Received: 2 October 2018 | Revised: 30 January 2019 | Accepted: 8 March 2019

DOI: 10.1002/nem.2068

SPECIAL ISSUE PAPER

WILEY

Beyond VNFM: Filling the gaps of the ETSI VNF manager to fully support VNF life cycle operations

Giovanni Venâncio[1] | Vinícius Fulber Garcia[2] | Leonardo da Cruz Marcuzzo[2] | Thales Nicolai Tavares[2] | Muriel Figueredo Franco[3] | Lucas Bondan[3] | Alberto Egon Schaeffer-Filho[3] | Carlos Raniery Paula dos Santos[2] | Lisandro Zambenedetti Granville[3] | Elias P. Duarte Jr.[1]

[1]Department of Informatics, Federal University of Paraná, Curitiba, Brazil

[2]Department of Applied Computing, Federal University of Santa Maria, Santa Maria, Brazil

[3]Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre, Brazil

Correspondence
Giovanni Venâncio, Department of Informatics, Federal University of Paraná, Curitiba, Brazil.
Email: gvsouza@inf.ufpr.br

Funding information
Rede Nacional de Ensino e Pesquisa (RNP)

Summary

One of the main challenges to use network functions virtualization (NFV) is to properly manage the life cycle of the virtualized network functions (VNFs). Current solutions based on the ETSI standard NFV architectural framework are complex and require the knowledge of a myriad of details of the underlying infrastructure. This work proposes a VNF manager (VNFM) specification that supports different platforms and VNF utilization scenarios. The proposed VNFM specification defines a set of APIs, modules, and components, for both the end user and the back-end that provide the fundamental set of operations required to fully manage the VNF life cycle. Our primary goal is to simplify the operations, in particular reducing the need for the network operator to know the details of the virtualized infrastructure. Note that the proposed solution fills a gap and complements the ETSI VNFM module. A prototype was implemented, and experimental results show the effectiveness and low overhead of the proposed solution.

1 | INTRODUCTION

Network functions virtualization (NFV) is a paradigm that presents several advantages in comparison with traditional middle boxes, including the flexibility for service provisioning and the reduction of both operational (OPEX) and capital (CAPEX) expenditures.[1,2] NFV uses virtualization techniques to provide network services through virtual devices running on generic hardware (eg, x86 architecture). Thus, it is much simpler to make new services available as well as to modify and update existing services. There is no need for a proprietary or specialized hardware equipment, resulting in cost reduction and flexible service provisioning.[3]

Nevertheless, there are several challenges to the practical use of NFV technology, including the effective management and performance assurance of virtualized network functions (VNFs). The European Telecommunications Standards Institute (ETSI) has been working on an NFV architectural framework. One of the main components of that architecture is the VNF manager (VNFM), an entity responsible for managing the life cycle of VNFs. Management operations allow for example instantiate, delete, and update virtualized functions, but there are also other important operations, such as automatically scaling up/down resource usage and VNF recovery in case of failure.[4]

Several platforms have attempted to provide a suitable implementation for the ETSI VNFM component.[5-9] In general, these platforms require laborious work to execute tasks related to the life cycle management of VNFs, from the creation

of virtual machine descriptors to the manual configuration of the software that will execute the network function. All platforms require a deep understanding of the underlying infrastructure[10] and are complex to use. In addition, there are problems in terms of the interoperability in those platforms, which in general are difficult to be integrated with other systems. Finally, some of these solutions do not share a standard set of VNF life cycle management operations and thus VNF operators have no other option but the use of additional tools or manually manage network functions at the software level, ie, configuring system packages and executing multiple scripts inside the virtual device.

To overcome these shortcomings, in our previous work,[11] we defined a set of APIs to simplify the VNF life cycle management operations, such as those for creating, updating, and deleting VNFs. In this work, we propose a complete VNFM specification to provide, in addition to the previously defined APIs, a set of modules and components with well-defined functionalities for the VNF life cycle management. The proposed specification provides the fundamental set of operations and also defines how these operations should be handled to fully support VNF life cycle operations. These operations are accessible through a high-level API which is provided to the end user. Moreover, the VNFM itself uses two other APIs (also defined in the specification) to increase the flexibility and to simplify the compatibility of VNF management with different NFV platforms, making it easily adaptable to new scenarios and environments. By using the proposed APIs, the network operator is not required to understand all details of the underlying virtualization infrastructure. Furthermore, VNF management operations are simplified, once this specification abstracts several management procedures. Additionally, our VNFM specification uses all the defined APIs to autonomously control the entire VNF life cycle, from the hardware to the software level, while being fully compliant to the ETSI specifications.

As a proof-of-concept, a prototype implementation is presented and evaluated. In particular, details of how the VNFM uses the proposed APIs to implement the instantiation, deletion, and update of VNFs are described. Experimental results are presented and show the effectiveness of the solution for performing all VNF management operations, as well as its low overhead.

The rest of this work is organized as follows. In Section 2, a brief background on NFV management and orchestration platforms is presented. In Section 3, the proposed VNFM specification is described, including details of its internal components. In Section 4, the prototype implementation is detailed, and experimental results are discussed. Finally, the conclusion follows in Section 5.

2 | BACKGROUND AND RELATED WORK

According to the ETSI, NFV technology must support the execution of VNFs developed by different vendors.[12] This is achieved by supporting standardized interfaces to abstract the computational resources required to execute VNFs.[4] The ETSI reference architecture is composed of three main blocks: VNFs; NFV Infrastructure (NFVI), and NFV Management and Orchestration (NFV-MANO).

The Management and Orchestration Module (NFV-MANO) provides functionalities related to the provisioning and management of VNFs, as well as management of the infrastructure where those VNFs will be deployed, being responsible for both virtualized and physical resources of the infrastructure. Figure 1 shows the NFV-MANO internal architecture, which is composed of three functional blocks: virtualized infrastructure manager (VIM), VNF manager (VNFM), and NFV orchestrator (NFVO). Those blocks are supported by a collection of data repositories (eg, NS catalog, VNF catalog, NFV instances, and NFVI resources) where information about the state of the entire environment is stored. The modules also share information in order to maintain a updated state of the infrastructure.

Regarding the internal modules, the VIM module manages a NFV infrastructure, keeping track of physical and virtual resources allocated and used in a given domain, allowing it to optimize the resources used. The NFVO module orchestrate multiple functions in order to deploy an end-to-end service. This way, the NFVO provides abstraction to NFVI resources across multiple VIMs, as well as automation of the provisioning of different VNFs and topologies which comprises a service. Finally, the VNFM module manages the life cycle of VNF instances. A VNFM can be associated with multiple VNFs and has access to a VNF directly or through an element management system (EMS). Moreover, VNFM operations must be supported by every VNF, regardless of their implementation.[4] Examples of tasks performed by the VNFM module are instantiation, initial configuration, deletion, updating, and scaling instances of virtualized functions. It is worth noting that an NFVO offers a high-level abstraction to manage and monitor end-to-end network services. However, the VNFs are effectively controlled by the VNFM, which can receive commands from the NFVO and individually configure the VNFs to deploy an NS.

