Open source companies like Ethereum [182] and many companies which operate on bitcoin and blockchain technologies are beginning to support the smart contracts.

The blockchain orders the transactions by placing them in groups named *blocks* and then these blocks are linked to form what is called a *blockchain*. The transactions in one block are considered to have happened at the same time. These blocks are linked to each other in a proper chronological and linear manner with every block containing the hash of the previous block (See Figure 7.2). In addition, each block has a time-stamp that refers to the moment it was created.

To decide which block should be next in the blockchain, a mathematical mechanism known as *proof-of-work* is being used in Bitcoin and Ethereum [135]. With this method, each node generating a block in the network needs to prove that it can provide an adequate computing power to solve this mathematical puzzle. For example, each node has to find a *nonce* value which when hashed with the current data records and the previous hash attached in that block should generate a specific number of leading zeros. However, other alternative algorithms have been proposed in the literature (such as proof-of-stake [192])

Moreover, valid transactions should be registered on the blockchain in a certain time interval instead of every time. This time is called *block creation time* which refers to the average time it takes for a peer in the network to generate one more block in the blockchain.

Whenever there is a new result that needs to be published to the blockchain, the creation of a new block will not occur immediately. Thus, we need to wait to have enough data in order to create a new block as creating a new block imposes an overhead. In particular, we need to keep a lot of information in addition to the new data ensure the resistance of the blockchain such as a cryptographic hash of the previous block and time-stamp. Typically, in Bitcoin, the expected block time is 10

minutes, while in Ethereum it is between 10 to 19 seconds. However, faster block creation times are good because they provide more granularity of information.

7.2.2.3 Blockchain for Decentralized MARS

MARS is presented as a decentralized MapReduce framework eligible to work under the BAR model with no central control and in presence of nodes that cannot be trusted. Thus, we build it around the blockchain as the latter has the required characteristics to ensure the needed trust on the result of MapReduce computation.

The MARS system revisits MapReduce by proposing a new way of tracking jobs with a Blockchain. The Blockchain is used every time a decision has to be taken in the system. Namely, the blockchain has three roles.

(*i*) It is used to keep the metadata of the programs that need to be executed by the MapReduce instances. When someone wants to run a job on data available in the system, it publishes a smart contract with a link to the program that needs to be executed on the data and the information about it. This contract is immutable and can be verified by any node in the system.

(*ii*) The Blockchain is used to distribute the tasks of the various published jobs to the network nodes. For this purpose, a global consensus is required to assign and schedule the tasks on the nodes. All decisions will be published on the Blockchain. Thus, the system can verify that the entire execution of the job has respected the specifications. If the assignment and scheduling algorithms used to execute the jobs are able to ensure both safety and liveness properties (see Sec. 7.3.2), then publishing all decisions in the Blockchain guarantees the correct execution of the programs published in the Blockchain as any misbehavior would be spotted.

(*iii*) MapReduce is all about reading and generating data. However, as MARS targets untrusted distributed systems, it cannot rely on the data storage systems usually used in MapReduce frameworks managed by a reliable NameNode.To guarantee that the storage system can be trusted even though some nodes may provide incorrect data, the blockchain is also used to ensure the integrity of data used throughout the execution of the job. Thus, MARS uses the Blockchain assisted distributed storage system proposed by Kumar and Rahman [95]. With this solution, the NameNode of MapReduce framework becomes distributed and becomes BAR Tolerant. See Sec. 7.2.2.5 for more details. When a consensus is reached on the correctness of a data, it will be published to the Blockchain with the hash of the data the nodes agreed to be correct and the incorrect data will be discovered and ignored.

As all decisions are published on the blockchain, the system can verify that the entire execution of the job has respected the specifications. If the assignment and scheduling algorithms used to execute the jobs are such that they can ensure safety and liveness properties, then publishing all decisions in the blockchain guarantees the correct execution of the programs published in the Blockchain as any misbehavior would be spotted and denounced.

7.2.2.4 Distributed Tasks Assignment and Scheduling

One of the main problems with centralized systems is their vulnerability to a central point of failure if this central node fails, service stops. Moreover, in order to be competitive in today's rapidly changing business world, organizations have moved from a centralized to a more decentralized structure in many areas of decision making including scheduling.

In this context, problems are delegated to lower levels of the organizational hierarchy and solved locally and independently by different entities of the system. The solutions are then coordinated together under a global objective. In reality, the need for such a system was felt since the early days of computing, but throughout the recent years, their applications increased drastically.

In such a distributed environment, there is a need for a new strategy for assigning jobs to nodes that constitute the whole system, or so called *scheduling algorithm*. A scheduling algorithm deals with the allocation of scarce resources to tasks over time [145].

The need for making scheduling decisions in decentralized systems has given rise to a new area, that is called distributed scheduling (*DS*). The distributed scheduling [171] is defined as an approach in which smaller parts of a scheduling problem are solved by local decision makers (i.e., peers) who possibly have inconsistent objectives, but coordinate their sub-solutions through certain communication mechanisms to achieve the overall system objectives.

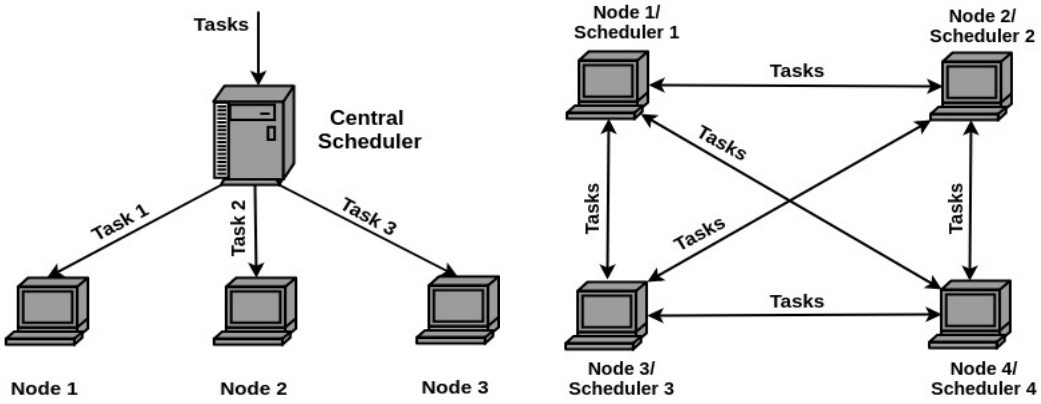
In a centralized environment, a global scheduler issues a schedule for the entire system (See Figure 7.3a [1]). A central node, called master, acts as a resource manager to schedule jobs to all the surrounding nodes that are part of the system. Here, each job is first submitted to the central scheduler, which then dispatches the tasks to the appropriate nodes. This scheduling paradigm is often used in data centers where resources have similar characteristics and usage policies.

Centralized scheduling can produce better scheduling decisions because it has all the necessary information about the available resources in the system. However, it does not scale well with the increasing size of the environment. Moreover, the master itself could become a bottleneck, and if there is a failure, it presents a single point of failure in the system.

Differently from centralized systems, there is no single decision maker in a decentralized system (Figure 7.3b). The participating parties make their decisions locally to solve the smaller parts of the scheduling problem. To determine to which node a specific task is assigned to, each node executes its own scheduling algorithm to decide which task to execute next, as there is not a single master job tracker that schedules the tasks.

[1] Figure 7.3 is inspired by https://www.slideshare.net/sandpoonia/11-grid-scheduling-and-resource-managament.

Because the peers that compose the system prepare their schedules independently from each other, these partial schedules may produce some conflicts. Carefully designed communication mechanisms are required to eliminate such conflicts. There are two mechanisms for communicating between peers [101]: (*i*) Direct Communication in which each peer can directly communicate with other peers for job dispatching, and (*ii*) Indirect Communication where peers can communicate via a central job pool. In this scenario, jobs that cannot be executed immediately are sent to a central job pool.



(a) Centralized Scheduling (b) Decentralized Scheduling

Figure 7.3: Scheduling Paradigms

Another fundamental problem in distributed systems is to achieve overall system reliability in the presence of a number of faulty peers. This often requires network peers to agree on some data value that is needed during computation. This problem is referred to as *consensus*. Examples of applications of consensus include whether to commit a transaction to a database, state machine replication, blockchain, and others.

The global consensus abstraction deals with having many nodes in a network trying to agree on a common value. Initially, each node proposes its value to all other nodes, but when the consensus algorithm terminates, each node should decide the same value. [48].

One approach to generating consensus is for all peers to agree on a majority value. In this context, a majority requires at least one more than half of the available votes (where each process is given a vote).

A consensus protocol tolerating failures (any byzantine failure) must satisfy the following properties: (*i*) *Termination* which means that eventually, each correct node decides some value, (*ii*) *validity*: if all the correct nodes proposed the same value v, then v must have been proposed by some correct nodes, and (*iii*) *Agreement* which means that all correct nodes must agree on the same value. A protocol that can correctly guarantee consensus among n different nodes of which at most f fail is said to be f-resilient [23].

To achieve distributed MapReduce jobs on systems that sustain the BAR environment, the MARS framework supported by the blockchain is proposed with distributed scheduling algorithm to achieve the correct execution of MapReduce jobs, where no master node is available to decide the scheduling of map and reduce tasks on participated peers.

Therefore, to determine to which node a specific task will be assigned, each node executes its own scheduling algorithm to decide which task to execute next, as there is no master job tracker that schedules the tasks. In addition, to force the rational nodes to follow the standard protocol execution, each peer needs to know to which nodes a given task is assigned. In this way, in case of an uncompleted task, it is possible to check which nodes were accountable for that work. The consensus concerning tasks assignment is reached as a result of this deterministic tasks' assignment.

In MARS, byzantine and rational nodes may return incorrect results for the tasks there are assigned to. However, the system does not know which nodes have such behavior neither know the correct result of a task. It is thus necessary to replicate tasks on multiple nodes and to rely on a distributed consensus mechanism to determine, among all potential results, which one is considered as the correct one.

For that reason, to ensure the safety propriety, each task should be assigned to $2f+1$ different nodes, where f is the number of misbehaving nodes [97]. Note that this high replication strategy for each MapReduce task comes at a high cost. However, we can be more optimistic as the byzantine failures can be assumed to be a rare event and the rational nodes do not collude with the byzantine ones and they are forced to follow the exact protocol otherwise they will be denounced with a PoM. Therefore, we can start with only $f + 1$ replicas of the same task [31, 43, 45]. If a consensus could not be reached for one result, more replicas (up to f) are started, until there are $f + 1$ matching replies.

7.2.2.5 BART Distributed File System

The Hadoop Distributed File System (*HDFS*) for MapReduce framework was designed for trusted environments (more details are presented in Sec. 2.5.2). Therefore, the model to store data is built around the NameNode that keeps track of where files are located, called metadata [165], since the data are distributed across different nodes called *DataNodes*. Then, as all nodes are correct, the destination node just needs to make a *get data request* to the source node to get the required data. This NameNode is a centralized entity that can be replicated for robustness reasons but in all circumstances, the NameNode is considered trustworthy and reliable.

However, MARS is working with no central control under the BAR model, where nodes can never be totally trusted and the metadata is not secured. Thus, the MapReduce framework cannot rely on a single node as it could compromise its safety and liveness properties.

To tackle this problem, we use the distributed NameNode solution proposed by Kumar and Rahman [95]. In their paper, the authors modify the standard HDFS [165] such that the functions of the NameNode (e.g., managing information regarding the

distribution of data blocks, replicas management, and block size) are fulfilled in a distributed way without a single point of decision or failures, by the means of a Blockchain for safe distribution of metadata over a distributed system (more details are provided in Sec. 7.3.1).

The nodes that constitute the MARS system are then used as storage units connected in a peer-to-peer way. Data are stored in DataNodes and not on the blockchain. However, at any time, any node can verify the consistency of the stored data to a related event on the blockchain (i.e., each node can check who has registered that data and who has accepted it). This distributed data is available and accessible by all participated nodes and also replicated for robustness reasons. Distributed storage with enough replicas satisfies the requirements of availability and data preservation in case of byzantine DataNodes and crashes.

