Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Intent-Based Service Management for Heterogeneous Software-Defined Infrastructure Domains

Gianluca Davoli[1] | Walter Cerroni*[1] | Slavica Tomovic[2] | Chiara Buratti[1] | Chiara Contoli[1] | Franco Callegati[1]

[1]University of Bologna, Bologna, Italy
[2]University of Montenegro, Podgorica, Montenegro

Correspondence
*Walter Cerroni, DEI - University of Bologna, via Machiavelli, 47522 Cesena (FC) Italy. Email: walter.cerroni@unibo.it

Summary

One of the main challenges in delivering end-to-end service chains across multiple Software Defined Networking (SDN) and Network Function Virtualization (NFV) domains is to achieve unified management and orchestration functions. A very critical aspect is the definition of an open, vendor-agnostic, and interoperable northbound interface (NBI) that should be as abstract as possible, and decoupled from domain-specific data and control plane technologies. In this paper we propose a reference architecture and an intent-based NBI for end-to-end service management across multiple technological domains. The general approach is tested in a heterogeneous OpenFlow/Internet-of-Things (IoT) SDN test bed, where the proposed solution is applied to a rather complex service provisioning scenario spanning three different technological domains: an IoT infrastructure deployment, a cloud-based data collection, processing, and publishing platform, and a transport domain over a geographic network interconnecting the IoT domain and the data center hosting the cloud services.

KEYWORDS:
Software-Defined Networking, Internet of Things, Multi-domain Management and Orchestration, Intent-based Networking

1 | INTRODUCTION

Service provisioning in today's communication infrastructures is being revolutionized by the unprecedented central role of software-based networking solutions, following the recent innovations brought about by cloud computing and resource virtualization[1,2]. In particular, the Network Function Virtualization (NFV) paradigm fosters flexible and cost-effective service provisioning by deploying network functions as pieces of software running on vendor-independent hardware platforms, bringing the benefits of cloud computing to network infrastructure management[3]. At the same time, Software Defined Networking (SDN) decouples software-based network control and management planes from the hardware-based forwarding plane, turning traditional vendor locked-in infrastructures into communication platforms that are fully programmable via a standardized, open, southbound interface (SBI)[4].

In this framework the term *Service Function Chaining (SFC)* is used to describe the deployment of composite services that are obtained from a concatenation, i.e., a *chain*, of one or more basic services typically provided by a single network function implemented in some form of virtualized environment (e.g., virtual machine, container, etc.). The SFC[†] is fundamentally the series of

[†]In this manuscript the SFC acronym will be used to refer to both Service Function *Chaining* and Service Function *Chain*, depending on the context.

service functions that a packet or a traffic flow must traverse from its source to its destination. Thanks to the capabilities offered by SDN and NFV, the SFC can be dynamically controlled and modified over a relatively small time scale, increasing the service provisioning flexibility while significantly reducing the management burden compared to traditional network architectures.

Within a single technological and administrative domain, such as for instance a single data center, the SFC-related operations, i.e., composition, maintenance, modification, etc., can be successfully achieved with the help of the native domain management system, as recently demonstrated by specific proof-of-concept implementations[5]. However, end-to-end services and the related SFCs must very often be provided to customers across different network administrative and/or technological domains. Guaranteeing specific functionality and quality of service has always been a challenging task in multi-domain environments[6,7,8]. One of the most critical issues to achieve unified management and orchestration of end-to-end services across multiple domains is the definition of an open, vendor-agnostic, and interoperable northbound interface (NBI), through which applications are allowed to control the underlying heterogeneous NFV and SDN infrastructures and take advantage of dynamic SFC. Although a standard NBI definition is still under discussion, a commonly accepted approach is to adopt a so-called *intent-based* interface that allows to declare service outcomes and high-level operational goals rather than specify detailed networking mechanisms[9].

In this paper we present a reference architecture and define a related intent-based NBI for end-to-end service management and orchestration across multiple technological domains, extending our preliminary work on heterogeneous OpenFlow/IoT SDN domains[10]. In particular, we consider the use case of a rather complex service provisioning scenario spanning different technological domains:

1. an Internet of Things (IoT) infrastructure deployment, representing the first technological domain;
2. a cloud-based data collection, processing, and publishing platform, representing the second technological domain;
3. a transport domain over a geographic network interconnecting the IoT domain and the data center hosting the cloud services, enhanced with proper SDN control capabilities to implement dynamic SFC.

The data “produced” in the IoT domain are “consumed” inside a cloud domain where different data streams traverse different SFCs. The goal is to dynamically differentiate the Quality of Service (QoS) of different data streams representing different end-to-end services. This is achieved by means of an effective integration of computing and networking resource management in a cloud infrastructure[5]. The infrastructure is fully automated both in the cloud deployment of a given set of network functions and in the capability of reacting to changes in the overall network conditions, safeguarding the service level agreement. The conditions of the network infrastructure underlying the set of functions and implementing the required SFC can be monitored and modified to pursue the QoS objectives for the various active end-to-end services.

The remainder of the paper is organized as follows. In Section 2, we present existing work related to different aspects of our approach. Then we propose our reference architecture and define the intent-based NBI in Sections 3 and 4, respectively. We provide specific examples and technical details related to IoT, cloud and transport domains in Sections 5, 6 and 7. We report the experimental validation in Section 8, and finally conclude the paper in Section 9.

2 | RELATED WORK

2.1 | Intent-based networking

The concept of *intent-based networking* has recently gained increasing attention from both industry and academia. One of the earliest definitions of an intent-based NBI came from the industry[11], and included the following features: invariance, portability, composability, scalability, and context-awareness. Then the first step toward standardization in the SDN context was made by the Open Networking Foundation (ONF)[12], which defined an intent-based NBI as non-prescriptive, provider-independent and declarative. ONF also specified that a set of mechanism, named *mappings*, are required to translate intent NBI requests into forms that lower-level entities can understand, thus making consumer and provider systems separately implemented but able to communicate in terms that are “natural” to each. As discussed in Section 4, we follow the ONF approach in our definition of intent-based NBI, which must allow an abstract yet flexible definition of a service chain, without knowledge of technology-specific details.

Other examples of adoption of an intent-based approach include: abstraction for virtualized network management in a multi-tenant data center environment[13]; high-level specification of network slicing requirements and automated configuration in an SDN infrastructure[14]; definition of a service-oriented architecture for service composition based on microservices[15]; scalable label-based abstraction of policy requirements for large cloud computing environments[16]. However, each of those solutions focuses on a single specific domain, whereas our approach takes advantage of the powerful abstraction level offered by an intent-based NBI to manage end-to-end services across multiple technological domains.

The use of intent-based networking in multi-domain scenarios is still an open research issue. An intent-based mobile backhauling interface for 5G networks has been proposed and prototyped in[17]. The specific characteristics of 5G mobile networks require the integrated management of multiple technological domains in the radio access and backhaul segments, including e.g. Wi-Fi access points and OpenFlow switches. The platform design accounts for several service scenarios, including mobility management, uplink/downlink decoupling, and fine grained packet processing. However, the intent-based interface defined in[17] is not completely decoupled from the underlying infrastructures, as it requires knowledge of low-level details such as switch IDs, port numbers and MAC addresses, making it difficult to extend it to heterogeneous technological domains, such as IoT. More recently, the problem of multi-domain intent decomposition into local intent graphs for each domain has been addressed in[18]. The latter work is complementary to our experimental study reported in this paper, as we do not focus on the intent decomposition problem, but provide a generalized intent-based definition of service function chains to be applied to each domain involved.

The most widely used SDN control platforms offer some sort of intent-based NBIs. OpenDaylight and OpenStack Neutron offer the Group Based Policy (GBP) tool, which allows to specify communication policies (or contracts) between groups of endpoints (i.e., VMs, containers, ports)[19]. For instance, a typical GBP specification could be such as “allow web traffic to web server endpoint group,” which will automatically reconfigure the firewall security groups in order to allow access to the requested service. However, differently from our approach, GBP was not intended for specifying complex SFCs without the support of ad-hoc configuration tools specific to the OpenStack platform. OpenDaylight also provide Network Intent Composition (NIC), which basically allows to specify connectivity requirements between endpoints with redirection, such as “connect endpoint A to endpoint B redirecting through C”[20]. Similarly, ONOS offers the Intent Framework, which is also a way to express connectivity requirements, specifying (even multiple) endpoints or connect-points, e.g., “connect endpoint A to endpoints B and C”[21]. However, differently from our approach, both solutions still require knowledge of low-level details such as switch IDs, port numbers, VLAN IDs, and MAC addresses.