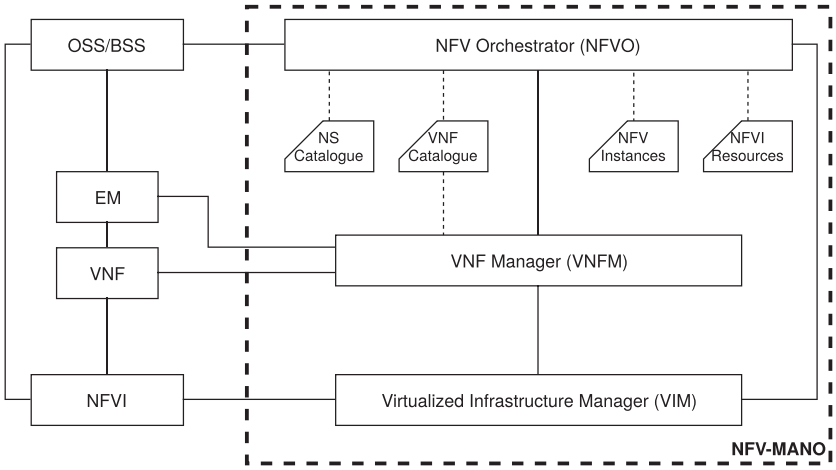

FIGURE 1 NFV-MANO module

Multiple systems that implement NFV-MANO solutions have been proposed recently.[5-7] Although most of these platforms are fully compliant with the ETSI specification, their operation requires an extensive knowledge of the underlying infrastructure, with network operators having to understand the entire VNF workflow (ie, sequence of system commands) in order to deploy and manage VNFs and service function chains (SFCs). Additionally, another important feature of NFV platforms is VNF placement,[2,13] which is not present in most of these solutions. VNF placement aims to choose the best instantiation site for a VNF in order to optimize physical resource utilization and performance of services.[14] Moreover, another important aspect in offering an NFV solution is the interoperability, ie, the ability of the platform to interact with multiple vendors solutions transparently, allowing for example the use of different VIMs (eg, OpenStack, OpenVIM[15]) or different VNFMs.

Open Source MANO (OSM)[7] presents implementations of all NFV-MANO's operational elements (ie, NFVO, VNFM, and VIM) to support the configuration and abstraction of VNFs, as well as resource orchestration. OSM provides a generic VNFM that can be integrated with other specific VNFMs, but it is necessary to carefully configure the NFV orchestrator to achieve complete consistency. Despite those numerous capabilities, OSM usage is challenging to use due to its lack of flexibility. For example, in order to deploy a single VNF, operators must first upload a disk image to the VIM, onboard this disk VNF, onboard a VNF descriptor (VNFD), and finally instantiate the VNF. This is an overly complex process that, although allowing a fine granularity for configuring each step, may introduce errors by less experienced network operators.[16]

The Open Baton platform[6] also implements the NFV-MANO elements based on the ETSI specification. The platform consists of an NFVO, a VNFM, and support for multiple VIMs. In addition to VNF life cycle monitoring and control, Open Baton uses routines defined by a VNF package to configure each network function. In other words, it is possible to internally change the state of any software executed by a Virtual Machine (VM). Even with the possibility of using heterogeneous VIMs and the provisioning of a friendly user interface, the Open Baton platform does not present any tool for VNF chaining.[17] Due to the lack of support to SFCs, it is hard to create, provide and manage complex services using this platform.

Tacker[5] is a platform that implements both a VNFM and a NFVO integrated to OpenStack (acting as VIM)[18] to offer management operations. Among the implemented VNFM functionalities are the control of basic function life cycle, monitoring, scaling, and initial configuration of VNFs. The module that acts as NFVO is able to instantiate and scale VNFs according to policies defined by the network operators, as well as to manipulate complex structures of network services, ie, SFCs.[19] Despite implementing many MANO's functions, Tacker can only perform the life cycle control of the VM where the VNF is running, not offering a standard API to configure the VNF software level (eg, initialize scripts, install system packages, reconfigure a function). Moreover, internal statistics of the VNF, such as packet count, rules, CPU and memory usage, plus VNF state are not monitored by Tacker and cannot be retrieved by it; only the status of the instance is shown. Also, similarly to OSM, the process of onboarding a VNF requires multiple steps which can be difficult for a network operator to understand.

Open Platform for NFV (OPNFV)[20] is another solution from the Linux Foundation that builds a platform to facilitate the development and execution of NFV components and VNFs. The OPNFV project aims to provide a reference NFV

TABLE 1 Qualitative comparison between VNF managers

	OSM	Open baton	Tacker	OPNFV
ETSI compliant	✓	✓	✓	✓
Complete management		✓		
Interoperability	✓			✓
Placement algorithms			✓	✓

platform that leverages multiple NFV solutions from different vendors. Although the OPNFV provides high interoperability between different NFV platforms, its deployment and configuration are complex. For example, to instantiate a VNF, it is necessary to configure multiple parameters manually, such as the network interfaces and the initialization of the scripts responsible for VNF functionality.

Table 1 shows a qualitative comparison of the main NFV solutions described in this section. Most of the existing platforms are ETSI compliant and offer a large set of operations that can be applied in the management of the NFV architecture. Nevertheless, typically they are not concerned about ease of use, leaving to the network operators the responsibility to learn various specific low level operations which must be executed in the right way. In addition, management is not always done in both the hardware and software levels. In most cases, platforms do not have a specific set of functions to allow software configuration of VNFs.

In conclusion, no VNFM available so far offers a comprehensive and complete set of functionalities that have been specified by the ETSI reference architecture. In this work, we propose a VNFM specification that addresses the problem of VNFM complexity and provide high level life cycle management operations.

3 | VNF MANAGER SPECIFICATION

In this section, the VNFM is specified. First, in Section 3.1, the VNFM architecture is presented and discussed, including a detailed description of components and their functionalities. Section 3.2 defines the three communication APIs. Finally, Section 3.3 describes how the communication APIs are used to fully manage the VNFs life cycle.

3.1 | VNFM architecture

One of the main goals of the VNFM is to control the life cycle of VNFs, allowing the instantiation, deletion, and (re-)configuration of the virtual devices (eg, VMs, containers) that host VNFs. The VNFM also manages and monitors VNF functionalities, besides allowing the configuration and execution of the software that performs the virtualized function. Current platforms for the VNF management require users to execute a series of exhaustive procedures, including the creation and deletion of VNF descriptors, detailed configurations at software level, and procedures related to the execution of the VNF itself. Moreover, the user is required to grasp a large amount of information about the operation of the system itself.[10] This information varies from knowledge about network configuration details to specific system parameters.

