ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
SCUOLA DI INGEGNERIA E ARCHITETTURA
(SEDE DI BOLOGNA)

CORSO DI LAUREA IN INGEGNERIA INFORMATICA M

A NETWORK FUNCTION VIRTUALIZATION ARCHITECTURE FOR DISTRIBUTED IOT GATEWAYS

Tesi di laurea in
STSTEMI DISTRIBUITI

Relatore
Prof. Ing. Paolo Bellavista

Correlatori
Dr. Roch H. Glitho
Dott. Ing. Luca Foschini

Candidato

Luca Montanari

Anno Accademico 2015/2016

Sessione III

Abstract Italiano

La virtualizzazione permette a diverse applicazioni di condividere lo stesso dispositivo IoT. Tuttavia, in ambienti eterogenei, reti di dispositivi IoT virtualizzati fanno emergere nuove sfide, come la necessità di fornire on-the-fly e in maniera dinamica, elastica e scalabile, gateway. NFV è un paradigma progettato per affrontare queste nuove sfide. Esso sfrutta tecnologie di virtualizzazione standard per consolidare specifici elementi di rete su generico hardware commerciale. Questa tesi presenta un'architettura NFV per gateway IoT distribuiti, nella quale istanze software dei moduli dei gateway sono ospitate su un'infrastruttura NFV distribuita, la quale è operata e gestita da un IoT gateway Provider. Considereremo diversi IoT Provider, ciascuno con le proprie marche, o loro combinazioni, di sensori e attuatori/robot. Ipotizzeremo che gli ambienti dei provider siano geograficamente distribuiti, per un'efficiente copertura di regioni estese. I sensori e gli attuatori possono essere utilizzati da una varietà di applicazioni, ciascuna delle quali può avere diversi requisiti per interfacce e QoS (latenza, throughput, consumi, ecc...). L'infrastruttura NFV consente di effettuare un deployment elastico, dinamico e scalabile dei moduli gateway in questo ambiente eterogeneo e distribuito. Inoltre, l'architettura proposta è in grado di riutilizzare moduli il cui deployment è stato precedentemente compiuto. Ciò è ottenuto attraverso Service Function Chaining e un'orchestrazione dinamica a runtime. Infine, presenteremo un prototipo basato sulla piattaforma OpenStack.

Abstract

Virtualization enables multiple applications to share the same IoT device. However, in heterogeneous environments, networks of virtualized IoT devices raise new challenges, such as the need for on-the-fly, dynamic, elastic, and scalable provisioning of gateways. NFV is a paradigm emerging to help tackle these new challenges. It leverages standard virtualization technology to consolidate special-purpose network elements on commodity hardware. This article presents NFV architecture for distributed IoT gateways, in which software instances of gateway modules are hosted in a distributed NFV infrastructure operated and managed by an IoT gateway provider. We consider several IoT providers, each with its own brand or combination of brands of sensors and actuators/robots. We assume the providers' environments to be geographically distributed, to efficiently cover extensive physical areas. The sensors and actuators can be accessed by a variety of applications, each of which may have different interface and QoS (latency, throughput, etc.) requirements. The NFV infrastructure allows dynamic, elastic, and scalable deployment of gateway modules in this heterogeneous and distributed environment. Furthermore, the proposed architecture is capable of reusing already deployed modules, achieved through service function chaining and dynamic runtime orchestration. We also present a prototype that is built using the OpenStack platform.

Table of Contents

Chapter 1

Introduction

Research on Internet of Things (IoT) devices virtualization has become prominent in recent years. Virtualization technology abstracts device resources as logical units, and allows for their efficient and simultaneous usage by multiple simultaneous applications, even if they have conflicting requirements and goals. This capability permits to transform a network of IoT devices into a multi-purpose platform, in which several virtual IoT devices are created on demand, each tailored for a specific task or objective. Moreover, IoT devices are heterogeneous by nature, depending on their functionalities, hardware capabilities, and vendor. Consequently, challenges arise when trying to communicate with them in a simple and unique way.

Gateways are required for the interactions between applications and IoT devices. They are generally complex. Furthermore, it is difficult and expensive to upgrade them when new-brand sensors and actuators/robots are deployed. In addition, their capabilities do not scale when the number of applications and the corresponding workload in the IoT devices change dynamically.

Network functions virtualization (NFV) [1] is an emerging paradigm in overcoming the aforementioned challenges. NFV permits standard virtualization technology to consolidate dedicated network elements (e.g., firewalls, network address translation [NAT]) onto commodity hardware. By implementing network functions as software instances called virtual network functions (VNFs), NFV reduces the operational costs and provides hardware independence. Moreover, on-the-fly, dynamic, scalable, and elastic provisioning of network services is among its benefits.

In this thesis, we present an NFV architecture for distributed IoT gateways. The firmware/hardware used to provide IoT gateway functionalities are replaced by VNFs deployed in an NFV infrastructure. We enable granular provisioning of NFV to decompose the gateway into fine-grained modules, such as metadata extractor and information model converter, to be implemented as VNFs. More importantly, granular NFV is best suited, since the dynamic growth in the number of applications and the addition of new-brand sensors require rapid introduction of new VNFs and update of the existing ones. VNFs are instantiated on the fly and chained to realize an IoT gateway.

Some providers of IoT devices might not have a centralized environment. Their infrastructure could be scattered across multiple Points of Presence (PoPs), which are locations where network functions are implemented. Accordingly, it may not be possible to deploy the gateway modules on top of the same PoP. Hence, the need arises for a distributed architecture capable of deploying and managing them on a distributed environment. Moreover, the architecture allows for the reuse of already deployed functions, for cost efficiency purposes. This is achieved through dynamic runtime orchestration. Finally, given the elevated number of possible functions that can be employed to generate a gateway, the architecture provides a way to describe the functions.

The architecture introduces a new business actor — the IoT gateway provider — in addition to the traditional ones, meaning the end user applications and the IoT providers. This new actor plays a dual role. On one hand, it provides two algorithms: one to find and obtain the VNFs, and one to chain them to make on-the-fly gateways. On the other hand, it operates and manages the infrastructure in which the VNFs are executed.

The rest of the thesis is organized as follows. In Chapter 2 we discuss the background information and the motivations behind this research. Chapter 3 introduces an illustrative use case, the architectural requirements, and the critical review of the state of the art. In Chapter 4 we present our architecture and its functioning. Chapter 5 first evaluates different solutions for the implementation, then it describes the prototype that validates the architecture, along with its implementation details. Chapter 6 concludes the thesis, and outlines the future work to be done.

Chapter 2

Context and Motivations

This chapter presents the necessary context information and the motivations behind this research.

2.1 Context

This section presents the background information that is relevant to our research domain. It covers two topics: Network Functions Virtualization (NFV), with a focus on ETSI NFV Management and Orchestration (MANO) and Internet of Things (IoT).

2.1.1 Network Functions Virtualization

NFV is an emerging paradigm that offers a new way to design, deploy and manage network services, by leveraging virtualization technology [2]. The main goal of NFV is the decoupling of network functions from the underlying proprietary hardware appliances. This allows for the consolidation of many network equipment types on high volume servers, switches and storage, which could be located in data centers, network nodes and the end-user premises. In NFV, a service can be decomposed in a set of Virtual Network Functions (VNFs), which are stand-alone pieces of software that can run on one or more infrastructure resources, either physical or virtual.

NFV aims at bringing several benefits to Telecommunication Service Providers (TSPs). Some of them are listed below:

- Reduced equipment cost and energy consumption, achieved by equipment consolidation.

- Reduced development cost and time to market, achieved by decoupling the software from the hardware. This allows network operators to focus solely on the software development.

- Reduced CAPEX (Capital Expenses) and OPEX (Operational Expenses), since NFV allows for flexible network function deployment. This means that network operators can deploy new network services over the same physical platform.

- Dynamic scaling of the services by need. NFV allows to decouple the functionality of a network function into instantiable software components. This permits to scale the actual VNF performance more dynamically, with greater flexibility and finer granularity. One example can be VNF capacity provisioning in response to the actual traffic.

The European Telecommunications Standards Institute (ETSI) Industry Specification Group for Network Functions Virtualization (ISG NFV) has defined a reference architectural framework for NFV [3]. Figure 2.1 illustrates the high-level NFV framework, in which three main architectural components are identified: VNFs, NFV Infrastructure (NFVI), and NFV MANO.

NFVI is the combination of physical and virtualized resources, subdivided in compute, storage and networking, that make up the environment in which VNFs are deployed, managed and executed. Virtual resources (i.e., compute, storage, and network) are an abstraction of the physical ones, this abstraction is achieved using a virtualization layer based on a hypervisor. For instance, the virtual computing and storage resources can be represented in terms of Virtual Machines (VMs).

A VNF is the software implementation of a network function (e.g., firewall, NAT, DHCP) that is deployed on top of NFVI resources, for instance, a VM. A VNF can also be decomposed in its constituent parts, therefore it can be deployed over multiple VMs. In the same way, a VM can host multiple VNFs. The order, number, and type of VNFs constituting a network service is dependent on the functionality and the behavior of the service itself.

NFV MANO covers the orchestration and lifecycle management of VNFs, network services and physical and hardware resources. This component will be further explored in the next paragraph.

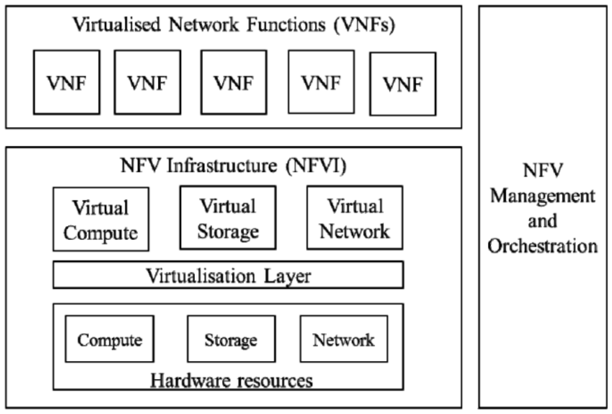


Figure 2.1 – High-level NFV framework

2.1.2 NFV Management and Orchestration

The NFV MANO [4] architectural framework has the role to manage the NFVI and orchestrate the allocation of resources needed by the Network Services (NSs) and the VNFs. This level of coordination is necessary, given the decoupling of the network functions from their infrastructure.

The management and orchestration of NFVI covers both physical and virtualized resources. For the physical ones, the management mainly focuses on connectivity aspects between physical and virtualized resources. For the virtualized resources, the management aims at handling NFVI resources in NFVI Points of Presence (NFVI-PoPs). A Network PoP is a location where a physical or virtual network function is implemented [5]. Some of the NFVI management operations include: service discovery; resource availability, allocation, release; resource fault and performance management.

The management and orchestration aspects of VNFs focus on VNF lifecycle operations. Some of these include: VNF instantiation (create a VNF and allocate proper NFVI resources to it), VNF scaling, VNF update and upgrade (support software or configuration changes), VNF termination (release the NFVI resources associated with the VNF).

The scaling operation allows to increase or reduce the capacity of a VNF, in response to its actual performances. The scale can be either "vertical" or "horizontal". In case of vertical scaling, the management increases or diminishes the VNF computational resources (e.g., CPU, memory). Horizontal scaling means that the management can instantiate (respectively terminate) multiple instances of the same VNF so that the workload can be split between them.

Another important aspect of the VNF Management is the monitoring of the Key Performance Indicators (KPIs) of a VNF, mainly for scaling purposes. For instance, if the incoming traffic through a VNF is too high, the VNF Management might scale up the VNF to increase its capabilities and avoid a possible bottleneck in the network.

The Network Service Orchestration focuses on the management of the lifecycle of Network Services. Some of the operations include: NS registration, instantiation, scaling (grow or reduce the capacity of the NS), update (service reconfiguration such as changing inter-VNF connectivity or the constituent VNF instances) and termination.

The NS Orchestration manages the lifecycle of VNFs that realize an NS and it performs its services by using the VNF Management services and by orchestrating the NFV Infrastructure in which the VNFs run.

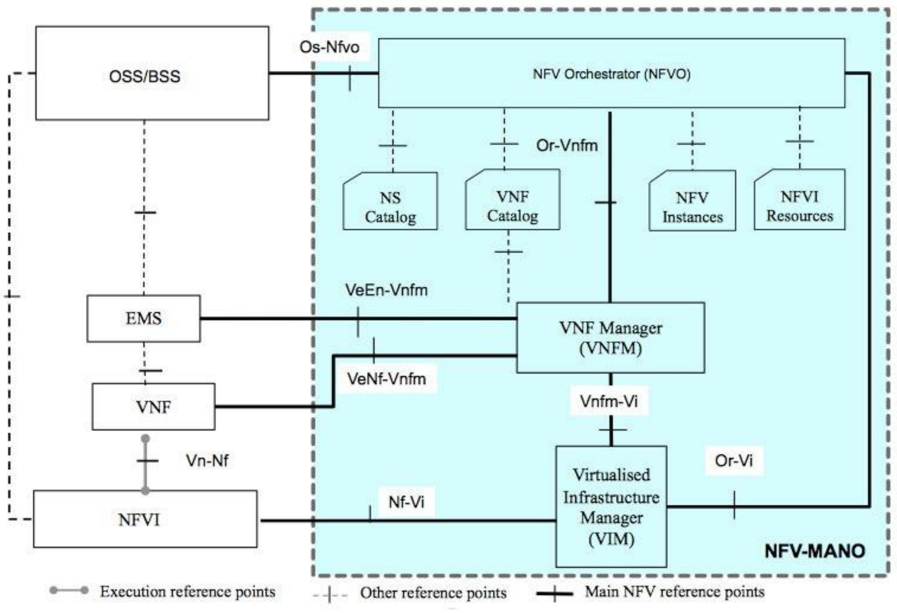


Figure 2.2 – NFV MANO architecture

Figure 2.2 shows a detailed view of the NFV MANO architecture, with all its components and reference points. For the purposes of this thesis it is only important to note that the NFV Orchestrator (NFVO) covers the functionalities of the Network Service Orchestration and it is responsible for the orchestration of NFVIs across multiple VIMs. The VNF Manager (VNFM) covers the management and orchestration aspects of VNFs. The Virtualized Infrastructure Management (VIM) is responsible for the management of the NFVI resources.

2.1.3 Internet of Things

The following section will introduce the IoT, a novel paradigm that extends Internet to real world things, such as physical devices, vehicles, and buildings. The first part will provide an overview on IoT, its enabling technologies and its possible applications. Then, a high level architecture of IoT is presented. The last part will introduce the IoT middleware, its architecture, and current solutions for it.

2.1.3.1 Overview and Enabling Technologies

IoT is considered as part of the internet of the future. Its basic idea is to allow connection and exchange of data between real world devices, called *things*, and applications. The IoT bridges real life and physical activities with the virtual world [6]. The IoT devices comprise sensors, Radio Frequency Identification (RFID) tags, Near Field Communication (NFC) tags, actuators, mobile phones, etc. These devices have certain unique features. They are uniquely identifiable and accessible to the Internet and are able to interact with each other and cooperate to achieve common goals [7].

IoT applications can be present in a variety of fields in our daily life. Possible examples can be smart home design, environment monitoring and natural disasters prediction, intelligent transport systems, smart cities design, medical and industry applications, etc.

The IoT concept can be realized by several enabling technologies. One example is identification, sensing and communication technologies such as RFID tags that are characterized by a unique identifier and sensor networks that are composed of several nodes communicating in a wireless multi-hop fashion. IEEE 802.15.4 is a widely adopted standard for wireless sensor networks. It defines the physical and MAC layers for low-power, low bit rate communications in wireless personal area networks (WPAN).

At the application layer, the Constrained Application Protocol (CoAP) has been introduced. It is a specialized internet protocol for constrained nodes and networks in the IoT. It allows for communication between these constrained devices and general nodes on the Internet. CoAP is designed to be easily translated to HTTP for faster integration with the web. Like HTTP, it is based on the Representational State Transfer (REST) model: Servers make resources available under a URL and Clients access them using HTTP methods such as GET, PUT, POST and DELETE. CoAP is designed to be extremely lightweight. To achieve this, at the transport layer it uses UDP on top of IP. Moreover, messages have a 4-byte header and a compact encoding of options to minimize fragmentation at the link layer.

A basic simplified workflow of IoT can be described as follows: some RFID tags can be sensed by smart sensor devices which will then send the retrieved information over the internet to a computational or processing unit. The data is then processed and the result is passed to a decision making and action invoking system. This system determines an automated action to be executed, for example, on an actuator or a robot.

2.1.3.2 IoT Architecture

The architecture of IoT is generally divided into five layers, as shown in Figure 2.3:

- Perception or Device Layer. This layer consists of the physical objects or sensors, which mainly have identification or sensing purposes. The collected data is passed to the Network layer for secure transmission to the information processing system.

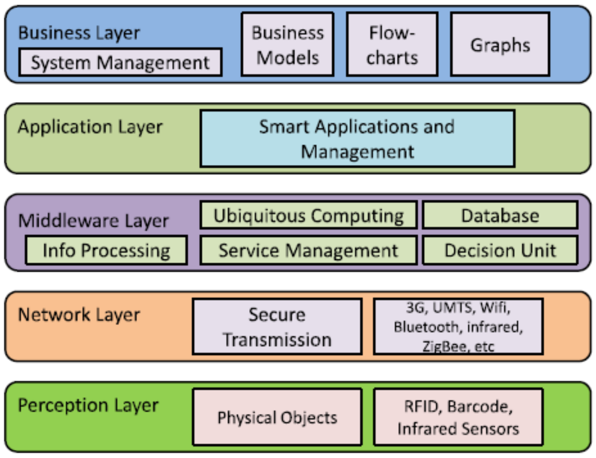


Figure 2.3 – IoT architecture

- Network or Transmission Layer. This layer transfers information from sensor devices to information processing systems, thus bridging the perception layer with the Middleware layer. The transmission medium can be either wired or wireless, depending on the devices. Depending on the sensor device, the transmission technology can be 3G, Wi-Fi, Bluetooth, ZigBee, infrared, etc.

- Middleware Layer. IoT devices can connect and communicate only with other devices that implement the same service type. This layer is responsible for service management. The other role of the Middleware layer is to receive information from the Network layer, store it into a database, process it and take automatic decisions based on the results.

- Application Layer. This layer is responsible for the global management of the application based on the processed data obtained from the Middleware layer. Possible IoT applications can be smart home, smart city, smart health, etc.

- Business Layer. This layer manages the overall IoT system, including applications and services, based on the data received from the Application layer.

Given the growth of IoT popularity in research, industry, and government, many standardization efforts are being carried and many international organizations are involved in the development of IoT. For instance, the Internet Engineering Task Force (IETF) introduced the IPV6 over Low-Power Wireless Personal Area Networks (6LoW-PAN) which defines a set of protocols that can be used to integrate sensors nodes into IPV6 networks.

2.1.3.3 IoT Middleware

Another example of IoT concept realization is the middleware, which is a software layer positioned between the technological and the application levels. It hides the heterogeneity of IoT devices and communication technologies. Therefore, a programmer is exempted from the exact knowledge of the underlying technologies while developing IoT applications. IoT middleware also simplifies the integration of legacy technologies with the new ones.

IoT middleware architectures proposed in the last years often follow the Service Oriented Architecture (SOA) approach. It allows for the decomposition of complex and monolithic systems into applications consisting of simpler and well-defined components. Architectures following the SOA approach have five layers (Figure 2.4):

- Applications are on top of the architecture, they exploit the functionalities of the other layers and provide it to the end-user.

- The service composition layer provides the functionalities for the composition of different services by networked objects to build specific applications.

- The service management layer provides functionalities such as object discovery, status monitoring, and service configuration. It enables the remote deployment of new services during run-time to meet the application requirements. The upper layer can then compose complex services by joining the ones provided at this layer.

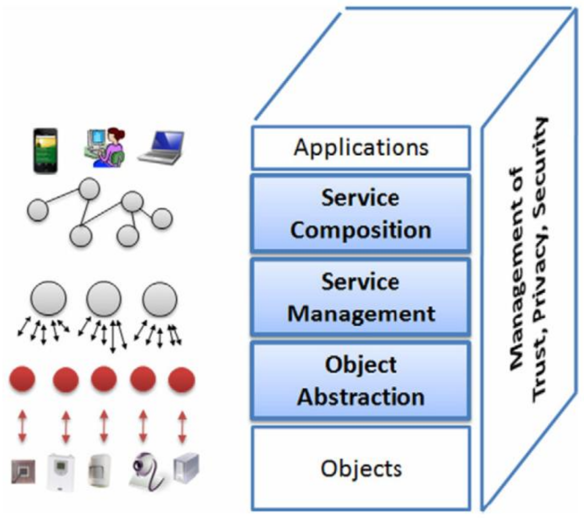


Figure 2.4 – SOA-based architecture for IoT middleware

- The object abstraction layer provides an abstraction of the heterogeneous IoT devices by harmonizing the access to them. This is done through offering common languages and procedures.

- The trust, privacy, and security management layer provides functionalities related to the security and the privacy of the exchanged data.

Popular solutions for IoT middleware include:

- Oracle Fusion Middleware [8]. It is an open source, comprehensive middleware that spans across multiple services, such as cloud applications, service integration, business intelligence, performance management, etc. The services implemented for the IoT middleware include real-time analysis (with a module specifically built for IoT gateways), device and service integration (using an SOA approach), security and monitoring.

- WSO2 Middleware platform [9]. It is an open source, SOA-based middleware that provides API management, integration and analytics offerings. It also has a focus on mobile and IoT.
- MachineShop [10] is an API-centric platform for enterprise IoT. It delivers a services-based architecture allowing the user to create and manage its own APIs. It leverages the REpresentational State Transfer (REST) API for interaction between components.
- Red Hat JBoss Middleware [11]. It is a complete and open source middleware that focuses on business process automation, system integration and accelerated development.