Finally, it is worth to mention that intent-based networking is often perceived as similar to the concept of high-level policy-based network management. This is particularly true in the case of policy refinement techniques, aimed at deriving (or refining) lower-level policies from higher-level, goal-oriented specifications[22]. In our approach we share the point of view recently expressed by IETF concerning the conceptual differences between “intent” and “policy”[9]. Both terms refer to high-level abstractions for managing networks without delving into device-specific details. However, a policy typically involves a set of rules used to define what to do under what circumstances (events, conditions, actions), but it does not necessarily specify a desired outcome. Differently, an intent is used to define network-wide outcomes and high-level operational goals, without the need to enumerate specific events, conditions, and actions. In this sense, policy refinement can be considered equivalent to the ONF mappings needed to translate intents into lower-level policies.

2.2 | Standards for SFC

The implementation of a given SFC that spans several network domains with non homogeneous forwarding technologies is very challenging and is usually solved by means of some form of network overlay (e.g., tunneling). This problem was addressed by the Internet Engineering Task Force (IETF), which suggests that the service-specific overlay can be obtained by applying suitable packet encapsulation[23]. One option being considered by IETF is the so-called *Network Service Header* (NSH) standard[24], which intends to provide a flexible, dynamic, and transport-independent SFC solution for the data plane. The NSH standard focuses on data plane aspects only, and very little has been said about a possible SFC control plane solution. To the best of our knowledge, the only document that attempts to do so is an IETF draft that, at the time of writing, has already expired[25]. Therefore, here we adopt a possible implementation of a NSH-aware control plane inspired by the concepts discussed in the IETF draft[26]. It is based on the use of SDN-like technology inside NSH nodes and on the adoption of the OpenFlow protocol for the communication between the SFC Control Plane and the NSH enabled nodes. This allows for a seamless integration of the related NBI with the NBI adopted in the IoT and cloud domains.

2.3 | Software-defined IoT

IoT facilitates billions of devices to be enabled with network connectivity to collect and exchange information for providing different services. IoT should allow connected devices to be controlled and accessed remotely, in an efficient manner. However, traditional network infrastructures in many cases cannot satisfy IoT requirements and new approaches, based on the application of the SDN concept, have been recently proposed[27]. Different works discuss advantages of applying SDN to IoT[28,29,30]. From the *network management* viewpoint, SDN may enable traffic control and load balancing. As an example, in[31] it is shown that the SDN controller may steer traffic in non-interfered area, improving network performance. When it comes to *resource utilization*, the SDN approach allows viewing nodes as resource providers and to efficiently map the users' requests into the proper resources. Multiple applications could run concurrently on different WSNs by optimizing their resource use according to availability and other cost metrics[32]. Regarding *energy management*, a solution focused on both centralized device and topology management was proposed[33], allowing to switch on/off devices to reduce energy consumption compared to a decentralized solution.

The papers mentioned above report theoretical analysis or discussion, while few practical deployments are available. SDN-WISE is a stateful SDN solution pursuing the reduction of the amount of information exchanged between sensor nodes and the SDN controller[30]. However, no resource management or QoS-based network management is considered. An extension of the 6LoWPAN protocols stack was proposed to implement SDN[31], demonstrating performance improvement with respect to the RPL (Routing Protocol for Low Power and Lossy Networks) based decentralized solution. Results are obtained via a platform emulating TI CC2420 devices and not using real devices as we do in our paper.

In contrast, in this paper we propose an SDN-based IoT architecture and validate it with an experimental testbed integrated with our end-to-end service management platform. The architecture allows to jointly implement resource management and network management, and to characterize performance at different planes (data, control and management plane), as never done in previous literature to the best of our knowledge. The proposed SDN IoT solution builds on existing work[30,34], where however resource management was not allowed and where the multi-domain integration was not present.

3 | REFERENCE NETWORK ARCHITECTURE

The reference multi-domain SDN/NFV architecture considered in this paper is shown in Fig. 1. Although our approach to intent-based service management can be generalized to any SDN/NFV technology domain, the domains included in Fig. 1 are those involved in the use case considered in this paper: data collected from sensor and actuator devices of a software-defined IoT domain are dispatched across the transport network to reach a set of suitable consumers, implemented by means of virtualized network functions (VNFs) and deployed within a cloud computing domain.

Considering the purpose of our study and the nature of the orchestration features we are interested in, our reference architecture is inspired by the ETSI NFV specifications, with particular reference to the Management and Orchestration (MANO) framework[35], although our approach is focused on an end-to-end service perspective. The rationale behind this choice is that, on one hand, the proposed architecture has the advantage to be consistent with the most relevant NFV standard initiative to date; on the other hand, the architecture itself can be seamlessly extended to include any further SDN/NFV domain and technology as part of the underlying virtualized infrastructure.

Each SDN/NFV domain in Fig. 1 consists of a technology-specific infrastructure, including:

- data plane components, such as IoT nodes and gateways, NSH network nodes, SDN switches, virtual machines running in cloud computing nodes, physical and virtual interconnecting links; these components provide the network, compute, and storage resources to be orchestrated;
- control plane components, such as SDN and cloud controllers with related data stores and interfaces; these components are responsible for proper VNF deployment and traffic steering across VNFs and domains;
- management plane components, such as Virtualized Infrastructure Managers (VIMs) and WAN Infrastructure Manager (WAN-IM), for managing resources in the technological domains; based on the available implementations, some of these components could be in charge of multiple domains[36], as in the case of the cloud VIM in Fig. 1.

The overarching VNF Manager (VNFM) and NFV Orchestrator (NFVO) components are responsible for programming the underlying VIMs/WAN-IM and infrastructure controllers in order to implement and maintain the required service chains in a consistent and effective way, for both intra- and inter-domain scenarios. While technology- and domain-specific northbound

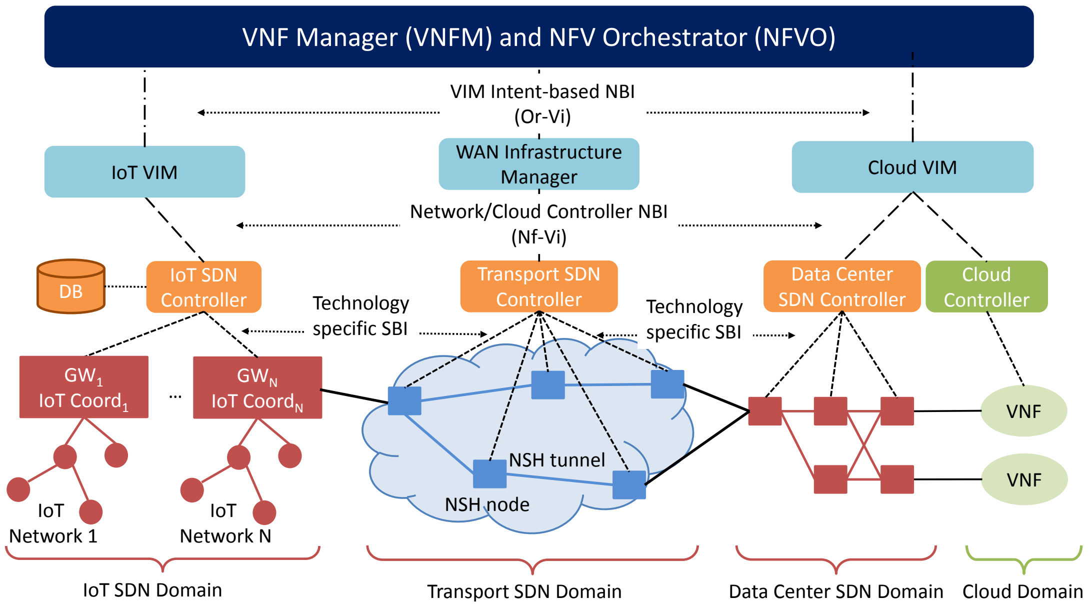

FIGURE 1 Reference multi-domain SDN/NFV architecture. Three different technological domains are displayed here, including an IoT domain, a data center and cloud domain, and a geographical transport network domain.