As an example of a management operation, consider the instantiation of a VNF. To perform this task on most platforms, the user is usually required to execute a series of commands that involve registering descriptors (eg, VNF Descriptors) and also the instantiation of the virtual machine where the function will be hosted. Furthermore, some platforms do not automatically initialize the VNF functionalities, requiring the manual execution or the development of additional tools that carry out software-level configuration and the execution of the network function. In this sense, this work proposes a VNFM specification that uses a set of modules and APIs to fully manage VNFs based on requests issued by the user. Considering our previous VNF instantiation example, when using the proposed VNFM specification, with a single request, the user fully creates the VNF and configure the function at the software level.

Figure 2 depicts the detailed VNFM architecture. This architecture consists of two layers: (a) User Layer and (b) NFV Layer. At the User Layer, the end user communicates with the NFV layer provided by any platform that offers user-level services (eg, marketplaces[21] and tools for NFV visualization[22]). Those platforms communicate with NFV-MANO modules through the Management API, which offers features to control the entire VNF life cycle, and plus monitoring and efficient resource management.

The NFV Layer consists of an extended NFV-MANO architecture, similar to that presented in Section 2. We extend the architecture by specifying internal modules and components of the VNFM, as well as a set of communication APIs. In

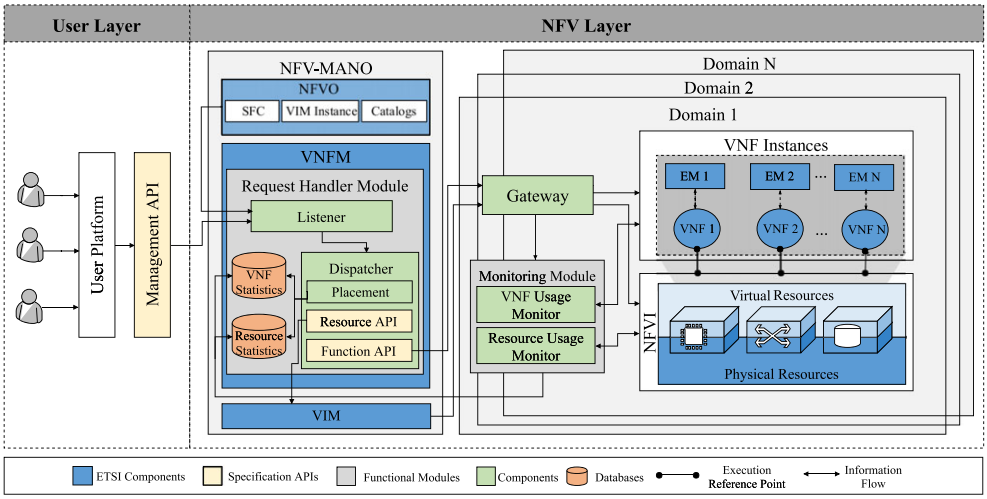

FIGURE 2 The VNFM architecture

order to allow straightforward VNF instantiation and execution, a key module called *Request Handler Module* (RHM) is proposed. The RHM is the core module of the VNFM and it is responsible for receiving requests directly from the User Layer. From these requests, the RHM automatically performs all necessary procedures related to the VNF life cycle. The RHM is composed of three main components: *Listener*, *Dispatcher*, and the NFV monitoring databases.

Every request made to the Management API is received by the *Listener* component, which is responsible for interpreting and storing the requests in a queue. Each request is then forwarded to the *Dispatcher* component, which is the main component of the RHM. The *Dispatcher* aims to abstract and autonomously control the management operations performed at the User Layer. Particularly, for every operation available at the Management API, the *Dispatcher* knows exactly which procedures should be executed, and therefore, no other user intervention is required.

Upon receiving a request, the first step performed by the *Dispatcher* is to verify if the operation is a VNF instantiation. If it is not, the operation relies on an existing VNF, and the *Dispatcher* needs to make a query to the local catalog which contains the corresponding VNF to get domain information (eg, IP address, authentication credentials), in order to continue with the management procedures. If it is an instantiation, the first step is to define in which domain the VNF should be instantiated. In this sense, the *Dispatcher* has a VNF placement functionality, which decides the location where the VNF will be hosted. VNF placement is specified through a parameter in the Management API and can be of two types. The first type corresponds to manual placement, where the user specifies exactly which domain will host the VNF. The second type is the parameterized placement, in which the *Dispatcher* itself analyzes the best VNF instantiation site, based on a user-specified parameter. Some possible parameters are the lowest CPU and memory usage, latency, and bandwidth usage. Also, more complex VNF placement algorithms[23,24] could be employed. In the parameterized VNF placement, the *Dispatcher* uses information from the monitoring databases to decide the best domain among all those available to instantiate the VNF. The monitoring databases are described as follows.

We specify two NFV monitoring databases to store relevant information for the management operations: Resource Statistics and VNF Statistics. In Resource Statistics, the monitoring metrics collected are related to the physical resources available on each domain, such as CPU, memory, and disk usage. In turn, VNF Statistics stores metrics related to the VNF function usage, such as the number of packets processed and operations latency. Once these metrics are specified, it is possible to perform VNF management routines (eg, on-demand auto-scaling, failure recovery, VNF placement).

Once the RHM acquires information about where to perform the management operation, the *Dispatcher* starts to manipulate two low-level APIs in order to proceed with the management operation. These two APIs, named Resource and Function API, are only used by the VNFM itself and are not manipulated by the end user in any way. The Resource API is responsible for configuring the entire hardware environment of the VNF (ie, virtual device management) while the Function API is responsible for configuring the software level (ie, initializing and configuring the VNF function). Details about the communication APIs are given in Section 3.2.

All requests made by the *Dispatcher* through the Resource and Function API are forwarded to the *Gateway* module, placed at each domain. The Gateway is responsible for receiving requests forwarded by the defined APIs and performing the corresponding operations on the domain. In particular, the Gateway performs three sets of operations: (a) NFVI

modifications; (b) VNF Instances modifications; and (c) Monitoring Module requests. The first set of operations represents all the modifications required by the VIM through the Resource API that the Gateway must perform in the NFVI. These modifications are related to the virtual device that hosts the VNF, such as the creation of a new VM or updating the amount of physical resources. In the second set of operations, the Gateway is responsible for applying modifications requested by the *Dispatcher* through the Function API directly in the VNF Instances. The purpose is to manage the VNF at the software level, modifying the VNF function itself. For example, the Gateway can insert scripts into the VM and install system packages necessary for the correct execution of the function. Finally, the last set of operations are requests made to the Monitoring Module, adding or removing monitoring probes to or from the VNFs.

The Monitoring Module consists of two components which act as monitoring agents. The first agent is the VNF Usage Monitor, responsible for collecting metrics specific to the VNF function usage. This information is then stored in the VNF Statistics monitoring database at the RHM. The second is the Resource Usage Monitor and is used to collect metrics about system resource usage, which is also stored at the RHM in the Resource Statistics database. Both agents collect data every time interval, which should be customizable based on the real-time data requirements.

The next subsections will detail the three APIs defined as well as describe how they are handled by the *Dispatcher* to automate management operations for the User Layer.

