Towards a coordination-free orchestration approach to manage consistent service reconfiguration in NFV multi-domain environments

Josué Castañeda Cisneros

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par :
l'Institut National des Sciences Appliquées de Toulouse (INSA de Toulouse)

Présentée et soutenue le *09/12/2021* **par :**

Josue CASTAÑEDA CISNEROS

Towards a coordination-free orchestration approach to manage consistent service reconfiguration in NFV multi-domain environments

---

JURY

Roch H. GLITHO	Professeur	Président du Jury
Noura LIMAM	Professeure Assistante	Membre du Jury
Carla-Fabiana CHIASSERINI	Professeure	Membre du Jury
Saúl Eduardo POMARES HERNÁNDEZ	Professeur	Membre du Jury

École doctorale et spécialité :
EDMITT : Informatique et Télécommunications
Unité de Recherche :
Laboratoire d'analyse et d'architecture des systèmes
Directeur(s) de Thèse :
Khalil DRIRA et *Sami YANGUI*
Rapporteurs :
Djamal ZEGHLACHE et *Mohamed MOSBAH*

Acknowledgments

Thank you to everyone who made this thesis and research possible. From the people who gave me their support at the beginning, those who gave me good directions, and the ones who stayed. I say thanks to my team who offer good ideas and discussions, in both formal and informal events. To my new friends and aquatiances

Abstract: Under the multi-domain orchestration approach of Network Function Virtualization, network providers share services, creating a federation. In it, many orchestrators jointly manage the lifecycle tasks of network services, such as reconfiguration. To consistently reconfigure VNF-based network services; all replicas of the service in each orchestrator have the same state. The literature proposes sending grants to prevent unwanted side-effects when reconfiguring such services. If a conflict arises because of inconsistencies, the orchestrators either choose an all-knowing global orchestrator or solve consensus. Both approaches introduce overhead that clashes with the goal of Network Function Virtualization. In this research, we propose identifying dependencies by an intermediary consistency model. We ask if it is possible to reconfigure consistently VNF-based network services without coordinating the orchestrators. To answer this, first, we study the problem of migrating shared virtualized network functions, highlighting why orchestrators need to coordinate. Then, we consider the more general problem of end-to-end consistent reconfiguration for dependent VNF-based network services. We identify that is possible, at least theoretically, to solve the previous problem without coordinating the orchestrators; but, in practical terms, they coordinate. Finally, we study the related problem of reconfiguring the VNF-Forwarding Graph of a shared VNF-based network service. We show the limits of the coordination approach and propose a coordination-free algorithm to reconfigure consistently the VNF-Forwarding Graphs. After this research, we open the door for other orchestration algorithms without coordination for managing the lifecycle tasks of VNF-based network services. This means strong consistency guarantees without the overhead of coordination.

Keywords: Coordination-free Orchestration, Consistent VNF-based Service Reconfiguration, Multi-domain Orchestration, Network Function Virtualization

Résumé :

Dans le cadre de l'approche d'orchestration multi-domaine de la virtualisation des fonctions réseau, les fournisseurs de réseaux partagent des services, créant ainsi une fédération. Dans celle-ci, de nombreux orchestrateurs gèrent conjointement les tâches du cycle de vie des services réseau, telles que la reconfiguration. Pour reconfigurer de manière cohérente les services réseau basés sur VNF, toutes les répliques du service dans chaque orchestrateur ont le même état. La littérature propose l'envoi de subventions pour éviter les effets secondaires indésirables lors de la reconfiguration de ces services. Si un conflit survient en raison d'incohérences, les orchestrateurs choisissent un orchestrateur global omniscient ou résolvent le consensus. Ces deux approches introduisent des frais généraux qui vont à l'encontre de l'objectif de la virtualisation des fonctions réseau. Dans cette recherche, nous proposons d'identifier les dépendances par un modèle de cohérence intermédiaire. Nous nous demandons s'il est possible de reconfigurer de manière cohérente les services réseau basés sur les VNF sans coordonner les orchestrateurs. Pour répondre à cette question, nous étudions d'abord le problème de la migration des fonctions réseau virtualisées partagées, en mettant en évidence les raisons pour lesquelles les orchestrateurs doivent se coordonner. Ensuite, nous considérons le problème plus général de la reconfiguration cohérente de bout en bout pour les services réseau dépendants basés sur VNF. Nous identifions qu'il est possible, au moins théoriquement, de résoudre le problème précédent sans coordonner les orchestrateurs ; mais, en termes pratiques, ils se coordonnent. Enfin, nous étudions le problème connexe de la reconfiguration du VNF-Forwarding Graph d'un service réseau partagé basé sur VNF. Nous montrons les limites de l'approche de coordination et proposons un algorithme sans coordination pour reconfigurer de manière cohérente les VNF-Forwarding Graphs. Après cette recherche, nous ouvrons la porte à d'autres algorithmes d'orchestration sans coordination pour gérer les tâches du cycle de vie des services réseau basés sur VNF. Cela signifie de fortes garanties de cohérence sans la surcharge de la coordination.

Mots clés : Orchestration sans coordination, Reconfiguration Cohérente des Services Basés sur les VNF, Orchestration Multi-domaine, Virtualisation des Fonctions de Réseau.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Future networks, like 6G, bring new services and surpass the limits of current technologies [Barakabitze 2020]. Examples of such limits are the speed of connection, availability of services, privacy, scalability, among others [Vaquero 2019]. The projected number of users consuming network services aggravates the problem. For the short-term future, projections show users will consume mobile data up to 49 exabytes per month [Forecast 2019]. Such growth forces service providers to deploy specialized hardware to offer new network services and stay competitive in a changing market [Checko 2015]. When scaling up such middle-boxes, two factors are increased: the cost of network deployment, also called Capital Expenditure (CAPEX), and the cost to operate the network, called Operating Expenditure (OPEX). Even worse, launching a new service under these conditions is costly, error-prone, and time-consuming.

Commercial off-the-shelf network hardware has been proposed to replace middle-boxes to reduce CAPEX and OPEX [Yi 2018]. The European Telecommunications Standards Institute (ETSI) introduced the decoupling of dedicated hardware and functionality. ETSI called this the Network Function Virtualization (NFV) [ETSI 2014] paradigm. Under the ETSI NFV standard, the service providers distribute networks functions at different points of presence [Yong Li 2015]. Each location has physical resources (i.e. compute, memory, network) and can exploit these resources to instantiate Virtual Network Functions (VNF). The providers chain together the VNFs, creating dedicated end-to-end VNF-based network services. Such services belong to a single administrative domain. To ensure the correct execution order of the service's VNFs, a VNF-Forwarding Graph (VNF-FG) specifies a topology associating rules and connection points. The lifecycle of each of these components is managed by an orchestrator that has complete information about the domain [Gil Herrera 2016].

Traditionally, a single orchestrator manages all the resources of a domain by sending instructions to other VNFs and their managers. These virtual resources communicate with the orchestrator, ensuring state consistency [ETSI 2014]. Having a single entity facilitates managing network services; however, such architecture raises some problems. First, a central orchestrator creates a single point of failure [Frick 2018]. Second, since the other resources communicate only with the orchestrator, scaling services becomes challenging because of the bottleneck created [Saraiva de Sousa 2019]. Third, the orchestrator's location affects the service's performance, since the virtual resources need to send all messages while being close to the service's users [Antevski 2020]. Fourth, the service providers have limited resources that limit the reach of services. Finally,

keeping an up-to-date policy and topology information remains challenging, especially when many administrative domains create a federation as they want autonomy and privacy [Joshi 2020]. These problems limit the revenue for service providers; not only do they increase the CAPEX and OPEX, but also limit their market share. To address these problems, ETSI proposed a new paradigm where many service providers share resources, namely multi-domain orchestration.

The ETSI NFV standard proposed multi-domain orchestration to prevent the previous problems [Saraiva de Sousa 2019]. Multi-domain orchestration enables federating network services, such that service providers share network resources [ETSI, NFVISG 2018]. Thus, the providers' orchestrators deploy services in multiple domains. Unlike dedicated services, shared services belong to many administrative domains' external dependencies, ensuring the services locate close to users [Taleb 2019, Saraiva de Sousa 2019]. Federating services also offer resiliency properties, such as improved scale, increased performance, and greater robustness against failures [Katsalis 2016]. Furthermore, unlike in single domain, in multi-domain orchestration many orchestrators manage the services' lifecycle tasks. Among such lifecycle tasks are discovering, placing, chaining, monitoring, and reconfiguring network services [Saraiva de Sousa 2019]. Reconfiguration of VNF-based network services is the most complex task as it often involves all the previous ones [Vaquero 2019].

Reconfiguration is an integral part of VNF-based network services' lifecycle. It comprises updating services accommodating for contextual changes. Yet, most of the literature considers reconfiguring services using the single-domain approach [Yi 2020, Eramo 2017a]. However, assuming that an orchestrator has global information for all administrative domains is unrealistic for distributed federations. Limited knowledge of multi-domain federations makes reconfiguring VNF-based network services challenging. For example, after reconfiguring a shared VNF-based network service, some service replicas can have different states because of non-deterministic network delays, creating inconsistencies. This translates to costs for service providers as the inconsistencies violate both functional and non-functional properties [Dang 2020]. Thus, service providers want to achieve a consistent reconfiguration that meets the user's new demands while satisfying both types of properties.

Orchestrators achieve consistent reconfiguration when the state of multiple services is the same before and after reconfiguring such services [Saraiva de Sousa 2019, Vaquero 2019]. **For multi-domain orchestration, orchestrators achieve consistency when all replicas of VNF-based network service and its component have the same value**. When federating shared services, all replicas apply the same change, ensuring the service's proper behavior (i.e. the service satisfies functional and non-functional properties). Otherwise, future changes diverge the replicas' state, creating failures. Nowadays, the literature mostly considers eventual consistency when reconfiguring shared VNF-based network services.

Currently, state of the art algorithms consider applying updates as they come with-

out timing constraints [ETSI, NFVISG 2018]. Eventual consistency ensures that, at the end of the reconfiguration, all replicas for dedicated VNF-based network services have the same state. However, for shared services, this is not the case, as they have service dependencies among them. Thus, for shared services, federations require stronger consistency guarantees.

The consensus among orchestrators is one way to provide the strongest consistency guarantees. Ensuring that all the orchestrators agree on a single value prevents inconsistencies, even for shared VNF-based network services [Okusanya 2019]. However, this approach hinders the performance of the whole federation in terms of latency and poor scalability [Hao 2018]. In this research, first, we explore an intermediary consistency model to achieve a suitable trade-off between consistency guarantees and performance while reconfiguring shared services. Then, we propose the first coordination-free algorithm that achieves consistent reconfiguration without compromising performance. Next, we define more precisely the problem addressed in this research.

Problem statement

The problem of consistent VNF-based service reconfiguration asks the following question: **How to change the configuration of a given VNF-based network service such that all replicas of such service have the same configuration?**

The configuration of a VNF-based network service depends on the context considered. For this, we decompose the problem into multiple sub-problems. We describe each sub-problem. In this research we focus on the migrating, scaling and updating tasks for VNF-based network services. We briefly state the meaning of consistent reconfiguration for that sub-problem.

- Migrating. How to change the location of one component of the service (either a VNF or VNF-based network service). For stateful VNFs, consistency reconfiguration means no state is lost or changed [ETSI, NFVISG 2014].

- Scaling. Ability to dynamically extend/reduce resources granted to the VNF as needed. This includes scaling up/down and scaling out/in [ETSI, NFVISG 2018]. Consistent reconfiguration means that all the replicas have the same resources.

- Updating. This operation allows changing the configuration and parameterization of an instantiated virtualized resource. Such configuration could be the connection points, name, metadata, among others [ETSI, NFVISG 2014]. Consistent reconfiguration means that all parameters for each replica are equal after an update.

- Healing. Ability to recover a service component, like a VNF, from an error. The orchestrator monitors the components and in case of a failure, it will attempt to restart the component(s).

- Terminating. Stop and deallocate the Virtual Infrastructure resources. However, the resources for the component instance remain reserved.
- Deleting. The resources for the service component(s) are fully released.

Aim of the work

The research work's aim is the design and development of a mechanism to reconfigure consistently VNF-based network services in distributed multi-domain environments. Until now, service providers reconfigure a VNF-based network service mostly by assuming a single orchestrator having complete administrative domain knowledge. This research considers the problem from a distributed approach in which orchestrators have only limited knowledge. This, ensures providers' autonomy and privacy. Moreover, it also consider the asynchronous and non-deterministic behaviors of distributed systems.

Specifically, to lead the research, we propose that consistently reconfiguring VNF-based network services in multiple administrative domains, while facing asynchronous network and partial knowledge, can be achieved by coordinating the orchestrators. Thus, exchanging causal messages about the services' dependencies allows us to reconfigure consistently services. Furthermore, we propose we can achieve the same consistent network service reconfiguration in a coordination-free approach.

Thesis contributions

The main contribution of this research is **the establishment of causal and coordination-free principles to ensure consistent VNF-based network service reconfiguration in distributed multi-domain federations**. Next, we describe them:

- The consistent migration of shared VNFs in federated environments despite each orchestrator having only local information about its domain. We propose a coordination algorithm for shared VNFs in federated environments. It achieves a lower overhead in terms of service violations than the traditional greedy approach that minimizes only the current VNF without considering the impact of other VNFs in the chain. (Chapter 3, Section 3.4).
- An inconsistent pattern for the NFV dependent reconfiguration is identified and formally defined from a temporal and logical perspective (Chapter 4, Section 4.4.2, 4.4.3). To prevent such inconsistencies, a causally-consistent orchestration algorithm is proposed.
- Identifying and reducing inconsistencies created while the orchestrators reconfigure shared VNF-FGs by coordinating among themselves through messages without global references. We advance the state of the art for managing the VNF-FG by

not only considering local domain information but also addressing the temporal dependencies among shared services under multi-domain federations, as described in the VNF-FG (Chapter 5, Section 5.4).

- The design of coordination-free orchestration algorithm for consistent VNF-FG reconfiguration under multi-domain federations. Our proposed algorithm supports non-functional dependencies that have not been addressed so far in the literature for VNF-FG reconfiguration. Unlike current orchestration algorithms, our proposed algorithm consistently reconfigures the VNF-FG of a shared network service without a coordination phase between the orchestrators. By skipping the coordination phase we open the door for dynamic federations where orchestrators join and leave temporarily. (Chapter 6, Section 6.3)

Thesis organization

The thesis is organized in different chapters corresponding to the different sub-problems we tackled during the course of the thesis. Figure 1.1 shows the thesis organization including papers accepted and submitted. Each chapter goes more in depth towards the thesis main contribution. They are as follows:

- Chapter 2 describes the concepts and system model used through the manuscript.
- Chapter 3 describes the problem of consistent migration for VNF sharing. This is the first step towards the consistent reconfiguration of VNF-based services.
- Chapter 4 goes more in depth about the dependent reconfiguration of network services. We identify the conditions required for inconsistent dependent reconfigurations.
- Following that, in Chapter 5, we focus on the problem of VNF-FG reconfiguration. Unlike the previous two chapters, were we consider sequential reconfigurations only, for this chapter we consider concurrent reconfigurations and we highlight the limits of coordination approaches to achieve consistent reconfiguration.
- To met the limitations posed by concurrent reconfigurations for shared VNF-Forwarding Graphs, Chapter 6 introduces the first coordination-free orchestration algorithm for consistent VNF-FG reconfiguration.
- Finally, Chapter 7 presents the perspectives, the state before and after this thesis, and future work after the thesis.

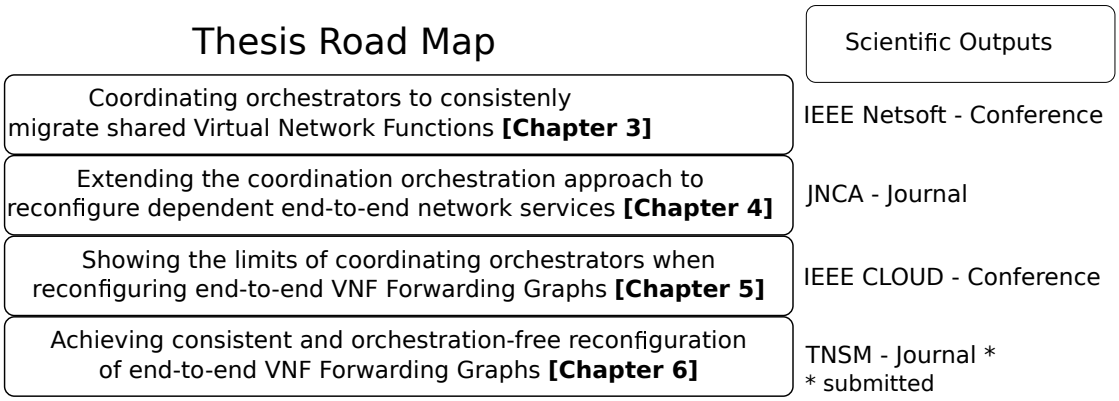

Figure 1.1: Manuscript organization including chapters and the associated publications.

Chapter 2

Background and definitions

2.1 Network function virtualization: Basic concepts

Next-generation networks like 5G will bring new services and surpass the limits of current technologies [Barakabitze 2020]. Among such limits, the service's speed, availability, and latency are some examples [Mijumbi 2016]. The number of users consuming the network services aggravates these limits. For example, users will consume over 49 exabytes per month by 2021 [Forecast 2019]. Such growth forces service providers to deploy specialized hardware. Thus, they offer new network services staying competitive in a growing market [Checko 2015]. However, by doing so, the providers lose flexibility as they deploy services manually with only a few providers. This leads to tedious, inflexible, and often error-prone services [Gil Herrera 2016].

Such deployment strategies leads to inflexible, timely, and costly updates [Bhamare 2016]. For example, the literature has estimated that deploying a service takes about 90 days using dedicated hardware [Martini 2016]. Thus, with such strategies, the capital and operating expenditure for such network services continue to increase. To remedy this, some providers explored virtualization of different resources, such as storage, processors, and the network infrastructure, allowing for flexible and quick deployment; yet, interoperability remained a challenge. The European Telecommunications Standards Institute (ETSI) proposed the Network Function Virtualization (NFV) standard to ensure interoperability among different providers [Open Source 2018].

The ETSI NFV standard virtualizes network services that traditionally run on proprietary and dedicated hardware [ETSI 2018b]. It decouples network services from the underlying hardware by using Virtual Network Functions (VNFs). The VNFs replace dedicated network hardware such as firewalls or routers enabling agile and flexible deployment of network services. ETSI introduced one architecture to support interoperability among multiple service providers [ETSI 2014]. Following the service-oriented computing paradigm, a service provider manages a set of VNFs implementing composite services. Such services represent more complex and sophisticated network functions. Using the ETSI terms, an orchestrator builds a chain of VNFs to create a network service provisioning functionality for multiple users.

2.1.1 Goals and objectives of network function virtualization

The NFV standard tries to reduce the time to market network services, decreasing the cost of hardware and forming scalable and elastic ecosystems. To achieve this, the NFV sets multiple goals. The first goal is to decrease time spent on service evaluation and testing, as the decoupling function from hardware improves managing heterogeneous services. The second goal is to replace expensive hardware with cheap alternatives that provide at least as good service provision experience [Shin 2015]. The third goal is to make old equipment inter-operable with Virtual Network Functions [Nguyen 2017].

With the advent of virtualized networks, the previous goals also extended beyond the traditional network setup. In the short future, complex network services can be shared among many providers, creating a federation [Toka 2021]. Such federations extend the limited range or a sole provider while re-using components to lower the cost of VNFs [Malandrino 2019]. This sharing of VNFs and network services creates new challenges that need to be satisfied by NFV. For example, migrating a shared VNF in a federation not only has to consider the VNF but also the different services it serves [Cisneros 2020a].

2.1.1.1 Different standards

The term NFV was originally coined by multiple network operators such as AT&T, British Telecom, and Deutsche Telekom [Yi 2018]. Different standard organizations have worked to establish a common vision in different areas surrounding NFV, such as Software Defined Network (SDN) [Alam 2020], Cloud Computing [Rankothge 2015], and 5G [Abdelwahab 2016]. Such efforts ensure interoperability coming from open interfaces [Saraiva de Sousa 2019]. ETSI defined the main terminology and orchestration architecture framework to support VNFs on top of virtualized infrastructure. The Internet Engineering Task Force (IETF) focused on the service chaining concept which relates SDN with NFV [Halpern 2015]. The Next Generation Mobile Networks (NGMN) introduced principles and high-level architecture principles to support NFV services for 5G [3GPP 2017]. **In this work, we consider the terminology defined by ETSI**.

2.1.2 The relation between Network Function Virtualization with Cloud Computing and Software Defined Networking

NFV complements the goals of Cloud Computing and SDN paradigms. Cloud computing provides resource virtualization, such as networks, storage, and servers, with high flexibility [Le 2016]. SDN decouples the control plane (i.e., control logic) from the data plane (i.e., data forwarding equipment) reducing the vertical integration of networks. NFV decouples the network functions from dedicated hardware. These three paradigms combine in different scenarios. From a management perspective, a service provider would like to be able to deploy flexible services, configure automatically the traffic flow of such

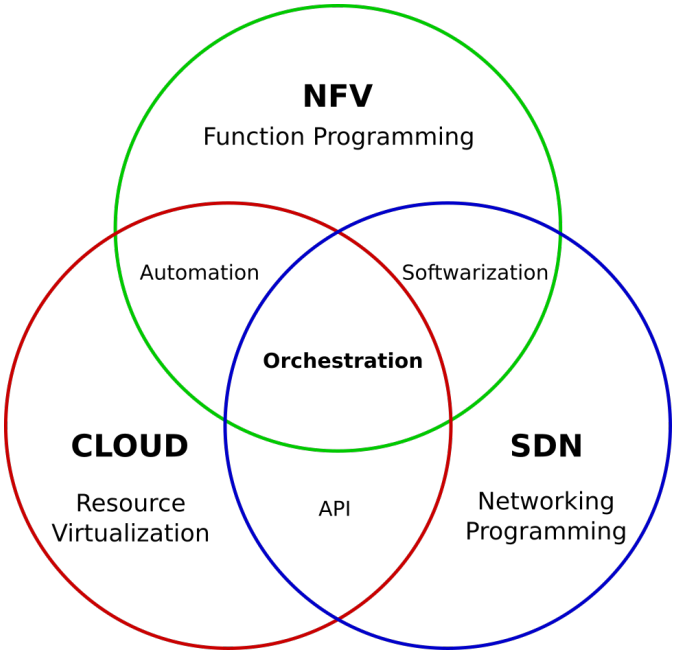

Figure 2.1: Relationship between Network Function Virtualization, Software Define Networking, and Cloud Computing.

services, and monitor and reconfigure them using open APIs. Figure 2.1 shows the different ways the paradigms integrate with each other.

2.1.3 The relation between Network Function Virtualization and Service Oriented Architecture

NFV is at the crossroad of networking and service-oriented computing research fields. It is advocated that VNF falls into the definition of IT services at large [Katsalis 2016]. Indeed, VNFs could be provisioned in the same way as any other kind of service such as telco services and Web services. In fact, Service-Oriented Architecture (SOA) principles (e.g., service abstraction, discoverability, and composability) could ensure the viability of an ecosystem of network services with respect to the NFV paradigm. A SOA is usually characterized by: (i) the service provider and publishes in a registry; (ii) a user consumes the service; (iii) a broker mediates the interactions between providers offering discovery, matching, and composition [Martini 2016]. Thus, this architectural solution maps to different contexts, such as NFV [Stal 2006]. Similarly, the VNF lifecycle phases are directly inspired from the service provisioning lifecycle detailed in SOA [Yi 2018]. For example, the service provider models the network components. Such modeling involves resource requirements, connections, and scripts to execute. Then, the providers publish the components in a market place so users discover such network component.

After, a provider that selected a network component, allocates the resources to place the network component in a point of presence. Next, the provider selects how multiple components should connect to create a complex service. Finally, by monitoring the service, the provider can reconfigure the network component. Thus, the lifecycle between web services and virtual network functions are similar. However, there are differences between the traditional SOA services and VNF-based network services.

Unlike web services, VNFs are not remotely invoked from a service access points; but, they must be instantiated locally in a point of presence. The providers has a control entity to manage the technical details, such as configuration, settings, management, and connections [el houda Nouar 2021]. Heterogeneity also poses a problem in the NFV context, not seen in SOA services. These services need access only to input and output parameters, while VNF-based network services must ensure the interoperability of each network component [Bouras 2017]. In summary, for the NFV context, service providers need to handle more configurations that appear from the networking domain of NFV. This means that solutions coming from the SOA paradigm lack the details necessary for the NFV context.

2.2 Networking Function Virtualization: Advance concepts

After briefly describing the NFV initiative, we now describe the main concepts that create the core of the NFV initiative. All these concepts come from ETSI [ETSI 2020b].

2.2.1 Virtual Network Function

A Virtual Network Function (VNF) is the implementation of a functional block having a well-defined external interface and well-defined functional behavior deployed on a virtual infrastructure [ETSI 2020b]. To define the interfaces, a service provider can use a formal language like YALM [Mechtri 2016]. Service providers use this language to identify a VNF that belongs to a service provider's orchestrator using a VNF descriptor. Such descriptor contains the required resources for each component, such as memory, virtual processors, and storage. It also contains the networking information to join components through Connection Points and Virtual Links [ETSI, NFVISG 2020]. The descriptor also specifies all the required day-1 (i.e., doing the initial deployment) configuration, such as scripts, key management, and security rules. A VNF can have one or multiple components from the same point of presence where a service provider deploys a virtual infrastructure manager.

Depending on the VNF type, the orchestrator can deploy them on physical machines, virtual machines, containers, and microservices [Gedia 2018, Kawashima 2016]. For the VNF deployment, the VNF descriptor has a field allowing VNFs to interact with others through an infrastructure manager. Virtual machines require an image pre-loaded in

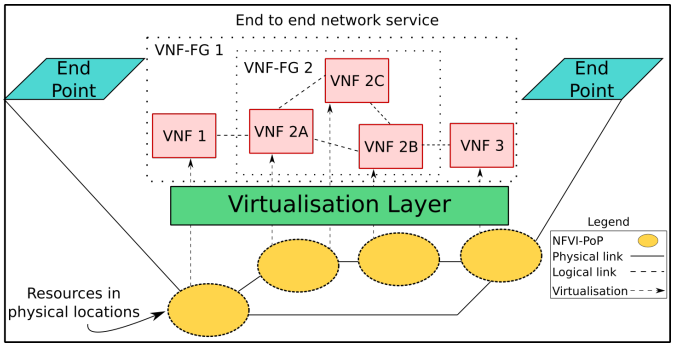

Figure 2.2: VNF-Forwarding Graph and supporting infrastructure. In this example, a network service is served by two different nested VNF-FGs.

the point of presence; for example, using OpenStack with cloud images [Callegati 2015]. The literature has explored containers to reduce the virtual machine's overhead trying to improve the VNFs' performance. The trade-off is limited to the functionality provided, especially when the containerized VNFs work together with virtual machines ones [Kouchaksaraei 2019]. Despite these multiple types, the NFV's state of the art uses virtual machines providing VNFs [Gil Herrera 2016].

2.2.2 VNF-Forwarding graph

The VNF Forwarding Graph (VNF-FG) specifies a topology of connected VNFs' composing services. With such topology, the service providers connect the VNFs by using virtual links through interfaces called connection points, and the associated rules (e.g. network protocol, source/destination IP addresses, and ports) [Quang 2019b]. The VNF-FG also has forwarding rules applicable to the traffic conveyed over the topology [Cao 2017]. It also saves a list of nested services associated with internal and external service access points [ETSI 2017]. Figure 2.2 shows an example of a network service with two VNF-FGs built on top of a virtual infrastructure.

All these VNF-FG resources are managed by an orchestrator [ETSI, NFVISG 2016]. After placing the VNF-FGs, the orchestrator can reconfigure them to optimize profit or answer to unforeseen and developing conditions. This reconfiguration aims to satisfy the cost-effective goal of NFV.

2.2.3 VNF-based network services

Network services are the building units for next-generation network applications. Under the NFV concept, multiple VNFs compose a network service according to one or more forwarding graphs [ETSI 2019b]. Network services belong to distinct classes according to their users via the service's access points. *Dedicated* network services belong to a single domain and only have VNFs as internal dependencies. *Composite* ones belong to at least

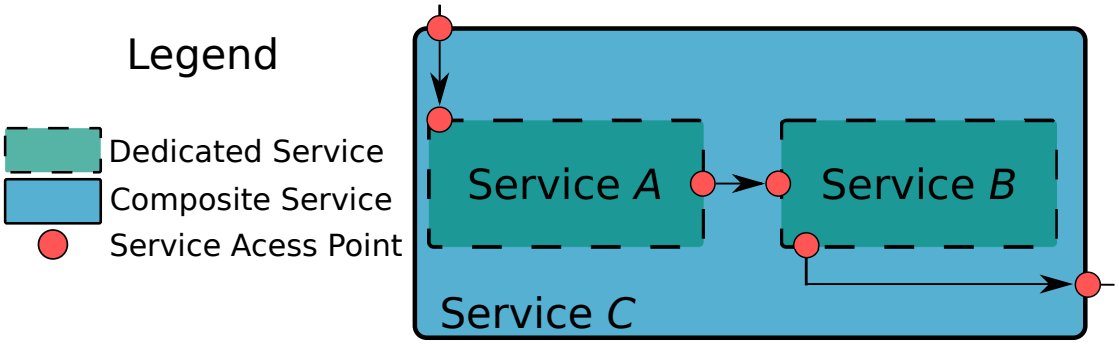

Figure 2.3: Example of a composite and two dedicated network services.

two different domains and have external dependencies as network services [ETSI 2020b]. The external dependencies are also called *nested services* as the provider combines them to create a composite service [ETSI 2018c]. Under multi-domain orchestration, many orchestrators manage external dependencies, unlike internal domains where a single orchestrator manages all the dependencies that belong to a unique domain.

Figure 2.3 shows a composite service (C) having two dependencies (A, B). In this example, the two *dedicated* network services that belong to the *composite* network service are offered by different domains. The service C has internal and external connection points to deliver features to consumers. Detailed information about the *dedicated* network services A and B is unavailable to the orchestrator that manages service C, such as topology, lifecycle management policies (e.g. scaling rules), and communication endpoints. This is because the services A and B are external dependencies managed by other administrative domains. The limited knowledge of orchestrators outside their administrative domain enables providers to share their resources without compromising the orchestrators' privacy or autonomy. For example, orchestrators can set up complex service function chains without access to detailed information of other orchestrators [Liu 2020].

2.3 Networking Function Virtualization management and orchestration

In this section, we detail two types of orchestration, namely single and multi-domain orchestration. Then, we go more in-depth for the latter, since multi-domain orchestration introduces new assumptions that completely change the network service's lifecycle management.

2.3.1 Single-domain orchestration

In non-virtualized networks, network functions run on top of vendor-specific software and hardware [Luizelli 2015]. This leads to stagnant service deployment and management,

which in turn reduces the revenue for service providers. NFV introduces a more flexible approach to the provision of such services [ETSI 2014]. In summary, it considers:

- Software decoupled from hardware. As a network service is no longer a collection of integrated hardware resources from the same vendor, the evolution of the software and hardware are independent of each other.
- Flexible deployment. The detachment of hardware and software enables reassigning and sharing resources. Thus, assuming there is a pool of resources in a location, the service deployment is more automated by leveraging different technologies.
- Dynamic operations. The decoupling allows for greater flexibility to scale, migrate, and heal network services on the fly. This enables finer granularity. For example, the orchestrator can identify network traffic to only provide a subset of resources by increasing or reducing capacity.

Figure 2.4 shows the high-level framework for NFV according to the ETSI standard [ETSI 2014]. The framework enables the dynamic construction and management of VNF instances. It considers the following layers:

- Virtual Network Function. The software implementation of a network function that runs on top of a virtualized infrastructure.
- The virtual infrastructure. It includes many physical resources and a virtualization layer to support the execution of VNFs. Such resources include compute, storage, and network hardware.
- The NFV Management and Orchestration (MANO). It covers the lifecycle management of VNFs and VNF-based network services. The orchestrator on top communicates with both the VNF Manager and the Virtual Infrastructure Manager. It also communicates with the OSS/BSS to send/receive instructions regarding services.

2.3.2 Distributed multi-domain orchestration

Harnessing the benefits of network services requires control to support their lifecycle. Under the NFV paradigm, the Network Functions Virtualization Orchestrator — simply orchestrator — manages the network service's lifecycle tasks (i.e. building, instantiating, executing, reconfiguring, and monitoring) [ETSI 2020b]. The orchestrator also administers all the systems and networks operated by a single organization [ETSI, NFVISG 2016]. It also handles the Virtual Infrastructure Manager and VNF Manager to support all the VNFs that compose the services. Multi-domain orchestration extends the capacities of the single orchestrator by offering network

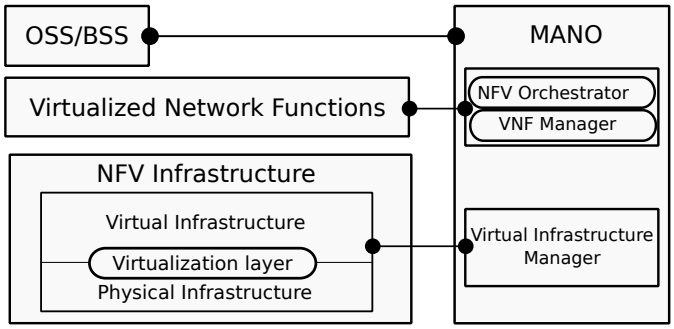

Figure 2.4: Example of single domain architecture. The three biggest parts belong to the MANO reference architecture.

services within the same organization and facilitating these services to another network operator [ETSI 2018a]. Three types of architectures support the multi-domain orchestration as defined by the ETSI Standard: Centralized, distributed, and hybrid [Rosa 2015, Katsalis 2016]. Centralized solutions establish a global orchestrator that coordinates other orchestrators via vertical calls. Distributed architectures lack a central coordinator and orchestrators communicate to support the network services' lifecycle. Hybrid ones create hierarchies where orchestrators coordinate both horizontally and vertically. All the previous architectures enable a federation. However, for this research, we consider only distributed architectures, since they offer the greatest flexibility and autonomy for service providers.

2.3.3 Differences with single-domain orchestration

Although both types of orchestration share tasks, they are different. Figure 2.5 shows these differences. We summarize such differences between single-domain and multi-domain orchestration in the following list:

- Limited information. Under single-domain orchestration, a single global orchestrator manages all resources in its administrative domain. In multi-domain orchestration, no global orchestrator exists; thus, orchestrators have only partial information solely in their administrative domain. Examples of this include the services topology, lifecycle policies, such as scaling or migrating rules, and endpoints.

- Shared resources. In multi-domain orchestration, the orchestrators share VNFs and network services among multiple providers. This contrasts with a single-domain orchestrator, where may orchestrator can instantiate the services in different points of presence that belong to a unique administrative domain. Thus, many orchestrators manage services.

- Lifecycle management. Since orchestrators can share services and orchestrators have limited information, the network services' lifecycle management changes from

the single domain orchestration. In summary, when orchestrators share services, the orchestrators must send grant messages to other orchestrators before applying certain lifecycle operations (e.g., scaling or migrating a VNF). This prevents unwanted side-effects of a lifecycle operation. For example, migrating a shared VNF can lower the latency in one domain, but increase it in another domain; another example would be the insufficient resources of one orchestrator to scale a given shared VNF.

- Asynchronous Communication. Orchestrators send messages to coordinate themselves. Unlike in single domain orchestration where it is possible to establish temporal references because of a single global orchestrator, in multi-domain orchestration, it is not always possible to establish them because of limited information of each orchestrator and the non-deterministic network channels used to communicate. Thus, when applying lifecycle managing operations to shared resources, the orchestrators need to establish an order of execution.

- Agnostic interfaces and heterogeneous services. In a single domain federation, the orchestrator models, instantiates, and manages VNF-based network services. By using components in the same administrative domain, the orchestrator manages network services. The multi-domain approach extends this provision philosophy by integrating different components of a service in a federation. There are two federation types: close and open. Service providers know all orchestrators in close federations. Open ones allow orchestrators to come and leave at will; thus, service providers need agnostic interfaces to support this more flexible service deployment philosophy. In this research, we consider trustful service providers.

- Orchestrator coordination. Unlike the vertical approach of single domain orchestration, where all the information must pass towards the global orchestrator, multi-domain orchestration considers horizontal coordination where the information is distributed among different orchestrators. This prevents the point of failure for single-domain orchestration and enables dynamic federations.

2.3.4 Distributed multi-domain federations

A Federation is a collective group of service providers who share resources to support complex network services [Baranda Hortiguela 2020]. This reduces the costs of each individual provider and extends the capacities despite the limited resources of each provider. Orchestrators access the network services of different providers by negotiating the limited resources among them. This creates *shared* network services that multiple services can use in the federation. *Composite* services in such federations can be *shared* services [ETSI, NFVISG 2018]. They can also have these types of services as external dependencies. Thus, for the rest of the manuscript, we use the terms *composite* and

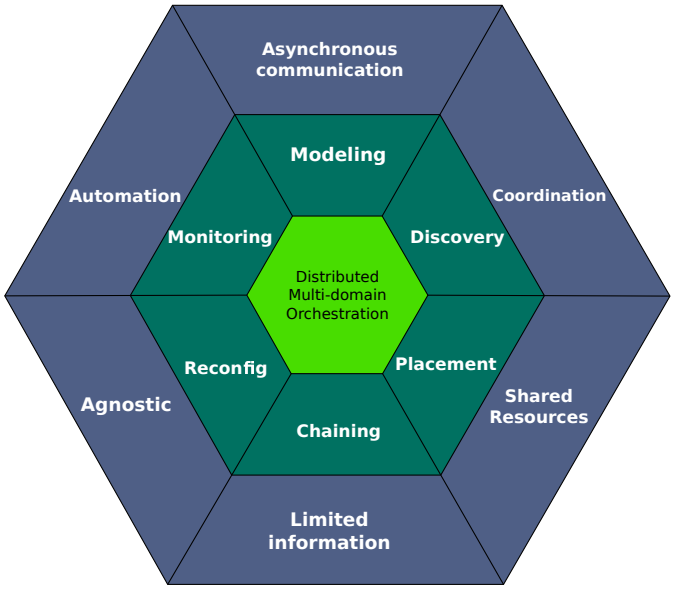

Figure 2.5: The layers of distributed multi-domain orchestration. The second layer has the tasks of single-domain orchestration. The third layer contains all the overhead of multi-domain orchestration.

shared services interchangeably. Due to *shared* services in a distributed multi-domain federation, the orchestrators coordinate to overcome the limited knowledge of each participant. On the one hand, these federations offer flexible, scalable, and extendable network services; on the other, they add overhead to the service's lifecycle management. We consider closed federations that reject new participants to enter; thus lowering the overhead, but keeping the benefits of a federation. Figure 2.6 shows an example of a distributed multi-domain federation composed of three domains. Each domain has its orchestrator and different technologies, such as NFV, SDN, or legacy. The orchestrators communicate with each other or through a marketplace.

2.3.5 Lifecycle management of network services in multi-domain federations

Network services must meet their required service level agreement despite changes in the network. Such a service reconfiguration triggers, by energy consumption, fault tolerance, higher revenues, or improvement of the Quality of Service (QoS) [Kim 2016, Eramo 2017a, Yang 2018, Eramo 2019a, Wang 2018, Liu 2017]. The ETSI standard identifies different tasks at the service level, such as scaling, migrating, and restoring a network service to meet the service level agreement requirements of a *composite* service [ETSI 2019a]. In multi-domain federations, ETSI defines a special communication reference point between orchestrators, called the orchestrator to orchestrator

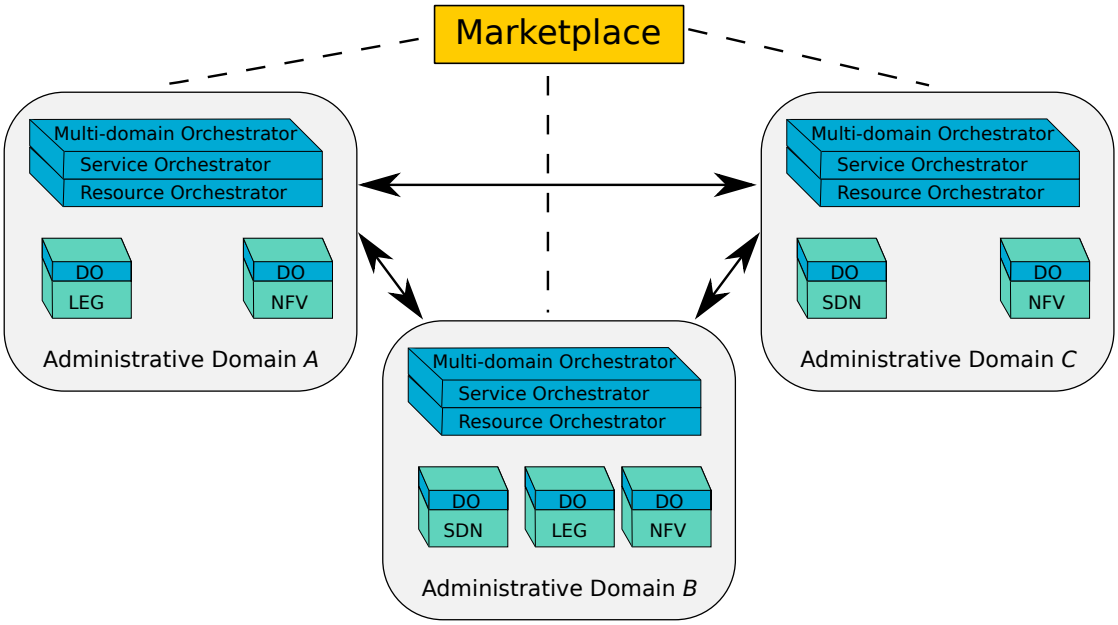

Figure 2.6: Example of the distributed multi-domain federation. Each administrative domain is managed by an orchestrator. They communicate directly or by a marketplace.

or – *or* [ETSI 2020a]. Figure 2.7 shows the reference point, where the orchestrators can set up a network service for a content delivery network by chaining four VNFs: (1) A translator (TRA), (2) streamer (ST), (3) encoder (ENC), and finally (4) decoder (DEC). Different orchestrators manage these four VNFs. For example, the *NFVO-C* manages the DEC VNF; while the *NFVO-A* manages both TRA and ST VNFs, respectively.

The orchestrators coordinate over the *or-or* point, used for the exchanges between orchestrators in different administrative domains. This reference point enables interfaces to support complex multi-domain tasks via grant messages [ETSI 2018a]. According to the ETSI standard, the following tasks in multi-domain environments require coordination by sending grants over the *or* – *or* reference point:

- Scale Network Service.
- Terminate Network Service.
- Heal Network Service.
- Subscription/Notification.

In this work, we consider close federations. In them, a fixed number of trusted orchestrators expose connections points of their shared network services. Orchestrators communicate with each other via messages, since distributed multi-domain federations lack global references. Thus, there is a flexible hierarchy according to each service that

enforces the use of coordination among the orchestrators because of limited knowledge. Figure 2.7 shows an example. The orchestrator *NFVO-C* plays the role of a consumer and provider of network services. The VNF Managers in each administrative domain interact with their orchestrator. However, the orchestrators are not aware of the constituent VNF instances of the *shared* service instance and do not interact with the VNF Managers of other administrative domains. This is the case for orchestrator *NFVO-C* and the VNF managers of *NFVO-A* and *NFVO-B*, respectively.

2.4 Basic concepts in distributed systems

In a distributed system, entities communicate with each other by exchanging messages. It is assumed that there is no global reference and transmission delay is bounded but arbitrary. A distributed system is composed of the sets P, M, E which belong to the set of processes, messages, and events, respectively.