Moreover, all storage related metadata are managed in a distributed way with the Blockchain. Hence, the destination node makes the first request to the source node that maintains the required data; if this request is valid; the source node will reply with the information. If anything goes wrong, the destination node can use the blockchain to request safely the message from the other node. If a source node was rational or byzantine (trying to deny or send incomplete data to the destination node), this would result in sending the request to blockchain and this node will be punished with PoM.

7.3 MARS Protocol Working Methodology

In this section, we provide a detailed description of MARS protocol design. We enhance the previous Map-Reduce BFT protocols based on voting mechanism to introduce a decentralized deterministic assignment for the tasks, a realistic-threat based data propagation BART protocol, a decentralized BART *map, shuffle and reduce* phases.

7.3.1 MARS Workflow

When a user wants to run a job on the data available in our system, the user creates the corresponding job jar file and then publishes a smart contract with a blockchain with the program and tasks to be run on the data while storing the jar on the distributed filesystem. This contract is immutable and can be verified by any node in the system. The Blockchain is then used to distribute the tasks of the various published jobs to nodes in the network.

After the job is being recorded on the blockchain, it will be executed at a deterministic moment by the participating nodes. Then, each node will check the blockchain for new job execution requests and retrieves the job jar file from the distributed storage to start the execution.

Moreover, as we are under the BAR model, the nodes are untrusted and both byzantine and rational nodes could return wrong results. However, which nodes are correct

and which are not is unknown in advance, thus, each MapReduce task in MARS system will be replicated and assigned up to $(2f+1)$ different nodes [97], where f refers to the number of tolerated byzantine nodes.

For the map phase, the scheduling choice is made following the locality principle. First, to respect the data-locality principle, a list of nodes that are storing the chunk is prepared, then this list is used in the random assignment of nodes described above. Second, if the number of nodes that contains the chunk is less than $(2f + 1)$, the remaining nodes are chosen randomly from all the nodes, without locality considerations. This decision is made to avoid the transfer of large chunks of information across the data center.

Thereafter, the assigned nodes need first the definition of the map tasks. The dataset metadata is retrieved, describing which files are composing the dataset, and the size of the chunks that compose each file. The total number of chunks that constitute the whole dataset is the number of map tasks that have to be executed by the job.

When the mapper node has retrieved the information (data fetching is assumed to be secure), it starts computing the *map* function over it to produce the Key-Value pairs. Then, when mapper has finished, it publishes the result to the system declaring the reach of the result. When consensus is reached by receiving $f + 1$ declarations of the same results, i.e., with the same result hashes, the map task is considered completed. The consensus is made as a transaction to the blockchain, in which the result is identified by its hash. Then, each peer checks the blockchain for updates about new task results and maintain a mapping on the results and their locations in the system.

Even if a byzantine or rational node claimed to have reached a result by copying the hash published on the blockchain by the other mappers, during the shuffling phase the map results are requested and verified by the reducer nodes which makes this node accountable for misbehaving. Rational node won't try to defer the execution of the map computation. Moreover, a byzantine node can simulate to crash, but this action does not hamper the liveness of the protocol as a task's rescheduling will be performed.

When the map phase has finished, the shuffle phase will start. Typically, the shuffling phase has a high impact on the performances, as it requires to move the large results of the map tasks to the Reducers, putting on stress the network.

To ensure the safety property, each reducer must get the correct key-values pairs which need to be the same in all the replicas of the reduce task. Meanwhile, the liveness property must guarantee that the shuffling phase ends in a finite time with all the reducer with their key-values.

All the scheduled reduce tasks are known, as well as the nodes that have been assigned to a map task and which one of the map tasks have produced a correct result. Each reducer has been assigned to process a determined key, and for each key may have multiple values produced by map tasks. All this information is shown in the metadata published on the blockchain. Finally, to move the key-value pairs of each map task we use a threat-based data propagation protocol, in which the mapper is the source node and the reducer is the destination node.

During the data propagation, a source node must provide the information to the destination node, when is required by the destination node. However, in MARS, rational nodes can refrain from transmitting partial information or nothing at all to avoid the data propagation cost. The byzantine nodes can have the same behavior of the rational nodes. Moreover, they can try to create an effective DoS attack requesting to all the nodes all the chunks, to congest the network.

Thus, to create a BART data propagation, we make the source node accountable to give information exclusively to the destination node. The accountability is proved by the context available to everyone, in which the responsibilities are assigned (e.g., a reducer needs the result of a certain mapper). In addition, the destination node has secure access to the metadata of the information it retrieves to verify its integrity (i.e., a hash from the blockchain).

The destination node makes an authenticated get data request to the source node. If the get data request is not legit (i.e., the node is not authorized to make this request), the blockchain publishes the request as a PoM and does not send the data. Otherwise, the source node replies with the information to the destination node that checks its validity. If the data is not valid or the source node refrains from sending the data, the destination can use the blockchain to request safely the message from the other node, and a PoM is published denouncing the source node. On the other hand, if the destination node does not make the request to the source node, a PoM is published denouncing the destination node.

By doing this, a rational mapper will not deny or make an incomplete reply to the legitimate reducer as it will trigger the sending of the request to the blockchain. A byzantine source node cannot give incomplete or incorrect information to the legitimate destination node, as it will constitute a PoM against this node. A byzantine mapper can exploit the data propagation protocol to slow down the system, but it can't affect the safety or liveness property. If it tampers with the key-value pairs the protocol will create a PoM, as in case the information is missing or is refused.

If a mapper crashes, another deterministic assignment is used to assign another mapper (i.e., mapper have the same data as the failed one) to the reducer. However, if all the correct mappers for a specific map task failed, a failed job will be declared.

Finally, the reduce phase takes a couple $< Key, List < Values >>$ to produce a unique result for each key. For the reduce phase, the scheduling algorithm is based on the deterministic node assignment up to $2f + 1$, similar to the one used in the map phase, but without the locality principle.

When a reducer has completed a reduce task, it broadcasts the hash of this result to all peers and when $f+1$ same results are collected for a reduce task this is considered completed. However, if a rational reducer avoids the computation of the reduce task, it will be accountable for not publishing the result at the end of the reduce task. A rational node also will not support any broadcasted result that does not have accumulated more than $f + 1$ votes without making the computation on its own, as it cannot know if the result has been published by a byzantine node. Furthermore, a byzantine node cannot affect the safety principle, as the voting mechanism is applied, nor the liveness principle, as a crashed node is re-assigned and even in case

of f byzantine nodes there will be $f + 1$ altruistic and rational nodes that will get to the final result.

Eventually, the MapReduce job ends successfully with a result or with failure, and its state and history of published results are always available on the blockchain. The job ends successfully when f+1 different nodes declared the job finished with success. This is triggered when at the end of the reduce phase of the job, more than $f + 1$ different result confirmations are provided for each key produced by the previous map tasks. The job failure is declared when $f + 1$ different nodes declared the job failed (i.e., too many nodes crashed and it is impossible to schedule the map tasks on $2f + 1$ different nodes).

At the end of the jobs, the clients can access to the results by copying the information using the data propagation protocol described above, in which all the nodes are considered legitimate destinations of the latest results of the last phase. This motivates the rational clients to keep a copy of their achieved results as long as requested.

7.3.2 Unpredictable Deterministic Assignment

As there is not a single master job tracker that schedules the tasks to workers, to determine to which node a specific task will be assigned, each node will execute locally its own scheduling algorithm to decide which task to execute next. Moreover, each node in the MARS system will have information about all tasks' assignment.

The randomized choice, for which tasks to be executed whenever a node is free, would have been made randomly among all the ready tasks. Random scheduling works well with the presence of only byzantine nodes. However, adding rational nodes to the system will impact the liveness property. Thus, there is a need for a new strategy to tolerate rational behavior by forcing rational nodes to follow the standard protocol. Moreover, the resource consumption will not be optimal, as we could have had cases in which a task will be repeated unnecessarily even by all the participant nodes, as they independently decided randomly.

To solve these problems, we used the guideline proposed in the original BAR paper [4] to minimize the uncertainty and move to a deterministic scheduling algorithm that assigns each task to a fixed number of nodes. Thus, the rational nodes are accountable for the decision of non-executing the assigned tasks, that would constitute a PoM, as they were assigned provably to them.

Removing random decisions requires the usage of deterministic repeatable algorithms for scheduling. The deterministic scheduling algorithm used in MARS is based on a Pseudo Random Number Generator (*PRNG*) [176], where the PRNG is deterministic and initiated by a seed that is known by all peers in the P2P network.

However, this seed must be unpredictable from any nodes before starting the job execution to prevent the byzantine and rational nodes from tampering the results. The deterministic assignment for tasks will help in accountability where each misbehaved node would be spotted and each participating node can assess the correct assignment of tasks on nodes. This unpredictable deterministic is fundamental to

achieve a true BART protocol that bears rational nodes without the need for using external incentives [4].

As the blockchain information is immutable and accessible by every node, we use the hash value of the current block in the blockchain as a seed in this scheduling algorithm since the current hash value is unpredictable and all peers share the same blockchain.

To ensure the determinism, each peer needs to know to which nodes a given task is assigned. In this way, in case of an uncompleted task, it is possible to check which nodes were accountable for that work. An accountability measure is a way of forcing rational nodes to execute the assigned tasks. Algorithm 5 shows the pseudo-code of our unpredictable deterministic task assignment.

Algorithm 5: Task assignment algorithm

```
Input: Byzantine nodes: f
1          mapper: required number of map workers
2          reducer: required number of reducer workers
3
4  setOfTasks = GetTasks(mapper, reducer)
5  nbMinOfNodes = 2 · f + 1
6  foreach (T ∈ setOfTasks) do
7      PRNG.seed (currentBlockhash)
8      assignedNodes[T] = { }
9      while len(assignedNodes[T]) ≤ nbMinOfNodes do
10         node_id = PRNG.getNode()
11         assignedNodes[T].add(node_id)
```

This PRNG is generating a sequence of numbers that refer to the identity of nodes that are responsible for performing a specific task. The PRNG-generated numbers are random, but they are still deterministic because they are completely determined by the unpredictable seed value. As each node executes this scheduling algorithm, it will know if it is in charge of any task and also all tasks assignments.

7.3.3 Tasks Completion

When a node has completed its assigned job, it broadcasts the hash of the obtained result to all the other nodes in the system (more details about broadcasting are available in Sec. 7.2.2.1). To accept the result of any task, a global consensus is needed. Reaching consensus is a fundamental problem in the MARS framework for both byzantine broadcast and byzantine agreement with the assumption ($f << n$) (See Sec. 7.2.2.4 for more details).

In order to reach a global consensus, each node fetches the output from all task replicas and then chooses the most voted result. In MARS, at least ($f+1$) matching

results for each task should be received to confirm completion of the tasks. This approach is motivated by previous works in the literature [45, 151].

When a global consensus is reached for a specific task, it will be confirmed that the result of the task is the one provided by these $f + 1$ nodes. Thus, other nodes that execute the task can safely stop the execution of the task to free their resources for other tasks. As the results of the task have been globally approved, every node can use the result, meaning that it can safely be used in the following steps of the job, just by checking that the data of the result correspond to a hash value published in the Blockchain are corresponding regardless of from where the data was taken.

In Figure 7.4, we explain in details the task' completion step. Figure 7.4 shows a P2P network with three nodes, namely N1, N2, and N3 that perform the task *T1*. In this system, we assume that in the number of byzantine nodes f is 1. Thus, to be BFT, *T1* should be replicated and executed on at least 3 different nodes.