(NBI) and southbound interfaces (SBI) are used inside each domain to efficiently control and manage the relevant components, the design of the overarching VNFM and NFVO should be as technology-agnostic as possible, so that a service chain to be deployed can be specified by a customer using a high-level, intent-based description of the service itself. This would also allow the proposed architecture to be more general and capable of being extended to different SDN technologies and domains.

In order to achieve such generality in the high-level management and orchestration components, we argue that *the act of decoupling service abstractions from the underlying technology-specific resources should be performed mainly by the infrastructure managers (VIMs and WAN-IM)*. Therefore, we extend the concept of interactions based on intents to the NBI offered by the VIMs/WAN-IM, which should be defined as an open and abstract interface, independent of the specific technology used in the underlying domains. This approach could also allow different administrative domains to expose only service abstractions without disclosing sensitive details related to the underlying infrastructures.

4 | VIM NORTHBOUND INTERFACE

In general, the definition of an open, vendor-agnostic, and interoperable interface will foster improved and standardized procedures for service specification to the underlying multi-domain NFV and SDN platforms. In particular, the powerful abstraction level offered by an intent-based NBI allows to specify service outcomes and high-level operational goals rather than mechanisms, by taking advantage of formalism close to the customer's natural language[9], [12]. Therefore, in our architecture we assume that some kind of intent-based interface is offered to the customer by the overarching VNFM and NFVO components.

When a given service request is received, the high-level management and orchestration functions must convert that request into a set of suitable service chains and pass them to the relevant VIMs[‡] in charge of the underlying infrastructures and domains involved in the service composition. Then each VIM must coordinate the respective controllers in order to:

- verify availability and location in the cloud infrastructure of the VNFs required to compose the specified service, instantiating new ones if needed;
- program traffic steering rules in the network infrastructure to deploy a suitable network forwarding path.

[‡]For the sake of simplicity, in the remainder of the manuscript we use the term VIM to refer also to the WAN-IM in charge of managing the transport domain, considering that the underlying NSH-based network overlay exposes a sort of virtualized infrastructure to the orchestrator.

The NBI exposed by the VIMs should allow an abstract yet flexible definition of the service chain, without knowledge of the technology-specific details such as devices, ports, addresses, etc. This means that a request sent to the VIMs should specify not only the sequence, but also the nature of the different VNFs to be traversed, which is strictly related to the service component they implement, as well as other peculiar characteristics of the service itself, such as quality of service (QoS) metrics and thresholds. In particular, the NBI should allow an abstract representation of the QoS features for the requested service and the topological characteristics of each VNF to be applied in the service chain.

A possible definition of the VIM NBI is presented here, considering the following service and function abstractions.

- A *QoS feature* is defined in qualitative terms relevant to the specified service, e.g. guaranteed bit rate or limited delay.
- A *QoS threshold* can be specified for the metric of interest, e.g. a minimum bit rate or a maximum delay value.
- A VNF can be *terminating* or *forwarding* a given traffic flow. For instance, a deep packet inspection (DPI) function usually terminates a mirrored copy of a given flow, whereas a network address translator (NAT) forwards incoming flows.
- A forwarding VNF can be *port-symmetric* or *port-asymmetric*, depending on whether or not it can be traversed by a given traffic flow regardless of which port is used as input or output. For instance, a NAT is port-asymmetric, because it must receive inbound and outbound traffic from a port connected to a public and private network, respectively. A basic IP routing function can be considered port-symmetric, as it forwards packets based on the destination address.
- A VNF can be *path-symmetric* or *path-asymmetric*, depending on whether or not it must be traversed by a given flow in both upstream and downstream directions. For instance, an intrusion detection system (IDS) is typically path-symmetric, because it needs to analyze packets in both directions of a given flow. A traffic shaper can be considered path-asymmetric if it must limit only outbound traffic.

In order to implement the aforementioned abstractions, we define a service function chaining template adopting the well-known JSON format. This template should be coupled with other deployment templates defined by the ETSI MANO specifications in order to complete service provisioning. However, in this work we focus only on the service function chaining aspects of the NBI. A service chain is therefore defined as follows:

```
{
  "src": "node_value",
  "dst": "node_value",
  "qos": "qos_type",
  "qos-thr": "qos_value",
  "vnfList": [vnf],
  "dupList": [dup]
}
```

where: `src` and `dst` represent the endpoint nodes of the service chain, either global or limited to a given VIM domain; `node_value` is a text string that contains a high-level unique identifier of a node known to both orchestrator and VIMs, e.g. by means of some form of mapping mechanism as defined in[12]; `qos` represents the QoS feature to be provided with the service chain; `qos_type` is a text string that contains a high-level unique identifier of a QoS metric known to both orchestrator and VIM; `qos-thr` represents the QoS threshold to be applied to the specified metric; `qos_value` is the actual value assigned to the threshold; `vnfList` is the ordered list of VNFs to be traversed according to the specified service; `dupList` is the list of VNFs towards which the traffic flow must be duplicated; each VNF included in `dupList` must be also included in `vnfList` to specify at which stage of the SFC the traffic must be mirrored.

Each VNF is described in terms of its topological abstractions with the following template:

```
vnf ::= {
  "name": "node_value",
  "terminal": "bool_value",
  "port_sym": "bool_value",
  "path_sym": "bool_value"
} | ϵ
```

where `bool_value` is a text string representing either a Boolean or a null value, and the ϵ symbol indicates the possibility that `vnf` is an empty element. Considering that some network functions (e.g., DPI, IDS) require traffic flows to be mirrored, the (possibly empty) list of VNFs towards which the traffic flow must be duplicated is specified with the following template:

```
dup ::= {"name": "node_value"} | ϵ
```

The NBI offered by VIMs can be implemented through the mechanisms of a REST API, and should provide the following methods:

- define a new service chain;
- update an existing service chain;
- delete an existing service chain.

These actions are basically in line with the operations foreseen by the ETSI MANO specifications with reference to the interface between NFVO and VIM. It is worth highlighting that the NBI description given above is indeed based on the concept of intent according to IETF and ONF definitions[9],[12]. QoS metric, VNFs and service chains are specified in a high-level, goal-oriented format, without the need to enumerate specific events, conditions, and actions, and without any knowledge of the technology-specific details. A non-intent-based description of a service chain, e.g. using the OpenFlow expressiveness to steer traffic flows and compose the network forwarding path, would require the customer to specify low-level policies to install multiple flow rules in each forwarding device for each traffic direction, involving technology-dependent details such as IP and MAC addresses, device identifiers and port numbers.

The NBI defined above is used, according to the architecture in Fig. 1, to specify an IoT data gathering service crossing two different SDN domains and an NFV chain, as well as to specify the characteristics that the transport service from the IoT domain to the cloud domain should provide. For the use case considered here, the high-level QoS features offered by the SDN/NFV platform include “delay-sensitive” and “loss-sensitive” services, with the possibility to specify a threshold for the relevant metric. Such quantitative QoS objectives are not as easy to obtain in the transport network, where most of the infrastructure may not be under the control of the final user, therefore it is assumed that the QoS objectives are given in a more qualitative way, with the same syntax as above but without specifying any quantitative threshold. This means that the service expects the transport domain to do “its best” with reference to that particular QoS aspect.

Although the above intent-based NBI definition is common to all VIMs considered in our use case, the orchestrator must specify different content for each VIM depending on the specific resources to be programmed and the specific segment of the service chain to be deployed in each domain. This approach allows the definition of the NBI to be more flexible, facilitating new technological extensions and new domains integration.

5 | IOT SDN DOMAIN

The IoT SDN domain included in the architecture of Fig. 1 is composed of: i) a VIM able to manage components and resources in the IoT domain; ii) an *IoT SDN controller* (IoTC), implementing the software-defined control plane of the IoT domain; iii) a set of IoT networks, where different devices send the measured data via multi-hop paths to a coordinator node that forwards them to the final consumer. Since the different IoT networks will possibly use different technologies (e.g., Zigbee, LoraWAN, 6LowPAN, etc.), each IoT coordinator will be connected to a specific gateway (GW) in charge of forwarding data outside the IoT domain.