3.2 | Communication interfaces

The VNFM architecture described in Subsection 3.1 is based on three APIs that simplify and improve the flexibility of life cycle operations. One of these APIs is the *Management API*, which is used by the end user to manage the VNFs life cycle. The other two APIs are called Resource and Function API, which are responsible for FCAPS (Fault, Configuration, Accounting, Performance, Security) management of the VNFs. These two APIs are used only by the VNFM itself (in special, by the RHM) to manage the life cycle of VNFs at both hardware and software levels, autonomously dispatching the necessary procedures of the requested operation. As an example, consider that the user has requested a VNF instantiation. The RHM receives and processes this request and, without any intervention of the user, executes the necessary procedures related to the instantiation, such as the creation of the VM and software configuration and initialization. Note that the functions presented in these APIs are the fundamental set of operations to control the entire life cycle of VNFs. However, these APIs can be easily extended according to the need of the network operators.

The Management API defines a set of high-level operations that provide an interface to allow the user to control the entire VNF life cycle. In this way, the user is not required to execute multiple manual procedures that changes from one system to another. Moreover, the user does not need to specify multiple minute system parameters. All features provided by the Management API are described in Table 2. Although one of the main objectives of the specification proposed in this work is to simplify and abstract NFV management procedures, there are some parameters that the user must specify when executing operations in the Management API.

The first parameter is the *domain*, which consists of an IP address and an authentication token. As described in Section 3.1, upon creating a VNF, it is possible to choose the instantiation site either manually or automatically, based on a parameter. If the manual option is selected, the user defines, through the *domain* parameter in which domain the VNF will be hosted. If the option is automatic, the user chooses the instantiation site in a parameterized way. In this case, the value of the parameter depends on the information in the NFV monitoring databases. For example, CPU, memory, latency, or bandwidth usage can be considered parameters for instantiation. The second parameter to be defined by the user is the VNF Descriptor (VNFD), which is a template that is responsible for specifying a VNF in terms of operational and deployment requirements.[4] VNFDs also define other VNF aspects, such as network configurations and computational resources. In general, VNFDs are based on the TOSCA (*Topology and Orchestration Specification for Cloud Applications*) standard.[25] The third parameter is the *vnf_function*, which is used to send all the necessary files to execute the VNF, such

TABLE 2 The Management API

Operation	Description
create(domain, vnfd, vnf_function)	Create a VNF
migrate(domain, vnf_id)	Migrate a VNF to another domain
resource_update(vnf_id, vnfd)	Update VNF resources
function_update(vnf_id, vnf_function)	Update VNF function
delete(vnf_id)	Delete a VNF

as scripts and configuration files. Finally, when creating a VNF, a VNF Identifier (*vnf_id*) is returned by the Management API, which will be used in the other operations to reference the instantiated VNFs.

Once requests are received by the Management API from the User Layer, the VNFM communicates with several other modules integrating the ETSI NFV architecture. One of these modules is the VIM, a subsystem that manages resources and controls VNF interactions with the virtualized environment. Existing VNFM implementations usually communicate with the necessary modules for VNF life cycle control through specific REST (REpresentational State Transfer) APIs. In general, these implementations are not flexible, since each VNF management platform provides a solution that implements the set of functionalities necessary for the VNFM to communicate with these modules. As a consequence, integrating other technologies with those systems is a laborious and complex task, which requires great efforts from developers to adapt the tools to be used. To overcome these limitations, we propose two other APIs, the Resource API and the Function API, that improve the extensibility and flexibility of the VNFM.

The Resource API defines the necessary functionalities for the VNFM to be able to manage the physical resources of the virtual machines that will host the VNFs. Table 3 describes the minimum set of operations that should be implemented in this API. In general, this API is provided through a plug-in or driver on the VIM itself. Functions in the Resource API usually require only two parameters. The first is the VNFD, which is defined and sent by the user through the Management API. Upon creating a VNFD, a VNFD Identifier (*vnfd_id*) is returned, which is used later to reference VNFDs in the other operations.

Finally, the Function API defines the functionalities that the VNFM must perform at the software level. Table 4 describes the operations that are implemented by this API. In particular, through this API, it is possible to define customized monitoring metrics obtained by the VNF Usage Monitor, such as the number of packets blocked in a firewall. In general, VNFs themselves should provide the implementation of these functions, so that the VNFM can collect data and execute instructions. Usually, the only required parameters in this API are the *vnf_function* parameter, obtained from the Management API and the IP address of the VNF (*vnf_ip*), extracted from the *domain* parameter.

The APIs required to fully manage the life cycle of VNFs can be implemented using different technologies available from different vendors. As a result, this reduces the complexity for adopting new virtualized functions and also facilitates the implementation of new network services and routines on different infrastructures. For example, the prototype implemented in this work (described in details in Section 4.1), uses OpenStack (as a VIM) and Tacker (as the Resource API) as the main NFV solutions. However, these solutions could be replaced by others that provide the same features defined in the Resource and Function API, without any changes in the VNFM and consequently improving the interoperability of the platform.

TABLE 3 The Resource API

Operation	Description
vnfd_create(vnfd)	Create a VNF Descriptor
vnfd_delete(vnfd_id)	Delete a VNF Descriptor
vm_create(vnfd_id)	Create a VM based on VNFD
vm_configure(conf)	Configure a VM
vm_delete(vnf_id)	Delete a VM
vm_update(vnf_id, vnfd)	Update a VM template
vm_reboot(vnf_id)	Reboot a VM
vm_list()	List all VMs
vm_show(vnf_id)	Get VM information

TABLE 4 The Function API

Operations	Description
write_function(vnf_ip, vnf_function)	Insert VNF function on VM
start_function(vnf_ip)	Start VNF function
stop_function(vnf_ip)	Stop VNF function
get_metrics(vnf_ip)	Get VNF function usage metrics
get_log(vnf_ip)	Get VNF log
get_running(vnf_ip)	Verify if VNF function is running

3.3 | Management operations

The following subsections describe how the VNFM handles the APIs described in Section 3.2 to fully manage VNFs, from the hardware to the software levels. In particular, it is described how the RHM autonomously controls the management operations.

3.3.1 | VNF instantiation

There are two phases for the instantiation of a VNF: (a) creation and configuration of the virtual machine (Resource API) and (b) execution of the virtualized function (Function API). The first phase consists of two steps, the first step starts when the RHM executes the *vnfd_create(vnfd)* operation to add the VNFD received from the Management API to the catalog of VNFs, generating a unique identifier called *vnfd_id*. Then, the RHM calls the *vm_create(vnfd_id)* operation to create a virtual machine based on the VNFD created earlier. Given that the time to initialize the virtualized infrastructure varies for each system, the RHM runs a polling mechanism, that checks at each time interval whether the virtual machine is active. When the polling mechanism detects that the virtual machine is ready and no errors have occurred, the RHM starts the second phase.

The second phase starts with the RHM inserting the VNF function, also received as a parameter, in the virtual machine through the execution of the *write_function(vnf_ip, vnf_function)* operation. Once inserted, the VNF function is executed with the *start_function(vnf_ip)* operation and is now fully functional. Finally, if no error occurs, the RHM adds this VNF to the catalog of VNF instances, so that this VNF can be monitored by the monitoring module. For each procedure executed, it is verified whether it has been completed successfully. Otherwise, the RHM issues an error message and starts a rollback mechanism that will undo all modifications made so far (eg, remove VNFD, databases entries).