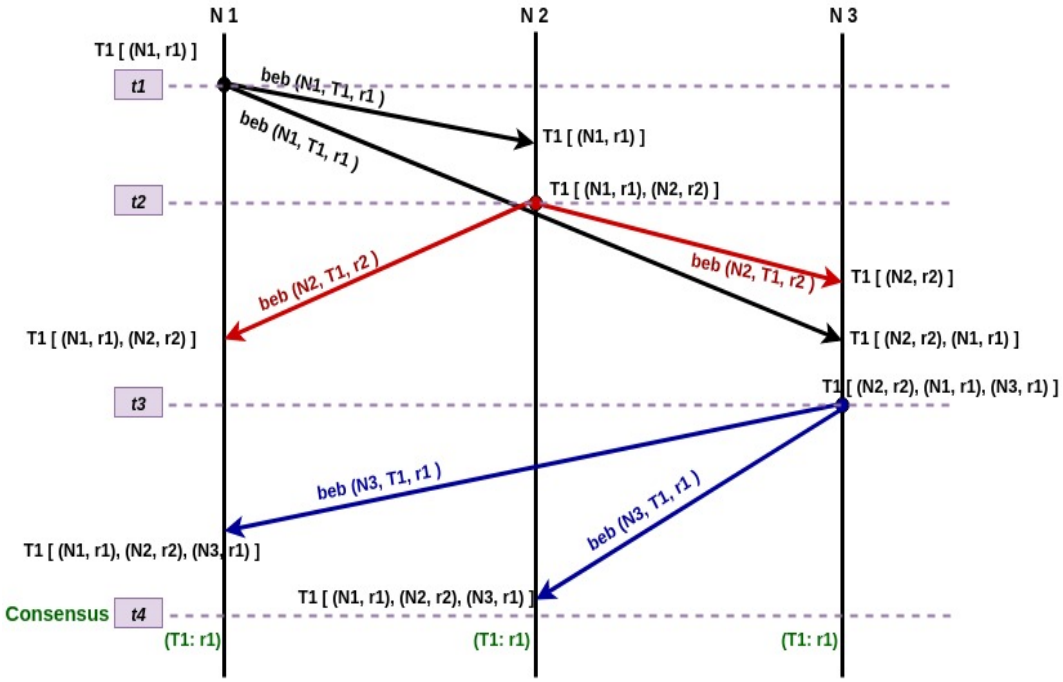


Figure 7.4: **Task confirmation** in MARS System, where *beb* refers to best-effort broadcast

First, at time t1 the node $N1$ finishes this task and broadcasts a message with the obtained result ($r1$) (namely, best-effort broadcast (beb)). This message contains the following information: (i) Node Id that has sent this message, (ii) The Id of the executed task, and (iii) The hash value of the obtained result. Then, this message will be received by other nodes, at different times.

As the tasks assignment is deterministic, each destination node in the system can verify if the task has been executed by the appropriate node. It then saves the message locally otherwise, the message will be ignored and the node that sent the message will be spotted and denounced with a Proof-of-Misbehavior.

At the time $t2$ the node $N2$ finishes the same task, but with a different result ($r2$), assuming that N2 is a byzantine node. Then, a new broadcast will be initiated by this node and will be received by the two nodes and also kept in local storage. Finally, the last node $N3$ completes the execution of this task at time $t3$ and then broadcasts the obtained result ($r1$) to all nodes.

Eventually, at time $t4$, all task replicas are finished. A global consensus will be reached as the majority voting is given the result $r1$ with $f+1$ matching results from N1 and N3 correct nodes (with MARS, rational nodes are obliged to be correct).

Algorithm 6 shows the pseudo-code of our task completion decision algorithm. The $new\_confirmation\_received_T$ refers to receiving a new message with a hash value of result for the executed task from a peer in the system. This received value is similar to another stored result.

Algorithm 6: Task completion algorithm

```
   Input: Byzantine nodes: f
1          runningTasks: list of the ongoing map and reduce tasks
2
3  completedTasks= { }
4  for T ∈ runningTasks do
5      n_confirmation = 0
6      Consensus= f + 1
7      if new_confirmation_received_T then
8          n_confirmation += 1
9          if n_confirmation == Consensus then
10             completedTasks.add (T)
```

A simplistic solution to make MARS BART MapReduce was to replicate and schedule each map/reduce task to $2f + 1$ different nodes. Then, the voting mechanism can be used to agree on a value for each task. However, this solution is very expensive as it replicates everything $2f + 1$ times: task execution, map task inputs reading, message communication, and storage of reduce task outputs.

To avoid this cost, we can be somewhat optimistic and say that the byzantine failures could be assumed as uncommon events, once they are guaranteed to be discovered [45]. Moreover, being optimistic makes sense as we assume that ($f << N$) (See Sec. 7.2.1), so when nodes are chosen randomly it is unlikely to have misbehaving nodes between the elected nodes.

Thus, instead of executing ($2f+1$) replicas for each map/reduce task, we can execute simply the first ($f + 1$) nodes generated by the scheduling algorithm [31, 43, 45] and defer the execution of the last f copies until there is a need. In case of failure to reach the global consensus with the correct result for the same task or if a task timeout elapses, the execution of replicas (up to f) from the deferred nodes that are

already produced with the deterministic assignment algorithm are started until the consensus is reached.

Tasks timeouts are used to avoid starving by waiting a too long time for a task to be completed. When a timeout occurs, the task is canceled on the node and it is rescheduled on another node chosen from the list of nodes generated by the PRNG (in Algorithm 5). If a node selected by the PRNG is not available (e.g., overloaded or already executed the same task) then the node corresponding to another random-generated number will be selected.

Timeouts are triggered observing the blockchain, as to know when a task has not published a result for more than *Timeout* time. Our unpredictable deterministic assignment and scheduling method prevents nodes from providing incorrect results. In conjunction with the timeout, it also guarantees that the task will be eventually completed with a correct value. In addition, our mechanism incentivizes nodes only for their useful work as wrong results are discarded. This means that while byzantine behavior can still be observed– but without impairing safety and liveness– rational nodes will only properly execute the assigned tasks as they have no incentive left in not doing so.

As the result of the task has been globally approved, one of the nodes will publish the hash of this result to the blockchain (See Sec. 7.2.2.2 for more details). Thereafter, every node in the system can check the blockchain for updates about new task results and maintain a mapping on the results and their location in system and the result could be used safely in the following steps of the job, where the node (i.e., DataNode component) retrieves the information and verifies what has been downloaded comparing it with the relative hash retrieved by the node (i.e., NameNode component) from the blockchain.

Here, we present one way where we publish only the hash value of the final result to the blockchain. This strategy is good as we do not generate a lot of blocks and do not save a lot of information about MapReduce jobs as the created blocks are permanent. However, if some nodes crash during the progress of a MapReduce job, the results of these nodes will be lost. This problem could be solved by saving all the information (i.e., publish the results obtained by each node not only the global consensus) to the blockchain instead of the final result. In this way, if a node crash after finishing the tasks and writing the result to the blockchain, there is no need to reschedule this task again but with a cost of saving a lot of irrevocable data in the blockchain.

Furthermore, whenever a new result needs to be published to the blockchain, the creation of a new block will not be immediate. Thus, we need to wait for some time, namely *block creation* time, to have the required data registered as a new block. Thus, increased block creation time will increase the required time to finish MapReduce jobs.

Another important thing to consider is that waiting for $f + 1$ matching map results before starting a reduce task can delay the job completion. One possible solution is to start executing the reduce tasks just after receiving the first copies of the required map results, and then, while the reducer is still running, the consensus is checked.

If at some point it is detected that the input used by the reducer is not correct, the reduce task needs to be restarted with the correct input. However, this solution wastes computing resources.

7.4 Evaluation

7.4.1 Simulation environment

To evaluate MARS, we have adapted the MRSG MapReduce simulator [93] that is built on top of SimGrid [30]. More specifically, we have implemented our task replication and distributed scheduling algorithms and the blockchain in the simulator (see Figure 7.5). In addition, we have developed an analysis framework that automatically reproduces simulation and generates figures.

We compare MARS with the following reference platforms: (*i*) the standard version of *Hadoop*, only crash-fault, as reference of best-performance, (*ii*) the *BFT Hadoop* version, with $f+1$ replicas, proposed by Costa et al. [43][2], as it is the nearest platform to ours while it does not support rational behavior and is centralized, (*iii*) finally a replica-state-machine version of MapReduce execution, in which all the peers perform all the map and reduce tasks independently and use majority at the end to get the right result. This final version is called *blockchain*.

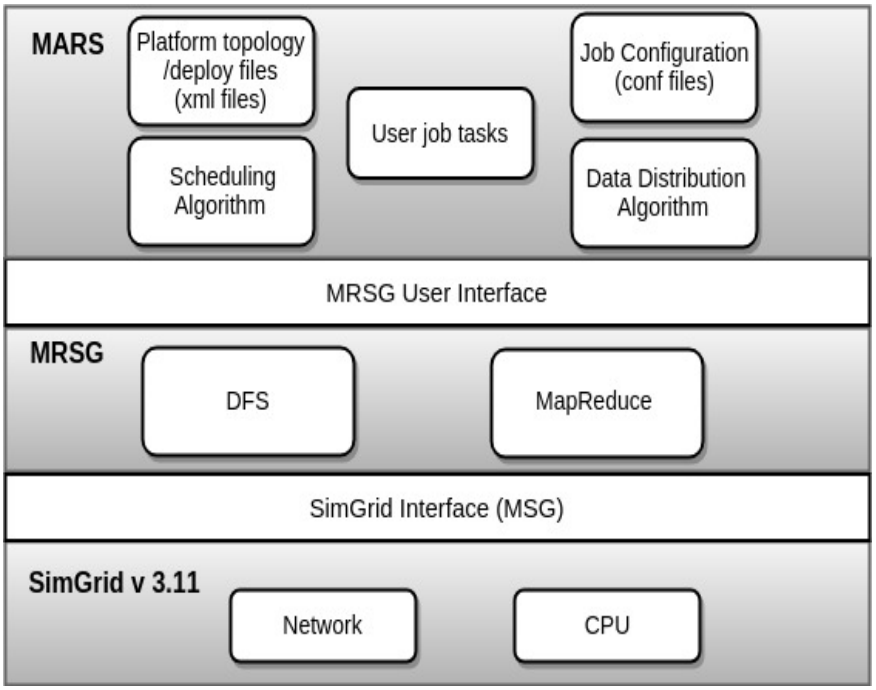


Figure 7.5: MARS Simulator

[2]It was modified to have a majority voting, task replicas scheduling and retrieval of intermediate results by their location

Moreover, we use the following versions of MARS: (*i*) The optimistic MARS, in which each peer participates with one node that acts as master and worker, and an external blockchain service with replication factor of $f + 1$ and (*ii*) the unoptimized version, called (*pessimistic MARS*), with the replication factor of $2f + 1$. Byzantine nodes are defined randomly in the phase of initialization where the user defines in the configuration file the percentage of Byzantine nodes in the system, and then the system instantiates them.

The byzantine nodes make a vote with a byzantine result[3] while the correct nodes make a vote with a correct one. A task is considered completed when the master receives f+1 correct result messages, terminating the task. In case it receives a byzantine result, the task is re-scheduled on a different worker, that never executed the task before.

The whole implemented code for the experiment is opensource and available on GitHub[4]. Table. 7.1 summarizes the simulation parameters we use in this evaluation.

Infrastructure	
Number of nodes	**10**
Fraction of byzantine nodes	$\{0.0; 0.1; 0.2; 0.3; 0.4\}$
Executed job	
Sum job	
MapReduce parameters	
Number of Reduce tasks	1
Chunk size	64 MB
Number of Map tasks	40
Number of DFS Replicas	Infinite
Number of Map slots	1
Number of Reduce slots	1
Map task cost	100 gigaFLOPS
Reduce task cost	500 gigaFLOPS
Blockchain parameters	
Block size	1000 transactions
Block period	$\{0.5s; 15s\}$

Table 7.1: MARS Simulation Parameters

[3]Other byzantine behaviors such as not giving the result or tampering the stored data chunk are not considered in this work

[4]MARS simulator on https://github.com/quellobiondo/BAR-decentralized-MapReduce-sim.

7.4.2 Work reproducibility

As the reproducibility of the research experiments has become fundamental in research, we built our simulator on this idea where Simgrid is an efficient tool to make a reproducible simulation as its executions are deterministic, giving always the same communications patterns and results.

Moreover, Docker containers are used to isolate the environment from the execution. Each different platform has its own Dockerfile for the creation of its container and each platform has been configured to trace information during the execution for later analysis (namely job duration and resources usage). For results analysis, a python script is provided to generate the CSV files which are visualized by R scripts included in the project that is available online.

7.4.3 Simulation results

To assess the performances of MARS, we study the following four metrics: (*i*) the job completion time; (*ii*) the CPU usage; and (*iii*) the bandwidth usage. We study the evolution of these metrics against two important factors: the fraction of byzantine nodes in the system and the blockchain block duration period as this time affects the general performance of the job for the platforms that depend on the blockchain to register metadata. The values for block creation duration are as the one delivered by Ethereum [182]. The executed MapReduce job is called *sum*: it emulates the extraction of information with just one key and its combination in the result. The job configuration is shown in Table 7.1.