- Processes: Programs and instances running simultaneously that communicate with other programs. Each process belongs to the set of processes of P. A process

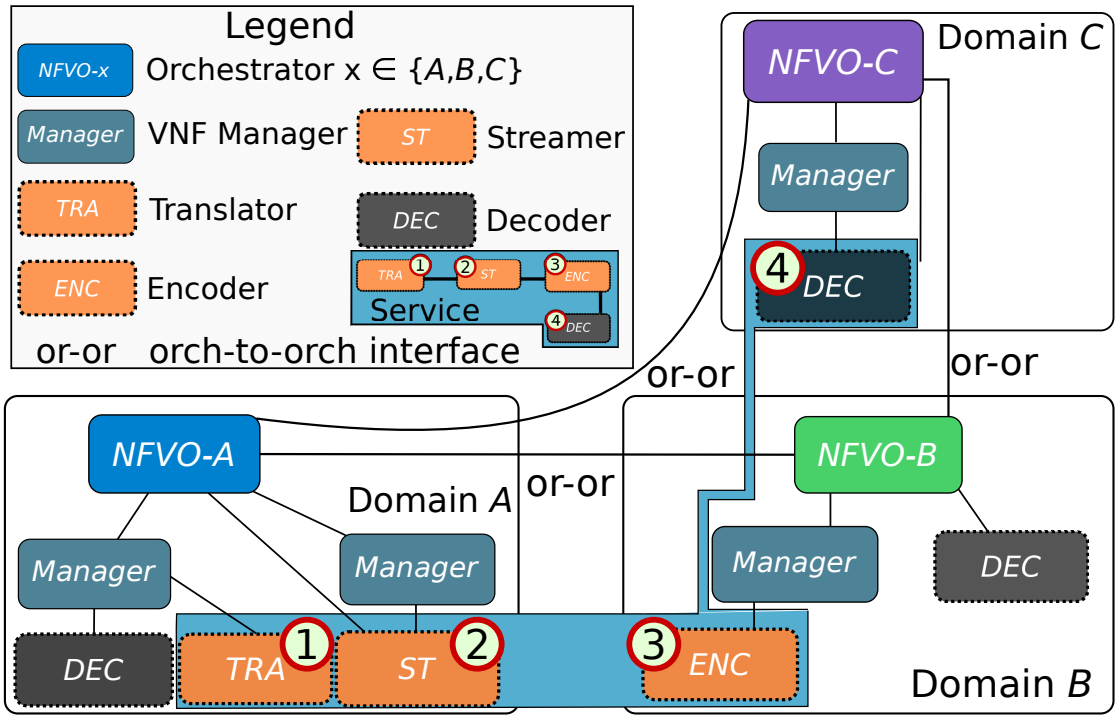

Figure 2.7: Network services federation provided using multiple administrative domains. An orchestrator can be both a provider and a consumer of other services. A VNF-based network service is created by chaining the Virtual Network Functions in the order specified by the numbers.

$p \in P$ communicates with another process $p' \in P$ by message passing over an asynchronous, non-deterministic, and reliable network.

- Messages: Abstraction of any type of message which contains data structures. Each message in the system belongs to the set M.

- Events: An event e is an action performed by a process $p \in P$. All events in the system belong to the set of events E. We consider two types: internal, external. An internal event occurs in a process locally hidden from other processes. An external also happens locally, but other processes see the event; thus affecting the global system state. For external events, orchestrators consider *send* and *delivery* events. A *send* event emits a message $m \in M$ executed by a process. *Delivery* events identify the execution performed of received messages by a process.

2.4.1 Relation with multi-domain orchestration

A federation of multi-administrative domains can be modeled as a distributed system where the orchestrators are processes that exchange messages. All the tasks in the lifecycle management of network services can also be modeled as events. Internal events are related to the execution of tasks only in a single administrative domain. External events are related to tasks in multiple administrative domains, such as grants for reconfiguring shared services. In subsequent chapters, we extend the abstract notions of messages and events of generic distributed systems for a specific problem or task in the lifecycle management of VNF-based network services.

2.5 Distributed multi-domain orchestration model for Network Function Virtualization

In this section, we describe the system model and notation used for all subsequent chapters. First, we introduce a generic distributed system model. Then, we extend this model for the context of NFV.

We adapt the previous generic system model (see Section 2.4) with the entities and types of messages for the context of multi-domain orchestration. Figure 2.8 shows the relationship between all the entities of the distributed multi-domain orchestration system model. The federations have two or more domains. Each domain manages both network services and VNFs. Depending on their dependencies, services can be external or internal. Internal dependencies have only VNFs or other services managed by a single administrative domain. External dependencies are managed by many administrative domains. Unlike the single domain, many administrative domains manage external dependencies in federated environments. All these domains' orchestrators coordinate through messages. For internal dependencies, a scale message suffices. For external dependencies, the orchestrator must gain a grant from all the other affected orchestrators.

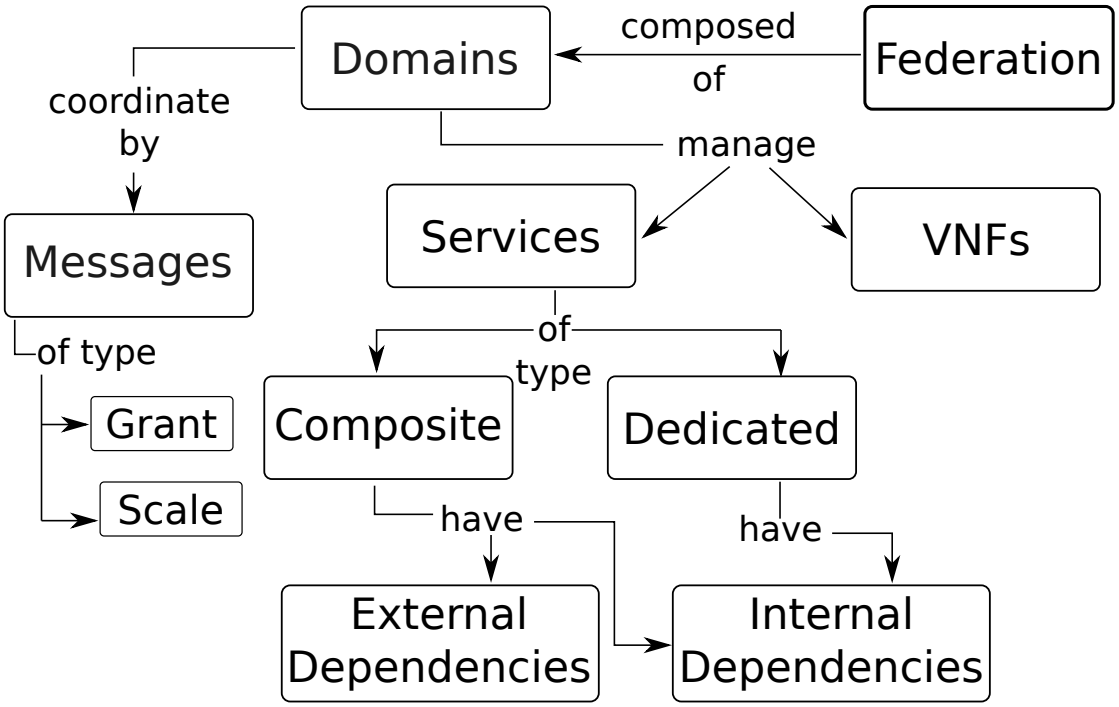

Figure 2.8: Relations between the entities of the distributed multi-domain orchestration system model. The Federation on top is composed of domains that manage services and Virtual Network Functions.

Since the orchestrator handles the lifecycle management of network services, we define a management relation as follows:

Definition 1 (Relation *is-managed by*)
The relation $\sim$ identifies the management of domain d for either a VNF, v, or VNF-based network service, s, according to:

1. *If $v \sim d \equiv True$, means the VNF v is instantiated at domain d.*

2. *If $s \sim d \equiv True$, means the service s has either: (i) only internal dependencies managed by the domain d (ii) if there are external dependencies, there is at least one VNF dependency v such that $v \sim d$ and all other external dependencies are managed by other domains.*

We develop the distributed multi-domain orchestration model by adapting the sets of processes P, messages M, and events E (defined in Section 2.4) to the NFV context by adding and defining the sets of events and messages which are specific to the model specification.

- Domains: The collection of systems and networks operated by a single organization. In the case of multi-domain federations we denote this set as $D = \{d_1, d_2, \ldots, d_p\}$. The number of domains p is known beforehand since we consider a close federation.

- Virtual Network Functions: The basic components to instantiate complex network services. We follow the ETSI standard by using the abstraction of virtual network function components that enables the VNF to operate. The super-set of VNFs V is composed of disjoint sets $V_1, V_2, \ldots, V_p$ where $\forall v \in V_i, v \sim d_i$ where v is a VNF and d_i is the i-th domain in the federation. Each $v \in V$ is defined as a function $f : x \Longrightarrow y$ where x, y are input and output traffic flows, respectively.

- VNF-based network services: The entities which offer complex solutions by aggregating VNFs with unspecified connectivity between them or according to forwarding graphs. A forwarding graph describes a topology of network services by referencing a pool of connection points and service acess points. The network service set is composed by a set of processes $S = \{s_1, s_2, \ldots, s_k\}$. Each service s is associated with a domain $d \in D$ denoted as $s \sim d$. Internal dependencies I_s of a network service s are VNFs $v \in V$ or network services $s' \in S$ such that $\forall v, s' \in I_s, v \sim d, s' \sim d$ where $s \sim d$. Similarly, external dependencies Γ_s are only network services such that $\forall s' \in \Gamma_s, s' \not\sim d$ where $s \sim d$. The set of total dependencies of service s is denoted as $\Delta_s = I_s \cup \Gamma_s$. Let Ω_s be the set of orchestrators that manage the external dependencies of service s such that $\forall o \in \Omega_s, \exists s' \in \Gamma_s \mid s' \sim o$. Network services in the set S can be of type *dedicated* or *shared* according to their dependencies [ETSI, NFVISG 2018]. We formalize this with the following:

  Definition 2 (Shared and dedicated network services)
  The service, s, managed by orchestrator o, belongs to the type shared if it is an external dependency of another service s' unmanaged by the same orchestrator o: $\exists s' \in S, o' \in O \mid s' \sim o', o \neq o', s \in \Gamma_{s'}$; otherwise, is dedicated.

- VNF Forwarding Graph. The VNFs and the links of a service $s \in S$ are ordered by a VNF-Forwarding Graph g that belongs to the set $G = \{g_1, g_2, \cdots, g_r\}$. The classifier rules that contain the information to forward traffic belong to the super set $C = \{c_1, c_2, \cdots, c_j\}$. Each set $c \in C$ has a list of matching attributes that show the protocol, ip, and ports to be visited by the income traffic of the VNFs; they belong to the set $c = \{ma_1, ma_2, \cdots, ma_u\}$. The rendered service path that relates the virtual links and VNFs belong to the set $X = \{x_1, x_2, \cdots, x_k\}$. Connection points have the input and egress points where the traffic flows belong to the set $P = \{p_1, p_2, \cdots, p_v\}$. Each VNF-FG g_r is defined by a tuple $\{id, C_r, X_r\}$ where id is an identifier, $C_r \in C$ is the *r-th* classifier rule set, and $X_r \in X$ is the *r-th* rendered service path. Each classifier rule $c \in C$ has a list of matching attributes ma that

belong to the set Ma. Similarly, each rendered service path $x \in X$ has a list of connection points cp that belong to the set Po.

We also define a function that computes the state of a VNF-FG by going trough each item in the data structure.

Definition 3 (State function)
The state function $\phi(g)$ takes as input either: a VNF-FG, a classifier rule $c \in C$, or a rendered service path $x \in X$ and outputs their current tuple according to the type of input.

$$\phi(g) = (c_g, x_g) = ([ma_1, \ldots, ma_u], [p_1, \ldots, p_v]) \tag{2.1}$$

- Messages: We extended the concept of abstract messages in distributed systems, described in Section 2.4, with the specific types required for reconfiguration of network services. All messages have the following parameters: $m = \{(sender, receiver, data, type)\}$. We consider the following type of messages:
  - *scale*: Increase or decrease the number of instances that belong to either a VNF or Network Service.
  - *response*: Acknowledgment of a complete scaling operation of an external dependency.
  - *notification*: To signal the sender of a Scale message there has been an update in the lifecycle management.
  - *grant*: Permission to scale external dependencies.
  - *updateVnffgClassifier*. Instructs an orchestrator to update a classifier of a VNF-FG
  - *updateVnffgRenderedServicePath*. Updates a rendered service path.
  - *notifyVnffgUpdate*. It signals to an orchestrator a change on a VNF-FG took place.
- Events: As mentioned in Section 2.4, there are two types of events: internal and external ones. The set of internal events $E_{internal}$ is the following:
  - *VnfMScaleRequest(v, data)* denotes the orchestrator's request to the VNF manager to initiate the scaling of VNF v specifying *data*.
  - *VimChangeResource(v, data)* is the event that the orchestrator sends to the virtual infrastructure manager to change either processor, storage, or network information of the VNF v according to the *data*.
  - *VimModifyConnectivity(v, data)* refers to the changing of connection points of VNF v by the virtual infrastructure manager.

- *VimInstantiate(V, data)* denotes the instantiation or shutdown of VNFs $v \in V$ for a particular service according to the *data*.
- *CheckCompositeNSConsistency(s)* denotes the verification before the scaling of service s. Since we consider the complete ETSI orchestrator's architecture as a single process for ease of understanding, all the details of this event are abstracted in a single execution; however, in reality, multiple entities play a role in this message, such as the VNF Manager.
- *ScaleNS(s)* denotes the scaling of the *dedicated* network service s whose only dependencies are internal. The dependencies belong to the set V_p managed by orchestrator o_p.

The external events considered are send, receive, delivery and the set is denoted as $E_{external} = E_{send} \cup E_{receive} \cup E_{delivery}$. Since we consider a single orchestrator per domain, we re-write the *is-managed by* relation (see Definition 1) as $s \sim o$ for any service $s \in S$ and orchestrator $o \in O$.

The set of send events E_{send} is the following:

- *ScaleCompositeNS(s, data)* denotes the petition to scale the *composite* network service s by using parameters in *data*, this could trigger multiple *ScaleNestedNS* or *SdNSLCMGrant* events by the external dependencies of s.
- *ScaleNestedNS(s, data)* denotes the petition to scale a *nested* network service s by using parameters in *data*, this will trigger a grant $SdNSLCMGrant(s')$ request event for the external dependency $s' \in \Gamma_s$.
- *SdNSLCMGrant(s')* refers to the request coming from orchestrator o to verify and scale service $s' \mid s' \sim o', s' \in \Gamma_s, o \neq o'$. This event can trigger multiple *ScaleCompositeNS*, *ScaleNestedNS*, and *SdNSLCMGrant* events, respectively.

The set $E_{receive}$ is composed of the following events:

- *RecResponseNestedNsScaling(s')*. This message is received by orchestrator $o \in O$ that manages service s. This service s has service s' as a external dependency. It denotes the answer (positive or negative) to a previous *SdNSLCMGrant* related to nested service $s' \mid s' \in \Gamma_s$.
- *RecResponseCompositeNsScaling(s')*. This message is received by orchestrator $o \in O$ that manages service s. This service s has service s' as a external dependency. It denotes the answer (positive or negative) to a previous *SdNSLCMGrant* related to *composite* service $s' \mid s' \in \Gamma_s$.
- *RecNSLifecycleChangeNotification({start, result}, s')* refers to the acknowledgement or result from orchestrator o' to o that has sent an event of type *ScaleNestedNS(s')* or *ScaleCompositeNS(s')* such that $s' \sim o'$ and $s' \mid s' \in \Gamma_s$ for any service $s \in S$.

The set $E_{delivery}$ has a unique event:

- *DlvNSLCMGrant(s′)* denotes that an orchestrator $o' \in O$ delivered a *grant* message sent by a *SdNSLCMGrant(s′)* event. It identifies the execution performed by the orchestrator o' associated to service s' managed by orchestrator o'. After delivery, o' will send a notification to the sender of the grant.

After defining the general system model we define the consistency models considered for the research.

2.6 Consistency models

Consistency is the desired property in distributed multi-domain orchestration [Vaquero 2019]. Next, we present a brief description of the concepts related to this topic. First, we introduce sequential consistency and discuss its drawbacks. Then, we describe eventual consistency as the alternative to such drawbacks, highlighting the limitations of this consistency model. Finally, we introduce strong eventual consistency, which eliminates the gap between consistency guarantees and performance.

2.6.1 Sequential consistency

Distributed systems need to ensure consistency because of concurrent operations on shared data. Sequential consistency was proposed to make the illusion of having the semantics of a single-system image system. Under sequential consistency, there is a single execution that follows a specific order. However, in reality, distributed systems, like multi-domain federations, run on top of multiple autonomous nodes, without global knowledge. Since these nodes communicate over a faulty network, non-deterministic conditions bring conflicts when nodes try to modify the state of a node concurrently. To prevent inconsistencies, solving consensus was proposed.

2.6.2 Causal consistency

Distributed systems need shared references, such as the physical time, to decide correctly how to execute transactions. But, because of the lack of global references, the difficulty arises to find if an event takes place in another. Thus, the distributed systems need another reference to circumvent the absence of synchronized clocks.

Logical time introduces an execution order between events based on a partial order known as the Happened-Before Relation that establishes a precedence order between two events in the following way [Lamport 1978]: let e and e' be two events causally related. According to the happened-before relation, e *happened before* e' if there is a transference of information from e to e'. Thus, according to the relation, e must be processed before e'. Formally, the Happened-Before Relation denoted "$\rightarrow$", is defined as follows:

Definition 4 (Happened-Before Relation)
The relation $\rightarrow$ is the smallest relation on a set of events E satisfying:

1. *If e and e' are events belonging to the same process and e originated before e', then $e \rightarrow e'$.*

2. *If e is the sending of a message by one process and e' is the receipt of the same message by another process, then $e \rightarrow e'$.*

3. *If $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$.*

The causal consistency allows to order events in a set according to a partial order. For the set of events of our general system model (see Section 2.4) we define a causal delivery order in Definition 5 as follows:

Definition 5 (Causal order delivery in distributed models)
$\forall((send(e), send(e')) \in E, send(e) \rightarrow send(e') \implies delivery(e) \rightarrow delivery(e')$ *for any two internal events* $e, e' \in E$. *We denote* $\hat{E} = \{E, \rightarrow\}$ *as the set of events causally ordered.*

The set E for events (see Section 2.4) is augmented. We consider internal and external events as defined by the sets $E_{internal}$, $E_{external}$, respectively such that $\hat{E} = \{E \cup E_{internal} \cup E_{external}, \rightarrow\}$. For simplicity, we represent the send events of set E_{send}: *ScaleCompositeNS*, *ScaleNestedNS*, *SdNSLCMGrant* as $send(s)$. The delivery event of set $E_{delivery}$ *DlvNSLCMGrant* as *delivery(s)* for any service $s \in S$. This set is causally ordered by the $\rightarrow$ relation (see Definition 4).

Causal consistency ensures an execution order for events that are related. This means causality can capture the temporal information for these events without global references. However, for certain applications, orchestrators can use a lesser consistent model. The trade-off between performance and guarantees might be too detrimental for certain applications. Thus, the literature proposed eventual consistency to get better performance at the cost of consistency guarantees.

2.6.3 Eventual consistency

Consensus is the convergence to a common value among all participants [Wei Ren 2005]. It achieves sequential consistency for distributed systems. However, the high complexity of implementing consensus and its low performance make it a bottleneck for distributed systems [Howard 2020, Xiao 2020]. To improve performance on non-critical applications, eventual consistency was proposed [Bailis 2013]. Informally, eventual consistency guarantees that if no additional updates are made to a given data, all reads to that item will eventually return the last updated value [Vogels 2008]. Eventual consistency is stated in Definition 6 [Shapiro 2011b].

Definition 6 (Eventual Consistency)
Eventual delivery: *An update delivered at some replica i is eventually delivered to all replicas: $\forall i, j : f \in c_i \implies \Diamond f \in c_j$, where f is an update, and c_i, c_j are replicas of the same node c.*
Convergence: *Replicas that have delivered the same updates eventually reach equivalent state: $\forall i, j : \Box c_i = c_j \implies \Diamond \Box s_i \equiv s_j$.*
Termination: *All method executions terminate.*

Under Eventual consistency, all participants eventually "converge"; however, it does not provide single-system image semantics as it does not specify which value is eventually chosen and no time of convergence is known [Bailis 2013]. This weak model applies to some classes of problems [Bailis 2013]. Thus, replicas can execute an operation without synchronizing *a priori* with other replicas, making data available at any given moment.

2.6.4 Strong eventual consistency

Despite the consensus being moved off critical paths of applications, reconciliation is still complex to achieve [Shapiro 2011a]. Strong Eventual Consistency was proposed to remove consensus altogether. We take the formal definition of Strong Eventual Consistency [Shapiro 2011b].

Definition 7 (Strong Eventual Consistency (SEC))
An object is strongly eventually consistent if it is Eventually Consistent and:
Strong Convergence: *Replicas that have delivered the same updates have equivalent state: $\forall i, j : c_i = c_j \implies s_i \equiv s_j$.*

To achieve SEC, Conflict-Free Replicated Data Types (CRDTs) (e.g., those data types in which operations commute) can ensure that there are no conflicts, hence, no need for consensus-based concurrency control [Shapiro 2011b]. Currently, there is a portfolio of CRDTs for counters, registers, sets, and graphs that act as a building stone for more complex algorithms [Shapiro 2011a].

Next, we introduce the case for coordinating orchestrators and highlight the limitations of current migration algorithms.

Chapter 3

Migrating shared Virtual Network Functions in federations. The case for coordinating orchestrators

In this chapter, we highlight the differences between algorithms for traditional single-domain and multi-domain orchestration. We show that heuristic algorithms, for migrating VNFs in single domain orchestration, result in unwanted side-effects when the orchestrators share Virtual Network Functions (VNFs). Indeed, because of limited knowledge of each orchestrator, the orchestrators must coordinate to reduce the unwanted behavior. Single-domain orchestration approaches lack the subtlety required for multi-domain problems. This makes coordinating orchestrators almost a requirement for federated deployments. Figure 3.1 shows the current step towards the coordination-free orchestration algorithm. In this chapter, by coordinating orchestrators, we show it is possible to reconfigure network resources consistently; in this case, the migration of shared VNFs. This is the first step towards coordination-free orchestration, as we focus on the VNF-level only. In the following chapters, we will extend this approach to consider end-to-end VNF-based network services.

3.1 Introduction

VNF migration is a key task in the lifecycle management of network services [Xia 2016]. It involves changing the location of VNFs to another Point of Presence (PoP); with stateful VNFs, migration also transfers the states between old and new instances. In a federated environment, reconfiguration operations such as migration might not lead to the desired impact, or even to the totally opposite effect sometimes. For instance, migrating a VNF would eventually increase its performance and/or reduce its cost; however, this could trigger the degradation of a whole service chain where this VNF is involved with others that did not migrate. Therefore, it is necessary to coordinate the migration of VNFs in the chain, even more in federations where services contain shared VNFs as external dependencies. This chapter focuses on the problem of migrating shared VNFs in federated environments. Our research questions are:

- What problems can occur during the migration of shared VNFs under multiple service function chains in Network Function Virtualization (NFV) federated environments where orchestrators have only local information?
- How can seamless migration of shared VNFs be achieved in NFV federated environments when there is no global view of the federation by a single entity?

We propose a coordination algorithm for shared VNFs in federated environments that relies only on the local domain knowledge of orchestrators. The algorithm is designed and implemented using open source technologies. It achieves a lower overhead in terms of service violations than the traditional greedy approach that minimizes only the current VNF without considering the impact of other VNFs in the chain. Our principal contribution is the migration of shared VNFs in federated environments and how can it be achieved despite having only local information of each orchestrator.

The rest of the chapter is as follows: First, we describe a use case to illustrate the concept of sharing VNFs in Section 3.2. Also, in Section 3.2.1, we define the problem of sharing VNFs in a distributed multi-domain federation. Then, in Section 3.3, we describe the state of the art and highlight its limitations. After, we present our proposed coordination algorithm between orchestrators to enable sharing VNFs in Section 3.4. We show the algorithm satisfies two important properties to validate the correctness of the proposed algorithm in Section 3.4.2. Next, in Section 3.5, we evaluate our proposed algorithm. Finally, we present the lessons learned and perspectives in Section 3.6. We present the notation for this chapter in Table 3.1 (see Section 2.4 for a more detailed description of each variable).

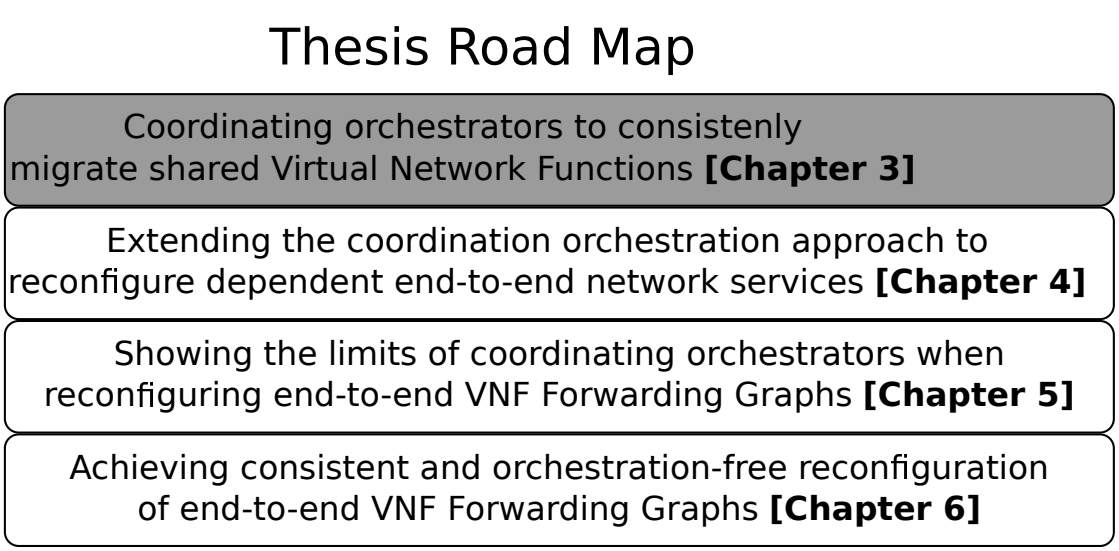

Figure 3.1: Thesis roadmap. In this chapter, we study the problem of migrating shared Virtual Network Functions.

Table 3.1: Notation for this chapter. Some of the variables were defined in the system model (see Section 2.4 for a more detailed description of each variable). The other variables are defined in this chapter.

Variable	Meaning
F	A federation composed of many administrative domains.
$D = \{D_1, D_2, D_3\}$	The set of domains that create a federation.
$\Upsilon^f = (V^f, E^f)$	A graph where V^f are VNFs and E^f are links from domain D_f.
$s_i \in S$	The i-th service from the service set S.
α_k	The k-th VNF in a service chain.
Ξ_{α_k}	The non-functional requirements of the VNF α_k.
R_{α_k}	The set of VNFs that must process traffic before the VNF denoted as α_k.
P_{α_k}	The internal states for VNF α_k.
Q_{α_k}	The buffer of states when a stateful VNF migrates.
Λ_{α_k}	The set of affected VNFs for VNF α_k.
$delay(\alpha_k)$	A function that computes the delay for VNF α_k.
Ψ	Time required to establish a virtual link two VNFs.
$\varkappa$	Time to migrate a VNF in the chain.
Π	The handover time of the flow between a pair of VNFs.

3.2 The problem of migrating shared Virtual Network Functions

Consider the case of a city where a monitoring system is deployed to help citizens and prevent disasters. A small council of people must decide to coordinate the city's resources to mitigate the damage done. To make an informed decision, multiple sources of information are obtained from crowdsensing. The citizens send data when they stream video and audio from their smartphones. After data is sent by the citizens, it is processed to get contextual information and presented to consumers of the monitoring service. One critical requirement is the synchronization between data and the information presented. Thus, NFV paradigm is considered.

Multiple domains are used to instantiate VNFs, as shown in Figure 3.3. Data comes from heterogeneous devices, thus the VNFs have to be instantiated in different domains. Such flexible deployment requires the orchestrators to share VNFs to support more complex services. In this deployment, no single orchestrator knows the entire information about the federation. We consider only cooperative federations.

During the operation of a network service, the failure of a VNF can trigger a migration. We illustrate an example of a failed service after migration in Figure 3.3. Domain A will instantiate a new mixer VNF based only on his service constraints. However, the new VNF affects another service in domain B since the new delay is greater than the service's constraint. Failure happens since orchestrators have limited information and do not coordinate.

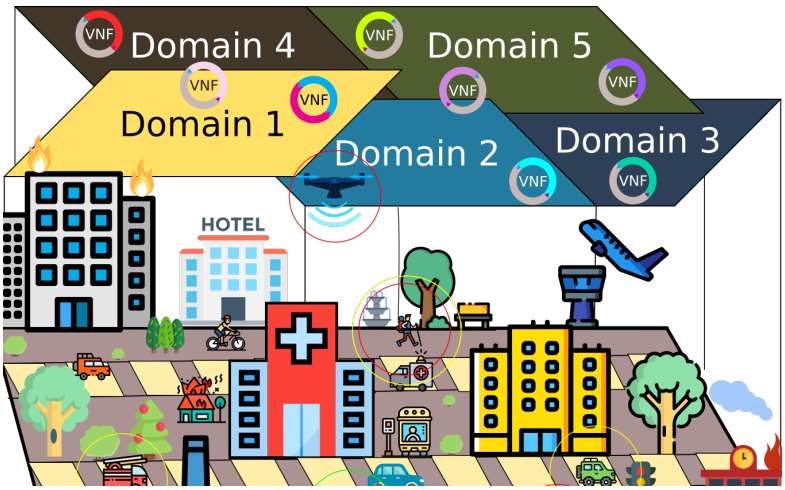

Figure 3.2: Multiple domains build the Network Function Virtualization federation. In each domain, there is an orchestrator that manages all the Virtual Network Functions in the domain.

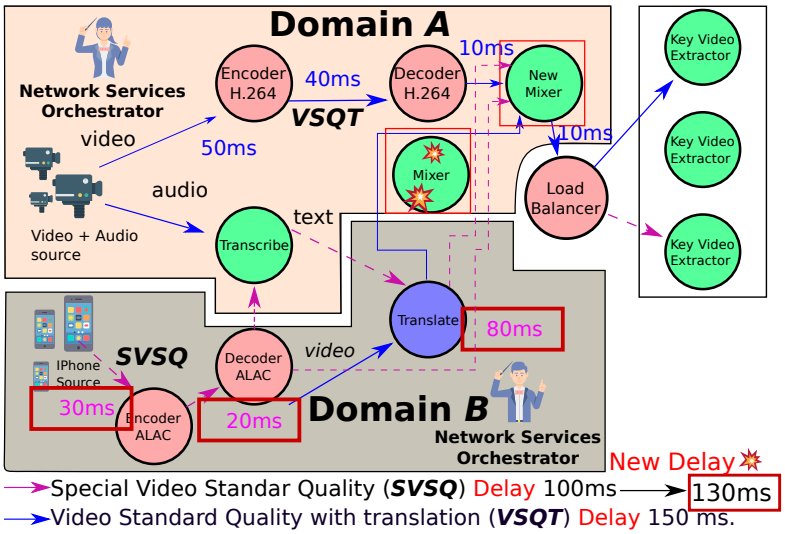

Figure 3.3: A Virtual Network Function is shared among two services from different domains. In case of a migration, domain A will instantiate a new Virtual Network Function; however, migration affects the other service since the new delay is greater than the maximum.

3.2.1 Formal problem definition of migrating a shared Virtual Network Functions

We consider a federation of orchestrators deployed in multiple domains with PoPs that enable the instantiation of VNFs. Each architecture for the domain is aligned with the European Telecommunication Standards Institute (ETSI) standard. For each domain there is an orchestrator that manages the deployment, instantiation, and reconfiguration of network services. The federation is modeled as a set F that contains domains $D_1, D_2, D_3, \ldots, D_f$ where D_f is the fth domain in the federation. For each domain D_f, there is a set $\Upsilon^f = (V^f, E^f)$ where V^f is the set of VNFs and E^f is the set of links from domain D_f. Also, for any domain D_f there exists an orchestrator o_f that manages all the VNFs that belong to V^f. There could be a connection of VNFs from one domain D_f to another $D_{f'}$ such that $\{(p,q)|p \in V^f, q \in V^{f'}, p \neq q\}$. Services are in the set S, such that for any given $s_i \in S$, it can be instantiated by VNFs from multiple orchestrators.

A service s_i is an ordered set of VNFs α_0, α_1, α_2, $\cdots$, α_n where n is the length of the service s_i managed by orchestrator o_f. A VNF p is considered shared if for any pair of services s_i, $s_{i'}$, it belongs to a given $\alpha_k \in s_i$ and $\alpha_{k'} \in s_{i'}$ such that $s_i \neq s_{i'}$ where $p \equiv \alpha_k \equiv \alpha_{k'}$. We denote R_{α_k} as the set of VNFs that must process data traffic before α_k in a given service s_i. Furthermore, every VNF α_k that belongs to a service s_i has non-functional requirements denoted Ξ_{α_k}. Also, a VNF α_k has internal states denoted as P_{α_k} for incoming messages from the previous VNF, Q_{α_k} for buffered messages. A VNF α_k must migrate if a requirement of Ξ_{α_k} is not met.

For the migration scenario, the current migrating VNF is denoted as α_k, the new VNF as $\alpha_{k'}$, the previous VNF in the chain α_{k-1}, the next VNF as α_{k+1}, and the first one in the chain as α_0, respectively. We denote the set of affected VNFs $\Lambda_{\alpha_k} \subseteq R_{\alpha_k}$ as the list of previous VNFs α_{k-1} such that after the migration of α_k, $\Xi_{\alpha_{k-1}}$ is no longer met. Therefore, the migration of every VNF in Λ_{α_k} must precede the migration of α_k. For the next sections, we make the following assumptions:

1. There is reliable communication among orchestrators.
2. Sent messages arrive in the order they were sent.
3. There are no dependency cycles of VNFs.

Given a federation of orchestrators, where each orchestrator has only local knowledge of its domain, the goal is to coordinate them to satisfy the non-functional requirements of the services and VNFs being used in the federation before and after migration. Inspired by the transparent flow/state migration problem [Yang Wang 2016], seamless VNF migration in a service chain has the following constraints:

1. **State consistency**: The state of the target VNF after the migration is the same as the source VNF if no migration occurred.

2. **Packet safety**: The execution of packets is safe (i.e. loss-free and order-preserving).
3. **Finite time**: Migration time is bounded.
4. **Efficient overhead**: The messages sent in the control plane are sufficient and necessary.

Next, we present in more detailed the relevant works in the literature that focus on the problem of VNF sharing. We highlight their limits, and show how our work extends the state of the art.

3.3 The state of Virtual Network Function migration in federations

3.3.1 Virtual machine migration

Migration of VNFs was first attempted on Virtual Machines (VMs) [Clark 2005] and containers [Ma 2019]. These migrations can be disruptive or not, usually referred to as cold or live [Eramo 2017b]. Cold migration is done with stateless VMs, while live migration requires state transfer among the different VMs. However, VM migration solutions are not suitable for VNFs as they are energy demanding [Eramo 2017a] and resource consuming [Mattos 2015]. Migration of the whole VM is disruptive since many VNFs could reside in a single VM [Xia 2016]. Therefore, migration of VNFs to different PoPs requires new techniques that consider the limitations of complete VMs migration.

3.3.2 Virtual Network Function migration

The migration of VNFs has been investigated from two directions: placement [Yi 2018] and flow/state migration [Liu 2016]. On the one hand, the VNF placement problem refers to moving a single VNF to a new point of presence because of specific requirements, such as load balance [Bari 2015] or delay minimization [Zou 2018]. This NP-hard problem [Tomassilli 2018] has been approached by various heuristics (i.e. [Hawilo 2017, Abu-Lebdeh 2017]). On the other hand, the problem of flow migration is how to orchestrate state and packet transfer to ensure state consistency with high efficiency and minimal overhead. It is closely related to the migration of switches in SDN, since the VNF's state changes based on incoming packets [Nobach 2017]. They both consider liveness, safety, transparency, and load balancing properties for the packages being migrated [Ghorbani 2014]. However, they are distinct problems based on the level of abstraction: SDN is concerned with the reconfiguration of flow tables and routing by separating data and control information, while VNF is mostly concerned with the separation of dedicated hardware through virtual functions at the application level. Both can have the migration of state, but SDN is concerned with the state of the packages

and flow of information, while VNF state migration is mostly concerned with the state of the application. Thus, the migration of VNF is a higher level of abstraction than the migration of controllers. Existing flow/state migration techniques transfer a snapshot of the service instance and resynchronize inconsistent states [Clark 2005]. To alleviate the high-volume traffic by re-synchronization, flow termination and flow-quality degradation have been studied [Sugisono 2018]. Further improvements in control information sent during migration are explored in [Yang Wang 2016, Nobach 2017].

All the previous approaches of migration consider a single and isolated VNF instead of an end-to-end network service. This type of migration could trigger new VNF migrations that need to be taken into consideration. To do this, coordination among the multiple VNFs is necessary to ensure the migration to a new location that does not affect the performance of other VNFs in the chain. Our work, unlike the state of the art, considers the impact of shared VNFs on the whole service. Thus, orchestrators must ensure that the service is not affected by the migration of a VNF used by multiple services. Next, we introduce our coordination algorithm for orchestrators to support the migration of shared and stateful VNFs.

3.4 Coordination algorithm for stateful Virtual Network Function migration

The proposed solution is described in the pseudocode of Algorithm 1. First, the orchestrator checks if the previous VNF α_{k-1} is in his domain. If it is, it will begin the procedure to instantiate a new VNF to migrate the previous VNF if is necessary. If it is the case, then recursively call migration for the previous VNF. If the previous VNF α_{k-1} is not in his domain, the orchestrator will send a request to other orchestrators in the federation for information α_{k-1}(line: 7). If there is one orchestrator that manages the previous VNF, it will reply with information. Then, a check is done to see if the new VNF $\alpha_{k'}$ disrupts the previous one α_{k-1}(line: 9). If this happens, the orchestrator will request information for a new VNF α'_{k-1}(line: 10) and trigger the migration of the previous VNF (line: 12). In case the previous VNF α_{k-1} is not affected by the new VNFα'_k, then exchange states between the current migrating VNF and the others. A step-by-step procedure is shown in Figure 3.4 where messages are exchanged among orchestrators to achieve the migration. At the beginning of step 4, the exchange of messages begins.

3.4.1 Migration algorithm

The following algorithm migrates a stateful VNF α_k in a chain from $\alpha_0, \alpha_1, \cdots, \alpha_k, \alpha_{k+1}, \cdots, \alpha_n$ where n is the n-th VNF in a VNF-based service service. First, the orchestrator checks if the previous VNF in the chain belongs to its administrative domain. If it is the case, it will instantiate a new VNF in the domain (it could be located in another point of presence) and will execute the migration.

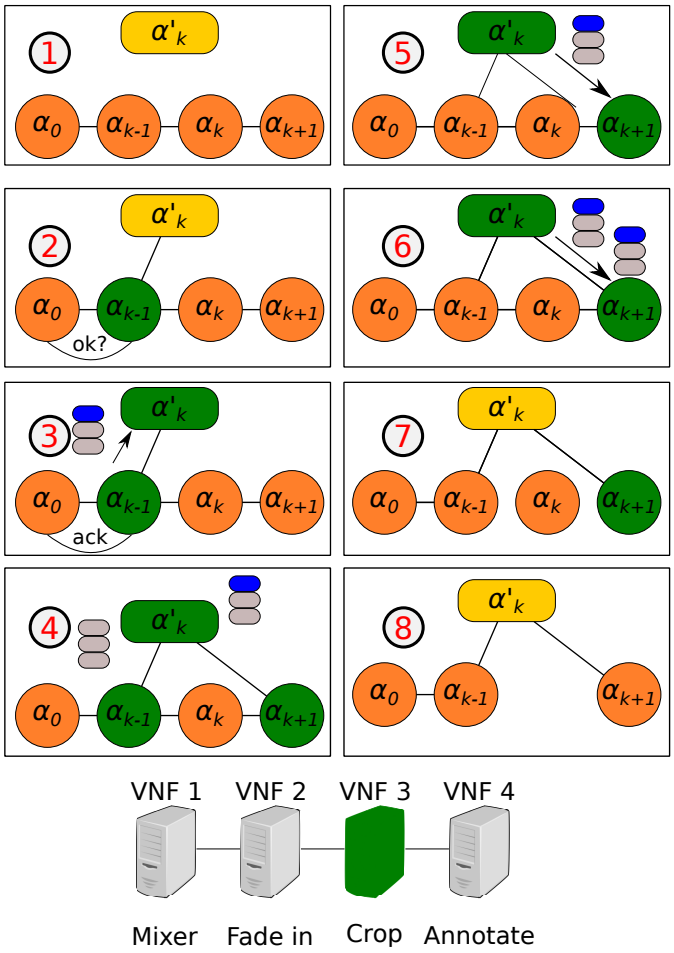

Figure 3.4: Steps to achieve migration. Each α_k is a Virtual Network Function in the chain. There is an exchange of messages among orchestrators of the Virtual Network Functions to coordinate the migration.

3.4.2 Validation for the algorithm

As stated in Section 3.2.1, seamless migration requires state consistency, bound migration, packet order-preserving, and minimal overhead. In this work, we focus on the first two since the last two constraints are related to routing and control, and are out of the scope of this work. The proofs rely on the assumptions listed in Subsection 3.2.1.

3.4.2.1 State consistency proof

The internal states of the VNF are linked to the two buffers Q_{α_k} and P_{α_k} and the virtual link between α_{k-1} and α_k. To ensure state consistency, the state of the source and target migration must be equal ($\alpha_k \equiv \alpha_k^{'}$). Before migration, the number of messages in buffer

Algorithm 1: Migration algorithm. Migrates a Virtual network function α_k to another one $\alpha_k^{'}$

```
1  if α_{k-1} ∈ o_f          /* Check if the previous VNF belongs to the same
     domain */
2  then
3      α'_{k-1} ← o_f.instantiate_new_vnf()    /* Create a new VNF to store the
         states */
4
5      if α_{k-1} ∈ Λ_{α_k} then
6          migration(α_{k-1} ∈ o_f, α'_{k-1})    /* If the previous VNF is affected,
             then recursively call migration */
7  o_f.request_information(α_{k-1})            /* The orchestrator queries all
     orchestrators to get the required information to migration */
8  if ∃α_{k-1} ∈ o_{f'} ∈ D then
9      if is_invalid(Ξ_{α_{k-1}}, Ξ_{α_{k'}}) then
10         α'_{k-1} ← o_f.request_new_vnf(Ξ_{α_{k'}})   /* Request a new VNF from the
             external orchestrator */
11
12         migration(α_{k-1} ∈ o_{f'}, α'_{k-1})        /* Start the migration with the
             external VNF and the new previous one */
13
14 exchange_states(α_k, α_{k-1})    /* Exchange states between the target VNF
     and the previous one in the chain */
15
16 exchange_states(α_k, α_{k'} )    /* Exchange states between the target VNF
     and the new one */
17
```

Q_{α_k} is r and the number of messages in P_{α_k} is 0 because the previous VNF has not received a signal from the orchestrator to begin a migration procedure. To achieve state consistency, we need to prove that no state is lost (i.e. $|Q_{\alpha_k}| = |Q_{\alpha_k^{'}}|$) after the migration. During the migration procedure, a link between the previous VNF α_{k-1} and the new VNF $\alpha_k^{'}$ is created and all the messages from Q_{α_k} are sent to $Q_{\alpha_k^{'}}$. By Assumptions 1 and 2, all the messages arrive in the same order they were sent. After sending the contents from R_{α_k} to $R_{\alpha_k^{'}}$ and P_{α_k} to $P_{\alpha_k^{'}}$, respectively, the number of messages in $Q_{\alpha_k^{'}}$ is r; since it is the same number of messages as before migration, it is what we wanted to prove. ■

3.4.2.2 Bound migration proof

The migration time from the current VNF α_k is the time required to: (1) establish the virtual link between α_k and target VNF $\alpha_k^{'}$, denoted as Ψ, (2) time to migrate the previous VNF α_{k-1} if necessary, denoted as $\varkappa$, (3) handover time of the flow between α_k and $\alpha_k^{'}$, denoted as Π. Because of the recurrence relation, this introduces a delay equivalent to $delay(\alpha_k) = d + delay(\alpha_{k-1})$ where d is the amount of time required for link establishment and handover. The objective is to prove that migration is finite (i.e. $\Psi + \varkappa + \Pi \equiv delay(\alpha_k) < \infty$).

