Coordination-free Multi-domain NFV Orchestration for Consistent VNF Forwarding Graph Reconfiguration

Josué Castañeda Cisneros*, Saul E. Pomares Hernández*†, Julio César Pérez Sansalvador†‡
Lil M. Rodríguez Henríquez†‡, Sami Yangui*§, Khalil Drira*
*LAAS-CNRS, Université de Toulouse, §INSA, F31400, Toulouse, France. † INAOE, 72840, Santa María Tonantzintla,Puebla, Mexico. ‡ INAOE - Cátedra CONACyT, 72840, Santa María Tonantzintla,Puebla, Mexico.{jcastane, yangui, drira}@laas.fr;{spomares, jcp.sansalvador, lmrodriguez}@inaoep.mx

***Abstract*—Multi-domain federations support shared network services. Many orchestrators manage the service's lifecycle. For the shared VNF Forwarding Graph (VNF-FG) reconfiguration, orchestrators update the graph's logical information, ensuring a consistent behavior for replicas. Only one work in the literature considers sharing the VNF-FG. However, it offers weak consistency guarantees, without considering the VNF-FG's non-functional dependencies. In case of a conflict, while updating the VNF-FG, the orchestrators solve consensus. However, this adds latency, undermining the goal of Network Function Virtualization. This paper introduces the first coordination-free multi-domain orchestration algorithm for consistent shared VNF-FG reconfiguration. Unlike the current state of the art, the proposed algorithm skips the coordination phase, offers strong eventual consistency, and supports non-functional dependencies. We present two variants: the preventive, where transient inconsistent states are prevented; the corrective, where intermediary inconsistent states, during the updating process, are tolerated. We prove the correctness of our algorithm and evaluate it. The variants, unlike the state of the art, reconfigure consistently the shared VNF-FGs without solving consensus. They offer stronger guarantees compared to the literature, like allowing orchestrators to reject ongoing reconfigurations without a high impact on performance.**

***Index Terms*—Coordinated-Free Orchestration, Multi-domain Orchestration, VNF Forwarding Graph (VNF-FG), Network Function Virtualization (NFV), Virtual Network Functions (VNF)**

I. Introduction

Multi-domain orchestration is the next step towards future network services [1]. It promises stronger performance by placing network services close to users, performing dynamic and flexible on-demand service provisioning, and improving resiliency for network services [2]. According to the European Telecommunications Standards Institute's (ETSI) Network Function Virtualization (NFV)[1] standard, such network services are managed by local orchestrators. They have limited information, as they do not know the resources and topologies used by other providers [3], [4]. This has a profound effect on how services are provisioned, enabling service sharing. Indeed, network services can be shared among multiple providers, reducing the cost for providers. For such shared services, many orchestrators handle the services' lifecycle management. This joint management creates both internal and external dependencies for network services. Internal dependencies are managed by a single orchestrator under a unique administrative domain. External dependencies are managed by many orchestrators from different administrative domains. Thus, managing the lifecycle of shared network services introduces new challenges unforeseen in single-domain orchestration [5]. This happens as no single orchestrator has global knowledge to provisioning network services. For example, when an orchestrator reconfigures a shared service's VNF Forwarding Graph (VNF-FG) in a multi-domain environment, the orchestrator has to ensure that all orchestrators who also use the shared service have the same information (i.e. the replicas of the VNF-FG are consistent). According to the ETSI information model, the reconfiguration of the VNF-FG means updating the data structure for classification rules and rendered service paths [6]. If one VNF-FG replica has different values than others, there is a risk of inconsistencies created by conflicting operations made by different orchestrators. This is the problem of consistent VNF-FG reconfiguration.

The problem of consistent VNF-FG reconfiguration involves updating or extending a VNF-FG responding to new demands [7]. Since the VNF-FG defines a logical order of execution for each dependency of a service [8], the orchestrator can change the order of execution by updating connection points or classifier rules [9]. To ensure the other orchestrators 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' VNF-FG 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 with two conflicting updates. 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 replicas, it may not be the case, as these replicas can be used by other services. 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

[1]https://www.etsi.org/technologies/nfv

reconfiguration, limited knowledge, and non-functional dependencies introduces conflicts that must be fixed to achieve a consistent state at the end of reconfiguration.

Current orchestration algorithms for the VNF-FG are focused mostly on the embedding problem without considering reconfiguration [10]–[12]. A couple of works consider reconfiguration; but they only consider single-domain orchestration [7], [13]. To the best of our knowledge, there are two works for reconfiguring shared VNF-FGs in multi-domain federations [14], and [15]. Orchestrators in [14] and [15] can execute a coordination phase to resolve inconsistencies due to non-deterministic network conditions (e.g. delay, repeated messages, orchestrators connect/disconnect). Traditionally, orchestrators choose either to select a single orchestrator to serve as a referee or they solve consensus [16]. These two choices reflect the two competing goals of distributed computation in terms of performance and consistency, as presented in the previous two works. For example, in [14], performance (i.e. fast reconfiguration) is chosen over strong consistency by ensuring eventual consistency among orchestrators [17]; while in [15], stronger consistency guarantees are chosen by ensuring causal consistency among orchestrators. However, both [14] and [15] do not consider non-functional dependencies of such shared VNF-FGs. This unrealistic reconfiguration limits the benefits of multi-domain federations since the administrative domains should be autonomous to share their VNF-FGs to different providers without a limitation.

In this paper, we focus on the consistent VNF-FG reconfiguration under multi-domain environments, considering non-functional dependencies among the VNF-FGs. Unlike in our previous published work [15], in this paper we consider concurrent reconfigurations and non-functional dependencies **without a coordination phase**, which has not been explored in the literature for VNF-FG reconfiguration in multi-domain orchestration. Our major contributions are:

- The design of a coordination-free orchestration algorithm for consistent VNF-FG reconfiguration under multi-domain federations. 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.
- Additionally, our proposed algorithm supports non-functional dependencies that have not been addressed so far in the literature for VNF-FG reconfiguration (Section IV-D). Unlike current orchestration solutions, orchestrators can negate ongoing reconfigurations for their VNF-FG replica. Thus, we consider a more general consistent VNF-FG reconfiguration problem.
- The tailoring of two different variants of our proposed algorithm to address various applications. We propose a preventive variant that ensures no transient inconsistent states happen (Section V-A). Additionally, we propose a corrective variant that tolerates contingent inconsistent states, as reconfigurations execute as soon as they arrive (Section V-B).
- The formal proof of the correctness of both variants (Appendix C) and their evaluation (Section VI). We compare them to the closest work found in the literature and discuss the trade-offs in terms of cost and performance.

The rest of the paper is organized as follows: Section II describes more in detail all the concepts used in the paper. Section III details the works in the literature for VNF-FG reconfiguration to position this work with respect to the literature. Section IV details the problem of consistent reconfiguration with non-functional dependencies. Section V 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 VI presents the evaluation and comparison of the two variants of our proposed algorithm and the current VNF-FG reconfiguration algorithm. Section VII concludes the paper.

II. Background

In this section, we describe the required concepts used in this paper. Firstly, we describe the concepts from the ETSI's NFV standard, such as the multi-domain orchestration. Secondly, we describe three consistency models present in distributed environments as in multi-domain orchestration. For the rest of the paper, we use the ETSI NFV architecture and terminology [3]. Table I shows all the abbreviations used in the article, along with their definition.

A. Network Function Virtualization

NFV decouples network services from the underlying hardware. Under NFV, network service providers replace physical network appliances with software-based Virtual Network Functions (VNFs) [18]. By using such VNFs, the providers create network services by chaining VNFs according to a forwarding path with an execution order encoded in a VNF-FG. The VNF-FG specifies a network topology connecting the VNFs using virtual links through interfaces called connection points and associated rules applicable to the traffic conveyed over the topology [11], [19]. To achieve such deployment of services under NFV, an orchestrator handles the lifecycle management of services [14]. Each unique and central orchestrator handles all these tasks [4]. With NFV federations, the concept of multi-domain orchestration was proposed to extend the limited resources of each participant in the federation while remaining autonomous [2].

TABLE I
Abbreviations used in the article

Abbrevation	Definition
ETSI	European Telecommunications Standards Institute
NFV	Network Function Virtualization
VNF	Virtual Network Function
VNF-FG	VNF Forwarding Graph
SEC	Strong Eventual Consistency
CRDT	Conflict-free Replicated Data Type
VNF-FGR	VNF-FG Reconfiguration
CVNF-FGR	Consistent VNF-FG Reconfiguration
CVNF-FGR-NF	Consistent VNF-FG Reconfiguration with Non-Functional Dependencies

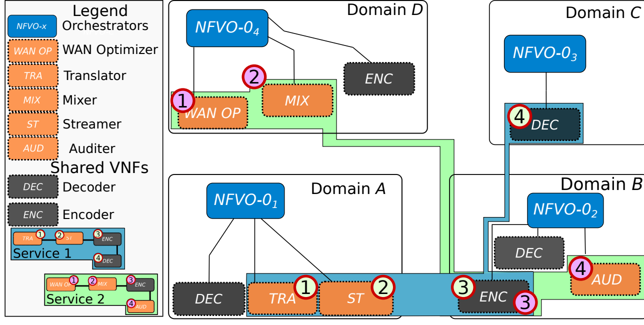

Fig. 1. Example of composite services with shared VNF-FGs. Four administrative domain have composite services that use shared VNFs and their replicas. This creates dependencies in the VNF-FGs.

A federation is a group of service providers who share resources to support complex network services [20]. Each provider manages an administrative domain [1]. This sharing scheme reduces the cost for each provider. When a service is shared by many orchestrators, they manage the service's lifecycle (e.g. placing, embedding, and reconfiguring the service). This extends the traditional notion of a network service [14]. Traditionally, network services were *dedicated*, as they belonged to a single domain. *Composite* services are created when two or more dependencies (i.e. their VNFs) belong to different administrative domains. Another way to think of such types of services is by their type of dependencies. *Dedicated* services only have internal dependencies; *composite* also has external dependencies. We consider all *composite* network services to be *shared* among many orchestrators. An example of *composite* services with shared VNF-FGs is shown in Fig. 1. In that example, four administrative domains use both dedicated and shared VNFs, creating *composite* services. Thus, their VNF-FGs are also shared among the orchestrators who use replicas of the VNFs.