3.3.2 | VNF deletion

The removal process starts when the RHM receives a VNF deletion operation with a unique VNF Identifier (*vnf_id*) and removes the corresponding VM through the *vm_delete(vnf_id)* operation. Similarly to the instantiation, the RHM starts the polling mechanism to periodically verify the proper removal of the virtual machine. The VNF is then removed from the catalog of VNF instances and is no longer monitored by the monitoring modules. Finally, the VNFD of the VNF is removed from the catalog.

3.3.3 | VNF update

VNF updates can be classified in two types: (a) physical resource update (Resource API) and (b) update of the virtualized function (Function API). In the case of resource updating, the RHM receives the *vnf_id* and the updated VNFD and issues an update on the virtual machine hosting the VNF. The RHM then uses the *vm_update(vnf_id, vnfd)* operation to update several system parameters, such as the number of CPUs, and the amount of RAM and disk. Once the update is completed, the RHM reboots the virtual machine and concludes the update operation.

For the update of the VNF function, the RHM receives the *vnf_id* and the updated virtualized function and inserts the new function into the virtual machine, in the same way as in the instantiation. Afterwards, the old function is stopped with the execution of the *stop_function(vnf_ip)* operation and the new function is executed.

4 | EXPERIMENTAL EVALUATION

As a proof-of-concept, a prototype of the proposed VNFM specification was implemented and used to conduct an empirical evaluation. In Section 4.1, we show details of the implementation. In Section 4.2, the evaluation scenario is presented. Finally, Section 4.3 shows experimental results.

4.1 | Implementation of the prototype

As described in Section 3, the VNFM specification consists of the *Request Handler Module*, which uses the Management, Resource, and Function APIs to communicate with the other components. The RHM is responsible for receiving

high-level requests from the Management API to perform the necessary operations through the Resource and Function APIs. In our prototype, requests are made using REST, and all configuration data are defined in JSON. The communication between modules can be either internal (ie, centralized components execution) or external (ie, distributed components execution). This way, RHM components can be executed in a distributed or centralized manner. All modules and components were implemented using Python. As for the VIM module, we use OpenStack by default.

The RHM contains three main components: *Listener*, *Dispatcher*, and the NFV monitoring databases. The *Listener* implements, through the Eve* Python library, a RESTful Web Service to receive requests from the Management API. All incoming requests are first stored in a queue and then forwarded to the *Dispatcher*—a back-end component that performs NFVI modifications through REST requests to the Resource and Function API. The *Dispatcher* creates a new process for each request received, allowing parallel execution of the management operations. The *Dispatcher* is also responsible for controlling the status of operations performed. If errors occur during the execution of management operations, the *Dispatcher* has a rollback mechanism to undo all the procedures performed so far. Thus, the *Dispatcher* keeps a log of the operations performed, which is used to undo the procedures related to that operation.

For the Resource API, all operations described in Table 3 were implemented. In particular, we have used and extended Tacker to provide all the virtual device management features. Likewise, the Function API implements all the operations described in Table 4. In our prototype, VNFs based on Click-on-OSv[26] were used, which have a REST Interface through which the RHM performs the software configuration of the VNF. However, VNFs based on any other operating system could be employed. Finally, the NFV monitoring databases store the metrics collected from the NFVI and from the VNF instances. This was implemented using the InfluxDB† database, which processes collected data in real time.

In addition to the RHM, the VNFM specification defines a monitoring module that is executed on the NFVI itself. This monitoring module uses two agents to collect information about the NFV environment: VNF Usage and Resource Usage Monitor. Both agents communicate with the VNFs at a periodic time interval. We set a 30-second time interval by default, which for our needs represents a satisfactory number of data collected. This value however can be customized as needed. The collected data are sent to a central server (ie, NFV monitoring databases in the RHM) where the data are processed.

The prototype deployment is straightforward and consists of only two stages. The first stage is the configuration of the NFVI, which includes the VIM credentials, controller node URL, database connection, and customizable parameters of VNFM, such as polling and monitoring time intervals. Figure 3 shows a minimal VNFM configuration
file.

In the second stage, the user must set the Resource and Function API endpoints that corresponds with the NFV environment. Figure 4 shows a minimal example of endpoint configuration. These endpoints set the URL to perform the corresponding function, allowing interoperability between different NFV platforms. Finally, after configuring these files, the RHM service must be initialized so that the VNFM is fully operational.

4.2 | Evaluation scenario

Our evaluation scenario is based on three servers that are geographically distributed over the country. One server is the controller node, which hosts the implementation of our VNFM and its modules (eg, RHM, databases). The other two servers are used as NFVI (ie, compute nodes) and host OpenStack/Tacker, in addition to the other modules proposed by our specification (eg, Gateway, Monitoring Module). VNFs are also instantiated in these compute nodes. All three servers have Intel(R) Xeon(R) CPU E3-1220 @ 3.00GHz with four cores, 8 GB memory, and are configured with Ubuntu 18.04 and OpenStack Rocky.

The prototype was evaluated considering four main high-level operations: `Create`, `Delete`, `Resource Update`, and `Function Update`. The prototype evaluation was performed as follows. The `Create` operation was executed by instantiating a Click-on-OSv VNF with a firewall, allocating 4 GB of RAM, 2 CPU, and 10 GB of disk. Next, the `Resource Update` operation was performed to increase the allocated RAM from 4 GB to 512 MB, while the remaining resources kept unchanged. Then, the `Function Update` operation was evaluated by changing the running function to a traffic shaper. Finally, the VNF was deleted using the `Delete` operation. Experiments were executed 30 times, and results are presented with a confidence interval of 95%.

*http://python-eve.org/
†https://www.influxdata.com/

```
[auth]
username = nfv_user
password = mypassword
tenant_name = nfv_tenant

[openstack]
url = http://192.168.0.58/

[vnfm]
polling_interval = 1
polling_timeout = 300
monitoring_interval = 1
request_timeout = 2

[database]
address = 192.168.0.58:8086
```

FIGURE 3 VNFM minimal configuration file

```
[resource_endpoint]
identity = identity/v3/auth/tokens
vnfd_create = v1.0/vnfds
vnfd_delete = v1.0/vnfds/{vnfd_id}
vnf_create = v1.0/vnfs
vnf_delete = v1.0/vnfs/{vnf_id}
vnf_update = v1.0/vnfs/{vnf_id}
vnf_show = v1.0/vnfs/{vnf_id}

[function_endpoint]
get_running = {vnf_ip}:8000/running
get_log = {vnf_ip}:8000/log
get_metrics = {vnf_ip}:8000/metrics
write_function = {vnf_ip}:8000/write_file
stop_function = {vnf_ip}:8000/stop
start_function = {vnf_ip}:8000/start
```

FIGURE 4 Resource and Function API endpoints

4.3 | Results and discussion

Since VNFs must be instantiated, configured, and dynamically scaled, the execution time of these operations is a relevant metric. Thus, the total execution time of each operation was measured. As a baseline, we also measured the operation time using only Tacker. However, note that our approach does several other tasks that Tacker does not do, as described in Section 2. Moreover, the execution time of the individual tasks composing these management operations was also evaluated.