It is necessary to construct the total sum of time required to achieve migration, as shown in Figure 3.4. Creating the virtual link and sending the messages to begin migration procedure is noted as t_0. Queuing messages until all received messages takes $\sum_{i=1}^{|\Lambda_{\alpha_k}|} t_1^i$. Creating a virtual link between $\alpha_k^{'}$ and α_{k-1} takes t_2. The waiting time to receive all the reply messages is noted as $\sum_{i=1}^{|\Lambda_{\alpha_k}|} t_3^i$. The dequeuing and transmission of data from α_k to $\alpha_k^{'}$ is bounded by Assumption 1 denoted as t_4. The waiting period to receive all acknowledgement messages from affected VNF is $\sum_{i=1}^{|\Lambda_{\alpha_k}|} t_5^i$. The transmission of messages to all the requesters not affected is achieved in t_6. Therefore, the total sum of time is:

$$delay(\alpha_k) = \sum_{i=1}^{3} t_{2i}^{\alpha_k} + \sum_{m=1}^{|\Lambda_{\alpha_k}|} \sum_{i=0}^{2} t_{2i+1}^{m(\alpha_{k-1})} \tag{3.1}$$

By Assumptions 1 $\sum_{i=1}^{3} t_{2i}^{\alpha_k}$ is bounded, as no message is lost. Since more migrations can be triggered, the time taken $\sum_{m=1}^{|\Lambda_{\alpha_k}|} \sum_{i=0}^{2} t_{2i+1}^{m(\alpha_{k-1})}$ is a recurrent equation. The time taken by α_0 denoted as $delay(\alpha_0)$ is finite since no virtual link is created due to $|R_{\alpha_0}| = 0$. By Assumption 1 and 3, all messages sent by any migrating VNF in the service chain of α_k are received ultimately until α. This means that if the transmission of any message takes place in time t, there is a time $t^{'}$ such that $t^{'}$ is greater than t but less than ∞. This means that for any NFV in the chain the time taken for migration is bounded $delay(\alpha_{k-1}) < \infty$. Therefore, the delay in Equation 3.1 is bounded, which is what we wanted to prove. ■

3.5 Evaluation

The aim of the performed evaluation is the comparison of our proposed algorithm performance with the traditional approach, where a new VNF is selected that maximizes the performance of the current VNF. We evaluate our algorithm in two steps: simulation and implementation.

3.5.1 Simulation setup and results

A simulation was done to grasp the performance of different approaches using many services. Each orchestrator creates services with a chain having different requirements in terms of resources and maximum delay tolerance. The considered parameters during the simulation are listed in Table 3.2. During the simulation, we first randomly placed the VNFs in valid PoPs. Then, the chains are created using a greedy approach that chooses the VNF that minimizes the service's constraints. After this setup, an update phase follows where each VNF is assigned a probability of required migration. Here, we consider the possibility of having unfeasible PoPs to migrate.

Table 3.2: The considered parameters for the simulation

Entity	**Parameters**	**Range**
PoP	Max CPU	2.2 Ghz
	Max memory	4 GB
	Delay	10 - 100 ms
VNF	CPU	180 - 200 Mhz
	Memory	380 - 400 MB
	Delay VNFM	9 - 10 ms
	Delay operation	32 - 70 ms
	Max Delay	100 ms
VNFM	Delay NFVO	10 - 30 ms
	Delay VIM	11 - 20 ms
Service	Max Delay	100 - 500 ms
	Required CPU	200 - 1000 Mhz
	Required Memory	400 - 1000 MB
	Required VNFs	2 - 10

We considered four algorithms for the VNFs: No migration, random, greedy, and our proposed algorithm. After migrating the VNFs, we checked the services. For each service, we evaluated the validity of each algorithm configuration according to the required migrations. We evaluated up to 200 services with different service chains. Each evaluation was done 100 times and the averages number of failures was registered. A service failure can occur in two cases: either the VNF is placed in an invalid PoP because of a lack of migration or the service does not satisfy the constraints originally placed. Figure 3.5 shows the results.

3.5.2 Implementation results

We measured the performance of our algorithm and compare it to the greedy approach using current tools found in the literature and industry. Based on a study of the state of the art [Komarek 2017, Gil Herrera 2016] we used the Open Source Mano (OSM) platform, which follows the ETSI standard, to develop a MANO software stack. However, currently, software tools focus only on single domain orchestration, thus we used MeDICINE [Peuster 2016] to set up a multi-domain environment. Each domain has an orchestrator that has local information regarding its domain. We implemented VNFs that enable the use case described in Section 3.2 that belong to two classes of services: short and large. In the short case, services are composed of 4 VNFs chained, while on the large is 8 chains. Each VNF used in the implementation is from the Content Delivery Network context (i.e. rotate, crop, annotate). The services were created at random from the pool of VNFs that were distributed among the domains of the orchestrators. Services are grouped such that multiple services can run in a trial. For example, a service can be composed of the following four VNFs: {*speed up*, *black white*, *annotate*, *composition*}. The execution begins at VNF *speed up* and ends at *composition*. For performance; we measure two metrics for both algorithms:

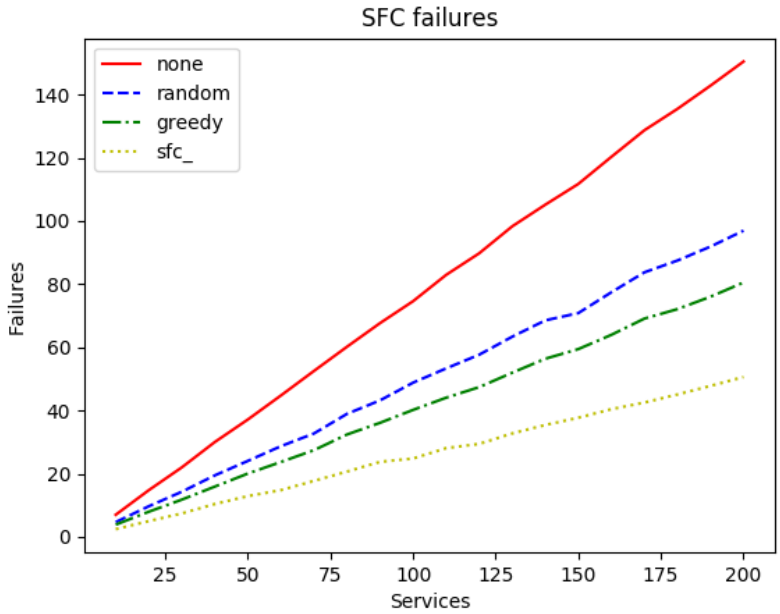

Figure 3.5: Average service failures for the migration algorithms, lower is better. A failure could be an invalid location of Virtual Network Function or a violation of service constraints. Our proposed algorithm gets the least number of failures.

- The time for migration: The time for the execution of the service plus the migration time for both the greedy and our proposed algorithm. This is used to measure the overhead of our proposed algorithm against the traditional state of the art approaches.
- The number of missed states: What states are not present due to a missed migration that was necessary for any service. This measures the consistency and safety of the algorithm as described in Section 3.2.1.

For the implementation, we considered that (i) VNFs have at least one affected VNF, (ii) there is a single migration of a VNF, and (iii) no cycles exist among the chains. We let these cases for future work. Also, we consider 4 orchestrators and each trial group has 3 services. These two parameters can increase, but for performance issues related to our experiment testbed, we chose this number and let a greater number of services for future work. Figures 3.6 and 3.7 show the results of the implementation.

3.5.3 Discussions

Based on the simulation and implementation, our proposed algorithm gets the least number of misses and service failures, with a slight overhead in the migration phase. For most cases, it surpasses the state of the art approach that does not consider dependency

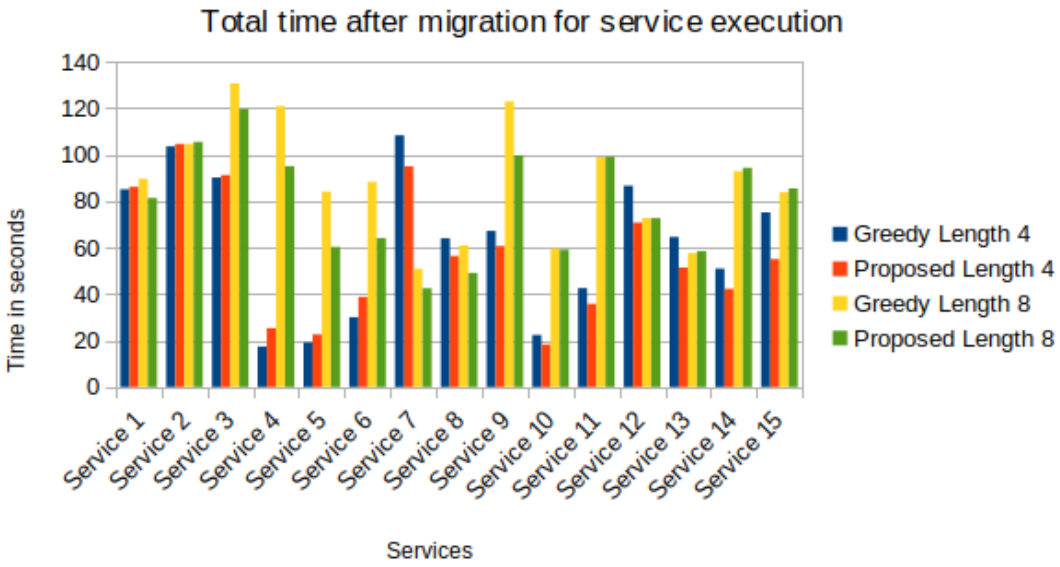

Figure 3.6: The time after migration for service in seconds, lower is better. Our proposed algorithm is slightly slower on average than the greedy migration. Variances in time occur due to network delay communication.

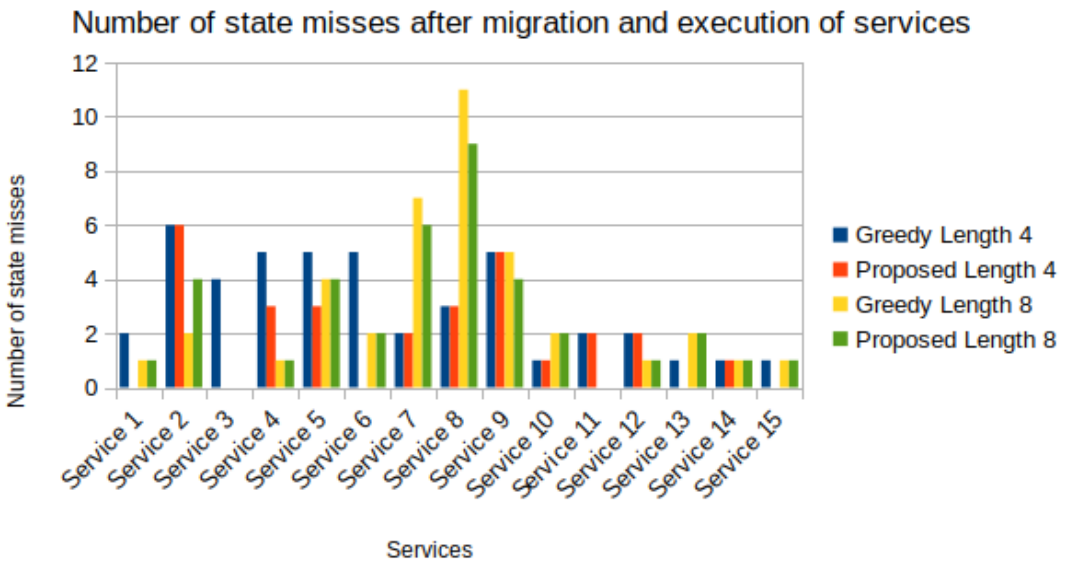

Figure 3.7: The rounded number of misses for services after migration for both algorithms, lower is better. Our proposed algorithm gets better results for most of the services. Variation appears to be related to dependency among VNFs in a chain for a service.

in the VNFs. However, there are some outliers in the number of misses as shown in Figure 3.7. A further inspection of services 7, 8, 9 that belong to the same group, we have the following services:

1. Service 7: *painting* ⇒ *annotate* ⇒ *invert_colors* ⇒ *Mirror Y* ⇒ *Rotate* ⇒ *Resize* ⇒ *Crop* ⇒ *speed_up*
2. Service 8: *fade_in* ⇒ *composite* ⇒ *annotate* ⇒ *crop* ⇒ *painting* ⇒ *mirror_X* ⇒ *rotate* ⇒ *invert_ colors*
3. Service 9: *annotate* ⇒ *fade_out* ⇒ *speed_up* ⇒ *mirror_x* ⇒ *fade_in* ⇒ *composite* ⇒ *crop* ⇒ *mirror_Y*

For this group, the migrating VNF is *rotate*. In the worst-case scenario, the migration could trigger the following chain of migrations: *rotate* ⇐ *mirror_X* ⇐ *painting* ⇐ *crop* ⇐ *annotate* ⇐ *composite* ⇐ *fade_in*. Thus, there are many affected VNFs, which seems to suggest that a finer approach to migration is required to minimize the disruption of services. Overall, the proposed algorithm has better results than the traditional greedy approach when orchestrators have access only to their local domain. The results highlight a gain in terms of consistency and loss-free with a small overhead associated with the coordination of orchestrators when they migrate shared VNFs.

3.6 Lessons learned and perspectives

This chapter focused on the problem of coordination of orchestrators to achieve the migration of shared VNFs in federations with no access to global knowledge. We proposed a novel coordination algorithm that migrates shared and stateful VNFs. The proposed algorithm show satisfies the restrictions of VNF migration considering the end-to-end network service. We simulate, implement and compare our proposed algorithm to the traditional optimization approach that maximizes current service performance. Our proposed algorithm gets better results with a slight overhead because of the coordination of orchestrators.

As described in this chapter, solutions to migrate shared resources like VNFs must consider the impact of other administrative domains. Unlike the state of the art, where migration solutions consider isolated VNFs, our proposed orchestration algorithm considers the VNFs could be shared among services that belong to many administrative domains. By coordinating the orchestrators, the number of failures diminishes compared to the heuristic approaches found in the literature, where optimization happens only in the local domain. This reduces costs for the service providers and better experience for the users. Thus, the case for coordinating orchestrators in tasks is appropriate to meet the challenges present with distributed multi-domain orchestration. However, inconsistencies still plague the reconfiguration of VNFs. In the next chapter, we will go more in depth in the reconfiguration tasks for VNF-based network services by considering scaling as a use case.

Chapter 4

Far beyond shared Virtual Network Function migration or: How to consistently reconfigure dependent network services by coordinating orchestrators

---

In the previous chapter, we explored sharing Virtual Network Functions (VNFs) among different services. Coordination was required as it reduced unwanted side-effects (e.g. partial service failures) from decisions taken locally in one domain. In this chapter, we explore more in-depth the coordination of orchestrators to tackle the more general problem of dependent reconfiguration. Unlike the previous chapter, where we focused on single VNFs, in this chapter we consider the end-to-end network service. Furthermore, contrary to the previous one, in this chapter, we prevent inconsistencies when reconfiguring dependent services. Figure 4.1 shows the current step towards a coordination-free orchestration algorithm. In this chapter, we generalize the dependent reconfiguration and identify two causes that bring inconsistencies. One of these conditions could be satisfied, in theory, without coordinating orchestrators. This brings light for the possibility to achieve consistent-free reconfiguration of VNF-based network services in federations.

4.1 Introduction

By exchanging messages over a well-defined interface, orchestrators create a federation. In it, unlike single-domain orchestration, many orchestrators manage lifecycle tasks of VNFs and VNF-based network services. These tasks include instantiating, reconfiguring, and monitoring the network services. Yet, federations bring new challenges for decentralized and asynchronous tasks [Katsalis 2016]. The orchestrators try to guarantee functional and non-functional properties of the services by reconfiguring shared services [Cisneros 2020b]. This reconfiguration must be consistent among all orchestrators.

Ensuring consistency in shared network services entails having the same up-to-date information of each orchestrator before and after they execute a lifecycle task. Before

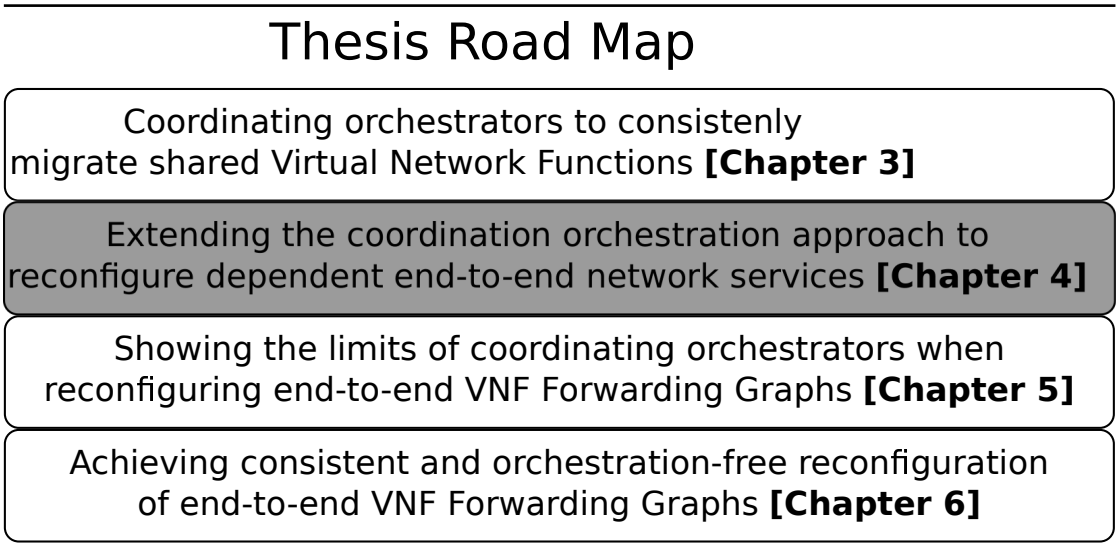

Figure 4.1: Thesis roadmap. In this chapter, we study the problem consistently reconfiguring dependent end-to-end VNF-based network sevices.

executing it, orchestrators must coordinate themselves to prevent unwanted behavior of network services. The orchestrators generate grants to notify and validate a reconfiguration when a network service has external dependencies. Each grant's recipient verifies the internal consistency in its domain by checking if the operation affects other network services.

The current specification of Network Function Virtualization (NFV) federations claims to ensure no undesired effect occurs while reconfiguring takes place, but no order or timing constraints exist between two consecutive grants [ETSI 2018a]. Thus, grants execute non-deterministically, in any order, without satisfying the service's external dependencies. Such execution could bring inconsistencies and lead to greater costs for the provider. For example, during the scaling of a shared service managed by many orchestrators, the service could be redundantly scaled and could also have deprecated connections. Thus, it is necessary to enforce a correct grant ordering for shared service reconfiguration to prevent inconsistencies induced by unsatisfied shared services' dependencies. Such correct grant ordering is the founding principle of the distributed approach of managing the consistent dependent reconfiguration problem for VNF-based network services in distributed multi-domain federations.

This chapter focuses on the problem of consistently reconfiguring shared VNF-based network services by identifying inconsistent patterns in the grant's order execution. Unlike the previous chapter, where we focused only on the migration of VNFs, this chapter deals with the full end-to-end network service. We consider the scaling of a network service as a use case; however, the approach presented can extend to other reconfiguration operations, such as healing VNF-based network services. Our research questions are:

Table 4.1: Notation for this chapter. Some of the variables were defined in the system model (see Section 2.4 for a more detailed description of each variable). The other variables are defined in this chapter.

Variable	Meaning
s	A service (either composite or dedicated).
m	A message, its type is defined in the text.
Γ_s	The external dependencies of service s.
$o \in O$	An orchestrator that belongs to the set of orchestrators.
$e \in E$	An generic event. Its type is defined in the text.
Ω_s	The set of orchestrators who manage service s.
T	Function that computes the time for an event.
$\rightarrow$	The happened before relation.

- How to consistently reconfigure VNF-based network services through grants given the limitations of local information and non-deterministic network conditions?

In this work, we consider on-the-fly dynamic dependent reconfiguration of VNF-based network services. We consider non-deterministic networks and dependencies among shared services. Based on such dependencies, we coordinate reconfigurations by ordering grants sent by orchestrators. Our contributions are:

- An inconsistent pattern for the NFV dependent reconfiguration is identified and formally defined from a temporal and logical perspective (Section 4.4.2, 4.4.3).

- A causally consistent orchestration algorithm is presented based on the proposed orchestration model to prevent inconsistencies that may occur when VNF-based network services have external dependencies (Section 4.4.4).

We show the viability of the approach via simulations using the scaling of VNF-based shared network services as the target operation. For this, we measure the inconsistencies, time to reconfigure, and message overhead and compared them to the current reconfiguration algorithm (Section 4.5).

The rest of the chapter is organized as follows: The problem and a use case are shown in Section 4.2. Section 4.3 contains the related work. We formalize the inconsistencies while reconfiguring network services in Section 4.4. Our solution is presented in Section 4.4.4. Section 4.5 contains the evaluation, results, and discussions. Our perspectives and insights are presented in Section 4.6. We present the notation for this chapter in Table 4.1 (see Section 2.4 for a more detailed description of each variable).

4.2 The Network Function Virtualization dependent reconfiguration problem

Dependent reconfiguration of VNF-based network services in multi-domain federations considers the internal VNFs, the services, and the external dependencies. A service's reconfiguration can be simple if the service has zero external dependencies (i.e. *dedicated* service); otherwise, it is dependent (i.e. *composite* service). Consider the *composite* service C shown in Figure 4.2, the service has two external dependencies in service A, B. Shared external dependencies introduce new challenges to the service's reconfiguration tasks, such as scaling. For example, Figure 4.3 shows a composite scaling with a single dependency (i.e. one dependency is shared by two services). The *NFVO-C* orchestrator manages a *composite* service s_0 (i.e. *NFVO-C* $\sim s_0$) with two external dependencies, s_1 managed by *NFVO-B*, and $s_3 \equiv s_2$ managed by *NFVO-A*, respectively. It is important to note that the other orchestrators, namely *NFVO-C* and *NFVO-B*, ignore that $s_3 \equiv s_2$ is a shared external dependency of both s_0 and s_1. This is because of the limited knowledge constrained by the local domain of each orchestrator. The orchestrators send grants to prevent service disruption when scaling a service with external dependencies/nested services. These grants allow the orchestrators to coordinate the composite scaling. The composite scaling is as follows:

1. Initially, the *NFVO-C* orchestrator sends a *ScaleNestedNS* (event c_1) operation to orchestrators *NFVO-A* and *NFVO-B* via a multi-cast message m_1 to scale services s_1 managed by *NFVO-B*, and s_3 managed by *NFVO-A*, respectively. A *ScaleNestedNS* instruction denotes the petition to scale a *nested/composite* network service that is an external dependency.

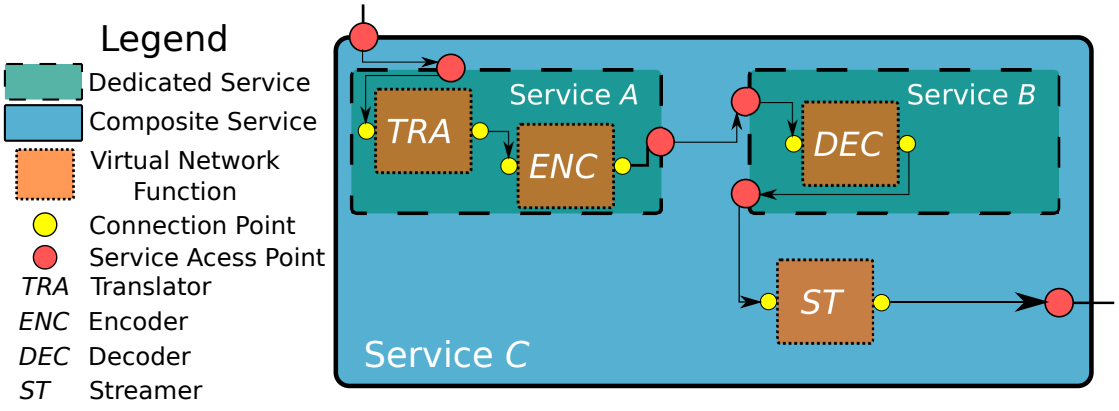

Figure 4.2: Complete *composite* service C with two *external* dependencies (i.e. services A, B) as shared services. Both *external* dependencies have *internal* dependencies as VNFs (i.e. TRA, ENC, DEC, and ST) linked by connection points.

2. The orchestrator *NFVO-B* receives message m_1 with the *ScaleNestedNS* instruction (event a_1). Since the service s_2 is an external dependency of service s_1 managed by *NFVO-A*, the *NFVO-B* sends a grant G_1 so scale service s_2 to *NFVO-A*.

3. Assume that the *ScaleNestedNS* instruction, sent by message m_1, arrives first to *NFVO-A* (event b_1). Since s_1 is an external dependency of service s_3, *NFVO-A* sends a second grant G_2 to *NFVO-B*.

4. The orchestrator *NFVO-B* validates, scales the service s_1 (event a_2), and sends a positive acknowledgment to *NFVO-A*. The orchestrator *NFVO-A* receives the positive reply via message m_2 and triggers a scale event for service s_3 (event b_2).

5. After scaling the service s_3, *NFVO-A* sends an acknowledgment to *NFVO-C* via message m_3. *NFVO-C* stores the positive answer (event c_2).

6. *NFVO-A* gets the first grant G_1 from *NFVO-B* (event b_3); but, it will not scale the service s_2 since the scaling already took place by executing event event b_2 since service s_2 and service s_3 are the same (i.e. $s_3 \equiv s_2$). Thus, *NFVO-A* sends a positive reply to *NFVO-B* via message m_4.

7. *NFVO-B* receives the positive reply from *NFVO-B* (event a_3); however, it will also not scale network service s_1 since it has already scaled it by executing event a_2, and sends a positive reply to *NFVO-C* via message m_5.

8. Finally, *NFVO-C* scales the *composite* service s_0 after receiving two positives replies from *NFVO-A* and *NFVO-B*. Because of the dependent reconfiguration, there were three scale operations (event c_3).

The exchange of messages, as defined by the European Telecommunications Standards Institute (ETSI) NFV standard, suffices to achieve consistent reconfigurations in an ideal scenario. In such a scenario, the network does not lose messages and the orchestrators synchronize via global references. However, orchestrators lose, send, and deliver messages asynchronously and out of order. **Such network properties can lead to inconsistencies during the network services reconfiguring tasks**. For example, Figure 4.4 shows an inconsistency during another possible dependent reconfiguration for the same example. The order of the scale tasks is different as the orchestrator executes grant G_1 before the *ScaleNestedNS* operation. This example shows four scale tasks, where the fourth is redundant, as only three reconfigure the *shared* network service. The extra scale task happens because the asynchronous message delivery leads to an execution that does not satisfy the dependent relations of the *shared* network services. Figures 4.3, 4.4 illustrate a single dependency between a pair of services; however, network services can have multiple dependencies that have the possibility of one or more inconsistencies. An inconsistency increases the costs for the provider; even worse, with a long chain of network services, the cost compounds along all the chains which could

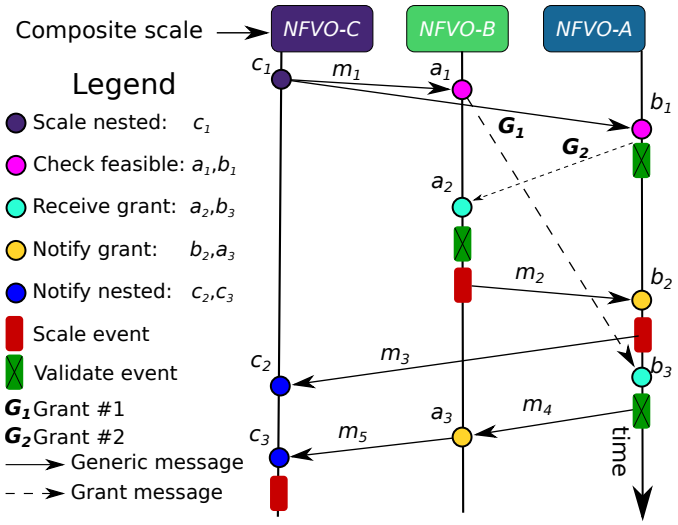

Figure 4.3: The orchestrators consistently scale a *composite* service that is shared among them. In this execution of the *composite* scale operation, only **three scale** operations are done.

violate the service level agreement. Not only does cost increase, but the network services can have partial or total failures. Is necessary to impose an execution order to prevent inconsistencies while doing dependent reconfigurations with shared external dependencies for VNF-based network services. Before introducing our proposed orchestration algorithm, we present the relevant work in the literature and highlight its limitations.

4.3 The state of shared network service reconfiguration

We discuss the relevant work for the dependent reconfiguration task. First, we present single-domain reconfiguration algorithms, focusing on the VNF scaling problem. Then, we present reconfiguration algorithms for multi-domain environments and highlight the drawbacks of the current state of the art solutions. Finally, we briefly describe how our proposed model and algorithm extend the state of the art for VNF-based service reconfiguration in multi-domain environments. Figure 4.5 shows how we organized the related work. Our work is positioned in the colored branch with the bold font for multi-domain VNF-based shared network services.

4.3.1 Reconfiguration of VNF-based network services in single-domain environments

The network service's reconfiguration focuses on three tasks of the life cycle management of network services. The orchestrators execute tasks such as migrating, updating, and scaling VNFs. The problem of migration focuses on the new placement of a VNF while

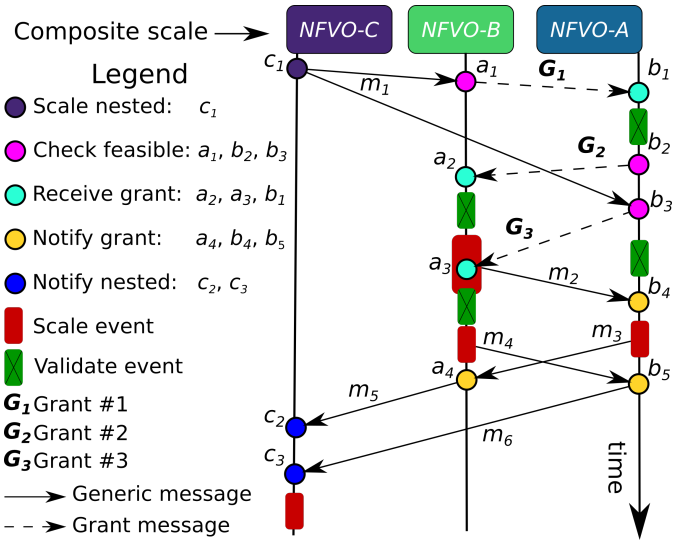

Figure 4.4: Redundant dependent reconfiguration during a scaling-out operation when the *grant* messages arrive in a different order. In this execution, **four scale** operations are done. This entails costs to the service provider while three only are necessary.

optimizing resources such as energy, time, and latency in a single domain [Eramo 2017a, Yang 2018, Wang 2018]. Other works focus on the network update problem that changes

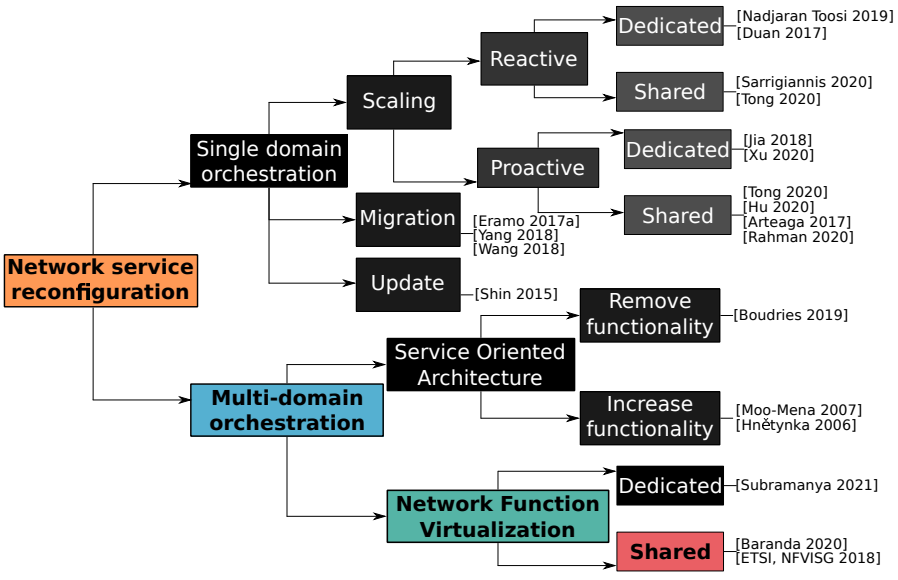

Figure 4.5: Taxonomy for network service reconfiguration. Our work is positioned in the lower branch with Network Function Virtualization multi-domain shared network services.

a VNF descriptor to include more functionalities [Shin 2015]. Scaling with NFV allows operators to resize network services at runtime to handle load surges with performance guarantees [Adamuz-Hinojosa 2018]. In this work, we focus on the VNF scaling as it is the closest related to our work. We classified the related work for scaling shared resources in a single domain as either reactive (i.e. monitor the traffic) or proactive (i.e. predict future traffic).

4.3.1.1 Single-domain reactive works

Most of the works in the literature consider dedicated services that belong to a single network service despite having VNFs in distributed data centers. Auto-scaling orchestration mechanisms have been proposed to minimize the cost of scaling a service while meeting end-to-end delay [Nadjaran Toosi 2019, Duan 2017]. These works propose heuristic algorithms by monitoring algorithms to scale the VNFs. As far as we know, only two works in the literature consider shared resources for a single domain. The first work proposed a latency-aware mechanism [Sarrigiannis 2020]. It offers a scheduling algorithm for the initial placement and reallocation of VNFs. The second work proposed a VNF scaling on-line algorithm that considers all the costs associated with provisioning network resources [Tong 2020]. It achieves an upper-bounded competitive ratio. The major drawback of reactive works is the negative impact of the reconfiguration. Since they only reconfigure services when they capture a problem, the orchestrators must stop service or temporally degrade them while the changes take place. To prevent this, the literature proposes proactive scaling mechanisms. They mitigate the negative impact of reconfiguration.

4.3.1.2 Single-domain proactive works

Proactive works predict future traffic and try to scale network services or VNFs to address these changes. While some works were proposed for dedicated services [Jia 2018, Xu 2020], we focus on shared services. For shared VNF-based network services, the NFV literature considers several works. The first work proposed a traffic model based on Gated Recurrent Units [Tong 2020]. After the prediction, many independent agents explore the network to get optimal placement. Another work proposed a log-linear Poisson auto-regression model to forecast the traffic [Hu 2020]. Based on the model's output, an evolution-based algorithm scales automatically the VNFs. Similarly, an adaptive scaling mechanism based on Q-learning and Gaussian Processes to train a single agent was proposed [Arteaga 2017]. The agent learns the scaling policy despite traffic variations. Another interesting work that allows tenants to refuse scaling was proposed [Rahman 2020]. The method offers a negotiation phase where, based on the predicted traffic and goals for each tenant, under the same domain, the orchestrators scale the VNFs.

All the previous approaches rely on a single administrative domain under a global

orchestrator. The advantages of such deployment are ease of use and simple life cycle management. However, this approach has drawbacks, such as scalability, security, and limited flexibility. Multiple administrative domains want to keep autonomy from a single orchestrator. Decentralized solutions face the shortcomings of the global deployments [Chen 2010].

4.3.2 Reconfiguration of VNF-based network services in multi-domain environments

Decentralized approaches achieve better performance since the orchestrators distribute traffic among participants [Nanda 2004]. Since NFV falls into the definition of IT services at large [Katsalis 2016], service provides can provision VNFs as any other type of service. Service-Oriented Architecture (SOA) principles (e.g., service abstraction, discoverability, and composability) ensure the viability of an ecosystem of network services. For example, in the NFV paradigm, multi-domain environments become these ecosystems [Yi 2018]. Thus, first, we present the SOA reconfiguration solutions. Then, we describe why these solutions do not fully align with the NFV paradigm. Finally, we describe the NFV reconfiguration solutions for VNF-based network services in multi-domain environments.

4.3.2.1 Service-oriented architecture reconfiguration for network services

Before NFV, in the domain of web services, choreographies have been proposed to handle the reconfiguration of a service. A service choreography achieves service composition without centralized control through a protocol via observable events [Leite 2013]. The collaborative protocol, encoded in the choreography, ensures correctness properties such as deadlock prevention, conformance to message specification, and realizability [Kattepur 2013]. Some works propose a coordination protocol to reconfigure services where a shared global state is maintained without a central orchestrator [Kazhamiakin 2006, Salaün 2009]. Works can either remove faulty components by choosing the optimal and correct candidates from a limited pool of options [Boudries 2019], or bring new functionalities on the fly and add them to the existing chaining [Moo-Mena 2007, Hnětynka 2006]. Previous works adapt network services to changes in the environment; however, they exclude consistency issues brought by dependencies among the services that arise while reconfiguring VNF-based network services.

The VNF life-cycle task was inspired by SOA [Yangui 2016]; however, discrepancies between web services and VNFs make SOA solutions inappropriate to address VNF-based network services tasks [el houda Nouar 2021]. For instance, unlike web services, VNFs are not remotely invoked but must be downloaded and executed locally in different administrative domains. The VNFs, and by extension VNF-based network services, include technical details such as the supported technologies, the configuration settings, and their operation management for each task. Moreover, the service choreography in SOA lacks elements present in NFV such as the orchestrator who has a well-defined workflow

for the life cycle of network services [ETSI, NFVISG 2016]. Another challenge present in the NFV context is the heterogeneity of VNFs [Bouras 2017], unlike web services that only consider input and output parameters. Finally, since administrative domains have different capabilities (e.g. CPU, RAM, bandwidth) the reconfiguration operation for VNF-based network services must consider such resources to ensure functional and non-functional requirements [Xu 2020]. Thus, solutions for reconfiguring VNF-based network services with a focus on consistency need to be explored in the NFV context, considering both internal (hidden) and external (observable) events.

4.3.2.2 Network Function Virtualization reconfiguration for VNF-based services

NFV reconfiguration for network services under multi-domain considers federations that share resources by negotiating among many participants [Pham 2019]. Some works consider multiple administrative domains, but the solutions still manage only dedicated services. The literature also contains machine learning approaches, such as deep learning orchestration algorithm to predict traffic and scale VNF instances [Subramanya 2021]. Unlike in single domain orchestration that only considers dedicated network services, multi-domain federations create composite services by sharing resources [ETSI 2020b]. As far as we know, only a few works have considered the scaling of composite network services under multi-domain orchestration. The ETSI NFV standard specifies an algorithm for scaling composite network services [ETSI, NFVISG 2018]. The algorithm proposes a workflow based on grants to coordinate orchestrators. A custom platform was deployed using the ETSI standard to scale composite services [Baranda 2020]. The previous works handle the composite scaling of services in ideal conditions (e.g. ordered messages, no latency in transmission, zero messages lost). However, the non-deterministic conditions of the network and limited information about each orchestration bring new challenges, such as preventing inconsistencies [Vaquero 2019]. Preventing inconsistencies is a desired property when reconfiguring VNF-based network services, as it prevents unwanted effects from rippling across the federation. However, currently, there are no formal models to identify and prevent inconsistencies while reconfiguring composite VNF-based services.

4.3.3 Synthesis

The review of the relevant literature shows that many works consider a centralized orchestrator. Others, non-centralized, consider approaches not completely compatible with NFV. Some works do not consider sharing services. These limitations reduce the applicability of works as providers want to: have autonomy and privacy for their domains, achieve local interoperability, and share resources to extend their market share. The ETSI standard orchestration algorithm addresses some of these limits; however, it considers ideal conditions to reconfigure network services without time constraints. Currently, no grant message exchange pattern has been identified to prevent inconsistencies.

In this chapter, we extend the state of the art by proposing a distributed multi-domain orchestration model that, unlike the state of the art, considers non-deterministic network conditions where services can have multiple dependencies. The model allows to identify an inconsistency pattern for a dependent reconfiguration of VNF-based network services, that is missing today in the literature. To prevent this pattern, we propose a causally consistent orchestration algorithm to prevent inconsistencies while doing dependent reconfiguration.

4.4 Modeling dependent reconfigurations in distributed multi-domain orchestration

In this section, we evaluate the NFV dependent reconfiguration problem from a temporal/event point of view and identify key information to support the consistent dependent reconfiguration of VNF-based network services.

4.4.1 Dependent reconfiguration of network services in distributed multi-domain federations

A dependent reconfiguration happens when an orchestrator requests a reconfiguration for service s and has external dependencies such that the set $\Gamma_s \neq \emptyset$. In this case, multiple grant requests are sent to orchestrators bounded by $|\Gamma_s|$. According to the ETSI standard, the following steps are required for a dependent reconfiguration (we don't consider notifications or answers) [ETSI, NFVISG 2018]:

1. Scale a *composite* network service.
2. Scale nested network services.
3. Request grants to scale external dependencies.
4. Validate requests, check feasibility, and consistency of grants.

We formally define the dependent reconfiguration as follows:

Definition 8 (Dependent reconfiguration)
Let $s \in S$ be a composite service that has at least one external dependency (i.e. $\Gamma_s \neq \emptyset$ where Γ_s is the set of external dependencies) managed by orchestrator $o \in O$. Let event $e \in E$ be of type ScaleCompositeNS executed by orchestrator o. A dependent reconfiguration happens during the composite scaling of e if and only if service s has an external dependency $s' \in \Gamma_s$ such that $\exists s'' \in \Gamma_{s'}, s'' \in \Gamma_s$. In other words, both network services share an external dependency; thus, the joint set $\Gamma_s \cap \Gamma_{s'} \neq \emptyset$.

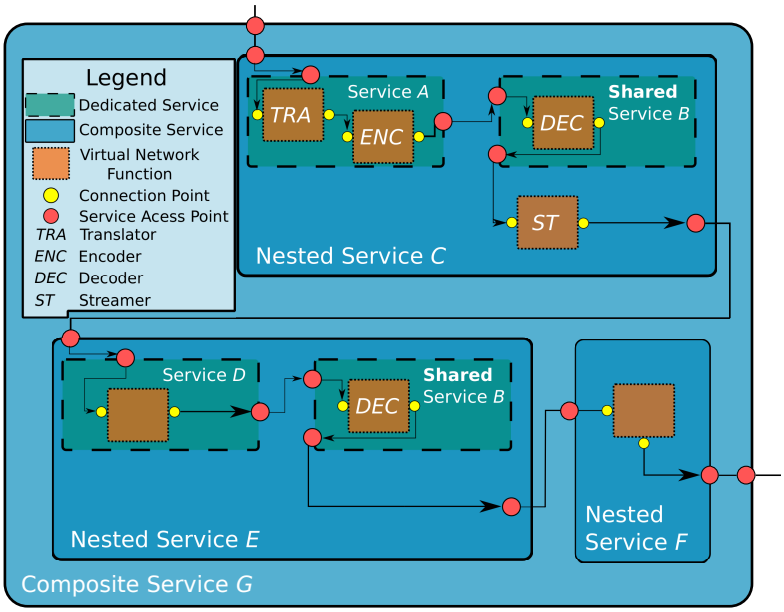

Figure 4.6: Example of the conditions for a dependent reconfiguration. The *composite* service G has two *nested* services that share service B as an *external* dependency. To reconfigure this *shared* service, the orchestrators must coordinate by grants.