When a service request is received from the high-level management and orchestration functions, the IoT VIM accesses the IoTC, including a database that stores information about devices of the different networks, such as the IP address of the corresponding GW, the service provided, and the related QoS feature that could be guaranteed. The VIM tries to map the incoming request with the resource knowledge available in the database, in order to select the proper IoT device to forward the request to (the details of this operation are presented in subsection 5.1). According to the decision taken, the IoTC will: i) program the selected IoT network to make sure that the requested QoS would be guaranteed; and ii) forward the request to the identified GW. More details about the different components are provided in the rest of this section.

5.1 | The IoT VIM and database

The VIM is capable of handling requests containing either the particular IoT device to be queried, or a high-level description of the service requested by the customer, together with some other possible specification related to the QoS (in terms of maximum latency). Let us consider the case of a customer that wishes to periodically collect temperature values in a given room and monitor them by means of a processing/publishing service called `ServP` running as a virtual function in the cloud domain. Assume that the customer is interested in having a sort of real-time monitoring of the measured temperature, thus requiring a delay-sensitive service. Then the intent-based request sent to the IoT VIM, expressed according to the JSON format specified in Section 4, could be as follows:

```
{
  "src": "ServP",
  "dst": "Temperature Room X",
  "qos": "Delay Sensitive",
  "qos-thr": "15 ms",
  "vnfList": "null",
  "dupList": "null"
}
```

In the IoT domain, following the typical IoT device query approach, `src` represents the source of the query, that is the final consumer of the data to be collected. In our example, this is the processing/publishing service in the cloud. `dst` represents the final endpoint of the query, that could be one or multiple IoT devices. This text string may contain i) a unique identifier of a specific IoT device, or ii) a high-level intent-based description of the requested service. The second option is used in our example above. `qos` represents the requested QoS feature either in terms of maximum latency, expressed as data plane round-trip time (see below), or minimum loss, expressed as the probability of successfully receiving the data from that device. If needed, the user may also provide a quantitative threshold `qos-thr`. In the example above, a delay-sensitive service with a 15 ms threshold is requested. Finally, `vnfList` and `dupList` are not specified in the example because we assume that the orchestrator opted for VNFs located only in the cloud domain.

At this point the VIM sends a query to the database module (located in the controller in our implementation): if the VIM is searching for a specific node, the controller notifies back to the VIM the presence or not of the node; if a service is requested instead, the controller replies with the list of nodes which can provide that service and the related QoS parameters. Finally, the VIM determines which nodes comply with the QoS requirement, and sends queries to those nodes (again over the controller) by specifying their ID. The latter is possible thanks to the information contained in the database, where an entry per IoT device is generated and each entry includes:

- The unique MAC address (e.g., the IEEE 802.15.4 64-bit address);
- The corresponding network address (i.e., the short address used in IEEE 802.15.4 at 16-bit);
- The ID of the IoT network the device belongs to;
- The service provided by the device (e.g., temperature sensor, light sensor, etc.);
- The value and timestamp of the last measurement gathered from the device;
- The corresponding QoS in terms of latency: these values are computed by averaging among different measurements taken over time.

When the IoTC receives a new measurement from a device, the data is stored in the database, together with the instant in which it was received. Once a new request for the same device arrives the VIM checks the timestamp and decides if the data should be updated or not (if not the value is immediately returned). With reference to the QoS, it is important to underline that in case the same device could reach the IoT coordinator via different paths (e.g., having different number of hops), the corresponding QoS values are stored in the database. A simplified example is reported in Table 1, where we are considering a room having two carbon monoxide sensors detecting the presence of smoke (devices 1 and 2) and a light sensor (device 3). Device 1 can reach the coordinator via three different paths, characterized by 1, 2 or 3 hops, and different resulting QoS values, namely round-trip time RTT_i in the case of i hops. Device 2 has two possible paths, whereas device 3 has only one path. If a user asks for the level of CO

TABLE 1 Example of QoS values stored into the IoT database.

Node ID	Service	RTT_1	RTT_2	RTT_3
1	Smoke Detector Room X	12 ms	24 ms	36 ms
2	Smoke Detector Room X	null	23 ms	38 ms
3	Light Room X	null	null	34 ms

in room X and wants the data in real time (delay-sensitive service), with a maximum latency (`qos-thr`) of 15 ms, the VIM will select device 1 and will notify to the IoTC the topology to be used to trigger such node in order to guarantee the requested QoS feature. However, if the delay requirement is relaxed the IoTC may decide for a longer path, possibly characterized by a lower loss probability (loss-sensitive service)[27]. Once the IoTC receives the request from the VIM, it will program the IoT network according to the selected topology.

5.2 | The IoT controller and network

The main functionalities of the IoT controller are: gathering information from devices, maintaining a representation of the network and establishing routing paths.

In order to achieve the decoupling of the control plane from the data plane, it is fundamental that each device can discover a path toward the coordinator. This is done during the network initialization, where the coordinator sends an *Hello* packet in broadcast; this packet is forwarded by nodes, after updating the number of hops (i.e., number of hops separating the node from the coordinator). In this phase each node selects the best next hop to be used to reach the coordinator, that is the one characterized by the lowest number of hops. Since *Hello* packets are sent in broadcast, they are also used to create neighbours tables, containing the RSSI (received signal strength indicator) received from each neighbour. These neighbours tables are periodically sent to the coordinator using the best next hop selected in the initial phase, and then forwarded to the IoT controller. In this way, the controller will update the database with the current map of devices, and will compute paths based on the RSSI matrix (power received by each node when another is transmitting). In our implementation, we assign to each link a cost having an inverse proportionality w.r.t. the RSSI and then, by running Dijkstra, we select for each node the path characterized by the lowest path cost (sum of the costs of the links in the path). Moreover, to limit delays, we also impose a maximum number of hops, denoted as H; by changing H different topologies are obtained, corresponding to possibly various performance levels, as shown in Table 1.

Requests coming from the VIM are forwarded by the IoTC to the proper IoT coordinator, together with the information about the selected path connecting the coordinator and the intended device to be setup to guarantee the requested QoS. This path is then forwarded by the coordinator to all devices belonging to the route itself (through the transmission of a packet called *Path*), in order to update the flow tables at devices. In case a device receives a packet for handling which it has no information, a *PathRequest* packet is sent through the route defined in the initial phase to the controller, that after elaborating it, will reply sending a *PathResponse*.

6 | DATA CENTER SDN AND CLOUD DOMAINS

In this section we consider both the data center SDN domain and the cloud computing domain depicted in Fig. 1, assuming that they are managed by a single VIM. The data plane topology assumed for the use case considered in this paper is shown in Fig. 2. An OpenFlow-based SDN data center infrastructure is assumed to be in charge of the connectivity within the cloud domain, thus providing programmable traffic steering functionality to implement suitable SFCs. All the switches included in the topology (`s1`, `s2`, …, `s7`) are OpenFlow-enabled devices and are governed by an SDN controller (e.g., ONOS[37]), whereas the computing infrastructure is managed through a cloud platform (e.g., OpenStack[38]).

Switch `s6` is an edge device that represents the ingress point to the cloud network. Incoming traffic flows, carried by means of suitable tunnels in the transport network, will be terminated here. Router `vrl` is the (virtual) edge router of the (virtual) tenant network responsible for the connectivity within the cloud domain of the requested IoT data collection service. Switches `s1` to `s5` are either physical or virtual switches used by the tenant network for VNF connectivity. Two VNFs are deployed in the cloud: `chk` performs integrity and sanity check on the collected data for improved reliability, whereas `bck` is used to store backup copies of the collected data. Router `vrr` is the (virtual) edge router of the (possibly different) tenant responsible for the

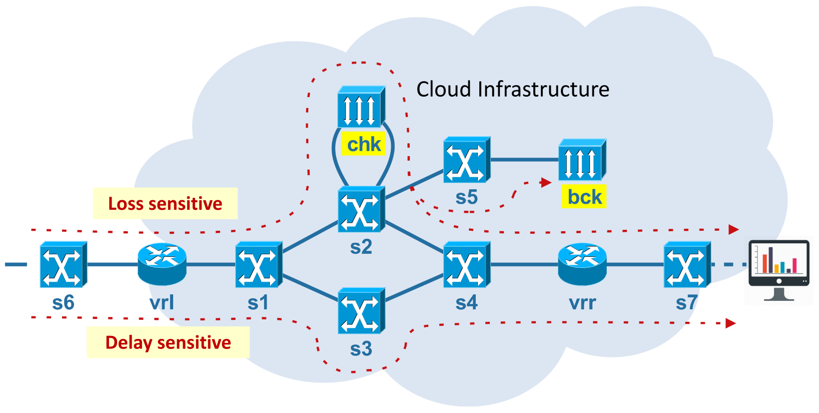

FIGURE 2 Data plane topology of the data center SDN and cloud domains considered in the use case.