2.2 Motivations

Recently, a research effort has been conducted to solve the issue that emerges when heterogeneous applications want to communicate with heterogeneous IoT devices. This section briefly presents the work, its characteristics, and what has accomplished. However, that work suffers from various limitations. Therefore, in this thesis a new architecture is proposed to address them.

2.2.1 Previous Work

IoT devices (i.e., sensors and actuators) are very heterogeneous by nature. In fact, different devices may belong to different providers and have different hardware and/or capabilities. This raises a challenge when multiple and heterogeneous applications need to communicate with these multivendor IoT devices. In order for these heterogeneous IoT devices to interact with the applications, there is need for gateways to support these interactions.

A gateway architecture has been proposed in [12] to solve the aforementioned challenge. The proposed architecture is based on NFV. It decomposes the gateway into fine-grained modules (e.g. protocol converter, information model converter) implemented as VNFs and deployed in an NFV infrastructure.

The architecture satisfies the following requirements:

- The gateway supports *standard northbound* and *proprietary southbound interfaces.* One example of standard northbound interface could be the widely adopted Sensor Markup Language (SenML), carried over HTTP. It is designed to encode sensor measurements and device parameters.

- The architecture is extensible, elastic and scalable. Extensibility means that the architecture supports the introduction of new applications and domains. Elasticity allows for efficient utilization of underlying physical resources. Finally, scalability promotes the increasing amount of applications.

- The NFV architecture is flexible enough to support the integration of sensors from various brands.

The overall functioning can be summarized as follows: an end-user application requests services of sensor and/or actuator belonging to a Virtualized Wireless Sensors and Actuators Networks (VWSAN) Provider. The application provides the description of its northbound interface. Accordingly, the VWSAN Provider finds the proper sensor, retrieves its southbound interface description and sends a VNF request containing the description of the northbound and southbound interface description to a Gateway Provider. The Gateway Provider retrieves the VNFs necessary to compose the requested gateway, chains them, and migrates them as a whole package to a centralized location in the VWSAN Provider domain. Finally, the VWSAN Provider notifies the end-user application on the service availability. The application can now interact with the sensors through the provided gateway.

This approach has mainly two limitations. The first one is that the architecture assumes the VWSAN as a centralized domain. This might not be the case, as a VWSAN may be distributed across multiple locations. The second one is that completely new gateways are dispatched each time a new brand of sensors and/or actuators is deployed. This could lead to cost inefficiencies, because some of the services that compose the new gateway might already be present in the VWSAN domain.

Chapter 3

Use Case, Requirements and Related Work Evaluation

This chapter introduces an illustrative scenario and the set of requirements derived from it. After that, the state-of-the-art is reviewed in sight of these new requirements.

3.1 Illustrative Use Case

Over the last few years, the occurrence of large-scale wildfire episodes with extreme fire behavior has affected different regions of Europe: Portugal (2003 and 2005), south-eastern France (2003), Spain (2006 and 2009), and Greece (2000, 2007, and 2009). In such cases, continuous monitoring of a fire outbreak within fire-prone areas is critical. The monitoring can be done through IoT devices such as sensors scattered throughout a forest, and all linked back to a disaster management application (Figure 3.1). These sensors can be of various capabilities including temperature, humidity, rain gauge, CO2 detectors, and wind speed sensors. When a fire is broken out, the sensors inform the disaster management application (Figure 3.2). The application then dispatches rescue robots as another type of IoT device (Figure 3.3). In order to collect measurements from the sensors and send commands to the robots in a heterogeneous environment, a gateway is needed for the interactions between the application and these IoT devices. These gateways implement functions such as protocol converter, information model converter, data analytics, etc.

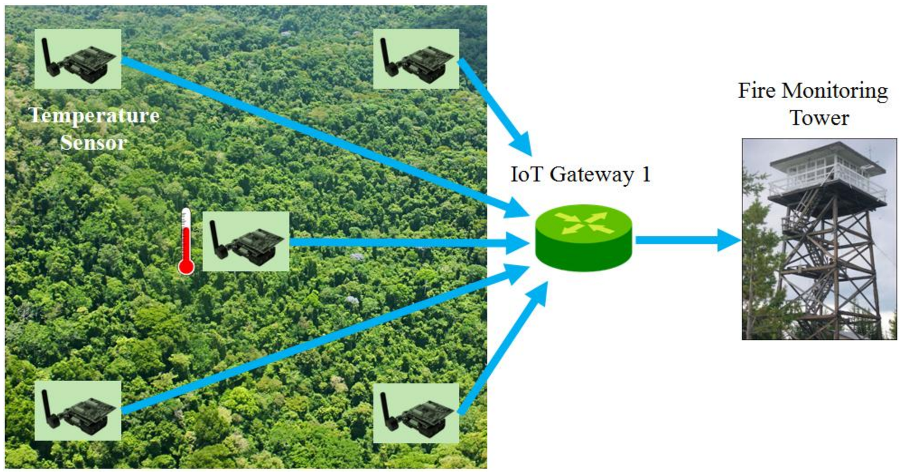


Figure 3.1 – Forest Monitoring

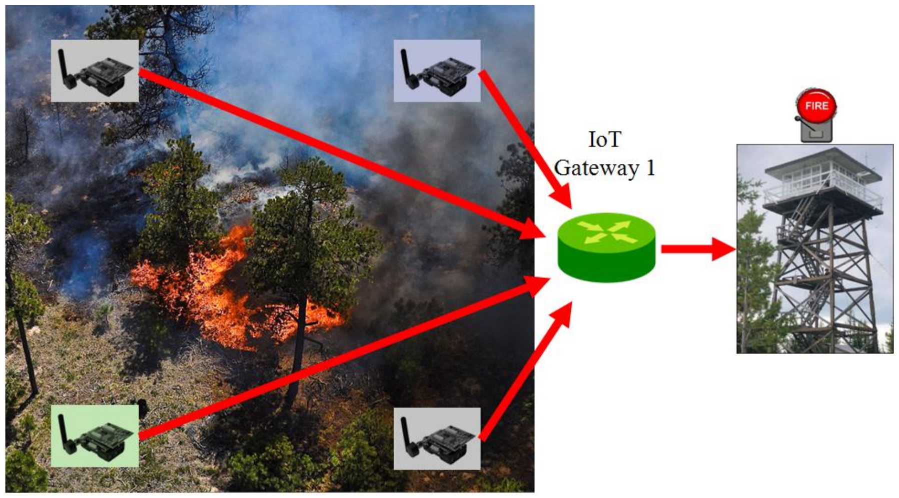


Figure 3.2 – Fire Outbreak

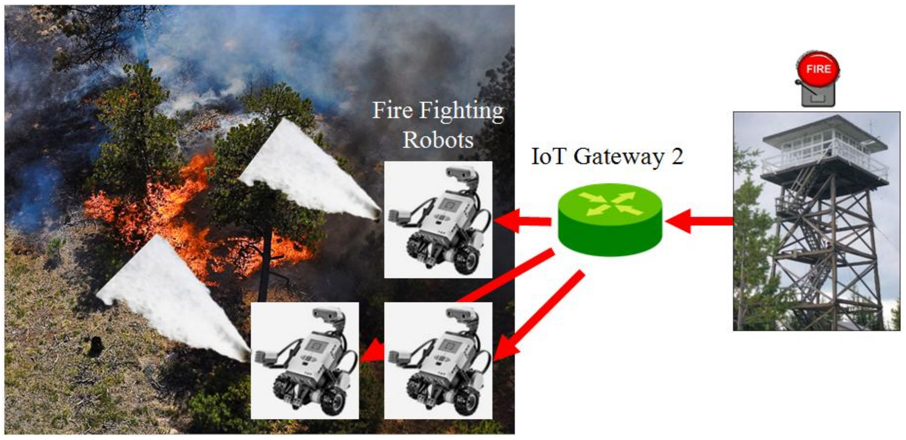


Figure 3.3 –Dispatch of Robots

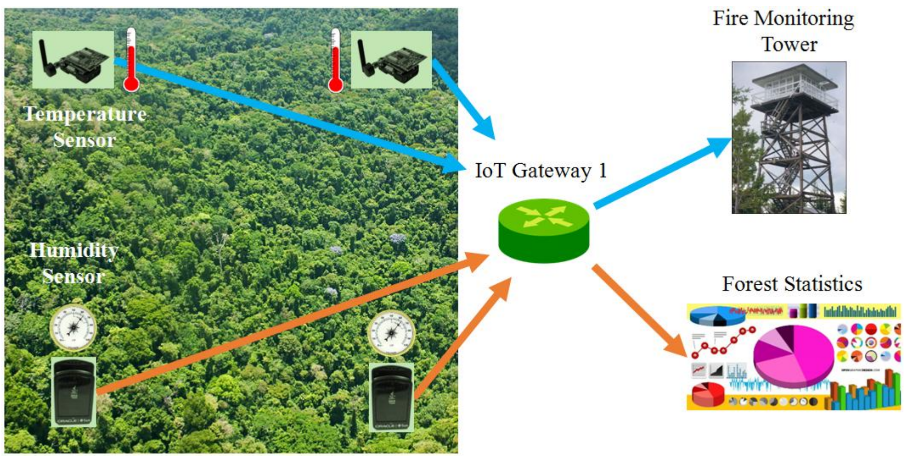


Figure 3.4 – Deployment of Different Applications and Sensors

In some cases, new brand of IoT devices or new applications may join and use the same protocol used by an already deployed IoT device or an application. Let us consider an environment where Preon32 sensors from Virtenio [13] and TelosB sensors from Advanticsys [14] have been deployed. Both can employ the CoAP protocol for communications. If an application (using HTTP) wants to collect data from them, a protocol conversion function must be used for the communication. Since both brands employ CoAP, the same protocol conversion function can be exploited. Accordingly, one function (e.g., protocol converter) may belong to different flows corresponding to different gateways (Figure 3.4).

3.2 Requirements

In order to address the limitations of the previous work [12], the architecture must be redesigned in order to satisfy the following requirements:

- Discovery of the required functions for a given application. In the previous work, a gateway was requested by simply providing a description of the northbound and southbound interfaces. This description is no longer sufficient because the range of available gateway modules is wide and diverse. For example, a metadata extractor, while being dependent from a particular IoT device, cannot be demanded by only providing the southbound interface description. Instead, a specific request must be made. Consequently, more information is needed in order to find the proper gateways. This information regards the functions to use, but could also cover characteristics of the deployment environment, latency, energy consumption, etc.

- Finer granularity of deployment. The micro-services that implement a gateway must be deployed separately and not as a package. Also, the architecture should allow for the reuse of an already deployed function. In the previous architecture, a brand new gateway was dispatched each time an application requested one. All the micro-services composing the gateway were deployed, as a package, even if some of them were already provisioned in the environment. In order to avoid cost inefficiencies, it is necessary to deploy each function separately and to implement a mechanism capable of checking if the functions are already available.

  Currently, gateway are often deployed on dedicated hardware, and in most solutions they are not split into their composing functions. On the other hand, NFV allows for the deployment of single functions, but there are currently no solutions that address IoT gateways.

- Finer granularity of management. The management of the gateway modules should be flexible enough to cope with the fact that the same function could belong to different execution flows. Accordingly, each flow may need to be managed in a specific manner corresponding to the application it belongs to. As an example, let us consider a gateway function used both by a fire monitoring application and a data analytics application. For the first application, low latency must be guaranteed at all times, while for the second one, the requirements might be less stringent. Consequently, the function must be managed in different ways, depending on the application it belongs to. If we consider the gateway modules as VNFs, many solutions have been proposed for their management. However, most of these do not provide a way to define custom sets of policies and operations.

- Dynamically orchestrating the flow when executing the gateway modules. The execution flow of the gateway must not be hard coded, instead it should be determined at runtime. In the previous work, the gateways were deployed as a package, therefore, the routing was implemented in a hard coded and static way. However, now there is need to reuse the already deployed functions, when possible. This means that the execution of the gateway cannot be hard coded. The introduction of the Service Function Chaining (SFC) paradigm can be of great aid to address the requirement. Some possible architectures will be presented in the next section, although none of them will provide a comprehensive solution that can cover the other requirements as well.

3.3 State-of-the-Art Evaluation

The illustrative scenario presented the possible use of IoT devices in large-scale wildfires and demonstrated the need for IoT gateways, for proper interaction between the IoT devices and the applications using them. In the state of the art for WSN/IoT gateway architectures, the main focus has been on bridging different sensor domains with public communication networks and the Internet.

Also, the existing literature describes a growing trend in NFV-based middlebox design. Since an IoT gateway falls under the taxonomy of middlebox, a brief overview of NFV architectures within the context of middleboxes is relevant.

The literature overview on current management and orchestration solutions will cover NFV MANO and SFC architectures. SFC is a novel paradigm introduced to address dynamic service composition and orchestration.

Therefore, the state-of-the-art is classified into three categories: Traditional architectures (WSN/IoT gateways), NFV architectures (middleboxes), and management and orchestration architectures (NFV MANO and SFC).

3.3.1 Traditional Architectures (WSN/IoT Gateways)

An architecture for an in-home IoT gateway is proposed in [15]. It consists of three subsystems: sensor node, gateway, and application platform. The gateway is deployed as a package on top of dedicated hardware, so it does not allow for the reuse of an already deployed component.

Jiang et al. [16] present an IoT gateway architecture for a CorbaNet-based digital broadcasting system, designed to lessen the effects of IoT technology on backbone networks. However, the gateway is considered as a monolithic block so it cannot be distributed.

A configurable, multifunctional, and cost-effective architecture for smart IoT gateways is proposed in [17]. It is possible to plug different modules into the architecture and the gateway can be dynamically configured. Dynamic flow orchestration is not discussed.

In [18], the authors propose an IoT gateway-centric architecture that provides various M2M services, such as association of metadata to sensor and actuator measurements using SenML. The gateway functionalities are hard-coded so finer granularity of deployment is not supported.

In [19], a gateway architecture for home and building automation system is proposed. The gateway is managed remotely by the network operator. However, reusability of already deployed functions is not discussed.

3.3.2 NFV Architectures (Middleboxes)

ClickOS [20] is a Xen-based software platform that allows hundreds of middleboxes to run on commodity hardware. It includes both simple middleboxes (e.g., packet forwarding from input to output interfaces) and full-fledged middleboxes (e.g., IPv4 router, firewall, etc.). However, virtualization of IoT gateway modules is not investigated.

T-NOVA [21] is an integrated architecture that enables network operators and service providers to manage their NFVs. It provides VNFs, like flow handling control mechanisms, as value-added services to its customers. T-NOVA allows third party developers to publish their VNFs as independent entities. Dynamic flow orchestration is not discussed.

In [22], NFV is used to virtualize an IP telephony function called Session Border Controller (SBC), which operates on both the control plane (i.e., load balancing and call control) and the media plane (i.e., media adaptation capabilities). The architecture is flexible and scalable.

The use of NFV for the virtualization of routing functions in OpenFlow-enabled networks is explored in [23]. All these works do not target the IoT domain.

3.3.3 Management and Orchestration Architectures

In [24], ETSI proposes a hierarchical MANO architecture composed by an umbrella NFVO and several administrative domains. The architecture is based on NS decomposition in which a NS is split in its sub-services. Each administrative domain manages the NSs that are part of its domain. The umbrella NFVO manages the whole NS. However, it is just a high level overview and no implementation is provided.

TeNOR [25] is a core component of the T-NOVA architecture. It allows for automated deployment and configuration of services. The architecture is based on microservices. VNFs can be deployed over multiple PoPs. The NSs are defined through descriptors, therefore dynamic orchestration is not supported.

Kataoka et al. present DiNO [26], an architecture for distributed NFV deployment. Hypervisors, VNFs, and network equipment can be deployed in an incremental and distributed manner. The architecture supports dynamic allocation of VNFs based on resource state and Service Level Agreement (SLA) monitoring. It also provides a built-in VNF load balancer. Function discovery and dynamic orchestration are not addressed.

In [27] the authors propose a Virtual Network Platform-as-a-Service (VNPaaS) for network services. The architecture focuses on distributed life cycle management of NSs and VNFs across geographically distributed locations. Moreover, the system allows for service decomposition and distributed management and orchestration. However, the architecture does not support dynamic orchestration.

Vilalta et al. [28] present an architecture for Software Defined Networks (SDN) / NFV orchestration for 5G services. They propose a hierarchical SDN orchestrator and virtualized function orchestration at the edge of the network. Although the work takes IoT into account, finer granularity of management is not discussed.

VLSP is a Service-Aware Virtualized Software-Defined Infrastructure proposed in [29]. The architecture is distributed, hierarchical and scalable. It is SDN-based. However, automated function discovery is not addressed and the work does not take into account the IoT domain.

[30], [31] and [32] propose policy-based or policy-driven architectures for dynamic service chaining and orchestration. These works focus on the composition of services based on user-defined SLA policies. They don't address service definition.

Martini and Paganelli present a Service-Oriented Approach for dynamic chaining of VNFs [33]. Key feature of this architecture is that services are defined at an abstract level and their concrete implementation is derived according to QoS-based utility functions. However, finer granularity of management is not discussed and the IoT domain is not considered.

Table 3.1 gives an overview of the solutions analyzed. For each project, the "Key Features" summarizes its main characteristics, while the limitations and drawbacks are addressed in the "Missing Requirements".

Table 3.1 – Summary of the related work

Project	**Key Features**	**Missing Requirements**
Architecture for in-home IoT gateway [15]	Three subsystems: sensor node, gateway, and application platform.	No reuse of already deployed components.
IoT gateway architecture for CorbaNet-based digital broadcasting system [16]	The architecture lessens the effects of IoT technology on backbone networks.	Gateway not distributed.
Configurable, multifunctional and cost-effective architecture for smart IoT gateways [17]	Pluggability of modules and dynamic configuration of gateways.	Dynamic flow orchestration not discussed.
IoT gateway-centric architecture to provide M2M services [18]	Association of metadata to sensor and actuator measurements using SenML. Scalable architecture.	Finer granularity of deployment not supported.
Gateway architecture for home and building automation system [19]	The gateway is managed remotely by the network operator.	No reuse of already deployed components.
ClickOS [20]	Xen-based software platform that allows hundreds of middleboxes to run on commodity hardware	Virtualization of IoT gateway modules not investigated.
T-NOVA [21]	VNFs provided as value-added services to its customers. Third party developers can publish their VNFs as independent entities.	Dynamic flow orchestration not discussed.
Session Border Controller virtualization [22]	It operates on both the control and the media plane.	IoT domain not targeted.
Implementation of NFV over an OpenFlow Infrastructure [23]	NFV used for the virtualization of routing functions in OpenFlow-enabled networks.	IoT domain not targeted.
Umbrella NFVO [24]	NS decomposition. Management roles split between NFVO and administrative domains.	No implementation provided.
TeNOR [25]	Automated deployment and configuration of services. VNF deployment over multiple PoPs.	Dynamic flow orchestration not supported.

DiNO [26]	Distributed NFV deployment. Dynamic allocation of VNFs based on resource state SLA monitoring.	Function discovery and dynamic flow orchestration not supported.
VNPaaS for network services [27]	Distributed management and orchestration of NSs and VNFs across geographically distributed locations. Service decomposition.	Dynamic flow orchestration not supported.
Architecture for SDN / NFV orchestration for 5G services [28]	Hierarchical SDN orchestrator. Virtualized function orchestration at the edge of the network.	Finer granularity of management not discussed.
VLSP [29]	SDN-based architecture. Distributed, hierarchical and scalable.	Function discovery and IoT domain not targeted.
Policy-based or policy-driven architectures for dynamic SFC [30] [31] [32]	Composition of services based on user-defined SLA policies.	Service definition not addressed.
Service-Oriented Approach for dynamic chaining of VNFs [33].	Services defined at an abstract level. Concrete implementation derived according to QoS-based utility functions.	Finer granularity of management not discussed. IoT domain not targeted.

We conclude that, with the exception of limited support for gateway modules, the existing IoT gateway architectures fall short of satisfying the requirements. With regard to NFV-based solutions, the current NFV architectures for middleboxes allow for finer granularity of deployment. However, they focus primarily on network elements, e.g., firewall, proxies, and NATs. NFV MANO and SFC architectures allow for finer granularity of management and/or dynamic service orchestration, but almost no solutions are provided for IoT gateways.

Chapter 4

Proposed Architecture for IoT Gateways

In this chapter, we present our NFV-based IoT Gateway architecture. The architectural principles are discussed first, followed by the architectural modules, interfaces, control plane, and an end to end scenario.

4.1 Architectural Principles

Our first architectural principle is the use of NFV concept when designing the IoT gateway. The gateway modules are then implemented as VNFs. NFV brings agility, flexibility, and dynamicity by decoupling network functions from the underlying hardware. The second principle is that the interaction interfaces between different domains are REpresentational State Transfer (REST)-based. REST is selected because it is lightweight, standard-based, and can support multiple data representations (e.g., plain text, JSON, and XML).

4.2 Overall Architecture

Figure 4.1 shows the proposed architecture. It extends the VWSAN Gateway architecture proposed in [12]. Some new modules have been introduced and some modules have been extended. This is done in order to support the description and the discovery of the VNFs required by the application and to support the management of the gateway modules.

The architecture comprises several Application Domains, IoT Provider Domains, and an IoT Gateway Provider Domain. The modules and interfaces are presented, followed by the control plane.