We illustrate an example of the conditions required for a dependent reconfiguration as described in Definition 8 for a *composite* service by extending the example shown in Figure 4.2. Figure 4.6 shows an example of the *composite* service G. This service has three nested services in C, E, and F, respectively. Each nested service has its own internal and external dependencies, as shown by the differences between services A and B. Since the two nested services C and E share an external dependency (i.e. *shared* service), they will trigger a dependent reconfiguration in case one orchestrator reconfigures the *shared* service B. The orchestrators exchange grants by executing events in the set $E_{external}$ (i.e. *SdNSLCMGrant* and *DlvNSLCMGrant*). With dependent reconfiguration executed during the scaling for *shared* network services in multi-domain federations, the delivery of *DlvNSLCMGrant* events must be respected to consistently execute the reconfiguration.

4.4.2 Modeling the dependent inconsistency reconfiguration of VNF-based network services from a temporal perspective

Consider the diagram of Figure 4.7. It shows the relevant events during a dependent reconfiguration (see Definition 8) according to the ETSI standard. In this scenario, the federation considers four orchestrators. According to our model (see subsection 2.5), the set of processors for this scenario is $O = \{o_{-1}, o_0, o_1, o_2\}$. The update begins when orchestrator o_{-1} sends a message m_0 to o_0 where $m_0 = (o_{-1}, o_0, e_{x0} = \{(s_0, data)\})$ and $s_0 \sim o_0$ and e_{x0} is a *ScaleCompositeNS*, where $\sim$ is the *is-managed by* relation (see Definition 1,

Section 2.4). The orchestrator checks the validity and feasibility on event e_{00}; this entails validating the parameters sent, the authority of the sender, and checking the feasibility for the VNF manager to scale all the VNFs. In the case the service has external dependencies, the orchestrator sends a multi-cast message $m_1 = (o_0, (o_1, o_2), e_{01} = \{((s_1, s_2), data)\})$ where e_{01} is of type *Scale nested*. If an orchestrator receives a *Scale nested* event, it will also validate the request, as shown in event e_{11}. If the service has an external event, it will send a *grant* through a message as shown in m_2 and will wait for a positive acknowledgment before scaling takes place; if there are no external dependencies, a scaling will take place. Scaling entails checking the validity and feasibility of the VNFs by the VNF Manager, instantiating or removing resources by the virtual infrastructure manager, and finally changing connections as shown in events $e_{12}, e_{13}, e_{23}, e_{24}$. In the event of receiving a *grant*, the orchestrator checks the consistency of all network services affected by the grant, as shown in event e_{21}, and sends a notification through message m_3. If network services are affected, the orchestrator can request a *scale* composite instruction and begin another dependent reconfiguration.

According to the ETSI standard, the execution of events shown in Figure 4.7 is valid; however, it introduces an inconsistency for dependent network services being scaled. The inconsistency is created because service s_1 managed by orchestrator o_1 has outdated information after the scaling operation of service s' triggered by event e_{23} if s' is an external dependency of s_1. More precisely, the inconsistency is brought by the execution of e_{20} before e_{22}.

One way to see this out-of-order execution is the delay brought by asynchronous communication and lack of global references; in turn, this creates a non-deterministic execution of reconfiguration operations. This is the temporal perspective encoded in the message sent by orchestrators to signal the relevant steps of dependent reconfiguration. Upon analyzing the communication diagram corresponding to the execution diagram shown in Figure 4.7, we can observe the inconsistency is created when the transmission time interval of m_1 is greater than the transmission of time interval m_2 plus the message forwarding time of all the *grant* of dependencies. We generalize and formalize the inconsistency pattern from a temporal perspective in the Definition 9.

Definition 9 (Temporally inconsistent dependent reconfiguration)
Let s be a composite service managed by orchestrator o such that $\Gamma_s \neq \emptyset$. Let $m = \{o, \Omega_s, ScaleNestedNS(\Gamma_s)\}$ be a message to scale the nested external dependencies of s, where Ω_s is the set of orchestrators who manage the external dependencies of service s. Let $m' = \{o, \Omega_s/o', ScaleNestedNS(\Gamma_s/s')\}$ be a message to scale all external network services except a single service $s' \in \Gamma_s$ managed by $o' \in \Omega_s$. Let $m'' = \{o', \Omega_{s'}$, $SdNSLCMGrant(\Gamma_{s'})\}$ be a message to grant the operation of service s'. Let $T(x)$ be a function that measures the time taken to send, receive, and deliver a message $x \in M$. A temporal inconsistency during dependent reconfiguration is created if:

$$T(m') > T(m) + T(m'') \mid \Gamma_s \cap \Gamma_{s'} \neq \emptyset, s' \in \Gamma_s$$

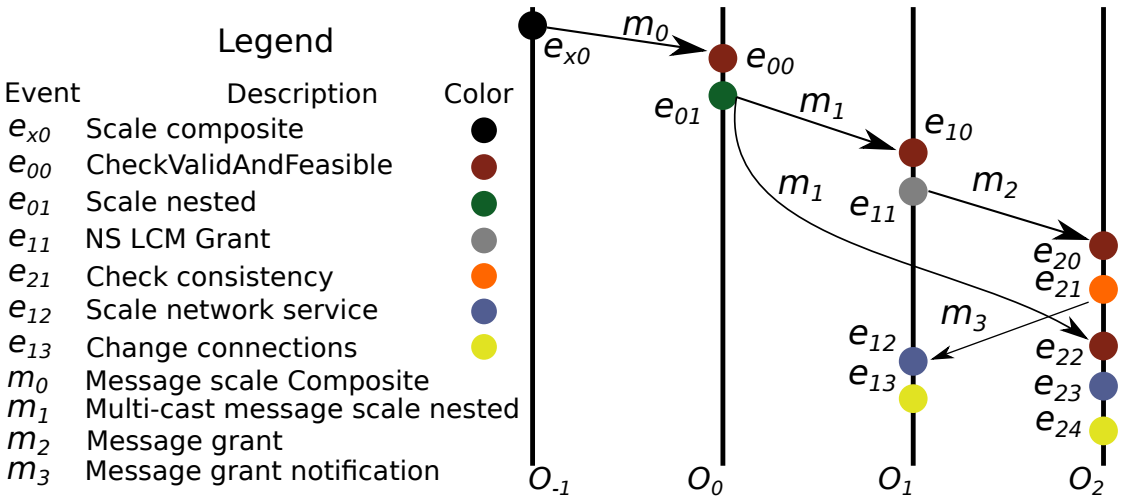

Figure 4.7: Inconsistency during dependent reconfiguration for a *composite* service. All events concerning acknowledgments and notifications are hidden and abstracted. Messages arrive arbitrarily and events can be executed out of order.

A solution to the problem of inconsistent, dependent reconfiguration posed by Definition 9 is to establish common temporal references for all orchestrators and perform the execution of operations according to this reference. However, as discussed in Section 2.3.4 such a solution is unpractical since each orchestrator has limited knowledge. Even clock synchronization algorithms such as the Network Time Protocol [Mills 1991] make it difficult to synchronize clocks across all the network entities.

4.4.3 Modeling the dependent inconsistency reconfiguration of VNF-based network services from an event perspective

In this section, we discuss the inconsistency during the dependent reconfiguration (see Definition 8) of network services from an event perspective. The focus centers on the relevant events during the scaling operation. Similarly to Definition 9 we use the execution flow presented in Figure 4.7 as an inconsistent example. We define the conditions of inconsistency as follows:

Definition 10 (Event related inconsistency of dependent reconfiguration) *Let s_1, s_2 be two network services managed by o, o' respectively. Let $\Gamma_{s_1} \neq \emptyset$, $\Gamma_{s_2} \neq \emptyset$ be the sets of external services' dependencies of s_1, s_2 respectively. Let $\Omega_{s_1}, \Omega_{s_2}$ be the set of orchestrators that manage the external dependencies of service s_1, s_2 respectively. Let e_{sc} be a **ScaleComposite** event of network service s_1 requested by an orchestrator $\hat{o} \notin \Omega_{s_1} \cup \Omega_{s_2}$. Let e_{sn} be a **ScaleNested** event of network services Γ_{s_1} such that $e_{sc} \rightarrow e_{sn}$ and the scale instruction is sent to Ω_{s_1} using the message m. Let e be the execution of the **ScaleNested** e_{sn} instruction at an orchestrator $o \in \Omega_{s_1}$. Let e_g be a **Grant** of*

network services Γ_{s_2} such that $e_{sn} \rightarrow e_g$ and the grant instruction is sent to Ω_{s_2} using the message m'. Let e' be the execution of the ***Grant*** *e_g at an orchestrator $o \in \Omega_{s_2}$. There is an inconsistency if the following conditions hold:*

1. $\exists(\bar{s}, \hat{s})$, $\bar{s} = \hat{s} \in \Gamma_{s_1} \cap \Gamma_{s_2}$ *such that* $\bar{s} \in \Gamma_{s_1}, \hat{s} \in \Gamma_{s_2}$, *and*
2. $e' \rightarrow e$

Figure 4.8 shows a graphic representation of Definition 10. In Figure 4.8 (I) we observe the composite scaling of service s_1 with event e_{sc}, given that this service has external dependencies it sends a scale nested message to all orchestrators that manage its dependencies by e_{sn}. Figure 4.8 (II) shows the delivery of the nested message at orchestrator o'. Since the external dependency s_2 also has dependencies, it sends a grant message to all the managers of its dependencies with event e_g. Figures 4.8 (III,IV) show the case when both s_1 and s_2 have a common external dependencies. In Figure 4.8 (III) the reconfiguration is consistent in the purple rectangle as the scale precedes the grant instruction. Figure 4.8 (IV) shows an inconsistent, dependent reconfiguration as the grant operation precedes the scale instruction. Even if some reconfigurations are consistent, a single inconsistent one suffices to bring the whole service down. More precisely, inconsistencies bring both partial and total failures for network services, reflected in a greater cost for the providers.

A key property for any reconfiguration is to be consistent and stop the conditions of Definition 10. To prevent them at least one condition from Definition 10 must remain unsatisfied. A federated environment makes preventing Condition 1 challenging as the core of these environments is sharing resources; plus, is difficult to ensure because of the limited information each orchestrator has. Thus, **the goal is to prevent Condition 2 from happening; that is, a nested scaling should always precede a grant operation**. Our proposed algorithm identifies and prevents the second condition from happening via causal consistency.

4.4.4 Consistent algorithm for dependent reconfigurations

Based on the formalization presented in Section 4.4, we show how the presented algorithm allows us to capture the causality and avoid inconsistency of network services during dependent reconfigurations, such as scaling in NFV. First, we present an overview of the algorithm in Subsection 4.4.4.1. Then, a simplified workflow of the algorithm is shown in Section 4.4.4.2. The rest of functions are detailed in Sections 4.4.4.4, 4.4.4.5, 4.4.4.6, 4.4.4.7, and 4.4.4.8, respectively.

4.4.4.1 Algorithm overview

Our algorithm complies with the ETSI standard procedure to provision network services (i.e. we consider the same definitions for VNFs, services, messages, events). However,

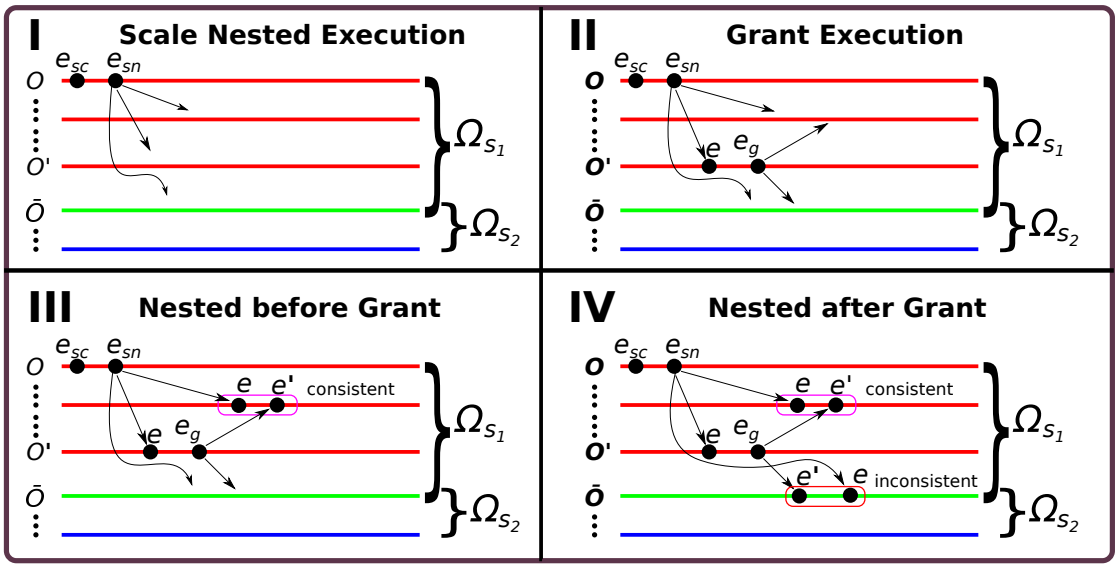

Figure 4.8: Inconsistency during dependent reconfiguration for a *composite* service. An orchestrator executes scale and grant events of a shared external dependency out-of-order.

we propose a new orchestration algorithm to reconfigure *composite*/*shared* VNF-based network services not considered in the standard. The federation's orchestrators execute the algorithm when they send and receive standard grant messages. Such an algorithm bifurcates with multiple function calls because of asynchronous calls while reconfiguring *shared* services. The recursive nature of our solution handles the dependencies of network services by emitting grants as messages among orchestrators. The orchestrators deliver the messages in causal order preventing inconsistencies (see Definition 10). Each orchestrator in the federation executes the following algorithm to support the causal delivery of messages according to the dependencies in the service's descriptors. We summarize it as follows:

- Input: The set of messages presented in Section 2.5 and a vector clock create a tuple that communicates to other orchestrators the changes seen so far from the sender.
- Execution: All reconfiguration messages are asynchronously disseminated and no upper bound on delay is considered. Whenever an orchestrator o sends a *LifeCycleManagement* message m to the orchestrator o', it never blocks and waits for an acknowledgment message of the delivery of m. The clock of each orchestrator is independent of each other such that there is no synchronization with another orchestrator, thus, the execution is fully asynchronous.
- Data Structures: We consider two data structures: orchestrators and network

services.

- Orchestrator:
  - *ID*: The unique identifier of the orchestrator.
  - *vectorClock*: Control information that stores the dependency information between messages being exchanged. The size of the vector is equal to the number of orchestrators in the federation. Each element of the vector clock of orchestrator o is a tuple of the form $(o_{id}, logical\_clock)$ which records the last events seen by o.
  - *pendingOperations*: The external dependencies that wait for the confirmation of the scaling operations.
  - *externalOrchestrators*: The list of all orchestrators in the federation.
  - *internalDependenciesToScale*: The list of all VNFs and network services waiting to be scaled. They are stored to prevent scaling them and then receiving a failure for an external dependency.
- Network Service:
  - *ID*: The service unique identifier.
  - *dependencies*: The list of dependencies of the service. In case of scaling, all must confirm the scaling otherwise the operation is aborted.
  - *orchestratorID*: The identifier of the service's orchestrator.
  - *originalService*: The service who originally sent the scaling operation in case of a dependent reconfiguration.
  - *type*: They type of the component. It could either be a service or VNF.

4.4.4.2 Algorithm details

Figure 4.9 shows the flowchart to scale a VNF-based network service. First, the orchestrator increments its vector clock by one. Then, it adds the scaling to pending operations since the network service could have external dependencies. After, it sends a *grant* to all the orchestrators who manage the service's external dependencies. Finally, to ensure the causal delivery of messages, the orchestrator notifies the others. Algorithm 2 shows the function to scale a VNF-based network service.

Once an orchestrator receives a *grant* request to scale a VNF-based network service, it executes the workflow, as shown in Figure 4.9. First, the orchestrator receives the grant to scale the service. Then, it compares the received clock with its own. If it is greater by more than one value, it stores the *grant* in the list of pending operations. After, if the clock's difference is one, the orchestrator checks if the service to be scaled has any external dependencies, — scaling them in the positive case or just the internal ones. In the end, the orchestrator tries to execute pending operations, thus ensuring events are delivered according to the causal order. Algorithm 3 shows the function to apply the grant to scale a VNF-based shared network service.

4.4.4.3 Request service scale algorithm

We present in detail the algorithms to achieve consistency while handling dependent reconfigurations for distributed multi-domain federations.

4.4.4.4 Grant lifecycle management algorithm

The orchestrator scales a component (e.g. VNF, network service) as follows: First, it validates the internal logic and policies of the request scale while ensuring the scaling will not violate the service's SLA. Then, the orchestrator updates his vector clock, stores the request as a pending operation if the service has external dependencies, and sends the respective grant or scale request to other orchestrators as shown in Function 1; otherwise, all internal dependencies are scaled as shown in Function 2. Finally, the orchestrator notifies all other orchestrators in the federation to enforce the causal delivery of messages. A chain of scaling is created when dependencies of service have external dependencies themselves.

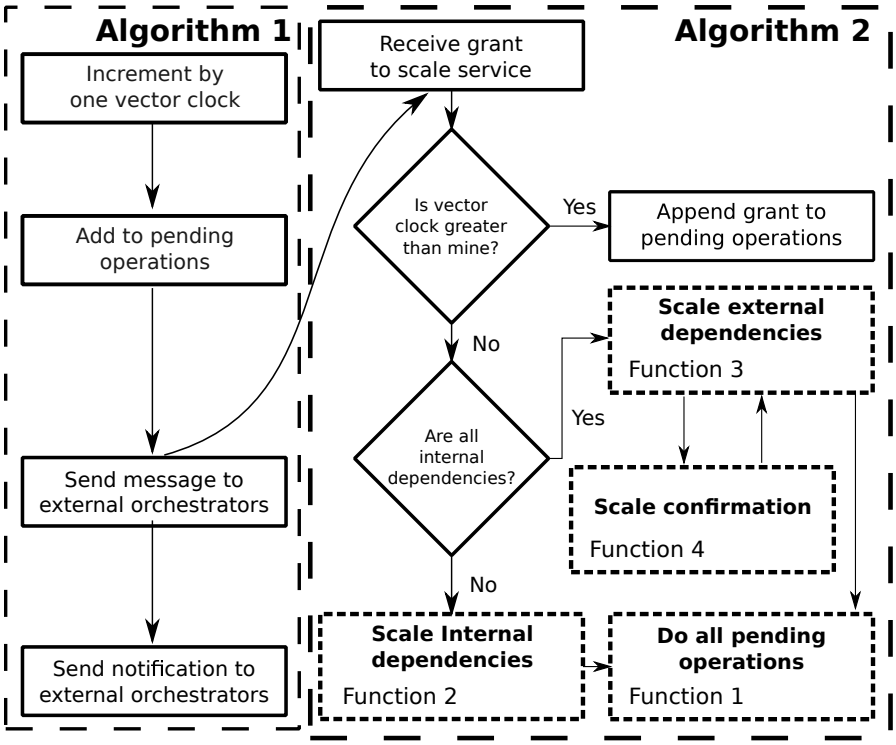

Figure 4.9: Simplified workflow to consistently scale a shared VNF-based network service.

Algorithm 2: Request service scale algorithm. An orchestrator checks the dependencies of a service. In case these are managed by other orchestrators it sends them a *grant* message

```
   Input: Composite network service service
   Input: Orchestrator myOrchestrator
 1 myOrchestrator[vectorClock].increment()     /* Increase the vector clock
    associated to the network service */
 2
 3 myOrchestrator[pendingOperations].append(service)     /* Add the scaling
    to the pending operations since it has not received the grants
    from external dependencies */
 4
 5 myClock ← myOrchestrator[vectorClock] /* Create a clock to be sent by
    a grant request message */
 6
 7 myID ← myOrchestrator[ID]    /* Obtain the unique identifier for the
    service  */
 8
 9 foreach dependency in service[dependencies] do
10 |   newOrch ← dependency[orchestrator]
11 |   newID ← dependency[id]
12 |   send(newOrch,GrantLCM(newID, myClock))     /* Send the message to
   |    the orchestrators who manage external dependencies */
13 |
14 end
15 foreach externalOrchestrator in orchestrator[externalOrchestrators] do
16 |   send(externalOrchestrator, NotificationLCM(myClock,myID))   /* Send
   |    notification to all orchestrators, this ensures the correct
   |    ordering for other orchestrators */
17 |
18 end
```

4.4.4.5 Scale external dependencies function

Collect all dependencies of a given service. If it is a VNF (i.e. an internal dependency) it is stored and appended to a list of pending operations. This, because all external dependencies must accept the dependent reconfiguration, before internal dependencies can be changed. The orchestrator generates a grant message for all the external dependencies. This is why some reconfigurations are dependent of others.

4.4.4.6 Scale internal dependencies function

The orchestrator triggers a reconfiguration operation, in this case a scaling out/in, for all internal dependencies.

4.4.4.7 Scale confirmation function

The dependency sends a *ScaleConfirmation* message to the orchestrator once scaling has finished. Once the message is delivered, the orchestrator executes Function 3 as follows: First, the orchestrator checks if the scaling confirmation relates to a pending operation and waits to receive all external confirmations. Then, after waiting for all internal dependencies scale as this ensures all-or-nothing scaling. Finally, the orchestrator acknowledges the sender of the scaling request by confirming everything went fine. However, if the pending operation is local, only the scaling takes place.

4.4.4.8 Do pending operations function

Function 4 is the most complex of all functions. First, it evaluates if there is at least a single operation that can be executed when the difference of vector clocks of the operation and the current clock is one. If it is the case, it validates the operation by checking if the request has the correct permissions and resources. In case of a valid operation, it checks the dependency type of the service or VNF referenced by the request. In case all dependencies are internals (usually only VNFs) it calls Function 2. For external dependencies Function 1 is called.

4.5 Evaluation

We implemented our proposed algorithm to measure both performance and correctness criteria (i.e. zero inconsistencies while reconfiguring). The following sections comprise the distributed setup (Section 4.5.1), metrics evaluation (Section 4.5.2), experiments (Section 4.5.3), and discussions (Section 4.5.4).

4.5.1 Distributed federation setup

We evaluated our algorithm using Azure's cloud infrastructure. We chose multiple domains from the cloud provider from the following locations: North Europe, West US, South Korea, East US, and the UK. For each domain, we instantiated a virtual machine to host the orchestrator software. All virtual machines have the same configuration: 2 CPUs, 30GB of hard drive, 4GB of RAM, and Linux 18.04-LTS. Each domain has its policies, topology, and is managed by a single orchestrator. Nowadays, multiple open-source orchestrators follow the ETSI standard like Open Source MANO [Israel 2019],

however, none implements the required interfaces to support a federation. Thus, we implemented an orchestration platform in Python. The source code can be found in [1].

Network services are created by chaining VNFs, internal, and external services. The VNFs considered for the network services are part of content delivery networks functions that process video such as encoders. The specification for the service is stored in JSON files that contain parameters for network services and their corresponding VNFs. Table 4.2 shows the parameters used for all experiments. We created multiple experiments by randomly assigning VNFs, network services, and their constraints to the orchestrators in each of our domains. We also generated a random set of scale requests to test our algorithm. To simulate asynchrony in the network, all messages have random waiting times.

4.5.2 Metrics to evaluate

To measure the benefits and trade-offs of our algorithm we evaluated the following metrics:

- Inconsistencies: The number of differences when two or more orchestrators have different configurations for a shared network service. They should be minimized or prevented while reconfiguring dependent network services.
- Message overhead: The number of messages sent to coordinate orchestrators. Messages induce a waiting time until the appropriate one is received.
- Reconfiguration time: The time taken to achieve the reconfiguration. Ideally, this would be short; otherwise, the user of the network service suffers an interruption.
- Memory overhead: The amount of information stored in memory to coordinate the orchestrators.

Ideally, an orchestration algorithm would have zero inconsistencies while achieving a fast reconfiguration with few messages to coordinate the orchestrators. However, preventing inconsistencies has an associated cost. Next, we measure the performance of our algorithm compared to the ETSI standard [ETSI, NFVISG 2018] as it is the closest work in the literature as shown in Figure 4.5 (see Section 4.3).

4.5.3 Experiments

We evaluated the performance of our algorithm and the current ETSI standard [ETSI 2018a] for *composite/shared* services. This work is the closest work to ours as it includes: (i) multiple administrative domains, (ii) *composite* VNF-based network services, and (iii) dependent reconfiguration. To support inconsistency detection, we

[1] https://doi.org/10.5281/zenodo.3989957

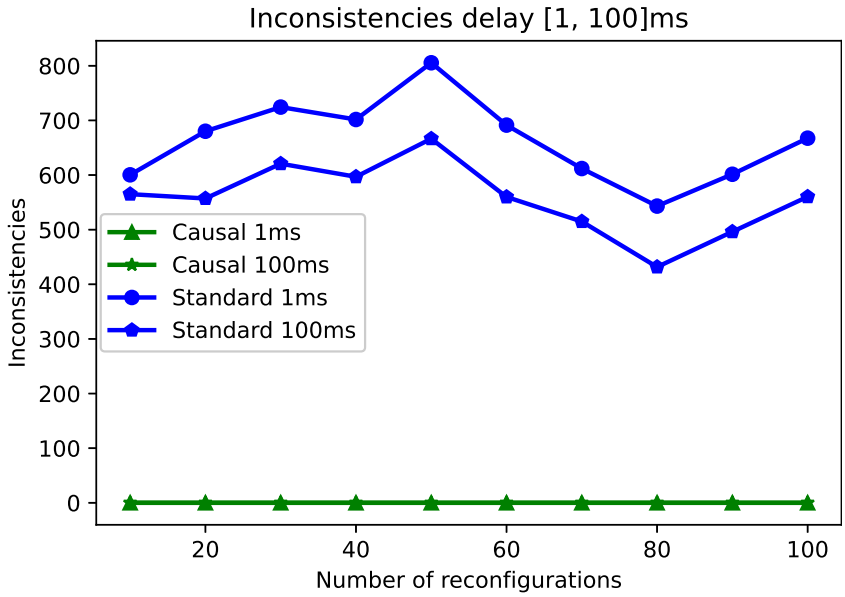

Figure 4.10: Inconsistencies per number of reconfigurations, lower is better. Our algorithm obtains zero inconsistencies, unlike the standard.

implemented and added vector clocks [Fidge 1988] to the ETSI standard. The original implementation does not contemplate this. We consider two experiments. For the first one, we deploy multiple *dedicated* and *composite* VNF-based network services and reconfigure them. For the second one, we deploy only a single service reconfiguration and measure the effects of dependencies for each metric considered. Both experiments had a threshold of 60 seconds. If the scaling of a network service takes longer than the threshold, we consider it invalid.

4.5.3.1 Experiment 1. Single service reconfiguration

This experiment aims to measure the overhead of our proposed algorithm compared to the ETSI standard for a dependent reconfiguration of network services. In this scenario, each reconfiguration is done one at a time. We tested over 5500 random reconfigurations for all the services we generated, as shown in Table 4.2. We consider intervals with increments of ten, up to 100, to measure the performance of both algorithms in terms of the metrics considered (see Section 4.5.2). A time-out of 60 seconds was set. If the time to reconfigure exceeded the time-out, we consider the reconfiguration as invalid. Figures 4.10, 4.11, 4.12, 4.13, 4.14 show the results.

4.5.3.2 Experiment 2. Performance with respect to the number of dependencies

This experiment aims to measure how the overhead metrics increase concerning the total external dependencies. In this experiment, we count all of them, not only the immediate dependency. For example, if a service has 3 external dependencies and one of these external dependencies has also 2 external dependencies, the service will have 3 immediate dependencies but 5. Thus, the experiment reveals more about the relationship between the external dependencies and the solution overhead. Table 4.3 shows the parameters for the experiment. Similarly to Experiment 2 (see Section 4.5.3.1) a time-out of 60 seconds was set for invalid services. Figures 4.15, 4.16, 4.18, and 4.17 show the results for each metric considered.

4.5.4 Discussion

Our algorithm gets the expected performance of zero inconsistencies; unlike the ETSI standard. Experiments validate the algorithm and show that the current standard gets inconsistencies while reconfiguring the network services. Even if network services have few external dependencies, the standard still has inconsistencies, as shown in Figure 4.15. It can be seen how, despite reconfiguring a single *composite* service, when the number of dependencies is greater than two, the standard already has inconsistencies.

For multiple reconfigurations, our proposed algorithm prevents inconsistencies, unlike the standard as shown in Figure 4.10. For the standard, the number of inconsistencies

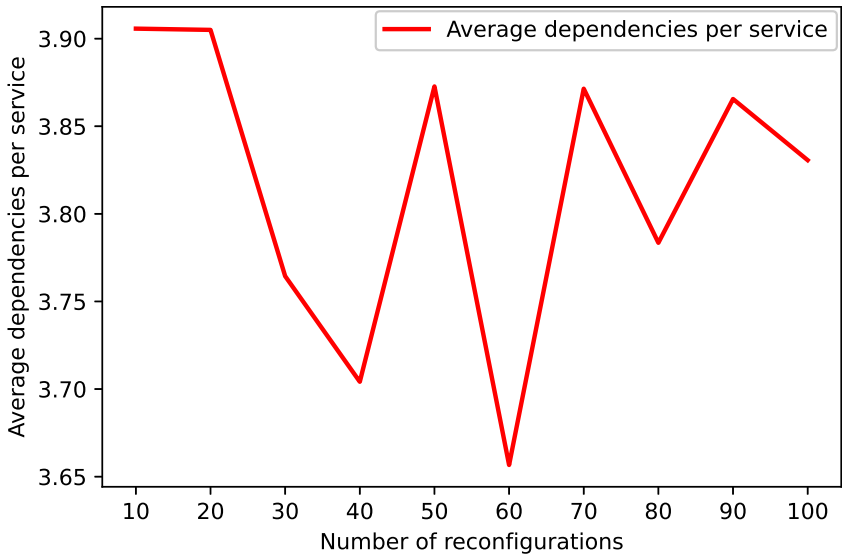

Figure 4.11: Average dependencies per VNF-based network service. On average, services have between 3,4 dependencies.

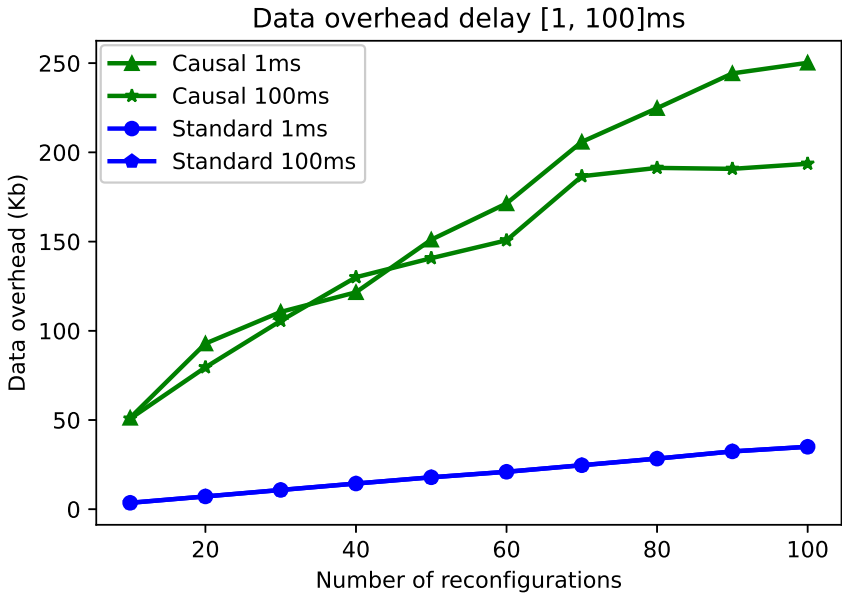

Figure 4.12: Memory overhead per number of reconfigurations, lower is better. Our proposed algorithm has a greater cost.

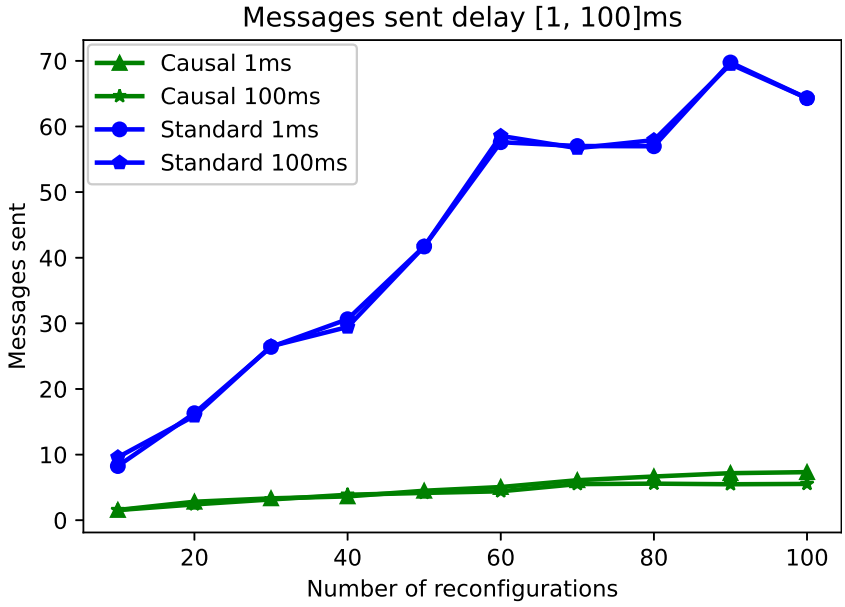

Figure 4.13: Sent messages to request grants and notify orchestrators, lower is better. Our proposed algorithm obtains better performance as it prevents inconsistencies that create redundant messages.

changes over the number of reconfigurations. This variation happens as the services,

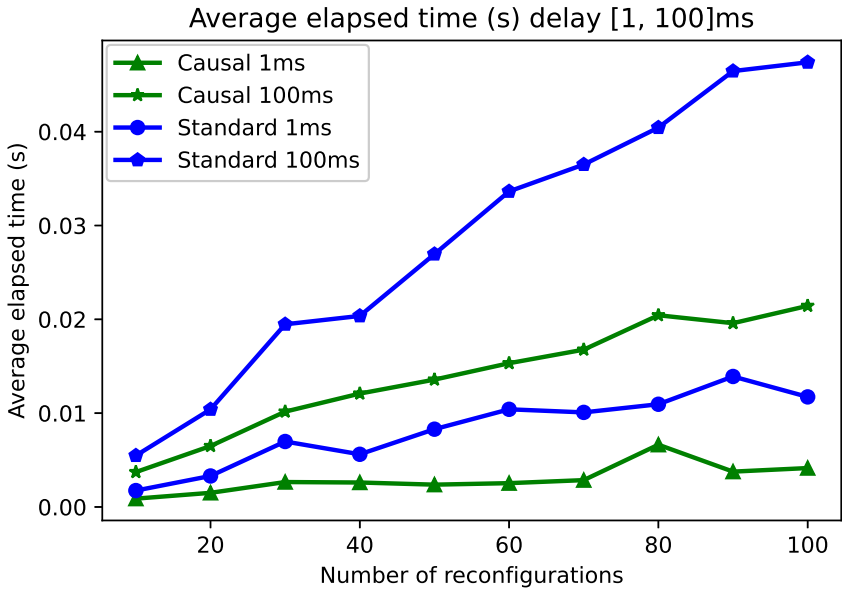

Figure 4.14: Time spent reconfiguring the VNF-based services, lower is better. Our proposed solution reconfigures faster than the standard. This is in part because of the number of messages the standard must process, unlike ours.

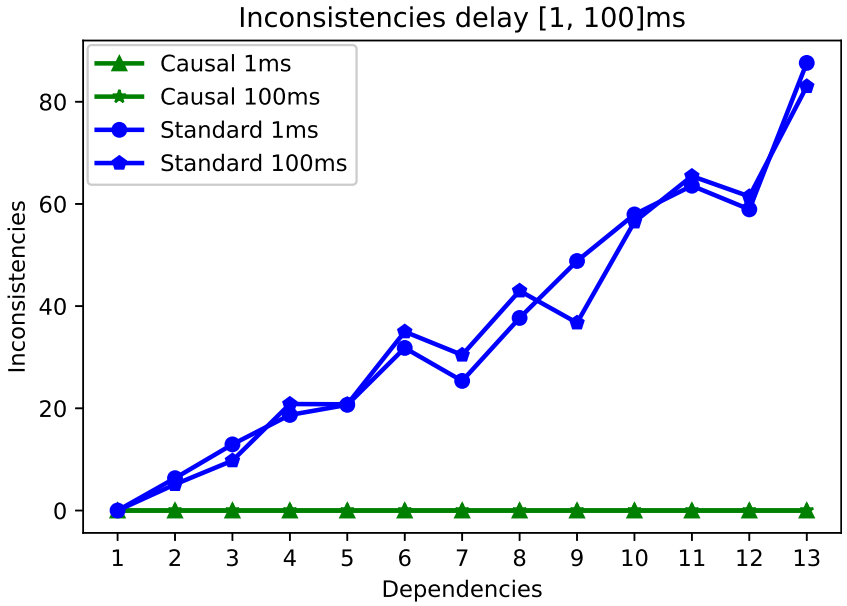

Figure 4.15: Inconsistencies per dependencies, lower is better. Our algorithm obtains zero inconsistencies. For the standard, the inconsistencies increase with more dependencies.

for each step, were created and selected at random using the range of parameters (see Table 4.2). At the first sight, it appears there is no relation between the number of dependencies and the inconsistencies as shown in Figure 4.11. However, the more detailed analysis of the second experiment, where we fixed the dependencies instead of having services with different dependencies, reveals that there is a relation between them as shown in Figure 4.15 where the number of inconsistencies grows as a function of the number of dependencies. Moreover, since we considered both *dedicated* and *composite* VNF-based network services, it is likely that for larger experiments a higher number of *dedicated* were chosen. Thus, we see the downtrend between steps 60-80 in Figure 4.10. Nevertheless, our algorithm prevents inconsistencies irrespective of the number of reconfigurations, unlike the ETSI standard.

Preventing such inconsistencies comes with a cost associated with it. First, we evaluated the complexity of our proposed algorithm in terms of the number of dependencies n and orchestrators m. Then, we measure the performance of our proposed algorithm and compared it to the ETSI standard.

The time complexity of our proposed algorithm is $\mathcal{O}(n^2)$ where n is the number of dependencies. The space complexity is $\mathcal{O}(m)$, where m is the number of orchestrators. It is linear since our algorithm keeps track of the affected orchestrators using vector clocks and stores out-of-order instructions as pending operations for each orchestrator. Our algorithm has greater time complexity than the ETSI standard [ETSI, NFVISG 2018] who has a time complexity of $\mathcal{O}(n)$ and a space complexity of $\mathcal{O}(m)$ in ideal conditions (i.e. one reconfiguration at a time, deterministic network). However, the added cost

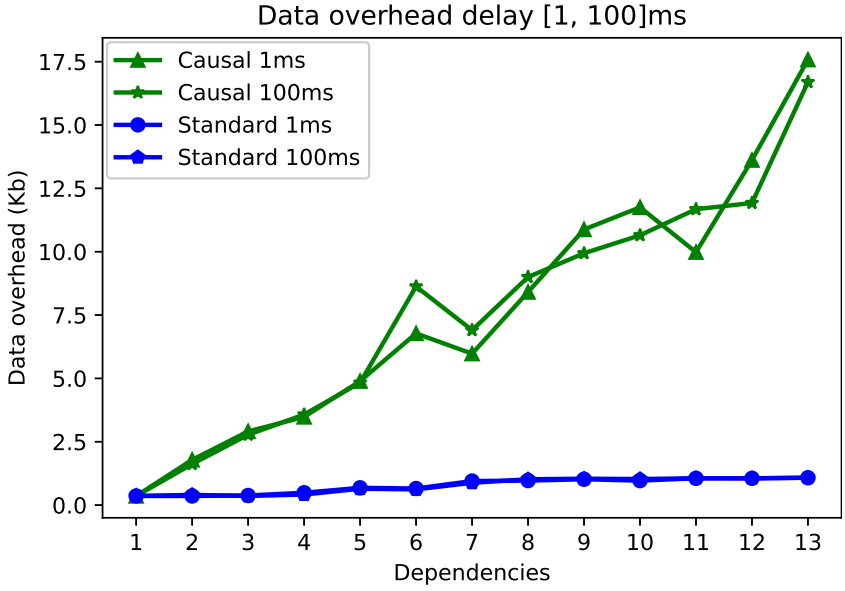

Figure 4.16: Memory overhead per dependencies, lower is better. For our algorithm the overhead increases dependencies. For the standard, the growth is below ours.

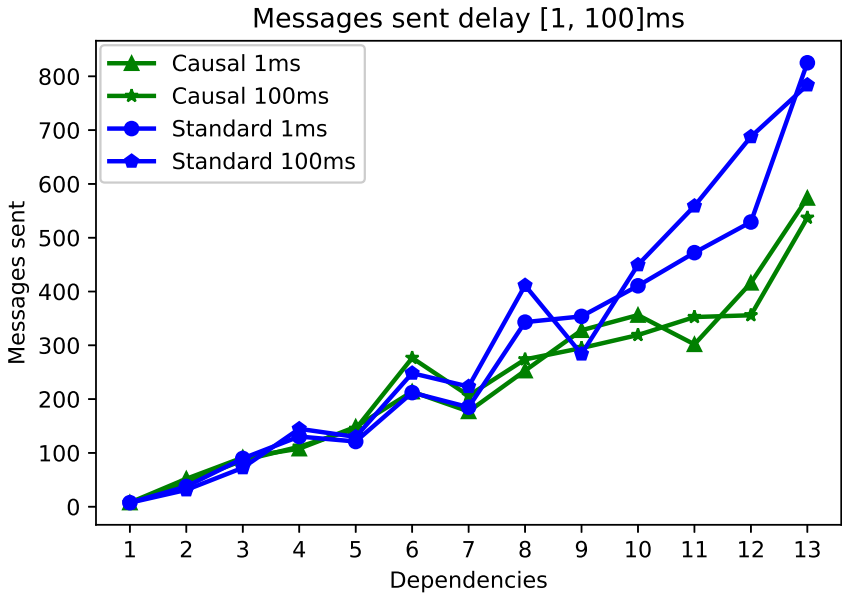

Figure 4.17: Message overhead increases as a function of dependencies, lower is better. For services with a higher number of dependencies, the standard behaves worst due to inconsistencies.

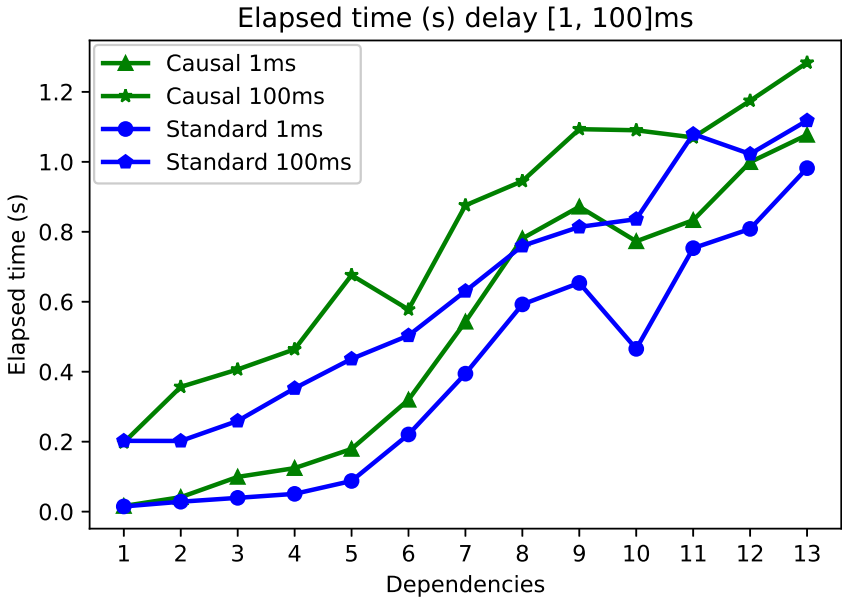

Figure 4.18: Time for a dependent reconfiguration increases with more external dependencies, lower is better.

of our algorithm, in terms of time and space, prevents inconsistencies for dependent reconfigurations.