Because services are *shared* by many orchestrators, they can also be used by different services. Thus, we consider that orchestrator replicas may negate ongoing reconfigurations due to 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; thus, tolerate different requirements (e.g. response time, availability, throughput). An orchestrator rejects a reconfiguration if this will affect dependencies after applying the changes. For example, when a replica of a VNF-FG receives an instruction to update connection points and classification rules, the orchestrator will verify the following: Firstly, that the functional requirements are met after applying the reconfiguration. Secondly, measure non-functional requirements, such as response time. One way this can be achieved is by creating a new VNF-FG by considering the changes in connection points and classifying rules. Finally, it will connect the new VNF-FG and monitor the changes. If both the traffic flow and the considered metrics are within a predefined threshold, the update will be done on the original VNF-FG instance. Moreover, since orchestrators can update the VNF-FGs at any time, these new changes can affect ongoing VNF-FG updates, disrupting the whole service.

To ensure proper service functioning, the consistency between all the replicas needs to be ensured, as those replicas can also be external dependencies of other services.

B. *Consistency models*

Consistency is a desired property in distributed multi-domain orchestration [5]. Next, we present a brief description of the concepts related to this topic. Firstly, we introduce sequential consistency and discuss its drawbacks. Secondly, we describe eventual consistency as the alternative to such drawbacks, highlighting the limitations of this consistency model. Finally, we introduce strong eventual consistency, which circumvents the trade-off between consistency guarantees and performance.

Distributed systems need to ensure consistency because of concurrent operations on shared data, such property is desired in multi-domain orchestration [5]. 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.

Consensus is the convergence to a common value among all participants [21]. It achieves sequential consistency for distributed systems. However, the high complexity of implementing consensus [22] and its low performance [23] make it a bottleneck for distributed systems. To improve performance on non-critical applications, eventual consistency was proposed [17]. 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 [24]. Eventual consistency is stated in Definition 1 [25].

Definition 1 (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, $\Diamond$ is a random and finite amount of time, and c_i, c_j are replicas of the same node c.*
Convergence: *Replicas that have delivered the same updates eventually reach an equivalent state:* $\forall i,j : c_i = c_j \implies \Diamond s_i \equiv s_j$, *where $\Diamond$ is a random and finite amount of time, s_i is the state of the ith replica.*
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 the convergence time is unknown [17]. Under eventual consistency, replicas can execute an operation without synchronizing *a priori* with other replicas, making data available at any given moment. Despite the consensus being moved off critical paths of applications, reconciliation is still complex to achieve [26]. Thus, Strong Eventual Consistency was proposed. We take the formal definition of Strong Eventual Consistency by [25].

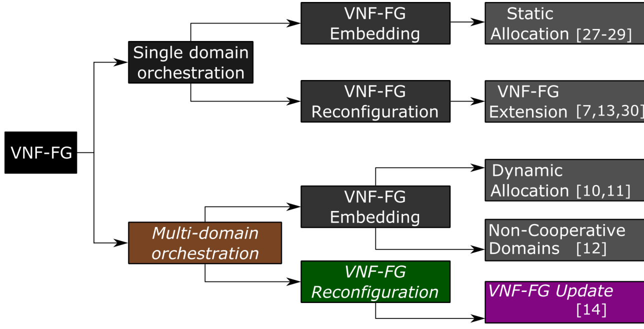

Fig. 2. Taxonomy with representative works for the VNF Forwarding Graph. Each branch solves a different problem for the VNF-FG. The closest work to ours is positioned in the colored/italicized path below.

Definition 2 (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 [25]. Currently, there is a portfolio of CRDTs for counters, registers, sets, and graphs that act as a building stone for more complex algorithms [26].

III. Related work

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 [27]. However, these tasks address different requirements and needs. The embedding problem asks how does the orchestrator selects the virtual network instances and their connection links [27]. The reconfiguration problem asks how to best update the VNF-FG (e.g. extending it, changing the order of the VNFs) while ensuring properties of the service, such as availability [11]. We focus on the reconfiguration problem, since composing and embedding only consider static deployment of the VNF-FGs. Moreover, the reconfiguration can also include the embedding and composing, if the orchestrator extends the VNF-FG.

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. Fig. 2 shows a taxonomy of representative works for the VNF-FG's lifecycle management. Our work is positioned in the colored lower branch along with the ETSI standard.

A. Single domain static deployment

The first category of works in the single domain considers a unique orchestrator where the VNF-FGs stay static. The primary goal of these works is to optimize metrics (e.g. latency [28], revenue [29], and energy [30]) 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 of 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 of works in this category.

B. 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 [7], extending the VNF-FG [13], and bi-directional chaining [31]. In the first two works, the orchestrator extends a VNF-FG by adding VNFs and links to respond to new demands. The authors propose an eigendecomposition algorithm [7] and a Steiner Tree-based algorithm [13] to solve the optimal extended embedding problem. The bi-directional chaining supports adding optional VNFs instances for deployed services [31]. While these works reconfigure the VNF-FG, they do not consider the problem of inconsistency when multiple orchestrators concurrently try to update the VNF-FG. 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. Additionally, providers prefer to keep their key information private [4].

C. Multi-domain works

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 [10]. 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 [11]. The last work considered in the third category considers a close and competitive environment where orchestrators hide their infrastructure from others [12]. The earlier works focus on the embedding problem, not on the VNF-FG reconfiguration problem. The two works that consider VNF-FG reconfiguration are the ETSI NFV standard [14] and our previous published work [15]. The ETSI NFV standard proposes a reconfiguration algorithm where the orchestrators coordinate through grants, checking the network service consistency. In our published work, coordination is enforced with causal consistency. This is achieved by making the orchestrators wait until they receive the appropriate reconfiguration instruction, while tracking the causal information with vector clocks.

D. Synthesis

The closest works to the reconfiguration problem we address in this paper are the ETSI proposed algorithm [14] and our

TABLE II
NOTATION

Variable	Meaning
$O = \{o_1, o_2, \ldots\}$	The set of orchestrators
$G = \{g^1, g^2, \ldots\}$	The set of VNF-FGs each numbered.
c_g	Classifier rule of VNF-FG g.
x_g	Rendered service path of VNF-FG g.
ma	Matching attribute, that belongs to a classifier rule.
p	Connection point of a rendered service path.
δ	The dependency relation
Δ	A VNF-FG Reconfiguration operation
L_θ	Pending operations for the the orchestrator θ
h_g^θ	The heap of accepted values for VNF-FG g from θ
$\Diamond$	A random and finite amount of time
s_i	State of the ith replica
$.id$	The identifier of a given entity, such as a VNF-FG
l_g^θ	The list of negated values for VNF-FG g from θ
ϵ	The initial value for a data structure.
$\phi(x_i)$	The state of the i-th VNF-FG x replica
r_o	Reply to update coming from orchestrator o
k_o	Orchestrator identifier
k_o^*	Highest identifier in the federation
x_i	the i-th replica of a VNF-FG x
$C(x_i)$	Causal history of the i-th replica of a VNF-FG x
$<_d$	The delivery order for reconfigurations
τ	The top operation

published work [15]. The former is considered the baseline for VNF-FG reconfiguration for ETSI-aligned VNF-based network services. The latter was designed to handle causal changes when VNF-FGs have shared external dependencies. In addition, our previous published work focused on non-concurrent reconfigurations, unlike the standard that allows them by design. In both works, in case of conflicts because of concurrent updates, the orchestrators must resolve conflicts/inconsistencies by solving consensus; thus, sacrificing performance over consistency. This means that the closest works execute a coordination phase to prevent inconsistencies. Moreover, they do not consider non-functional dependencies that might negate ongoing reconfigurations.

In this paper, we propose the first **coordination-free** orchestration algorithm to achieve consistent reconfiguration of the VNF-FG in a multi-domain federation. Our proposed orchestration algorithm, unlike the previous works, considers the consistency reconfiguration of a VNF-FG by updating either the connection points or classification rules. Also, we consider the non-functional dependencies inherent in sharing network services in distributed multi-domains.

IV. Consistent VNF Forwarding Graph Reconfiguration in Multi-Domain environments

In this section, we introduce first the reconfiguration of VNF-FGs. Then, we describe the consistent reconfiguration where many orchestrators managed *shared* services and their VNF-FGs. Finally, we introduce a generalized version of the consistent VNF-FG reconfiguration, where non-functional dependencies can negate ongoing reconfigurations. Table II shows the notation.

A. High-level description

Before introducing the formal notation used in the paper we describe briefly the high-level scenario of the entities

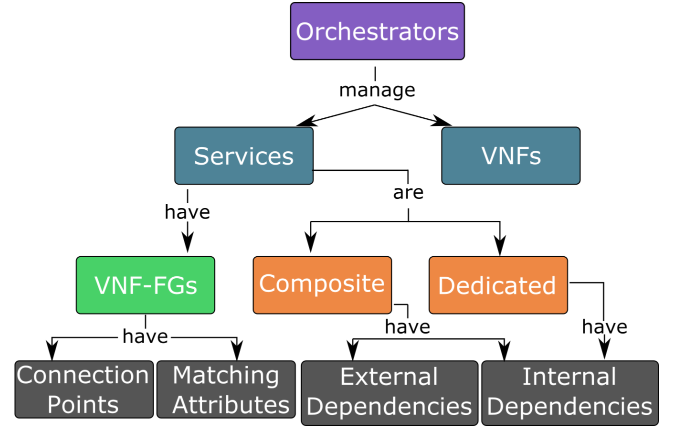

Fig. 3. Relations between the entities of the distributed multi-domain orchestration system model with services and VNF-FGs.

considered in the paper. Figure 3 show these entities and their relations. Federations are composed of many domains that share services managed by different orchestrators. Services can be of type composite and dedicated. The latter has only internal dependencies (e.g. VNFs) the former have also external services as dependencies. Dependencies can be both functional and non-functional. Moreover, each of these dependencies is connected with other VNFs or services, as specified in the VNF-FG associated with the service. The VNF-FG contains a list of matching attributes and connection points that detail how the dependencies are connected and how network traffic needs to be processed.