However, while the platform can tolerate up to f byzantine nodes[5][6], generally, none of them is present in the system. So we configure the parameter *byzantines* with the percentage of byzantine nodes that the platform has to tolerate with the percentage of byzantine nodes that are really present in the system. In particular, we have two scenarios: (*i*) the scenario in which all the nodes are not byzantines. The configurations that terminate with (*no_real_byz*), and (*ii*) the second scenario has the maximum number of byzantine nodes that should be tolerated by the platform.

Fig.7.6 shows the evolution of the completion time with the fraction of byzantine nodes in the system, for two different block creation duration periods. The configurations sumzeroblocktime and sumzeroblocktime_no_real_byz are with (0.5 s) block creation while the other configurations (i.e., sum and sum_no_real_byz)are configured with 15 s[7]. As the vanilla MapReduce is not designed to support byzantine nodes, it is only evaluated for 0% of such nodes. Furthermore, for the other platforms, we cannot go over 40% of byzantine nodes in our simulation as one of MARS assumptions is that we can tolerate up to ($f < n/2$) byzantine nodes (see Sec. 7.2.1 for more details).

[5] Byzantine nodes are assigned at the beginning of each execution using the PRNG.

[6] Each execution is repeated with 3 different seeds for PRNG and the average of the performances is reported.

[7] In this figure *zeroblocktime* configuration is 0.5 s while the standard one is 15 seconds.

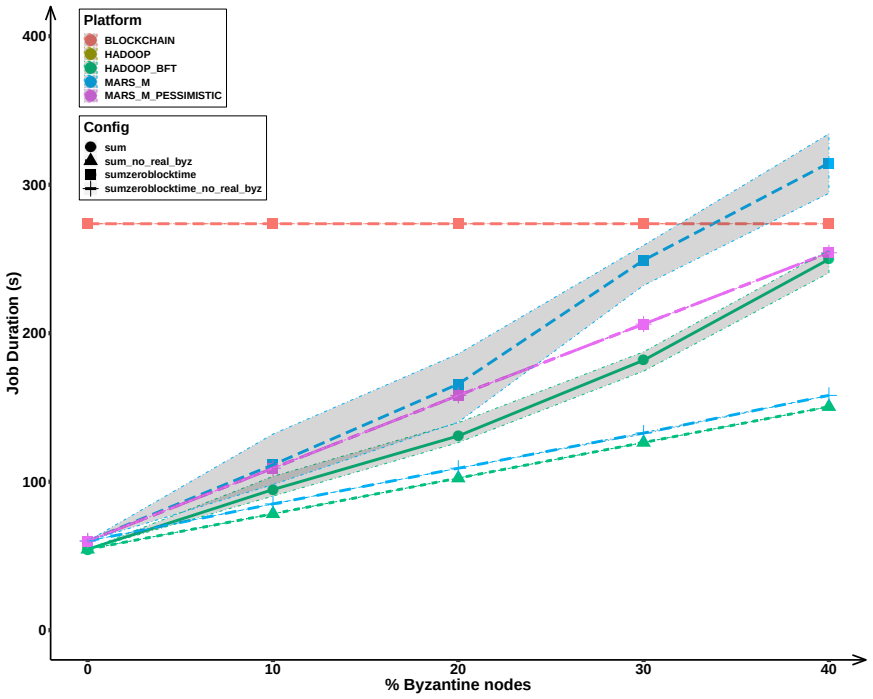


Figure 7.6: Job completion time

As a result, we have the following configuration: (*i*) *sum*: sum job with 15 s block creation time and with the maximum number of byzantine nodes that should be tolerated by the platform, (*ii*) *sum_no_real_byz*: sum job with 15 s block creation time and with no byzantine nodes presented in the system, (*iii*) *sumzeroblocktime*: sum job with 0.5 s block creation time and with the maximum number of byzantine nodes that should be tolerated by the platform,, and (*iv*) *sumzeroblocktime_no_real_byz*: sum job with 0.5 s block creation time and with no byzantine nodes presented in the system.

The completion time shows a linear increase with the number of byzantine nodes for Hadoop_BFT and MARS as these two solutions adapt the number of task replicas to the number of byzantine nodes. The Blockchain proposition is insensitive to the number of byzantine nodes as it always fully replicates the tasks to all nodes. By design, as soon as MARS encounters at least 50% of byzantine nodes, it replicates the tasks to all the nodes in the system, which explains why completion time saturates after 50% of byzantine nodes. With Hadoop_BFT, the maximum is reached only when all nodes are byzantine. The difference with MARS is because MARS is more conservative than Hadoop_BFT and always makes $2f + 1$ task replicas that guarantees that a correct solution will always be found (as long as there is less than 50% byzantine nodes) without having to reschedule tasks. On the contrary, Hadoop_BFT starts with $f + 1$ task replicas and make new replicas if it cannot reach a consensus.

However, in case we have byzantine nodes in the system, the tasks need to be re-executed in Hadoop_BFT and MARS optimized. While Hadoop_BFT has a faster response time to the event, scheduling immediately another task when it receives a wrong result, MARS optimized needs to wait until the tampered result is published with a block, delaying the task rescheduling. For this reason, its completion time

increases sharply even with moderate percentages of byzantine nodes.

The Hadoop and Hadoop_BFT solutions should have the same completion time in absence of byzantine nodes. However, the implementation of Hadoop in MRSG is not optimized. In our implementation of Hadoop_BFT we made this optimization, which explains the difference. We decided to keep the Hadoop implementation of MRSG as it is, instead of modifying it to have our optimization, to serve as a reference point for all research work already done with MRSG.

To see the impact of the block creation time in the execution of MARS, Fig 7.7 shows the distribution of the waiting times in the ten nodes cluster for Hadoop and MARS. For Hadoop (see Fig. 7.7a) the nodes don't have to wait a lot of time to get a feedback. For example, at the second 55 the job ends, the latest node publishes the result and immediately the master declares the job as finished. On the contrary, for MARS (see Fig 7.7b) at the second 78, the node (*graphene-2*) has to wait 30 seconds to know that the job has been completed. First, there is no master, so to know that the job is finished, all the results have to be published to the blockchain. Publishing the results to the blockchain implies delays, as the transaction has to wait until it is included in a creating-block. This block will be created after a time that is necessary for the consensus to be applied in the blockchain system.

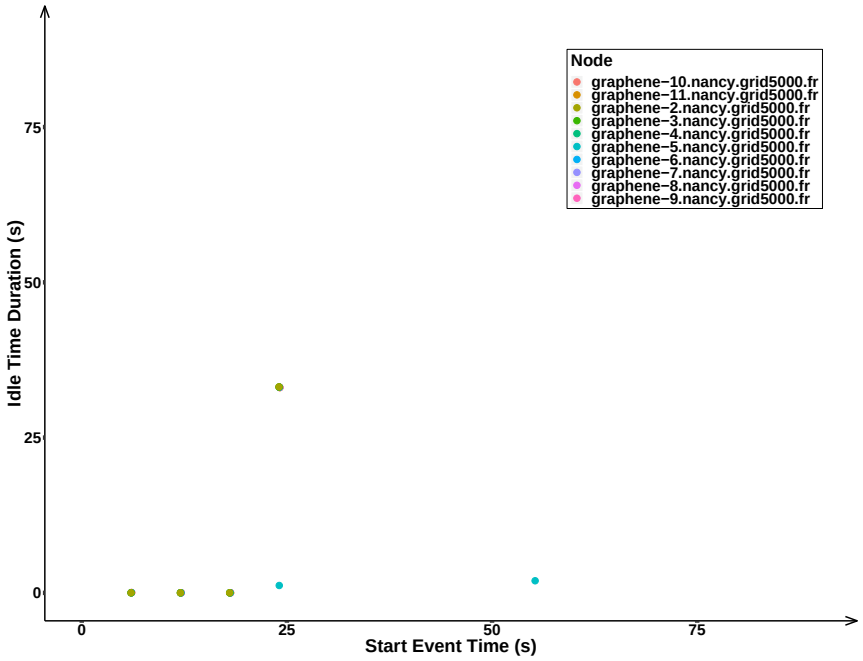
Fig. 7.8a shows the total amount of CPU time consumed to complete the job, not taking into account the CPU time used for Blockchain related computations. As MARS and Hadoop_BFT replicate the tasks proportionally to the number of byzantine nodes, we can observe that the CPU usage increases rather linearly with the fraction of byzantine nodes in the system (with almost the same behavior for the MARS and Hadoop_BFT). As the Blockchain solution always replicates the tasks to all nodes, its total CPU consumption is independent of the number of byzantine nodes. Moreover, the replicated execution of all tasks has fewer power requirements that re-executing the tasks as in MARS and Hadoop_BFT.

For the Bandwidth usage (see Fig7.8b), Hadoop solution is also independent of the number of byzantine nodes. Regarding optimized MARS, as it needs to reschedule tasks many times until consensus reached which implies retrieving required data to do the new rescheduled tasks, thus consuming more bandwidth. On the contrary, with MARS pessimistic solution, as we start with $2f+1$ replicas from the beginning, there is no need to do rescheduling to reach the consensus.

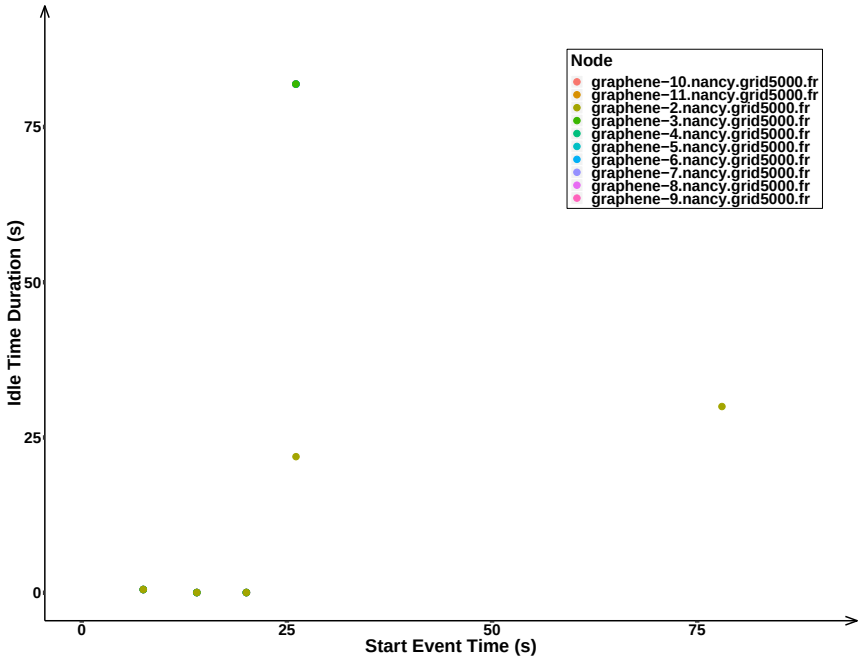
7.5 Conclusion

In this chapter, we have proposed the first collaborative MapReduce for BAR systems, dubbed MARS. This decentralized framework is based on an untrusted P2P network of nodes instead of using the Client-Server paradigm, supported by Blockchain/DLT to have a consistent safe view of metadata. The framework tolerates up to f byzantine nodes and an unbounded number of rational nodes.