IoT data collection, processing, and publishing services. Switch `s7` is a (virtual) switch in the latter tenant's network, providing layer-2 connectivity to the server `ServP` where collected data are processed and published.

According to the QoS features of the use case considered here, we assume that the connectivity service offers two different paths in the OpenFlow domain. One path is characterized by minimum latency, where switches have been configured with small buffers to limit queuing delay. Those switches are continuously monitored by the SDN controller to detect traffic levels that can lead to possible congestion. In addition, no VNF processing is performed along this path, which could introduce additional delay. This path is more suitable for delay-sensitive flows. The second path is dedicated to loss-sensitive traffic flows, where switches have large buffers to reduce losses, and data are processed by `chk` and duplicated at switch `s2` in order to be stored in `bck`.

Therefore, depending on the QoS feature requested by the customer, the high-level management and orchestration functions can specify two different service chains. Assuming that, based on the interaction between the orchestrator and the IoT VIM, incoming data will be collected from IoT network *k* and then forwarded to `ServP`, according to the JSON format specified in Section 4 the intent-based request to the cloud VIM NBI could be

```
{
  "src": "IoT-GW[k]",
  "dst": "ServP",
  "qos": "Delay Sensitive",
  "qos-thr": "10 ms",
  "vnfList": "null",
  "dupList": "null"
}
```

for the delay-sensitive QoS feature, or

```
{
  "src": "IoT-GW[k]",
  "dst": "ServP",
  "qos": "Loss Sensitive",
  "qos-thr": "99.999%",
  "vnfList": [chk, bck]
  "dupList": [bck]
}

chk ::= {
  "name": "chk",
  "terminal": "false",
  "port_sym": "true",
  "path_sym": "false"
}

bck ::= {
  "name": "bck",
  "terminal": "true",
  "port_sym": "null",
  "path_sym": "false"
}
```

for the loss-sensitive QoS feature. The SDN controller must implement a data plane monitoring service to make sure that, in the former case, the minimum latency path guarantees the requested maximum delay of 10 ms, whereas in the latter case the VNFs inserted in the service chain and the more reliable path ensure the required 99.999% availability.

We developed the VIM for the data center and cloud domains as an application running on top of the ONOS platform. It is worth to note that ONOS already provides a built-in, intent-based NBI that can be used to program the SDN domain and deploy the required network forwarding paths. However, in order to specify the ONOS intents, some knowledge is required of the specific data-plane technical details, while in our approach we prefer to expose only high-level abstractions to the orchestrator. Therefore, one of the main functions of our VIM is to implement new, more general and abstract intents that can be expressed according to the NBI specification given above. Then the VIM takes advantage of the network topology features offered by the SDN/cloud controllers to discover VNF location in the cloud and relevant connectivity details, and eventually it is able to compose native ONOS intents and build more complex network forwarding paths.

The VIM can be instantiated as an ONOS service called *ChainService*, which provides the capability of dynamically handling the VNF chains through the abstract NBI defined in Section 4. To achieve extensibility and modularity, the implementation of ChainService is delegated to a module called *ChainManager*, which is in charge of executing all the required steps to translate the high-level service specifications into ONOS-native intents. The input to ChainManager can be given through either the ONOS command line interface (CLI) or a REST API. The latter is preferable because it allows remote applications to use standard protocols (e.g., HTTP) to access resources and configure services. In our current implementation, the REST API provides the following service endpoints[§]:

```
POST /chaining/{action}/{direction}
DELETE /chaining/flush
```

In the former endpoint, the `action` variable indicates the operation that the orchestrator intends to perform on a specified service chain (`add`, `update`, or `delete`), whereas in case of an update the `direction` variable (`forth`, `back`, or `both`) defines whether the modified chain specification refers to the existing forwarding path from `src` to `dst`, the opposite way, or both directions. Parameters and identifier of the specified service chain are included (in JSON format) in the message body of the POST request method. So the basic operations of this endpoint are as follows:

- If the `add` action is given, this will result in defining a new service chain, based on the JSON specification included in the message body. This means that a forwarding path will be created for traffic flowing from `src` to `dst` and another one in the opposite direction. Note that the two paths are not necessarily symmetric, based on the topological abstractions defined by the NBI.

- If the `update` action is given, then the direction is taken into account and the forward path, backward path, or both paths of the specified existing service chain are changed. In fact, a user may be interested in changing only a segment of the forwarding path and only in one direction, to reduce the control plane latency and limiting the impact that a path change can have on the existing traffic flows.

- If the `delete` action is given, then both forwarding paths of the specified existing service chain are removed.

ChainService provides also the `flush` operation through another endpoint, thus offering the possibility of deleting in a single step the forwarding paths of all the service chains previously created.

[§]This is a first implementation of the API. In future versions we will consider to modify it according to a more accurate design following the REST principles, e.g., by using different HTTP methods to perform different actions and by specifying only resources in the URIs.

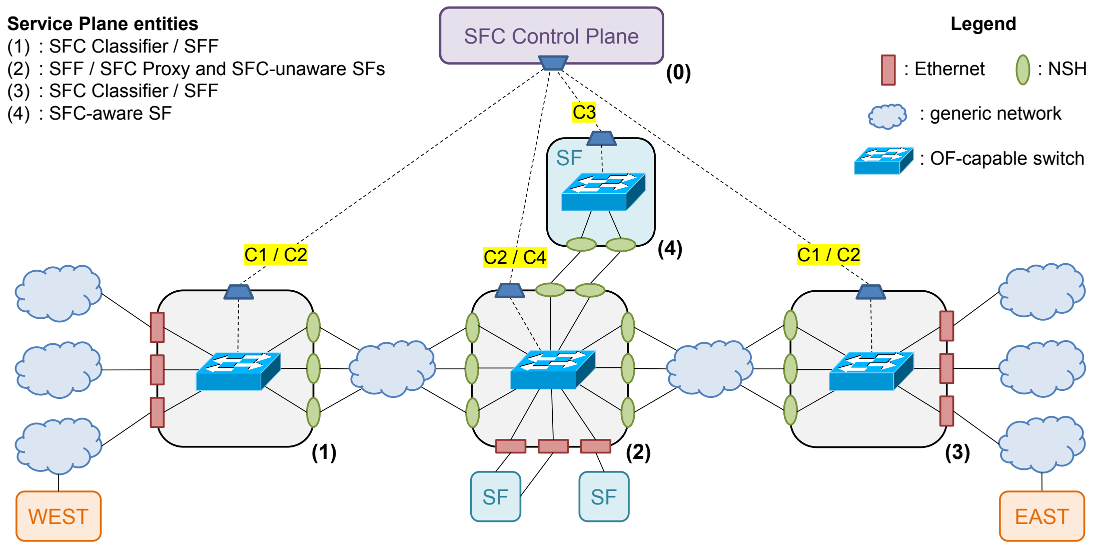

FIGURE 3 Reference NSH-based transport architecture: the role of Nodes (1) to (4) is shown in the upper left corner.

7 | THE TRANSPORT NETWORK DOMAIN

The role of the transport domain depicted in Fig. 1 is to provide inter-domain connectivity between IoT and data center/cloud domains across a general geographical network, as required by the service chain to be instantiated. Although the SDN concept has been recently extended to inter-data center transport networks[39] and to flexible wide area network (WAN) interconnections[40],[41], in our study we choose to be independent of the control capabilities offered by the transport network. As previously mentioned, an overlay approach allows to deal with heterogeneous forwarding technologies in the transport domain.

In particular, in order to keep service provisioning operations separate from and independent of the underlying transport infrastructure, we adopt the IETF SFC approach and take advantage of NSH encapsulation to properly steer traffic flows between the different domains involved in a given SFC[24]. In fact, when used in conjunction with some tunneling technoligy (e.g., VXLAN), NSH can be seen as a new way to implement a network overlay enabling service function chaining on top of legacy transport networks. As detailed in this section, our SDN-like solution for implementing the NSH control plane enables a seamless integration of the transport infrastructure manager's NBI with the NBI adopted in the IoT and data center/cloud domains, as well as the ability to dynamically adapt traffic flow forwarding to the requirements of the SFC being deployed.