Performance-wise, we see how our algorithm requires storing more information to coordinate the orchestrators compared to the ETSI standard as shown in Figures 4.12, 4.16. Delay affects the amount of information stored, as shown by the gap between the two lines of our proposed algorithm. For smaller waiting times, more messages arrive out of order and the orchestrators must store causal information to deliver them in the correct order to prevent the inconsistency pattern identified of Definitions 6,7 (see Sections 4.4.2 and 4.4.3). The ETSI standard is unaffected by the delay as it not keeps any information to coordinate the orchestrators outside the grants. Delay also affects, to a lesser extent, the other metrics when there is over one service reconfiguration. This can be seen when comparing Figures 4.10 and 4.15; in the first one, there are more inconsistencies when the delay is higher, unlike the latter. This would suggest that concurrent updates have a greater impact. However, we discuss this in the coming chapters, as preventing inconsistencies when concurrent updates take place means there must be a way to establish precedence, not currently captured by both considered algorithms.

The inconsistencies increase the number of redundant messages, as shown in Figure 4.13. The standard sends about 7 times more messages than our proposed algorithm when multiple services as considered. For a single service reconfiguration, this factor is only 2, as shown in Figure 4.17. Moreover, it can be seen that, for services with few dependencies, our proposed algorithm sends almost the same amount of messages. For services with over 9 dependencies, the standard sends more redundant messages because of inconsistencies. The amount of messages sent by both algorithms reflects on the time spent on the reconfiguration.

Based on the complexity analysis of our algorithm, we expect the time for reconfiguration of our algorithm to be greater compared to the standard. However, Figures 4.14 and 4.18 show that standard behaves worst as it takes about double the time compared to our proposed algorithm. This could be explained by the number of messages that need to be processed by the standard compared to our proposed algorithm. As previously mentioned, one effect of inconsistencies is that orchestrators send more messages to reconfigure a network service. This can be seen by analyzing Figures 4.15, 4.17 and 4.18. With one dependency, the standard has no inconsistencies; consequently, the messages sent are the same as our proposed algorithm. The time is also the same. As the number of inconsistencies becomes greater, the disparity between our algorithm and the standard is also greater. Our algorithm, by preventing inconsistencies, reduces the time it takes for a reconfiguration. For ideal conditions (i.e. deterministic network conditions, one reconfiguration at a time), the ETSI standard would reconfigure faster than our proposed algorithm.

Our proposed algorithm prevents the inconsistency pattern for dependent reconfiguration by ordering and executing grants in the correct order; unlike the standard.

Nevertheless, our algorithm has limitations. First, we assume a known set of orchestrators. This means that our algorithm only works in cooperative environments where the providers will share some information to coordinate with other orchestrators. Second, currently our algorithm stores causality information for both *dedicated* and *composite/shared* services. To reduce some of the redundant information, we consider future work optimizations, such as detecting immediate causal relations to store less information and reduce the number of messages sent. Third, our algorithm supports only sequential reconfiguration, as the HBR relation does not capture concurrent events. In NFV, is possible to have concurrent reconfigurations for shared services. We will discuss in coming chapters the management of such type of reconfiguration. Our proposed model and algorithm can apply to other lifecycle management operations of shared VNF-based network services such as healing, terminating, and monitoring. Moreover, since we followed many of the ETSI standard guidelines to implement the orchestration algorithm, our work can be integrated to open source solutions that are ETSI compliant.

4.6 Lessons learned and perspectives

This chapter focused on the VNF-based network service reconfiguration. It describes how, by coordinating the orchestrators, it is possible to share complete end-to-end services. Sharing such services creates dependencies among them. To ensure consistent behavior of services while reconfiguring network services, the orchestrators not only need to coordinate, but also identify such dependencies.

We prevent inconsistencies while reconfiguring shared VNF-based network services by removing one condition that triggers such inconsistencies. We identify an inconsistent pattern by analyzing the grant order. As a result, we found two conditions that trigger inconsistencies. Both need to happen simultaneously; thus, our goal is to prevent one from happening. One condition is prevented by coordinating the orchestrators by establishing a causal order. Such an order ensures a grant execution order. The other condition, in theory, can be prevented without coordinating the orchestrators by prohibiting certain sharing schemes. This is an interesting finding, as coordination hinders performance. Indeed, to achieve coordination, the orchestrators must wait until they receive the correct grant. However, in reality, prohibiting service providers to share resources would hinder the applicability of federations. The point of federations is that service providers have the autonomy to cooperate with others. Thus, we ask if it is possible to achieve a coordination-free orchestration approach for one task of the VNF-based service reconfiguration.

In the next chapter, we will delve into a problem related to scaling VNF-based network services, namely the reconfiguration of a VNF-Forwarding Graph. We will discuss how can orchestrators coordinate to reconfigure consistently the VNF-Forwarding Graph, but highlight some drawbacks of coordinating the orchestrators.

Algorithm 3: Grant lifecycle management algorithm. The orchestrator checks the validity of the grant. If valid, it executes the grant; otherwise, it will store the grant in a buffer.

```
   Input: Composite network service service
   Input: Sender's vector clock senderVC
   Input: Orchestrator myOrchestrator
 1 myClock ← myOrchestrator[vectorClock]
 2 myPendingOperations ← myOrchestrator[pendingOperations]
 3 myID ← myOrchestrator[ID]     /* Obtain all the required information
    for local service to compare it with the value received */
 4
 5 if compare(myClock, senderVC) ≤ 1 then
 6     otherOrchestrator ← service[orchID]       /* If the difference of the
        vectors clocks is equal to one, the grant could be applied */
 7
 8     myClock[otherOrchestrator] += 1
 9     if validateScaling(service) then
10         serviceID ← service[ID]   /* If the grant is valid, then scaling
            takes place.  If the service has a external dependencies
            too, it will start another dependent reconfiguration */
11
12         myPendingOperations.append(serviceID)     /* Add the operation to
            the waiting list as this local service can also have
            external dependencies */
13
14         myOrchestrator[vectorClock][myID] += 1
15         if allDependenciesInternal(service[dependencies]) then
16             scaleAllInternalDependencies(service)     /* Scaling VNFs and
                network services.  See Function 2 from Section 4.4.4.6 */
17
18         end
19         scaleExternalDependencies(service)   /* If the previous VNF is
            affected, then recursively call migration.  See Function 1
            from Section 4.4.4.5 */
20
21     end
22     send(otherOrchestrator, NotifyFail(myClock, myID, serviceID))   /* If the
        new operation is invalid, the orchestrator sends a negative
        reply and the operation will not take effect */
23
24     doPendingOperations() /* See Function 4 from Section 4.4.4.8 */
25
26 end
27 myPendingOperations.append(service, senderVC)
```

Function 1: Scale external dependencies function - scaleExternalDependencies - The orchestrator needs to verify the validity of the reconfiguration operation. If its valid, it checks if all dependencies are local or they are external. In case of external dependencies it will send a grant message and wait until all external dependencies accept the change.

Input: Composite network service *service*
Input: Orchestrator *myOrchestrator*

```
1  myID ← myOrchestrator[ID]    /* Get the required information to sent
     to all external orchestrators */
2
3  serviceID ← service[ID]
4  myClock ← myOrchestrator[vectorClock]
5  inDependencies ← myOrchestrator[internalDependenciesToScale]    /* Obtain
     the list of internal dependencies, if all external orchestrators
     accept, these dependencies will be scaled */
6
7  originalServiceID ← service[originalService]
8  myOrchestrator[pendingOperations][serviceID][originalServiceID] ← list()
     /* Create an empty list to store the replies for the grant
     request */
9
10 foreach dependency in service[dependencies] do
11     dependencyID ← dependency[ID]
12     if dependency[type] == VNF then
13         inDependencies.append(dependency)  /* Add the VNF to the list of
             internal dependencies.  These VNFs belong in the same
             administrative domain; but different location.    */
14
15     end
16     else
17         otherOrchestrator ← dependency[orchestratorID]
18         originalDependencyServiceID ← dependency[originalServiceID] ]
19         send(otherOrchestrator, GrantLCM(dependencyID, myClock,
             originalDependencyServiceID))      /* Send a grant message to an
             external orchestrator so it can evaluate the validity and
             either accept or reject the grant */
20
21     end
22 end
```

Function 2: Scale internal dependencies function - scaleInternalDependencies - The orchestrator sends an instruction to the manager of the Virtual Network Function requesting a scaling.

```
   Input: Composite network service service
   Input: Orchestrator myOrchestrator
1  myID ← myOrchestrator[ID]    /* Get the required information to send
     an instruction to the VNF Manager */
2
3  serviceID ← service[ID]
4  originalService ← service[originalService]      /* Since the scale can come
     from another request, we need to know which orchestrator was the
     first who requested the scale */
5
6  foreach dependency in service[dependencies] do
7  |   send(dependency, Scale(myID, serviceID, originalService ))      /* Send a
   |    scale instruction to the VNF manager of an internal dependency
   |    */
8  |
9  end
```

Function 3: Scale confirmation function - scaleConfirmation - The orchestrator checks if all pending operations for a given service have been accepted by all orchestrators. If it is the case, it will apply the reconfiguration.

```
   Input: VNF Component ID vnfcID
   Input: Composite Network Service service
   Input: Original Orchestrator originalOrchestrator
   Input: My Orchestrator myOrchestrator
 1 currentOperation ← orchestrator[pendingOperations][service[id]     /* Gather
    all the required information for to check if the service will be
    scaled */
 2
 3 pendingOperations ← currentOperation[pendingOperations]
 4 myPendingOperations ← myOrchestrator[pendingOperations]
 5 if pendingOperations.isNotEmpty() then
 6     pendingOperations.remove(vnfcID)  /* Remove the pending operation,
        the orchestrator will check if a logical clock was correctly
        received before applying such instructio */
 7
 8     if mypendingOperations.isNotEmpty() then
 9         myPendingOperations.remove(currentOperation)
10         if myOrchestrator.isExternalDependency(currentOperation) then
11             send(originalOrchestrator, ScaleConfirmation(currentOperation)
                /* Check if the original request is valid, call
                recursively the scale confirmation */
12
13             return
14         end
15         if myPendingOperations.isEmpty() then
16             scaleAllInternalDependencies(currentOperation)      /* If all
                pending operations are accepted, then the internal
                dependencies can be scaled.    */
17
18         end
19     end
20 end
```

Function 4: Do pending operations function - doPendingOperations - The orchestrators verifies all the pending operations to check for a valid reconfiguration. In case one is valid, it will continue to cycle until one is not valid or there are no more pending operations.

```
   Input: Orchestrator myOrchestrator
 1 clockUpdated ← True      /* Gather all variables for the algorithm */
 2
 3 myClock ← myOrchestrator[vectorClock]
 4 myID ← myOrchestrator[ID]
 5 myPendingOperations ← myOrchestrator[pendingOperations]
 6 myIndex ← myOrchestrator[orchestratorID]
 7 otherOrchestrator ← service[orchestratorID]
 8 while clockUpdated do
 9     atLeastOneClockChanged ← False     /* As long as one clock has been
        updated keep iterating.  Since grants arrive at any time, it is
        necessary to check many */
10
11     for operation in orch[pendingOperations] do
12         opClock ← operation[vectorClock]
13         opIndex ← operation[orchestratorID]
14         if compare(myClock, opClock) ≤ 1 then
15             myClock[opIndex] += 1  /* If the received clock is bigger by
                only a single value, then procced to validate scale
                instruction */
16
17             service ← operation[service]
18             serviceID ← service[ID]
19             if validateScaling(service) then
20                 myPendingOperations.append(service[ID])          /* If the
                    orchestrator validates the scaling, then, it will
                    check if all dependencies are external or internal */
21
22                 myClock[myIndex] += 1
23                 if areAllDependenciesInternal
24                 (service[dependencies]) then
25                     scaleInternalDependencies(service)
26                 end
27                 scaleExternalDependencies(service)
28             end
29             send(otherOrchestrator, NotifyFail(myClock, myID, serviceID)) /* If
                the orchestrator did not validates the scaling, it will
                send a fail notification the the other orchestrators */
30
31             myPendingOperations.remove(operation)    /* Remove the pending
                operation since it was a failure */
32
33             atLeastOneClockChanged ← True
34         end
35     end
36     clockUpdated ← atLeastOneClockChanged
37 end
```

Variable	Range
Number of services	3000
Number of reconfigurations	10, 20, $\cdots$, 100
Repetitions per experiments	30
Number of dependencies	1 - 6
VNFs per orchestrator	600
Random delay range	[1 - 100]ms

Table 4.2: Experiment's parameters and their range.

Variable	Range
Number of network services	215
VNF Components per service	1 - 13
VNFs per orchestrators	30
Random Delay	[1, 100]ms

Table 4.3: Parameters for the second experiment.

Chapter 5

The limits of coordinating orchestrators. Rethinking how to consistently reconfigure VNF-Forwarding Graphs. The quest for a coordination-free approach.

In the previous chapter, we detail how, by coordinating orchestrators, the service providers prevent the inconsistent reconfiguration of shared VNF-based services. This coordination prevented one condition required for inconsistency patterns. We identified that the other condition could, in theory, be achieved without coordination. Yet, when orchestrators execute concurrent reconfigurations for the same service in the federation, there is a need to solve the conflict of two operations happening at the same time; if not, two or more orchestrators will diverge leaving the services in a failed state. Thus, as far as we know, reconfiguring shared network services requires coordination.

In this chapter, we analyze the problem of VNF-Forwarding Graph (VNF-FG) reconfiguration. This problem is related to service reconfiguration, such as scaling, migrating, or healing Virtual Network Functions (VNFs). However, there are differences between the two problems as the the VNF-FG reconfiguration involves only updating values; unlike the reconfiguration of services, which requires other tasks, such as migration. In other words, the VNF-FG reconfiguration is a task of the service reconfiguration. We explore this problem to identify the conditions required to solve it. Also, we highlight why coordinating orchestrators might not fully solve the reconfiguration problem. To do so, we go more in depth with the coordination approach and highlight the limitations of the VNF-FG reconfiguration problem in the next sections. Unlike our previous chapters, in this chapter we also consider concurrent reconfigurations. Figure 5.1 shows the current step towards a coordination-free orchestration algorithm. In this chapter, we explore a more relaxed problem related to the end-to-end VNF-based network service reconfiguration. Namely, we highlight the troubles that coordinating orchestrators entails in terms of performance and consistency guarantees.

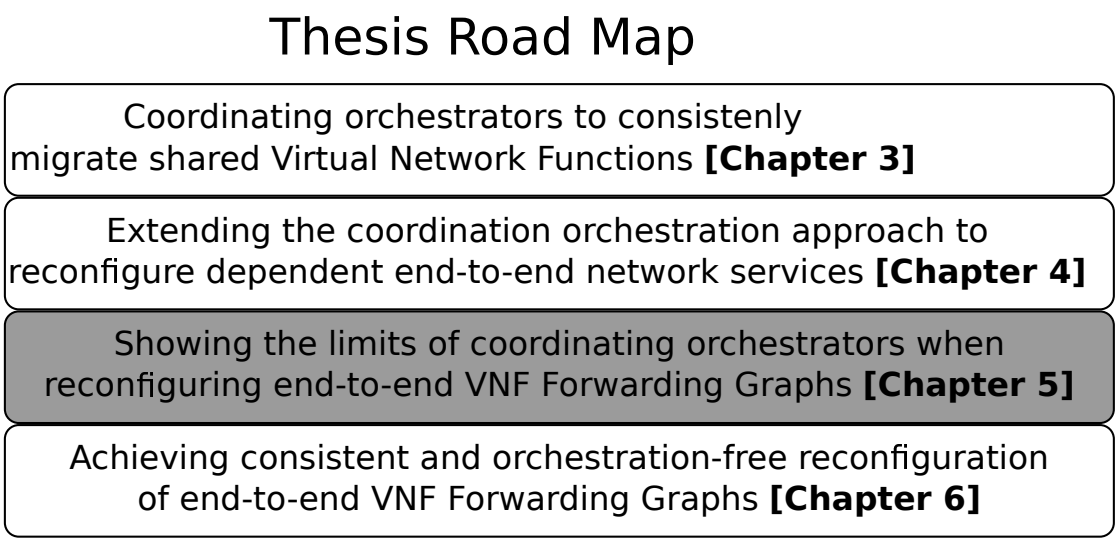

Figure 5.1: Thesis roadmap. In this chapter, we study the problem of consistently reconfiguring the VNF-Forwarding Graphs of end-to-end network services. We consider the case of coordinating orchestrators and show the limitations of such approach.

5.1 Introduction

The VNF-FG defines a logical order of execution for each dependency of a service [ETSI 2014]. Reconfiguration of the VNF-FG is a key task of the lifecycle management of services as it allows for services to answer to unforeseen and dynamic conditions [Zheng 2019]. It comprises changing either connection points, references to VNFs or services; as well as classification and selection rules [ETSI, NFVISG 2014, ETSI, NFVISG 2020]. It is also possible to extend the number of VNFs or services to bring new functionalities to users [Houidi 2020]. Under multi-domain orchestration, the orchestrators send messages to signal changes in service access points. When another orchestrator receives this message, it changes its local copy. In an ideal scenario, this suffices to reconfigure the VNF-FG. Yet, because of the inherent properties of the network (e.g. network delay, redundant messages) and the absence of global references, after reconfiguring VNF-FGs can let them in an inconsistent state; which, in turn, leaves the service with partial or total failures. In the first case, some functionality is lost or does not meet the non-functional requirements established in the service descriptor. Total failure occurs when the whole service stops working and all functionality is lost. Such failures during reconfiguration are translated to a loss in revenue for service providers and could defeat the purpose of reconfiguration [Eramo 2019b]. To prevent this, the orchestrators need to ensure a consistent reconfiguration of the VNF-FGs.

A few works in the literature consider the VNF-FG reconfiguration. Most works only consider onetime provisioning without changes by focusing on either placing or embedding the VNF-FG [Schardong 2021, Anh Quang 2020, Quang 2019a]. To the best of our

knowledge, only a few consider reconfiguration: in [Quang 2019b] the authors propose a model that migrates VNF to get optimal performance. They consider the case of cooperative multi-domain orchestration. They are interested in placing the VNF-FGs that can change. Their aim is to cut costs and maximize performance. Another works propose extending an already deployed VNF-FG to respond to new demands while minimizing the effects of extending the original graph [Houidi 2020, Khebbache 2018]. Despite having reconfiguration as their focus these works do not consider the inconsistencies that can appear while reconfiguring the VNF-FG under multi-domain environments. The asynchronous channel in the network creates inconsistencies while orchestrators reconfigure the VNF-FG. Currently, NFV multi-domain orchestration algorithms do not assume timing constraints exist between two consecutive notifications. This assumption could create inconsistencies, as no global reference exists among the orchestrator replicas [ETSI, NFVISG 2018]. For example, messages can arrive in a different order than they were sent. This leads to inconsistencies in the dependency relations that enforce order among VNFs of the VNF-FG, and in turn, to partial or total failures. The existing solutions proposed only consider eventual consistency, they do not consider the temporal dependencies described in the VNF-FG. Thus, stronger guarantees than eventual consistency are necessary for the VNF-FG reconfiguration.

We focus on the consistent reconfiguration for VNF-FG in multi-domain federations. Our major contribution is identifying and reducing inconsistencies created while the orchestrators reconfigure the shared VNF-FGs by coordinating among themselves through messages without global references. We advance the state of the art for managing the VNF-FG by not only considering local domain information but also addressing the temporal dependencies among shared services under multi-domain federations, as described in the VNF-FG. We propose a causal order among reconfigurations to reduce inconsistencies for both sequential and concurrent ones. After evaluating our proposed algorithm, we discuss the trade-offs between consistency guarantees, performance, and limits of the proposed consistency model in NFV.

The rest of the chapter is as follows: Section 5.2 presents a use case to describe the inconsistent VNF-FG reconfiguration problem under multi-domain orchestration. Section 5.3 details the works in the literature for VNF-FG reconfiguration to position this work regarding the literature. The orchestration algorithm is detailed in Section 5.4. The proposed algorithm is evaluated and compared against the current European Telecommunications Standards Institute (ETSI) compliant standard reconfiguration algorithm in Section 5.5. Finally, Section 5.6 outlines the lessons learned and the perspectives towards coordination-free approaches. We present the notation for this chapter in Table 5.1 (see Section 2.4 for a more detailed description of each variable).

Table 5.1: Notation for this chapter. Some of the variables were defined in the system model (see Section 2.4 for a more detailed description of each variable). Others are defined in this chapter.

Variable	Meaning
$O = \{o_1, o_2, o_3\}$	The set of orchestrators
$G = \{g^1, g^2\}$	The set of VNF-FGs each numbered.
Δ	A VNF-FG Reconfiguration operation
ϕ	The function that computes the state of an system model's entity.
c_g	A classifying rule for a VNF-FG g
ma	A matching attributes, usually a VNF-FG has a list of them.
x_g	A service path for a VNF-FG G
p	A connection point, sually a VNF-FG has a list of them
$\rightarrow$	The happened-before relation
v_i	The logical clock for the i-th orchestrator
q	The buffer of pending operations.

5.2 The incosistent VNF-Forwarding Graph reconfiguration problem in multi-domain federations

Content delivery networks provide value-added services to consumers by replicating content to reduce latency [Pathan 2008]. The orchestrators execute VNFs to place them in the delivery network. We consider a use case like the one provided in [Dieye 2018], where three orchestrators (in France, the USA, and Brazil) provide a video delivery service to end-users with heterogeneous devices (e.g. codecs, resolution, processing power). Whenever an end-user requests a video, the delivery network adds an overlay video that enriches the users' experience. To implement such experiences, a closed federation of trustful providers deploys multiple video processing VNFs, such as encoder, translator, mixers, and decoders. Trustful means providers are willing to share accurate information and not lie about their resources. Mechanisms can be imposed in the federation to prevent fraudulent behavior from providers [Dieye 2020]; however, we leave this for future work.

The close federation enables deploying shared network services. Network service providers ensure the correct order of execution of each VNF of such services by instantiating VNF-FGs [Herrera 2016]. Such VNF-FGs contain both internal and external dependencies of services that correspond to VNFs or other services managed by many orchestrators. Figure 5.2 illustrates both types of dependencies for a service composed of four VNFs: *encoder (ENC)*, *decoder (DEC)*, *translator (TRA)*, and *streamer (ST)*. For the service in domain A (managed by the first orchestrator) the *encoder* and *translator* are local dependencies; while the *decoder* and *streamer* VNFs are external. The VNF-FG is composed of ordered VNFs, as shown in Figure 5.2.

Deploying the VNF-FG for the previous service requires negotiating: resource al-

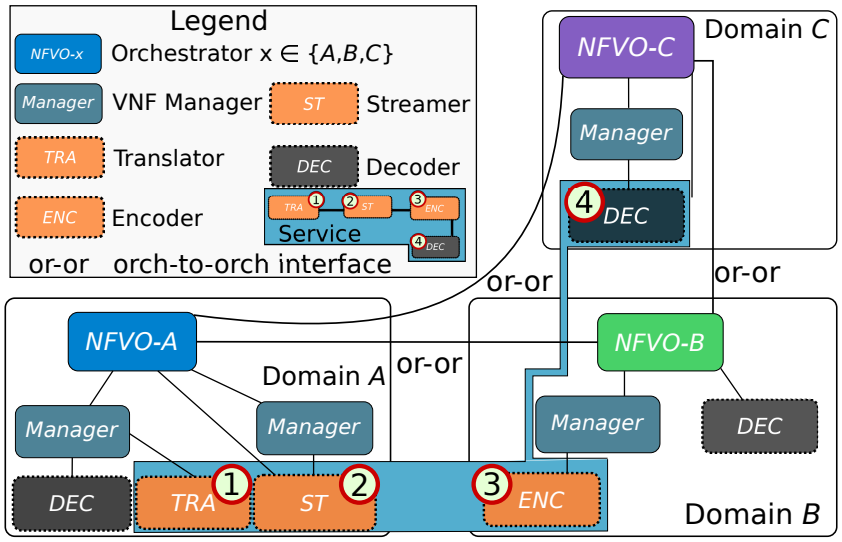

Figure 5.2: Three orchestrators manage a VNF-Forwarding Graph with numbered Virtual Network Functions. The black *decoder* (*DEC*) Virtual Network Functions is shared among all orchestrators.

location, flow steering policies, and updates for the VNF-FG [Rosa 2015]. Once the deployment of the VNF-FGs and services has finished, the orchestrators monitor the services to detect changes in the environment (e.g. a failure of a VNF, overloading of a service). To mitigate the negative effects of such changes, the orchestrators reconfigure VNF-FGs; yet, since the VNF-FGs can be shared among orchestrators, changes applied to one copy should be reflected on other replicas; otherwise, services become faulty. In turn, increasing costs for providers or reducing their revenue. Such changes entail updating matching attributes and connection points [Han 2018]. For example, in Figure 5.2 if the orchestrator in the USA updates the connection point of the *DEC*, all the others should also apply the same change to their respective *DECs*.

The orchestrators coordinate by message passing using different platforms, e.g. Kafka, among themselves to achieve VNF-FG reconfiguration using the or-or standard interface, as shown in Figure 5.2. To prevent faulty services, the orchestrators apply the reconfiguration locally. They also notify other orchestrators, that have the same VNF-FG, so they can apply the same reconfiguration and updates. In an ideal scenario, the orchestrators receive messages in the same order they were sent; however, this is not the case in reality. Messages can get lost, drop, and sent many times, among other faulty network behaviors. Thus, the copies of the VNF-FG may have different values at the end of a reconfiguration. We illustrate the previous behavior in Figure 5.3. Three orchestrators in France, the United States, and Brazil managed copies of a *decoder* VNF-FG. In step I, all copies start with the same value. In step II, the second orchestrator updates a connection point of the *Decoder* and notifies the other orchestrators. The

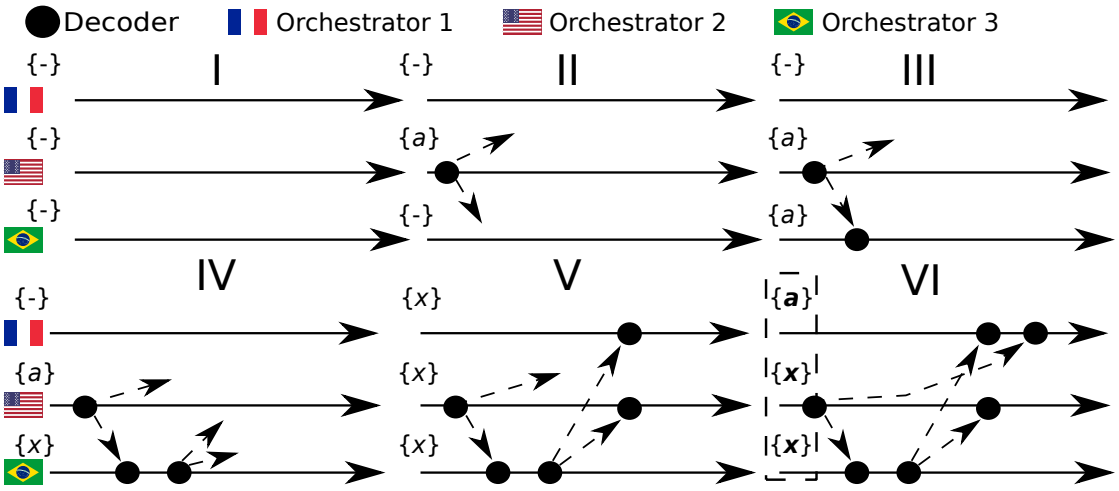

Figure 5.3: VNF-Forwarding Graph reconfiguration. All copies of the *shared* VNF-Forwarding Graph have different values. This is an example of an inconsistent reconfiguration.

third orchestrator receives this notification and reconfigures its local copy (step III). In step IV, the third orchestrator updates its local copy and sends a notification to other orchestrators. In step V, the first orchestrator receives the second notification coming from the third orchestrator and changes its VNF-FG. Finally, in step VI, the first orchestrator receives the first notification and erroneously updates the VNF-FG once more. Since it is impossible for the first orchestrator to discern that the last update is related to the first he received, at the end of the reconfiguration, the values of the copies for the VNF-FG are different; this is represented by the a_z and x_z values for the *z-th* decoder replica (i.e. $a_1 \not\equiv (x_2 \equiv x_3)$). This is an inconsistent reconfiguration; otherwise, all values would be equal. Moreover, the effects of inconsistencies are compounded because if any subsequent reconfiguration of the VNF-FG happens, the whole service will be affected as replicas diverge. To achieve consistency again, a conflict resolution mechanism must be applied to ensure eventual consistency. A manual fix defeats the purpose of automation of NFV; while solving consensus to achieve consistency defeats the performance goal of NFV because of high latency per transaction [Marandi 2014]. Thus, an intermediary consistent model that satisfies the dependencies can be explored for the consistent reconfiguration of the VNF-FG.

5.2.1 Formal problem definition for VNF-Forwarding Graph reconfiguration

A network service under NFV has associated a VNF-FG that defines the logical order of the traffic flow between the VNFs that belong to a service [ETSI 2014]. To achieve this logical order, the VNF-FG has associated a rendered service path and classifying

rules [Schardong 2021]. In an ideal scenario, the service is static; however, because of inherent and dynamic conditions of the environment, such as the number of users, random failures, and extra features, the providers reconfigure services along with their VNF-FGs.

The reconfiguration of a VNF-FG involves changing the list of connection points and matching attributes [ETSI, NFVISG 2014]. This change can be done by updating the values either via changing a connection point or matching attributes and adding/removing more elements to the lists. We present this formally. Let g be a VNF-FG that belongs to the set G. Each g has a pair of classifier rules c_g and rendered service path x_g. The classifier rule c_g has a list of matching attributes $[ma_1, ma_2, \ldots, ma_u]$. And the rendered service path x_g a list of connection points $[p_1, p_2, \ldots, p_v]$.

Each matching attribute $ma \in c_g$ has the protocol, IP, and ports to be visited by incoming traffic. The connection points $p \in x_g$ have both input and egress points. A reconfiguration of the VNF-FG changes multiple values by either a matching attribute or connection points. It is also possible to delete or add classifier rules and rendered service paths; but, we let this feature for future work. We formally define the reconfiguration operation Δ between a pair of VNF-FGs g, g' in Equation 5.1.

$$\begin{aligned} \Delta : g \rightarrow g' = \exists p \in x_g, p' \in x_{g'}, p.id = p'.id \mid \phi(p) \neq \phi(p') \\ \exists ma \in c_g, ma' \in c_{g'}, ma.id = ma'.id \\ \mid \phi(ma) \neq \phi(ma') \end{aligned} \tag{5.1}$$

where ϕ is the state function (see Definition 3, Section 2.4). We name the problem of reconfiguration for a VNF-FG as VNF-FGR. If a network service belongs to a single domain, the reconfiguration is trivial as the orchestrator of such domain manages all the resources and resolves conflicts easily. However, when services are *shared*, the chief interest is that all affected orchestrator replicas have the same view after a reconfiguration. This is the goal of the consistent VNF-FG reconfiguration problem. Next, we describe the state of the art of VNF-FG reconfiguration, and illustrate the limitations and how our proposed algorithm extends the state of the art.

5.3 The state of VNF-Forwarding Graph reconfiguration in the literature

Multiple works in the literature studied the management for lifecycle tasks of the VNF-FG (e.g. embedding, reconfiguring, and composing), the major focus being the embedding [Zheng 2019]. However, these tasks address different requirements and needs. The embedding problem asks how does the orchestrator select the virtual network instances and their connection links [Zheng 2019]. The reconfiguration problem asks how to best update the VNF-FG (e.g. extending it, changing the order of the VNFs) while

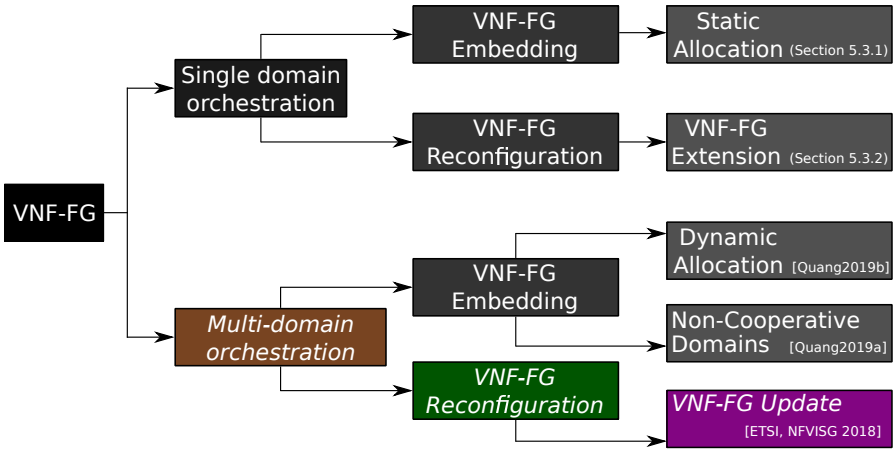

Figure 5.4: Taxonomy with representative works for the VNF-Forwarding Graph. Each branch solves a different problem for the VNF-Forwarding Graph. The closest work to ours is positioned in the colored/italicized path below.

ensuring properties of the service such as availability [Quang 2019b]. We focus on the reconfiguration problem.

We classify reconfiguration works as either single or multi-domain, each one having sub-categories if they are static or dynamic. We describe each category using representative works of each one. Figure 5.4 shows a small taxonomy of representative works for the VNF-FG. Our work is positioned in the colored lower branch along with the ETSI standard.

5.3.1 Single domain static deployment

The first category of works in the single domain considers a unique orchestrator where the VNF-FG stay static. The primary goal of these works is to optimize metrics (e.g. latency [Kim 2018], revenue [Zeng 2016], and energy [Soualah 2018]) while deploying the VNF-FG. Since optimizing the placement is NP-Hard, the authors propose heuristics to solve the problem in larger instances. However, these works rely on a single global orchestrator having full knowledge about the underlying domain, such as topology or network policies. They also assume a fixed and static federation. Both strong assumptions for multi-domain federations limit the applicability for works in this category.

5.3.2 Single domain dynamic deployment

The second category of works tries to remedy the limits of the first category by allowing online changes in the VNF-FG embedding algorithms. These works consider a new VNF placement [Houidi 2020], extending the VNF-FG [Khebbache 2018], and bi-directional chaining [Spinnewyn 2020]. In case of a change in service usage, the central orchestrator

computes the new place to instantiate a VNF or it can update the VNF-FG by changing the VNFs' execution order. Consistency in the VNF-FG management here is not an issue as the global orchestrator has all the required knowledge. Thus, the orchestrators synchronize the updates according to the global orchestrator. Despite this ease of consistency, these works do not scale well for multi-domain federations, as the fundamental assumption of complete knowledge is costly to implement. Moreover, providers prefer to keep their key information private [Saraiva de Sousa 2019].

5.3.3 Multi-domain static deployment

The third category uses a decentralized approach, where multiple orchestrators jointly manage network services. For the third category, a deep reinforcement learning technique was proposed to learn the dynamic behavior of the federation [Anh Quang 2020]. This avoids recalculating from scratch the new placement of VNFs. Another work includes the migration of existing VNFs using an adaptive centralized and decentralized orchestration algorithm to reallocate VNF-FGs [Quang 2019b]. The last work considered in the third category considers a close and competitive environment where orchestrators hide their infrastructure from others [Quang 2019a]. The earlier works focus on the embedding problem, not on the VNF-FG reconfiguration problem.

5.3.4 Synthesis

The closest works to the reconfiguration problem we address in this chapter are the dynamic VNF-FG extension works [Houidi 2020, Khebbache 2018] and the ETSI proposed algorithm to reconfigure network services [ETSI, NFVISG 2018]. In the first works, the orchestrator extends a VNF-FG by adding VNFs and links to respond to new demands. The authors propose a Steiner Tree-based algorithm [Khebbache 2018] and an eigendecomposition algorithm [Houidi 2020] to solve the optimal extended embedding problem. While these works reconfigure the VNF-FG, they does not consider the problem of inconsistency when multiple orchestrators concurrently try to update the VNF-FG. Moreover, both works consider a single global orchestrator. The ETSI NFV standard proposes a reconfiguration algorithm where the orchestrators coordinate through grants, checking the network service consistency [ETSI, NFVISG 2018]. The standard belongs to the third category, which is decentralized.

Unlike the previous works that either consider the initial and static deployment of the VNF-FG or focus on security/optimization under single-domain orchestration, our work focuses on the consistent reconfiguration of VNF-FGs under multi-domain environments (e.g. federations). We consider the temporal dependencies among reconfigurations, unlike the current ETSI-reconfiguration algorithm, which has no timing constraints. We also consider non-deterministic network conditions and only use local knowledge without a global orchestrator or synchronized operations among all orchestrators in the federation.

5.4 Causal consistent VNF-Forwarding Graph reconfiguration

To prevent inconsistencies that reduce the providers' revenue and increase their operational costs, the orchestrators in the federation must coordinate. Thus, when a reconfiguration for a *shared* VNF-FG is applied, the updates made by the reconfiguration must apply to all copies in different orchestrators so that eventually the state of the VNF-FG is the same for all copies. For example, when the orchestrator changes a copy of a shared VNF-FG by updating the order of a connection point in the rendering service path, it will apply to all other copies of the VNF-FG managed by different orchestrators. We formalize this on Equation 5.2.

$$\Delta(g) : \forall g' \in G | g.id = g'.id \longrightarrow \phi(g) = \Diamond\phi(g'). \tag{5.2}$$

where Δ is a reconfiguration operation (Equation 5.1, see Section 5.2.1), ϕ is the state function (Definition 3, see Section 2.5), and $\Diamond$ is a finite but random amount of time. We call the consistent VNF-FG reconfiguration problem as CVNF-FGR.

An inconsistency occurs if, after a reconfiguration Δ on a *shared* VNF-FG g, the state of a copy of g' do not match. For example, if a connection point belonging to a VNF (i.e. *decoder*) of the *shared* VNF-FG g is updated as a response to a provider's domain's failure, this update must be seen by all orchestrators that have a copy of the VNF-FG. The inconsistency appears when two orchestrators have the same copy of the VNF-FG, but the order of the connection point differs. In one instance, the network service will execute either before or after, resulting in the possibility of running the *decoder* before the *encoder* which violates the expected behavior of the network service. To prevent this, dependencies between reconfigurations can be enforced through causal relations.

We consider two VNF-FG reconfiguration operations Δ and Δ' to be causally related if:

1. If Δ and Δ' are VNF-FG reconfigurations belonging to the orchestrator and Δ originated before Δ', then $\Delta \rightarrow \Delta'$.

2. If Δ is the reconfiguration of a VNF-FG asked by the notification message by one orchestrator and Δ' is the reconfiguration of a VNF-FG from the receipt of the same notification message by another orchestrator, then $\Delta \rightarrow \Delta'$.

3. For any reconfiguration operation Δ'', if $\Delta \rightarrow \Delta'$ and $\Delta' \rightarrow \Delta''$, then $\Delta \rightarrow \Delta''$.

To ensure the causal consistent reconfiguration for VNF-FGs any two reconfigurations must wait until the causally precedent operations have finished their updates (such as modifying a connection point or matching attribute). We next, describe our proposed algorithm to ensure the expected behavior for *shared* VNF-FGs.

5.4.1 Causal consistent reconfiguration algorithm for VNF-Forwarding Graphs

The VNF-FG is enriched with a causal relation in the following way: If two VNF-FGs have the same identifier and are managed by different domains, then causal order is enforced by the Happened-before relation $\rightarrow$. To achieve this, each orchestrator in the federation maintains a vector clock of each entry of the VNF-FGs they manage. Whenever an orchestrator reconfigures a VNF-FG, because of a response to changes, such as migrating a VNF, they notify all orchestrators that manage the affected VNF-FGs by increasing their local vector clock and sending a message to apply the same operation on the remote VNF-FGs. The orchestrator triggers a reconfiguration based on the received notification. If the received vector clock is lower than the current one in the orchestrator, the update of either a connection point or matching attribute takes place; otherwise, it will wait until the valid notification arrives. Next, we describe our proposed algorithm and then illustrate via an example.

reconfigureVNF-FG*(m)* reconfigures a VNF-FG based on an instruction message m. The reconfiguration Δ of a VNF-FG g is triggered by either an *updateVNFFGClassifier* or *updateVNFFGRenderedServicePath* message. After an update takes place, replicas of the VNF-FG are also reconfigured using the same type of messages sent to other orchestrators. The last message is a *notifyVNFFGUpdate* sent to all orchestrators that do not have a replica of the VNF-FG and triggers only a clock update. We assume all copies of the VNF-FG g have the same state such that $\forall g' \in G \mid g.id = g'.id \longrightarrow \phi(g) = \phi(g')$, where $g.id, g'.id$ is the identifier of the original VNF-FG g and all its external copies g'.

The major steps of the algorithm are as follows.

1. Check the type of notification message. If the instruction is *hidden*:
   (a) Reconfigure the VNF-FG.
   (b) Increase the clock.
   (c) Send a notification to all other orchestrators.
2. Check the type of notification message. If the instruction is *visible*:
   (a) Increase internal logical clock based on the received one.
   (b) Compare the internal logical clock with the received one. If the difference is equal to one, apply the reconfiguration; otherwise, save in a buffer as a pending operation.
   (c) If the reconfiguration took place, check and apply all stored reconfigurations.

We now examine these steps in detail.

1. Check the type of notification message.
   The orchestrator checks what the type of message m has arrived.

   If the message m contains an *hidden* instruction (triggered from another orchestrator)

   (a) Reconfigure the VNF-FG.
   The orchestrator o applies the change to the VNF-FG g. The change Δ can be either a connection point for a *updateVNFFGRenderedServicePath* message; otherwise, a matching attribute for a *updateVNFFGClassifier*. This modifies the state of the VNF-FG g to a new state to reflect the changes done e.g. $\phi(g) \longrightarrow \phi'(g)$.

   (b) Increase the clock.
   The logical clock of either the matching attribute or connection point increases by one based on the index of the orchestrator e.g. $v_i \leftarrow v_i + 1$ where i is the index of the orchestrator o and v is the logical clock.

   (c) Send notification to all other orchestrators.
   This step is divided into two parts. In the first part, the orchestrator creates either a *updateVNFFGRenderedServicePath* or a *updateVNFFGClassifier* message. It appends his logical clock v_i along with the new state of the VNF-FG $\phi'(g)$ and sends it to other orchestrators, $o' \in O$, that manage replicas of the VNF-FG g. In the second part, the orchestrator creates a *notifyVNFFGUpdate* by appending his logical clock v_i and sends it to the other orchestrators that were not present in the first part.

2. Check the type of notification message. If the instruction is external:

   (a) Compare the internal logical clock with the received one. If the difference is equal to one (i.e. they only differ by one value), apply the reconfiguration; if it is greater than one, save in a buffer as a pending operation. However, if either (i) a difference of zero, or (ii) the local clock is strictly greater than the received one, discard the message as it is repeated.
   The two vector clocks v, v' are compared by checking all indexes:

   - If $\forall k, v'_k > v_k$ discard the redundant message.
   - If $\sum_{k=0}^{n} |v'_k - v_k| \geq 2$ save the reconfiguration instruction in a buffer until message with the correct logical clock arrives, e.g. the difference between the clocks is equal to one.
   - If the previous cases are false, then reconfigure the VNF-FG g' as described before in step 1.a

   (b) If the reconfiguration took place, check and apply all stored reconfigurations. Compute the length of the buffer q, $|q|$, and then check all pending operations stored in the buffer to see if now one or more are valid. Iterate over the

length $|q|$ and take one reconfiguration instruction and apply step 2.b. If the instruction is valid (e.g. the difference between vector clocks equals one) the reconfiguration takes place; otherwise, store it again in the buffer.

We illustrate the execution of our proposed algorithm in Figure 5.5 by comparing it to the problem of inconsistent reconfiguration we described in the use case from Section 5.2, as shown in Figure 5.3. The orchestrator in the United States updates the *shared* VNF-FG for the *decoder* by changing the connection point with the value a. Differently from the example shown in step V from Figure 5.3, here, the update is stored while the correct message arrives. In step VI, the first update arrives and the first orchestrator reconfigures the VNF-FG. Then, in steps VII-IX, the stored update is finally applied. At the end of the reconfiguration, all VNF-FG copies have the same values for x_z for the *z-th decoder* (i.e. $x_1 \equiv x_2 \equiv x_3$). This is an example of a consistent reconfiguration.