The evaluation of MARS grants the execution integrity in MapReduce linearly with the number of byzantine nodes in the system. For 10% of byzantine nodes tolerated in



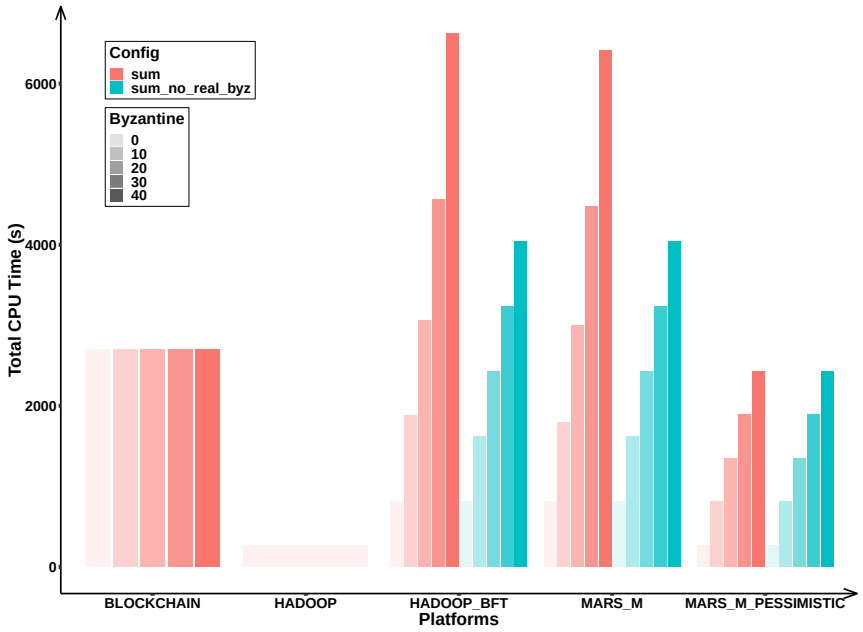
(a) Hadoop



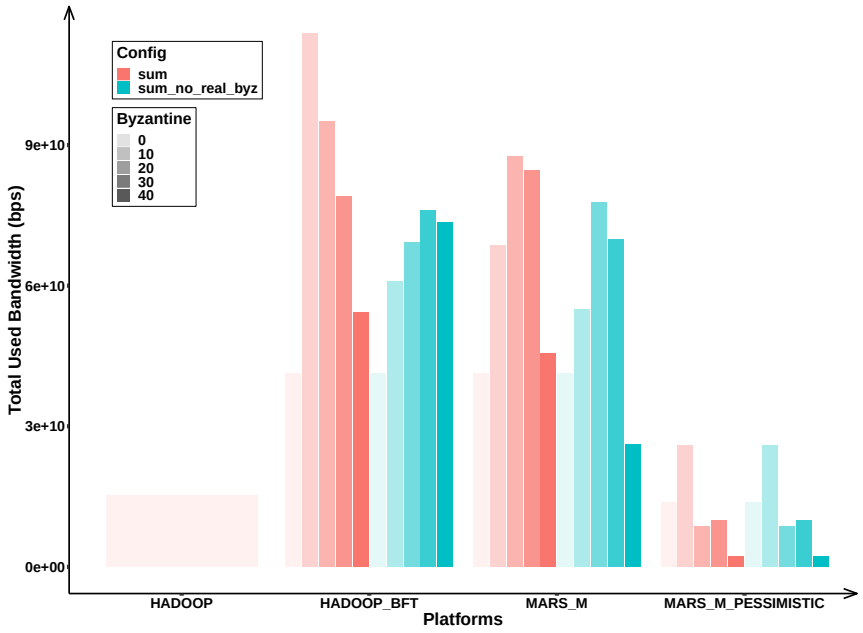
(b) MARS

Figure 7.7: IDLE time

the system, MARS complete the job in twice the time of the standard MapReduce, as a result of replication approach. The resources used are dependent on the byzantine nodes as it implies replication and rescheduling.



(a) CPU Usage



(b) Bandwidth Usage

Figure 7.8: Resources consumption per platform

In future work, MARS is a preliminary work that should be studied in cases in which the storage is limited, and the resources provided by the enterprises are heterogeneous. In such cases, a scheduler should not blindly assign the tasks equally or based on the locality principle. Instead, it should find the best nodes to perform the tasks while providing fairness in the assignment. Thus, the placement of data should be

explored as where we place the data can affect the performance as all intermediate data need to be moved from mapper nodes to the reducer nodes that consume both bandwidth and time which could increase the needed time to complete the requested MapReduce job. However, the problem of data placement is not solved because of time limitation and is left to future work.

8 Conclusions and Future Work

Contents

8.1 Summary and Conclusions

The network is becoming more and more ossified with the increased amount of middleboxes. Thus, network operators are facing great challenges to incorporate new communication technologies and to keep up with the increasing and evolving demands of new network services addressing the emerging use cases. The necessity to innovate, develop, test, deploy, integrate new network services, upgrade existing ones and configure them using vendor-specific tools at a rapid pace have made networks more complex and costly to operate.

The NFV paradigms promise to transform the network architecture by *softwarizing* it. It has been proposed to eliminate the previous challenges by decoupling network functions from the proprietary hardware, thereby providing a cost-effective softwarized approach for network services development, deployment, integration, management, and upgradation. In the NFV environment, the network operators and service providers can run software-based functions (called virtual network functions) on Commercial-Off-The-Shelf based appliances instead of using expensive purpose-built hardware, which actually reduces CAPEX and OPEX costs.

The appearance of NFV makes it possible for network operators to achieve strong network flexibility and fast new service deployment cycle. In this way, network operators are able to satisfy the constantly growing customer requirements and reduce the network operation and maintenance cost at the same time. Nevertheless, challenges always coexist with NFV benefits such as performance issues. Due to the use of commodity hardware, it is still difficult for the virtual network functions to offer comparable or even better performance than the functions running on dedicated hardware.

Moreover, as network functions for a given service are separated from the underlying hardware and implemented as VNFs, a new critical problem appears of where to place these functions such that the service requirements can be satisfied. This problem is called VNF resource allocation or placement problem. It is usually hard to find the optimal solution for VNFs placement problem and it is more challenging when the service requests come in an online manner.

Ensuring end-to-end service continuity in the presence of physical failures is another important concern that should be taken into account when moving to the NFV environment as the commodity hardware that host VNFs are more prone to failures compared with dedicated hardware and as virtualization also introduces more vulnerabilities.

In this thesis, we started by presenting the basic concepts of NFV and SFC (including motivation, standardization efforts, architecture, and challenges). Then we showed the main works in the literature and provided a new taxonomy for the main efforts that have been done in the past for the placement problem. Thereafter, we emphasized on the benefits of smart placements which keep into account resiliency in addition to the usual performance metrics in Chapter 4. We advocated that resiliency should not be left to the orchestrator but should be carefully embedded to the criteria considered at the virtual-to-physical placement phase. We showed that just by taking smart placement decisions, we can ensure that continuity of the provided services in the presence of simple physical network failures, and thus avoiding the need for the orchestrator intervention.

Based on the insights obtained for Chapter 4, we proposed in Chapter 5 a deterministic solution to deploy online tenants' service function chain requests in public cloud data centers to guarantee some robustness level R in the presence of independent fail-stop physical node failures. Our proposed algorithm solves the placement problem in two steps. In the first step, the algorithm tries to map the functions of the requested services to the convenient physical hosts using two different solutions, namely optimal and greedy solutions. Once the placement of functions is decided, a feasible path is decided in the second step using the shortest path algorithm. We extensively evaluated our solution on very large data center networks (up to 30,528 nodes), differently from the state of the art, to assess the feasibility of our proposition in very large scale data-centers.

The deterministic solution proposed in Chapter 5 is suitable for the situation in which the service providers have a full control and knowledge of the underlying infrastructure. Thus, in Chapter 6, we have moved beyond this deterministic solution and proposed a stochastic solution to solve the placement of topology-oblivious SFC demands such that placed SFCs respect availability constraints imposed by the tenants. In this solution, we took the availability of the network components into account and solved the placement problem with the objective of maximizing the obtained availability level of the placed services. A large data center topology is also used as a referenced topology in this chapter.

In Chapters 4, 5 and 6, the targeted environment is considered to be trustworthy and controlled in a centralized manner by the infrastructure owner. However, when moving to a cooperative environment where many enterprises collaborate together to provide customer services namely MapReduce applications, there is no central authority that controls the participated nodes' actions. Thus, these services should be provided in a fully distributed and decentralized manner.

In such scenarios where the environment cannot be trusted and participants have no knowledge or control over this heterogeneous environment, a different problem appears of how to trust the result with the presence of byzantine nodes (i.e. nodes

that can have an arbitrary behaviour where they could even send incorrect messages) and rational partners that have self-interested behavior, which is called *BAR* model. Thus, we need to move from the network level to the application level and adapt the application with the presence of these new challenges of trust and resources heterogeneity.

Therefore, we introduced in Chapter 7 a blockchain-based decentralized MapReduce framework, dubbed *MARS*, eligible to work within untrusted peer-to-peer environments. To achieve this goal, an unpredictable and deterministic distributed scheduling algorithm is proposed to achieve the execution of jobs.

To evaluate the MARS framework, we adapted the MRSG simulator that is built on top of the SimGrid simulator. Furthermore, we compared its performance with Hadoop BFT proposed in the state of the art, as it is the closest solution to our work. The completion time and resource usage of the optimized version of MARS was close to the optimized Hadoop BFT; it increased linearly with respect to the number of tolerated byzantine nodes, demonstrating the feasibility of the framework with acceptable overhead for limited values of byzantine nodes expected in the system. To the best of our knowledge, MARS is the first cooperative MapReduce framework for decentralized BAR systems.

However, MARS is a preliminary work that present a first MapReduce framework eligible to work in decentralized environment. In the current version, the problem of trust is solved while problem of resources heterogeneity is not solved because of the time limitation and it is left to the future work.

8.2 Future Works and Perspectives

Here, we describe some of the future work for our contributions in this thesis along with possible future directions of the NFV paradigm.

Concerning Chapter 5, we have provided a solution that works efficiently with multi-tier data center topologies where fault domains could be defined easily. However, this solution should be adapted to consider any random topology or, more specifically, server-centric data center topologies such as Dcell and Bcube topologies where it is not possible to define the fault domains. Furthermore, the stochastic solution adopted in Chapter 6 needs to be investigated more as it is restricted to a single data center topology, namely the Fat-Tree network topology.

Moreover, the trade-offs between robustness and other important metrics (such as cost, latency or energy consumption) could be studied to show if resilient placements affect the remaining metrics. Formalizing the problem as a multi-objective optimization one as well as considering more specific or realistic service requests instead of synthetic ones for the performance evaluation could be other important extensions. In addition, we could avoid rejecting tenants SFC requests by queuing the unsatisfied requests for some time (some networks resources might be released during this time). By doing this, the obtained acceptance ratio could be increased.

Furthermore, all placement decisions that we have proposed are static where after a placement decision is taken by the algorithm, the placed functions of the accepted SFC will remain at their places and no re-placement or migration decision will be taken later. As each accepted SFC leaves the network and releases the reserved resources when the service time finishes, a dynamic placement algorithm within re-calculation of services VNFs placement could lead to a higher acceptance ratio. However, this dynamic solution may arise new challenges to other performance metrics such as the guaranteed latency as a result of the VNFs migration, especially for the short-term services.

Another extension to our work regarding the SFCs placement problem, would be using the failure prediction model to create a new replica in order to guarantee the continuity of the provided services. In this solution, for each tenant requested SFC we place only one replica for each function of this service chain. Thereafter, based on a periodical failure prediction, we can predict that at some point in the near future which physical nodes are more likely to encounter failure. Then, all the affected placed functions should be migrated to safe nodes. To that aim, we need to build good models that quantify the risk of failure for any node in any moment in time, using machine learning techniques, and then use this information to take VNFs re-placement decisions.

However, the success of these predictive failure models depends on the following conditions: (*i*) Having the right data available to build a failure model, as we need enough historical data to capture information about events that lead to failure. In general, more data is not always better, and (*ii*) How long in advance the model should be able to indicate that a failure will occur.

With regard to the work presented in Chapter 7, named MARS, is interesting to study the cases in which the storage is limited, and the resources provided by the enterprises are heterogeneous. In these cases, we can consider a scheduler that does not blindly assign the tasks equally or based on the locality principle (that is considered in Map phase), but also it should decide which are the best nodes to perform the tasks, while providing fairness in the assignment at the same time.

Beyond the topics discussed during this thesis, NFV is still in the infancy. Therefore, works will focus on the extensive activity around NFV soon as there are a plethora of new topics to be explored. For example, considering the combination between SDN and NFV as SDN is currently attracting significant attention from both academia and industry as an important architecture to managing large scale complex networks.