Since services are shared, orchestrators need to handle the service's replicas, and by extension, the service's VNF-FG. Thus, reconfiguring a VNF-FG for shared services means that the replicas (and their dependencies) need to agree on the new value for both the matching attributes and connection points.

B. 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 [8]. To achieve this logical order, the VNF-FG has a rendered service path and classifying rules [32]. 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 [33]. This change can be done by updating the values either via changing a connection point or matching attribute and adding/removing more elements to the lists.

Let g be a VNF-FG that belongs to the set of the federation's VNF-FGs 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 has a list of connection points $[p_1, p_2, \ldots, p_v]$. The function ϕ computes the state of the VNF-FG g as defined in Equation 1.

$$\phi(g) = (c_g, x_g) = ([ma_1, \ldots, ma_u], [p_1, \ldots, p_v]) \quad (1)$$

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 2.

$$\begin{aligned} \Delta : g \rightarrow g' = \exists p \in x_g, p' \in x_{g'}, p.id = p'.id \mid \phi(p) \neq \phi(p') \\ \exists ma \in c_g, ma' \in c_{g'}, ma.id = ma'.id \\ \mid \phi(ma) \neq \phi(ma'). \end{aligned} \quad (2)$$

We name the problem of reconfiguration for a VNF-FG as VNF-FGR. For a *dedicated* service, the reconfiguration is trivial as the service's orchestrator manages all the resources and resolves conflicts easily. However, for *shared* services, the chief interest is that all affected orchestrator replicas have the same view after a reconfiguration. This is the goal of the consistent VNF Forwarding reconfiguration problem.

C. Consistent VNF Forwarding Graph reconfiguration

When the orchestrator changes a copy of a *shared* VNF-FG $g \in G$ 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, (e.g. $\forall g' \in G | g.id = g'.id$). We formalize the consistent VNF-FG reconfiguration on Equation 3.

$$\Delta(g) : \forall g' \in G | g.id = g'.id \implies \phi(g) = \Box \phi(g). \quad (3)$$

where $\phi(g)$ is the state of the VNF-FG g and $\Box$ is a random and finite amount of time. **An inconsistency occurs if, after a VNF-FG reconfiguration Δ on a *shared* service g, the state of a replica of g' do does not match**. We call the consistent VNF-FG reconfiguration problem CVNF-FGR.

After a consistent reconfiguration of the VNF-FG, Equation 3 holds true. We illustrate a consistent reconfiguration of a VNF-FG for a content delivery *shared* service containing a *decoder* VNF. The service has been replicated in a multi-domain federation with three different providers that have three different orchestrators o_1, o_2, o_3. Because of a surge in service usage, the orchestrators must reconfigure the *shared* service along with its VNF-FG. We illustrate this in Fig. 4 (A) by changing the connection point of the *shared decoder* VNF represented by value a (e.g. changing the input point). In Step 1, all the VNF-FG replicas start in the same state o. In Step 2, the second orchestrator o_2 updates its VNF-FG replica with value a and sends a notification to other orchestrators. In Steps 3 and 4, the other orchestrators receive the notification and also update their VNF-FG replicas. At the end of the reconfiguration, all the replicas have the same value a. This is an example of a consistent VNF-FG reconfiguration.

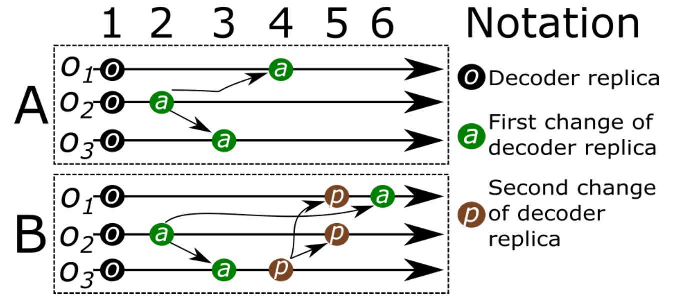

Fig. 4. Two examples of VNF-FG reconfiguration. (A) All replicas of the *shared* service's VNF-FG have the same value. This is an example of an consistent reconfiguration. (B) All replicas of the *shared* service's VNF-FG have different values. This is an example of an inconsistent reconfiguration.

In ideal conditions, consistency can be achieved by sending the notification to all affected replicas. But, non-deterministic network conditions, such as a random latency, can lead to an inconsistent reconfiguration. Such reconfiguration brings partial or total failures, violating the Service Level Agreements. We extend the example from Fig. 4 (A) to show an inconsistent VNF-FG reconfiguration. In this new example, concurrent updates can be done. We illustrate this in Fig. 4 (B). In Step 1, all orchestrators start with an initial value o. In Step 2, the second orchestrator o_2 updates its VNF-FG replica with value a and sends a notification to the other orchestrators. In Step 3, the third orchestrator receives the notification and updates its replica. In Step 4, the third orchestrator o_3 updates again the replica, based on the previous update, to a new value p (e.g. updating a different input point) and sends a notification to all the others. Two concurrent tasks execute in Step 5. Firstly, the first orchestrator o_1 receives the second update from o_3 with value p and updates its VNF-FG replica. Secondly, the second orchestrator o_2 also updates the replica with value p. In the Step 6, the o_1 receives the second update from o_2 with value a. Since the orchestrators have local knowledge of their administrative domain only, it updates its replica. In the end, VNF-FG replicas have different values. This is an example of inconsistent VNF-FG reconfiguration. Moreover, non-functional dependencies can make the VNF-FG reconfiguration more challenging. This problem is called the VNF-FG consistent reconfiguration with non-functional dependencies problem.

D. Consistent VNF Forwarding Graph Reconfiguration with Non-Functional Dependencies

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-FG Reconfiguration with Non-Functional Dependencies (CVNF-FGR-NF).

We illustrate an example of an inconsistent reconfiguration extending the same example for reconfiguration presented in Fig. 4 (A) but where the orchestrators can refuse a reconfiguration. This is shown in Fig. 5. In this scenario, four orchestrators (o_1, o_2, o_3, o_4) from different administrative domains manage two *shared* encoders g^1, g^2 and *shared* 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 *shared* encoders and decoders start with initial values represented by o, and $*$ as shown in the bottom of Fig. 5 (box A). Additionally, the encoders and decoders are related since decoder configuration depends of the encoder configuration, as shown in the bottom of Fig. 5 (box B). This means, that in order to reconfigure a decoder, both orchestrators that manage g^1, and g^2, respectively, have to accept the changes. The reconfiguration is as follows: In Step 1, both the third and fourth orchestrators change the value of their respective *shared* decoders g^3, g^4 with different values a, p, respectively. Step 2 shows how the first and second orchestrators update their *shared* encoders. The first orchestrator o_1 verifies the proposed reconfiguration. After accepting the reconfiguration, change the value of the encoder g^1 to a new value y, as shown in the second Step of Fig. 5. The second orchestrator o_2 does the same and updates the encoder value to y, as shown in the second Step of Fig. 5. In Step 3 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, keeps the value y, and sends a negative reply to o_4. Secondly, the second orchestrator o_2 gets the same notification from the fourth o_4. Since it accepts this most recent update, the orchestrator updates the value of the dependency to x; then, it replies positively to o_4. Thirdly, the concurrent task is the positive reply from o_1 to o_3, which updates the first of the two required answers, as shown in the third Step of Fig. 5. In Step 4, 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, and updates the first of two required answers; while, o_4 from o_2, also updating the second answer. 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 5, two concurrent events happen as messages arrive at 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, the dependencies have different values, as shown in the bottom of Fig. 5 (second row of box C). The other choice is to remain in the initial state; but the replicas have different values, as shown in the bottom of Fig. 5 (first row of box C). Moreover, because orchestrators can share the decoder among other orchestrators with limited knowledge, conflicts will arise as the replicas diverge further and further each concurrent reconfiguration. This is an example of an inconsistent reconfiguration with non-functional dependencies.

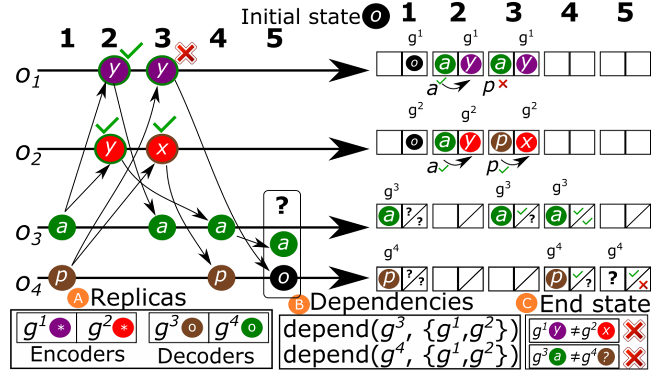

Fig. 5. VNF-FG reconfiguration with non-functional dependencies. Replicas of the *shared* service's VNF-FG and their dependencies have different values. The fourth orchestrator o_4 cannot determine what will be the VNF-FG value for the concurrent values. This is an example of an inconsistent reconfiguration.

The CVNF-FGR problem (Subsection IV-C) 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 by 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 supports fault-tolerance. We call this the corrective problem.

V. Coordination-free orchestration algorithm for multi-domain environments

In the CVNF-FGR-NF 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. 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 have 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 $G \cup G'$. When reconfiguring VNF-FGs with a dependency relation, 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 Se-

quential Consistency (see Section II-B). 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 model such as with the Strong Eventual Consistency (see Section II-B). Such a consistency model achieves the ideal trade-off between consistency, availability, and partitioning. Consistent free replicated data types (CRDTs) can offer such a stronger consistency model.