5.5 Implementation and evaluation

In this section, we compare algorithms by measuring the inconsistencies after reconfiguring a shared VNF-FG. Ideally, algorithms would prevent all inconsistencies while reconfiguring the VNF-FGs; preventing partial or total failures of the shared network services. In practice, this involves a cost. Identifying and preventing inconsistencies comes with an overhead that we measure by the number of messages sent and the time

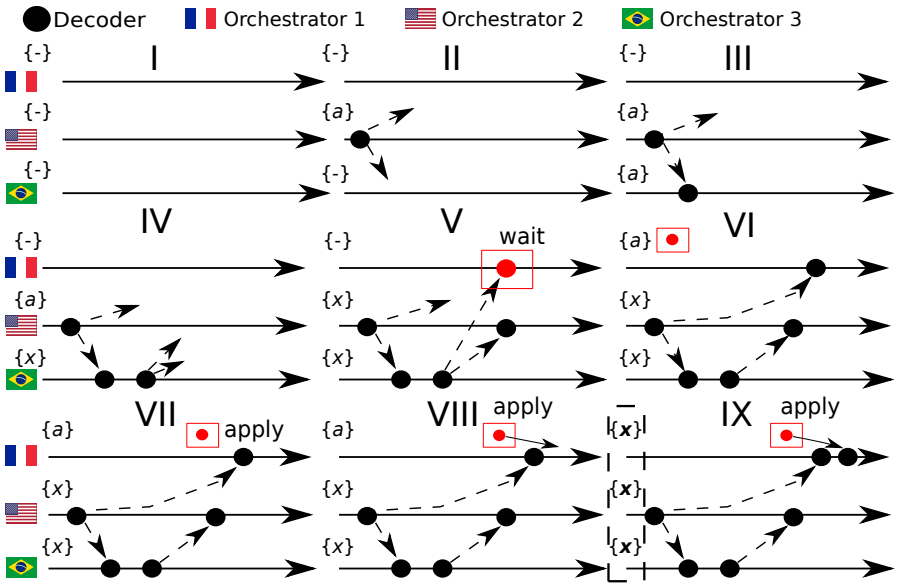

Figure 5.5: Causal VNF-Forwarding Graph reconfiguration. In the end, all copies of the *shared* VNF-Forwarding Graph have the same values. This is an example of a consistent reconfiguration.

Table 5.2: Evaluation parameters

	Scenarios		
	Best-case	Average-case	Worst-case
Reconfigurations	64,128,256	64,128,...,640	
Network delay	-	0 - 60 seconds	
Probability messages	-	10 - 50 %	
Evaluation Metrics	Inconsistencies, Messages sent, Time to reconfigure		
Repetitions	5		

spent waiting for the arrival of the correct shared VNF-FG. We only consider stateless VNFs and let the impact on stateful VNFs for future work. Moreover, since we followed the ETSI reference architecture, our solution could be integrated with the ETSI MANO framework within software implementations.

We consider two algorithms: The ETSI-compliant reconfiguration and our proposed causal-based algorithm. The ETSI reconfiguration applies updates with no strict order or timing constraints [ETSI 2019b]. The causal algorithm ensures an ordering of updates according to the happened-before relation to reflect changes as they happened in the system for dependent scalings.

To compare the algorithms, we deployed locally five domains. Each domain has its policies, resources, and VNFs to support services. Many domains that entail shared VNF-FGs can create shared services using random seeds. Reconfiguring a shared VNF-FG involves two or more domains. When an orchestrator receives a request to reconfigure a given shared VNF-FG, it will apply the change locally and notify other orchestrators that share the same VNF-FG. According to the reconfiguration logic of each algorithm, the orchestrator reconfigures the shared VNF-FG immediately or waits until the correct message arrives (e.g. the difference between their logical clocks is one).

We summarize the evaluation parameters in Table 5.2. Scenarios deploy random VNF-FG reconfigurations and we repeated them five times, taking the average for the number of inconsistencies and messages sent for reconfiguration. The previous two metrics allow us to compare the performance of the two algorithms. To prevent a long reconfiguration time, we set a threshold of five minutes; after this, the reconfiguration is inconsistent.

We variate two parameters: network delay and probability of redundant messages. The network delay measures the effects of messages arriving out of order. We set four different limits of 0, 10, 30, and 60 seconds. Messages have a higher probability of being sent out of order with greater limit values. For example, a message will probably be sent as it was generated with a delay of 10 seconds instead of waiting a maximum of 60 seconds. We changed the number of repeated messages according to a threshold probability. We set the probability according to four values of 0, 10, 30, and 50%. A higher probability means redundant messages are more likely to be sent by the orchestrators. We chose these parameters and their values to reflect different scenarios and measure

Table 5.3: Results for the ideal scenario. There are no inconsistencies and the standard obtains a better performance.

		Causal	**Standard**
		Messages	Messages
Reconfigurations	64	249	137
	128	505	269
	192	712	396
	256	1017	533
Inconsistencies		0	

the performance of each algorithm. Under these conditions, the algorithms must detect repeated messages and drop them. We implemented the algorithms in an x64 Ubuntu 18.04 LTS machine with an Intel i7 processor with 32 GB of memory. The source code can be found in [1].

5.5.1 Evaluation of our proposed algorithm

We evaluate the algorithms based on three scenarios: Best, average, and worst cases. **Best-case scenario: Sequential reconfigurations**. In this setup, orchestrators can send multiple reconfigurations at the same time with zero network latency and no repeated messages. This case reflects the fundamental assumptions of the ETSI current reconfiguration. **Average-case scenario: Sequential and concurrent reconfigurations**. In this setup, orchestrators reconfigure multiple VNF-FGs at the same time with variable network latency and a probability of repeated messages. The delay shows how well the algorithms perform when messages arrive out of order. Repeated messages show how well an algorithm identifies and prevents unnecessary reconfiguration, which already took place. **Worst-case scenario: Concurrent-only reconfigurations**. Under this setup, orchestrators reconfigure VNF-FG at any given time.

Table 5.3 shows the results for both the proposed algorithm and the standard. Figure 5.6 shows the results for the network delay variation. Figure 5.7 shows the number of messages sent by each algorithm.

5.5.2 Discussions

As shown by Table 5.3, in the ideal case, both the current and our proposed algorithm can reconfigure VNF-FGs without inconsistencies. However, our algorithm has almost double the messages sent compare to the standard. More precisely, our algorithm has quadratic time complexity $\mathcal{O}(n^2)$ compared to the linear $\mathcal{O}(n)$ of the standard; for space, both have linear complexity $\mathcal{O}(n)$. Thus, under this ideal scenario, the standard algorithm is a better solution. In more real-life scenarios, this is not the case, and our

[1] https://doi.org/10.5281/zenodo.5336650

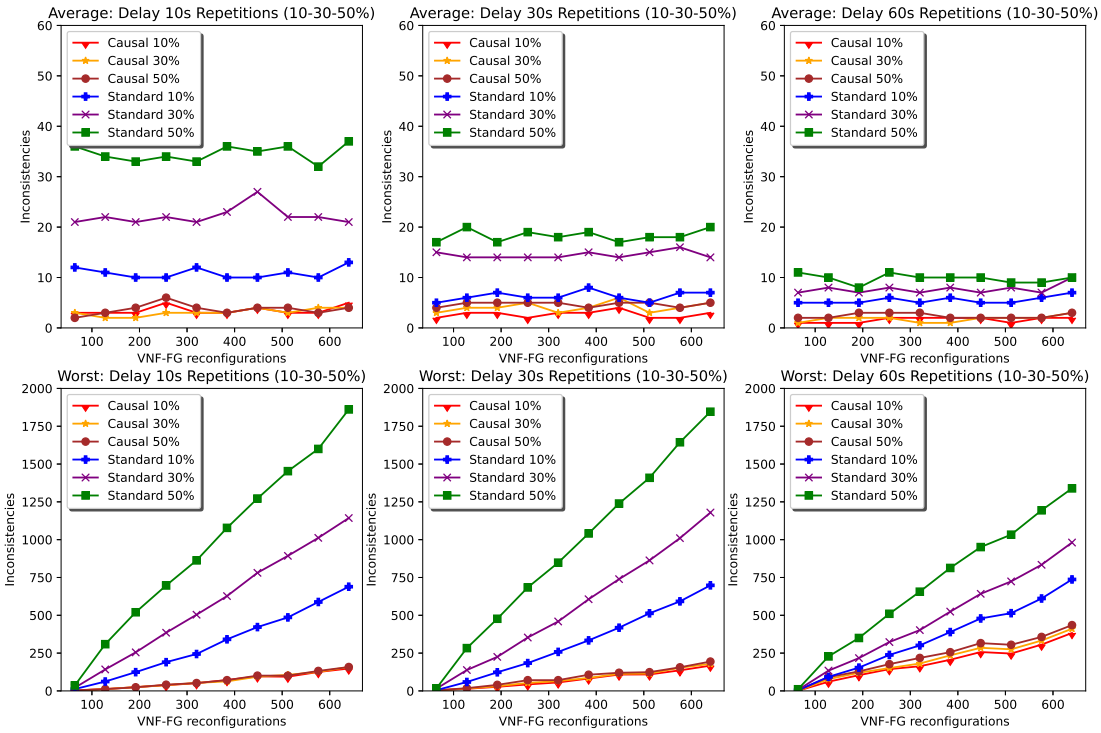

Figure 5.6: Inconsistencies as a number of VNF-Forwarding Graph reconfigurations, lower is better. The top row (A, B, C) shows the average scenario. The bottom row (D, E, F) shows the worst scenario. **The proposed causal algorithm gets fewer inconsistencies in both scenarios despite unwanted network conditions**.

proposed solution is preferred. To get a better understanding, we now discuss the effects of redundant messages and network delay under concurrent reconfiguration.

The results for the average and worst-case scenarios for inconsistencies are shown in Figure 5.6. Based on the causal-consistency model, we can track the causal history of the VNF-FG reconfigurations by multiple orchestrators. This means that redundant messages can be identified and removed, leading to a more robust behavior of our algorithm compared to the current standard. This is shown for both the average and worst-case scenarios for our proposed algorithm, where the lines are closer than that of the standard algorithm. Identifying redundant messages on the standard could be done to mitigate the effects of such network conditions; however, even with a low probability for redundant messages, the causal algorithm gets better performance with fewer inconsistencies. Interestingly, both algorithms get fewer inconsistencies with greater delays in both the average and worst cases. This could be explained because higher delays mean it is less probable to have concurrent reconfigurations since the instructions wait before being sent. This explanation is supported because, in the ideal case, where no concurrent reconfiguration was allowed, not a single inconsistency appeared while reconfiguring the

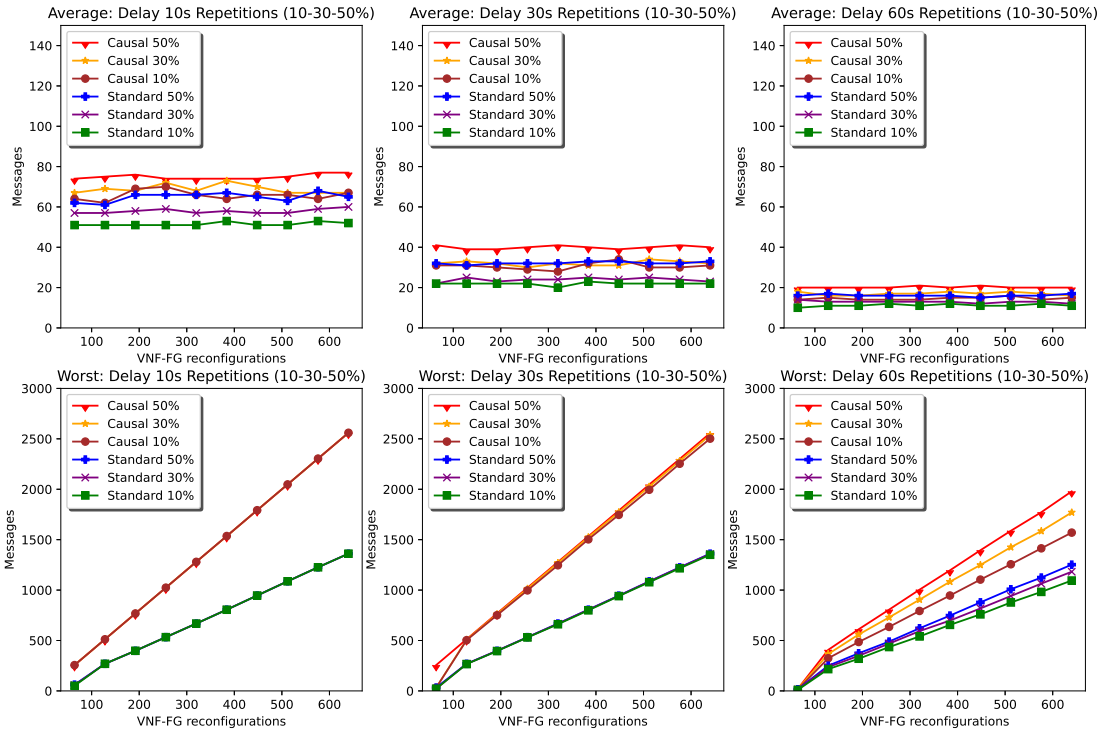

Figure 5.7: Messages sent per number of VNF-Forwarding Graph reconfigurations, lower is better. This reflects the cost to prevent inconsistencies. The top row (A, B, C) shows the average scenario. The bottom row (D, E, F) shows the worst scenario. The causal algorithm sends more messages than the standard algorithm.

VNF-FGs.

The results for the number of messages sent by both algorithms in the average and worst-case scenarios are shown in Figure 5.7. As expected, our algorithm sends more messages than the standard in all cases since it has to coordinate the orchestrators. A clear division between our proposed algorithm and the standard is shown in the bottom row of Figure 5.7 for the worst-case scenario. As discussed before, the effects of delay affect the number of messages sent, as shown in the 60 delay graph. Since fewer concurrent reconfigurations take place, there is less need of coordinating the orchestrators. As a result, the number of messages is lower for both the standard and our proposed algorithm. The number of reconfigurations measures the overhead of our solution; however, more metrics (e.g. cost/size per message) need to be evaluated in future work.

The causal algorithm gets fewer inconsistencies in both scenarios, as shown in Figure 5.6; however, concurrency still seems to pose a problem. This could be explained by the poor ability to track concurrent reconfigurations using causal relations. Moreover, despite our proposed causal algorithm getting better results than the standard, more work is required to get a more robust and practical solution for large federations. First,

in terms of scalability, while our algorithm has fewer inconsistencies than the standard, it is affected by the number of concurrent reconfigurations. This is shown in the last graph from the second row of Figure 5.6, where the number of reconfigurations is set to 640 and the inconsistencies are almost the same. Second, the number of messages while at first negligible cost grows quickly, as shown in Figure 5.7 where the number of messages is higher than 2000. Third, more evaluation is required to measure the impact of the reconfiguration algorithm on the network services. For example, such reconfiguration could have unwanted effects and violate on-functional requirements, such as QoS. We would also like to evaluate the cost of reconfiguration for each VNF-FG as now we use as a proxy the number of inconsistencies; however, measuring the cost for reconfiguration for different cloud providers would complement these findings. All the previous limitations of the algorithm and evaluation are considered as future work. Additionally, we envision handling stateful VNFs for future work. To do so, our implementation would require tracking the state of each VNF while the reconfiguration takes place. This would increase the overhead in terms of memory and time. To mitigate this overhead and the current one, we will explore a relation that identifies only the relevant events for causal dependencies. Thus, the algorithm would track only a subset of the whole causal information.

5.6 Lessons learned and perspectives

Our proposed algorithm reduces the number of inconsistencies during a VNF-FG reconfiguration compared with the standard for both the sequential and concurrent reconfigurations. For sequential ones, the delay variation and redundant messages do not seem to affect the performance of both algorithms. For sequential and concurrent reconfigurations, our proposed algorithm is more robust to such network conditions. Results highlight the limitations of tracking causality by coordinating orchestrators for concurrent reconfigurations (to a lesser degree sequential, too).

In previous chapters, we considered mostly sequential updates where the orchestrators updated either VNFs or services. This means that conflicts were not as many since the latency time, introduced to simulate non-deterministic network conditions, was short enough to apply all the operations. In previous chapters, we tested on smaller scenarios where the max number of reconfigurations was set to 100; unlike in this chapter were we considered more than 600 reconfigurations. For greater federations with lots of orchestrators and many concurrent reconfigurations, the time required to coordinate, the time to wait for the correct message, and the amount of information sent becomes a burden to service providers. One alternative to coordination is to use eventual consistency, where at some given time all the replicas for a VNF-FG converge to the same value, and by extension, they are consistent. However, this approach obfuscates that consensus among the orchestrators still needs to happen, and therefore coordination still is executed. However, there is another way to achieve consistent VNF-FG reconfiguration. In the next

chapter, we describe how the problem of VNF-FG reconfiguration can be solved without coordination.

Chapter 6

Squaring the circle: Achieving coordination-free consistent orchestration that supports non-functional dependencies when reconfiguring VNF Forwarding Graphs

---

So far, we have been advocating for orchestrators to coordinate in a federation. By doing so, we have achieved consistent migration, consistent reconfiguration of dependent services, and consistent reconfiguration of VNF-Forwarding Graphs (VNF-FG). Yet, in the last chapter, the limitations of coordination, using only causal consistency, were highlighted when concurrency entered play. The previous coordination algorithm for reconfiguring VNF-FGs, despite reducing the inconsistencies, did not prevented inconsistencies. This translates to partial failures and additional costs to the service provider. One possibility to expunge these inconsistencies is to solve consensus among orchestrators. However, this is expensive in terms of latency and scalability, which defeats one purpose of Network Function Virtualization (NFV) to have low waiting times. In this chapter, we go more in depth on the consistent VNF-FG reconfiguration and show how to square the circle. Achieving a coordination-free orchestration algorithm to support the consistent VNF-FG reconfiguration without consensus. Figure 6.1 shows the current step towards a coordination-free orchestration algorithm. In this chapter, we present the first coordination-free algorithm to reconfigure network resources for VNF-based network services. We propose two variants of such algorithm to cope with different use cases. This chapter ends the contributions of the research, in the next one we discuss the perspectives of the research and future work.

6.1 Introduction

The problem of consistent VNF-FG reconfiguration involves updating or extending a VNF-FG responding to new demands [Houidi 2020]. To ensure the other orchestrators

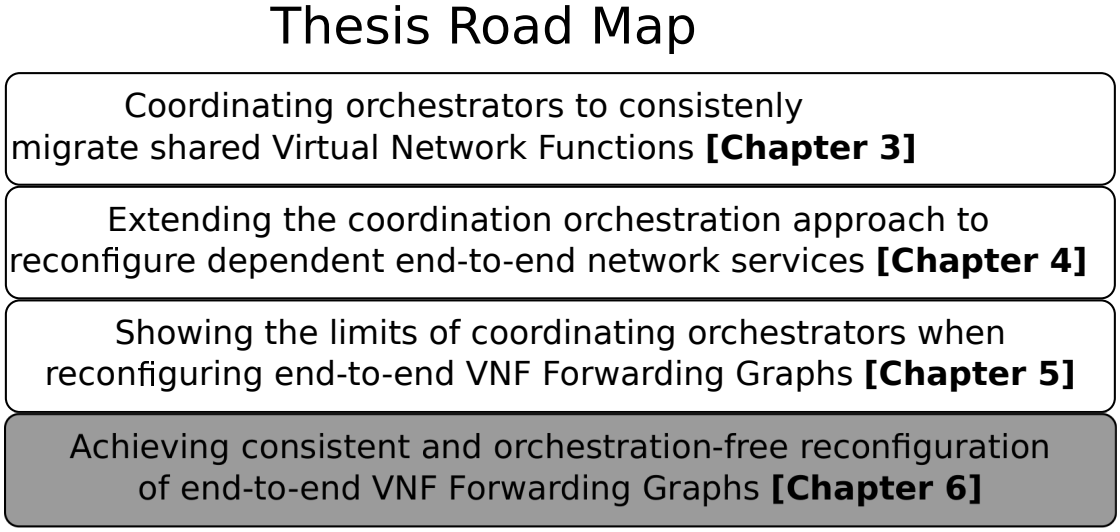

Figure 6.1: Thesis roadmap. In this chapter, we go more in depth for the problem of consistently reconfiguring the VNF-Forwarding Graphs of end-to-end network services. We show why a coordination-free approach is viable; even when orchestrators refuse a reconfiguration on the fly.

apply the same updates, the orchestrator that changed the VNF-FG sends messages to notify other orchestrators of the changes. In an ideal scenario, the orchestrators' replicas always achieve a consistent state; however, since there is no shared global reference between the orchestrator replicas, it is possible to update concurrently a shared VNF-FG. Moreover, since orchestrators share services, external dependencies introduce non-functional dependencies to prevent unwanted effects of reconfiguration. For example, updating the connection point of a VNF-FG can optimize the latency in a given administrative domain; however, for other replicas, it may not be the case as many services can use these replicas. In other words, orchestrators can reject new changes from incoming replicas and decide only to reconfigure if their non-functional dependencies are satisfied after the reconfiguration. Thus, the combination of concurrent reconfiguration, limited knowledge, and non-functional dependencies introduces conflicts that must be fixed to achieve a consistent state at the end of reconfiguration.

In the previous chapter, we highlighted the limits of the state of the art reconfiguration algorithms. Our proposed algorithm obtained better results than the European Telecommunications Standards Institute (ETSI) standard reconfiguration algorithm. However, concurrent updates posed a problem for both algorithms. Indeed, concurrent reconfigurations induce inconsistencies that drift replicas each time an orchestrator triggers a reconfiguration. Traditionally, to resolve inconsistencies because of non-deterministic network conditions (e.g. delay, repeated messages, orchestrators connect/disconnect), the orchestrators choose between two competing goals in terms of per-

formance and strong consistency. Nowadays, performance is preferred over strong consistency properties by ensuring eventual consistency among orchestrators [Bailis 2013]. In case of a conflict, orchestrators must coordinate among themselves and solve consensus or select a single orchestrator to resolve conflicts [Samdanis 2018]. However, the performance of both undermines the goals of multi-domain orchestration.

In this chapter, we focus on the consistent VNF-FG reconfiguration under multi-domain environments, considering non-functional requirements. Our main contributions are:

- The design of a coordination-free orchestration algorithm for consistent VNF-FG reconfiguration under multi-domain federations. Our proposed algorithm supports non-functional dependencies that have not been addressed so far in the literature for VNF-FG reconfiguration. Unlike current orchestration algorithms, our proposed algorithm consistently reconfigures the VNF-FG of a shared network service without a coordination phase between the orchestrators. By skipping the coordination phase, we open the door for dynamic federations where orchestrators join and leave temporarily. (Section 6.3).
- The tailoring of two different variants of our proposed algorithm to address various applications. For critical systems, we propose a preventive variant that ensures reconfigurations are always accepted by all replicas (Section 6.3.1). For less stringent applications, we propose a corrective variant that enables contingent reconfigurations to happen as they arrive (Section 6.3.2).

We formally proof the correctness of both variants (Section 6.3.3) and evaluate both (Section 6.4). We compare both variants to the ETSI standard and discuss the trade-offs in terms of cost and performance.

The rest of the chapter is organized as follows: Section 6.2 details the problem of consistent reconfiguration with non-functional dependencies. Section 6.3 describes the idea to achieve coordination-free orchestration. It also details more of the proposed variants and presents the proof of correctness for both. Section 6.4 presents the evaluation and comparison of the two variants of our proposed algorithm and the current VNF-FG reconfiguration algorithm. Section 6.5 concludes the chapter. We present the notation for this chapter in Table 6.1 (see Section 2.4 for a more detailed description of each variable).

6.2 Consistent VNF-Forwarding Graph reconfiguration with non-functional dependencies in multi-domain environments

In Chapter 5, the problem of consistent VNF-FG reconfiguration was explored. An example of a consistent VNF-FG reconfiguration is shown in Figure 6.2. Recall that

Table 6.1: Notation for this chapter. Some of the variables were defined in the system model (see Section 2.4 for a more detailed description of each variable).

Variable	Meaning
$O = \{o_1, o_2, o_3\}$	The set of orchestrators
$G = \{g^1, g^2\}$	The set of VNF-FGs each numbered.
depends	The dependency relation
Δ	A VNF-FG Reconfiguration operation
χ_g	The counter for VNF-FG g
θ	The identifier of a receiver orchestrator
ϑ	The identifier of an sender orchestrator
L_θ	Pending operations for the the orchestrator θ
h_g^θ	The heap of accepted values for VNF-FG g from the θ orchestrator
l_g^θ	The list of negated values for VNF-FG g from the θ orchestrator
ε	The initial value for a data structure.
$\phi(x_i)$	The state of the i-th VNF-FG x replica
k_o^*	The highest orchestrator identifier
x_i	the i-th replica of a VNF-FG x
τ	The top operation
Ω	the set of orchestrators who manage a network resource.

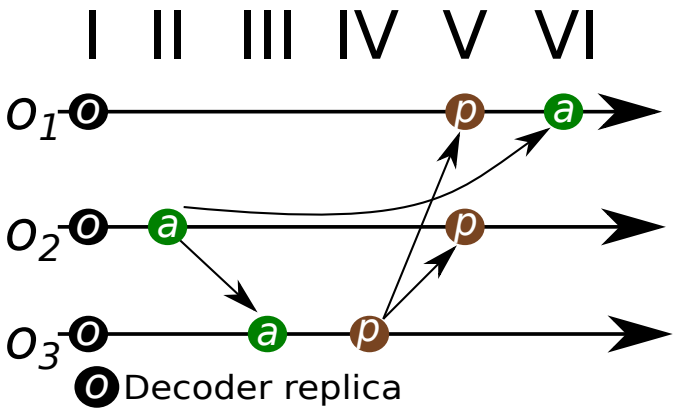

Figure 6.2: VNF-Forwarding Graph reconfiguration. All replicas of the *shared* service's VNF-Forwarding Graph have the same value. This is an example of an consistent reconfiguration.

in the previous chapter, we consider the case where the reconfigurations were always accepted by the replicas. However, this is not always the case, as services belong to many services with different non-functional dependencies.

We now introduce the problem of consistent reconfiguration with non-functional dependencies. Because services are *shared* with many orchestrators, they can also be used by different services. Thus, we consider that orchestrator replicas may negate ongoing

reconfigurations because of non-functional dependencies. Such reconfigurations could affect non-functional requirements of another *shared* network service. In this work, we generalize non-functional requirements as either being accepted or not. The details of why an orchestrator rejects an ongoing reconfiguration are out of the scope of this chapter.

Network services are *shared* with multiple orchestrators via replicas. These replicas can also be used as external dependencies for other services. To prevent violation of service level agreements, such as increasing latency, exposing security flaws, or degrading the QoS, the orchestrator verifies if the reconfiguration proposed by an external orchestrator replica will affect the service's non-functional requirements. If it is the case, the orchestrator will not accept the reconfiguration. To stay consistent, all orchestrators need to consider the negation by non-functional dependencies. We name this problem the Consistent VNF-Forwarding Graph Reconfiguration with Non-Functional Dependencies (CVNF-FGR-NF).

We illustrate an example of an inconsistent reconfiguration extending the same example for reconfiguration presented in Figure 6.2 but where the orchestrators can refuse a reconfiguration. This is shown in Figure 6.3. In this scenario, four orchestrators (o_1, o_2, o_3, o_4) from different administrative domains manage two encoders g^1, g^2 and decoders g^3, g^4. The first orchestrator o_1 manages the encoder g^1, the second orchestrator o_2 manages the encoder g^2, and so on. At the beginning, all *encoders* and *decoders* start with initial values represented by o, and $*$, as shown in the bottom of Figure 6.3. In step I, both the third and fourth orchestrators change the value of their respective decoders g^3, g^4 with different values a, p, respectively. Step II shows how the first and second orchestrators update their encoders. The first orchestrator o_1 verifies the proposed reconfiguration. After accepts the reconfiguration changing the value of the encoder g^1 to a new value y. The second orchestrator o_2 does the same and updates encoder value to y. In step III there are three concurrent tasks. Firstly, the first orchestrator o_1 gets the notification from the fourth o_4 to reconfigure the VNF-FG. The first orchestrator does not accept the reconfiguration and sends a reply o_4. Secondly, the second orchestrator o_2 gets the same notification from the fourth one o_4. Since it accepts this most recent update, the orchestrator updates the value of the dependency to x; then, it replies to o_4. Thirdly, the concurrent task is the positive reply from o_1 to o_3. In step IV, both o_3 and o_4 get a positive reply from one of their dependencies. The third orchestrator o_3 receives a positive reply from o_2; while, o_4 from o_2. Since the third orchestrator o_3 received both positive replies from its dependencies, it will notify the fourth orchestrator o_4 to change the value of the VNF-FG replica. In step V, two concurrent events happen as messages arrive to the fourth orchestrator o_4. Firstly, the instruction to update the value of the VNF-FG replica arrives from o_3. Secondly, the negative reply from o_1 arrives. Thus, the fourth orchestrator o_4 can choose between doing the reconfiguration or remaining in the initial state. Here a non-deterministic output creates an inconsistency. The first choice is that the fourth orchestrator updates the value of the VNF-FG replica to a; however,

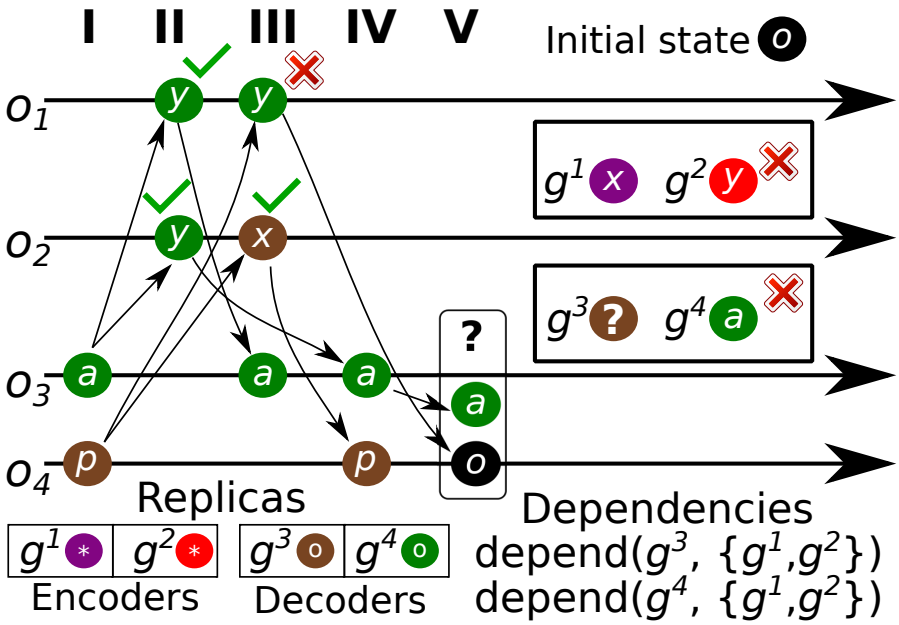

Figure 6.3: VNF-Forwarding Graph reconfiguration with non-functional dependencies. Replicas of the VNF-Forwarding Graph and their dependencies have different values. The fourth orchestrator o_4 cannot determine what will be the VNF-Forwarding Graph value for the concurrent values. This is an example of an inconsistent reconfiguration.

the dependencies have different values. The other choice is to remain in the initial state; but, the replicas have different values. Moreover, because orchestrators can share the decoder among other orchestrators with limited knowledge, conflicts will arise as the replicas diverge further and further after each reconfiguration. This is an example of an inconsistent reconfiguration with non-functional dependencies.

6.2.1 Relation between different problems

The CVNF-FGR problem (Chapter 5, see Section 5.4) is a particular case of the CVNF-FGR-NF, where orchestrators always accept the reconfiguration proposed by replicas coming from other orchestrators. Furthermore, the CVNF-FGR-NF is divided into two classes according to the wanted behavior in terms of consistency. If a reconfiguration applied always is valid to all orchestrators, then the problem does not support fault-tolerance (i.e. critical applications). We call this the preventive problem. If reconfigurations can be undone, we consider the problem to support fault tolerance. We call this the corrective problem.

6.3 Coordination-free orchestration algorithm for multi-domain environments

In the Consistent Dependent VNF-Forwarding Graph Reconfiguration problem, the goal is that all VNF-FG replicas and their dependencies have the same values. However, non-deterministic network conditions (e.g. out-of-order delivery of messages) and concurrent updates from different orchestrators can have unwanted effects while reconfiguring the VNF-FG. Moreover, dependencies can negate the updates, amplifying the negative effects of the previous conditions. At the of reconfiguration, orchestrators can have different values for the VNF-FG. To prevent inconsistencies, a conflict resolution mechanism is required.

Traditionally, in distributed systems, conflict resolution is done by consensus among all orchestrators to achieve Sequential Consistency/Linearizability (see Section 2.6.1). The problem is that consensus has a lackluster performance that hinders the applicability of such solutions. Even worse, NFV is expected to have low latency (e.g. milliseconds) for network services. One way to circumvent this low performance is to relax the consistency guarantees such as with the Strong Eventual Consistency (SEC) (see Section 2.6.4). Such a consistency model achieves the ideal trade-off between consistency, availability, and partitioning. Next, we introduce the dependency relation δ.

The binary dependency relation δ takes as input two different VNF-FGs ($g \in G, g' \in G'$) (i.e. is not possible to have a dependency with itself). As the VNF-FGs have replicas they must the same values, thus if the relation $\delta(g, g')$ holds, this implies that every element in G has a dependency with every other element in G' (i.e. $\exists \delta(g, g'), g \in G, g' \in G' \implies \delta(G, G')$. If δ is a bi-directional relationship, then it also implies that all elements in G' have a dependency relation (i.e. $\delta(G', G)$). This means whenever a reconfiguration takes place, the orchestrator who does the update has to notify other orchestrators who manage a VNF-FG in the super-set $G \cup G'$.

We consider Conflict-Free Replicated Data Types (CRDTs) [Shapiro 2011b] as an automatic conflict resolution mechanism. The idea is to design an orchestration algorithm using data structures that support SEC to avoid coordination phase while having consistency. We now describe the two variants of our algorithm.

The first variant prevents inconsistencies by applying updates only when all replicas and dependencies have positively answered an updated proposal. This means that an orchestrator that manages a VNF-FG $g \in G$ with a dependency $g' \in G'$ sends a reply to all other orchestrators in the set $G \cup G'$. Whenever an orchestrator that manages a dependency of the VNF-FG g receives the proposal, it will reply either positively or negatively. The proposal is not applied until all answers are received and discarded if any reply is negative. The second variant is more permissive, as it allows updates to take place at the moment the notification arrives. Similar to the first variant, under the corrective variant, the orchestrator that updates the VNF-FG $g \in G$ must send to all orchestrators that manage a VNF-FG in the set $G \cup G'$. However, the receiving orches-

trator will only notify a negative answer to the others. When an orchestrator receives a negative answer, it must make the required changes to be in the most promising, consistent state. Both variants of our proposed algorithm achieve SEC. Next, we describe in detail the two variants to achieve the consistent dependent VNF-FG reconfiguration, and how they achieve SEC. Thus, they eliminate the inconsistent reconfiguration with the example shown in Figure 6.3 (see Section 6.2).

6.3.1 Preventive variant

The preventive algorithm reconfigures a VFN-FG only when all affected orchestrators have accepted the changes ensuring that the VNF-FG and their non-functional dependencies are consistent for all replicas. We name the preventive variant as *CF-P*. Algorithm 4 (see Section 6.3.1.2), Algorithm 5 (see Section 6.3.1.2), and Algorithm 6 (see Section 6.3.1.3) describe in detail the *CF-P* variant.

Consider the same reconfiguration of a shared VNF-FG as in Section 6.2 where four orchestrators o_1, o_2, o_3, o_4 manage two encoders g^1, g^2 and two decoders g^3, g^4, respectively. The first orchestrator o_1 manages g^1, the second orchestrator o_1 manages g^2, and so on. Figure 6.4 shows the execution for the reconfiguration to ensure all replicas are consistent by having the same values. In step 1, the third and fourth orchestrators try to update concurrently the VNF-FG. The third orchestrator o_3 proposes a new value a for his VNF-FG; while, the fourth orchestrator o_4 updates his to p, respectively. Both store them in their lists and send the proposal to all affected orchestrators (in this example all the others). In step 2 three concurrent tasks execute. Firstly, the first orchestrator o_1 receives the proposal from o_3. After validating this proposal it stores it in a list of pending reconfigurations, as shown in the right side of Figure 6.4, where a question symbol is stored in the entries for g^1 and g^3, respectively. After, o_1 sends notification to all affected orchestrators. Secondly, similarly to o_1, the second orchestrator o_2 accepts, stores, and sends notifications for value a. Thirdly, o_4 receives the proposal from o_3 and also does the three operations as before. In Step 3 four concurrent operations execute, we will focus only on the first two as the two others also apply the same three operations. For the first task, the first orchestrator o_1 receives the proposal from o_4. The orchestrator verifies if the reconfiguration is valid; however, it decides not to accept it. The first orchestrator o_1 then adds the proposal as negative as shown in the right side of Figure 6.4. All subsequent notifications of proposal for value p will be automatically negated. For the second task, the second orchestrator o_2 validates the proposal for value p and applies the same three operations (accept, store, send) as in previous steps. Steps 4 and 5 shows how more notifications arrive to the orchestrators. We focus on the notification from o_1 to o_4. Since all values are already validated by replicas, the fourth orchestrator finally can reconfigure its VNF-FG replica. This is shown on Figure 6.4 by the color change and value of the g^4 decoder. Eventually all notification and proposals arrive with steps 6-8. At the end of reconfiguration, all VNF-FG replicas have the same values. Comparing Figures 6.3 and 6.4, it can be seen how the preventive variant achieves a consistent

reconfiguration despite non-functional dependencies.

6.3.1.1 Preventive variant algorithm definition

When an orchestrator needs to update a VNF-FG, first it will create an operation to be applied to the VNF-FG (line 1). Then, the orchestrators increase the counter of the VNF-FG to help distinguish from other concurrent (line 2). After, the orchestrators compute the set of affected orchestrators (line 3). A list is created that has a dictionary relating the orchestrators with the answer (line 4). Then, it initializes the entry with his id as true (line 5). After, it appends this list to the list of pending operations (line 6). Finally, it creates a notification message and sends it to the affected orchestrators (lines 7 - 8).

6.3.1.2 Receive notification from another orchestrator function

When an new notification for an update arrives to an orchestrator it will decide to accept it or negate it. In any case, it will send a reply to all affected orchestrators. To do so

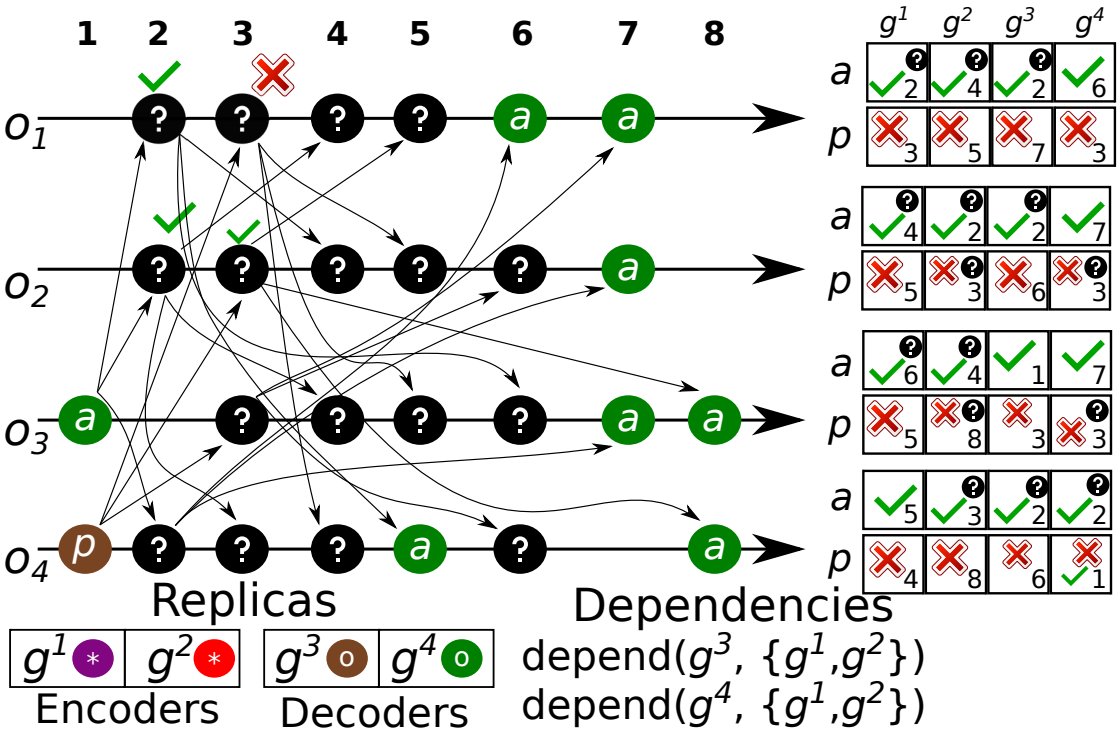

Figure 6.4: VNF-Forwarding Graph reconfiguration with our proposed **preventive** variant. At the end of the reconfiguration, replicas of the VNF-Forwarding Graph have the same value. Unlike Figure 6.3, this is an example of a consistent reconfiguration with non-functional dependencies.

Algorithm 4: Preventive variant algorithm definition. Updates a VNF-FG g, with counter χ_g, managed by orchestrator θ

1 $g \leftarrow \Delta(g)$ /* Update by applying reconfiguration operation */
2 $c_g^{\Delta} \leftarrow \chi_g + 1$ /* Create counter for the reconfiguration */
3 $O_g \leftarrow (\forall o \in O, g' \in G; \| \; o \sim g', g.uid = g'.iud) \cup (\forall o \in O, g'' \in G \mid o \sim g'', \delta(g, g''))$
/* Compute orchestrators to sent */
4 $l_g^{\Delta} \leftarrow [0 \mid \forall o \in O_g]$ /* Create empty list */
5 $l_g^{\Delta}[\theta, c_g^{\Delta}] \leftarrow True$ /* Initialize list entry */
6 $L_\theta \leftarrow L_\theta \cup l_g^{\Delta}$ /* Add list to pending operations */
7 $\sigma = \{\theta, g.uid, \Delta, \chi_g\}$ /* Create a notification update message */
8 $\forall o \in O_g, send(o, \sigma)$ /* Send notification to all ids */

first, it computes the set of affected orchestrators (line 1). Then if the notification for the update is the first one received, it will create a list and add it to the list of pending operations (lines 3 - 5). Then, if it is feasible to update, it will (i) set his entry and the sender orchestrators as accepted (lines 8 - 9), (ii) apply the update if all values are accepted (lines 11 - 15). If it is not accepted, the orchestrator marks all entries as false (lines 17 - 18). Finally, the orchestrator creates a reply message containing the answer (either accepted or negated) and sends it to the affected orchestrators (lines 20 - 21). This is more detailed in Function 5.

6.3.1.3 Receive reply for preventive variant function

Whenever an orchestrator receives a notification from another orchestrator it will first add the reply to his list of orchestrators (line 1). Then it verifies if all entries of the entry are filled and accepted (line 2). If it is so, it will accept the update, apply it and update the counter of the VNF-FG (lines 3 - 6). This is more detailed in Function 6.

6.3.2 Corrective variant

The corrective algorithm reconfigures a VFN-FG when a notification arrives without waiting and does not send a notification to other orchestrators. Only when a dependency does not accept a reconfiguration due to violating non-functional requirements, the orchestrator will send a negative notification to the others. Whenever an orchestrator receives a negative notification, it will reconfigure the VNF-FG to a provisional state after merging with the notification. We name the corrective variant as *CF-C*. Algorithm 5, Algorithm 7 (see Section 6.3.2.2), and Algorithm 8 (see Section 6.3.2.3) describe in detail the *CF-C* variant.