Increasingly, researchers are focusing nowadays on applying this integrated architecture into other scenarios such as 5G, Internet of Things (IoT) and Information-Centric Networking (ICN) areas. Security and privacy issues are also pivotal to the success of NFV and still need more research.

A Appendices

Contents

A.1 Linearization of Nonlinear Constraints

Here we show how to linearize Constraints (6.16) and (6.17). For constraint (6.16) we introduce auxiliary binary variables $\varepsilon_{p,i}$ for $i \in [0, n]$ where n is the maximum number of host nodes in one pod and $\sigma_{p,j}$ for $j \in [0, m]$ where m is the maximum number of ToRs in a pod. $\varepsilon_{p,i}$ refers to each possible number of used hosts under each pod, and $\sigma_{p,i}$ refers to each possible number of used ToRs under each pod. $\forall p \in P$:

$$\varepsilon_{p,i} = 1 \;\; if \; \varepsilon_p = i, \; else \; \varepsilon_{p,i} = 0, \tag{A.1}$$

$$\sigma_{p,j} = 1 \;\; if \; \sigma_p = j, \; else \; \sigma_{p,j} = 0. \tag{A.2}$$

Constraint (A.1) ensures that this variable equals to 1 when the total number of used host is equal to i else it is equal to 0. Then, Constraint (A.2) ensures that this variable equals to 1 when the total number of used ToR is equal to j else it is equal to 0.

$$\sum_{i=0}^{n} \varepsilon_{p,i} = 1 \tag{A.3}$$

$$\sum_{j=0}^{m} \sigma_{p,j} = 1 \tag{A.4}$$

Constraints (A.3, A.5) and Constraints (A.4, A.6) ensure that exactly one of $\varepsilon_{p,i}$ / $\sigma_{p,i}$ binary variables is equal to 1 respectively.

$$\varepsilon_p = \sum_{i=0}^{n} i \cdot \varepsilon_{p,i} \tag{A.5}$$

$$\sigma_p = \sum_{j=0}^{m} j \cdot \sigma_{p,j} \tag{A.6}$$

Now, we rewrite Constraint (6.16) as:

$$\begin{aligned}\forall_{p\in P} : \alpha_p = \varepsilon_{p,1}.A_h + \sigma_{p,1}.A_s.\Sigma_{i>1}^{n}\varepsilon_{p,i}.A_h^i + \\ \Sigma_{i>1}^{n}\varepsilon_{p,i}.A_h^i.\Sigma_{j>1}^{m}\sigma_{p,j}.A_s^j.(1-(1-A_s)^{\frac{k}{2}}).\end{aligned} \tag{A.7}$$

Regarding the other nonlinear constraint (i.e., Equation (6.17)), the following procedure will be considered to overcome the problem. Firstly, Equation (6.17) can be written as follows:

$$ln(1-\tau_\alpha) = \sum_{p\in P} ln(1-\alpha_p). \tag{A.8}$$

It is well known that for any real number $x > -1$, [85]:

$$ln(1+x) \leq x. \tag{A.9}$$

Inequality (A.9) is also held for any function of the form $F(x) = ax + b$, which is a tangent linear function. Therefore, this inequality can now be written as:

$$ln(1+x) \leq ax + b, \tag{A.10}$$

where the functions $F(x) = ax + b$ and $ln(1 + x)$ have intersection at a certain point x_0. Note that a and b are constant, and they are calculated later. Based on Equation (A.10), we can write:

$$ln(1-\alpha_p) \leq -a_p\alpha_p + b_p \tag{A.11}$$

or

$$\sum_{p\in P} ln(1-\alpha_p) \leq \sum_{p\in P} \left(-a_p\alpha_p + b_p\right). \tag{A.12}$$

From Equation (A.8) and Inequality (A.9), we find that:

$$ln(1-\tau_\alpha) \leq \sum_{p\in P} \left(-a_p\alpha_p + b_p\right). \tag{A.13}$$

However, we need to make sure that $\tau_\alpha \geq R$, which translates into:

$$ln(1-\tau_\alpha) \leq ln(1-R). \tag{A.14}$$

Inequalities (A.13) and (A.14) do not guarantee whether $ln(1 - R) \leq \sum_{p\in P}(-a_p\alpha_p + b_p)$ or the contrary.

But if we guarantee that the inequality $\sum_{p\in P}(-a_p\alpha_p + b_p) \leq ln(1 - R)$ is met, Inequality (A.14) is then met, and consequently:

$$\sum_{p\in P} \left(-a_p\alpha_p + b_p\right) \leq ln(1-R). \tag{A.15}$$

As the function $F(x)$ is the tangent line, the constant a can be easily calculated by taking the differentiation of the function $ln(1+x)$ at a certain point x_0:

$$a = \frac{\mathrm{d}}{\mathrm{d}x}\bigg|_{x=x_0} F(x) = \frac{1}{1+x_0}. \tag{A.16}$$

The other constant b can be calculated by taking the value of the two functions at the point x_0, and thus:

$$ln(1+x_0) = ax_0 + b \tag{A.17}$$

or

$$b = -ax_0 + ln(1+x_0). \tag{A.18}$$

As our interest is to find whether $\tau_\alpha \geq R$, we need to intersect the tangent line with the logarithmic function at the point $x_0 = -R$, and thus $a_p = \frac{1}{1-R}$ and $b_p = \frac{R}{1-R} + ln(1-R)$.

Finally, in the formal model (6.3.2), we replace Constraint (6.16) by the new Constraints (A.1, A.2, A.3, A.4, A.5, A.6 and A.7), while Constraints (6.17) and (6.7) are replaced by the new Constraint (A.15).

Bibliography

[1] P. A. Kullstam. Availability, mtbf and mttr for repairable m out of n system. *Reliability, IEEE Transactions on*, R-30:393 – 394, 11 1981.

[2] Z. Abbasi, M. X. M, M. Shirazipour, and A. Takacs. An optimization case in support of next generation NFV deployment. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, 2015. USENIX Association.

[3] E. Adar and B. Huberman. Free riding on gnutella. *First Monday*, 5, 04 2001.

[4] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. Bar fault tolerance for cooperative services. *SIGOPS Oper. Syst. Rev.*, 39(5):45–58, Oct. 2005.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, Aug. 2008.

[6] M. M. Alam Khan, N. Shahriar, R. Ahmed, and R. Boutaba. Multi-path link embedding for survivability in virtual networks. *IEEE Transactions on Network and Service Management*, 13(2):253–266, June 2016.

[7] E. Amaldi, S. Coniglio, A. M. Koster, and M. Tieves. On the computational complexity of the virtual network embedding problem. *Electronic Notes in Discrete Mathematics*, 52:213 – 220, 2016. INOC 2015 – 7th International Network Optimization Conference.

[8] Amazon. Amazon ec2 instance types.

[9] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. *Computer*, 38(4):34–41, April 2005.

[10] E. Banks. Data center network design moves from tree to leaf. *Searchdatacenter. techtarget. com*, 2016.

[11] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte. Orchestrating virtualized network functions. *IEEE Transactions on Network and Service Management*, 13(4):725–739, Dec 2016.

[12] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. On orchestrating virtual network functions. In *2015 11th International Conference on Network and Service Management (CNSM)*, pages 50–56, Nov 2015.

[13] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Oper. Res.*, 46(3):316–329, Mar. 1998.

[14] E. Bauer and R. Adams. Reliability and availability of cloud computing. *Wiley-IEEE Press*, July 2012.

[15] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In vini veritas: Realistic and controlled network experimentation. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 3–14, New York, NY, USA, 2006. ACM.

[16] D. Bertsimas, D. B. Brown, and C. Caramanis. Theory and applications of robust optimization. *SIAM Rev.*, 53(3):464–501, Aug. 2011.

[17] D. Bhamare, R. Jain, M. Samaka, and A. Erbad. A survey on service function chaining. *Journal of Network and Computer Applications*, 75:138 – 155, 2016.

[18] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan. Optimal virtual network function placement in multi-cloud service function chaining architecture. *Computer Communications*, 102:1 – 16, 2017.

[19] M. Bhandarkar. Mapreduce programming with apache hadoop. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–1, April 2010.

[20] E. Bin, O. Biran, O. Boni, E. Hadad, E. K. Kolodner, Y. Moatti, and D. H. Lorenz. Guaranteeing high availability goals for virtual machine placement. In *2011 31st International Conference on Distributed Computing Systems*, pages 700–709, June 2011.

[21] Bitcoin project . Bitcoin - open source p2p money. https://bitcoin.org/en/. Accessed: 2019-02-26.

[22] J. F. Botero, X. Hesselbach, M. Duelli, D. Schlosser, A. Fischer, and H. de Meer. Energy efficient virtual network embedding. *IEEE Communications Letters*, 16(5):756–759, May 2012.

[23] G. Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, Nov. 1987.

[24] J. C. Morris. Distriwiki:: a distributed peer-to-peer wiki network. In *Int. Sym. Wikis*, pages 69–74, Jan 2007.

[25] R. N. Calheiros, R. Buyya, and C. A. F. De Rose. A heuristic for mapping virtual machines and links in emulation testbeds. In *2009 International Conference on Parallel Processing*, pages 518–525, Sep. 2009.

[26] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In *The 6th IEEE/ACM International Workshop on Grid Computing, 2005.*, pages 8 pp.–, Nov 2005.

[27] F. Carpio, W. Bziuk, and A. Jukan. Replication of virtual network functions: Optimizing link utilization and resource costs. In *2017 40th International*

Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), pages 521–526, May 2017.

[28] F. Carpio, S. Dhahri, and A. Jukan. VNF placement with replication for load balancing in NFV networks. *CoRR*, abs/1610.08266, 2016.

[29] F. Carpio and A. Jukan. Improving reliability of service function chains with combined VNF migrations and replications. *CoRR*, abs/1711.08965, 2017.

[30] H. Casanova. Simgrid: a toolkit for the simulation of application scheduling. In *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 430–437, May 2001.

[31] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, Nov. 2002.

[32] M. Chen, S. Mao, and Y. Liu. Big data: A survey. *Mobile Networks and Applications*, 19(2):171–209, Apr 2014.

[33] Y. Chen, J. Li, T. Wo, C. Hu, and W. Liu. Resilient virtual network service provision in network virtualization environments. In *2010 IEEE 16th International Conference on Parallel and Distributed Systems*, pages 51–58, Dec 2010.

[34] M. Chiosi, D. Clarke, P. Willis Cablelabs, C. Donley, L. Johnson Centurylink, M. Bugenhagen, J. Feger, W. Khan, C. China, H. Cui, C. Chen China Deng, T. , L. Baohua, S. Zhenqiang, and S. Wright. Network functions virtualisation (nfv) network operator perspectives on industry progress. *SDN and OpenFlow World Congress*, Oct 2013.

[35] N. M. K. Chowdhury and R. Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862 – 876, 2010.

[36] N. M. M. K. Chowdhury, M. R. Rahman, and R. Boutaba. Virtual network embedding with coordinated node and link mapping. In *IEEE INFOCOM 2009*, pages 783–791, April 2009.

[37] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: An overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, July 2003.

[38] Cisco. Cisco Data Center Infrastructure 2.5 Design Guide. Technical report, Cisco Systems, Inc., Dec. 2007. http://www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf.

[39] Cisco. Cisco Data Center Spine-and-Leaf Architecture: Design Overview. https://www.cisco.com/c/en/us/products/collateral/switches/nexus-7000-series-switches/white-paper-c11-737022.pdf, Apr. 2016.

[40] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[41] A. Clement, H. Li, J. Napper, J. Martin, L. Alvisi, and M. Dahlin. Bar primer. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 287–296, June 2008.

[42] R. Cohen, L. Lewin-Eytan, J. S. Naor, and D. Raz. Near optimal placement of virtual network functions. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1346–1354, April 2015.

[43] M. Correia, P. Costa, M. Pasin, A. Bessani, F. Ramos, and P. Verissimo. On the feasibility of byzantine fault-tolerant mapreduce in clouds-of-clouds. In *2012 IEEE 31st Symposium on Reliable Distributed Systems*, pages 448–453, Oct 2012.

[44] P. Costa, M. Pasin, A. N. Bessani, and M. Correia. Byzantine fault-tolerant mapreduce: Faults are not just crashes. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 32–39, Nov 2011.