We consider CRDTs (see Section II-B) as an automatic conflict resolution mechanism. The idea is to design an orchestration algorithm using data structures that support SEC to avoid the coordination phase ensuring consistency. For the proposed algorithm, whenever an orchestrator executes a reconfiguration, it will change either the matching attribute and/or the connection points, as specified by the ETSI's NFV standard information model [6]. This means changing the data structure that holds all the required information for them. For example, to change the matching attribute, the orchestrator will set a new value for the protocol recognized, the source and destination addresses, and ports. For different paradigms during conflict management, we now describe briefly the two variants of our algorithm.

The first variant prevents inconsistencies by applying updates only when all replicas and dependencies have asserted 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 change is not applied until all answers are received. If any is negative, the reply will be discarded. 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 this 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 orchestrator 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 the two variants to achieve the Consistent Dependent VNF-FG Reconfiguration and we prove they achieve SEC.

A. 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*. Algorithms 1, 2, and 3 describe in detail the *CF-P* variant. When an orchestrator tries to update a VNF-FG it first executes Algorithm 1.

For Algorithm 1, first, the orchestrator applies the reconfiguration to a copy of the VNF-FG g (line 1). Then, the orchestrator increases the counter for the reconfiguration to uniquely identify it (line 2). After, it computes the set of orchestrators who manage replicas of the VNF-FG (line 3). Next, the orchestrator creates a list to store the replicas' replies (lines 4, 5). Finally, it appends the reconfiguration to a list of pending operations and sends the instruction to reconfigure to other orchestrators (lines 6-8). When an orchestrator receives a notification, it will execute Algorithm 2.

For Algorithm 2 the orchestrator first will compute the set of orchestrators that have replicas of the VNF-FG (line 1). Then, it creates a temporary answer that will change depending if the change will be accepted or not (line 2). If the orchestrator has not previously received a reply to this reconfiguration, it will create a new empty list and add it to its pending operations (line 3). After, the orchestrator checks the feasibility of the reconfiguration (i.e. the replica and its dependencies satisfy non-functional properties); if valid, the orchestrator will mark the reconfiguration as accepted (line 6); otherwise, as false (lines 12, 13). If the orchestrator accepted the reconfiguration and the counter of the new operation is greater than the current one, the orchestrator applies the changes and it updates the counter of the VNF-FG (lines 8, 9). Finally, the orchestrator creates a message and sends the reply to all affected orchestrators (lines 15, 16). When an orchestrator receives a reply, it will execute Algorithm 3.

For Algorithm 3 the orchestrator first stores the reply in the list for the VNF-FG reconfiguration (line 1). If all entries are positive, the orchestrator will apply the update (lines 3, 4).

Algorithm 1 update_vnffg

```
1: update_vnfforwarding_graph_copy()
2: create_unique_identifier_for_reconfiguration()
3: get_list_of_orchestrators_to_send_message()
4: create_empty_list_to_store_answers()
5: initialize_list_with_positive_value()
6: add_new_list_to_pending_operations()
7: create_message_to_notify_update()
8: send_message_to_all_affected_orchestrators()
```

Algorithm 2 receive_notification_update_message

```
1: get_list_of_orchestrators_to_send_reply()
2: create_answer_and_set_as_false()
3: if_first_create_list_and_append_to_pending_operations()
4: if check_feasibility(Δ) then
5:     mark_entry_as_positive()
6:     update_answer_to_positive_reply()
7:     if all_entries_are_positive_and_greater_counter then
8:         reconfigure_the_vnffg()
9:         update_the_counter_with_new_one()
10:    end if
11: else
12:    mark_entry_as_negative_in_list()
13:    add_entry_as_negative_in_list()
14: end if
15: create_message_to_notify_update()
16: send_message_to_all_affected_orchestrators()
```

Algorithm 3 receive_reply_message

1: mark_entry_with_answer()
2: **if** all_entries_of_list_are_positive_and_counter_is_greater **then**
3: reconfigure_the_vnffg()
4: update_the_counter_with_new_one()
5: **end if**

Consider the same reconfiguration of a shared VNF-FG as in Section IV-D 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. Fig. 6 shows the execution for the reconfiguration to ensure all replicas are consistent by having the same values. For ease of readability, the images only show a single value that changes whenever a reconfiguration happens; however, in reality our algorithm considers the whole data structure of the VNF-FG according to the information model [6]. 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 Fig. 6, 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 Fig. 6. 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*. Steps 4 and 5 shows how more notification 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 Fig. 6 by the change of color 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 5 and 6 it can be seen how the preventive variant achieves a consistent reconfiguration despite non-functional dependencies. A more in-depth description of the preventive algorithm is detailed in the Appendix A in Algorithms 7, 8, 9. The correctness proof is detailed in Appendix C Section C-A.

B. 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*. Algorithms 4, 5, and 6 describe in detail the *CF-C* variant.

When the orchestrator reconfigures a VNF-FG it starts by executing Algorithm 4. First, the orchestrator updates the VNF-FG directly (line 1). Then, it increases the counter to uniquely identify the operation (line 2). After, the orchestrator computes the list of orchestrators that have replicas of the VNF-FG (line 3). Next, it adds the reconfiguration to the heap of accepted reconfigurations (line 4). Finally, the orchestrator creates a message and sends it to the list of orchestrators (lines 5, 6). When an orchestrator receives this message, it executes Algorithm 5.

For Algorithm 5, the orchestrator first checks if the reconfiguration already is stored in the list of negated reconfigurations; if not it continues (line 1). After, the orchestrator checks if the reconfiguration is feasible (i.e. replicas and dependencies' non-functional properties are satisfied after reconfiguration). If it is valid, the orchestrator checks if both counters are greater than the current ones, if they are the VNF-FG is reconfigured and the top counter is updated (lines 4, 5). Otherwise, it is added to the heap in the correct place by using the counters as identifiers (line 7). If the reconfiguration is not accepted, the orchestrator computes the list of affected orchestrators, adds the reconfiguration to the list of rejected operations, creates a negative reply message, and sends it to all affected orchestrators (lines 9-12). When an orchestrator receives the negative reply, it executes Algorithm 6.

For Algorithm 6, the orchestrator removes the operation from the heap (this could be the top or any other position),

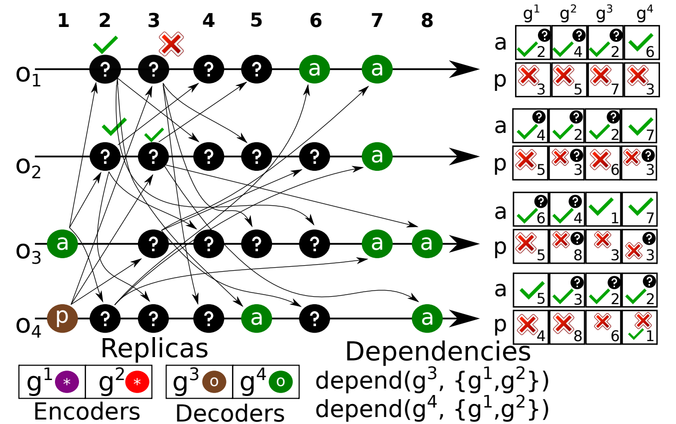

Fig. 6. VNF-FG reconfiguration with our proposed **preventive** variant. At the end of the reconfiguration, replicas of the *shared* service's VNF-FG have the same value. Unlike Fig. 5, this is an example of a consistent reconfiguration with non-functional dependencies.

adds the reconfiguration to the list of negated operations, re-orders the heap, and applies the reconfigurations starting from the initial consistent state (lines 1-4).

Consider the same reconfiguration of a shared VNF-FG as in Section IV-D where four orchestrators o_1, o_2, o_3, o_4 manage two *shared* encoders g^1, g^2 and two *shared* 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. Fig. 7 shows the execution for the reconfiguration to ensure all VNF-FG replicas are consistent by having the same values. For ease of readability, the images only show a single value that changes whenever a reconfiguration happens; however, in reality, our algorithm considers the whole data structure of the VNF-FG according to the information model [6]. In Step 1, the third and fourth 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 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 on the right side of Fig. 7. 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_1. 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_1, o_2 accepts the proposal and reconfigures the VNF-FG g^2 to match the state of value p. This is shown in Fig. 7 where the top now is p; unlike in the previous Step 2. Thirdly, the proposal from o_3 arrives at 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 the 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 in Fig. 7. At the end of reconfiguration, all VNF-FG replicas have the same values. Comparing Figures 5 and 7 it can be seen how the preventive variant achieves a consistent reconfiguration despite non-functional dependencies. A more in-depth description of the preventive algorithm is detailed in the Appendix B in Algorithms 10, 11, 12. The correctness proof is detailed in Appendix C in Section C-B.

Algorithm 4 update_vnffg

1: apply_reconfiguration_to_vnffg()
2: increase_the_vnffg_counter()
3: get_list_of_orchestrators_to_send_notification()
4: add_reconfiguration_entry_to_heap()
5: create_update_notification_message()
6: send_notification_to_all_affected_orchestrators()

Algorithm 5 receive_notification_update()

1: pass_if_entry_is_negated()
2: **if** check_feasbility(Δ) **then**
3:     **if** counter_greater_equal_greater_identifier **then**
4:         apply_the_reconfiguration_to_the_vnffg()
5:         increase_the_counter_for_the_vnffg()
6:     **end if**
7:     add_entry_to_heap()
8: **else**
9:     get_list_of_affected_orchestrators()
10:     add_reconfiguration_to_list_of_rejected_operations()
11:     create_negative_reply_message()
12:     send_negative_reply_to_affected_orchestrators()
13: **end if**

VI. Experiments and results

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 VI-A), metrics evaluation (Section VI-B), experiments (Section VI-C), and discussions (Section VI-D).