Consider the same reconfiguration of a shared VNF-FG as in Section 6.2 where four orchestrators o_1, o_2, o_3, o_4 manage two encoders g^1, g^2 and two decoders g^3, g^4, respectively. The first orchestrator o_1 manages g^1, the second orchestrator o_1 manages g^2,

Function 5: Receive a notification from another orchestrator function. Receives notification update message $\sigma = \{\vartheta, g'.uid, \Delta, c^{\Delta}_{g'}\}$ for VNF-FG g, with counter χ_g

$O_g \leftarrow (\forall o \in O, g' \in G; \| \; o \sim g', g.uid = g'.iud) \cup (\forall o \in O, g'' \in G \mid o \sim g'', \delta(g, g''))$
/* Compute orchestrators to sent */
$\omega \leftarrow False$ /* Start answer as negative */
if $(g'.uid, \Delta) \notin L$ **then**
  $l^{\Delta}_{g'} \leftarrow [0 \mid \forall o \in O_{g'}]$
  $L_\theta \leftarrow L_\theta \cup l^{\Delta}_{g'}$ /* If entry not in list, create a list and append to orchestrator list. */
end
if *checkFeasibility(δ)* **then**
  $l^{\Delta}_{g'}[\theta, c^{\Delta}_g] \leftarrow True$
  $l^{\Delta}_{g'}[\vartheta, c^{\Delta}_g] \leftarrow True$ /* Mark entry as a positive. */
  $\omega \leftarrow True$ /* Change to positive answer */
  if $\forall i \in l^{\Delta}_{g'} = True \; \& \; c^{\Delta}_g > \chi_g$ **then**
    $l^{\Delta}_{g'}[\theta, c^{\Delta}_g] \leftarrow True$
    $g \leftarrow \Delta(g)$ /* Apply update */
    $\chi_g \leftarrow c^{\Delta}_g$ /* Update counter of the VNF-FG */
  end
else
  $l^{\Delta}_{g'}[\theta, c^{\Delta}_g] \leftarrow False$
  $l^{\Delta}_{g'}[\vartheta, c^{\Delta}_g] \leftarrow False$ /* Mark entry as negative */
end
$\varsigma = \{\theta, \vartheta, g.uid, \Delta, \omega, c^{\Delta}_g\}$ /* Create a reply message of the answer */
$\forall o \in O_g, send(o, \varsigma)$ /* Send message to all affected orchestrators */

and so on. Figure 6.5 shows the execution for the reconfiguration to ensure all VNF-FG replicas are consistent by having the same values. In step 1, the third and fourth

Function 6: Receive reply for preventive variant function. Receives a reply message $\varsigma = \{\vartheta, g'.uid, \Delta, \omega, c^{\Delta}_{g'}\}$ for VNF-FG g, with counter χ_g

$l^{\Delta}_{g'}[\vartheta, c^{\Delta}_g] \leftarrow \omega$ /* Mark entry with answer ω */
if $\forall i \in l^{\Delta}_{g'} = True \; \& \; c^{\Delta}_g > \chi_g$ **then**
  $l^{\Delta}_{g'}[\theta, c^{\Delta}_g] \leftarrow True$
  $g \leftarrow \Delta(g)$ /* Apply update */
  $\chi_g \leftarrow c^{\Delta}_g$ /* Update counter of the VNF-FG */
end

orchestrators update concurrently the VNF-FG. The third orchestrator o_3 updates its VNF-FG with a new value a; while, the fourth orchestrator o_4 with p, respectively. Both add the value to the their heap and then send the proposal to the affected orchestrators. In step 2, two concurrent tasks execute. Firstly, the first orchestrator o_1 receives the proposal from o_3. After validating this proposal it applies the reconfiguration to VNF-FG g^1 and adds the state to the heap. This is shown in the right side of Figure 6.5. Secondly, similarly to o_1, the second orchestrator o_2 accepts, reconfigures, and saves the state. In step 3, three concurrent tasks execute. Firstly, the proposal from o_4 arrives to o_3. The orchestrator verifies if the reconfiguration is valid; however, it decides not to accept it. The first orchestrator o_1 adds it to the list of negative proposals and notifies all the affected orchestrators (in this example all the others). Secondly, the proposal from o_4 arrives to o_2; unlike o_3, o_2 accepts the proposal and reconfigures the VNF-FG g^2 to match the state of value p. This is shown in Figure 6.5 where the top now is p; unlike in the previous step 2. Thirdly, the proposal from o_3 arrives to o_4 who accepts it. However, because his reconfiguration takes precedence, it will not apply the reconfiguration. In steps 4 and 5, the rest of notifications arrive. Whenever an orchestrator receives a negative reply, it reconfigures again to another state. The value in the heap is removed and added to the list of negated proposals. This is shown in the fourth and fifth steps for Figure 6.5. At the end of reconfiguration, all VNF-FG replicas have the same values. Comparing Figures 6.3 and 6.5 it can be seen how the preventive variant achieves a consistent reconfiguration despite non-functional dependencies.

6.3.2.1 Corrective variant algorithm definition

When the orchestrators apply a reconfiguration operation for a VNF-FG. First, it will apply the reconfiguration operation (line 1). Then, the orchestrator increases the counter (line 2). After, it computes the set of affected orchestrators (line 3). Then, it appends the reconfiguration operations in the heap which will be in the top since it has the greatest counter (line 4). After, the orchestrator creates a notification message for the update (line 5). Finally, the orchestrator sends a notification to all the affected orchestrators (line 6).

6.3.2.2 Update heap function

This functions computes the update of the heap when a new message has arrived. If the notification already is at the list of negated, it will pass the message (lines 1 - 3). Otherwise; the orchestrator will check if it the new update is accepted or not. If it is, then the orchestrator compares the new update to the top of the heap, and sorts the updates according to their counter (lines 5 - 9). If the orchestrator rejects the update, it will add it to the list of rejected updates, and will notify all orchestrators that manage a replica of the VNF-Forwarding Graph (lines 11 - 14). This is more detailed in Function 7.

6.3.2.3 Heapify function

This function re-orders the heap for each VNF-Forwarding Graph, by computing the reconfiguration again from the consistent state. First, remove the top reconfiguration operation from the top of the heap (line 1). After, add this operation to the list of rejected reconfigurations (line 2). Then, recompute the from the consistent state. This is possible since all replicas start with the same initial configuration and its always possible to solve

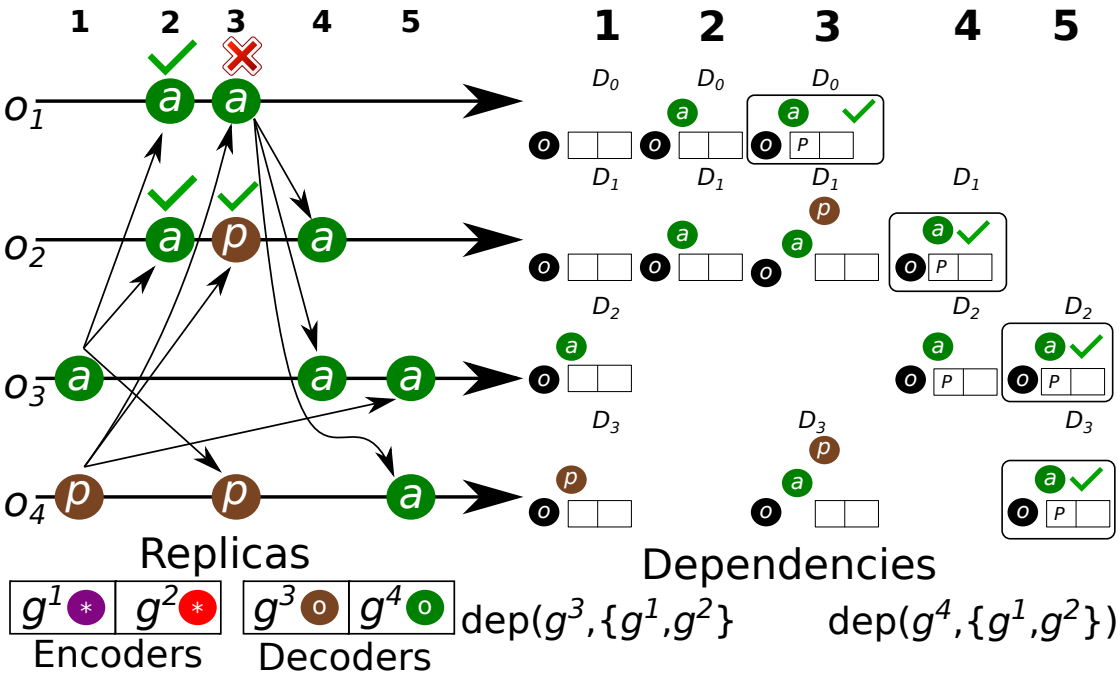

Figure 6.5: VNF-Forwarding Graph reconfiguration with our proposed **corrective** variant. At the of the reconfiguration, replicas of the VNF-Forwarding Graph have the same value. Unlike Figure 6.3, this is an example of a consistent reconfiguration with non-functional dependencies.

Algorithm 5: Corrective variant algorithm definition. Updates a VNF-Forwarding Graph g, with counter χ_g, managed by orchestrator θ

1 $g \leftarrow \Delta(g)$ /* Update by applying reconfiguration operation */
2 $\chi_g \leftarrow \chi_g + 1$ /* Increase the VNF-FG counter */
3 $O_g \leftarrow (\forall o \in O, g' \in G; \| \; o \sim g', g.uid = g'.iud) \cup (\forall o \in O, g'' \in G \mid o \sim g'', \delta(g, g''))$
/* Compute orchestrators to sent */
4 $h_g^{\theta}.insert(\{\theta, \Delta, \chi_g\})$ /* Add reconfiguration entry to heap. */
5 $\sigma = \{\theta, g.uid, \Delta, \chi_g\}$ /* Create a notification update message */
6 $\forall o \in O_g, send(o, \sigma)$ /* Send notification to all ids */

conflicts automatically (lines 3 - 4). This is more detailed in Function 8.

6.3.3 Proof of correctness for the two variants of the algorithm

We now prove the correctness of both variants of our proposed algorithm. To do so, we need to show the replicas of the VNF-FG converge eventually and both variants satisfy SEC (see Definition 7 from Section 2.6.4). Recall the replicas of each VNF-FG for the

Function 7: Update heap function. Receives a notification update message $\sigma = \{\vartheta, g'.uid, \Delta, c^{\Delta}_{g'}\}$ for VNF-Forwarding Graph g from orchestrator θ, with counter χ_g in orchestrator i

```
1  if {ϑ, Δ, c^Δ_g'} ∈ Ψ then
2  |  pass
3  end
4  if checkFeasbility(Δ) then
5  |  if c^Δ_g' > χ_g or (c^Δ_g' = χ_g and k^θ > k^i ) then
6  |  |  g ← Δ(g)
7  |  |  χ_g ← c^Δ_g'
   |  |    /* If received counter is greater than current top counter,
   |  |     apply reconfiguration operation and set new VNF-FG counter.
   |  |     */
8  |  end
9  |  h^i_g.insert({θ, Δ, c^Δ_g'})                          /* Add to heap.  */
10 else
11 |  O_g' ← (∀o ∈ O, g'' ∈ G; || o ∼ g'', g'.uid = g''.iud) ∪ (∀o ∈ O, g''' ∈ G | o ∼
   |  g''', δ(g', g''))                                    /* Compute ids to sent.  */
12 |  l^i_g.append({ϑ, Δ, c^Δ_g'})  /* Add reconfiguration to list of rejected.
   |   */
13 |  ς = {θ, g.uid, Δ, False, c^Δ_g}   /* Create a reply message of the answer.
   |     */
14 |  ∀o ∈ O_g', send(o, ς)                  /* Send message to all ids.    */
15 end
```

Function 8: Heapify function. Receives a reply message $\varsigma = \{\vartheta, g'.uid, \Delta, False, c^{\Delta}_{g'}\}$ for VNF-FG g, with counter χ_g in orchestrator i

```
1  h^i_g.remove(ϑ, Δ)                          /* Remove operation from heap.  */
2  l^i_g.append({ϑ, Δ, c^Δ_g'})     /* Add reconfiguration to list of rejected.
    */
3  Δ' ← h^i_g.top()  /* Compute new update after the new order from heap */
4  g ← Δ'(g)                                   /* Apply new update again */
```

two variants of our proposed algorithm are a distributed system. Thus, first we need to state the two conditions required for replicas of a distributed system to converge. Thus, the same conditions apply for the convergence of the VNF-FG's replicas. Shapiro et. al. defined these two conditions [Shapiro 2011a]. Definition 11 describes them.

Definition 11 (Eventual convergence conditions)
Two replicas x_i, x_j of a VNF-Forwarding Graph eventually converge if the following conditions are met:

- *Safety: $\forall i, j : C(x_i) = C(x_j)$ implies that the abstract states of i and j are equivalent.*
- *Liveness: $\forall i, j : f \in C(x_i)$ implies that eventually, $f \in C(x_j)$.*

where f is an reconfiguration operation and $C(x_i)$ the causal history of a VNF-FG replica x_i (see Section 2.5).

For both variants, we consider the following assumptions:

1. Eventual and reliable delivery. Messages have an arbitrary but finite delay. They can be sent multiple times but never lost.
2. Orchestrators can return to the federation after failure. When an orchestrator leaves, the network is left partitioned.
3. All replicas begin with an initial consistent state for all replicas. This is reflected with a special symbol in the top of all heaps ε. Such initial state is created during the deployment of the VNF-FG.

We need to prove that despite these non-deterministic network conditions, replicas still converge.

6.3.3.1 Proof of preventive variant

We prove that the preventive variant of the algorithm converges by showing it satisfies the properties of an operational-based Consistent Replicated-Free Data Type (CRDT).

Theorem 1 (Convergence of the CF-P)
The CF-P converges as an operation-based CRDT.

To prove Theorem 1 we need to show the CF-P satisfies the safety and liveness convergence properties for an operation-based CRDT. *Liveness*, for an operation-based CRDT, is satisfied if reliable broadcast channel guarantees that all updates are delivered at every replica, in a delivery order $<_d$ specified by the data type [Shapiro 2011a]. *Safety*, for an operation-based CRDT, is guaranteed when concurrent operations satisfy the property of commutativity stated by Definition 12 [Shapiro 2011b]:

Definition 12 (Commutativity)
VNF-FG updates (f,x) and (f',x') commute, iff for any reachable replicate state s, where both x, x' are enabled, replica x remains enabled in state s ∘ x' (respectively s ∘ x), and $s \circ x \circ x' \equiv s \circ x' \circ x$

which means replicas have the same state despite the order of reconfiguration operations. We now prove that the preventive variant satisfies a delivery order $<_d$ for all replicas and updates are commutative by proving Lemmas 1, 2.

Lemma 1 (CF-P liveness)
The CF-P satisfies the delivery order $<_d$.

To prove Lemma 1, we consider the case of restricting sequential updates until one is accepted. In this scenario, is not possible to update the same VNF-Forwarding Graph until the previous update has not been accepted. This means that the orchestrator has received all positives answers (Algorithm 5 Line 9). Since no sequential updates take place and Assumption 1 (i.e. we have an eventual/reliable delivery), all updates follow a single delivery order which satisfies the liveness property. For a more permissive scenario, where sequential updates can take place despite previous ones not being accepted, the system can converge in strange ways. For example, a given update can be accepted by all orchestrators, but not its previous update; such behavior is undesired. To prevent this and the effects of repeated messages, a delivery order $<_d$ needs to be enforced by replicas. Such delivery is enforced by our algorithm by considering the value of other replicas stored in the list of affected orchestrators along with their replies (either accept or decline Algorithm 6 Lines 2-4). This is because we can identify who send an instruction along with their local counter. Such delivery order prevents accepting out-of-order updates from the same orchestrator. Thus, for both scenarios, all replicas execute updates according to a delivery order $<_d$. Consequently, Lemma 1 is true. Thus, our proposed preventive variant, CF-P, satisfies the liveness property. Next, we show the CF-P satisfies the safety property.

Lemma 2 (CF-P safety)
Concurrent operations under CF-P satisfy commutativity (Definition 12).

To prove Lemma 2 recall the state ϕ of a replica x_i is encoded in its history C. For the preventive algorithm, the history C is the list of pending operations l. After an VNF-FG update operation Δ, the state of the replica is updated as $C(\Delta(x_i)) = C(x_i) \cup \{\Delta\}$ (Algorithm 6 Line 3, Algorithm 5 Line 10). We need to show replies for a reconfiguration operation f commute.

Assume without loss of generality that a replica x_i has causal history for each affected orchestrator $C(x_i) = \{r_o\} \mid \forall o \in \Omega$ (Algorithm 4 Line 6). There exists two replies $r_o, r_{o'}$ from reconfiguration operations Δ coming from affected orchestrators o, o'; such operations are proposed to replica x_i. We need to evaluate two cases: (i) $r_o = True$, $r_{o'} = True$, and (ii) $r_o = True$, $r_{o'} = False$ (conversely $r_o = False$, $r_{o'} = True$). For

the first case, the output of the algorithm (i.e. accepting the reconfiguration or not) is independent of the arrival of r_o and $r_{o'}$, such that $C(x_i) \cup r_o \equiv C(x_i) \cup r_{o'}$ (Algorithm 6 Line 2). Thus, for the first case, the replies $r_o, r_{o'}$ commute as the output depends on the other replies.

For the second case, where there's a negative reply $r_* = False$, our proposed algorithm only applies an update if all replies are positive; thus, accepting the reconfiguration or not depends only on the negative replies. Therefore, if one reply is negative, the output is unaffected by the negative reply's order of delivery (i.e. they commute). Thus, since Lemma 2 holds, the preventive variant ensures the safety property.

Since the preventive variant CF-P satisfies Lemmas 1 and 2 it converges and supports SEC, which is what we wanted to prove. ■

6.3.3.2 Proof of corrective variant

We prove the corrective variant also converges by showing it satisfies the properties of a state-based CRDT.

Theorem 2 (Convergence of the CF-C)
The CF-C converges as an state-based CRDT.

To prove Theorem 2 we need to show the CF-C satisfies the safety and liveness convergence properties for a state-based CRDT. For this type of CRDTs, the safety and liveness are encoded in the three following properties of Definition 13 [Shapiro 2011b]:

Definition 13 (Convergence properties state-based CRDTs)

1. *The payload must be a join-semilattice (safety).*
2. *The updates are monotonic (liveness).*
3. *The merge operation among such objects computes the Least Upper Bound (LUB) (safety).*

We need to show that the corrective variant of our proposed algorithm supports the properties of Definition 13 by proving Lemmas 3, 4, and 5.

Lemma 3
The CF-C payload is a join-semilattice. (Property 1 of Definition 13)

We show how Lemma 3 holds. By definition, the data structure we consider for accepted reconfigurations is a heap ordered by a $<$ operator, in other words a join-semilattice. Next, we show how the CF-C satisfies the third condition of Definition 13.

Lemma 4
The CF-C merge operation always computes the Least Upper Bound. (Property 3 of Definition 13)

We describe how Lemma 4 holds. Each orchestrator o in the federation has associated a unique identifier k_o in a total order. Each replica x_i has associated a VNF-FG v with a heap h_v^i for contingent reconfigurations, and a list l_v^i to keep track of invalid reconfigurations. We need to evaluate three cases when merging two heaps h_v^i, h_v^j from replicas x_i, x_j for the same VNF-FG v: (i) When the top of a heap is the initial state (i.e $\tau(h_v^i) = \varepsilon, \tau(h_v^j) \neq \varepsilon$), (ii) when the top of a heap has a greater reconfiguration number than another heap (i.e. $\tau(h_v^i) > \tau(h_v^j)$, (iii) when the top of both heaps has the same reconfiguration number (i.e. $\tau(h_v^i) = \tau(h_v^j)$). For the first case, since ε is the identity value, the merge operation computes the LUB between the two heaps (Algorithm 5 Line 4). For the second case, the heap operator $<$ will take the union and re-order both heaps based on the number associated to all reconfiguration operations f, on top it will be the greatest number such that $\tau(h_v^i \cup h_v^j) = \tau(h_v^i), \tau(h_v^i) > \tau(h_v^j)$ otherwise $\tau(h_v^j)$; thus, for the second case the operator $<$ computes the LUB (Algorithm 7 Line 8). Finally, for the third case, if two updates have the same number (i.e. two operations are concurrent) the winner of such reconfiguration operation will be the one with the highest orchestrator identifier k_o^* (Algorithm 7 Line 5). Since this identifier is totally ordered, the merge operation always computes the LUB. Therefore, Lemma 4 holds. Next, we show how the CF-C satisfies the second condition of Definition 13.

Lemma 5
The CF-C updates are monotonic. (Property 2 of Definition 13)

We show how Lemma 5 holds considering both data structures of the CF-C variant. First, consider the case of a notification with a heap with a single value aside the initial ε value (i.e. $|h_v^j| = 1$) and no negated elements (i.e. $|l_v^j| = 0$). Whenever a new update (either positive or negative) arrives to replica x_i, it could either: (i) be accepted and added to the heap h_v^i according to the $<$ operator (Algorithm 5 Line 4, Algorithm 7 Line 9) or (ii) if it is not accepted, it is added to the negative list l_v^i (Algorithm 7 Line 12, Algorithm 8 Line 2). Both options update the data structures; moreover, recall the causal history $C(x_i)$ encodes the state of replica $\phi(x_i)$. Particularly, for the corrective algorithm $C(x_i) = C(h_v^i) \cup C(l_v^i))$; thus, for this first case, the update is monotonic as it always increases the state after each reconfiguration.

Now consider the general case, where a notification arrives with a heap with more than one element (i.e. $|h_v^j| > 0$), and a list of negated elements (i.e. $|l_v^j| > 0$). Since the corrective variant of the algorithm computes the union of both negated elements after the update takes place, the state of such list increases containing more information about negated elements such that $C(l_v^i) = C(l_v^i) \cup C(l_v^j)$, which is monotonic. For the heaps, if all values are accepted, then the heap will also increase the state, although ordered differently according to the $<$ operator as $C(h_v^i) = C(h_v^i) \cup C(h_v^j)$. However, if any element e of a heap belongs to the union of negated lists ($e \in l_v^i \cup l_v^j$), this element is extracted from the heaps and added to the list such that after a merge $l_v^i = l_v^i \cup l_v^j \cup e$ (Algorithm 8). Thus, the state updates monotonically showing Lemma 5 holds true.

Since the corrective variant CF-C satisfies Lemmas 3, 4, and 5, the state-based CF-C converges and supports SEC, which is what we wanted to prove. ■

6.4 Implementation and evaluation

We implemented our proposed algorithm to measure performance, costs, and trade-offs of each variant of our proposed algorithm. The following sections comprise the distributed setup (Section 6.4.1), metrics evaluation (Section 6.4.2), experiments (Section 6.4.3), and discussions (Section 6.4.4).

6.4.1 Distributed federation setup

We tested our two variants using Azure's cloud infrastructure. We chose multiple domains from the cloud provider from the following locations: North Europe, West US, South Korea, East US, and the UK. For each domain, we instantiated a virtual machine to host the orchestrator software. All virtual machines have the same configuration: 2 CPUs, 30GB of hard drive, 4GB of RAM, and Linux 18.04-LTS. Each domain has its policies, topology, and manages a single orchestrator. Nowadays, many open-source orchestrators follow the ETSI standard; but none implement the required interfaces to support a federation. Thus, we implemented a multi-domain orchestrator in Python following the ETSI standard. The source code can be found in [1].

We measure the performance of the proposed algorithms under different scenarios. Thus, we consider different parameters for each experiment as shown in Table 6.2. Next, we describe them. **Maximum delay (Max_D):** This evaluates the performance of the algorithms under low and high non-deterministic conditions. Intuitively, with greater delays worst performance. **Probability of repetitions (Pb_R):** This measures the worst-case scenario as a higher number means higher non-deterministic network conditions. **Number of reconfigurations (Nm_R):** Allows to check the performance as a function of the number of reconfigurations. Intuitively, algorithms get the worst performance with a higher number of reconfigurations. **Type of reconfigurations (T_R):** Two possible values, sequential and concurrent. In sequential updates, the first reconfiguration is done after the other, depending on the algorithm. Concurrent is freer and allows for reconfiguration to happen at any time. **Probability to negate (Pb_N):** This allows us to get a better grasp on how the algorithms behave when the reconfigurations are negated by other orchestrators due to violating non-functional requirements.

The combination of these parameters creates different scenarios, each one of different complexity. An ideal scenario would have zero delay (i.e. **Max_D**=0), null probability to repeat (i.e. **Pb_R**=0), accept all reconfigurations (i.e. **Pb_N**=0), and only sequential ones (i.e. **T_R**=Sequential) which corresponds to the assumptions for

[1] https://doi.org/10.5281/zenodo.5336614

Table 6.2: Parameters considered for the experimentation. Each parameter configuration creates different test scenarios from the ideal to the worst case.

Paramter	Range
Maximum Delay (Max_D)	1 - 100 - 1000 ms
Probability of Repetitions (Pb_R)	0 - 10 - 30 - 70 - 90 %
Number of Reconfigurations (Nm_R)	0, 150, 300,..., 3000
Type of Reconfigurations (T_R)	Sequential - Concurrent
Probability to Negate (Pb_N)	0 - 5 - 10 - 30 %

the current orchestration algorithm. The worst case would have many concurrent (i.e. **T_R**=Concurrent) reconfigurations (i.e. **Nm_R**=3000) and the highest probability to negate reconfigurations due to non-functional dependencies (i.e. **Pb_N**=30%).

6.4.2 Algorithms and metrics to evaluate

We consider our proposed algorithm with two variants and the current ETSI standard orchestration algorithm for consistent VNF-FG reconfiguration. Next, we include a brief description of each one.

- The preventive variant of our proposed algorithm (**CF-P**). It tries to prevent temporary inconsistent periods between reconfigurations by waiting until all affected orchestrators either accept or negate.
- The corrective variant of our proposed algorithm (**CF-C**). It allows for a temporary inconsistent period while minimizing messages sent by keeping the current valid state, and in case of negation, rolling back reconfigurations until a correct state is achieved.
- The ETSI standard algorithm (**NCF-E**). It allows for reconfiguration to be done using eventual consistency by applying updates the moment they arrive.

Since each algorithm offers different behaviors, we consider multiple metrics applicable to all algorithms. Next, we briefly describe each metric and indicate if either a lower or higher value is better. **Total reconfiguration time:** This metric measures the time in milliseconds taken for the VNF-FG reconfiguration for each algorithm. A lower value is preferred. **Number of reconfigurations:** This metric measures the number of times a particular VNF-FG is reconfigured. Some algorithms allow a certain period of inconsistency, thus this metric measures the extra cost associated with more flexibility in terms of quick reconfigurations. A lower value is preferred. **Latency per operation:** This metric measures the time difference in milliseconds between the proposal for a VNF-FG reconfiguration and its actual reconfiguration. A high value means lower performance, thus, a lower value is preferred. **Overhead per data structure:** Measures the amount of information is stored to achieve a consistent state in bytes. This

metric is a cost associated with CRDT-based algorithms. Lower values are preferred. The overall goal with these metrics is to be able to compare the algorithms and evaluate the trade-offs of each to select the appropriate one for a given scenario.

6.4.3 Experiments

We consider all combinations of parameters for the experiments. Each combination is evaluated five times and we take the average values for each metric previously described. We only show relevant results to compare the three algorithms using only the concurrent scenarios as they are more representative for the algorithms' behavior. The proposed preventive variant is represented with a blue/circle line, the proposed preventive with a green/triangle line, and the standard with a red/square line. We present the results in Figures 6.6, 6.7, 6.8, 6.9. Next, we discussed in detail the results from our experiments.

6.4.4 Discussions

We evaluated our proposed algorithm compared to the ETSI standard [ETSI, NFVISG 2018] as it is the only relevant work aside ours that considers *shared* services and VNF-FGs, as shown in the second column of Table 6.3; the other

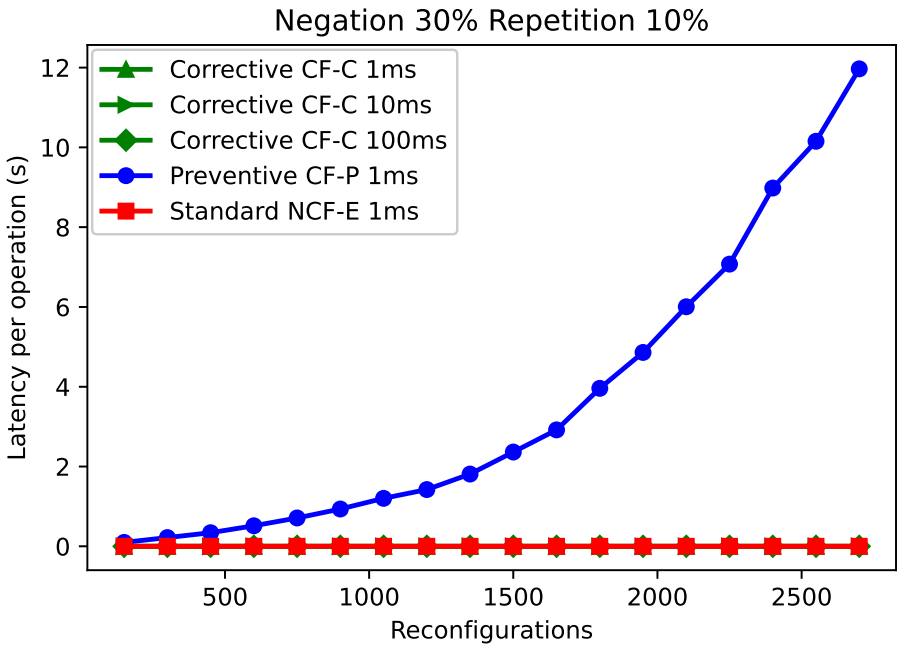

Figure 6.6: Latency per reconfiguration operation, lower is better. Both the corrective and standard algorithms are instantaneous, while the preventive variant must wait. For 3000 concurrent reconfigurations the average latency per operation is 12 seconds for preventive variant.

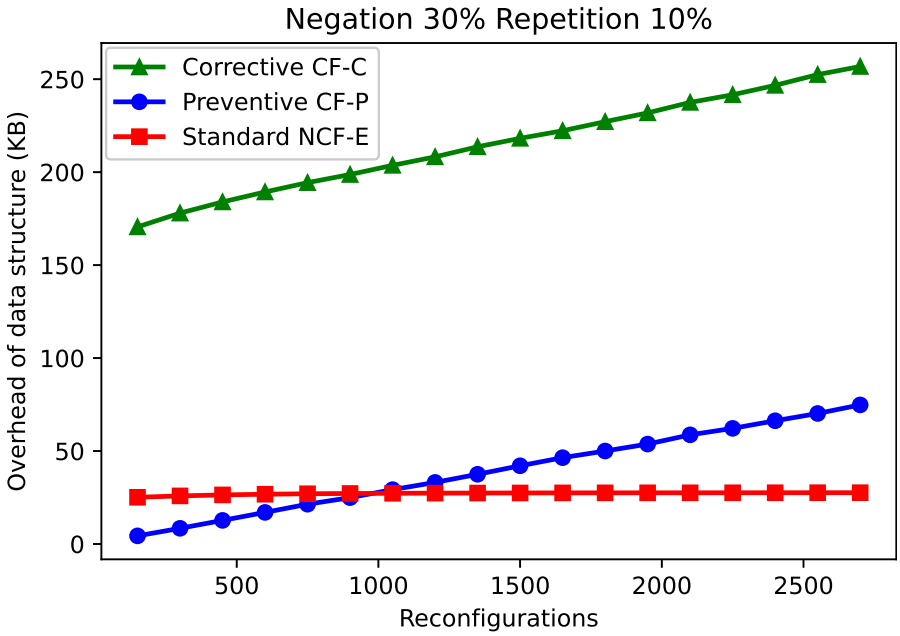

Figure 6.7: Data structure overhead per algorithm, lower is better. For small scenarios, the preventive variant offers better performance than the standard.

Table 6.3: Solutions for the VNF-FG reconfiguration with different functionalities. Our proposed variants offer all functionalities

Work	VNF-FG reconfigration	Shared VNF-FG	Non-Functional dependencies	Extra Reconfigurations	Inconsistencies
VNF-FG Extension [6,12]	Yes	No	No	No	N/A
ETSI Standard [13]	Yes	Yes	No	No	>0
Preventive Variant (CF-P)	**Yes**	**Yes**	**Yes**	**No**	**0**
Corrective Variant (CF-C)	**Yes**	**Yes**	**Yes**	**Yes**	**0**

works [Houidi 2020, Khebbache 2018] do not. Both of our proposed variants, unlike the standard [ETSI, NFVISG 2018], support non-functional dependencies that can negate on-going reconfigurations, as shown in the third column of Table 6.3. Moreover, unlike the standard that has inconsistencies, **the variants of our proposed algorithm reconfigure the replicas of the VNF-FGs without inconsistencies**, as shown in the fifth column of Table 6.3. However, supporting non-functional dependencies without a coordination phase has a cost. Next, we detail the results and discuss these costs.

Figure 6.6 shows the latency per reconfiguration operation (i.e. the time needed to wait before reconfiguring a VNF-FG) with a fixed delay of 100ms, a probability to negate of 30%, and probability of repetition of 10%. The preventive variant behaves worse than the standard and the corrective variant which applies the reconfiguration when it receives the reconfiguration instruction. In 3000 concurrent reconfigurations, the average latency per operations is 12 seconds for the preventive algorithm. While it might seem a high number compared to the corrective and standard, this waiting time is lower compared to the latency per transactions of consensus solutions (e.g 10 minutes

per transaction [Hao 2018]). The behavior presented in Figure 6.6 is representative for all the parameters' combinations.

Figure 6.7 shows the amount of information stored by each algorithm. For this metric, we only show a single delay of 100ms, probability to negate of 30%, and probability of repetition 10%. The corrective variant gets the worst performance with higher memory than the others. For small scenarios, the preventive variant has a smaller memory footprint than the standard algorithm; however, as the number of reconfigurations increases, the preventive behaves worst.

Figure 6.8 shows the messages sent to resolve conflicts between the orchestrators. The preventive variant got the worst performance of all. The corrective algorithm sits in the middle of the preventive and standard algorithms. However, with a high number of negations, the corrective algorithm behaves like the preventive. The delay affects the corrective variant, as shown by the widening gap between it and the standard algorithm in each line with delay from 1, 10, 100ms. This means that in the worst-case scenario, the corrective variant behaves like the preventive in terms of exchanging messages.

Figure 6.9 shows the number of extra reconfigurations per algorithm. Here the clear winner is the preventive algorithm getting zero extra reconfigurations, while the corrective behaves worst. Delay has the biggest impact on the correct algorithm, as shown by the three different lines of 1ms, 10ms, and 100ms in Figure 6.9. With delay of 1ms,

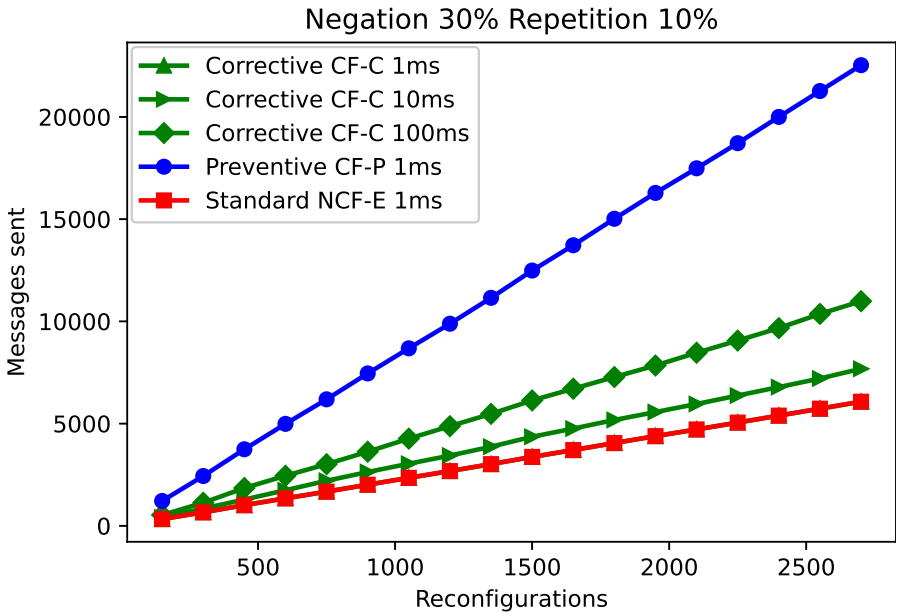

Figure 6.8: Messages sent by the algorithms in different scenarios, lower is better. The corrective algorithm is more sensitive to parameters. For scenarios where negation and delay are low, it behaves like the standard algorithm. Otherwise, it behaves like the preventive variant as the gap widens between 1 to 100ms.

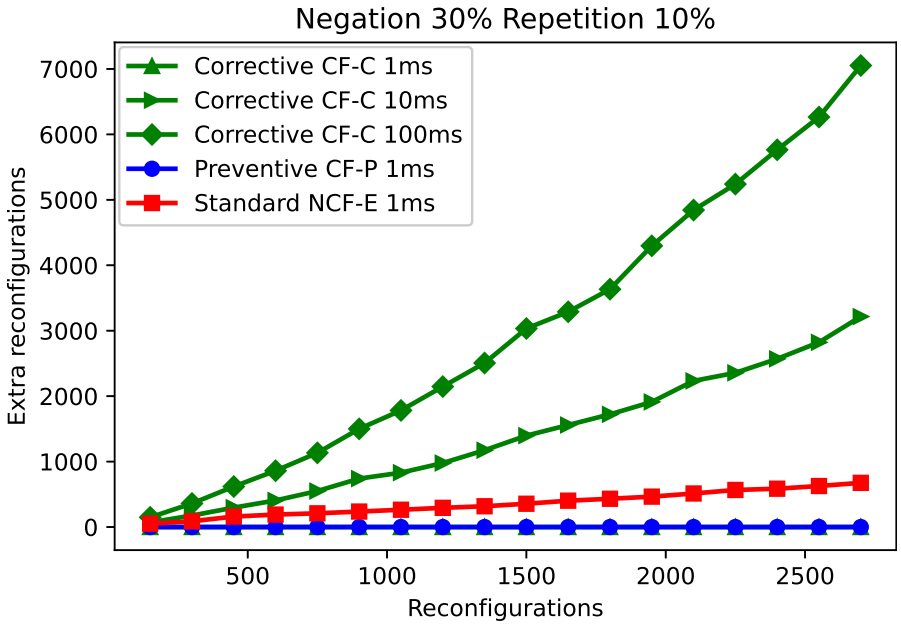

Figure 6.9: Extra reconfigurations done by each algorithm, lower is better. The preventive variant is the winner with zero extra reconfigurations. The corrective is sensitive to the delay, as shown by gap between 1-100ms.

the corrective and standard algorithm behave the same; this changes with higher delays, as the corrective algorithm behaves worst like with the number of messages sent. Since these two metrics relate with each other, Figures 6.8 and 6.9 show the same behavior.

Based on these results, we provide a better scope on which applications fit better the proposed variants. For critical applications, such as distributed resource-constrained federations, where consistency is a priority, extra reconfigurations are resource expensive, memory is limited, and latency is not a problem, service providers should select the preventive variant over the corrective. Service providers would use the corrective variant in more specialized environments where speed is king, memory is plenty, reconfiguration (in terms of resource consumption) is cheap, and the probability to negate reconfigurations is low. Moreover, the corrective variant does not require to known in advance the number of orchestrators in the federation, thus, it is possible to use it in open-federations where new orchestrators can join and leave temporally.

6.5 Lessons learned and perspectives

In this chapter, we proposed a coordination-free VNF-FG reconfiguration algorithm for network function virtualization in a multi-domain federation. This work addresses the noticed limitations in the relevant literature. Indeed, when the existing reconfiguration approaches require a coordination phase for conflict resolution, our proposed algorithm

achieves consistent reconfiguration without this coordination step. Unlike the current state of the art, our algorithm supports non-functional dependencies which could negate and roll back reconfigurations. Thus, we extend the problem of consistent VNF-FG reconfiguration.

To support these non-functional dependencies, we presented two variants of our proposed algorithm to target different applications. For critical and resource-constrained applications, where doing extra reconfigurations is undesired, we proposed a preventive variant. For less stringent applications, we likewise proposed a corrective variant. We formally proved both variants reconfigure consistently VNF-FG replicas without coordination. Since supporting non-functional dependencies has an associated cost in terms of delay and message/memory overhead, we evaluated the performance of both variants compared to the state of the art VNF-FG reconfiguration algorithm. The preventive variant is stable and its performance is similar in different parameters. The corrective variant is sensitive to parameters. With low delays, it offers performance similar to the standard but without the coordination phase and more functionalities. With higher delays (>100ms), it behaves like the preventive variant. For future work, we would like to explore ways to reduce the costs in terms of latency and extra reconfigurations. Also, we will explore more data structures that allow supporting more operations for the VNF-FG. Despite the overhead costs, our proposed algorithm, unlike the state of the art, works on a more general consistent VNF-FG reconfiguration problem. We have proposed the first coordination-free orchestration algorithm for NFV.

Our proposed algorithm opens the possibilities for future research works on coordination-free approaches for NFV. The question pertaining to the existence of such approaches has gotten a positive answer. Yet, new questions remain open. In the next chapter, we describe the impact of our research and put it into perspective with the state of the art.

Chapter 7

Conclusions and perspectives

Service providers are interested in preventing inconsistencies when their orchestrators reconfigure VNF-based network services. If providers take no care, these inconsistencies increase costs to providers, as they leave the services in partial or total failure. For shared services, this cost compounds over the federation, since shared services contain external dependencies. To prevent this, we proposed that, by coordinating orchestrators via causal dependencies, orchestrators prevent inconsistencies after reconfiguring VNF-based network services. We also stated that the orchestrators can prevent inconsistencies with no coordination phase. After this research, we bring consistent service provisioning to the foreground.

7.1 Research findings

We have shown the current limits of algorithms while reconfiguring VNF-based network services. In this research, we found out the negative side-effects of migrating shared Virtual Network Functions (VNFs). If orchestrators change the location of the VNFs, they can disturb the services. To prevent unwanted side-effects, we proposed coordinating the orchestrators so they can accept or reject migrating the shared VNF proposed by external orchestrators. By doing so, our algorithm got fewer inconsistencies than the classical heuristic algorithms.

We also found out that the grant messages, sent among orchestrators when they coordinate, capture some relations among the lifecycle tasks; however, they miss some of them. Inconsistencies can still arrive because the grants miss dependency relations. To prevent this, we identified the conditions required to create inconsistency patterns. Thus, we capture the grants' undetected patterns. We analyzed such patterns both in the time and event perspective. By generalizing the conditions, we could prevent inconsistencies while reconfiguring VNF-based network services.

Another important finding is the first coordination-free orchestration algorithm. We showed it is possible to achieve consistent VNF-FG reconfiguration without coordinating the orchestrators. Thus, opening a new research direction in service orchestration. Unlike all the previous works in the literature, including ours, this last contribution prevents inconsistencies without involving the classical trade-off between performance and consistency guarantees. We also consider the orchestrators can negate ongoing grants. Currently, the European Telecommunications Standards Institute (ETSI) NFV standard

partially supports negations, as all the orchestrators must agree before they reconfigure the service. The previous results highlight the implications of this research.

7.2 Research implications

We have identified three implications. The first one considers the limits of coordinating orchestrators through grants. Works that rely on such grants need to consider preventing the conditions that bring inconsistencies. Otherwise, they will trigger redundant tasks, wasting resources and time. Works that coordinate orchestrators only using grants need to reevaluate their approach. One way they can prevent inconsistency patterns is by removing one condition identified in this research.

The second implication relates to migrating shared VNFs. Before our research, most works considered dedicated services. However, after our research, service providers must consider composite services with external dependencies. This means they must coordinate to prevent unwanted effects. The last remains challenging, as many of the works only try to optimize the cost of migrating shared VNFs. However, by coordinating orchestrators, the number of inconsistencies reduces, as shown by our results.

Another important implication is the adoption of service providers for coordination-free approaches. The providers will study coordinate-free approaches to orchestrate VNF-based network services. Until now, all works considered coordinating orchestrators to execute VNF-based network services. This research shows how orchestrators reconfigure VNF-Forwarding Graphs with no coordination phase. They achieve consistent outcomes despite the possibility of negating ongoing reconfigurations. Next, we evaluate the limitations of this research. We put them in perspective by comparing them to the alternatives to handle these limitations.