[45] P. Costa, M. Pasin, A. N. Bessani, and M. Correia. Byzantine fault-tolerant mapreduce: Faults are not just crashes. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 32–39, Nov 2011.

[46] X. Costa-Perez, J. Swetina, T. Guo, R. Mahindra, and S. Rangarajan. Radio access network virtualization for future mobile carrier networks. *IEEE Communications Magazine*, 51(7):27–35, July 2013.

[47] D. Cotroneo, L. De Simone, A. K. Iannillo, A. Lanzaro, R. Natella, J. Fan, and W. Ping. Network function virtualization: Challenges and directions for reliability assurance. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 37–42, Nov 2014.

[48] G. F. Coulouris, J. Dollimore, and T. Kindberg. *Distributed systems : concepts and design.* New York ; Harlow, England : Addison-Wesley, 3rd ed edition, 2001. Previous ed.: 1994.

[49] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[50] P. Derbeko, S. Dolev, E. Gudes, and S. Sharma. Security and privacy aspects in mapreduce on clouds: A survey. *Computer Science Review*, 20:1 – 28, 2016.

[51] W. Ding, H. Yu, and S. Luo. Enhancing the reliability of services in nfv with the cost-efficient redundancy scheme. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2017.

[52] Domo. Data never sleeps 6 | domo, 2018.

[53] G. Dósa. The tight bound of first fit decreasing bin-packing algorithm is ffd. In B. Chen, M. Paterson, and G. Zhang, editors, *Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, pages 1–11, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[54] J. Elzinga and T. G. Moore. A central cutting plane algorithm for the convex programming problem. *Mathematical Programming*, 8(1):134–145, Dec 1975.

[55] A. Engelmann and A. Jukan. A reliability study of parallelized vnf chaining. In *2018 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2018.

[56] ETSI GS NFV 002. Network functions virtualization (NFV); architectural framework v1.1.1. Technical report, ETSI, October 2013.

[57] ETSI, ISGNFV. ETSI GS NFV 001 V1. 1.1: Network Functions Virtualisation(NFV); Use Cases, 2013.

[58] ETSI, ISGNFV. ETSI GS NFV-REL 001 V1. 1.1: Network Functions Virtualisation(NFV); Resiliency Requirements, 2015.

[59] J. Fan, C. Guan, K. Ren, and C. Qiao. Guaranteeing availability for network function virtualization with geographic redundancy deployment. Technical report, University of Buffalo, 2015.

[60] J. Fan, C. Guan, Y. Zhao, and C. Qiao. Availability-aware mapping of service function chains. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, May 2017.

[61] J. Fan, Z. Ye, C. Guan, X. Gao, K. Ren, and C. Qiao. Grep: Guaranteeing reliability with enhanced protection in nfv. In *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, HotMiddlebox '15, pages 13–18, New York, NY, USA, 2015. ACM.

[62] N. Feamster, L. Gao, and J. Rexford. How to lease the internet in your spare time. *SIGCOMM Comput. Commun. Rev.*, 37(1):61–64, Jan. 2007.

[63] M. D. Firoozjaei, J. P. Jeong, H. Ko, and H. Kim. Security challenges with network functions virtualization. *Future Generation Computer Systems*, 67:315 – 324, 2017.

[64] A. Fischer, J. Botero, M. Duelli, D. Schlosser, X. Hesselbach, and H. Meer. Alevin - a framework to develop, compare, and analyze virtual network embedding algorithms. *ECEASST*, 37, 01 2011.

[65] M. Fischetti, J. J. Salazar González, and P. Toth. Solving the orienteering problem through branch-and-cut. *INFORMS Journal on Computing*, 10:133–148, 05 1998.

[66] M. Gao, B. Addis, M. Bouet, and S. Secci. Optimal orchestration of virtual network functions. *Computer Networks*, 142:108–127, 2018.

[67] M. R. Garey, D. S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of operations research*, 1(2):117–129, 1976.

[68] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, pages 350–361, Aug. 2011.

[69] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[70] A. J. Gonzalez and B. E. Helvik. Guaranteeing sla availability in telecommunications networks. In *2012 15th International Telecommunications Network Strategy and Planning Symposium (NETWORKS)*, pages 1–6, Oct 2012.

[71] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag, Berlin, Heidelberg, 2006.

[72] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin, and A. D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, New York, USA, 2014. ACM.

[73] A. Gupta, M. F. Habib, P. Chowdhury, M. Tornatore, and B. Mukherjee. Joint virtual network function placement and routing of traffic in operator networks. *UC Davis, Davis, CA, USA, Tech. Rep*, 2015.

[74] T. H. Cormen, C. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, Jan 2001.

[75] A. Hadoop. Welcome to apachetm hadoop®!, 2017.

[76] A. Hagberg, P. Swart, and D. S Chult. Exploring network structure, dynamics, and function using networkx. In *In Proceedings of the 7th Python in Science Conference (SciPy*, pages 11–15, 2008.

[77] E. L. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE Journal on Selected Areas in Communications*, 9(7):1024–1039, Sep. 1991.

[78] J. M. Halpern and C. Pignataro. Service Function Chaining (SFC) Architecture. RFC 7665, Oct. 2015.

[79] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, Feb 2015.

[80] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal. Nfv: state of the art, challenges, and implementation in next generation mobile networks (vepc). *IEEE Network*, 28(6):18–26, Nov 2014.

[81] S. Herker, X. An, W. Kiess, S. Beker, and A. Kirstaedter. Data-center architecture impacts on virtualized network functions service chain embedding with high availability requirements. In *2015 IEEE Globecom Workshops (GC Wkshps)*, pages 1–7, Dec 2015.

[82] A. Hmaity, M. Savi, F. Musumeci, M. Tornatore, and A. Pattavina. Virtual network function placement for resilient service chain provisioning. In *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*, pages 245–252, Sep. 2016.

[83] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[84] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[85] B. Huo, X. Li, and Y. Shi. Complete solution to a problem on the maximal energy of unicyclic bipartite graphs. *Linear Algebra and its Applications*, 434(5):1370 – 1377, 2011.

[86] I. ILOG. Ibm, cplex solver. https://www.ibm.com/analytics/cplex-optimizer. Accessed: 2010-09-30.

[87] P. JACCARD. Distribution de la flore alpine dans le bassin des dranses et dans quelques regions voisines. *Bull Soc Vaudoise Sci Nat*, 37:241–272, 1901.

[88] N. Jain and R. Potharaju. When the network crumbles: An empirical study of cloud network failures and their impact on services. In *SoCC*. ACM, October 2013.

[89] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whally, and E. Snible. Improving performance and availability of services hosted on iaas clouds with structural constraint-aware virtual machine placement. In *2011 IEEE International Conference on Services Computing*, pages 72–79, July 2011.

[90] R. M. Karp. On the computational complexity of combinatorial problems. *Networks*, 5(1):45–68, 1975.

[91] J. Katz and C.-Y. Koo. On expected constant-round protocols for byzantine agreement. In *Proceedings of the 26th Annual International Conference on Advances in Cryptology*, CRYPTO'06, pages 445–462, Berlin, Heidelberg, 2006. Springer-Verlag.

[92] J. C. Knight, E. A. Strunk, and K. J. Sullivan. Towards a rigorous definition of information system survivability. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 1, pages 78–89 vol.1, April 2003.

[93] W. Kolberg, P. de B. Marcos, J. C. Anjos, A. K. Miyazaki, C. R. Geyer, and L. B. Arantes. Mrsg – a mapreduce simulator over simgrid. *Parallel Computing*, 39(4):233 – 244, 2013.

[94] S. G. Kulkarni, G. Liu, K. K. Ramakrishnan, M. Arumaithurai, T. Wood, and X. Fu. Reinforce: Achieving efficient failure resiliency for network function virtualization based services. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18. ACM, 2018.

[95] D. S. Kumar and M. A. Rahman. Simplified hdfs architecture with blockchain distribution of metadata. *International Journal of Applied Engineering Research*, 12(21):11374–11382, 2017.

[96] R. Kumar Gupta and P. R.K. Survey on virtual machine placement techniques in cloud computing environment. *International Journal on Cloud Computing: Services and Architecture*, 4:1–7, 08 2014.

[97] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[98] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. Bar gossip. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 191–204, Berkeley, CA, USA, 2006. USENIX Association.

[99] K. Li, H. Zheng, and J. Wu. Migration-based virtual machine placement in cloud systems. In *2013 IEEE 2nd International Conference on Cloud Networking (CloudNet)*, pages 83–90, Nov 2013.

[100] L. E. Li, V. Liaghat, H. Zhao, M. Hajiaghayi, D. Li, G. Wilfong, Y. R. Yang, and C. Guo. Pace: Policy-aware application cloud embedding. In *2013 Proceedings IEEE INFOCOM*, pages 638–646, April 2013.

[101] M. Li and M. Baker. *The grid: core technologies*. John Wiley & Sons, USA, 2005.

[102] Y. Li and M. Chen. Software-Defined Network Function Virtualization: A Survey. *IEEE Access*, 3:2542–2553, 2015.

[103] C. Lin, P. Liu, and J. Wu. Energy-efficient virtual machine provision algorithms for cloud systems. In *2011 Fourth IEEE International Conference on Utility and Cloud Computing*, pages 81–88, Dec 2011.

[104] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang. Moon: Mapreduce on opportunistic environments. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 95–106, New York, NY, USA, 2010. ACM.

[105] J. Lischka and H. Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. In *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*, VISA '09, pages 81–88, New York, NY, USA, 2009. ACM.

[106] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 226–238, New York, NY, USA, 1991. ACM.

[107] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Z. Cao, Q. Sun, and C. Pham. Service function chaining (sfc) general use cases. *Work in progress, IETF Secretariat, Internet-Draft draft-liu-sfc-use-cases-08*, 2014.

[108] D. Liveri, A. Sarri, and C. Skouloudi. Security and resilience in ehealth: Security challenges and risks. *European Union Agency For Network And Information Security*, 2015.

[109] J. Lu and J. Turner. Efficient Mapping of Virtual Networks onto a Shared Substrate. Technical report, Washington University in St. Louis, 2006.

[110] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary. Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions. In *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 98–106, May 2015.

[111] T. Lukovszki and S. Schmid. Online admission control and embedding of service chains. *CoRR*, abs/1506.04330, 2015.

[112] X. Lv, F. He, W. Cai, and Y. Cheng. A string-wise crdt algorithm for smart and large-scale collaborative editing systems. *Advanced Engineering Informatics*, 33:397 – 409, 2017.

[113] F. Machida, M. Kawato, and Y. Maeno. Redundant virtual machine placement for fault-tolerant consolidated server clusters. In *2010 IEEE Network Operations and Management Symposium - NOMS 2010*, pages 32–39, April 2010.

[114] G. Mailath. Do people play nash equilibrium? lessons from evolutionary game theory. *Journal of Economic Literature*, 36:1347–1374, 02 1998.

[115] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distrib. Comput.*, 11(4):203–213, Oct. 1998.

[116] M. Mao and M. Humphrey. A performance study on the vm startup time in the cloud. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 423–430, June 2012.

[117] A. Marotta and A. Kassler. A power efficient and robust virtual network functions placement problem. In *2016 28th International Teletraffic Congress (ITC 28)*, volume 01, pages 331–339, Sep. 2016.

[118] F. Marozzo, D. Talia, and P. Trunfio. A framework for managing mapreduce applications in dynamic distributed environments. In *2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 149–158, Feb 2011.

[119] F. Marozzo, D. Talia, and P. Trunfio. P2p-mapreduce: Parallel data processing in dynamic cloud environments. *Journal of Computer and System Sciences*, 78(5):1382 – 1402, 2012. JCSS Special Issue: Cloud Computing 2011.

[120] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Berkeley, CA, USA, 2014. USENIX Association.

[121] M. Mechtri, C. Ghribi, and D. Zeghlache. A scalable algorithm for the placement of service function chains. *IEEE Transactions on Network and Service Management*, 13(3):533–546, Sep. 2016.

[122] S. Mehraghdam, M. Keller, and H. Karl. Specifying and placing chains of virtual network functions. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 7–13, Oct 2014.

[123] M. Mihailescu, A. Rodriguez, and C. Amza. Enhancing application robustness in infrastructure-as-a-service clouds. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 146–151, June 2011.