However, note that our approach has several other functionalities that Tacker does not do, such as rollback mechanisms, software-level configuration, and multiple placement algorithms, as described in Section 3.

As can be seen in Figure 5, the `Create` operation took the longest time to execute, with the polling task consuming around 54% of the total execution time of the operation, according to Figure 6. This occurs because when the *Dispatcher* component is creating the virtual machine, the polling task should periodically check and wait for the NFVI to complete the instantiation process, so that the virtualized function can be configured. On the other hand, the `Function Update` operation needs to upload a new network function and wait for that function to start. Finally, the `Resource Update` operation, after updating the VNFD, must wait for the VNF to reboot, while the `Delete` operation sends a request for the VM to be deleted.

In this way, we can conclude that the overhead introduced by our solution when compared with the Tacker platform is low. For the operation `Create`, the increase was 4.3%. For the operations `Resource Update` and `Function Update`, the increase was 3.6% and 2.1%, respectively. Finally, for the operation `Delete`, the difference was 12.6%. However, note that this overhead point out to the other tasks that the RHM module performs to fully manage the VNFs life cycle. Taking the instantiation operation as an example, the 4.3% overhead of our approach represents the entire software configuration to make the VNF fully functional (ie, Function API tasks). Tacker in turn requires the VNF to be manually accessed and configured.

The previous experiment measures the execution time of a single management operation. However, we must consider that a real production environment receives multiple requests, possibly in parallel. Thus, the next experiment measures the total execution time of one up to 32 requests in parallel. All requests are VNF instantiations (ie, `Create` operation),

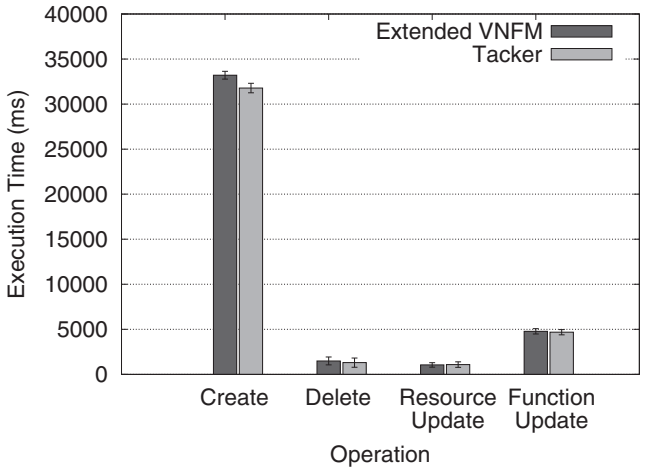

FIGURE 5 Total execution time of the operations

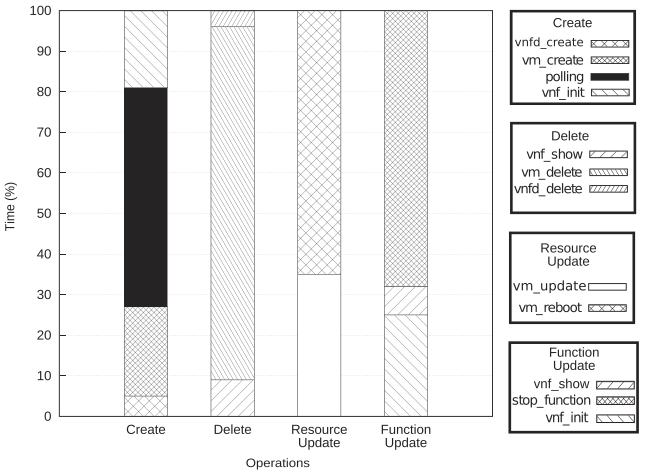

FIGURE 6 Execution time of tasks—percentages

since it is the operation that represents the highest computational cost. Moreover, this experiment considers two scenarios: (a) all VNFs instantiated on a single server (1 Compute Node); (b) VNFs are instantiated on two different servers (two Compute Nodes). Note that the maximum number of requests is 32 due to the limit of computational resources available in each compute node.

Figure 7 shows the variation in the total instantiation time of all VNFs as the number of requests increases. For the scenario with a single compute node, the difference in execution time between one and 32 requests is only 10.4 times higher. This is due to the fact that the RHM handle the request queue in parallel, as described in Section 4.1. In comparison to a sequential approach, which is used by some NFV platforms, the total expected time to instantiate 32 VNFs would be 1056 seconds. On the other hand, our prototype instantiates the same 32 VNFs in just 345 seconds (one-third of the time).

In the second scenario where two compute nodes are used, the difference is even lower: the total instantiation time between one and 32 VNFs is only 6.3 times higher. Although the result is already expected since VNFs are instantiated in two different locations, this shows that our approach also ensures good performance in distributed scenarios.

The use of NFV also depends on the use of high-density virtualization servers, and a shortage of computational resources may occur if many VNFs have been instantiated. Thus, the consumption of resources by our prototype should be minimal, considering that the OpenStack platform itself already has high execution requirements. Therefore, both CPU and memory consumption of the prototype were measured using the *psutil* Python library. Figure 8 shows both the amount of memory and CPU usage of each operation.