A. Distributed Federation setup

We deploy the federation in a public cloud provider infrastructure to achieve a good trade-off between cost and availability, as private providers are not representative of multi-domain federations due to their limited availability. 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. The goal is to cover a large geographical space that covers more than two continents. Each domain communicates

Algorithm 6 receive_reply_message

1: remove_operation_from_heap()
2: add_reconfiguration_to_list_of_rejected_operations()
3: compute_new_update_after_the_new_order_from_heap()
4: apply_new_update_from_consistent_state()

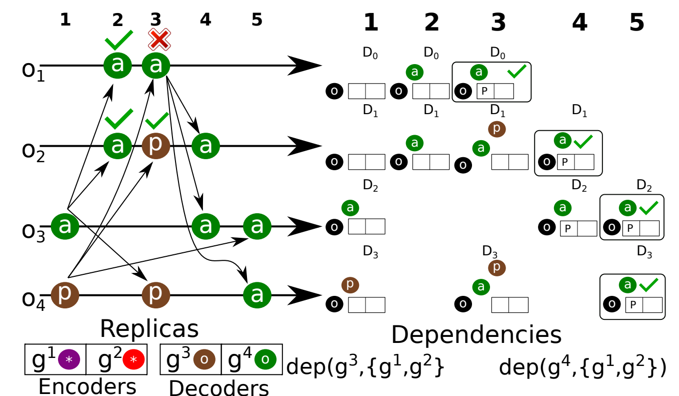

Fig. 7. VNF-FG reconfiguration with our proposed **corrective** variant. At the end of the reconfiguration, replicas of the *shared* service's VNF-FG have the same value. Unlike Fig. 5, this is an example of a consistent reconfiguration with non-functional dependencies.

over the Internet, such that average latency is representative of real conditions, while the orchestrators communicate. 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 Ubuntu 18.04-LTS. These resources were sufficient to deploy an orchestrator, and each service was created by running processes in the virtual machine. 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 [6]. Each domain has its policies, topology, and manages a single orchestrator. We defined the policies in a JSON file that contained all the required information to set up the experiment, such as the orchestrators, location, number of VNFs, types of VNFs, required information for VNF-based network services and their associated VNF-FGs. This descriptor was generated using random seeds. The topology is also specified in the experiment descriptor and states how services are connected. Connections are achieved either by internal or external dependencies. The way services are related and built depends on random seeds, also included in the JSON file. Finally, the workload is defined in another file that contains all the concurrent changes. Thus, the workload changes according to the experiment's parameters. More details about the Python libraries used, the topology of services, the random seeds, the VNF-FGs, and the environment used for testing can be found in the source code [2].

We measure the performance of the proposed algorithms under different scenarios. Thus, we consider different parameters for each experiment, as shown in Table III. The combination of these parameters creates different scenarios, each one of different complexity. For example, an ideal scenario would have zero delay (i.e. **Max_D**=0), accept all reconfigurations (i.e. **Pb_N**=0), and non-concurrent reconfigurations (i.e. **Nm_R**=0). A worse scenario, compared to the ideal, would have greater delay (i.e. **Max_D**=100ms), negate some reconfigurations (i.e. **Pb_N**=20), and have concurrent reconfigurations (i.e. **Nm_R**=1200).

B. 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 they achieved a correct state.
- The ETSI standard algorithm (**NCF-E**). It allows for reconfiguration to be done using eventual consistency by applying updates the moment they arrive.

Next, we describe each metric. The overall goal is to be able to compare the algorithms and evaluate the trade-offs of each to select the appropriate one for a given scenario.

- 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.
- Inconsistencies. Measures the number of updates that are different for each replica. A lower value is preferred since inconsistencies directly translate to cost for providers and lower performance for users.
- Overhead per messages. Measures the data each orchestrator sends to other replicas and dependencies when executing an algorithm. A lower value is preferred. This metric is associated with a cost for CRDT-based algorithms.

C. Experiments

We consider all combinations of parameters for the experiments. Each combination is evaluated thirty times and we take the average values for each metric previously described. Because of length limitations for the paper, we only show relevant results to compare the three algorithms using only the concurrent scenarios, as they are more representative of the algorithms' behavior. The two variables controlled for the experiments were a probability of negation and network delay. We chose the probability to negate in four scenarios (i.e. 0%, 5-10%, 20-40%, and 80%). A negation of 0% means the replicas and dependencies always accept changes. A negation of 5-10%, that replicas almost always accept changes. Higher values introduce more complexity for the algorithms considered, since they have to ensure no inconsistencies happen. Thus, these probability values characterize the environments in which the algorithms could be used; for low probabilities, the providers have sufficient resources to apply changes, therefore accepting the reconfigurations. Higher probability negations mean there is a lack of resources for providers to reconfigure. The network delay also was changed to evaluate different scenarios. However, because of the limitations of the paper, we only show the results of delays with delays of 100 and 1000ms. The rest of the Figures, as well as the data processing done, can be found in the repository[3]. We evaluated the different algorithms for

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

[3]https://doi.org/10.5281/zenodo.6534249

TABLE III
PARAMETERS CONSIDERED FOR THE EXPERIMENTATION. EACH PARAMETER CONFIGURATION CREATES DIFFERENT TEST SCENARIOS FROM THE IDEAL TO THE WORST CASE.

Parameter	Range	Explanation
Maximum Delay (Max_D)	[1, 10, ..., 100000] ms	This evaluates the performance of the algorithms under low and high non-deterministic conditions. Intuitively, the greater the delay; the worst the performance.
Number of Reconfigurations (Nm_R)	0, 150, 300, ..., 1800	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)	Concurrent	Concurrent updates allows for reconfiguration to happen at any time.
Probability to Negate (Pb_N)	0 - 5 - 10 - 20 - 40 - 80 %	Signals how the algorithms behave when the reconfigurations are negated by other orchestrators due to violating non-functional requirements.
Number of repetitions per experiment	30	Each combination of parameters was evaluated thirty times to reduce bias.

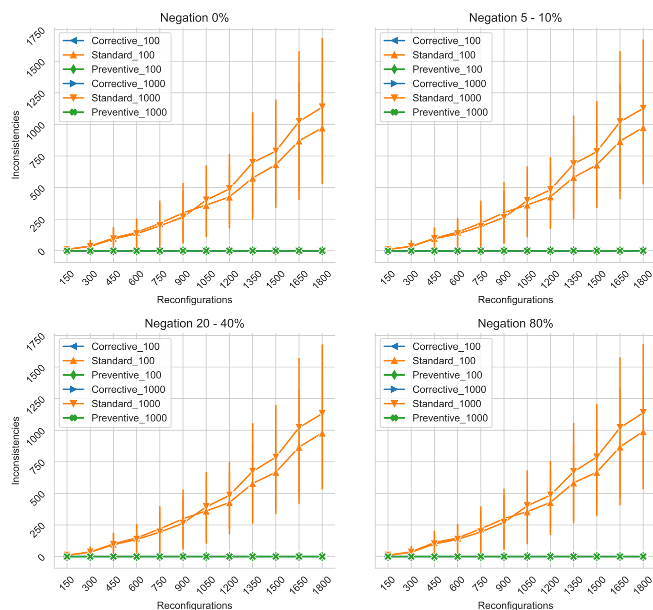

Fig. 8. Inconsistencies per number of reconfigurations, lower is better. Both variants prevent inconsistencies for any type of scenario; thus, negation and delay factors do not have an impact on the variants. The standard does not prevent inconsistencies.

each metric considered. The proposed preventive variant is represented with a green axis/diamond for delays for 100 and 1000ms, respectively. The proposed corrective variant is represented with a blue right/left triangle for 100 and 1000ms, respectively. Finally, the standard algorithm is represented with an orange up/down triangle for 100 and 1000ms, respectively. For all results we draw a confidence interval at 95% using the Seaborn python library using a bootstrapping algorithm, as detailed in the data process repository. We present the results in Figures 8, 9, 10, 12, 11, 13. Next, we discussed the results from our experiments.

D. Results

The first column of Table IV shows the related work and the proposed variants. All the works focus on the VNF-FG reconfiguration, as shown in the second column. We evaluated our proposed algorithm compared to the ETSI standard [14] as it is the only relevant work aside from ours that considers *shared* VNF-FGs, as shown in the third column (shared VNF-FG) of Table IV; the other works [7], [13] do not. Both of our proposed variants, unlike the standard [14], support non-functional dependencies that can negate ongoing reconfigurations, as shown in the fourth column of Table IV. The corrective variant allows extra reconfigurations, unlike the preventive variant, as shown in the fifth column. 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 sixth column of Table IV. Thus, our variants support a more general case of the VNF-FG reconfiguration. However, supporting non-functional dependencies without a coordination phase has a cost. Next, we detail the results and discuss these costs.

Fig. 8 shows the number of inconsistencies per reconfigurations. These inconsistencies happen because the replicas or dependencies have different values. The standard has the worst performance, as the number of inconsistencies increases with more concurrent reconfigurations. Unlike the standard, the corrective and preventive variants do not have incon-

Fig. 9. Latency per reconfiguration operation, lower is better. Both the corrective and standard algorithms are instantaneous, while the preventive variant must wait. For 1800 concurrent reconfigurations the average latency per operation is 1 minute for preventive variant.

TABLE IV
SOLUTIONS FOR THE VNF-FG RECONFIGURATION WITH DIFFERENT FUNCTIONALITIES. OUR PROPOSED VARIANTS OFFER ALL FUNCTIONALITIES

Work	VNF-FG Reconfiguration	Shared VNF-FG	On-going negations	Non-Functional dependencies	Extra Reconfigurations	Inconsistencies
VNF-FG Extension[7,13]	Yes	No	No	No	No	N/A
ETSI Standard [14]	Yes	Yes	N/A	No	No	> 0
Preventive Variant (CF-P)	**Yes**	**Yes**	**Yes**	**Yes**	**No**	**0**
Corrective Variant (CF-C)	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**0**

sistencies. This behavior follows our theoretical results (see Appendix C), where inconsistencies are prevented by enforcing strong eventual consistency. Moreover, since the standard does not consider negation of ongoing reconfigurations, as shown in Table IV, negation has no effect on it. Fig. 8 shows that results are equal irrespective of the probability of negation. This is explained by the fact that both variants enforce strong eventual consistency, and replicas converge to a consistent state irrespective of the number of negations and delay; for the standard, since it does not support on-going negations, the results were also the same when we measured the inconsistencies of the algorithms for different probabilities to negate.