4.2.1 Architectural Modules

Each Application Domain contains an Application that requires the services of one or more IoT Providers. The Application contains two modules: Infrastructure Agent and Sensor/Actuator Agent. The Infrastructure Agent is responsible for the signaling procedure. It communicates with the IoT Provider Domain to negotiate the use of an IoT infrastructure. The Sensor/Actuator Agent is responsible for gathering measurements from the sensor and sending commands to the robots.

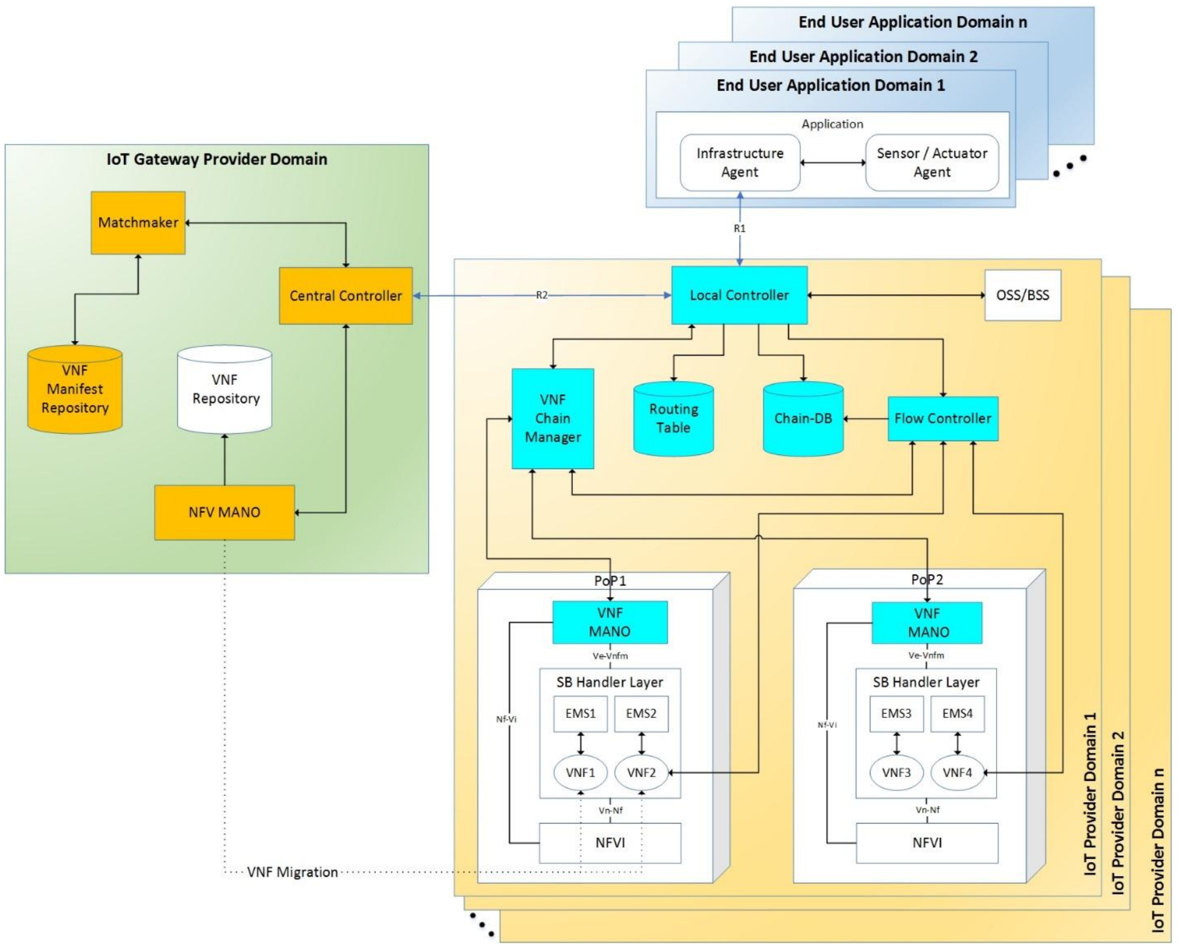


Figure 4.5 – Overall Architecture

The IoT Gateway Provider Domain consists of the following entities:

- **Matchmaker**: It is a novel module that receives the VNF discovery requests and performs a matchmaking procedure between the requested VNFs and the ones published in the VNF Manifest Repository. It then returns the list of the discovered VNF Manifests.

- **VNF Manifest Repository**: An XML based repository containing VNF Manifests of the published VNFs. In the published VNF Manifest the information includes description, function (e.g., a keyword used for publication/discovery purposes), image location (endpoint and/or download link), list of operations with related list and type of inputs, outputs, constraints and properties, list of VM requirements, list of management operations and related signatures.

- **NFV MANO**: This module has been extended such that the VNFs are not provisioned as a package anymore (i.e., the two VNFs at the same time). It enables for finer granularity of deployment/management; dispatching functions instead of packages. The module receives the list of VNFs to instantiate, it accesses the NFVI of the target IoT Provider and checks if the VNFs are already deployed in it. If not, the module downloads the VNFs from the VNF Repository, using the image location tag contained in the VNF Manifests, instantiates them, and then migrates them to the target domain. The module returns to the Central Controller the list of the VNF Instance IDs. The VNF Instance ID uniquely identifies the VM that is hosting a VNF. We assume that each VNF is deployed over one VM.

- **VNF Repository**: The VNF Repository contains the images of the published VNFs. It is accessed by the NFV MANO.

- **Central Controller**: The Central Controller has been extended in order to build the VNF chain (e.g., define the order in which traffic traverses the VNFs). It then sends the VNF Manifests and the VNF Instance IDs in proper order to the Local Controller in the IoT Provider Domain.

Each IoT Provider Domain comprises the following modules:

- **Southbound (SB) Handler Layer**: Contains VNFs that have been migrated from the IoT Gateway Provider Domain and their corresponding Element Management Systems (EMS). Each EMS is responsible for monitoring the resource utilization of its corresponding VNF [3].

- **NFVI**: Provides hardware and software resources, including computation, storage, and networking needed to deploy, manage, and execute VNFs.

- **Operational Support System/Business Support System (OSS/BSS):** Provides the description of IoT devices (e.g., sensor/robot brands).

- **Local controller**: Interacts with the Infrastructure Agent and the Central Controller. It has been extended by adding three new functionalities:

  1) Creating VNF discovery requests in the form of VNF Manifest files. A VNF Manifest contains basic VNF information and its structure is defined by an XSD file. In the requested VNF Manifest the information includes function, list and type of inputs and outputs (if any) and eventually a list of the capabilities of the hosting node. The data is derived by the information of the northbound interface used by the application (i.e., communication protocol, information model, etc.) and the information of the southbound interfaces used by the IoT devices (i.e., type of sensors/robots).

  2) Adding entries to the Chain-DB and to the Routing Table, after receiving the VNF chain from the Central Controller.

  3) At runtime, orchestrating the overall communication between the modules.

- **VNF Chain Manager**: Its role is to execute management operations that require a view of the whole system (monitoring total service time, VNF migration, etc.). Also, it has been extended such that it instructs the VNF MANOs on the specific VNF operations they have to execute.

- **VNF MANO**: A new module. The VNF MANOs are in charge of monitoring and executing specific operations on the VNFs. There is one VNF MANO per PoP, so the management is geographically close to its respective VNFs. VNF MANOs also need to notify the VNF Chain Manager when the state of a VNF changes (e.g., a VNF scales, or a VNF is no longer working) so that the Chain Manager can in turn notify the Local Controller to make appropriate changes to the tables.

- **Flow Controller**: it is a new module introduced in order to execute and manage the runtime traffic flows. It accesses the Chain-DB to retrieve the chains and communicates with the VNF Chain Manager to obtain the VNF addresses.

- **Routing Table**: This table is introduced in order to bind a VNF to the VM that is hosting it. The VNF is represented by its Manifest and the VM by the VNF Instance ID. The key of this table is the pair "VNF Manifest-VNF Instance ID", while the value is a VNF Unique IDentifier (UID). The VNF UID uniquely identifies a VNF inside the IoT Provider Domain. The VNF Manifest is not sufficient by itself because there might be multiple VNFs with the same Manifest (e.g., a scaled VNF) and the VNF Instance ID is not sufficient either, because a VM might host multiple VNFs. The pair is in fact the only way to grant uniqueness and allows to represent a specific VNF with a unique identifier. This UID is also used to define the VNF in the Chain-DB. Entries in this table are added and updated by the Local Controller. An update may occur, for instance, in case of scaling or migration. Table 4.1 shows an example.

Table 4.2 – Routing Table

VNF Manifest – VNF Instance ID	VNF UID
ProtocolConverter.xml – 0123-0000	01
InfoModel.xml – 0123-1111	02
DataAnalysis&Aggregation.xml – 0123-2222	03

- **Chain-DB:** It is a new database. It contains all the active chains in the IoT Provider domain. The database is structured as a key-value pair table. The key is a Chain-ID, while the value is the list of VNF UIDs. Table 4.2 shows an example of the database.

Table 4.3 – Chain-DB

Chain-ID	Chain
0000	03 – 01 – 02
0001	02 – 01
0002	03 – 05 – 07 – 03

4.2.2 Interfaces

The NFV components (i.e., NFVI, VNF MANO, SB Handler Layer) interact with each other through the interfaces defined by ETSI [3]. They include Vn-Nf, Nf-Vi, and Ve-Vnfm. Vn-Nf represents the execution environment provided by NFVI to SB Handler Layer. Nf-Vi is used for assigning virtualized resources in response to resource allocation requests (e.g., allocating VMs on hypervisors). It is also used by NFVI to communicate status information about virtualized and hardware resources to the VNF MANOs. Nf-Vi is also used to configure hardware resources. Ve-Vnfm carries out all operations during a VNF life cycle, including instantiation, scaling, updating, and termination. It is also used for exchanging VNF configuration information.

4.2.3 Control Plane

The control plane consists of signaling procedure and control interfaces, R1 and R2.

4.2.3.1 Signaling Procedure (Figure 4.2)

The signaling is initiated when the application requires services from the IoT Provider Domain. The Sensor/Actuator Agent instructs the Infrastructure Agent to start service negotiation. The Infrastructure Agent creates a service request that includes a description of the northbound interface (action 2) used by the application (e.g., communication protocol, information model, etc.) and sends it to the Local Controller of the IoT Provider Domain.

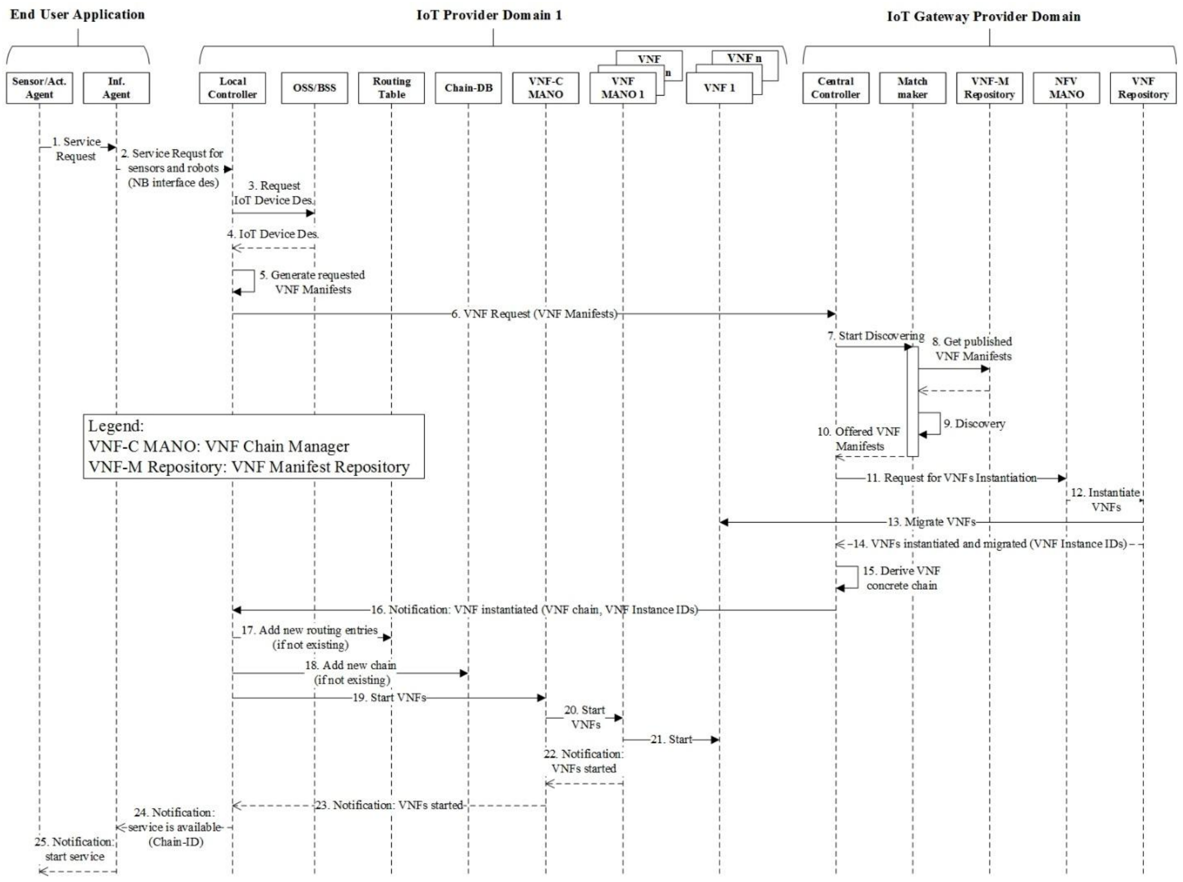

End User Application
IoT Provider Domain 1
IoT Gateway Provider Domain
Sensor/Act. Agent
Inf. Agent
Local Controller
OSS/BSS
Routing Table
Chain-DB
VNF-C MANO
VNF MANO 1
VNF n
VNF 1
VNF n
Central Controller
Match maker
VNF-M Repository
NFV MANO
VNF Repository
1. Service Request
2. Service Requst for sensors and robots (NB interface des)
3. Request IoT Device Des.
4. IoT Device Des.
5. Generate requested VNF Manifests
6. VNF Request (VNF Manifests)
7. Start Discovering
8. Get published VNF Manifests
9. Discovery
10. Offered VNF Manifests
11. Request for VNFs Instantiation
12. Instantiate VNFs
13. Migrate VNFs
14. VNFs instantiated and migrated (VNF Instance IDs)
15. Derive VNF concrete chain
16. Notification: VNF instantiated (VNF chain, VNF Instance IDs)
17. Add new routing entries (if not existing)
18. Add new chain (if not existing)
19. Start VNFs
20. Start VNFs
21. Start
22. Notification: VNFs started
23. Notification: VNFs started
24. Notification: service is available (Chain-ID)
25. Notification: start service
Legend:
VNF-C MANO: VNF Chain Manager
VNF-M Repository: VNF Manifest Repository

Figure 4.6 – Signaling Procedure Sequence Diagram

The signaling procedure includes several phases:

1) **Describing the functions for a given application (action 5)**

The process is initiated by the Local Controller when it receives the service request from the application. Upon receipt of this request, the Local Controller communicates with the OSS/BSS (action 3-4) to obtain information on parameters specific to the IoT devices it has (e.g., type of sensors/robots). It then creates a description of the functions needed in the form of VNF Manifest files. The Local Controller then sends to the Central Controller in the IoT Gateway Provider Domain a VNF Request containing the VNF Manifests (action 6).

2) Discovering the required functions for a given application

Once the matchmaker receives the VNF discovery requests (action 7), it first gets the list of published VNFs manifests from the VNF Manifest Repository (action 8), then it starts a matching procedure between the requested VNFs manifests and the published/offered ones (action 9). Figure 4.3 shows a flowchart of the process, which is done in three steps:

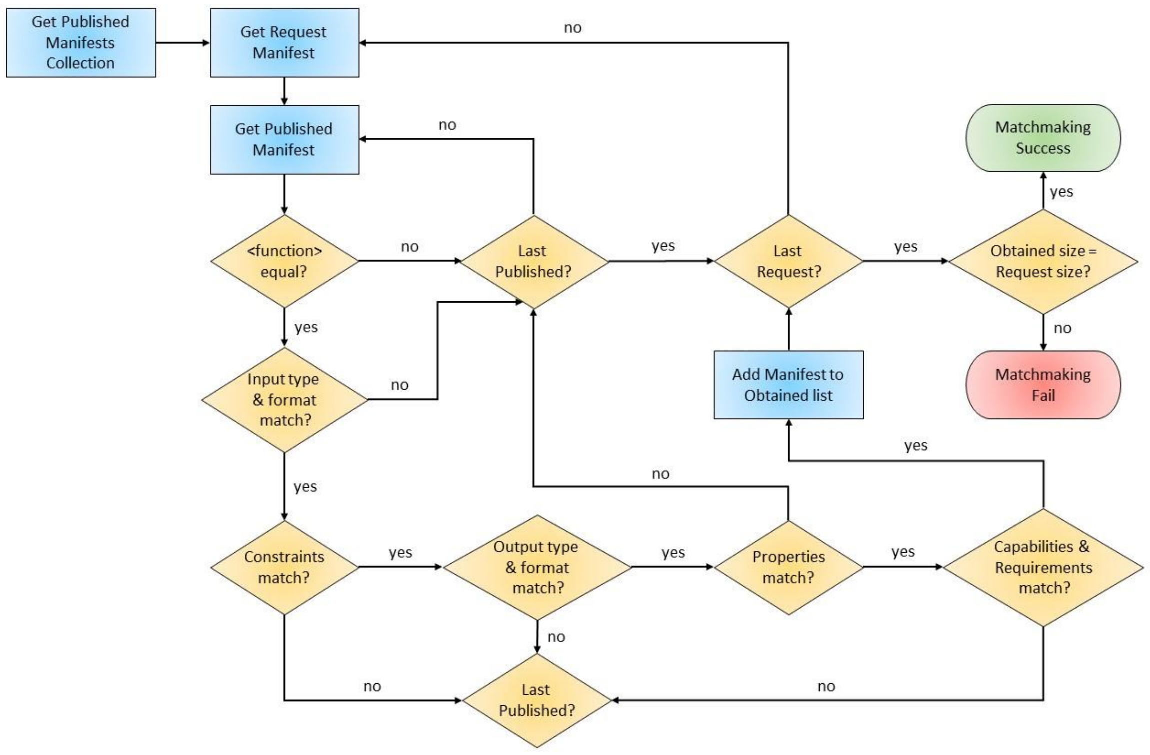
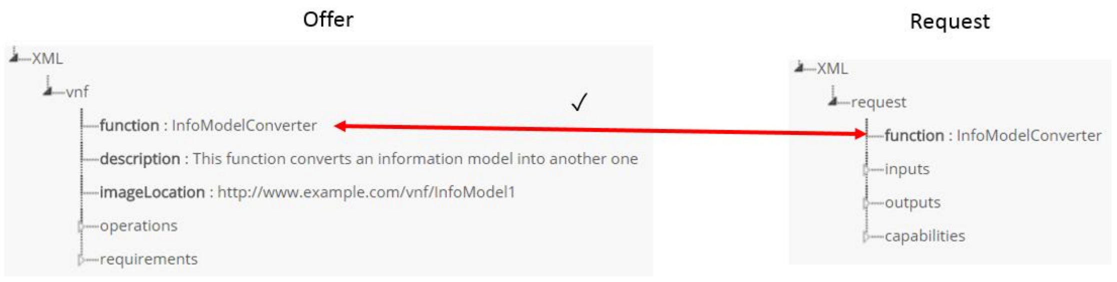


Figure 4.7 – Flowchart of Matchmaking Procedure

a) The first match is done using the <function> element. This element is present in both Request and Offer manifest files and contains a keyword that identifies the function (Figure 4.4).

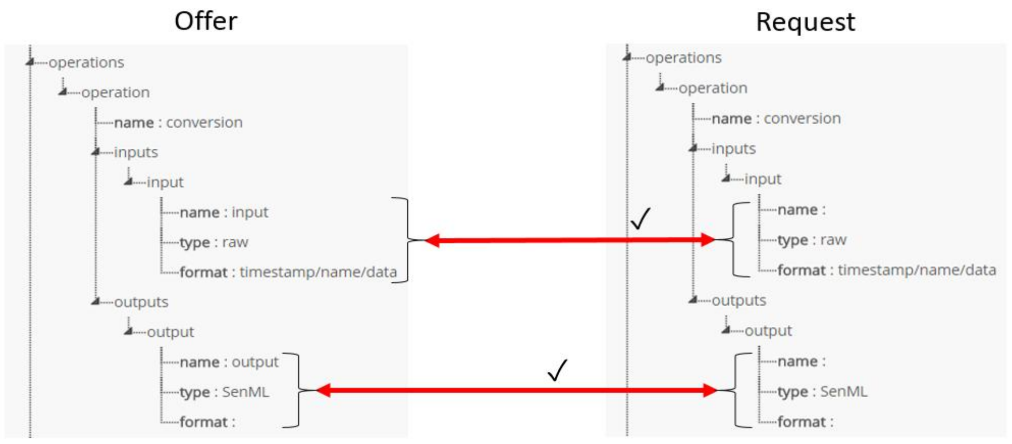
b) If the match succeeds, then the matchmaker will compare inputs and outputs of the various operations. At this stage, a specific rule is applied: if the Request specifies any inputs or outputs, the Offer must match number, format, and type (Figure 4.5).



```
XML
  vnf
    function : InfoModelConverter
    description : This function converts an information model into another one
    imageLocation : http://www.example.com/vnf/InfoModel1
    operations
    requirements
```

```
XML
  request
    function : InfoModelConverter
    inputs
    outputs
    capabilities
```

Figure 4.8 – Function Name Match



```
operations
  operation
    name : conversion
    inputs
      input
        name : input
        type : raw
        format : timestamp/name/data
    outputs
      output
        name : output
        type : SenML
        format :
```

```
operations
  operation
    name : conversion
    inputs
      input
        name :
        type : raw
        format : timestamp/name/data
    outputs
      output
        name :
        type : SenML
        format :
```

Figure 4.9 – Input & Output Match

The <constraints> tag is an optional tag that follows the inputs. It defines restrictions that might apply to certain input values. There is no general rule of comparison since these constraints might be of various nature, so they are evaluated case by case. The same criteria apply for the <properties> tag, which is instead related to the outputs. Figure 4.6 shows an example where an offered function takes two values as input, either integer or double. A constraint states that these two values must be of the same type. The Request Manifest requires an operation that takes a double and an integer as inputs. Although the Request inputs are compatible with the ones in the Offer, they don't satisfy the constraint, since they are of different type. Therefore, the match fails.

c) The last step will require comparing the Offer requirements (if any) with the Request capabilities. The requirements are the environmental properties necessary to execute a VNF (e.g., CPU, memory, software installed), while the capabilities are the current properties of the IoT Provider domain. The match can be exact (=), minimum (≥), or maximum (≤) (Figure 4.7).