7.1 | The IETF Service Function Chaining Architecture

The IETF SFC architecture for a given transport domain defines:

- *Service Function Path* (SFP): a specification of the path to be followed by packets assigned to a certain SFC, i.e. an abstraction of the sequence of nodes the packets will traverse;
- *SFC encapsulation* (SFC-En): a form of SFP identification that enables to follow the correct sequence of nodes in the SFC.

Moreover, the main components of the SFC architecture are:

- *SFC Classifiers* (SFC-Cl), which classify the incoming traffic based on predefined policies, in order for the flow to be steered through the required set of network service functions; the main task for the SFC-Cl is to add the SFC-En, which is then removed by the last node in the SFP, or by a SFC-aware function that consumes the packet;
- *Service Functions* (SF), which are the basic elements of a chain, and are responsible for a specific treatment of received packets; they can act at different levels of the protocol stack, and they can be implemented either as virtual elements hosted by a server, or as physical equipment with specialized hardware; a SF can be either SFC-aware (i.e., able to act on SFC-encapsulated packets) or SFC-unaware (i.e., it must receive only packets without SFC encapsulation);

- *Service Function Forwarders* (SFF), which are responsible for forwarding traffic to one or more connected SFs according to information carried in the SFC-En; they can also terminate the SFP;
- *SFC Proxies* (SFC-Pr), which remove and insert SFC-En on behalf of SFC-unaware SFs, before and after their action, respectively.

The implementation of the architecture requires a protocol to provide SFP identification, transport-independent chaining, and packet-based network and service metadata. The NSH protocol is designed to this purpose, with the goal to be easy to implement across a range of devices, both physical and virtual, including hardware platforms. The two most important fields in the NSH header are:

- *Service Path Identifier* (SPI): a 24-bit integer number assigned to packets by the first SFC-Cl in the SFP; all nodes taking part in that SFP must use the same SPI consistently;
- *Service Index* (SI): an 8-bit integer number, used to identify the current position within the SFP; it is set to its maximum value (i.e., 255) or to a value related to the length of the SFP, and is decremented of one unit by all SFC-aware SFs and SFC Proxies the packet traverses in the SFP.

7.2 | OpenFlow-based NSH Control Plane

As an example, the SFC architecture with all its building components is plotted in Fig. 3. It is composed of a SFC control plane entity, a pair of SFC-Cls, an intermediate node serving as both SFF and SFC-Pr towards SFC-unaware SFs, a SFC-aware SF, and two SFC-unaware SFs.

According to this proposal, the transport network can be controller by one or multiple network operators through a generic control plane paradigm, either SDN or non-SDN. As a matter of fact, *service providers and network providers can act as completely independent entities, each adopting its favorite control plane approach*. Nonetheless, the IETF does not specify how the architecture should be implemented. In[26] it was proposed to build it around an OpenFlow-capable switch (OF-S) as follows:

- SFC entities are interconnected by means of a tunneling technology (e.g., VXLAN) through an underlying network infrastructure;
- the SFP (i.e., a SPI/SI pair) is mapped into the ports of the employed OF-S, thus creating a SFP-to-transport mapping;
- such mapping is combined with the tunnels to deploy the SFC in the real transport network.

In summary, each NSH interface, corresponding to a specific SPI/SI pair, is bridged to a port on the node's internal OF-S and, by means of the association of SPI/SI pairs to ports on a OF-S, it is possible to have the node acting as a NSH Service Plane component while controlling it through the OpenFlow protocol from an SDN Controller, which takes the role of SFC Control Plane entity (SFC-Co) running applications that enforce Service Plane policies. The NSH mapping tables are therefore implemented in the form of flow tables inside the OF-S. As an example, assume port N of the OF-S is bridged to interface `nshM` of the node. Instructing the switch to send traffic out of port N will result in the node sending NSH-encapsulated traffic out of interface `nshM` with the corresponding SPI/SI values. Therefore, depending on what kind of flow rules are installed in the internal OF-S, a SFC node can be programmed to perform different Service Plane entity functions. With reference to Fig. 3, the entities are mapped to the nodes in the following way:

- Node (0) hosts the SFC-Co.
- Node (1) is responsible for adding the NSH tag to packets coming from the *WEST* domain and forwarding NSH-encapsulated packets to the first SFF in the SFP: in this role, it acts as SFC-Cl. Additionally, this node is also responsible for removing the NSH tag from packets assigned to a SFP which ends at Node (1), such as packets destined to the *WEST* domain, thus acting as SFF. Following this approach, the SFC classification is as expressive as OpenFlow matching is.
- Node (2) is responsible for handling the NSH encapsulation on behalf of SFC-unaware SFs, as well as for forwarding the NSH-encapsulated packets to the following SF or SFF in the SFP. In those two tasks, Node (2) acts as SFC-Pr and SFF, respectively.

- Node (3), similarly to Node 1, acts both as SFC-Cl and SFF for the traffic exchanged with the *EAST* domain.
- Node (4) acts as a SFC-aware SF, as it is able to receive NSH-encapsulated packets from the SFF and process them, before sending them back to the SFF after updating the SI.

7.3 | Transport VIM and NBI

According to the ETSI MANO specifications and architecture we can say that the transport network has its own VIM, more properly called WAN Infrastructure Manager (see Fig. 1). The WAN-IM gets service specifications from the NBI and talks to the SDN controller issuing flow rules to the OF-Ses inside the SFC nodes. Again according to the JSON format specified in Section 4, the intent-based request to the WAN-IM could have the form:

```
{
  "src": "IoT-Domain",
  "dst": "Cloud-Domain",
  "qos": "Delay Sensitive",
  "qos-thr": "null",
  "vnfList": "null",
  "dupList": "null"
}
```

For the transport domain it is not possible, in general, to specify the target value of a QoS parameter in absolute quantitative terms. The domain carries a variety of customers and the transport operator does not allow the final users to control the network behavior. Therefore, we assume that what can be done in this case is just to specify a “qualitative” objective, meaning that the `qos-thr` value is either unspecified or ignored. However, the qualitative QoS class specification is still useful to the WAN-IM to decide what is the best path to route the traffic flows and enforce that choice. In the example, `"qos": "Delay Sensitive"` means that the service request should be accepted by finding the path with the smallest latency possible, although a minimum latency value cannot be guaranteed. The transport network controller can enforce this decision by periodically monitoring path latency and possibly rerouting the flows if a different path with smaller latency becomes available. The requirement will then be honored only on an availability basis.

8 | EXPERIMENTAL VALIDATION

8.1 | Test bed setup

As a demonstration of the feasibility of the multi-domain service management solution proposed here, we developed a test bed to implement the reference architecture of the cloud-based IoT data collection service with quality differentiation illustrated in Fig. 1. The complete test bed setup, including the components discussed in Sections 5, 6 and 7, is shown in Fig. 4. The customer on the top-right corner requests the service to the high-level management and orchestration functions, specifying the desired QoS feature. The orchestrator then forwards the request to the VIM REST NBIs of the relevant domains using the JSON format described in the previous sections. Each VIM performs the operations required in the respective domain and programs the underlying controllers according to the requested service and QoS feature. Data generated by the IoT devices are sent by the relevant gateway via HTTP POST to the collecting/processing/publishing server in the cloud, where the customer can retrieve it (the case of delay-sensitive QoS feature is shown in the figure).

In the test bed, the data center SDN domain and the cloud domain were emulated using Mininet running in a virtual machine (VM)[42]. The data plane topology shown in Fig. 2 was built with a customized Mininet script specifying the required OpenFlow switches. Routers and VNFs were deployed as separated network namespaces in the same VM and connected to the virtual switches created by Mininet. Additional VMs were instantiated in the same physical server to deploy the data collection/processing server and the ONOS platform components needed by the SDN control plane. Those VMs were connected to the VM running Mininet through suitable virtual interfaces created inside the physical server. In order to provide the two paths with different latency, `chk` was configured to introduce an additional random delay uniformly distributed between 25 and 35 ms, with 25% correlation between consecutive samples.