Fig. 9 shows the latency per reconfiguration operation (i.e. the time needed to wait before reconfiguring a VNF-FG). The preventive variant behaves worse than the standard and the corrective variant which applies the reconfiguration when it receives the reconfiguration instruction. With 1800 concurrent reconfigurations and 0% negation probability, the average latency per operation is about 1 minute for the preventive algorithm. However, when the negation probability increases (e.g. 20-40%), the latency reduces. This behavior is consistent with the way the preventive algorithm executes. If one entry

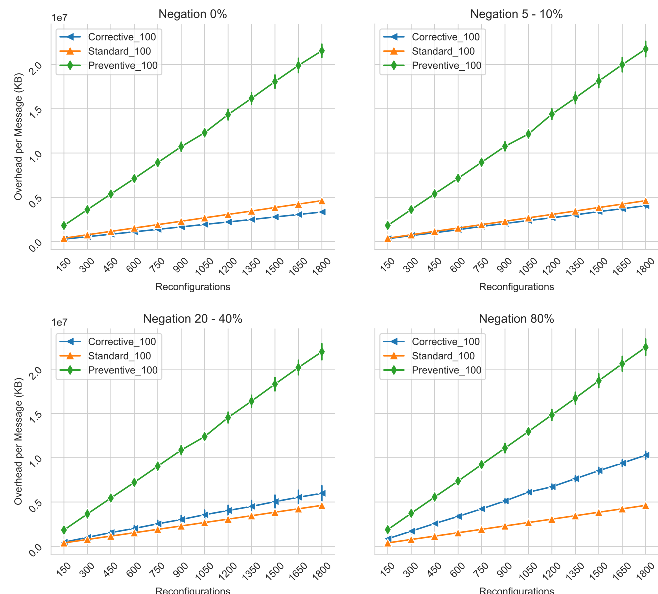

Fig. 11. Overhead per messages, lower is better. Delay and negation seem to not impact the preventive and the standard algorithms. For the corrective variant, negation has the greatest impact since more data is sent every time a negation happens.

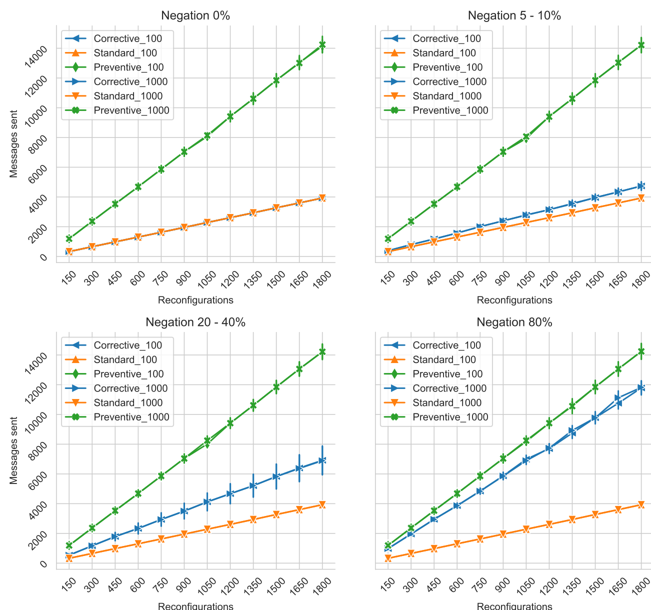

Fig. 10. Messages sent by the algorithms in different scenarios, lower is better. The corrective algorithm is more sensitive to parameters. For scenarios where negation and delay are low, it behaves like the standard algorithm. Otherwise, it behaves like the preventive variant as the gap widens between 1 to 100ms.

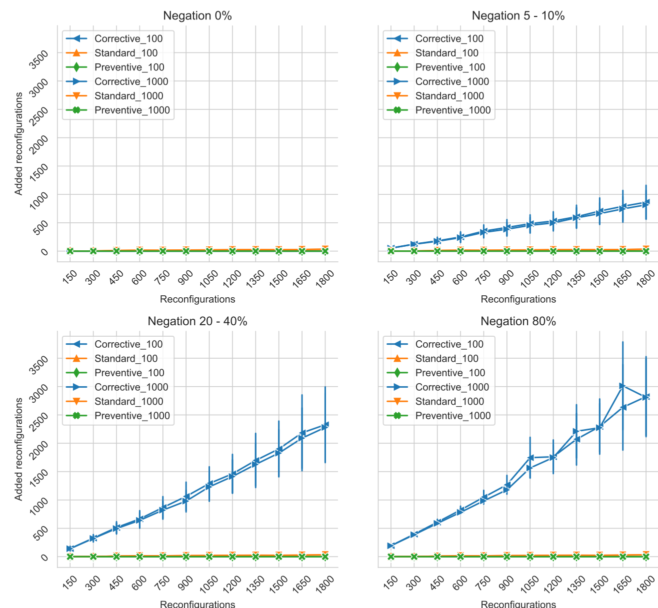

Fig. 12. Extra reconfigurations done by each algorithm, lower is better. The preventive variant has zero extra reconfigurations. The corrective is sensitive to the negation probability factor, as shown by the results from ideal conditions (0% negation) compared to more negations.

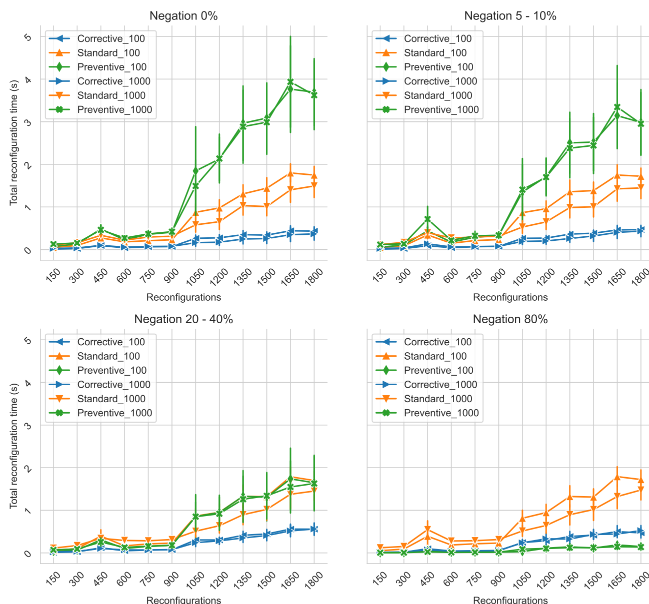

Fig. 13. Reconfiguration time for the VNF Forwarding Graph, lower is better. Negation and delay have a greater impact on preventive variants and the standard algorithm. With a higher probability of negations (e.g. 80%), the preventive is faster as a single element is required to abort the reconfiguration.

in the table is false (because either a dependency or replica did not accept the changes), the preventive algorithm can abort the reconfiguration without waiting for the rest of the answers. Thus, when orchestrators negate more updates, the latency for the preventive is reduced. Moreover, one minute of latency per operation might seem a high number compared to the corrective and standard; however, this latency is lower compared to the latency per transaction of consensus solutions (e.g. 10 minutes per transaction [34]). The behavior presented in Fig. 9 is representative for all the parameters' combinations.

Fig. 10 shows the messages sent to resolve conflicts between the orchestrators. The preventive variant gets 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 (i.e. $\geqq$ 20%), the corrective algorithm behaves like the preventive, as seen in how the corrective trend line moves towards the preventive one. In the worst-case scenario (i.e. 100% negation probability), the corrective variant will send the same number of messages. Delay has a slight impact on the number of messages sent. It widens the interval; however, behavior is still the same.

Fig. 11 shows the overhead per message exchanged between the orchestrators. Similar to the number of messages sent, the preventive variant gets the worst performance of the other variants. However, the negation probability seems to have a lesser impact on the corrective algorithm. It only doubles the amount of data sent, as shown with negations between 40% and 80%.

Fig. 12 shows the number of extra reconfigurations per algorithm. Here, the clear winner is the preventive variant getting zero extra reconfigurations, while the corrective variant behaves worst. The corrective variant is sensitive to the negation probability. For example, if all reconfigurations are accepted, the corrective variant does not have extra reconfigurations. However, with a greater probability to negate, the corrective algorithm must reconfigure more services, as shown in the number of reconfigurations done with 5-10% (e.g. 1000) and 20-40% (e.g. 3000).

Fig. 13 shows the average reconfiguration time for the VNF-FGs. Negation probability has greater impact than delay. With no negation, the corrective variant achieves the fastest reconfigurations; while the preventive variant takes more time. This is because the preventive variant sends more messages than the corrective in this case. With more negations, the preventive variant has achieved a faster reconfiguration time; while the standard algorithm takes more time. This is because, as previously mentioned, it only takes a single negated entry into the list of changes for the preventive variant to abort the reconfiguration. With higher negation probabilities (80%), the standard has the slowest reconfiguration.

E. Discussion

Delay does not seem to be of a great impact compared to the negation probability. Graphically this can be seen when two lines overlap, which happens for most Figures. However, in certain scenarios, it has some impact, such as in the number of inconsistencies for the standard algorithm. Moreover, with more concurrent reconfigurations, the variance increases for all metrics. Yet, for some, such as inconsistencies and added reconfigurations for the corrective variant, it has a greater impact.

Based on these results, we provide a better scope on which applications fit better the proposed variants. For resource-constrained federations, where consistency is a priority, extra reconfigurations are expensive, memory is limited, and latency is not a problem, service providers should select the preventive variant over the corrective, such as an IoT Virtualized network [35]. Service providers would use the corrective variant in more specialized environments where speed is required, memory is plenty, reconfiguration (in terms of resource consumption) is cheap, sending messages is costly, and the probability to negate reconfigurations is low, such as with NFV satellite networks [36]. Moreover, the corrective variant does not require knowing 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. These results match with the theoretical ones, which showed the applicability of both variants. Since we measure the behavior of both variants, implementing VNF-based services as processes suffices to describe the scope for each variant. Yet, we leave for future work a more in-depth experiment, where we consider more locations for the federation, covering all continents and places with high traffic density, and state of the art VNF implementations for a more precise performance evaluation.