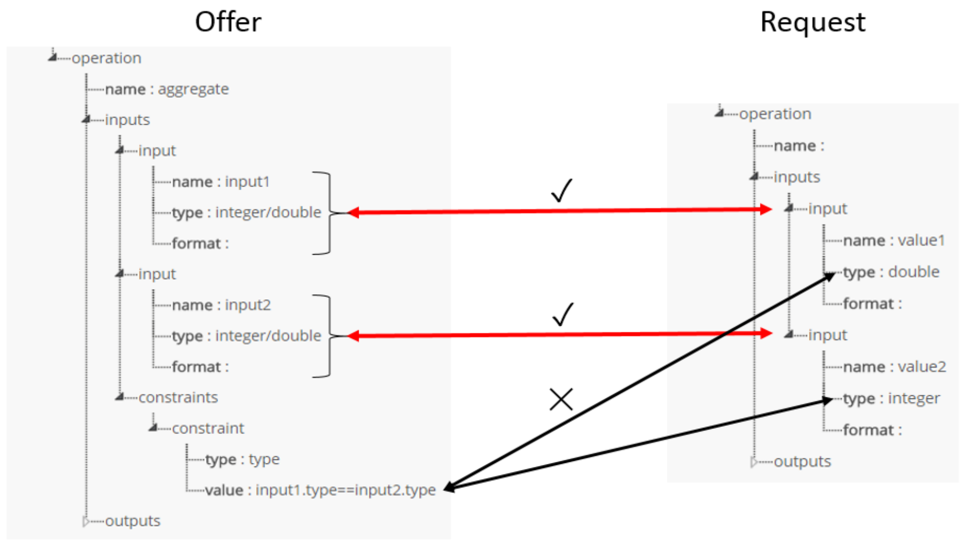


Figure 4.10 – Constraints Evaluation

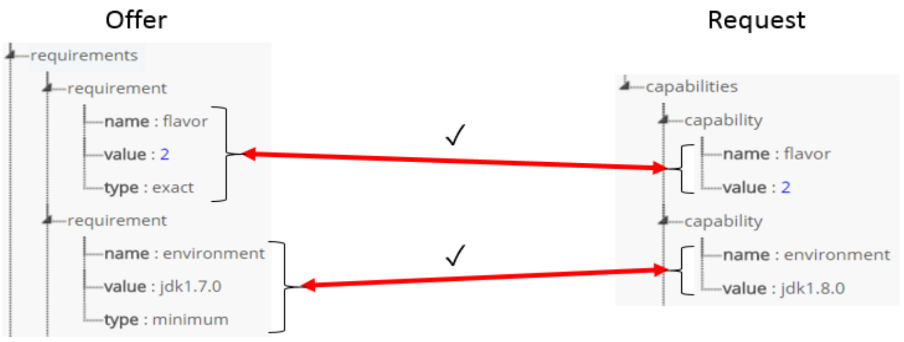


Figure 4.11 – Requirements & Capabilities Match

If the number of offered Manifests obtained is equal to the number of the requested ones, the overall procedure succeeds and the offered VNF Manifest are returned to the Central Controller. If the procedure fails, a notification of VNF unavailability is sent by the Central Controller to the Local Controller

In case of multiple matching for the same request, different choices can be adopted, depending on the Gateway Provider implementation. For example, the matchmaker can return the VNF with the least computational needs or latency. Or it can return the VNF that has the most number of operations. In our case, for simplicity reasons, we will adopt the "first matching" rule: only the first VNF that fulfills the matching gets returned.

Once the offered VNF Manifest are obtained (action 10), the Central Controller requests the NFV MANO in the IoT Gateway Provider Domain to instantiate and migrate the VNFs (action 11). The NFV MANO first checks if the VNF are already instantiated in the target domain, then it proceeds to instantiate and migrate the ones missing (action 13), after getting them from the VNF Repository (action 12).

The NFV MANO returns to the Central Controller the list of the VNF Instance IDs (action 14). These are unique identifiers for the VMs containing the VNFs.

3) Chain creation in the Central Controller

The Central Controller maintains a file that contains pre-built, abstract chains. These abstract chains can be provided by the VNF publishers or by the IoT Gateway Provider itself, since it knows the services it is offering (i.e., the gateways). An abstract chain defines a service at an abstract level, this way the elements comprising it are not bind to any specific implementation. One possible file format can be as follows, with one chain per line:

ProtocolConversion-InfoModelCoversion
DataAnalysis&Aggregation-MetadataExtraction-ProtocolConversion-InfoModelCoversion
MetadataExtraction-InfoModelConverison-ProtocolConversion

Each element in an abstract chain corresponds to a <function> element of a VNF manifest.

Once all the VNF Manifests are received from the matchmaker, the Central Controller derives the concrete chain (action 15).

A concrete chain is the actual implementation of an abstract chain, in which all the elements are bind to the instantiated VNFs. This means that the VNFs are properly chained in order to provide the service requested. In our case, the concrete chain will be represented by an ordered list of the VNF Manifests received.

A possible deriving procedure can be as follows: The Central Controller retrieves the <function> element of the first VNF Manifest and checks, in the file that contains the abstract chains, if there is a chain that starts with that element. If found, the length of the abstract chain is compared with the number of VNF Manifests we need to chain, otherwise the algorithm restarts with the next VNF Manifest.

If the length is the same, the abstract chain is a potential candidate, otherwise it is skipped and the algorithm continues with the next one. The Central Controller then tries to bind the second element in the chain with one of the VNF Manifests remaining. When the algorithm finishes, if all the Manifests are bind to the elements of one abstract chain, the concrete chain is derived and the corresponding ordered list of VNF Manifests is sent to the Local Controller, along with their VNF Instance IDs (action 16).

A notification gets sent to the Local Controller if no chain is found.

4) Chain Setup in the IoT Provider Domain

After receiving the chain of VNF Manifests and the corresponding ordered list of VNF Instance IDs, the Local Controller first verifies if any of the pairs "VNF Manifest - VNF Instance ID" is already present in the Routing Table (action 17). If the check is positive, it will retrieve the pair’s corresponding VNF UID, otherwise it will generate a new one, and it will put the entry "pair-VNF UID" into the table (action 18).

The Local Controller also accesses the Chain-DB (action 19) and checks if the chain is already stored in it. If the chain is not present, the Local Controller generates a Chain-ID and puts the pair "Chain-ID - chain" into the Chain-DB (action 20).

Lastly, the Local Controller requests the VNF Chain Manager to start the VNFs, by giving it the corresponding VNF Instance IDs (action 21). The VNF Chain Manager will forward the request to the proper VNF MANOs (action 22), which will start the VNFs (action 23).

The Chain-ID is returned to the application, in a notification of service availability (action 25).

5) Runtime Execution (Figure 4.8)

The runtime execution starts when the application sends a packet to the Local Controller. The packet contains the Chain-ID and the address of the receiving endpoint (action 1). The Local Controller forwards the packet to the Flow Controller (action 2). The Flow Controller uses the Chain-ID to lookup the Chain-DB and retrieve the corresponding chain (actions 3-4). The Flow Controller then requests to the VNF Chain Manager the current VNF addresses (action 5), by giving it the list of VNF UIDs. In turn, the VNF Chain Manager requests to the Local Controller the VNF Instance IDs (action 6) and the management operations related to the service and to the single VNFs composing it (action 7).

Using the VNF UIDs given to it by the VNF Chain Manager, the Local Controller accesses the Routing Table and retrieves the pair "VNF Manifest - VNF Instance ID" (actions 8-9). From the VNF Manifest, the Local Controller is able to get all the management operations and monitoring parameters related to a VNF.

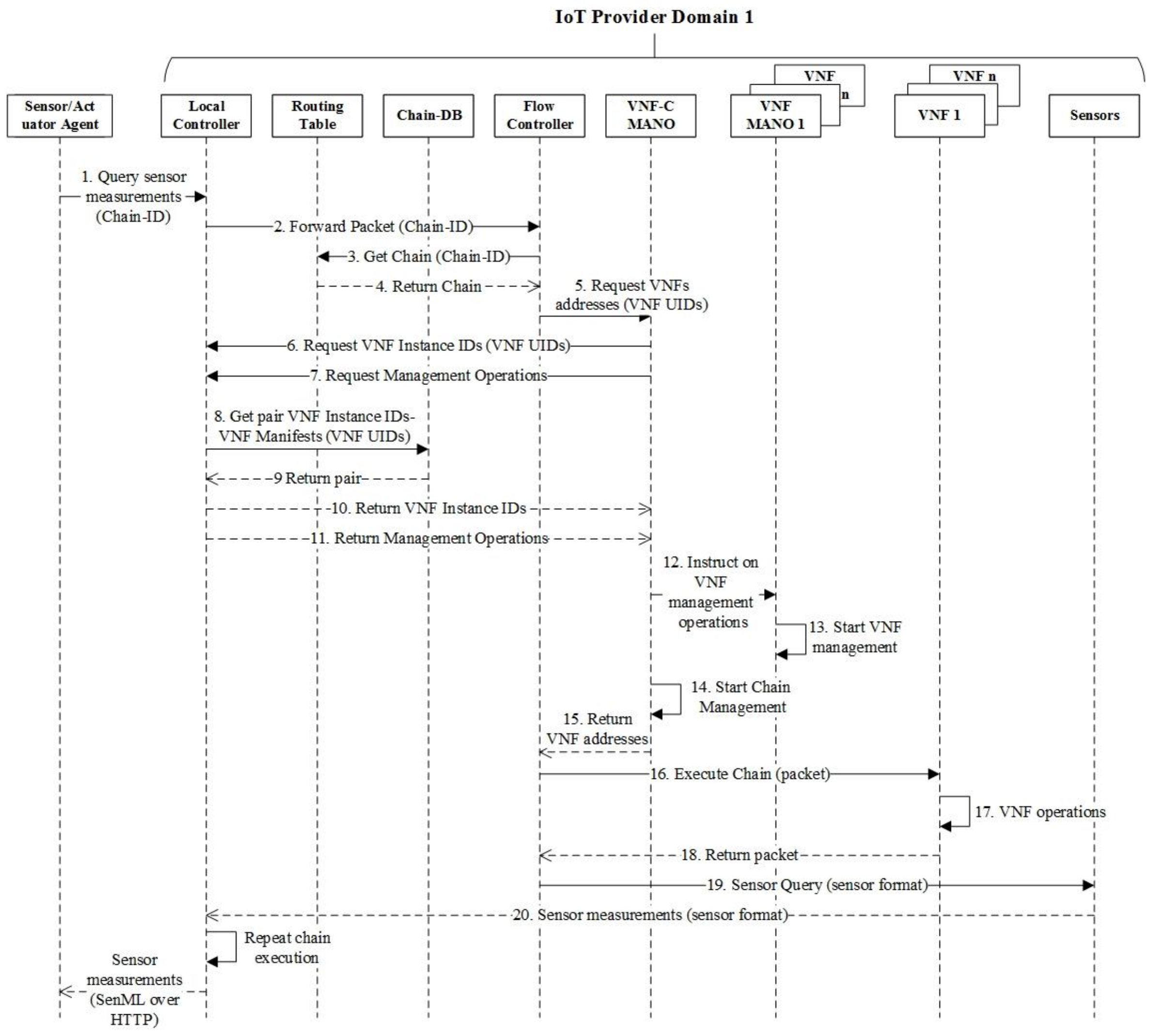

IoT Provider Domain 1
Sensor/Actuator Agent
Local Controller
Routing Table
Chain-DB
Flow Controller
VNF-C MANO
VNF MANO 1
VNF n
VNF 1
VNF n
Sensors
1. Query sensor measurements (Chain-ID)
2. Forward Packet (Chain-ID)
3. Get Chain (Chain-ID)
4. Return Chain
5. Request VNFs addresses (VNF UIDs)
6. Request VNF Instance IDs (VNF UIDs)
7. Request Management Operations
8. Get pair VNF Instance IDs-VNF Manifests (VNF UIDs)
9 Return pair
10. Return VNF Instance IDs
11. Return Management Operations
12. Instruct on VNF management operations
13. Start VNF management
14. Start Chain Management
15. Return VNF addresses
16. Execute Chain (packet)
17. VNF operations
18. Return packet
19. Sensor Query (sensor format)
20. Sensor measurements (sensor format)
Repeat chain execution
Sensor measurements (SenML over HTTP)

Figure 4.12 – Runtime Execution Sequence Diagram

Once the list of VNF Instance IDs and the management operations are returned (actions 10-11), the Chain NFV Manager uses the VNF Instance IDs to retrieve the addresses of the VNFs and it forwards them to the Flow Controller (action 15). Moreover, it instructs the VNF MANOs on the operations they have to perform (actions 12-13). The VNF Chain Manager also starts the monitoring of the whole service (action 14).

After the Flow Controller receives the VNF addresses, it will start the execution of the chain, by sending the packet to the VNFs in proper order (action 16-18). Once the packet has gone through the whole chain, the Flow Controller forwards it to the endpoint (action 19).

The entire process is repeated for each packet sent by an application or IoT device.

6) Runtime Management

As previously stated, the VNF Chain Manager is in charge of monitoring and executing operations that concern the whole chain (e.g., total service time) or that require a complete view of the system (e.g., VNF migration), while the VNF MANOs monitor and execute operations on single VNFs. By doing this, the management for a single VNF is personalized. Also, it is geographically close, since the VNF MANO is in the same PoP as the VNFs it is managing. This aims at achieving high responsiveness for example in case of status changes. For instance, when a VNF is paused, locked, or suspended, or when it sends a notification in response to an event (e.g., the sensed temperature reached a given threshold). Moreover, different chains might correspond to different management operations or monitoring parameters for a single VNF. Therefore, it is advantageous to distribute the management operations, because a single, centralized component may not be able to handle the computational load.

Tables 4.3a and 4.3b show a schematic view of how the management is organized. Table 4.3a displays the monitoring tasks and the management operations that the VNF Chain Manager has to execute. Each chain, identified by its Chain-ID, has associated all its global operations. Table 4.3b shows the tasks related to a single VNF MANO. For each VNF under its supervision, the VNF MANO has a list of all the chains that are executing that VNF. For each chain, all the specific operations and monitoring tasks related to the single VNF are displayed.

Some of the monitoring operations executed locally might trigger actions that change the state of a VNF (e.g., restart), create or delete a VNF (e.g., horizontal scaling), or cause operations that must be executed by the VNF Chain Manager (e.g., VNF migration). In any of these cases, a notification gets sent by the VNF MANO to the Chain one, with all the necessary information. If an operation leads to a configuration change on a VNF (e.g., the VNF switched state or it has been migrated to a different PoP), the VNF Chain Manager must send a notification to the Local Controller. Depending on the content of the message, the Central Controller can execute different actions. For example:

- Sending a notification to the End User Application in case of temporary or definitive service unavailability.
- Accessing the Routing Table to update the entry regarding a VNF. For instance, when a VNF is migrated, its VNF Instance ID changes.
- Adding or removing entries from the Chain-DB. This may happen for example after a VNF scaling-out operation, where part of the traffic going through the scaled VNF has to be routed to the new one.

Table 3a – VNF Chain Manager Management Schema

Chain-ID	Operations
0001	Total Service Time
	Operation 2
	...
0002	Operation 1
	Operation 2
	...

Table 3b – VNF MANO Management Schema

VNF	Chain-ID	Operations
VNF_1	0001	Op. 1
		Op. 2
	0002	Op. 1
VNF_2	0001	Op. 1
		Op. 3
	0004	Op. 2

Regarding the kind of monitoring operations supported by the architecture, a polling system and an event system are both feasible. This is due to the fact that our framework is designed to be general purpose. Consequently, the management is highly configurable. For example, in the same application it is possible to have both a fast polling system and a slow one, monitoring two different VNFs.

4.2.3.2 Control Interfaces

R1 is used for the interactions between Infrastructure Agent and Local Controller. R2 is used for the interactions between the Local Controller and Central Controller. R1 and R2 are based on the REST paradigm. The important information is modeled as resources, and each resource is uniquely identified by its Uniform Resource Identifier (URI).

Table 4.4 summarizes the proposed REST interface for the interactions between the Application Domain and the IoT Provider Domain. It defines resources on the IoT Provider Domain side, used to reserve resources when it receives a service request from the application domain with a description of parameters, such as protocol and information model used, etc. They also allow the Application Domain to modify parameters and delete resource of specific applications. Furthermore, they allow the IoT Gateway Provider domain to send notification to IoT Provider Domain about the availability of the requested VNFs.

Table 4.4 – Resources in the IoT Provider Domain

Resource	Operation	Http Action
List of applications service requests	Create: Add application information (protocol, information model, SLA, etc.)	POST: /ApplicationsServiceRequests
Specific application's service request	Update: Change information of specific application	PUT:/ApplicationsServiceRequests/(RequestId}
	Delete: Delete specific application information	DELETE: /ApplicationsServiceRequests /(RequestId}
Notification of service availability	Create: Send notification to IoT Provider Domain by the IoT Gateway Provider Domain about the availability of requested VNFs.	POST: /ServiceAvailabilityNotification

Follows a detailed description of the resources and their operations:

- List of applications service requests. This resource is a list of all the requests received from the various applications. The information stored includes the description of the northbound interface as well as the SLA between the end user and the IoT Provider. The IoT Provider adds the information related to an application's request using the "Create" operation. It includes all the parameters given by the Infrastructure Agent when it sends a service request. The format is *POST:/ApplicationsServiceRequests.*

- Specific application’s service request. An end user application can execute operations on its service requests. The "Update" operation allows the Infrastructure Agent to change some parameters of one request, such as a northbound interface description or an SLA, by giving its Id. The format is *PUT:/ApplicationsServiceRequests/{RequestId}*. The Infrastructure Agent of an application can also call the "Delete" operation to remove a specific service request. The format is *DELETE:/ApplicationsServiceRequests/{RequestId}*.

- Notification of service availability. This resource is a message containing the response status of the VNFs request operation. The content of the message can be:
    - "*VNFs unavailable"* if one or more requested VNFs were not found in the VNF Manifest repository;
    - "*Chain unavailable"* if none of the chains stored in the IoT Gateway Provider domain are compatible with the VNFs requested;
    - "*OK"* followed by the list of chained VNF Manifests and the related VNF Instance IDs, if the operation succeeded.

  The "Create" operation associated is called by the IoT Gateway Provider and it allows to send the message. The format is POST:/ServiceAvailabilityNotification.

Table 4.5 – Resources in the IoT Gateway Provider Domain

Resource	Operation	Http Action
Request for VNFs	Create: Send request from IoT Provider Domain to IoT Gateway Provider Domain for VNFs with requested VNF Manifests.	POST: /VNFsRequest
Specific request for VNFs	Update: Change information of specific request for VNFs.	PUT: /VNFsRequest/{VNFsRequest Id}
	Delete: Delete information of specific request for VNFs.	DELETE: /VNFsRequest/{VNFsRequest Id}

Table 4.5 summarizes the proposed REST interface for the communication between the IoT Provider Domain and the IoT Gateway Provider Domain. It defines resources on the IoT Gateway Provider Domain side. These resources allow the IoT Provider Domain to send VNF request to the IoT Gateway Provider Domain, including information such as protocols, data models, etc. They also allow the IoT Provider Domain to update or delete information (e.g., sensor/ robot brand) about specific VNF request.
More in detail, the resources are:

- Request for VNFs. This resource stores the list of requested manifests associated to a VNFs request. An IoT Provider can call the "Create" operation to request the VNFs necessary for a service. It passes the list of requested VNF Manifests that must be found and chained by the IoT Gateway Provider. The format of the request is *POST:/VNFsRequest* followed by the list of manifests.

- Specific request for VNFs. An IoT Provider can execute specific operations on its requests. The "Update" operation allows the provider to add or remove requested VNF Manifests. The format is *PUT:/VNFsRequest/{VNFsRequestId}*. An IoT Provider can also call the "Delete" operation to remove a specific request for VNFs. The format is *DELETE:/VNFsRequest/{VNFsRequestId}*.

4.2.4 End to End Scenario

This section presents an end-to-end scenario, wherein a forest monitoring application queries the sensors owned by IoT Provider 1 to collect their measurements, and a wildfire management application needs to be notified when fire occurs to deploy robots. Two different gateways are required, however they will share some of their constituent modules. Before using the IoT Provider Domain's service, the signaling procedure starts. The northbound interface description sent to the Local Controller for both sensors and robots is SenML over HTTP. On top of that, the forest monitoring application requests a metadata extractor, while the wildfire management one requests a data analyzer and aggregator.