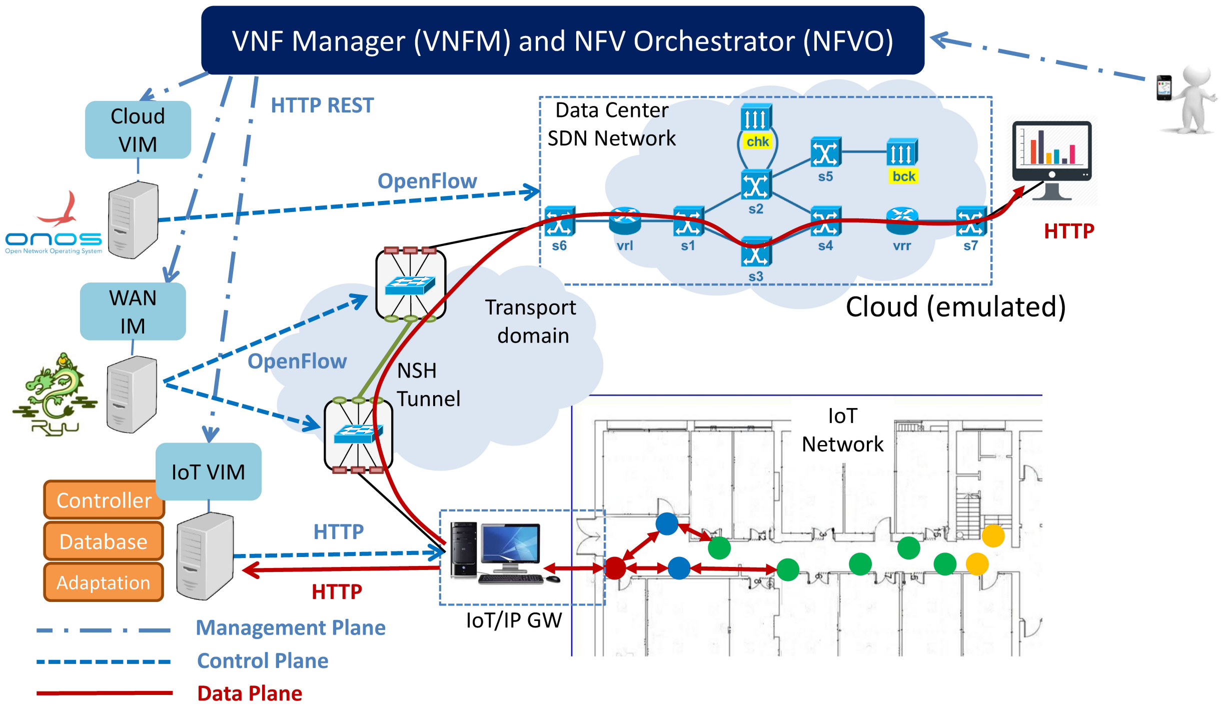

FIGURE 4 The NFV/SDN test bed setup developed to demonstrate end-to-end multi-domain service management. Information flow in the management and control planes is displayed with dashed lines. Information flow in the data plane is displayed with solid lines.

As far as the IoT domain is concerned, in our implementation we setup an IoT network using the "European Laboratory of Wireless Communications for the Future Internet" (EuWIn) platform and, in particular, the flexible topology test bed (Flextop) facility[43,44]. The lab is composed of TI CC2530 devices, compliant with IEEE 802.15.4[44], on top of which our SDN protocol stack is running. Nodes are located into boxes on the walls of a corridor at the University of Bologna. The map with the corresponding location of the nodes selected for the experiments (identified by colored circles) is shown in the bottom-right part of Fig. 4. In particular, in the figure we show an example of topology when setting $H = 3$ (i.e., maximum three hops to reach the coordinator): the red node is the coordinator, the blue nodes are connected via one hop, the green nodes via two hops, and the yellow nodes via three hops. In our experiments IoT data gathered by the gateway were then duplicated and sent to: 1) the IoTC and then to the IoT VIM; 2) to the transport domain, to be forwarded to the data center SDN domain and then to the cloud, from which the user could read the measured data.

Finally, the transport domain in our test bed was implemented on a legacy physical network, on top of which we enabled NSH encapsulation and VXLAN tunneling between pairs of NSH-capable nodes. The NSH endpoints serve as SFC-Cl's, as introduced in Section 7.1. An instance of the Ryu network controller[45] implements the SDN controller responsible for steering the traffic in the transport domain. It does so by means of NSH encapsulation and dynamic SPI/SI allocation.

In order to validate the adaptive traffic steering capabilities of the NSH-based transport domain, three NSH endpoints were deployed as ingress/egress nodes exchanging traffic with other domains. One endpoint was connected to the IoT domain gateway located at the University of Bologna facilities. A second endpoint was connected to the VMs previously mentioned, where the data center SDN domain and the cloud domain were emulated with Mininet. Those VMs were deployed on a physical server located in a research-oriented computing facility in Belgium. This setup was used in a first set of experiments, where we were interested in validating the intent-based service management capabilities of the IoT, data center and cloud domains involved in our reference architecture scenario.

A second set of experiments were run after instantiating the data collection/processing server also in a virtual machine located at the University of Bologna and connected to the third NSH endpoint. The latter setup was used to emulate the scenario where the required service is discovered in an edge or fog computing domain located closer to the IoT domain with respect to the remote cloud domain. In this case, the edge/fog node offering the service may not be continuously available, due to the limited and variable (e.g., due to mobility) number of resources available in such kind of computing environments. However, when the required resources can be found in a local edge/fog domain, it is preferable to take advantage of them so that a delay-sensitive service can be delivered with a reduced data plane latency, resulting also in a reduced traffic load in the transport network. The adaptive traffic steering capabilities of the NSH-based transport domain allow to dynamically change the end-to-end service deployment from the cloud-based scenario to the edge/fog-based one, as sketched in Figs. 5 and 6.

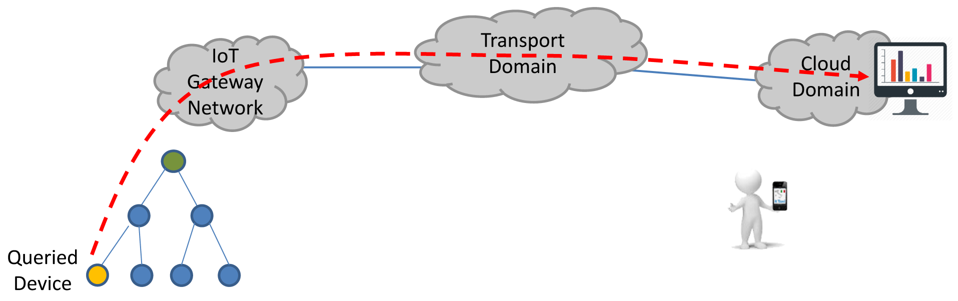

FIGURE 5 End-to-end service deployment across the IoT, transport and data center/cloud domains.

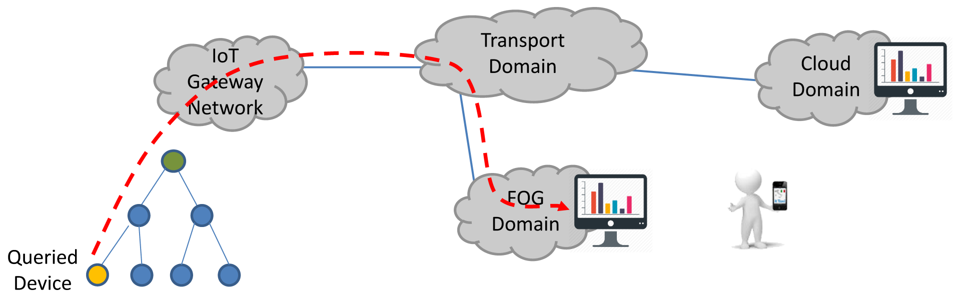

FIGURE 6 End-to-end service deployment when the requires resources are available in a fog domain located closer to the user or IoT domain.

In the following subsections, we first report the validation of the proposed intent-based service management approach in each of the different technological domains we included in our test bed. Then, we present the validation of the actual end-to-end service deployment.