[124] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and S. Davy. Design and evaluation of algorithms for mapping and scheduling of virtual network functions. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–9, April 2015.

[125] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys Tutorials*, 18(1):236–262, Firstquarter 2016.

[126] G. J. Mirobi and L. Arockiam. Service level agreement in cloud computing: An overview. In *2015 International Conference on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pages 753–758, Dec 2015.

[127] M. Moca, G. C. Silaghi, and G. Fedak. Distributed results checking for mapreduce in volunteer computing. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1847–1854, May 2011.

[128] H. Moens and F. D. Turck. Vnf-p: A model for efficient placement of virtualized network functions. In *10th International Conference on Network and Service Management (CNSM) and Workshop*, pages 418–423, Nov 2014.

[129] A. Mohammadkhan, S. Ghapani, G. Liu, W. Zhang, K. K. Ramakrishnan, and T. Wood. Virtual function placement and traffic steering in flexible and dynamic software defined networks. In *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, pages 1–6, April 2015.

[130] N. M. Mosharaf Kabir Chowdhury and R. Boutaba. Network virtualization: state of the art and research challenges. *IEEE Communications Magazine*, 47(7):20–26, July 2009.

[131] G. Moualla, T. Turletti, M. Bouet, and D. Saucez. On the necessity of accounting for resiliency in sfc. In *2016 28th International Teletraffic Congress (ITC 28)*, volume 02, pages 13–15, Sep. 2016.

[132] G. Moualla, T. Turletti, and D. Saucez. An availability-aware sfc placement algorithm for fat-tree data centers. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4, Oct 2018.

[133] G. Moualla, T. Turletti, and D. Saucez. Online robust placement of service chains for large data center topologies. *IEEE Access*, 7:60150–60162, 2019.

[134] A. I. Naimi and D. J. Westreich. Big Data: A Revolution That Will Transform How We Live, Work, and Think. *American Journal of Epidemiology*, 179(9):1143–1144, 04 2014.

[135] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *http://www.bitcoin.org/bitcoin.pdf*, 2008.

[136] S. Oechsner and A. Ripke. Flexible support of vnf placement functions in openstack. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–6, April 2015.

[137] B. M. Oki and B. Liskov. Viewstamped Replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.

[138] G. Optimization. Gurobi optimizer 5.0. *Gurobi: http://www.gurobi.com*, 2013.

[139] A. Oussous, F.-Z. Benjelloun, A. A. Lahcen, and S. Belfkih. Big data technologies: A survey. *Journal of King Saud University - Computer and Information Sciences*, 30(4):431 – 448, 2018.

[140] D. A. Patterson. Technical perspective: The data center is the computer. *Commun. ACM*, 51(1):105–105, Jan. 2008.

[141] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, Apr. 1980.

[142] K. Pentikousis, Y. Wang, and W. Hu. Mobileflow: Toward software-defined mobile networks. *IEEE Communications Magazine*, 51(7):44–53, July 2013.

[143] C. Pham, N. H. Tran, S. Ren, W. Saad, and C. S. Hong. Traffic-aware and energy-efficient vnf placement for service chaining: Joint sampling and matching approach. *IEEE Transactions on Services Computing*, pages 1–1, 2018.

[144] J. T. Piao and J. Yan. A network-aware virtual machine placement and migration approach in cloud computing. In *2010 Ninth International Conference on Grid and Cloud Computing*, pages 87–92, Nov 2010.

[145] M. Pinedo and K. Hadavi. Scheduling: Theory, algorithms and systems development. In W. Gaul, A. Bachem, W. Habenicht, W. Runge, and W. W. Stahl, editors, *Operations Research Proceedings 1991*, pages 35–42, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[146] R. Potharaju and N. Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, pages 9–22, New York, NY, USA, 2013. ACM.

[147] L. Qu, C. Assi, K. Shaban, and M. J. Khabbaz. A reliability-aware network service chain provisioning with delay guarantees in nfv-enabled enterprise datacenter networks. *IEEE Transactions on Network and Service Management*, 14(3):554–568, Sep. 2017.

[148] P. Quinn, U. Elzur, and C. Pignataro. Network service header (nsh). Technical report, Internet Engineering Task Force (IETF), Jan 2018.

[149] M. G. Rabbani, M. F. Zhani, and R. Boutaba. On achieving high survivability in virtualized data centers. *IEICE Transactions*, 97-B:10–18, 2014.

[150] M. R. Rahman and R. Boutaba. Svne: Survivable virtual network embedding algorithms for network virtualization. *IEEE Transactions on Network and Service Management*, 10(2):105–118, 2013.

[151] L. Ren, K. Nayak, I. Abraham, and S. Devadas. Efficient synchronous byzantine consensus. *CoRR*, abs/1704.02397, 2017.

[152] T. Rosado and J. Bernardino. An overview of openstack architecture. In *Proceedings of the 18th International Database Engineering & Applications Symposium*, IDEAS '14, pages 366–367, New York, NY, USA, 2014. ACM.

[153] K. W. Ross and D. H. K. Tsang. The stochastic knapsack problem. *IEEE Transactions on Communications*, 37(7):740–747, July 1989.

[154] S. Sahhaf, W. Tavernier, D. Colle, and M. Pickavet. Network service chaining with efficient network function mapping based on service decompositions. In *Proceedings of the 2015 1st IEEE Conference on Network Softwarization (NetSoft)*, pages 1–5, April 2015.

[155] J. Sahoo, M. A. Salahuddin, R. Glitho, H. Elbiaze, and W. Ajib. A survey on replica server placement algorithms for content delivery networks. *IEEE Communications Surveys Tutorials*, 19(2):1002–1026, Secondquarter 2017.

[156] A. M. Sampaio and J. G. Barbosa. Towards high-available and energy-efficient virtual computing environments in the cloud. *Future Gener. Comput. Syst.*, 40:30–43, Nov. 2014.

[157] F. B. Schneider. Byzantine generals in action: Implementing fail-stop processors. *ACM Trans. Comput. Syst.*, 2(2):145–154, May 1984.

[158] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, Dec. 1990.

[159] M. Schöller, M. Stiemerling, A. Ripke, and R. Bless. Resilient deployment of virtual network functions. In *2013 5th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, pages 208–214, Sept 2013.

[160] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings First International Conference on Peer-to-Peer Computing*, pages 101–102, Aug 2001.

[161] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[162] J. Shamsi, M. A. Khojaye, and M. A. Qasmi. Data-intensive cloud computing: Requirements, expectations, challenges, and solutions. *Journal of Grid Computing*, 11(2):281–310, Jun 2013.

[163] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 13–24, New York, NY, USA, 2012. ACM.

[164] J. Shneidman and D. C. Parkes. Specification faithfulness in networks with rational nodes. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 88–97, New York, NY, USA, 2004. ACM.

[165] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010.

[166] R. Sridharan. A lagrangian heuristic for the capacitated plant location problem with single source constraints. *European Journal of Operational Research*, 66(3):305 – 312, 1993.

[167] J. P. Sterbenz, D. Hutchison, E. K. Çetinkaya, A. Jabbar, J. P. Rohrer, M. Schöller, and P. Smith. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. *Computer Networks*, 54(8):1245 – 1265, 2010. Resilient and Survivable networks.

[168] K. Su, L. Xu, C. Chen, W. Chen, and Z. Wang. Affinity and conflict-aware placement of virtual machines in heterogeneous data centers. In *2015 IEEE Twelfth International Symposium on Autonomous Decentralized Systems*, pages 289–294, March 2015.

[169] Q. Sun, P. Lu, W. Lu, and Z. Zhu. Forecast-assisted nfv service chain deployment based on affiliation-aware vnf placement. In *2016 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, Dec 2016.

[170] A. Tomassilli, B. Jaumard, and F. Giroire. Path protection in optical flexible networks with distance-adaptive modulation formats. In *2018 International Conference on Optical Network Design and Modeling (ONDM)*, pages 30–35, May 2018.

[171] A. Toptal and I. Sabuncuoglu. Distributed scheduling: a review of concepts and applications. *International Journal of Production Research*, 48(18):5235–5262, 2010.

[172] R. Vaessens. *Generalized job shop scheduling : complexity and local search.* PhD thesis, Department of Mathematics and Computer Science, 1995.

[173] E. Vargas and S. BluePrints. High availability fundamentals. *Sun Blueprints series*, pages 1–17, 2000.

[174] D. Veitch. Rethinking virtual network embedding: Substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38:17–17, 04 2008.

[175] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144, Sep. 2009.

[176] J. von Neumann. Various techniques used in connection with random digits. In A. Householder, G. Forsythe, and H. Germond, editors, *Monte Carlo Method*, pages 36–38. National Bureau of Standards Applied Mathematics Series, 12, Washington, D.C.: U.S. Government Printing Office, 1951.

[177] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *2010 Proceedings IEEE INFOCOM*, pages 1–9, March 2010.

[178] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen. G-hadoop: Mapreduce across distributed data centers for data-intensive computing. *Future Generation Computer Systems*, 29(3):739 – 750, 2013. Special Section: Recent Developments in High Performance Computing and Security.

[179] T. Wang, Z. Su, Y. Xia, and M. Hamdi. Rethinking the data center networking: Architecture, network protocols, and resource sharing. *IEEE Access*, 2:1481–1496, 2014.

[180] X. Wang, C. Wu, F. Le, A. Liu, Z. Li, and F. Lau. Online vnf scaling in datacenters. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 140–147, June 2016.

[181] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 3rd edition, 2012.

[182] Wikipedia contributors. Ethereum — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Ethereum&oldid=879895047, 2019. [Online; accessed 1-February-2019].

[183] C. Wu and R. Buyya. *Cloud Data Centers and Cost Modeling: A Complete Guide To Planning, Designing and Building a Cloud Data Center*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2015.

[184] G. Wu, M. Tang, Y.-C. Tian, and W. Li. Energy-efficient virtual machine placement in data centers by genetic algorithm. In *International Conference on Neural Information Processing (ICONIP)*, pages 315–323, Renaissance Doha City Center Hotel, Doha, 2012. Springer Berlin Heidelberg. Neural Information Processing : 19th International Conference, ICONIP 2012, Doha, Qatar, November 12-15, 2012, Proceedings, Part III.

[185] B. Xia and Z. Tan. Tighter bounds of the first fit algorithm for the bin-packing problem. *Discrete Applied Mathematics*, 158(15):1668 – 1675, 2010.

[186] J. Xin, L. Guo, N. Huang, and R. Li. Network service reliability analysis model. In *Chemical Engineering Transactions*, volume 33, pages 511–516, Jan 2013.

[187] M. Yannakakis. Expressing combinatorial optimization problems by linear programs. *Journal of Computer and System Sciences*, 43(3):441 – 466, 1991.

[188] F. Yue. Network functions virtualization-everything old is new again. *F5 Neworks*, 2013.

[189] Q. Zhang, Y. Xiao, F. Liu, J. C. S. Lui, J. Guo, and T. Wang. Joint optimization of chain placement and request scheduling for network function virtualization. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 731–741, June 2017.

[190] Q. Zhang, M. F. Zhani, M. Jabri, and R. Boutaba. Venice: Reliable virtual data center embedding in clouds. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, pages 289–297, April 2014.

[191] M. Zhao. Availability for repairable components and series systems. *IEEE Transactions on Reliability*, 43(2):329–334, June 1994.

[192] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE International Congress on Big Data (BigData Congress)*, pages 557–564, June 2017.

[193] A. Zhou, S. Wang, B. Cheng, Z. Zheng, F. Yang, R. N. Chang, M. R. Lyu, and R. Buyya. Cloud service reliability enhancement via virtual machine placement optimization. *IEEE Transactions on Services Computing*, 10(6):902–913, Nov 2017.

[194] B. Zhu, S. Jajodia, and M. S. Kankanhalli. Building trust in peer-to-peer systems: a review. *Int. J. Secur. Netw.*, 1(1/2):103–112, Sept. 2006.

[195] M. Zhu, J. Cao, Z. Cai, Z. He, and M. Xu. Providing flexible services for heterogeneous vehicles: an nfv-based approach. *IEEE Network*, 30(3):64–71, May 2016.

[196] Y. Zhu and M. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*, pages 1–12, April 2006.