Upon receiving the description from the Infrastructure Agent, the Local Controller obtains a description of the sensors (i.e., Advanticsys) and the robots (i.e., Lego Mindstorms) from the OSS/BSS. This information is combined with the one obtained from the infrastructure agent to generate the Request VNF Manifest files. The signaling procedure continues as described above for both applications. In the IoT Gateway Provider the VNFs are discovered, instantiated, migrated to the IoT Provider Domain, and then chained to obtain the requested gateways.

After service negotiation, the Sensor/Actuator Agent of the forest monitoring application sends a query to the sensors through the VNFs. Upon receiving the query, Advanticsys sensors send their raw measurements over CoAP. These measurements are first processed by metadata extraction, which will store the sensor's metadata, followed by protocol conversion (encoded in HTTP), and by information model conversion (mapped to SenML format) in order to enable the application to interpret the measurements.

When the sensors send their measurements to the wildfire management application, the data is first elaborated by data analysis and aggregation, which will forward it only when a certain threshold is reached. Then, the data is processed by the same protocol conversion and information model conversion. If the application receives notification of a fire, it sends actuating commands to the robots in SenML format through HTTP, where the commands are mapped to the LeJOS Java API and Lego

Communication Protocol (LCP). The end-to-end service is completed when the robots are deployed.

Chapter 5

Implementation and Validation

This chapter is divided into two main sections. The first one discusses current solutions that can be adopted to implement some modules of our architecture. The second section presents the proof of concept we implement in order to validate the requirements of our architecture.

5.1 Current Solutions

This section presents and evaluates possible solutions for the implementation and the deployment of our architecture. The first part will cover the deployment environments, while the second one will discuss the existing solutions for implementing a management and orchestration system.

5.1.1 Environment Solutions

Infrastructure-as-a-Service (IaaS) is one of the three key facets of cloud computing, along with Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS). It is the actual dynamic pool of physical and virtualized computing resources used by applications.

In our architecture, NFVI is the set of resources necessary to deploy and execute the VNFs. By definition, IaaS corresponds to both the physical and virtual resources in the NFVI, as shown in [2]. Consequently, IaaS solutions can be used as NFVI for our project. Three popular IaaS implementations will be presented: Microsoft Azure, Amazon Web Services and Openstack, with a short focus on the Smart Applications on Virtual Infrastructure (SAVI) testbed.

5.1.1.1 Microsoft Azure

Microsoft Azure [34] is a proprietary solution developed by Microsoft for IaaS and Cloud computing in general. It allows to build, deploy, and manage applications and services through a global network of Microsoft-managed data centers.

Focusing on IaaS aspects, Azure permits its users to launch general-purpose Microsoft Windows and Linux virtual machines, as well as preconfigured machine images for popular software packages. Data access and storage on the cloud are provided through REST and SDK APIs. Azure grants its customers the ability to create hybrid public/private clouds, and provides services such as identity and access control, and monitoring and management of resources.

The great advantage of Microsoft Azure is that all the services and data centers are Microsoft-based, which allows for easier systems integration. The major drawback is that the platform is neither free nor open source. For this reason, we did not use it.

5.1.1.2 Amazon Web Services

Amazon Web Services (AWS) [35] is a proprietary product developed by Amazon.com. It offers a suite of cloud-computing services that make up an on-demand computing platform. These services include Amazon Elastic Compute Cloud (EC2), and Amazon Simple Storage Service (S3).

Amazon EC2 allows users to rent virtual compute resources, on which they can run their own applications. EC2 encourages scalable deployment of applications by providing a web service through which a user can configure a virtual machine, or instance, containing any software desired. A user can create, launch, and terminate server instances as needed. EC2 adopts a pay-as-you-go billing system.

Amazon S3 provides scalable object storage accessible from a Web Service interface. Applicable use cases include backup/archiving, file storage and hosting, static website hosting, application data hosting, and more.

Other services offered by AWS include networking, database, identity and access management, and resource management.

AWS is a comprehensive and widely used option for IaaS. However, it is a paid service, like Azure. For this reason, we did not employ it as our NFVI.

5.1.1.3 Openstack

OpenStack [36] is a free and open-source software platform for cloud computing. It consists of interrelated components that control diverse, multi-vendor hardware pools of processing, storage, and networking resources throughout a data center. Users either manage it through a web-based dashboard, command-line tools, or a RESTful API. OpenStack has a modular architecture, therefore users have the option to choose which elements to install. Figure 5.1 shows an example configuration in which the core components have been installed.

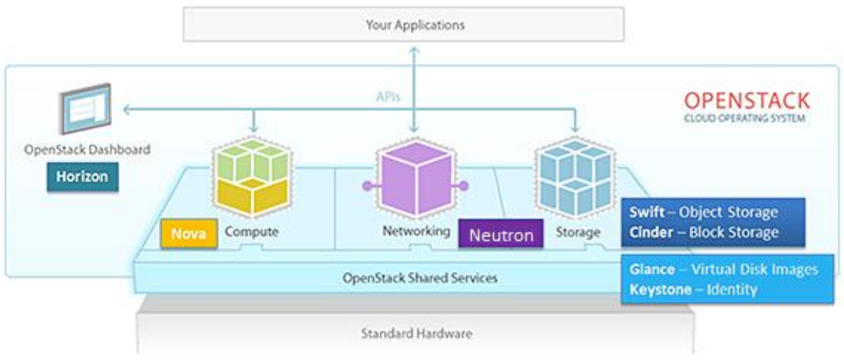


Figure 5.1 – Core OpenStack Configuration

Nova is the compute service of OpenStack. It is designed to manage and automate the lifecycle of compute instances, and it can work with widely available virtualization technologies, as well as bare metal and high-performance computing configurations. Its architecture is designed to scale horizontally on standard hardware. Nova responsibilities include on-demand spawning, scheduling and decommissioning of machines.

Neutron is a system for managing networks and IP addresses. It enables network connectivity as a service for other OpenStack services, such as Nova. It also provides an API for users to define networks and the attachments into them. Neutron has a pluggable architecture that supports many popular networking vendors and technologies. Moreover it manages IP addresses, allowing for dedicated static IP addresses or DHCP.

Cinder provides persistent block storage for running compute instances. **Swift** is a scalable, redundant storage system that grants high fault tolerance.

Glance is the OpenStack image service. It provides discovery, storage, and delivery services for virtual machine disk images. It can add, delete, share, or duplicate images. Glance is also used by other modules. For example, during instance provisioning, Nova makes use of this service to retrieve the image that will run in the instance.

Keystone is the identity service that provides an authentication and authorization service for other OpenStack services. It supports multiple forms of authentication including standard username and password credentials, token-based systems and AWS-style (i.e. Amazon Web Services) logins. Additionally, Keystone provides a catalog of all of the services deployed in an OpenStack cloud in a single registry.

Other OpenStack services include **Horizon** (dashboard), **Heat** (orchestration), **Ceilometer** (Telemetry), etc.

Openstack is our choice for NFVI, since it is a free and comprehensive platform for deploying, executing and managing VNFs. In particular, we deployed our VNFs on the SAVI testbed.

SAVI [37] is a partnership of Canadian industry, academia, research and education networks, and high performance computing centers to investigate key elements of future application platforms (e.g., platforms for IoT). The main research goal of the SAVI Network is to address the design of such platforms built on a flexible, versatile and evolvable infrastructure. This infrastructure can readily deploy, maintain, and retire the large-scale, possibly short-lived, distributed applications that will be typical in the future applications marketplace.

The testbed at our disposal is based on OpenStack and provides all the core services needed (compute, storage, networking, and identity).

5.1.2 Management Solutions

In section 4.2.1, the VNF Chain Manager and the VNF MANO have been introduced. These modules are necessary in order to monitor and manage the chains, and the VNFs composing them. In recent years, tools and platforms have been developed for the implementation of management components. This section will compare some of them, and demonstrate the choice for our prototype.

5.1.2.1 OPNFV

Open Platform for NFV (OPNFV) [38] is a platform to facilitate the development and evolution of NFV components across various open source ecosystems. Its current release (Colorado) provides an implementation of the NFV framework, including MANO. Figure 5.2 shows an architectural view of the Colorado release.

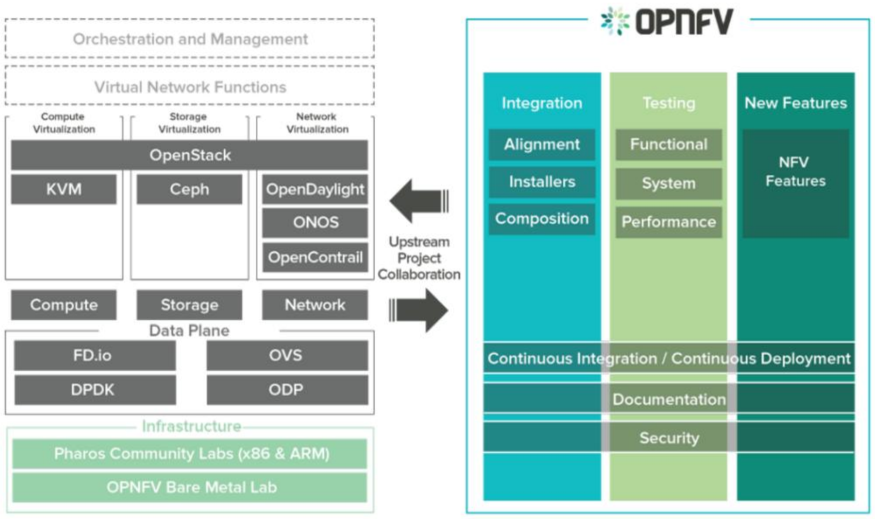


Figure 5.2 – OPNFV Colorado Release Architectural View

Key features of OPNFV include critical advances in security, IPv6, SFC, and VPN capabilities. Integration and automation of testing projects are supported, as well as deep cross-project collaborations with upstream communities such as OpenStack, OpenDaylight, ONOS, Open Baton, etc.

In our case, OPNFV can be implemented as a fully featured OpenStack environment with SFC and MANO modules in order to manage and orchestrate the services and the traffic.

5.1.2.2 Open Source MANO

ETSI Open Source MANO (OSM) [39] is an operator-led ETSI community that delivers a production-quality open source Management and Orchestration (MANO) stack aligned with ETSI NFV Information Models and that meets the requirements of production NFV networks. Release ONE (Figure 5.3) substantially enhances interoperability with other components (VNFs, VIMs, SDN controllers) and creates a plug-in framework to make platform maintenance and extensions significantly easier to provide and support.

The run-time scope of OSM includes:

- An automated end-to-end Service Orchestration environment. It enables and simplifies the operations performed during the lifecycle of a complex service based on NFV.
- A superset of ETSI NFV MANO, where the salient additional area of scope includes Service Orchestration, but also explicitly includes provision for SDN control.
- A plug-in model for integrating multiple SDN controllers.
- A plug-in model for integrating multiple VIMs, including one reference VIM that has been optimized for Enhanced Platform Awareness (EPA) to enable high performance VNF deployments.
- Integration of a “Generic” VNFM with support for integrating “Specific” VNFMs.
- Support for OSM to integrate Physical Network Functions into an automated Network Service deployment.
- GUI, CLI and REST interfaces to enable access to all features.

The design-time scope of OSM includes:

- Capability to execute Create/Read/Update/Delete (CRUD) operations on the Network Service Definition.

- Support for a Model-Driven environment with Data Models aligned with ETSI NFV MANO.
- Simplified VNF Package Generation.
- Graphical User Interface (GUI) to accelerate the network service design time phase.

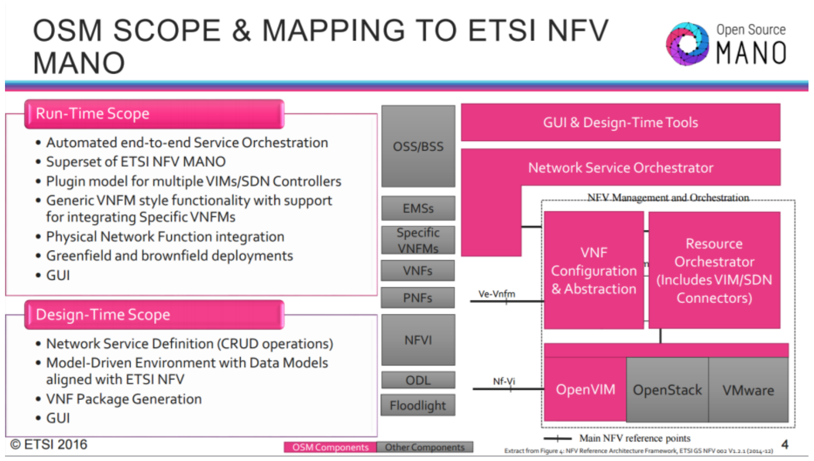

OSM SCOPE & MAPPING TO ETSI NFV MANO
Open Source MANO
Run-Time Scope
• Automated end-to-end Service Orchestration
• Superset of ETSI NFV MANO
• Plugin model for multiple VIMs/SDN Controllers
• Generic VNFM style functionality with support for integrating Specific VNFMs
• Physical Network Function integration
• Greenfield and brownfield deployments
• GUI
Design-Time Scope
• Network Service Definition (CRUD operations)
• Model-Driven Environment with Data Models aligned with ETSI NFV
• VNF Package Generation
• GUI
OSS/BSS
EMSs
Specific VNFMs
VNFs
PNFs
NFVI
ODL
Floodlight
GUI & Design-Time Tools
Network Service Orchestrator
NFV Management and Orchestration
VNF Configuration & Abstraction
Resource Orchestrator (Includes VIM/SDN Connectors)
Ve-Vnfm
Nf-Vi
OpenVIM
OpenStack
VMware
Main NFV reference points
© ETSI 2016
OSM Components
Other Components
Extract from Figure 4: NFV Reference Architecture Framework, ETSI GS NFV 002 V1.2.1 (2014-12)
4

Figure 5.3 – OSM Release 1 Scope

5.1.2.3 Cloudify

Cloudify [40] is an open source pure-play generic cloud orchestrator, optimized for management and orchestration targeted towards NFV. It is designed to integrate seamlessly with networking standards and modeling languages such as ETSI & MANO, YANG, NETCONF/RESTCONF, and TOSCA (Topology and Orchestration Specification for Cloud Applications). Cloudify provides many plugins to interface with peripheral network databases, such as DNS and LDAP, and SDN controllers, including full service management and chaining.

Cloudify is capable of fulfilling the role of NFVO, VNF manager (VNFM), and has multi-VIM capabilities. The Cloudify manager can support multiple clouds (e.g., VMware, OpenStack) as well as multiple data centers and availability zones. Its high level architecture is shown in Figure 5.4.

Cloudify is OpenStack native and it is based on TOSCA [41]. TOSCA is a modeling language based on YAML that provides specifications to describe cloud resources and applications topologies as typed graphs. With TOSCA it is possible to define the topology template of an application, the dependency between the components, their type, etc. It is also possible to define automated management operations that get triggered in response to monitored events (e.g., CPU or memory usage).

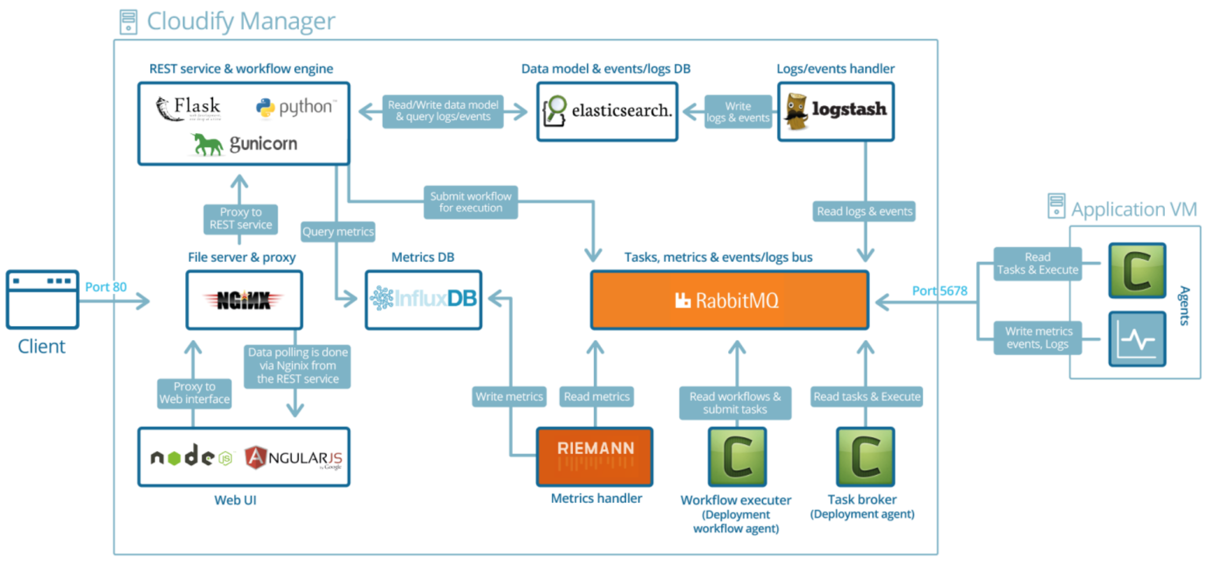


Figure 5.4 – Cloudify Manager Architecture

5.1.2.4 Open Baton

Open Baton [42] is an ETSI NFV compliant MANO framework. It enables virtual Network Services deployments on top of heterogeneous NFV Infrastructures. Open Baton is easily extensible, it integrates with OpenStack, and provides a plugin mechanism for supporting additional VIM types. It supports Network Service management either using a generic VNFM or interoperating with a VNF-specific VNFM. It uses different mechanisms (REST or PUB/SUB) for interoperating with the VNFMs. Open Baton also provides runtime management of Network Services. For instance, it provides auto scaling and fault management, based on monitoring information coming from the monitoring system available at the NFVI level.

Figure 5.5 shows the high level architecture of Open Baton. Its main components and features are:

- A Network Function Virtualization Orchestrator (NFVO) designed and implemented following the ETSI MANO specification.

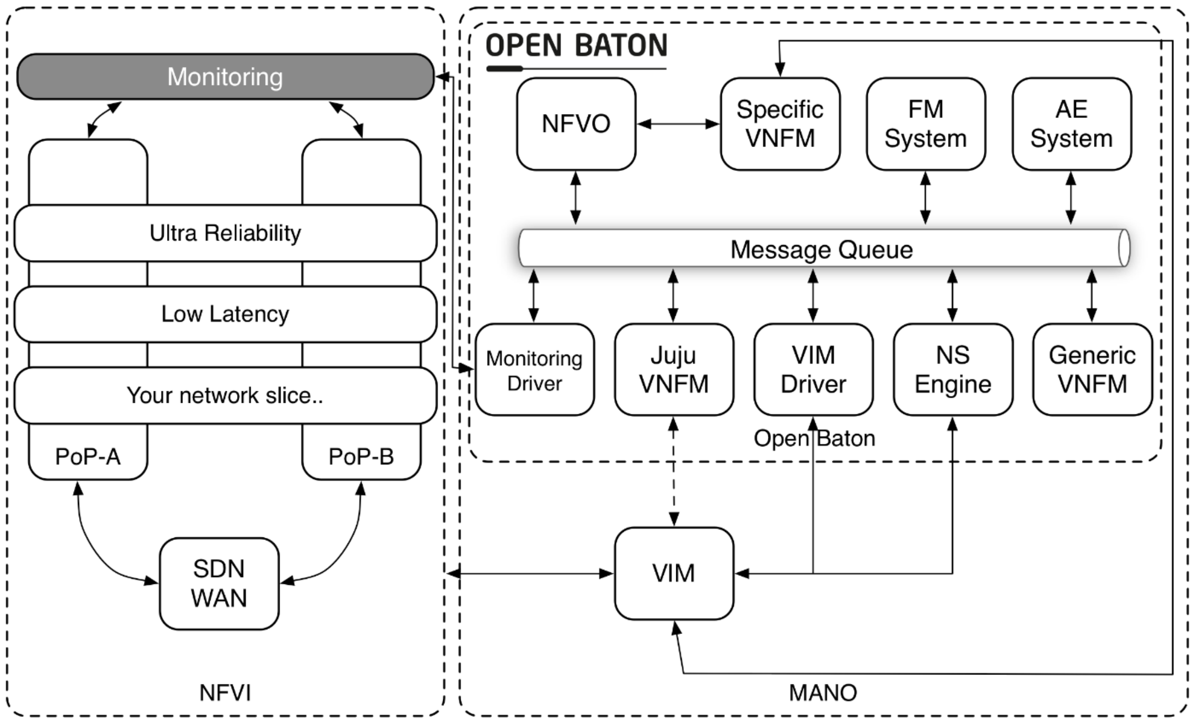


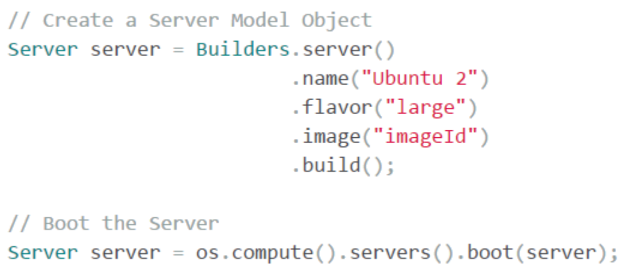
Figure 5.5 – Open Baton Architecture

- A generic Virtual Network Function Manager (VNFM) able to manage the lifecycle of VNFs based on their descriptors.
- A Juju VNFM Adapter, in order to deploy Juju Charms or Open Baton VNF Packages using the Juju VNFM.
- A plugin mechanism for adding and removing different type of VIMs without having to re-write the orchestration logic.
- An event engine based on a pub/sub mechanism for the dispatching of lifecycle events execution.
- An auto scaling engine which can be used for automatic runtime management of the scaling operations of the VNFs.
- A fault management system which can be used for automatic runtime management of faults which may occur at any level.