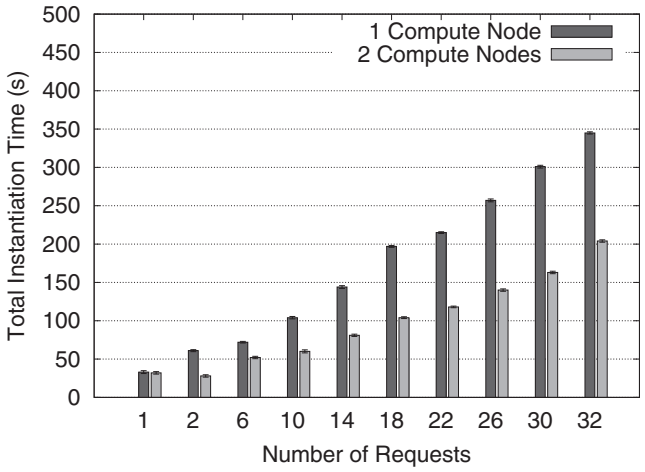

FIGURE 7 Instantiation time with multiple requests in parallel

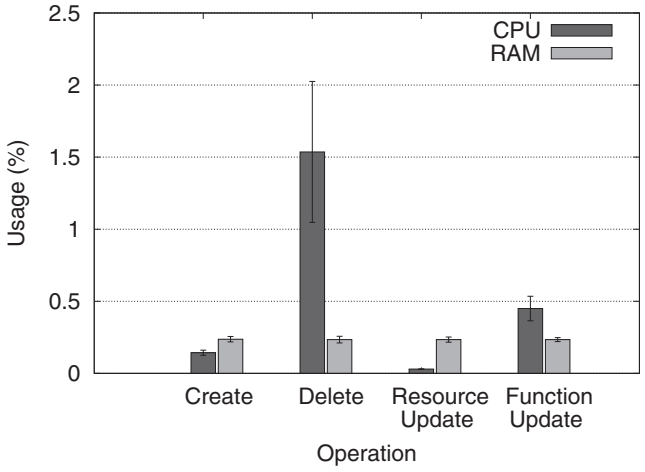

FIGURE 8 CPU and memory usage during operations

Regarding RAM usage, most of the memory consumed is due to the Python libraries. As these libraries are used by all operations, memory consumption is similar for all of them. For CPU usage, the `Delete` operation presents the highest CPU utilization, while the other operations, even with longer tasks, have lower CPU consumption. This occurs because in the case of the `Create` operation, although the polling task takes the longest, the RHM only waits for the response of the Resource API, doing no real processing in this interval. Similarly, the `Update Function` operation must wait for the VNF reboot to finish. This way, since the `Delete` operation it is not I/O bound, it does not have to wait for any type of system call; therefore, it is processing the whole time. Finally, the `Resource Update` operation does not have a significant CPU consumption, as it only sends the updated VNFD to the Resource API and waits for the VNF reboot, doing no processing during this time.

Therefore, the highest consumption among all operations peaked at only 1.5% of CPU usage for less than one second, while the RAM consumption was up to only 0.25% of the available memory. This indicates that the impact of the prototype on the host resources is minimal.

5 | CONCLUSION

Several platforms currently implement NFV-MANO architecture components, providing multiple operations to manage VNFs. However, most fail to provide NFV management at both the hardware and software levels. Moreover, these platforms are complex to use, since they require a large amount of information and procedures to be performed by network operators and do not allow the integration of different NFV platforms in a flexible way. In this work, we introduced a VNFM specification to simplify the management of VNF life cycle operations. By defining several APIs, this specification solves the compatibility issues among different NFV platforms besides reducing the number of procedures required by the user to fully manage VNFs. Our solution has advantages not only for human operators but also for NFV Orchestrators and other NFV modules. On the other hand, a disadvantage is that as we make it easier for the operator to manage VNFs by alleviating the configuration tasks, as other platforms require the operator to do it all by themselves they are certainly more customizable. Additionally, the proposed specification allows the management of virtualized functions at both hardware and software levels. It is important to note that the defined APIs provide the fundamental set of operations to control the life cycle of VNFs and they can be extended according to the need of the system.

As a proof-of-concept, a prototype was implemented to run an empirical evaluation. Experimental results show that our VNFM can manage the life cycle of VNFs by consuming a minimal amount of resources, averaging less than 1% CPU and 0.25% of RAM. Furthermore, the time to execute the main operations has been measured, and it is possible to conclude that the prototyped VNFM is effective and efficient. Future work includes extending the functionality of the proposed specification to include both the composition and orchestration of VNFs.

ACKNOWLEDGEMENT

The authors would like to thank Rede Nacional de Ensino e Pesquisa (RNP), for their support to the GT-FENDE project.

ORCID

Giovanni Venâncio https://orcid.org/0000-0001-7620-3793

REFERENCES

1. Mijumbi R, Serrat J, Gorricho J-L, Latré S, Charalambides M, Lopez D. Management and orchestration challenges in network functions virtualization. *IEEE Commun Mag*. 2016;54(1):98-105.
2. Bari F, Chowdhury SR, Ahmed R, Boutaba R, Duarte OCMB. Orchestrating virtualized network functions. *IEEE Trans Netw Serv Manag*. 2016;13(4):725-739.
3. Mijumbi R, Serrat J, Gorricho J-L, Bouten N, De Turck F, Boutaba R. Network function virtualization: state-of-the-art and research challenges. *IEEE Commun Surv Tutorials*. 2016;18(1):236-262.
4. ETSI. Network functions virtualisation (nfv); management and orchestration. *NFV-MAN*. 2014;1:v0.
5. Tacker. https://wiki.openstack.org/wiki/Tacker. Accessed: 2017-11-21.

6. OpenBaton. https://openbaton.github.io/. Accessed: 2017-11-21.
7. ETSI. Open source mano. https://osm.etsi.org/ . Accessed: 2017-11-21.
8. ONAP. https://www.onap.org/. Accessed: 2017-11-21.
9. Rift.io. https://riftio.com/. Accessed: 2017-11-21.
10. Shen W, Yoshida M, Minato K, Imajuku W. vConductor: an enabler for achieving virtual network integration as a service. *IEEE Commun Mag*. 2015;53(2):116-124.
11. Venâncio G, Garcia VF, da Cruz Marcuzzo L, et al. Simplificando o gerenciamento do ciclo de vida de funções virtualizadas de rede. In: XXIII Workshop de Gerência e Operação de Redes e Serviços (WGRS). XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos, Vol. 23. Campos do Jordão, São Paulo, Brazil; 2018.
12. ETSI. Network functions virtualization (nfv) infrastructure overview. *NFV-INF*. 2015;1:V1.
13. Herrera JG, Botero JF. Resource allocation in nfv: a comprehensive survey. *IEEE Trans Netw Serv Manag*. 2016;13(3):518-532.
14. Han B, Gopalakrishnan V, Ji L, Lee S. Network function virtualization: challenges and opportunities for innovations. *IEEE Commun Mag*. 2015;53(2):90-97.
15. OpenVIM. https://www.sdxcentral.com/projects/openvim/. Accessed: 2017-11-24.
16. Dmitriy Andrushko GE. What is the best nfv orchestration platform? a review of osm, open-o, cord, and cloudify. https://www.mirantis.com/blog/which-nfv-orchestration-platform-best-review-osm-open-o-cord-cloudify/. Accessed: 2017-12-14.
17. Mechtri M, Ghribi C, Soualah O, Zeghlache D. Nfv orchestration framework addressing sfc challenges. *IEEE Commun Mag*. 2017;55(6):16-23.
18. OpenStack. https://www.openstack.org/. Accessed: 2017-11-24.
19. Halpern J, Pignataro C. Service function chaining (sfc) architecture. RFC 7665, RFC Editor; 2015.
20. Foundation L. https://www.opnfv.org/. Accessed: 2018-09-14.
21. Bondan L, Franco MF, Marcuzzo L, et al. FENDE: Marketplace-based distribution, execution, and life cycle management of VNFs. *IEEE Commun Mag*. 2019;57(1):13-19.
22. Franco MF, dos Santos RL, Schaeffer-Filho A, Granville LZ. Vision–interactive and selective visualization for management of NFV-enabled networks. In: 2016 IEEE 30th International Conference on Advanced Information Networking and Applications (AINA). Crans-Montana, Switzerland: IEEE; 2016: 274-281.
23. Chaisiri S, Lee B-S, Niyato D. Optimal virtual machine placement across multiple cloud providers. In: IEEE Asia-Pacific Services Computing Conference, 2009. APSCC 2009. Singapore: IEEE; 2009: 103-110.
24. Gao Y, Guan H, Qi Z, Hou Y, Liu L. A multi-objective ant colony system algorithm for virtual machine placement in cloud computing. *J Comput Syst Sci*. 2013;79(8):1230-1242.
25. Li S, Crandall J. TOSCA Simple Profile for Network Functions Virtualization (NFV) Version 1.0. OASIS Committee Specification Draft, Open Standards for the Information Society; 2017. http://docs.oasis-open.org/tosca/tosca-nfv/v1.0/tosca-nfv-v1.0.html.
26. da Cruz Marcuzzo L, Garcia VF, Cunha V, et al. Click-on-osv: a platform for running click-based middleboxes. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM). Lisbon, Portugal: IEEE; 2017: 885-886.