7.3 Research limitations

The first limitation was considering close federations. They do not allow new participants to enter, ensuring trustful providers. However, they limit the reach of providers. Service providers do not always cooperate and compete with others. Orchestrators can apply proof of work (or another form of zero-knowledge proof) to authenticate the provider's shared information. However, such proofs introduce overhead as the orchestrators need to compute them. We require more research to establish the trade-offs between these approaches and the ones considered for this research. The second limitation is the type of VNFs used, namely prototype VNFs. While they give a perspective on performing our proposed algorithms, providers require more realistic VNFs. This will allow them to better understand the algorithm's trade-offs. Next, we develop some open questions that appear after this research.

7.4 Open questions

We focus on four open questions. The first one is how the algorithms we proposed for dependent VNF-based network services would perform with stateful VNFs. In our research, we consider stateless VNFs for ease of the approach; unless when migrating shared VNFS. Managing the state of VNFs has an associated overhead and introduces new dependencies. This means updating the solutions. For example, if it is possible to manage stateful VNFs using a coordination-free approach, the data structures will be likely changed to reflect the new VNFs' type.

The second question asks whether the approach we considered for the scaling task also applies to other tasks in the lifecycle. Indeed, according to the ETSI standard, some of these tasks require sending grant messages; however, it is still unknown if the inconsistency patterns we found for scaling can apply to these other tasks. For example, the standard considers that for healing and ending tasks; the orchestrators require sending grants. It is possible that the same analysis done for the reconfiguration tasks is valid. Orchestrators also would prevent one condition to ensure consistent service termination.

The third one asks how to extend the VNF-Forwarding Graph. In our work, we only consider updating this graph by chaining the values of connection points and classifying rules. Adding new services and VNFs is still an open question with a coordination-free approach. Some works consider this, but they rely on single-domain orchestration or do not consider the inconsistencies created by adding new services. This means that novel Conflict-free replicated data type (CRDT) data structures need to be proposed to add new services, and by extension, VNFs. For the fourth one, we ask what other problems in the provisioning of network services under distributed multi-domain federations can orchestrators achieve by coordination-free approaches. The first approach considers the current lifecycle tasks. The second approach relaxes the constraints of these tasks to support coordination-free approaches. For example, the problem of reconfiguring VNF-based services can be limited to only updating integer values of a service descriptor. Thus, orchestrators can deploy services using existing CRDTs. However, not all problems can be solved in a coordination-free approach. Next, we identify the future work by considering the limits, open questions, and our assumptions.

7.5 Future work

We start the discussion with our two considered assumptions for the research. First, we considered trustful providers to always provide accurate service information. Service providers can lie to other orchestrators. To handle adverse providers, some works in the literature consider competitive orchestrators who maximize their revenue. This makes these works more robust to failure; however, their overhead increases. This creates another trade-off between ease of use and complexity. Nowadays, whether a coordinated-free approach is valid under these assumptions remains unanswered. Thus, we make the

case for future work that will focus on adversarial networks and environments, especially with the popular blockchain approaches in recent NFV works. Second, we also assumed a well-defined and standard interface so orchestrators communicate. In a certain way, restful protocols can do this; yet interoperability remains unaddressed. Currently, no standard interface exists. This is one reason we implemented a federation from scratch for the orchestrators by following the ETSI standard. Based on the review of the literature for multi-domain architecture and the new uses cases introduced by network slicing, we think the future works and the ETSI standard will focus on this agnostic interface to coordinate orchestrators.

The results also would point to new directions to orchestrate VNF-based network services. The most promising one is the use of coordination-free algorithms. They enable federating geo-scale VNF-based network services without worrying about performance issues. Coordination-free algorithms bypass a consensus phase, unlike current reconfiguring algorithms. Yet, the question remains open about which other tasks coordination-free approaches can solve. We believe the future works that reconfigure federated VNF-based network services will explore new algorithms. Because of the memory overhead associated with keeping track of causal history, we think other works will work on optimizing data structures for the coordination-free approaches. Unsupervised learning techniques for reconfiguring VNF-based service promises interesting results too. The state of the art for the single domain has many examples of works that integrated machine learning and placing VNFs and network services. However, the NFV literature has not fully explored distributed machine learning approaches. We think more works will integrate these techniques into multi-domain federations. Having described the broad perspectives and future work, we put into perspective the research's impact.

Our research fills the gap in the lack of theoretical results for consistently reconfiguring VNF-based network services. We have shown the correctness of our algorithms both with experimental and theoretical analysis. Unlike most works in the literature that focus on experimental results, we generalize our results and present theoretical results. Indeed, by focusing on formal definitions, we discovered it was possible to reconfigure VNF-based network services without a coordination phase. Thus, we have established a new paradigm to orchestrate VNF-based network services, namely the coordination-free approach. Before our research, the literature did not consider this paradigm. Next, we detail the published articles and the ones being under revision at the time of this research.

7.6 Published and submitted articles

- Cisneros, J. C., Yangui, S., Pomares Hernandez, S. E., Perez Sansalvador, J. C., & Drira, K. (2020). Coordination Algorithm for Migration of Shared VNFs in Federated Environments. 2020 6th IEEE Conference on Network Softwarization (NetSoft), 252–256. https://doi.org/10.1109/NetSoft48620.2020.9165333

- Towards Consistent VNF Forwarding Graph Reconfiguration in Multi-domain Environments (Accepted for the IEEE CLOUD conference).
- VNF-Based Network Service Consistent Reconfiguration in Multi-domain Federations: A Distributed Approach (Accepted in the Journal of Network and Computer Applications).
- Coordination-free Multi-domain NFV Orchestration for Consistent VNF-Forwarding Graph Reconfiguration (Submitted to TNSM).

Bibliography

[3GPP 2017] 3GPP. *Study on management and orchestration of network slicing for next generation network*. 2017. (Cited in page 8.)

[Abdelwahab 2016] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani and Taieb Znati. *Network function virtualization in 5G*. IEEE Communications Magazine, vol. 54, no. 4, pages 84–91, apr 2016. (Cited in page 8.)

[Abu-Lebdeh 2017] Mohammad Abu-Lebdeh, Diala Naboulsi, Roch Glitho and Constant Wette Tchouati. *On the Placement of VNF Managers in Large-Scale and Distributed NFV Systems*. IEEE Transactions on Network and Service Management, vol. 14, no. 4, pages 875–889, dec 2017. (Cited in page 32.)

[Adamuz-Hinojosa 2018] Oscar Adamuz-Hinojosa, Jose Ordonez-Lucena, Pablo Ameigeiras, Juan J. Ramos-Munoz, Diego Lopez and Jesus Folgueira. *Automated Network Service Scaling in NFV: Concepts, Mechanisms and Scaling Workflow*. IEEE Communications Magazine, vol. 56, no. 7, pages 162–169, jul 2018. (Cited in page 50.)

[Alam 2020] Iqbal Alam, Kashif Sharif, Fan Li, Zohaib Latif, M M Karim, Sujit Biswas, Boubakr Nour and Yu Wang. *A Survey of Network Virtualization Techniques for Internet of Things Using SDN and NFV*. ACM Comput. Surv., vol. 53, no. 2, apr 2020. (Cited in page 8.)

[Anh Quang 2020] P T Anh Quang, Y Hadjadj-Aoul and A Outtagarts. *Evolutionary Actor-Multi-Critic Model for VNF-FG Embedding*. In 2020 IEEE 17th Annual Consumer Communications Networking Conference (CCNC), pages 1–6, jan 2020. (Cited in pages 80 and 87.)

[Antevski 2020] Kiril Antevski and Carlos J. Bernardos. *Federation of 5G services using distributed ledger technologies †*. Internet Technology Letters, no. 856709, pages 1–6, 2020. (Cited in page 1.)

[Arteaga 2017] Carlos Hernan Tobar Arteaga, Fulvio Risso and Oscar Mauricio Caicedo Rendon. *An adaptive scaling mechanism for managing performance variations in network functions virtualization: A case study in an NFV-based EPC*. In 2017 13th International Conference on Network and Service Management (CNSM), volume 2018-Janua, pages 1–7. IEEE, nov 2017. (Cited in page 50.)

[Bailis 2013] Peter Bailis and Ali Ghodsi. *Eventual Consistency Today: Limitations, Extensions, and Beyond*. Queue, vol. 11, no. 3, pages 20–32, mar 2013. (Cited in pages 25, 26, and 101.)

[Barakabitze 2020] Alcardo Alex Barakabitze, Arslan Ahmad, Rashid Mijumbi and Andrew Hines. *5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges.* Computer Networks, vol. 167, page 106984, 2020. (Cited in pages 1 and 7.)

[Baranda Hortiguela 2020] Jorge Baranda Hortiguela, Josep Mangues-Bafalluy, Ricardo Martinez, Luca Vettori, Kiril Antevski, Carlos J Bernardos and Xi Li. *Realizing the Network Service Federation Vision: Enabling Automated Multidomain Orchestration of Network Services.* IEEE Vehicular Technology Magazine, vol. 15, no. 2, pages 48–57, jun 2020. (Cited in page 15.)

[Baranda 2020] Jorge Baranda, Josep Mangues-Bafalluy, Luca Vettori and Ricardo Martínez. *Scaling composite NFV-network services.* In Proceedings of the International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc), pages 307–308, 2020. (Cited in page 52.)

[Bari 2015] Md. Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed and Raouf Boutaba. *On orchestrating virtual network functions.* In 2015 11th International Conference on Network and Service Management (CNSM), pages 50–56. IEEE, nov 2015. (Cited in page 32.)

[Bhamare 2016] Deval Bhamare, Raj Jain, Mohammed Samaka and Aiman Erbad. *A survey on service function chaining.* Journal of Network and Computer Applications, vol. 75, pages 138–155, 2016. (Cited in page 7.)

[Boudries 2019] Fouzia Boudries, Samia Sadouki and Abdelkamel Tari. *A bio-inspired algorithm for dynamic reconfiguration with end-to-end constraints in web services composition.* Service Oriented Computing and Applications, vol. 13, no. 3, pages 251–260, 2019. (Cited in page 51.)

[Bouras 2017] Christos Bouras, Anastasia Kollia and Andreas Papazois. *SDN & NFV in 5G: Advancements and challenges.* In 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), pages 107–111. IEEE, mar 2017. (Cited in pages 10 and 52.)

[Callegati 2015] Franco Callegati, Walter Cerroni, Chiara Contoli and Giuliano Santandrea. *Implementing dynamic chaining of Virtual Network Functions in OpenStack platform.* In 2015 17th International Conference on Transparent Optical Networks (ICTON), pages 1–4, jul 2015. (Cited in page 11.)

[Cao 2017] Jiuyue Cao, Yan Zhang, Wei An, Xin Chen, Jiyan Sun and Yanni Han. *VNF-FG design and VNF placement for 5G mobile networks.* Science China Information Sciences, vol. 60, no. 4, page 040302, 2017. (Cited in page 11.)

[Checko 2015] Aleksandra Checko, Henrik L Christiansen, Ying Yan, Lara Scolari, Georgios Kardaras, Michael S Berger and Lars Dittmann. *Cloud RAN for Mobile Networks—A Technology Overview.* IEEE Communications Surveys Tutorials, vol. 17, no. 1, pages 405–426, 2015. (Cited in pages 1 and 7.)

[Chen 2010] Xi Chen, Huaxin Zeng and Tao Wu. *Decentralized Orchestration with Local Centralized Orchestration for Composite Web Services.* In 2010 International Conference on Parallel and Distributed Computing, Applications and Technologies, pages 255–260. IEEE, dec 2010. (Cited in page 51.)

[Cisneros 2020a] J.C. Cisneros, S. Yangui, S.E. Pomares Hernandez, J.C. Perez Sansalvador and K. Drira. *Coordination algorithm for migration of shared VNFs in federated environments.* In Proceedings of the 2020 IEEE Conference on Network Softwarization: Bridging the Gap Between AI and Network Softwarization, NetSoft 2020, 2020. (Cited in page 8.)

[Cisneros 2020b] Josue Castaneda Cisneros, Sami Yangui, Saul E. Pomares Hernandez, Julio Cesar Perez Sansalvador and Khalil Drira. *Coordination Algorithm for Migration of Shared VNFs in Federated Environments.* In 2020 6th IEEE Conference on Network Softwarization (NetSoft), pages 252–256. IEEE, jun 2020. (Cited in page 43.)

[Clark 2005] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt and Andrew Warfield. *Live migration of virtual machines.* In Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2, pages 273–286, 2005. (Cited in pages 32 and 33.)

[Dang 2020] Xuan Thuy Dang and Fikret Sivrikaya. *A Lightweight Policy-aware Broker for Multi-domain Network Slice Composition.* 2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops, ICIN 2020, no. Icin, pages 123–130, 2020. (Cited in page 2.)

[Dieye 2018] M. Dieye, S. Ahvar, J. Sahoo, E. Ahvar, R. Glitho, H. Elbiaze and N. Crespi. *CPVNF: Cost-Efficient Proactive VNF Placement and Chaining for Value-Added Services in Content Delivery Networks.* IEEE Transactions on Network and Service Management, vol. 15, no. 2, pages 774–786, 2018. (Cited in page 82.)

[Dieye 2020] M. Dieye, W. Jaafar, H. Elbiaze and R. H. Glitho. *Market Driven Multidomain Network Service Orchestration in 5G Networks.* IEEE Journal on Selected Areas in Communications, vol. 38, no. 7, pages 1417–1431, 2020. (Cited in page 82.)

[Duan 2017] Jingpu Duan, Chuan Wu, Franck Le, Alex X Liu and Yanghua Peng. *Dynamic Scaling of Virtualized, Distributed Service Chains: A Case Study of IMS*. IEEE Journal on Selected Areas in Communications, vol. 35, no. 11, pages 2501–2511, nov 2017. (Cited in page 50.)

[el houda Nouar 2021] Nour el houda Nouar, Sami Yangui, Noura Faci, Khalil Drira and Saïd Tazi. *A Semantic virtualized network functions description and discovery model*. Computer Networks, vol. 195, no. March 2020, page 108152, aug 2021. (Cited in pages 10 and 51.)

[Eramo 2017a] Vincenzo Eramo, Mostafa Ammar and Francesco Giacinto Lavacca. *Migration Energy Aware Reconfigurations of Virtual Network Function Instances in NFV Architectures*. IEEE Access, vol. 5, pages 4927–4938, 2017. (Cited in pages 2, 16, 32, and 49.)

[Eramo 2017b] Vincenzo Eramo and Francesco Giacinto Lavacca. *Definition and Evaluation of Cold Migration Policies for the Minimization of the Energy Consumption in NFV Architectures*. In Alessandro Piva, Ilenia Tinnirello and Simone Morosi, editors, Digital Communication. Towards a Smart and Secure Future Internet, pages 45–60, Cham, 2017. Springer International Publishing. (Cited in page 32.)

[Eramo 2019a] V Eramo, A Cianfrani, T Catena, M Polverini and F.G. Lavacca. *Reconfiguration of Cloud and Bandwidth Resources in NFV Architectures Based on Segment Routing Control/Data Plane*. In 2019 21st International Conference on Transparent Optical Networks (ICTON), pages 1–5. IEEE, jul 2019. (Cited in page 16.)

[Eramo 2019b] Vincenzo Eramo and Francesco Giacinto Lavacca. *Proposal and Investigation of a Reconfiguration Cost Aware Policy for Resource Allocation in Multi-Provider NFV Infrastructures Interconnected by Elastic Optical Networks*. Journal of Lightwave Technology, vol. 37, no. 16, pages 4098–4114, 2019. (Cited in page 80.)

[ETSI, NFVISG 2014] ETSI, NFVISG. *GS NFV-MAN 001 V1. 1.1 Network Function Virtualisation (NFV); Management and Orchestration*, 2014. (Cited in pages 3, 80, and 85.)

[ETSI, NFVISG 2016] ETSI, NFVISG. *ETSI GS NFV-IFA 010, V2.1.1 Network Functions Virtualisation (NFV);Management and Orchestration; Functional requirements specification*, 2016. (Cited in pages 11, 13, and 52.)

[ETSI, NFVISG 2018] ETSI, NFVISG. *ETSI GR NFV-IFA 028 V3.1.1 Release 3; Management and Orchestration; Report on Architecture Options to Support Multiple Administrative Domains*, 2018. (Cited in pages 2, 3, 15, 21, 52, 53, 63, 68, 81, 87, 119, and 120.)

[ETSI, NFVISG 2020] ETSI, NFVISG. *ETSI GS NFV-IFA 010 V3.4.1 Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Functional requirements specification*, 2020. (Cited in pages 10 and 80.)

[ETSI 2014] NFVISG ETSI. *Network Functions Virtualisation (NFV); Architectural Framework*, 2014. (Cited in pages 1, 7, 13, 80, and 84.)

[ETSI 2017] NFVISG ETSI. *Network Functions Virtualisation (NFV) Release 2; Management and Orchestration; Network Service Templates Specification*, 2017. (Cited in page 11.)

[ETSI 2018a] NFVISG ETSI. *Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Multiple Administrative Domain Aspect Interfaces Specification*, 2018. (Cited in pages 14, 17, 44, and 63.)

[ETSI 2018b] NFVISG ETSI. *Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Report on Management and Connectivity for Multi-Site Services*, 2018. (Cited in page 7.)

[ETSI 2018c] NFVISG ETSI. *Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Report on Os-Ma-Nfvo reference point - application and service management use cases and recommendations*, 2018. (Cited in page 12.)

[ETSI 2019a] NFVISG ETSI. *Network Functions Virtualisation (NFV) Release 3; Reliability; Report on NFV Resiliency for the Support of Network Slicing*, 2019. (Cited in page 16.)

[ETSI 2019b] NFVISG ETSI. *Network Functions Virtualisation (NFV) Release 3;Management and Orchestration; Network Service Templates Specification*, 2019. (Cited in pages 11 and 92.)

[ETSI 2020a] NFVISG ETSI. *Network Functions Virtualisation (NFV) Release 3;Protocols and Data Models; RESTful protocols specification for the Or-Vnfm Reference Point*, 2020. (Cited in page 17.)

[ETSI 2020b] NFVISG ETSI. *Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV*, 2020. (Cited in pages 10, 12, 13, and 52.)

[Fidge 1988] Colin J Fidge. *Timestamps in Message-Passing Systems That Preserve the Partial Ordering*. In Proc. 11th Austral. Comput. Sci. Conf. (ACSC '88), pages 56–66, 1988. (Cited in page 64.)

[Forecast 2019] GMDT Forecast. *Cisco visual networking index: global mobile data traffic forecast update, 2017–2022*. Update, vol. 2017, page 2022, 2019. (Cited in pages 1 and 7.)

[Frick 2018] G. Frick, A. Lehmann, G. Frick, A. Paguem Tchinda and B. Ghita. *Distributed NFV Orchestration in a WMN-Based Disaster Network*. International Conference on Ubiquitous and Future Networks, ICUFN, vol. 2018-July, pages 168–173, 2018. (Cited in page 1.)

[Gedia 2018] Dewang Gedia and Levi Perigo. *Performance Evaluation of SDN-VNF in Virtual Machine and Container*. In 2018 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pages 1–7. IEEE, nov 2018. (Cited in page 10.)

[Ghorbani 2014] Soudeh Ghorbani, Cole Schlesinger, Matthew Monaco, Eric Keller, Matthew Caesar, Jennifer Rexford and David Walker. *Transparent, Live Migration of a Software-Defined Network*. In Proceedings of the ACM Symposium on Cloud Computing, SOCC '14, pages 1–14, New York, NY, USA, 2014. Association for Computing Machinery. (Cited in page 32.)

[Gil Herrera 2016] Juliver Gil Herrera and Juan Felipe Botero. *Resource Allocation in NFV: A Comprehensive Survey*. IEEE Transactions on Network and Service Management, vol. 13, no. 3, pages 518–532, sep 2016. (Cited in pages 1, 7, 11, and 38.)

[Halpern 2015] Joel Halpern, Carlos Pignataro and Others. *Service function chaining (sfc) architecture*. In RFC 7665. 2015. (Cited in page 8.)

[Han 2018] Yoonseon Han, Thomas Vachuska, Ali Al-Shabibi, Jian Li, Huibai Huang, William Snow and James Won-Ki Hong. *ONVisor: Towards a scalable and flexible SDN-based network virtualization platform on ONOS*. International Journal of Network Management, vol. 28, no. 2, page e2012, 2018. (Cited in page 83.)

[Hao 2018] Yue Hao, Yi Li, Xinghua Dong, Li Fang and Ping Chen. *Performance Analysis of Consensus Algorithm in Private Blockchain*. In 2018 IEEE Intelligent Vehicles Symposium (IV), pages 280–285. IEEE, jun 2018. (Cited in pages 3 and 121.)

[Hawilo 2017] Hassan Hawilo, Manar Jammal and Abdallah Shami. *Orchestrating network function virtualization platform: Migration or re-instantiation?* In 2017 IEEE 6th International Conference on Cloud Networking (CloudNet), pages 1–6, 2017. (Cited in page 32.)

[Herrera 2016] Juliver Gil Herrera and Juan Felipe Botero. *Resource allocation in NFV: A comprehensive survey*. IEEE Transactions on Network and Service Management, vol. 13, no. 3, pages 518–532, 2016. (Cited in page 82.)

[Hnětynka 2006] Petr Hnětynka and František Plášil. *Dynamic Reconfiguration and Access to Services in Hierarchical Component Models*. In Ian Gorton, George T

Heineman, Ivica Crnković, Heinz W Schmidt, Judith A Stafford, Clemens Szyperski and Kurt Wallnau, editors, Component-Based Software Engineering, pages 352–359, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited in page 51.)

[Houidi 2020] Omar Houidi, Oussama Soualah, Wajdi Louati and Djamal Zeghlache. *Dynamic VNF Forwarding Graph Extension Algorithms.* IEEE Transactions on Network and Service Management, vol. 17, no. 3, pages 1389–1402, sep 2020. (Cited in pages 80, 81, 86, 87, 99, and 120.)

[Howard 2020] Heidi Howard and Richard Mortier. *Paxos vs Raft: Have we reached consensus on distributed consensus?* Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2020, pages 8–10, 2020. (Cited in page 25.)

[Hu 2020] Guangwu Hu, Qing Li, Shuo Ai, Tan Chen, Jingpu Duan and Yu Wu. *A proactive auto-scaling scheme with latency guarantees for multi-tenant NFV cloud.* Computer Networks, vol. 181, no. September, page 107552, nov 2020. (Cited in page 50.)

[Israel 2019] A Israel, AT Sepúlveda, A Reid, F Vicens, FJR Salguero, GG de Blas, G Lavado, M Shuttleworth, M Harper, M Marchetti, Vilata R, Almagia S*et al.* *OSM Release FIVE Technical Overview* , January 2019. (Cited in page 62.)

[Jia 2018] Yongzheng Jia, Chuan Wu, Zongpeng Li, Franck Le and Alex Liu. *Online Scaling of NFV Service Chains Across Geo-Distributed Datacenters.* IEEE/ACM Transactions on Networking, vol. 26, no. 2, pages 699–710, 2018. (Cited in page 50.)

[Joshi 2020] Kalpana D. Joshi and Kotaro Kataoka. *pSMART: A lightweight, privacy-aware service function chain orchestration in multi-domain NFV/SDN.* Computer Networks, vol. 178, no. March, page 107295, 2020. (Cited in page 2.)

[Katsalis 2016] Kostas Katsalis, Navid Nikaein and Andy Edmonds. *Multi-Domain Orchestration for NFV: Challenges and Research Directions.* In 2016 15th International Conference on Ubiquitous Computing and Communications and 2016 International Symposium on Cyberspace and Security (IUCC-CSS), pages 189–195. IEEE, dec 2016. (Cited in pages 2, 9, 14, 43, and 51.)

[Kattepur 2013] Ajay Kattepur, Nikolaos Georgantas and Valerie Issarny. *QoS Composition and Analysis in Reconfigurable Web Services Choreographies.* In 2013 IEEE 20th International Conference on Web Services, pages 235–242. IEEE, jun 2013. (Cited in page 51.)

[Kawashima 2016] Ryota Kawashima and Hiroshi Matsuo. *vNFChain: A VM-Dedicated Fast Service Chaining Framework for Micro-VNFs.* In 2016 Fifth European

Workshop on Software-Defined Networks (EWSDN), pages 13–18, oct 2016. (Cited in page 10.)

[Kazhamiakin 2006] Raman Kazhamiakin and Marco Pistore. *Choreography Conformance Analysis: Asynchronous Communications and Information Alignment.* In Mario Bravetti, Manuel Núñez and Gianluigi Zavattaro, editors, Web Services and Formal Methods, pages 227–241, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. (Cited in page 51.)

[Khebbache 2018] S Khebbache, M Hadji and D Zeghlache. *Dynamic Placement of Extended Service Function Chains: Steiner-based Approximation Algorithms.* In 2018 IEEE 43rd Conference on Local Computer Networks (LCN), pages 307–310, oct 2018. (Cited in pages 81, 86, 87, and 120.)

[Kim 2016] Siri Kim, Yunjung Han and Sungyong Park. *An Energy-Aware Service Function Chaining and Reconfiguration Algorithm in NFV.* In 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), pages 54–59. IEEE, sep 2016. (Cited in page 16.)

[Kim 2018] Changwoo Kim, Yujeong Oh and Jaiyong Lee. *Latency-based graph selection manager for end-to-end network service on heterogeneous infrastructures.* In 2018 International Conference on Information Networking (ICOIN), pages 534–539. IEEE, jan 2018. (Cited in page 86.)

[Komarek 2017] Ales Komarek, Jakub Pavlik, Lubos Mercl and Vladimir Sobeslav. *VNF Orchestration and Modeling with ETSI MANO Compliant Frameworks.* In Olga Galinina, Sergey Andreev, Sergey Balandin and Yevgeni Koucheryavy, editors, Internet of Things, Smart Spaces, and Next Generation Networks and Systems, pages 121–131, Cham, 2017. Springer International Publishing. (Cited in page 38.)

[Kouchaksaraei 2019] Hadi Razzaghi Kouchaksaraei and Holger Karl. *Service Function Chaining Across OpenStack and Kubernetes Domains.* In Proceedings of the 13th ACM International Conference on Distributed and Event-Based Systems, DEBS '19, pages 240–243, New York, NY, USA, 2019. Association for Computing Machinery. (Cited in page 11.)

[Lamport 1978] Leslie Lamport. *Time, clocks, and the ordering of events in a distributed system.* Communications of the ACM, vol. 21, no. 7, pages 558–565, jul 1978. (Cited in page 24.)

[Le 2016] Nam Tuan Le, Mohammad Arif Hossain, Amirul Islam, Do-yun Kim, Young-June Choi and Yeong Min Jang. *Survey of Promising Technologies for 5G Networks.* Mobile Information Systems, vol. 2016, pages 1–25, 2016. (Cited in page 8.)

[Leite 2013] Leonardo A F Leite, Gustavo Ansaldi Oliva, Guilherme M Nogueira, Marco Aurélio Gerosa, Fabio Kon and Dejan S Milojicic. *A systematic literature review of service choreography adaptation.* Service Oriented Computing and Applications, vol. 7, no. 3, pages 199–216, sep 2013. (Cited in page 51.)

[Liu 2016] Libin Liu, Hong Xu, Zhixiong Niu, Peng Wang and Dongsu Han. *U-HAUL: Efficient State Migration in NFV.* In Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys '16, New York, NY, USA, 2016. Association for Computing Machinery. (Cited in page 32.)

[Liu 2017] Junjie Liu, Wei Lu, Fen Zhou, Ping Lu and Zuqing Zhu. *On Dynamic Service Function Chain Deployment and Readjustment.* IEEE Transactions on Network and Service Management, vol. 14, no. 3, pages 543–553, sep 2017. (Cited in page 16.)

[Liu 2020] Yi Liu, Hongqi Zhang, Dexian Chang and Hao Hu. *GDM: A General Distributed Method for Cross-Domain Service Function Chain Embedding.* IEEE Transactions on Network and Service Management, vol. 17, no. 3, pages 1446–1459, sep 2020. (Cited in page 12.)

[Luizelli 2015] Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol, Marinho Pilla Barcellos and Luciano Paschoal 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. IEEE, may 2015. (Cited in page 12.)

[Ma 2019] Lele Ma, Shanhe Yi, Nancy Carter and Qun Li. *Efficient Live Migration of Edge Services Leveraging Container Layered Storage.* IEEE Transactions on Mobile Computing, vol. 18, no. 9, pages 2020–2033, sep 2019. (Cited in page 32.)

[Malandrino 2019] Francesco Malandrino, Carla Fabiana Chiasserini, Gil Einziger and Gabriel Scalosub. *Reducing Service Deployment Cost Through VNF Sharing.* IEEE/ACM Transactions on Networking, vol. 27, no. 6, pages 2363–2376, dec 2019. (Cited in page 8.)

[Marandi 2014] Parisa Jalili Marandi, Samuel Benz, Fernando Pedonea and Kenneth P Birman. *The Performance of Paxos in the Cloud.* In 2014 IEEE 33rd International Symposium on Reliable Distributed Systems, pages 41–50. IEEE, oct 2014. (Cited in page 84.)

[Martini 2016] Barbara Martini and Federica Paganelli. *A Service-Oriented Approach for Dynamic Chaining of Virtual Network Functions over Multi-Provider Software-Defined Networks.* Future Internet, vol. 8, no. 2, 2016. (Cited in pages 7 and 9.)

[Mattos 2015] Diogo M F Mattos, Lyno Henrique G Ferraz and Otto Carlos M B Duarte. Virtual Machine Migration, chapter 3, pages 49–72. John Wiley & Sons, Inc, Hoboken, NJ, apr 2015. (Cited in page 32.)

[Mechtri 2016] Marouen Mechtri, Imen Grida Benyahia and Djamal Zeghlache. *Agile service manager for 5G.* Proceedings of the NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium, pages 1285–1290, 2016. (Cited in page 10.)

[Mijumbi 2016] Rashid Mijumbi, Joan Serrat, Juan-luis Gorricho, Steven Latre, Marinos Charalambides and Diego Lopez. *Management and orchestration challenges in network functions virtualization.* IEEE Communications Magazine, vol. 54, no. 1, pages 98–105, jan 2016. (Cited in page 7.)

[Mills 1991] D.L. Mills. *Internet time synchronization: the network time protocol.* IEEE Transactions on Communications, vol. 39, no. 10, pages 1482–1493, oct 1991. (Cited in page 56.)

[Moo-Mena 2007] Francisco Moo-Mena and Khalil Drira. *Reconfiguration of Web Services Architectures: A model-based approach.* In 2007 IEEE Symposium on Computers and Communications, pages 357–362. IEEE, jul 2007. (Cited in page 51.)

[Nadjaran Toosi 2019] Adel Nadjaran Toosi, Jungmin Son, Qinghua Chi and Rajkumar Buyya. *ElasticSFC: Auto-scaling techniques for elastic service function chaining in network functions virtualization-based clouds.* Journal of Systems and Software, vol. 152, pages 108–119, 2019. (Cited in page 50.)

[Nanda 2004] Mangala Gowri Nanda, Satish Chandra and Vivek Sarkar. *Decentralizing execution of composite web services.* ACM SIGPLAN Notices, vol. 39, no. 10, pages 170–187, oct 2004. (Cited in page 51.)

[Nguyen 2017] Van-Giang Nguyen, Anna Brunstrom, Karl-Johan Grinnemo and Javid Taheri. *SDN/NFV-Based Mobile Packet Core Network Architectures: A Survey.* IEEE Communications Surveys & Tutorials, vol. 19, no. 3, pages 1567–1602, 2017. (Cited in page 8.)

[Nobach 2017] Leonhard Nobach, Ivica Rimac, Volker Hilt and David Hausheer. *Statelet-Based Efficient and Seamless NFV State Transfer.* IEEE Transactions on Network and Service Management, vol. 14, no. 4, pages 964–977, dec 2017. (Cited in pages 32 and 33.)

[Okusanya 2019] Oluwadamilola Okusanya. *Consensus in Distributed Systems : RAFT vs CRDTs.* 2019. (Cited in page 3.)

[Open Source 2018] MANO Open Source. *OSM Release FOUR Technical Overview*, 2018. (Cited in page 7.)

[Pathan 2008] Mukaddim Pathan, Rajkumar Buyya and Athena Vakali. Content Delivery Networks: State of the Art, Insights, and Imperatives, pages 3–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. (Cited in page 82.)

[Peuster 2016] Manuel Peuster, Holger Karl and Steven van Rossem. *MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments*. In 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), pages 148–153. IEEE, nov 2016. (Cited in page 38.)

[Pham 2019] Quang Tran Anh Pham, Abbas Bradai, Kamal Deep Singh and Yassine Hadjadj-Aoul. *Multi-domain non-cooperative VNF-FG embedding: A deep reinforcement learning approach*. In IEEE Infocom 2019-IEEE International Conference on Computer Communications, 2019. (Cited in page 52.)

[Quang 2019a] Pham Tran Anh Quang, Abbas Bradai, Kamal Deep Singh and Yassine Hadjadj-Aoul. *Multi-domain non-cooperative VNF-FG embedding: A deep reinforcement learning approach*. In IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pages 886–891. IEEE, apr 2019. (Cited in pages 80 and 87.)

[Quang 2019b] Pham Tran Anh Quang, Abbas Bradai, Kamal Deep Singh, Gauthier Picard and Roberto Riggio. *Single and Multi-Domain Adaptive Allocation Algorithms for VNF Forwarding Graph Embedding*. IEEE Transactions on Network and Service Management, vol. 16, no. 1, pages 98–112, 2019. (Cited in pages 11, 81, 86, and 87.)

[Rahman 2020] Sabidur Rahman, Tanjila Ahmed, Minh Huynh, Massimo Tornatore and Biswanath Mukherjee. *Auto-Scaling Network Service Chains Using Machine Learning and Negotiation Game*. IEEE Transactions on Network and Service Management, vol. 17, no. 3, pages 1322–1336, sep 2020. (Cited in page 50.)

[Rankothge 2015] Windhya Rankothge, Jiefei Ma, Franck Le, Alessandra Russo and Jorge Lobo. *Towards making network function virtualization a cloud computing service*. In 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), pages 89–97, may 2015. (Cited in page 8.)

[Rosa 2015] Raphael Vicente Rosa, Mateus Augusto Silva Santos and Christian Esteve Rothenberg. *MD2-NFV: The case for multi-domain distributed network functions virtualization*. In 2015 International Conference and Workshops on Networked Systems (NetSys), pages 1–5. IEEE, mar 2015. (Cited in pages 14 and 83.)

[Salaün 2009] Gwen Salaün and Nima Roohi. *On Realizability and Dynamic Reconfiguration of Choreographies*, 2009. (Cited in page 51.)

[Samdanis 2018] Konstantinos Samdanis, Athul Prasad, Min Chen and Kai Hwang. *Enabling 5G Verticals and Services through Network Softwarization and Slicing.* IEEE Communications Standards Magazine, vol. 2, no. 1, pages 20–21, 2018. (Cited in page 101.)

[Saraiva de Sousa 2019] Nathan F. Saraiva de Sousa, Danny A. Lachos Perez, Raphael V. Rosa, Mateus A.S. Santos and Christian Esteve Rothenberg. *Network Service Orchestration: A survey.* Computer Communications, vol. 142-143, no. May, pages 69–94, jun 2019. (Cited in pages 1, 2, 8, and 87.)

[Sarrigiannis 2020] Ioannis Sarrigiannis, Kostas Ramantas, Elli Kartsakli, Prodromos-Vasileios Mekikis, Angelos Antonopoulos and Christos Verikoukis. *Online VNF Lifecycle Management in an MEC-Enabled 5G IoT Architecture.* IEEE Internet of Things Journal, vol. 7, no. 5, pages 4183–4194, may 2020. (Cited in page 50.)

[Schardong 2021] Frederico Schardong, Ingrid Nunes and Alberto Schaeffer-Filho. *NFV Resource Allocation: a Systematic Review and Taxonomy of VNF Forwarding Graph Embedding.* Computer Networks, vol. 185, no. July 2020, page 107726, 2021. (Cited in pages 80 and 85.)

[Shapiro 2011a] Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Europe, 2011. (Cited in pages 26 and 113.)

[Shapiro 2011b] Marc Shapiro, Nuno Preguiça, Carlos Baquero and Marek Zawirski. *Conflict-Free Replicated Data Types.* In Xavier Défago, Franck Petit and Vincent Villain, editors, Stabilization, Safety, and Security of Distributed Systems, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. (Cited in pages 25, 26, 105, 113, and 115.)

[Shin 2015] Myung-Ki Shin, Yunchul Choi, Hee Hwan Kwak, Sangheon Pack, Miyoung Kang and Jin-Young Choi. *Verification for NFV-enabled network services.* In 2015 International Conference on Information and Communication Technology Convergence (ICTC), pages 810–815. IEEE, oct 2015. (Cited in pages 8 and 50.)

[Soualah 2018] Oussama Soualah, Marouen Mechtri, Chaima Ghribi and Djamal Zeghlache. *A Green VNF-FG Embedding Algorithm.* In 2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft), pages 141–149. IEEE, jun 2018. (Cited in page 86.)

[Spinnewyn 2020] Bart Spinnewyn, Steven Latré and Juan Felipe Botero. *Delay-constrained NFV orchestration for heterogeneous cloud networks.* Computer Networks, vol. 180, page 107420, 2020. (Cited in page 86.)

[Stal 2006] M Stal. *Using architectural patterns and blueprints for service-oriented architecture*. IEEE Software, vol. 23, no. 2, pages 54–61, mar 2006. (Cited in page 9.)

[Subramanya 2021] Tejas Subramanya and Roberto Riggio. *Centralized and Federated Learning for Predictive VNF Autoscaling in Multi-Domain 5G Networks and beyond*. IEEE Transactions on Network and Service Management, vol. 18, no. 1, pages 63–78, 2021. (Cited in page 52.)

[Sugisono 2018] Koji Sugisono, Aki Fukuoka and Hirofumi Yamazaki. *Migration for VNF Instances Forming Service Chain*. In 2018 IEEE 7th International Conference on Cloud Networking (CloudNet), pages 1–3, oct 2018. (Cited in page 33.)

[Taleb 2019] Tarik Taleb, Ibrahim Afolabi, Konstantinos Samdanis and Faqir Zarrar Yousaf. *On Multi-Domain Network Slicing Orchestration Architecture and Federated Resource Control*. IEEE Network, vol. 33, no. 5, pages 242–252, sep 2019. (Cited in page 2.)

[Toka 2021] Laszlo Toka, Marton Zubor, Attila Korosi, George Darzanos, Ori Rottenstreich and Balazs Sonkoly. *Pricing games of NFV infrastructure providers*. Telecommunication Systems, vol. 76, no. 2, pages 219–232, feb 2021. (Cited in page 8.)

[Tomassilli 2018] A Tomassilli, F Giroire, N Huin and S Pérennes. *Provably Efficient Algorithms for Placement of Service Function Chains with Ordering Constraints*. In IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, pages 774–782, apr 2018. (Cited in page 32.)

[Tong 2020] Riming Tong, Siya Xu, Bo Hu, Jinghong Zhao, Lei Jin, Shaoyong Guo and Wenjing Li. *VNF Dynamic Scaling and Deployment Algorithm Based on Traffic Prediction*. In 2020 International Wireless Communications and Mobile Computing (IWCMC), pages 789–794. IEEE, jun 2020. (Cited in page 50.)

[Vaquero 2019] Luis M. Vaquero, Felix Cuadrado, Yehia Elkhatib, Jorge Bernal-Bernabe, Satish N. Srirama and Mohamed Faten Zhani. *Research challenges in nextgen service orchestration*. Future Generation Computer Systems, vol. 90, pages 20–38, 2019. (Cited in pages 1, 2, 24, and 52.)

[Vogels 2008] Werner Vogels. *Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs?Between Consistency and Availability*. Queue, vol. 6, no. 6, pages 14–19, oct 2008. (Cited in page 25.)

[Wang 2018] Gang Wang, Gang Feng, Tony Q S Quek and Shuang Qin. *On Fast Slice Reconfiguration*. In 2018 IEEE Global Communications Conference (GLOBECOM), pages 1–7. IEEE, dec 2018. (Cited in pages 16 and 49.)

[Wei Ren 2005] Wei Ren, R W Beard and E M Atkins. *A survey of consensus problems in multi-agent coordination.* In Proceedings of the 2005, American Control Conference, 2005., pages 1859–1864 vol. 3, jun 2005. (Cited in page 25.)

[Xia 2016] Jing Xia, Deming Pang, Zhiping Cai, Ming Xu and Gang Hu. *Reasonably Migrating Virtual Machine in NFV-Featured Networks.* In 2016 IEEE International Conference on Computer and Information Technology (CIT), pages 361–366. IEEE, dec 2016. (Cited in pages 27 and 32.)

[Xiao 2020] Y Xiao, N Zhang, W Lou and Y T Hou. *A Survey of Distributed Consensus Protocols for Blockchain Networks.* IEEE Communications Surveys Tutorials, vol. 22, no. 2, pages 1432–1465, 2020. (Cited in page 25.)

[Xu 2020] Ran Xu. *Proactive VNF Scaling with Heterogeneous Cloud Resources: Fusing Long Short-Term Memory Prediction and Cooperative Allocation.* Mathematical Problems in Engineering, vol. 2020, pages 1–10, jan 2020. (Cited in pages 50 and 52.)

[Yang Wang 2016] Yang Wang, Gaogang Xie, Zhenyu Li, Peng He and Kave Salamatian. *Transparent flow migration for NFV.* In 2016 IEEE 24th International Conference on Network Protocols (ICNP), pages 1–10. IEEE, nov 2016. (Cited in pages 31 and 33.)

[Yang 2018] Binxu Yang, Zichuan Xu, Wei Koong Chai, Weifa Liang, Daphne Tuncer, Alex Galis and George Pavlou. *Algorithms for Fault-Tolerant Placement of Stateful Virtualized Network Functions.* In 2018 IEEE International Conference on Communications (ICC), pages 1–7. IEEE, may 2018. (Cited in pages 16 and 49.)

[Yangui 2016] Sami Yangui, Roch H Glitho and Constant Wette. *Approaches to end-user applications portability in the cloud: A survey.* IEEE Communications Magazine, vol. 54, no. 7, pages 138–145, jul 2016. (Cited in page 51.)

[Yi 2018] Bo Yi, Xingwei Wang, Keqin Li, Sajal k. Das and Min Huang. *A comprehensive survey of Network Function Virtualization.* Computer Networks, vol. 133, pages 212–262, mar 2018. (Cited in pages 1, 8, 9, 32, and 51.)

[Yi 2020] Bo Yi, Xingwei Wang, Min Huang and Kexin Li. *Design and Implementation of Network-Aware VNF Migration Mechanism.* IEEE Access, vol. 8, pages 44346–44358, 2020. (Cited in page 2.)

[Yong Li 2015] Yong Li and Min Chen. *Software-Defined Network Function Virtualization: A Survey.* IEEE Access, vol. 3, pages 2542–2553, 2015. (Cited in page 1.)

[Zeng 2016] M Zeng, W Fang and Z Zhu. *Orchestrating Tree-Type VNF Forwarding Graphs in Inter-DC Elastic Optical Networks.* Journal of Lightwave Technology, vol. 34, no. 14, pages 3330–3341, jul 2016. (Cited in page 86.)

[Zheng 2019] Gao Zheng, Anthony Tsiopoulos and Vasilis Friderikos. *Dynamic VNF Chains Placement for Mobile IoT Applications.* In 2019 IEEE Global Communications Conference (GLOBECOM), pages 1–6. IEEE, dec 2019. (Cited in pages 80 and 85.)

[Zou 2018] Jian Zou, Wei Li, Jingyu Wang, Qi Qi and Haifeng Sun. *NFV Orchestration and Rapid Migration Based on Elastic Virtual Network and Container Technology.* In 2018 IEEE International Conference on Information Communication and Signal Processing (ICICSP), pages 6–10. IEEE, sep 2018. (Cited in page 32.)