- Integration with the Zabbix monitoring system.
- A Marketplace useful for downloading VNFs compatible with the Open Baton NFVO and VNFM.

5.1.2.5 Openstack4j Java library

OpenStack4j [43] is an open source OpenStack client which allows provisioning and control of an OpenStack system in a Java environment. It is a fluent based API that permits full control over the various OpenStack services. Openstack4j provides a Java library for each major OpenStack component. Figure 5.6 shows an example on how a server can be created and booted using the API.



```
// Create a Server Model Object
Server server = Builders.server()
                        .name("Ubuntu 2")
                        .flavor("large")
                        .image("imageId")
                        .build();

// Boot the Server
Server server = os.compute().servers().boot(server);
```

Figure 5.6 – Launching an Instance with Openstack4j API

Although it does not provide an implementation for the ETSI MANO architectural elements, Openstack4j is our tool of choice for implementing the management modules of the architecture. The main reason being that it allows for great integration with the OpenStack NFVI environment, and it permits to validate our architecture without the configuration of any external, and potentially complex, tool. Cloudify could be employed as well, but it has one major drawback: it is not possible to execute automated workflows in response to a user defined event, like a message or a method invocation. For example, in our forest monitoring scenario, the manager would not be able to automatically react if the temperature exceeds a threshold. Moreover, none of the other platforms and software presented allow for direct control and configuration from Java code. Another advantage of Openstack4j is that it allows us to run the full prototype within a single execution, since the management is perfectly integrated with our code.

5.2 Implementation

In order to validate the requirements listed in section 3.2, we built a Proof of Concept (PoC) that implements our proposed architecture. First, the software architecture is presented, followed by the setup of the environment. The last paragraph covers the validation process, along with some implementation details of the PoC.

5.2.1 Software Architecture

For the PoC, we implemented the scenario in which a forest monitoring agency is interested in collecting environmental data to monitor a forest. IoT devices have already been deployed in the forest to monitor it. Two different kinds of sensors were used. The sensors measure the temperature and can thereby detect fire outbreaks.

In order to communicate with different types of sensors, the application needs a gateway for handling different types of communication interfaces and different operations. A third party provider provides this gateway.

Figure 5.7 depicts the prototype architecture. Four VNFs are implemented:

- Protocol Conversion (PC). The protocol conversion function decodes the sensor measurements received in one protocol and encodes them into another one. In our PoC, since the sensor communicates using the HTTP protocol, and the application is employing HTTP as well, no operation is actually executed on the packet.

- Information Model Conversion (IMC). This function converts the representation model of the sensed data. It translates raw sensor measurements into the Sensor Markup Language (SenML) format. SenML is a JSON-like data representation designed to encode sensor measurements and device parameters. It contains named events together with an associated value and unit.

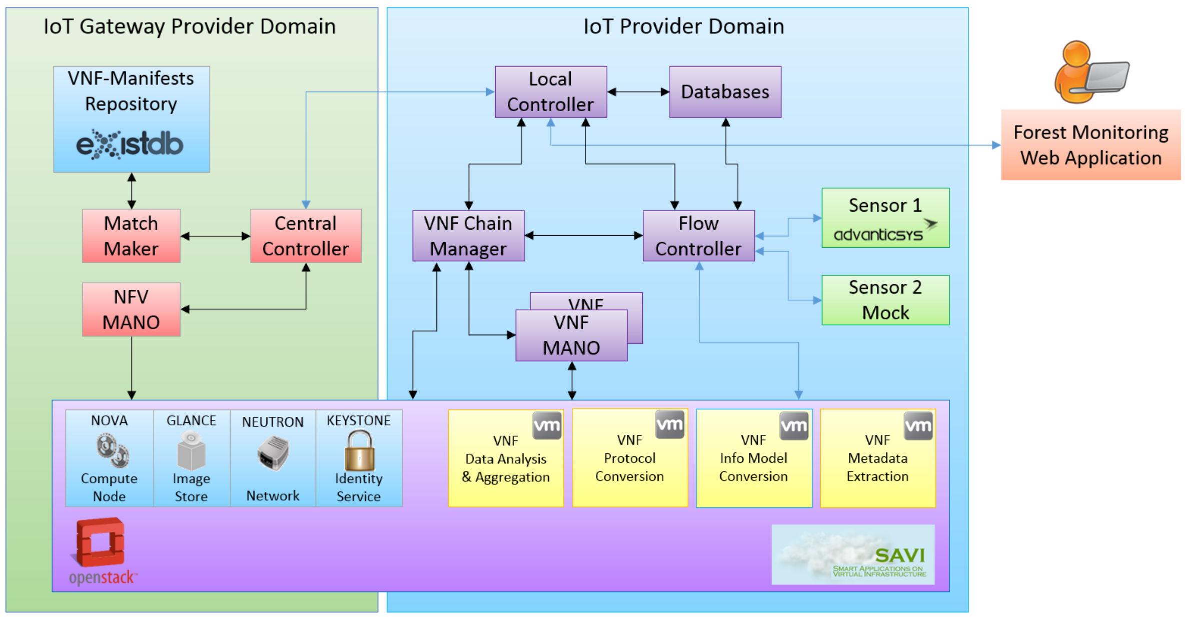


Figure 5.7 – Prototype Architecture

- Metadata Extraction (ME). The metadata extraction function is in charge of pulling out the metadata that is sent alongside a packet. This data can then be used by an application to execute specific operations (e.g., statistics). In the prototype, the VNF removes the metadata from the packet and prints them on standard output.

- Data Analysis and Aggregation (DAA). This function collects sensor measurements and forwards them only when the actual value breaches a given delta. This VNF grants two benefits: it reduces the overall traffic and it prevents from delivering unnecessary or unwanted data to the end user. For instance, an application that is only interested in temperature measurements above 35°C will not receive values below that threshold.

All the functions are implemented as Java dynamic web applications and they are hosted on Tomcat8 servers. The VNFs are deployed in the SAVI testbed. As mentioned in 5.1.1.3, the testbed is an OpenStack environment in which the following necessary components are installed: Identity Service-Keystone, Compute-Nova, Image-Glance, and Networking-Neutron.

The forest monitoring application was created using the Java dynamic web application and hosted on a Tomcat8 server. It simply displays the content of the packets received.

In the IoT Gateway Provider domain, the VNF Manifests Repository is implemented using eXistdb [44], an open source native XML database. The database stores all the published VNF Manifests. The Central Controller, the NFV MANO and the Matchmaker have been implemented as Plain Old Java Objects (POJO). The Matchmaker can access the VNF Manifest Repository using the eXistdb Java API to retrieve the Manifests. It also employs the JAXB library to easily parse and compare the XML Manifests files. The NFV MANO can access the OpenStack NFVI and deploy VNFs using the openstack4j java library.

The IoT Provider domain comprises a Java dynamic web application, a TelosB sensor, and a mock sensor. The application implements all the architectural modules discussed in chapter 4. The sensor implements a program that sends temperature measurements to the system through a USB connection. In the Java application, the SerialComm class connects to the sensor and receives its measurements. This is achieved through the RXTXcomm Java library, which allows to communicate with devices connected to a computer. The mock sensor is a Java class that emulates the behavior of a real sensor. It sends measurements along with some metadata.

The Flow Controller communicates with the VNFs and the sensors through a REST interface, using the RESTlet framework [45]. Communication between the Local Controller and the Central Controller, and between the Local Controller and the application domains are also achieved via REST interfaces.

In the PoC two chains are implemented, corresponding to two different gateways. The first gateway is used for the communication with the TelosB sensor. It is composed by the DAA VNF, the PC VNF, and the IMC VNF. The second gateway is interposed between the mock sensor and the End User Application. It is built using the ME VNF, the PC VNF and the IMC VNF. Figure 5.8 gives a schematic overview of the two chains.

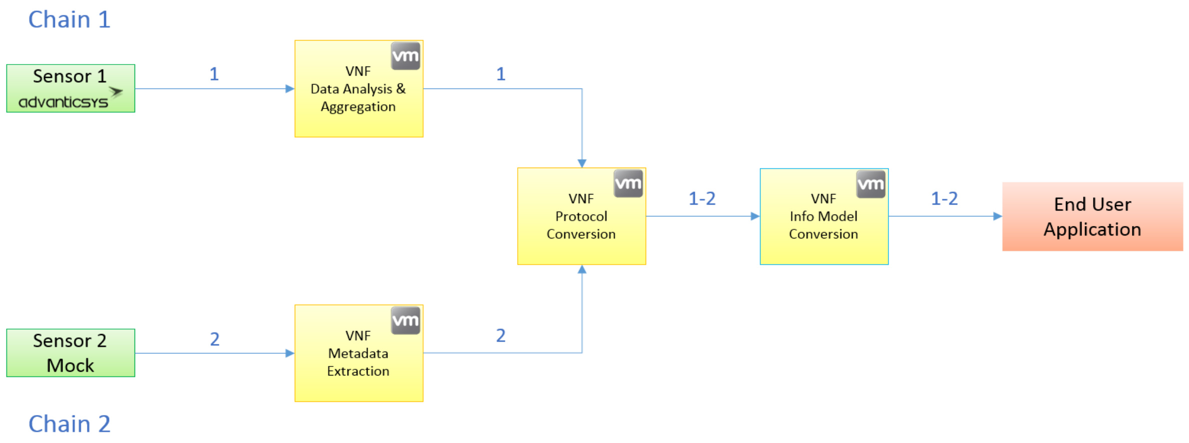


Figure 5.8 – Implemented Chains

Three different management operations are implemented in the prototype:

- Monitoring of total service time for Chain 2. The monitoring is executed by the VNF Chain Manager, which has to verify the time it takes for a packet to go through the gateway. If it is greater than a given threshold, a notification is printed in the standard output.

- Status monitoring of IM VNF, executed by a VNF MANO. The operation is implemented using a polling mechanism. Every few seconds, the VNF MANO checks the status of the VM hosting the VNF and, according to the reading, some operations might be executed. For instance, if the machine is in *pause* state, an "unpause" operation is executed.

- Management of notifications from the DAA VNF. It is a Publish/Subscribe mechanism in which the DAA is the publisher and a VNF MANO is one subscriber. Every time the DAA successfully forwards a packet, it also sends a notification to the message handler. This in turn will forward the notification to all its subscribers, in our case, the VNF MANO. The VNF MANO will simply print to standard output the notification received.

5.2.2 Setup

The various applications run on a PC with Intel® Core™ I7-3770 clocked at 3.40 GHz and 16 GB RAM with 64-bit Windows 7 Enterprise. This PC uses JVM version 1.8.0_65. Each VNF runs Linux Ubuntu Server 14.04 64bit on one VM, and is equipped with 2 VCPUs and 4 GB RAM. One TelosB sensor and one Java class acting as a sensor were used.

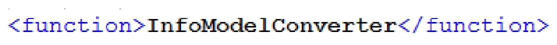
TelosB [46] is a lightweight mote included in the Advanticsys kit. It has multiple on-board sensors, but very low processing and storage capabilities. It supports popular operating systems like TinyOS [47] and Contiki [48]. TelosB is an early generation sensor and it is mainly used to demonstrate the support for legacy sensors and heterogeneity in the proposed architecture. The mote used in our PoC has an 8 MHz Texas Instruments® MSP430F1611 microcontroller with 10kB RAM, and it runs ContikiOS 3.0 as operative system. The temperature sensor range goes from -40°C to 123.8°C, with a resolution of 0.01°C and a ± 0.4°C accuracy.

5.2.3 Implementation details

This section will present the implementation details of the PoC.

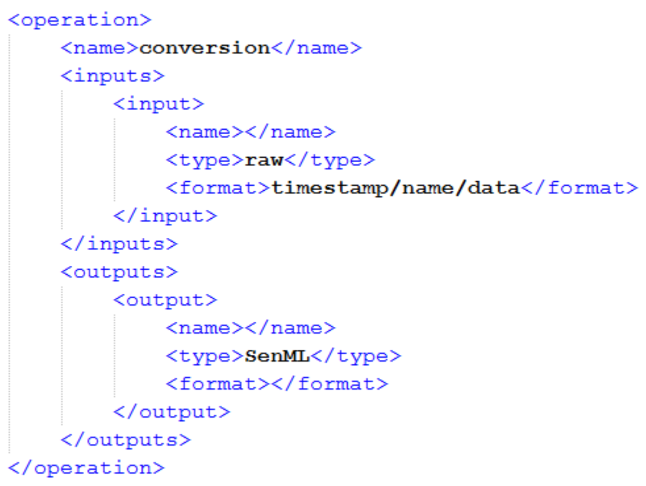
5.2.3.1 Description of the functions for a given application

The execution flow of the prototype starts in the Central Controller of the IoT Gateway Provider. We assume that it has already received the requested VNF Manifests from the Local Controller. The requested VNF Manifests are XML files that describe all the information needed in order to retrieve the proper VNFs. The structure of all the manifests, both requested and published, is defined by an XML Schema Definition (XSD) file. Figures 5.9a to 5.9c show the relevant elements of the XML file that describes a requested IMC function. The Manifest requests one "conversion" operation that takes raw sensor data as input ad converts them into SenML. It also declares that the NFVI of the IoT Provider has a VM with flavor 3 (2 VCPU, 4GB RAM) in which jdk1.8.0 and Apache Tomcat 8.0.26 are installed.



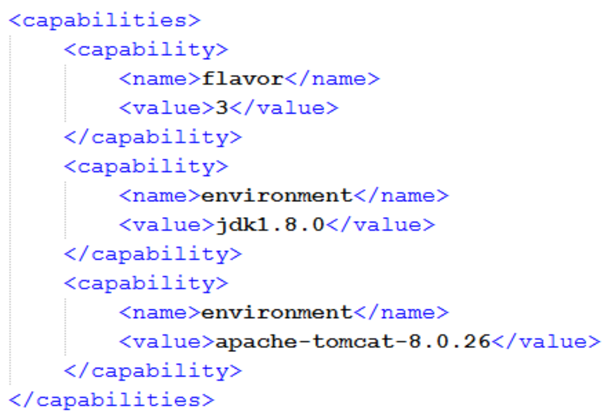
```
<function>InfoModelConverter</function>
```

Figure 5.9a – Function name



```
<operation>
    <name>conversion</name>
    <inputs>
        <input>
            <name></name>
            <type>raw</type>
            <format>timestamp/name/data</format>
        </input>
    </inputs>
    <outputs>
        <output>
            <name></name>
            <type>SenML</type>
            <format></format>
        </output>
    </outputs>
</operation>
```

Figure 5.9b – Operation, inputs, and outputs



```
<capabilities>
    <capability>
        <name>flavor</name>
        <value>3</value>
    </capability>
    <capability>
        <name>environment</name>
        <value>jdk1.8.0</value>
    </capability>
    <capability>
        <name>environment</name>
        <value>apache-tomcat-8.0.26</value>
    </capability>
</capabilities>
```

Figure 5.9c – Environment Capabilities

5.2.3.2 Discovery of the required functions for a given application

The first step done by the Central Controller is to call the matching procedure on the Matchmaker, by giving it the list of requested manifests. The matchmaker connects to the eXistdb repository and retrieves all the published manifests, as shown in figure 5.10.

```
final String URI = "xmldb:exist://localhost:8080/exist/xmlrpc/db/vnfrepository";
final String driver = "org.exist.xmldb.DatabaseImpl";
// initialize database driver
Class<?> cl = Class.forName(driver);
Database database = (Database) cl.newInstance();
database.setProperty("create-database", "true");
DatabaseManager.registerDatabase(database);
Collection col = DatabaseManager.getCollection(URI);
```

Figure 5.10 – Connect and Retrieve from eXistdb Repository

Then, the matching algorithm gets executed between each retrieved file and the set of requested manifests. The operation closely follows the one described in section 4.2.3.1. Figure 5.11 shows how the JAXB XML parser can be used to simplify the matching procedure. From the manifest XSD file, it creates a "Manifest" class containing all the fields and properties of that manifest. The "unmarshall ()" method takes an XML file as input and creates an instance of the Manifest class, with all the properties filled with the values in the XML file.

```
jc = JAXBContext.newInstance("Manifest.jaxb");
Unmarshaller unmarshaller = jc.createUnmarshaller();
Manifest offer = (Manifest) unmarshaller.unmarshal(reader);
Manifest request = (Manifest) unmarshaller.unmarshal(req);

// <function> match
if (request.getFunction().equals(offer.getFunction()))
{
    // all other matches...
}
```

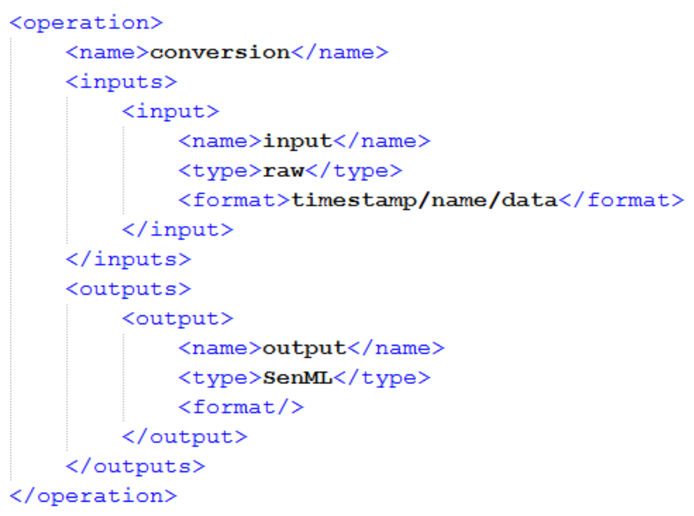
Figure 5.11 – JAXB Unmarshalling

If the operation succeeds, the proper VNF Manifest is stored into a list. Figures 5.12a to 5.12c shows the relevant elements of the manifest obtained for an Info Model Converter function. The <image location> tag defines the ID of the OpenStack image containing the VNF, while the <servlet> tag defines the path that must be used to access it.

If the number of published Manifests obtained is equal to the number of the requested ones, the overall procedure succeeds and the list of offered VNF Manifests is returned to the Central Controller.

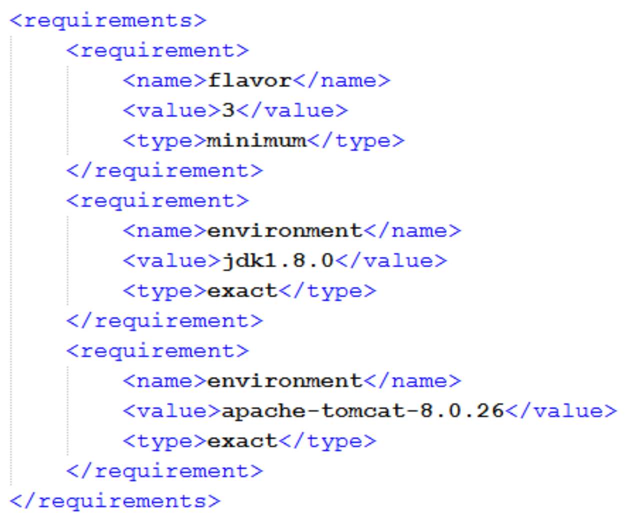
```
<imageLocation>03c93767-7f6c-4cf0-840b-3745f25df82c</imageLocation>
<servlet>/InfoModelServer/InfoModelServer/InfoModelReceiver</servlet>
```

Figure 5.132a – Image Location and Servlet



```
<operation>
    <name>conversion</name>
    <inputs>
        <input>
            <name>input</name>
            <type>raw</type>
            <format>timestamp/name/data</format>
        </input>
    </inputs>
    <outputs>
        <output>
            <name>output</name>
            <type>SenML</type>
            <format/>
        </output>
    </outputs>
</operation>
```

Figure 5.12b – Published Operation

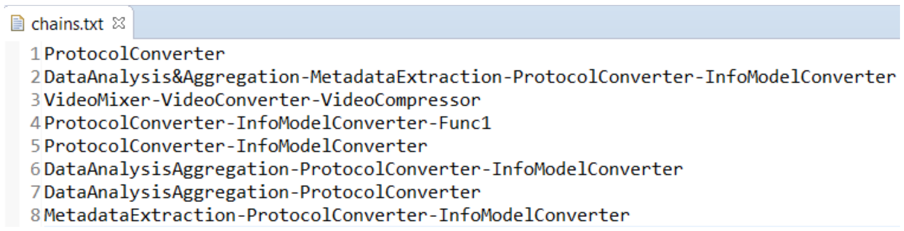


```
<requirements>
    <requirement>
        <name>flavor</name>
        <value>3</value>
        <type>minimum</type>
    </requirement>
    <requirement>
        <name>environment</name>
        <value>jdk1.8.0</value>
        <type>exact</type>
    </requirement>
    <requirement>
        <name>environment</name>
        <value>apache-tomcat-8.0.26</value>
        <type>exact</type>
    </requirement>
</requirements>
```

Figure 5.12c – Environment Requirements

5.2.3.3 Chaining of the functions

The Central Controller maintains a file that contains pre built, abstract chains. These chains define the order in which the VNFs need to be executed to obtain the services offered by the IoT Gateway Provider. Figure 5.13 shows the file used in out prototype, with one chain per line. Each element in a chain is a function name and can be compared with the <function> tag of a VNF Manifest.



```
chains.txt
ProtocolConverter
DataAnalysis&Aggregation-MetadataExtraction-ProtocolConverter-InfoModelConverter
VideoMixer-VideoConverter-VideoCompressor
ProtocolConverter-InfoModelConverter-Func1
ProtocolConverter-InfoModelConverter
DataAnalysisAggregation-ProtocolConverter-InfoModelConverter
DataAnalysisAggregation-ProtocolConverter
MetadataExtraction-ProtocolConverter-InfoModelConverter
```