AUTHOR BIOGRAPHIES

Giovanni Venâncio is a PhD student in Computer Science at the Department of Informatics of the Federal University of Paraná (UFPR—Brazil) under the supervision of Prof Dr Elias Procópio Duarte Júnior. Giovanni holds an MSc (2017) in Computer Science and a Computer Science degree (2016) at the same institution. His research interests include Network Function Virtualization and Fault-Tolerant Distributed Systems.

Vinícius Fulber Garcia is a PhD student in Computer Science at the Department of Informatics of the Federal University of Paranà (UFPR—Brazil) under the supervision of Prof Dr Elias Procópio Duarte Júnior. He holds a Computer Science degree from Federal University of Santa Maria (UFSM—Brazil) and a Master degree in Computer Science from UFSM Post-Graduate Program in Computer Science. His research interests include network functions virtualization, service function chaining, compression algorithms, and information theory.

Leonardo da Cruz Marcuzzo holds a degree in Computer Science from Federal University of Santa Maria (UFSM) and currently is an MSc Student in Computer Science at the same institution. His research interests include network functions virtualization and operating systems.

Thales Nicolai Tavares is a graduate of the course on Computer Network Technology at the Federal University of Santa Maria (2016) in Brazil. He is currently a master's degree student in Computer Science at the same university.

Has knowledge in the area of Computing, with emphasis on Computer Networks. Research interests in network management, software networks and virtualization of network functions.

Muriel Figueredo Franco is pursuing his PhD under the supervision of Prof Dr Burkhard Stiller at the University of Zurich (UZH). He is also a Research Assistant at the Communication Systems Group (CSG). Muriel holds an MSc (2017) in Computer Science from the Federal University of the Rio Grande do Sul (UFRGS) under the supervision of Prof Dr Lisandro Granville and obtained a BSc (2014) in Computer Science from the Federal University of Pelotas (UFPEL). His research topics include Network Functions Virtualization, Information Visualization, and Blockchain.

Lucas Bondan is a PhD student in Computer Science at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS) in Brazil and an R&D Project Manager at the Brazilian National Research and Educational Network (RNP). From 2016 to 2018 he was a PhD student fellow at the Department of Information Technology of Ghent University in Belgium, working with security-related areas of Network Functions Virtualization (NFV). He has a Computer Engineering degree from Pontifícia Universidade Católica do Rio Grande do Sul and a Master Degree in Computer Science from UFRGS. His research interests include network functions virtualization, network management and orchestration, service function chaining, cognitive networks, and wireless communication systems.

Alberto Egon Schaeffer-Filho holds a PhD in Computer Science (Imperial College London, 2009) and is an Associate Professor at Federal University of Rio Grande do Sul (UFRGS), Brazil. From 2009 to 2012, he worked as a research associate at Lancaster University, UK. Dr Schaeffer-Filho is a CNPq-Brazil Research Fellow and his areas of expertise are network/service management, network virtualization and software-defined networks, policy-based management, and security and resilience of networks. He has authored over 60 papers in leading peer-reviewed journals and conferences related to these topics and also serves as TPC member for important conferences in these areas, including IFIP/IEEE IM (2019), NetSoft (2019), CNSM (2018), and IEEE/IFIP NOMS (2018). He is the general chair for SBRC 2019, co-chair for IEEE ICC 2018 CQRM Symposium, and demo co-chair for IFIP/IEEE IM 2017. He is also a member of the IEEE.

Carlos Raniery Paula dos Santos is Adjunct Professor of Computer Science at the Department of Applied Computing of the Federal University of Santa Maria (UFSM), Brazil. He holds PhD (2013) and MSc (2008) degrees in Computer Science, both received from the Federal University of Rio Grande do Sul (UFRGS), where he was also Postdoctoral Research Fellow from October 2013 to September 2014. From May 2010 to April 2011 he was a visiting researcher at the IBM T.J. Watson Research Center—Hawthorne, where he developed projects on IT Service Management and Security Management. His current research interests focus on design and management of Future Networks and Technologies, including aspects such as network virtualization, quality of service management, network programmability, and security management. He has worked on the organization of several international conferences, is on the TPC of many network and service management events, including IM, NOMS and CNSM, and is reviewer of important journals in the area, such as IJNM, TNSM, and COMMAG.

Lisandro Zambenedetti Granville is an associate professor at the Institute of Informatics of the Federal University of Rio Grande do Sul (UFRGS), in Brazil. He received his MSc and PhD degrees, both in computer science from UFRGS, in 1998 and 2001 respectively. He has served as a TPC member for many important events in the area of computer networks (eg, IM, NOMS, and CNSM) and was TPC co-chair of DSOM 2007, NOMS 2010, NetSoft 2018, and ICC 2018.

Elias P. Duarte Jr. is a Full Professor at Federal University of Parana, Curitiba, Brazil, where he is the leader of the Computer Networks and Distributed Systems Lab (LaRSis). His research interests include Computer Networks and Distributed Systems, their Dependability, Management, and Algorithms. He has published more than 200 peer-reviewer papers and has supervised more than 130 students both on the graduate and undergraduate levels. Prof Duarte is currently Associate Editor of the IEEE Transactions on Dependable and Secure Computing, and has served as chair of more than 20 conferences and workshops in his fields of interest. He received a PhD degree in

Computer Science from Tokyo Institute of Technology, Japan, 1997, MSc degree in Telecommunications from the Polytechnical University of Madrid, Spain, 1991, and both BSc and MSc degrees in Computer Science from Federal University of Minas Gerais, Brazil, 1987 and 1991, respectively. He chaired the Special Interest Group on Fault Tolerant Computing of the Brazilian Computing Society (2005-2007); the Graduate Program in Computer Science of UFPR (2006-2008); and the Brazilian National Laboratory on Computer Networks (2012-2016). He is a member of the Brazilian Computing Society and a Senior Member of the IEEE.

How to cite this article: Venâncio G, Garcia VF, da Cruz Marcuzzo L, et al. Beyond VNFM: Filling the gaps of the ETSI VNF manager to fully support VNF life cycle operations. *Int J Network Mgmt*. 2019;e2068. https://doi.org/10.1002/nem.2068