VII. Conclusion

In this paper, we propose a coordination-free VNF-Forwarding Graph (VNF-FG) reconfiguration algorithm for network function virtualization in a multi-domain federation.

Algorithm 7 Update VNF-FG g, via Δ, with counter χ_g. The VNF-FG g is managed by an orchestrator θ with pending operation list L_θ.

1: $g \leftarrow \Delta(g)$
2: $c_g^\Delta \leftarrow \chi_g + 1$
3: $O_g \leftarrow (\forall o \in O, g' \in G; \| \; o \sim g', g.id = g'.id) \cup (\forall o \in O, g'' \in G \mid o \sim g'', \delta(g, g''))$
4: $l_g^\Delta \leftarrow [0 \mid \forall o \in O_g]$
5: $l_g^\Delta[\theta, c_g^\Delta] \leftarrow True$
6: $L_\theta \leftarrow L_\theta \cup l_g^\Delta$

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. Moreover, 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 present two variants of our proposed algorithm to target different applications. For critical and resource-constrained applications, where doing extra reconfigurations is undesired, we propose a preventive variant. For less stringent applications, we likewise propose a corrective variant. We formally prove 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 evaluate 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 with 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. 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 structure that allows support of more operations for the VNF-FG. Despite the overhead costs, our proposed algorithm, unlike the state of the art, works on a consistent VNF-FG reconfiguration problem.

Appendix A
Preventive algorithm: Formal specification

The formal specification of Algorithms 1, 2, and 3 is detailed in this Appendix using Algorithms 7, 8, and 9, respectively. All algorithms follow the notation presented in Table II.

Appendix B
Corrective algorithm: Formal specification

The formal specification of Algorithms 4, 5, and 6 is detailed in this Appendix using Algorithms 10, 11, and 12, respectively. All algorithms follow the notation presented in Table II.

Appendix C
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

Algorithm 8 Receive notification update message $\sigma = \{\vartheta, g'.id, \Delta, c_{g'}^\Delta\}$. The message contains the sender of the message ϑ, the identifier $g'.id$ of a VNF-FG g , the reconfiguration operation Δ, the counter of the reconfiguration operation $c_{g'}^\Delta$. for VNF-FG g, with counter χ_g and managed by orchestrator θ, with pending operation list L_θ.

1: $O_g \leftarrow (\forall o \in O, g' \in G; \| \; o \sim g', g.id = g'.id) \cup (\forall o \in O, g'' \in G \mid o \sim g'', \delta(g, g''))$
2: $\omega \leftarrow False$
3: **if** $(g'.id, \Delta) \notin L$ **then**
4: $\quad l_{g'}^\Delta \leftarrow [0 \mid \forall o \in O_{g'}]$
5: $\quad L_\theta \leftarrow L_\theta \cup l_{g'}^\Delta$
6: **end if**
7: **if** checkFeasibility(δ) **then**
8: $\quad l_{g'}^\Delta[\theta, c_g^\Delta] \leftarrow True$
9: $\quad l_{g'}^\Delta[\vartheta, c_g^\Delta] \leftarrow True$
10: $\quad \omega \leftarrow True$
11: $\quad$ **if** $\forall i \in l_{g'}^\Delta = True \;\&\; c_g^\Delta > \chi_g$ **then**
12: $\quad\quad l_{g'}^\Delta[\theta, c_g^\Delta] \leftarrow True$
13: $\quad\quad g \leftarrow \Delta(g)$
14: $\quad\quad \chi_g \leftarrow c_g^\Delta$
15: $\quad$ **end if**
16: **else**
17: $\quad l_{g'}^\Delta[\theta, c_g^\Delta] \leftarrow False$
18: $\quad l_{g'}^\Delta[\vartheta, c_g^\Delta] \leftarrow False$
19: **end if**
20: $\varsigma = \{\theta, \vartheta, g.id, \Delta, \omega, c_g^\Delta\}$
21: $\forall o \in O_g, send(o, \varsigma)$

Algorithm 9 Receive reply message $\varsigma = \{\vartheta, g'.id, \Delta, \omega, c_{g'}^\Delta\}$ for VNF-FG g, with counter χ_g. The message ς contains the sender ϑ, the identifier $g'.id$ of the VNF-FG g, the reconfiguration operation Δ, the result of the operation ω, and the counter $c_{g'}^\Delta$ for the reconfiguration operation Δ of the replica g' for VNF-FG g.

1: $l_{g'}^\Delta[\vartheta, c_g^\Delta] \leftarrow \omega$
2: **if** $\forall i \in l_{g'}^\Delta = True \;\&\; c_g^\Delta > \chi_g$ **then**
3: $\quad l_{g'}^\Delta[\theta, c_g^\Delta] \leftarrow True$
4: $\quad g \leftarrow \Delta(g)$
5: $\quad \chi_g \leftarrow c_g^\Delta$
6: **end if**

Algorithm 10 Update VNF-FG g, via Δ, with counter χ_g. The VNF-FG g is managed by an orchestrator θ with a heap h_g^θ for the VNF-FG g

1: $g \leftarrow \Delta(g)$
2: $\chi_g \leftarrow \chi_g + 1$
3: $O_g \leftarrow (\forall o \in O, g' \in G; \| \; o \sim g', g.id = g'.id) \cup (\forall o \in O, g'' \in G \mid o \sim g'', \delta(g, g''))$
4: $h_g^\theta.insert(\{\theta, \Delta, \chi_g\})$
5: $\sigma = \{\theta, g.id, \Delta, \chi_g\}$
6: $\forall o \in O_g, send(o, \sigma)$

Algorithm 11 Receive notification update message $\sigma = \{\vartheta, g'.id, \Delta, c^{\Delta}_{g'}\}$ for VNF-FG g from orchestrator θ, with counter χ_g in orchestrator i with a negation list l^i_g. The message σ contains the sender ϑ, the identifier $g'.id$ of the VNF-FG g, the reconfiguration Δ, and the counter $c^{\Delta}_{g'}$ for the reconfiguration for the replica of g. Both orchestrators θ and i have associated counter k^{θ} and k^i, respectively

1: **if** $\{\vartheta, \Delta, c^{\Delta}_{g'}\} \in l^i_g$ **then**
2: pass
3: **end if**
4: **if** checkFeasbility(Δ) **then**
5: **if** $c^{\Delta}_{g'} > \chi_g$ **or** $c^{\Delta}_{g'} = \chi_g$ **and** $k^{\theta} > k^i$ **then**
6: $g \leftarrow \Delta(g)$
7: $\chi_g \leftarrow c^{\Delta}_{g'}$
8: **end if**
9: $h^i_g.insert(\{\theta, \Delta, c^{\Delta}_{g'}\})$
10: **else**
11: $O_{g'} \leftarrow (\forall o \in O, g'' \in G; \| \ o \sim g'', g'.uid = g''.iud) \cup (\forall o \in O, g''' \in G \mid o \sim g''', \delta(g', g'''))$
12: $l^i_g.append(\{\vartheta, \Delta, c^{\Delta}_{g'}\})$
13: $\varsigma = \{\theta, g.uid, \Delta, False, c^{\Delta}_{g}\}$
14: $\forall o \in O_{g'}, send(o, \varsigma)$
15: **end if**

Algorithm 12 Receive reply message $\varsigma = \{\vartheta, g'.id, \Delta, False, c^{\Delta}_{g'}\}$ for VNF-FG g, with counter χ_g in orchestrator i. The message ς contains the sender ϑ, the identifier $g'.id$ for VNF-FG g, the reconfiguration operation Δ, the negative answer, and the counter $c^{\Delta}_{g'}$ for the reconfiguration for the replica of g. The orchestrator has a negation list l^i_g, and heap h^i_g.

1: $h^i_g.remove(\vartheta, \Delta)$
2: $l^i_g.append(\{\vartheta, \Delta, c^{\Delta}_{g'}\})$
3: $\Delta' \leftarrow \varphi_g.top()$
4: $g \leftarrow \Delta'(g)$

the VNF-FG converge eventually and both variants satisfy SEC (see Definition 2 from Section II-B). Recall the replicas of each VNF-FG for the 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. Shapiro et. al. defined these two conditions [26]. Definition 3 describes them.

Definition 3 (Eventual convergence conditions)
Two replicas x_i, x_j of a VNF-FG 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: $\forall i, j : \Delta \in C(x_i)$ implies that eventually, $\Delta \in C(x_j)$.*

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

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, ϵ. This is reflected with a special symbol in the top of all heaps. 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.

A. 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 [26]. *Safety*, for an operation-based CRDT, is guaranteed when concurrent operations satisfy the property of commutativity stated by Definition 4 [25]:

Definition 4 (Commutativity)
VNF-FG updates (Δ,x_i) and (Δ',x_j) commute, if and only if for any reachable replicate state ϕ, where both x_i, x_j are enabled, replica x_i remains enabled in state $\phi \circ x_j$ (respectably $\phi \circ x_i$), and $\phi \circ x_i \circ x_j \equiv \phi \circ x_j \circ x_i$, where $\circ$ is the symbol for function composition.

where ($i \neq j$), 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-FG until the previous update has not been accepted. This means that the orchestrator has received all positives answers (Algorithm 2 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 3 Lines 2-4). 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 4).

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 3 Line 3, Algorithm 2 Line 10). We need to show replies for a reconfiguration operation Δ commute.

Assume without loss of generality that a replica x_i has causal history for each affected orchestrator $C(x_i) = \{r_o\} \mid \forall o \in O$ (Algorithm 1 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 3 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.

B. 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 5 [25]:

Definition 5 (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 5 by proving Lemmas 3, 4, and 5.

Lemma 3
The CF-C payload is a join-semilattice. (Property 1 of Definition 5)

We show how Lemma 3 holds. By definition, the data structure we consider for accepted reconfigurations is a heap ordered by a partial $<$ operator, in other words a join-semilattice. Next, we show how the CF-C satisfies the third condition of Definition 5.