Figure 5.13 – Chains.txt File

Once the VNF Manifests are obtained from the matchmaking procedure, the Central Controller tries to order them by pairing each Manifest with an element of one of the abstract chains. If all the Manifest get paired with all the elements of one chain, we can order the Manifests according to that chain. Follows a pseudocode of the algorithm and the corresponding results.

```
Input: 'manifests', the set of Manifests obtained from the match-
making procedure.
Output: 'chain', the ordered list of the Manifests.

Foreach Manifest m in manifests:
```

Retrieve the value of the <function> tag from the Manifest and open the chains file.

```
   func <- m.getFunction;
   open file ("chains.txt");
   foreach line in file:
```

Split the abstract chain to get the single elements.

```
   elements[] = line.split("-");
```

Check if the value of the <function> tag from the Manifest is equal to the first element of an abstract chain, and if the size of the abstract chain is equal to the number of Manifests.

```
   if (func.equals(elements[0]) & elements.size = manifests.size)
      chain.add(man);
```

Check if all the remaining elements of the abstract chain match with all the remaining Manifests.

```
      for element e2 in elements:
         foreach Manifest m2 in manifests:
            func2 <- m2.getFunction;
            if (e2.equals(func2))
               chain.add(m2);
```

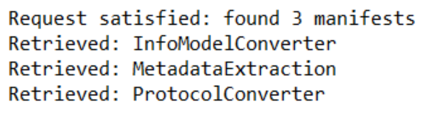
To verify whether the operation succeeded or not, the size of the resulted concrete chain is compared with the size of the set of Manifests. If the check is positive, the ordered list of manifests is returned, otherwise the algorithm proceeds with the next abstract chain in the “chains” file.

```
   if (chain.size = manifests.size)
     return chain;
```

In the end, if no chain is found, a failure message is raised.

```
return “no chain found”;
```

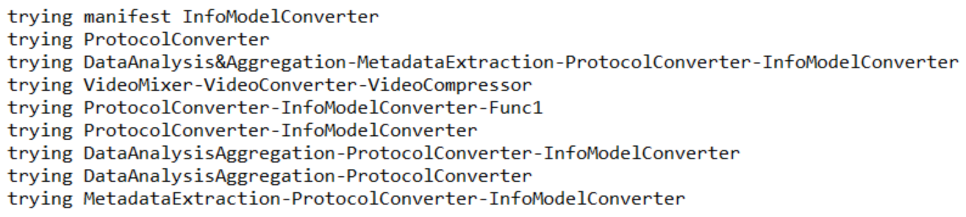
Results: Figure 5.14 shows the three Manifests obtained from the matchmaking procedure. These will be the input of the chaining function.



```
Request satisfied: found 3 manifests
Retrieved: InfoModelConverter
Retrieved: MetadataExtraction
Retrieved: ProtocolConverter
```

Figure 5.14 – Retrieved VNF Manifests

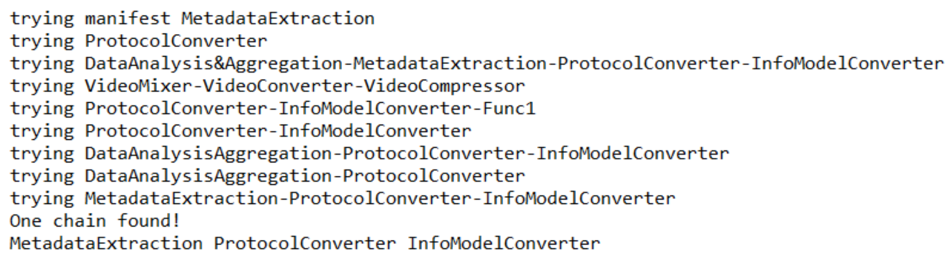
Figure 5.15 shows that the chaining algorithm could not find an abstract chain that started with an IMC function. The ME function is tried next.



```
trying manifest InfoModelConverter
trying ProtocolConverter
trying DataAnalysis&Aggregation-MetadataExtraction-ProtocolConverter-InfoModelConverter
trying VideoMixer-VideoConverter-VideoCompressor
trying ProtocolConverter-InfoModelConverter-Func1
trying ProtocolConverter-InfoModelConverter
trying DataAnalysisAggregation-ProtocolConverter-InfoModelConverter
trying DataAnalysisAggregation-ProtocolConverter
trying MetadataExtraction-ProtocolConverter-InfoModelConverter
```

Figure 5.15 – Chaining Algorithm Failure

Figure 5.16 shows that the algorithm was able to find an abstract chain that started with the ME function, and that the two other elements of it matched with the remaining two functions. The three Manifests obtained from the Matchmaker are now chained.



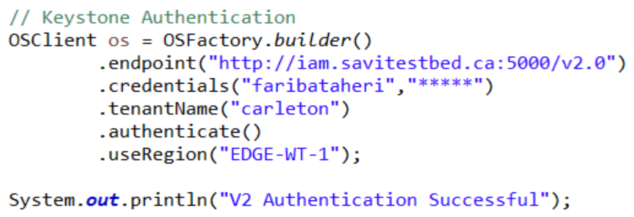
```
trying manifest MetadataExtraction
trying ProtocolConverter
trying DataAnalysis&Aggregation-MetadataExtraction-ProtocolConverter-InfoModelConverter
trying VideoMixer-VideoConverter-VideoCompressor
trying ProtocolConverter-InfoModelConverter-Func1
trying ProtocolConverter-InfoModelConverter
trying DataAnalysisAggregation-ProtocolConverter-InfoModelConverter
trying DataAnalysisAggregation-ProtocolConverter
trying MetadataExtraction-ProtocolConverter-InfoModelConverter
One chain found!
MetadataExtraction ProtocolConverter InfoModelConverter
```

Figure 5.16 – Chaining Algorithm Success

5.2.3.4 VNF deployment

The NFV MANO in the Gateway Provider domain is able to determine whether a VNF has already been deployed or not. The Central Controller calls it after the chaining procedure, and passes to it the ordered list of VNF Manifests. The MANO first verifies if a VNF has been previously deployed, then it eventually deploys it. The following three actions are performed:

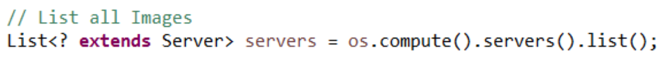
1) Using the Openstack4j API, the MANO accesses the OpenStack domain by providing proper credentials (Figure 5.17).



```
// Keystone Authentication
OSClient os = OSFactory.builder()
        .endpoint("http://iam.savitestbed.ca:5000/v2.0")
        .credentials("faribataheri","*****")
        .tenantName("carleton")
        .authenticate()
        .useRegion("EDGE-WT-1");

System.out.println("V2 Authentication Successful");
```

Figure 5.17 – Openstack Authentication

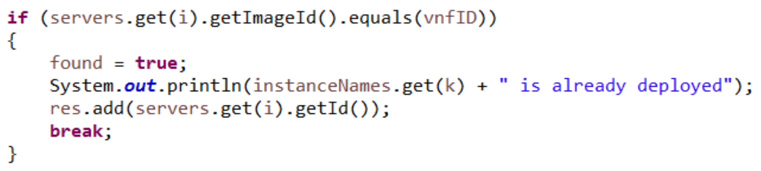
2) The list of all the running servers is retrieved (Figure 5.18).



```
// List all Images
List<? extends Server> servers = os.compute().servers().list();
```

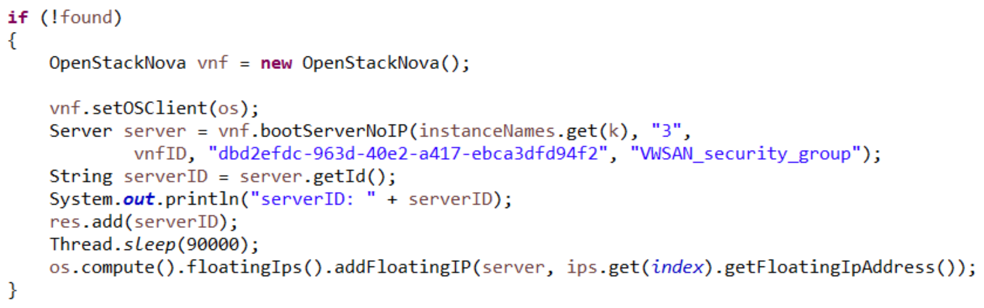
Figure 5.18 – Retrieval of Running Instances

3) From the “Server” object, the Image ID is retrieved and it is compared with the one contained in the VNF Manifest, under the <imageLocation> tag (Figure 5.19). The Image ID is the ID of the image running on the Server. If there is no match, the VNF is deployed and a public floating IP is assigned to it (Figure 5.20).



```
if (servers.get(i).getImageId().equals(vnfID))
{
    found = true;
    System.out.println(instanceNames.get(k) + " is already deployed");
    res.add(servers.get(i).getId());
    break;
}
```

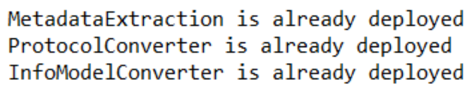
Figure 5.19 – Image ID Comparison



```
if (!found)
{
    OpenStackNova vnf = new OpenStackNova();

    vnf.setOSClient(os);
    Server server = vnf.bootServerNoIP(instanceNames.get(k), "3",
            vnfID, "dbd2efdc-963d-40e2-a417-ebca3dfd94f2", "VWSAN_security_group");
    String serverID = server.getId();
    System.out.println("serverID: " + serverID);
    res.add(serverID);
    Thread.sleep(90000);
    os.compute().floatingIps().addFloatingIP(server, ips.get(index).getFloatingIpAddress());
}
```

Figure 5.20 – VNF Deployment

In both cases, the Instance ID of the VM is stored. Once the procedure has been executed for all the VNFs, the list of instance IDs is returned to the Central Controller. Since the list of VNF Manifests given by the Central Controller was ordered, the Instance IDs are in the same order as the chain. This will be crucial for the IoT Provider to correctly store the entries in the Routing Table. Figure 5.21 shows a scenario where all the VNFs were already deployed in the environment.

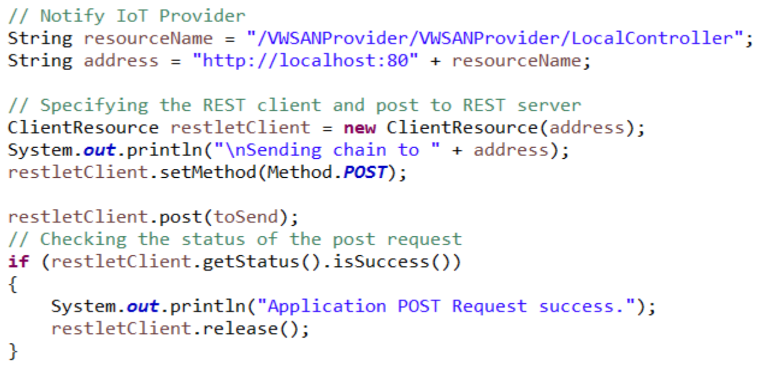


```
MetadataExtraction is already deployed
ProtocolConverter is already deployed
InfoModelConverter is already deployed
```

Figure 5.21 – VNFs Already Deployed

The last action executed by the Central Controller is to send to the Local Controller the chain of VNF Manifests and their corresponding Instance IDs. The operation is performed leveraging the RESTlet framework, which allows to use HTTP methods (GET, POST, DELETE, etc.) to send or receive messages. Figure 5.22 shows that the receiver is first defined, then the message is sent to it as a POST method. Since REST is based on HTTP, a return message containing the status code of the operation is expected.

The Local Controller is implemented as a Servlet, therefore its doPost() method will be executed, once the message is received.



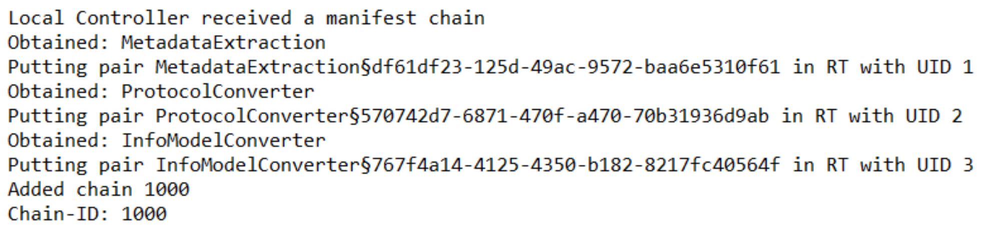
```
// Notify IoT Provider
String resourceName = "/VWSANProvider/VWSANProvider/LocalController";
String address = "http://localhost:80" + resourceName;

// Specifying the REST client and post to REST server
ClientResource restletClient = new ClientResource(address);
System.out.println("\nSending chain to " + address);
restletClient.setMethod(Method.POST);

restletClient.post(toSend);
// Checking the status of the post request
if (restletClient.getStatus().isSuccess())
{
    System.out.println("Application POST Request success.");
    restletClient.release();
}
```

Figure 5.22 – RESTlet Framework POST Request

5.2.3.5 Chain setup in the IoT Provider domain

As depicted in 4.2.1, the role of the Local Controller at this stage is to store the pairs "VNF Manifest - Instance ID" in the Routing Table, and the chain in the Chain-DB. However, The Local Controller must first verify if any of the pairs is already stored. The same goes for the chain. The two databases are implemented as Java maps in a singleton class. In the future, persistent storage will be considered, but for testing and validating purposes a Java class is sufficient.

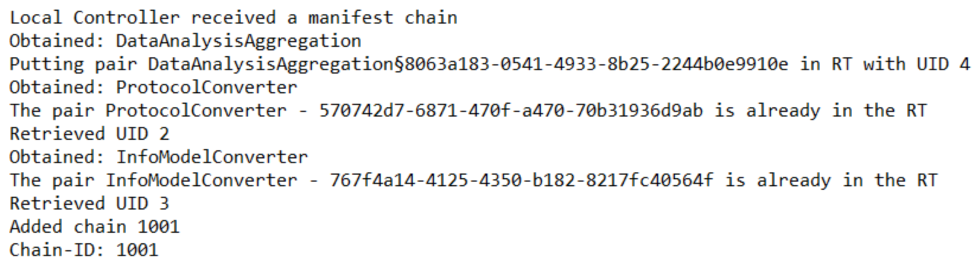
Figure 5.23 shows that the three manifests received were not stored in the Routing Table, so they were added to it. Every time a new pair is added, a VNF UID is generated. Also, a new Chain ID is generated, and the pair "Chain ID - list of VNF UIDs" is put into the Chain-DB.



```
Local Controller received a manifest chain
Obtained: MetadataExtraction
Putting pair MetadataExtraction§df61df23-125d-49ac-9572-baa6e5310f61 in RT with UID 1
Obtained: ProtocolConverter
Putting pair ProtocolConverter§570742d7-6871-470f-a470-70b31936d9ab in RT with UID 2
Obtained: InfoModelConverter
Putting pair InfoModelConverter§767f4a14-4125-4350-b182-8217fc40564f in RT with UID 3
Added chain 1000
Chain-ID: 1000
```

Figure 5.23 – Storage of Manifests and Chains

Figure 5.24 shows a second execution with a different chain, where two of the three pairs "VNF Manifest - Instance ID" were already stored in the Routing Table. In this case, the corresponding VNF UID is retrieved. A new chain is then added to the Chain-DB.



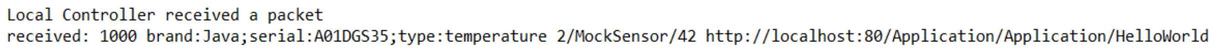
```
Local Controller received a manifest chain
Obtained: DataAnalysisAggregation
Putting pair DataAnalysisAggregation§8063a183-0541-4933-8b25-2244b0e9910e in RT with UID 4
Obtained: ProtocolConverter
The pair ProtocolConverter - 570742d7-6871-470f-a470-70b31936d9ab is already in the RT
Retrieved UID 2
Obtained: InfoModelConverter
The pair InfoModelConverter - 767f4a14-4125-4350-b182-8217fc40564f is already in the RT
Retrieved UID 3
Added chain 1001
Chain-ID: 1001
```

Figure 5.24 – Manifests Already Stored in the Routing Table

The Local Controller then sends to the end user (in our scenario, the sensors) the Chain ID of the gateway requested, using a RESTlet POST method.

5.2.3.6 Runtime flow orchestration

When the sensor sends a measurement to the Local Controller, it sends a packet containing the Chain ID, followed by the actual data and the address of the receiver. Figure 5.25 shows the packet received by the Local Controller.

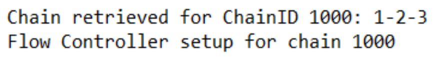


```
Local Controller received a packet
received: 1000 brand:Java;serial:A01DGS35;type:temperature 2/MockSensor/42 http://localhost:80/Application/Application/HelloWorld
```

Figure 5.25 – Packet Received by the Sensor

The Local Controller forwards the packet to the Flow Controller, whose job is to send it to the various VNFs, according to the chain. The operation is accomplished in 5 steps:

1) Using the Chain ID, the Flow Controller retrieves the chain (list of VNF UIDs) from the Chain-DB (Figure 5.26) and stores it in a local cache. The cache allows to skip steps 2 and 3 when multiple packets are received from the same source.



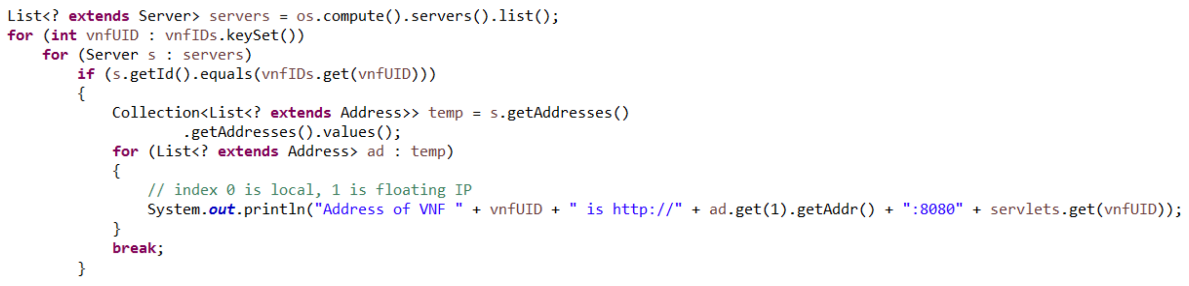
```
Chain retrieved for ChainID 1000: 1-2-3
Flow Controller setup for chain 1000
```

Figure 5.26 – Chain Retrieval

2) The Flow Controller requests to the VNF Chain Manager the addresses of the VMs that are hosting the VNFs.

3) The VNF Chain Manager forwards the request to the Local Controller. The latter accesses the Routing Table using the given VNF UIDs, and obtains the VM Instance IDs and the VNF servlet paths. With the Instance ID, The VNF Chain

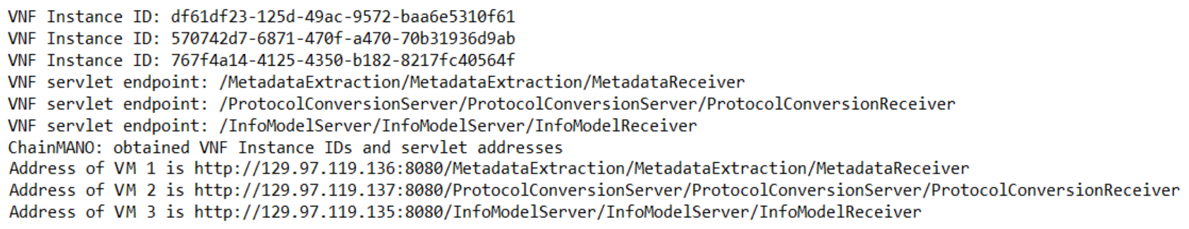
Manager accesses the VM and retrieves its address, using the openstack4j APIs (Figure 5.27).



```
List<? extends Server> servers = os.compute().servers().list();
for (int vnfUID : vnfIDs.keySet())
    for (Server s : servers)
        if (s.getId().equals(vnfIDs.get(vnfUID)))
        {
            Collection<List<? extends Address>> temp = s.getAddresses()
                    .getAddresses().values();
            for (List<? extends Address> ad : temp)
            {
                // index 0 is local, 1 is floating IP
                System.out.println("Address of VNF " + vnfUID + " is http://" + ad.get(1).getAddr() + ":8080" + servlets.get(vnfUID));
            }
            break;
        }
```

Figure 5.27 – VM Address Retrieval

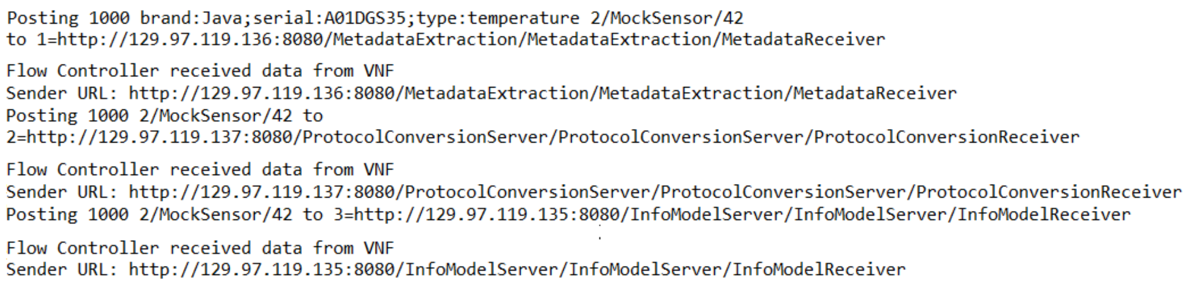
Then, the manager combines the address with the servlet path to obtain the full path, and returns it to the Flow Controller (Figure 5.28).



```
VNF Instance ID: df61df23-125d-49ac-9572-baa6e5310f61
VNF Instance ID: 570742d7-6871-470f-a470-70b31936d9ab
VNF Instance ID: 767f4a14-4125-4350-b182-8217fc40564f
VNF servlet endpoint: /MetadataExtraction/MetadataExtraction/MetadataReceiver
VNF servlet endpoint: /ProtocolConversionServer/ProtocolConversionServer/ProtocolConversionReceiver
VNF servlet endpoint: /InfoModelServer/InfoModelServer/InfoModelReceiver
ChainMANO: obtained VNF Instance IDs and servlet addresses
 Address of VM 1 is http://129.97.119.136:8080/MetadataExtraction/MetadataExtraction/MetadataReceiver
 Address of VM 2 is http://129.97.119.137:8080/ProtocolConversionServer/ProtocolConversionServer/ProtocolConversionReceiver
 Address of VM 3 is http://129.97.119.135:8080/InfoModelServer/InfoModelServer/InfoModelReceiver
```

Figure 5.28 – Step 3 Results

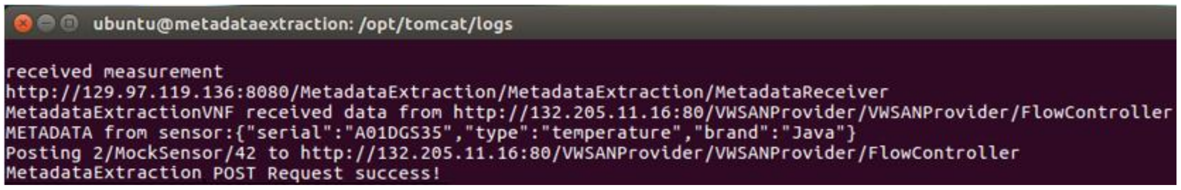
4) The Flow Controller sends and receives the packets from the VNFs (Figure 5.29), using RESTlet POST methods. Before sending, it adds its own address to the packet's header, so the VNFs are able to send back the data. The Chain ID is also added, this way the Flow Controller can recognize which chain a packet belongs to. Moreover, is able to handle multiple packets from different sources at the same time.



```
Posting 1000 brand:Java;serial:A01DGS35;type:temperature 2/MockSensor/42
to 1=http://129.97.119.136:8080/MetadataExtraction/MetadataExtraction/MetadataReceiver

Flow Controller received data from VNF
Sender URL: http://129.97.119.136:8080/MetadataExtraction/MetadataExtraction/MetadataReceiver
Posting 1000 2/MockSensor/42 to
2=http://129.97.119.137:8080/ProtocolConversionServer/ProtocolConversionServer/ProtocolConversionReceiver

Flow Controller received data from VNF
Sender URL: http://129.97.119.137:8080/ProtocolConversionServer/ProtocolConversionServer/ProtocolConversionReceiver
Posting 1000 2/MockSensor/42 to 3=http://129.97.119.135:8080/InfoModelServer/InfoModelServer/InfoModelReceiver

Flow Controller received data from VNF
Sender URL: http://129.97.119.135:8080/InfoModelServer/InfoModelServer/InfoModelReceiver
```

Figure 5.29 – Runtime Flow Orchestration

Figure 5.30 illustrates the console output on the Metadata Extraction VNF. The metadata are successfully removed from the packet before forwarding it back to the Flow Controller.



```
received measurement
http://129.97.119.136:8080/MetadataExtraction/MetadataExtraction/MetadataReceiver
MetadataExtractionVNF received data from http://132.205.11.16:80/VWSANProvider/VWSANProvider/FlowController
METADATA from sensor:{"serial":"A01DGS35","type":"temperature","brand":"Java"}
Posting 2/MockSensor/42 to http://132.205.11.16:80/VWSANProvider/VWSANProvider/FlowController
MetadataExtraction POST Request success!
```

Figure 5.30 – Metadata Extraction VNF Console Output

5) The Flow Controller finally sends the packet to the end user application which will display the data received (Figure 5.31).

```
Posting 1000 {"e":[{"t":"2","v":"42"}],"bn":"MockSensor"} to 100=http://localhost:80/Application/Application/HelloWorld
The Application received a packet!
chain 1000, value 42
```

Figure 5.31 – End User Application Output

The figure also illustrates the SenML format: "e" (events) is a mandatory field containing an array of events, "bn" (base name) acts as a name prefix for every event. The event object inside the events array contains "t" (time) and "v", the numeric value of the measurement.

5.2.3.7 Finer granularity of management

1) Monitoring of total service time

The VNF Chain Manager is in charge of monitoring the time it takes for a packet to traverse the whole chain and arrive to the end user. The timer starts before the packet gets sent to the first VNF and it is stopped right after the packet is sent to the application (Figures 5.32 to 5.34).

```
ChainMANO.startChainMonitoring();
send(chainID, chain.get(0), data);
```

Figure 5.32 – Start Timer

```
restletClient.post(chainID + "\n" + data);

// stop the monitoring timer (vnfUID 100 is the id of any endpoint)
if (vnfUID == 100)
    ChainMANO.stopChainMonitoring();
```

Figure 5.33 – Stop Timer

```
The Application received a packet!
chain 1000, value 42
Total service time for chain '1000' number 1: 407ms
```

Figure 5.34 – Result

If the timer exceeds a given threshold, an alert message is printed (Figure 5.35).

```
Total service time for chain '1000' number 2: 1376ms
ALERT! TOTAL SERVICE TIME EXCEEDS THRESHOLD!
```

Figure 5.35 – Exceeding Timer

2) Status monitoring of Information Model VNF

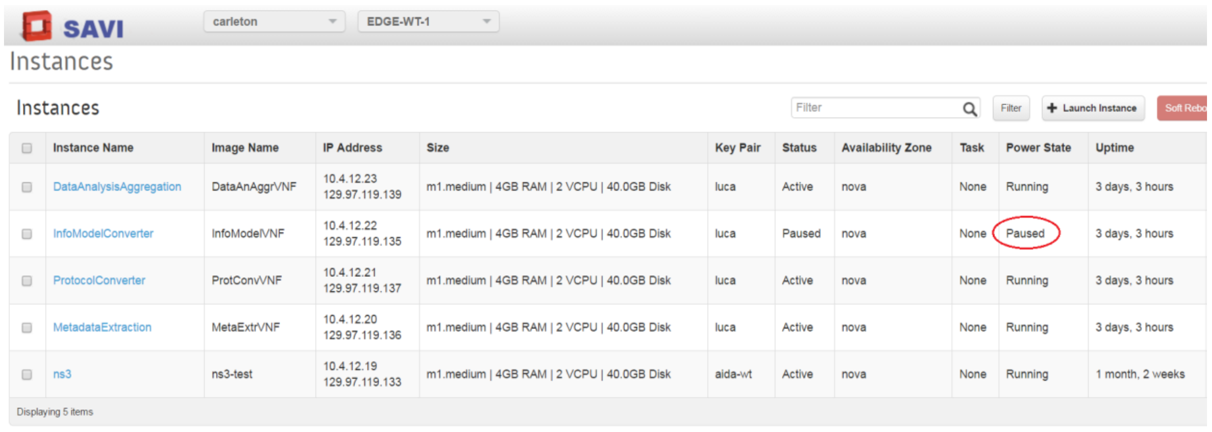
A VNF MANO is in charge of monitoring the status of the VM hosting the IM VNF. Every few seconds the information is collected using an openstack4j method. Depending on the reading, some counter-measurements could be adopted (Figure 5.36). Figure 5.37 shows the console output of the monitoring operation, while Figures 5.38 and 5.39 illustrate the effects of the unpause action on the SAVI testbed environment.

```
String state = os.compute().servers().get(serverId).getVmState();
System.out.println("Current status of InfoModelVNF is " + state);
switch (state)
{
    case "paused":
        os.compute().servers().action(serverId, Action.UNPAUSE);
        break;
    case "suspended":
        os.compute().servers().action(serverId, Action.RESUME);
        break;
    case "locked":
        os.compute().servers().action(serverId, Action.UNLOCK);
        break;
}
```

Figure 5.36 – Info Model VNF Monitoring

```
Current status of InfoModelVNF is active
Current status of InfoModelVNF is active
Current status of InfoModelVNF is paused
Current status of InfoModelVNF is active
Current status of InfoModelVNF is active
Current status of InfoModelVNF is suspended
Current status of InfoModelVNF is active
```

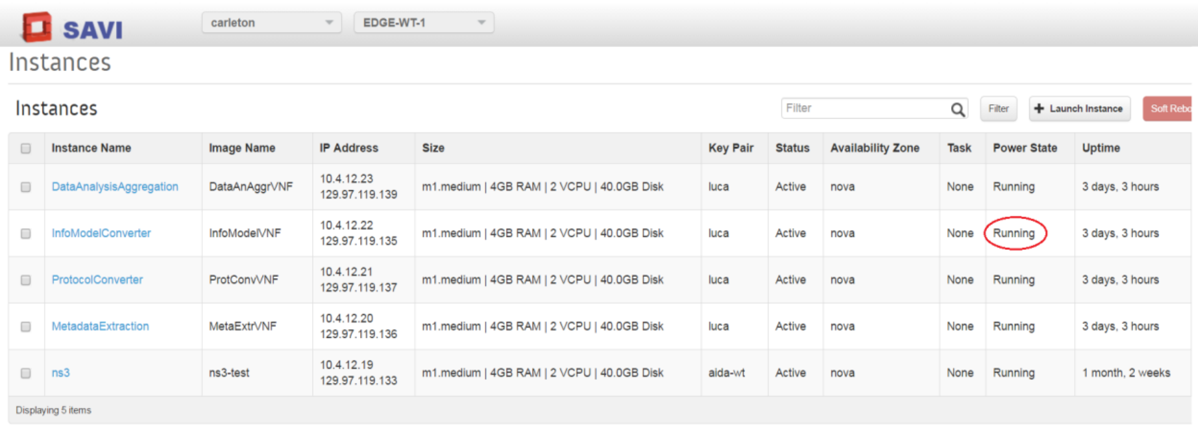
Figure 5.37 – Monitoring Output



Instance Name	Image Name	IP Address	Size	Key Pair	Status	Availability Zone	Task	Power State	Uptime
DataAnalysisAggregation	DataAnAggrVNF	10.4.12.23 129.97.119.139	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	luca	Active	nova	None	Running	3 days, 3 hours
InfoModelConverter	InfoModelVNF	10.4.12.22 129.97.119.135	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	luca	Paused	nova	None	Paused	3 days, 3 hours
ProtocolConverter	ProtConvVNF	10.4.12.21 129.97.119.137	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	luca	Active	nova	None	Running	3 days, 3 hours
MetadataExtraction	MetaExtrVNF	10.4.12.20 129.97.119.136	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	luca	Active	nova	None	Running	3 days, 3 hours
ns3	ns3-test	10.4.12.19 129.97.119.133	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	aida-wt	Active	nova	None	Running	1 month, 2 weeks

Displaying 5 items

Figure 5.38 – SAVI Environment Before Unpause



Instance Name	Image Name	IP Address	Size	Key Pair	Status	Availability Zone	Task	Power State	Uptime
DataAnalysisAggregation	DataAnAggrVNF	10.4.12.23 129.97.119.139	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	luca	Active	nova	None	Running	3 days, 3 hours
InfoModelConverter	InfoModelVNF	10.4.12.22 129.97.119.135	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	luca	Active	nova	None	Running	3 days, 3 hours
ProtocolConverter	ProtConvVNF	10.4.12.21 129.97.119.137	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	luca	Active	nova	None	Running	3 days, 3 hours
MetadataExtraction	MetaExtrVNF	10.4.12.20 129.97.119.136	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	luca	Active	nova	None	Running	3 days, 3 hours
ns3	ns3-test	10.4.12.19 129.97.119.133	m1.medium | 4GB RAM | 2 VCPU | 40.0GB Disk	aida-wt	Active	nova	None	Running	1 month, 2 weeks

Displaying 5 items

Figure 5.39 – SAVI Environment After Unpause

3) Management of notifications from Data Analysis and Aggregation VNF

Every time the DAA VNF forwards data, it means that its internal state is changed. Consequently, a notification message is sent to a message handler. A VNF MANO is subscribed to that handler and it will receive the notification (Figure 5.40). The VNF MANO will simply print the message.

```
Meessage Handler received a message: DataAnalysisAndAggregation CHANGED STATE! NEW STATE: 22.0
LocalMANO RECEIVED A MESSAGE!: DataAnalysisAndAggregation CHANGED STATE! NEW STATE: 22.0
```

Figure 5.40 – Notifications Management

5.2.4 Validation

This section summarizes how each of the architectural requirements presented in section 3.2 is validated. In detail:

- The requirement regarding the discovery of the requested functions is met by using an XSD file to define the structure of the VNF Manifests, and by a matching algorithm capable of retrieving the necessary functions.
- For the finer granularity of deployment, the prototype is capable of deploying the VNFs separately, and each one of them is independent from the others. Moreover, the prototype can also verify if a VNF is already deployed in the environment. In that case, the function will not be redeployed.
- The requirement on dynamic flow orchestration while executing the gateway modules is verified in two phases. In the first one, the PoC derives the service chain, the order in which the VNFs must be executed, using a dedicated algorithm. In the second phase, the prototype sends packets of data through the various gateways. To achieve this, it uses the Chain-DB and the Routing Table, which allow to store and retrieve information regarding the chains and the VNFs.
- For the finer granularity of management, we implemented a system where the VNF Chain Manager is able to monitor parameters related to a whole chain (i.e., total service time) and two VNF MANOs monitor two single VNFs. For this second part, both a polling mechanism and a publish/subscribe system are implemented.

In conclusion, the PoC we implemented meets all the requirements. Consequently, our proposed architecture is validated against them.

Chapter 6

Conclusions and Future Work

In this thesis, we discuss the importance of gateways for the communication between IoT devices and applications. They are necessary to hide the intrinsic heterogeneous nature of IoT devices to the end users. However, gateways are generally complex, and difficult to upgrade when new brands of devices are deployed.

We then introduce a previous research effort that tackled these challenges. It is an NFV-based architecture capable of deploying gateway modules as VNFs, into an NFV infrastructure. We also show that the work does not allow for the reusability of already deployed modules, and it assumes that IoT providers have a centralized environment.

The thesis presents a distributed NFV architecture that faces the aforementioned problems. With NFV, it is possible to deploy the gateway modules into heterogeneous and distributed IoT environments. Moreover, the architecture allows to describe the functions required to compose a gateway, and proposes a matchmaking procedure to discover those functions among the ones published.

Function chaining consents to create and orchestrate the gateways on-the-fly and in a more dynamic way. It also allows to reuse the already deployed modules.

Furthermore, the architecture proposes a hierarchical MANO to separate the chain management from the VNFs one. To that end, the VNF management can be personalized, since each function can belong to different flows corresponding to different applications.

We then discuss a proof of concept of the distributed NFV-based gateway. It validates all the architecture requirements

There are several potential items for future work. One is the use of SDN for the runtime traffic orchestration. By using network switches, it will allow to route the packets directly through the VNFs, instead of steering them every time to a central controller. Another example is the employment of management plans to better define the VNF and the chain management operations.

Yet another example is the design of optimal VNF placement algorithms. These will allow the IoT gateway provider to efficiently deploy the VNFs in the PoP that, for instance minimizes, a given cost function. Finally, another future work we consider is the deployment of the gateway modules, and their corresponding managers, in the sensor themselves. This would permit to employ IoT devices in harsh environments and in catastrophic scenarios. A potential starting point can be [49], where the author shows that the usage of container technologies on top of IoT devices carries negligible performance losses.

Appendix A

References

[1] Hawilo, Hassan, et al. "NFV: state of the art, challenges, and implementation in next generation mobile networks (vEPC)." *IEEE Network* 28.6 (2014): 18-26.

[2] Mijumbi, Rashid, et al. "Network function virtualization: State-of-the-art and research challenges." *IEEE Communications Surveys & Tutorials* 18.1 (2016): 236-262.

[3] ETSI ISG NFV. "ETSI GS NFV 002 V1.1.1: Network Functions Virtualisation (NFV); Architectural Framework." (2013-10).

[4] ETSI ISG NFV. "ETSI GS NFV-MAN 001 V1.1.1: Network Functions Virtualisation (NFV);

Management and Orchestration" (2014-12).

[5] ETSI ISG NFV. "ETSI GS NFV 003 V1.2.1: Network Functions Virtualisation (NFV); Terminology for Main Concepts in NFV." (2014-12).

[6] Khan, Rafiullah, et al. "Future internet: the internet of things architecture, possible applications and key challenges." Frontiers of Information Technology (FIT), 2012 10th International Conference on. IEEE, 2012.

[7] Atzori, Luigi, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey." Computer networks 54.15 (2010): 2787-2805.

[8] https://www.oracle.com/middleware/index.html

[9] wso2.com/platform

[10] http://www.machineshop.io

[11] https://www.redhat.com/en/technologies/jboss-middleware

[12] Mouradian, Carla, et al. "Network functions virtualization architecture for gateways for virtualized wireless sensor and actuator networks." IEEE Network 30.3 (2016): 72-80.

[13] http://www.virtenio.com/en/products/radio-module.html

[14] https://www.advanticsys.com/shop/mtmcm5000msp-p-14.html

[15] Zhu, Qian, et al. "Iot gateway: Bridgingwireless sensor networks into internet of things." *Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on*. IEEE, 2010.

[16] Jiang, Xianyang, et al. "An enhanced IOT gateway in a broadcast system." Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2012 9th International Conference on. IEEE, 2012.

[17] Guoqiang, Shang, et al. "Design and implementation of a smart IoT gateway." *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*. IEEE, 2013.

[18] Datta, Soumya Kanti, Christian Bonnet, and Navid Nikaein. "An IoT gateway centric architecture to provide novel M2M services." *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE, 2014.

[19] Fantacci, Romano, et al. "Short paper: Overcoming IoT fragmentation through standard gateway architecture." *Internet of Things (WF-IoT), 2014 IEEE World Forum on*. IEEE, 2014.

[20] Martins, Joao, et al. "ClickOS and the art of network function virtualization." Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, 2014.

[21] Xilouris, Georgios, et al. "T-NOVA: A marketplace for virtualized network functions." Networks and Communications (EuCNC), 2014 European Conference on. IEEE, 2014.

[22] Monteleone, Giuseppe, and Pietro Paglierani. "Session Border Controller Virtualization Towards" Service-Defined" Networks Based on NFV and SDN." *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013.

[23] Batalle, Josep, et al. "On the implementation of NFV over an OpenFlow infrastructure: Routing function virtualization." *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*. IEEE, 2013.

[24] ETSI ISG NFV. "ETSI GS NFV-IFA 009 V1.1.1: Network Functions Virtualisation (NFV);

Management and Orchestration; Report on Architectural Options" (2016-07).

[25] Riera, Jordi Ferrer, et al. "TeNOR: Steps towards an orchestration platform for multi-PoP NFV deployment." *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE, 2016.

[26] Kataoka, Kotaro, et al. "Orchestrating distributed mode of NFV." NetSoft Conference and Workshops (NetSoft), 2016 IEEE. IEEE, 2016.

[27] Abu-Lebdeh, Mohammad, et al. "A Virtual Network PaaS for 3GPP 4G and Beyond Core Network Services." *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*. IEEE, 2016.

[28] Vilalta, Ricard, et al. "SDN/NFV orchestration of multi-technology and multi-domain networks in cloud/fog architectures for 5g services." OptoElectronics and Communications Conference (OECC) held jointly with 2016 International Conference on Photonics in Switching (PS), 2016 21st. IEEE, 2016.

[29] Mamatas, Lefteris, Stuart Clayman, and Alex Galis. "A service-aware virtualized software-defined infrastructure." *IEEE Communications Magazine* 53.4 (2015): 166-174.

[30] Giotis, Kostas, Yiannos Kryftis, and Vasilis Maglaris. "Policy-based orchestration of NFV services in Software-Defined Networks." *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE, 2015.

[31] Scheid, Eder J., et al. "Policy-based dynamic service chaining in Network Functions Virtualization." *Computers and Communication (ISCC), 2016 IEEE Symposium on*. IEEE, 2016.

[32] Cunha, Vitor A., et al. "Policy-driven vCPE through dynamic network service function chaining." *NetSoft Conference and Workshops (NetSoft), 2016 IEEE*. IEEE, 2016.

[33] Martini, Barbara, and Federica Paganelli. "A Service-Oriented Approach for Dynamic Chaining of Virtual Network Functions over Multi-Provider Software-Defined Networks." *Future Internet* 8.2 (2016): 24.

[34] https://azure.microsoft.com

[35] https://aws.amazon.com

[36] https://www.openstack.org

[37] https://www.savinetwork.ca

[38] https://www.opnfv.org

[39] https://osm.etsi.org

[40] getcloudify.org

[41] https://www.oasis-open.org/committees/tosca

[42] https://openbaton.github.io

[43] www.openstack4j.com

[44] exist-db.org

[45] https://restlet.com

[46] http://www.memsic.com/userfiles/files/Datasheets/WSN/telosb_datasheet.pdf

[47] https://github.com/tinyos

[48] www.contiki-os.org

[49] Morabito, Roberto. "A performance evaluation of container technologies on Internet of Things devices." *Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on*. IEEE, 2016.