Lemma 4
The CF-C merge operation always computes the Least Upper Bound. (Property 3 of Definition 5)

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 g with a heap h_g^i for contingent reconfigurations, and a list l_g^i to keep track of invalid reconfigurations. We need to evaluate three cases when merging two heaps h_g^i, h_g^j from replicas x_i, x_j for the same VNF-FG g: (i) When the top of a heap is the initial state (i.e $\tau(h_g^i) = \epsilon, \tau(h_g^j) \neq \epsilon$), (ii) when the top of a heap has a greater reconfiguration number than another heap (i.e. $\tau(h_g^i) > \tau(h_g^j)$, (iii) when the top of both heaps has the same reconfiguration number (i.e. $\tau(h_g^i) = \tau(h_g^j)$). For the first case (i), since ϵ is the identity value, the merge operation computes the LUB between the two heaps (Algorithm 4 Line 4). For the second case (ii), 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 $\tau(h_g^i \cup h_g^j) = \tau(h_g^i), \tau(h_g^i) > \tau(h_g^j)$ otherwise $\tau(h_g^j)$; thus, for the second case the operator $<$ computes the LUB (Algorithm 5 Line 8). Finally, for the third case (iii), 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 5 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 5.

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

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. $|h_g^j| = 1$) and no negated elements (i.e. $|l_g^j| = 0$). 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_g^i according to the $<$ operator (Algorithm 4 Line 4, Algorithm 5 Line 9) or (ii) if it is not accepted, it is added to the negative list l_g^i (Algorithm 5 Line 12, Algorithm 6 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 $C(x_i) = C(h_g^i) \cup C(l_g^i))$; 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. $|h_g^j| > 0$), and a list of negated elements (i.e. $|l_g^j| > 0$). 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 $C(l_g^i) = C(l_g^i) \cup C(l_g^j)$ 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 $C(h_g^i) = C(h_g^i) \cup C(h_g^j)$. However, if any element e of a heap belongs to the union of negated lists ($e \in l_g^i \cup l_g^j$), this element is extracted from the heaps and added to the list such that after a merge $l_g^i = l_g^i \cup l_g^j \cup e$ (Algorithm 6), where e is the newest element. 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.

Acknowledgment

The research presented in this paper is supported by the Mexican Council for Science and Technology CONACYT (Grant 708000).

References

[1] K. Katsalis, N. Nikaein, and A. 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)*. IEEE, dec 2016, pp. 189–195. [Online]. Available: http://ieeexplore.ieee.org/document/7828601/

[2] R. V. Rosa, M. A. Silva Santos, and C. E. Rothenberg, "Md2-nfv: The case for multi-domain distributed network functions virtualization," in *2015 International Conference and Workshops on Networked Systems (NetSys)*, 2015, pp. 1–5.

[3] ETSI, NFVISG, "ETSI GS NFV 003 V1.4.1 Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV," 2018.

[4] N. F. Saraiva de Sousa, D. A. Lachos Perez, R. V. Rosa, M. A. Santos, and C. Esteve Rothenberg, "Network Service Orchestration: A survey," *Computer Communications*, vol. 142-143, no. May, pp. 69–94, jun 2019. [Online]. Available: https://doi.org/10.1016/j.comcom.2019.04.008

[5] L. M. Vaquero, F. Cuadrado, Y. Elkhatib, J. Bernal-Bernabe, S. N. Srirama, and M. F. Zhani, "Research challenges in nextgen service orchestration," *Future Generation Computer Systems*, vol. 90, pp. 20–38, 2019.

[6] ETSI, NFVISG, "GS NFV-SOL 004 V2. 3.1 Network Functions Virtualisation (NFV) release 2; Protocols and Data Models; NFV descriptors based on YANG Specification," 2019.

[7] O. Houidi, O. Soualah, W. Louati, and D. Zeghlache, "Dynamic VNF Forwarding Graph Extension Algorithms," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1389–1402, sep 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9079891/

[8] ETSI, NFVISG, "ETSI GS NFV 002 V1.1.1 Network Functions Virtualisation (NFV); Architectural Framework," 2013.

[9] ——, "ETSI GS NFV-IFA 010 V3.4.1 Network Functions Virtualisation (NFV) Release 3; Management and Orchestration; Functional requirements specification," 2020.

[10] 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)*, jan 2020, pp. 1–6.

[11] P. T. A. Quang, A. Bradai, K. D. Singh, G. Picard, and R. Riggio, "Single and Multi-Domain Adaptive Allocation Algorithms for VNF Forwarding Graph Embedding," *IEEE Transactions on Network and Service Management*, vol. 16, no. 1, pp. 98–112, 2019.

[12] P. T. A. Quang, A. Bradai, K. D. Singh, and Y. 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)*. IEEE, apr 2019, pp. 886–891. [Online]. Available: https://ieeexplore.ieee.org/document/8845184/

[13] 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)*, oct 2018, pp. 307–310.

[14] 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.

[15] J. C. Cisneros, S. Yangui, S. E. Pomares Hernández, J. C. P. Sansalvador, L. M. R. Henríquez, and K. Drira, "Towards Consistent VNF Forwarding Graph Reconfiguration in Multi-domain Environments," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021, pp. 355–366.

[16] K. Samdanis, A. Prasad, M. Chen, and K. Hwang, "Enabling 5G Verticals and Services through Network Softwarization and Slicing," *IEEE Communications Standards Magazine*, vol. 2, no. 1, pp. 20–21, 2018.

[17] P. Bailis and A. Ghodsi, "Eventual Consistency Today: Limitations, Extensions, and Beyond," *Queue*, vol. 11, no. 3, pp. 20–32, mar 2013. [Online]. Available: https://doi.org/10.1145/2460276.2462076https://dl.acm.org/doi/10.1145/2460276.2462076

[18] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-Art and Research Challenges," *IEEE Communications Surveys Tutorials*, vol. 18, no. 1, pp. 236–262, 2016.

[19] J. Cao, Y. Zhang, W. An, X. Chen, J. Sun, and Y. Han, "VNF-FG design and VNF placement for 5G mobile networks," *Science China Information Sciences*, vol. 60, no. 4, p. 40302, 2017. [Online]. Available: https://doi.org/10.1007/s11432-016-9031-x

[20] J. Baranda Hortiguela, J. Mangues-Bafalluy, R. Martinez, L. Vettori, K. Antevski, C. J. Bernardos, and X. Li, "Realizing the Network Service Federation Vision: Enabling Automated Multidomain Orchestration of Network Services," *IEEE Vehicular Technology Magazine*, vol. 15, no. 2, pp. 48–57, 2020.

[21] 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.*, jun 2005, pp. 1859–1864 vol. 3.

[22] H. Howard and R. 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*, pp. 8–10, 2020.

[23] 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, pp. 1432–1465, 2020.

[24] W. Vogels, "Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs?Between Consistency and Availability." *Queue*, vol. 6, no. 6, pp. 14–19, oct 2008. [Online]. Available: https://doi.org/10.1145/1466443.1466448

[25] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-Free Replicated Data Types," in *Stabilization, Safety, and Security of Distributed Systems*, X. Défago, F. Petit, and V. Villain, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 386–400.

[26] ——, "A comprehensive study of Convergent and Commutative Replicated Data Types," *Europe*, 2011. [Online]. Available: http://hal.archives-ouvertes.fr/inria-00555588/

[27] G. Zheng, A. Tsiopoulos, and V. Friderikos, "Dynamic VNF Chains Placement for Mobile IoT Applications," in *2019 IEEE Global Communications Conference (GLOBECOM)*. IEEE, dec 2019, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9014166/

[28] C. Kim, Y. Oh, and J. Lee, "Latency-based graph selection manager for end-to-end network service on heterogeneous infrastructures," in *2018 International Conference on Information Networking (ICOIN)*. IEEE, jan 2018, pp. 534–539. [Online]. Available: https://ieeexplore.ieee.org/document/8343176/

[29] 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, pp. 3330–3341, jul 2016.

[30] O. Soualah, M. Mechtri, C. Ghribi, and D. Zeghlache, "A Green VNF-FG Embedding Algorithm," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, jun 2018, pp. 141–149. [Online]. Available: https://ieeexplore.ieee.org/document/8460013/

[31] B. Spinnewyn, S. Latré, and J. F. Botero, "Delay-constrained NFV orchestration for heterogeneous cloud networks," *Computer Networks*, vol. 180, p. 107420, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128620311099

[32] F. Schardong, I. Nunes, and A. Schaeffer-Filho, "NFV Resource Allocation: a Systematic Review and Taxonomy of VNF Forwarding Graph Embedding," *Computer Networks*, vol. 185, no. July 2020,

p. 107726, 2021. [Online]. Available: https://doi.org/10.1016/j.comnet.2020.107726

[33] ETSI, NFVISG, "GS NFV-MAN 001 V1. 1.1 Network Function Virtualisation (NFV); Management and Orchestration," 2014.

[34] Y. Hao, Y. Li, X. Dong, L. Fang, and P. Chen, "Performance Analysis of Consensus Algorithm in Private Blockchain," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, jun 2018, pp. 280–285. [Online]. Available: https://ieeexplore.ieee.org/document/8500557/

[35] I. Alam, K. Sharif, F. Li, Z. Latif, M. M. Karim, S. Biswas, B. Nour, and Y. Wang, "A Survey of Network Virtualization Techniques for Internet of Things Using SDN and NFV," *ACM Computing Surveys*, vol. 53, no. 2, pp. 1–40, mar 2021. [Online]. Available: https://doi.org/10.1145/3379444https://dl.acm.org/doi/10.1145/3379444

[36] G. Gardikis, S. Costicoglou, H. Koumaras, C. Sakkas, A. Kourtis, F. Arnal, L. M. Contreras, P. A. Gutierrez, and M. Guta, "NFV applicability and use cases in satellite networks," in *2016 European Conference on Networks and Communications (EuCNC)*. IEEE, jun 2016, pp. 47–51. [Online]. Available: http://ieeexplore.ieee.org/document/7561002/