8.2 | IoT domain validation

We considered an application where the user asks for the data measured by an IoT device with a given QoS, and waits for the reply. In the IoT network, both the request and the reply data frames have a payload of 10 bytes, and queries were generated by the user every second. For each query we measured: i) the RTT at the IoT data plane, that is the interval of time between the reception of the query coming from the IoTC at the application layer of the IoT coordinator, and the reception of the reply from the target node, again at the application layer of the coordinator; ii) the RTT at the control plane, that is the interval of time between the reception of the query coming from the VIM at the IoTC, and the reception of the reply coming from the intended GW, again at the IoTC; iii) the RTT measured taking the time stamp at the IoT VIM. Moreover, to better validate the control plane, we also computed the controller processing time (CPT), defined as the time instant between the reception at the controller of the query from the VIM and the instant in which the query is forwarded to the gateway (this interval includes the time needed for paths computation).

Performance were evaluated by averaging over 10,000 queries generated by the user toward a node in case of a delay-sensitive service (i.e., we set $H = 1$, and the node was at 1 hop from the coordinator), and a loss-sensitive service (i.e., we set $H = 3$ and the node was at 3 hops from the coordinator). Results are shown in Table 2. The average RTT at the IoT data plane is mainly

TABLE 2 Average RTT at the IoT data plane (DP), control plane (CP) and VIM, and CP processing time for different QoS requirements.

QoS	Hops	RTT at DP	RTT at CP	RTT at VIM	CPT
Delay sensitive	1 hop	12.6 ms	516.7 ms	522.2 ms	239.7 ms
Loss sensitive	3 hops	40.4 ms	545.7 ms	550.5 ms	253.1 ms

TABLE 3 Average and standard deviation of data plane (DP) one-way latency computed at the emulated cloud network.

QoS feature	Average latency	Stdev
Delay sensitive	0.3 ms	0.28 ms
Loss sensitive	31.7 ms	2.41 ms

influenced by the number of hops, giving latency values in the order of tens of milliseconds that can be considered acceptable depending on the required QoS. As for the RTT measured at the control plane and VIM, results demonstrate that the most significant contribution depends on the IoTC response time. To better quantify the latter, the CPT is shown in the last column of the table. As can be noted, half of the RTT measured at the control plane is the time needed by the controller to process the query and generate paths. The remaining delay is due to the communication within the IoT network, for data transfer (i.e., data plane RTT), plus the time needed to install the paths in the IoT network. This also validates the correct behavior of the IoT control plane from the functional point of view.

It is important to underline that paths in the IoT network were refreshed periodically and not at every query received. In particular, in our implementation we sent one *Path* packet per device to be queried (all the nine nodes switched on in this case) every 250 s. As a result, the control plane RTT is not constant, but presents some peaks when a new *Path* packet is generated. We measured the standard deviation of such control plane RTT that was equal to 280 ms (considering all measurements taken).

8.3 | Cloud domain validation

We measured the performance within the emulated data center and cloud network when the customer requests the service specifying two traffic classes, according to the QoS features offered by the data center SDN domain: delay sensitive and loss sensitive. In this case, one-way latency in the emulated cloud network was measured by comparing timestamps of each packet captured at switches `s6` and `s7`. The capture was performed in the server hosting the Mininet virtual machine, so the same reference clock was used for the sake of accuracy. The measurements were made by averaging over 10,000 requests.

Results are reported in Table 3 in terms of average and standard deviation of the data plane one-way latency. The numbers show the correct behavior of the data center SDN domain with respect to the requested QoS feature: very limited latency was measured in the delay-sensitive case, whereas in the loss-sensitive case no packet was lost and `bck` successfully stored a copy of the entire data set transmitted by the IoT GW.

We also measured the NBI response time at the VIM implemented in ONOS, i.e. the time required by the VIM to process a JSON service chain specification and suitably program the SDN controller. To assess the scalability of the NBI, we generated an increasing number of requests (from 5 to 200) sent in a batch to the VIM. Each measured response time was obtained as an average over 20 runs with the same number of requests. Figure 7 shows the average NBI response time with 95% confidence intervals. The numbers show that the VIM is very responsive, in the order of tens of milliseconds. The setup of loss-sensitive service chains takes slightly longer than the delay-sensitive ones because of the relatively more complex service chain to be processed.

As already discussed in section 6, the VIM for the data center and cloud domains was developed as an application running on top of the ONOS platform and taking advantage of its connectivity-oriented, intent-based NBI. This means that the operations performed by the VIM (i.e., parsing and processing a request received through its service-oriented, intent-based NBI; connecting to the ONOS NBI; programming the relevant intents) are decoupled from the ONOS-based operations (i.e., installing the requested intents in its core modules and translating them into actual OpenFlow rules to be added to the controlled SDN

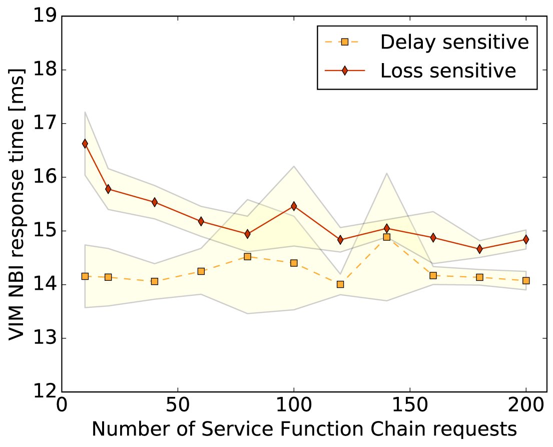

FIGURE 7 Average NBI response time and 95% confidence interval at the SDN/cloud VIM with increasing number of service chain requests.

TABLE 4 Average response time of the ONOS controller to execute the intent and flow installation in the data center SDN network.

No. of vCPUs	Delay sensitive	Loss sensitive
2	3321.4 ms	3468.9 ms
4	2071.7 ms	2984.7 ms
8	1617.9 ms	2866.6 ms

switches). Therefore, the response time reported in Fig. 7 does not include the time needed by ONOS to complete the flow rule setup. Since the latter depends on the specific SDN control technology adopted, we decided to keep it separate from the VIM response time.

However, for the sake of completeness, we report in Table 4 the time needed by ONOS to execute the intent and flow installation for the two QoS classes under different virtual machine resource configurations in terms of number of CPUs. The results, obtained from the average over 100 SFC requests, show how the ONOS response time decreases when more resources are dedicated to it, keeping the network programming time in the order of a couple of seconds. This also proves the correct behavior of the data center and cloud domain control plane from the functional point of view. A complete functional validation of the proposed NBI and the underlying control plane was performed on a very similar experimental environment[46].

8.4 | Transport domain validation

In the transport domain we measured the latency of the data plane between the NSH endpoint connected to the IoT domain and the NSH endpoint connected to the domain where the data "consumer" is located. To functionally validate the adaptive traffic steering capabilities of the SDN control plane adopted for the NSH-based overlay, we started with a delay-sensitive service located in the remote cloud and assumed that at some point suitable resources were discovered[¶] in a fog domain located closer to the IoT domain. We used ping-based periodic RTT measurements between each pair of NSH endpoints (IoT-to-cloud and IoT-to-fog) to choose the domain with the minimum data plane latency. In order to stabilize the RTT measurements, we evaluated the exponential weighted moving average (EWMA) of the collected RTT samples with weight $\alpha = 0.5$. Although a single-way latency measurement may be more significant, in our setup it was impossible to accurately assess it, as the source and destination NSH endpoints resided in different and remote physical machines, with non-synchronized clock sources.

[¶]We did not implement a fully fledged resource discovery mechanism, as this is out of the scope of this paper. We rely on ping response to detect when the virtual machine, representing the resource located at the edge/fog domain, becomes alive and add a 20 s interval to emulate the resource discovery phase. We consider this very simple, and definitely incomplete, resource discovery mechanism sufficient to demonstrate the correct behavior of the traffic steering in the NSH-based transport domain.

As a realistic estimation of the response time in the transport network data plane, we measured the time needed to complete a series of HTTP POST requests from endpoint to endpoint, taking into account TCP session setup, HTTP POST message request, and 200 OK response. The POST messages were generated and sent by the node serving as NSH endpoint connected to the IoT domain, and were received and acknowledged by the node serving as the NSH endpoint connected to either the cloud or the fog domain. We generated 100 POST requests, sending them one per second.

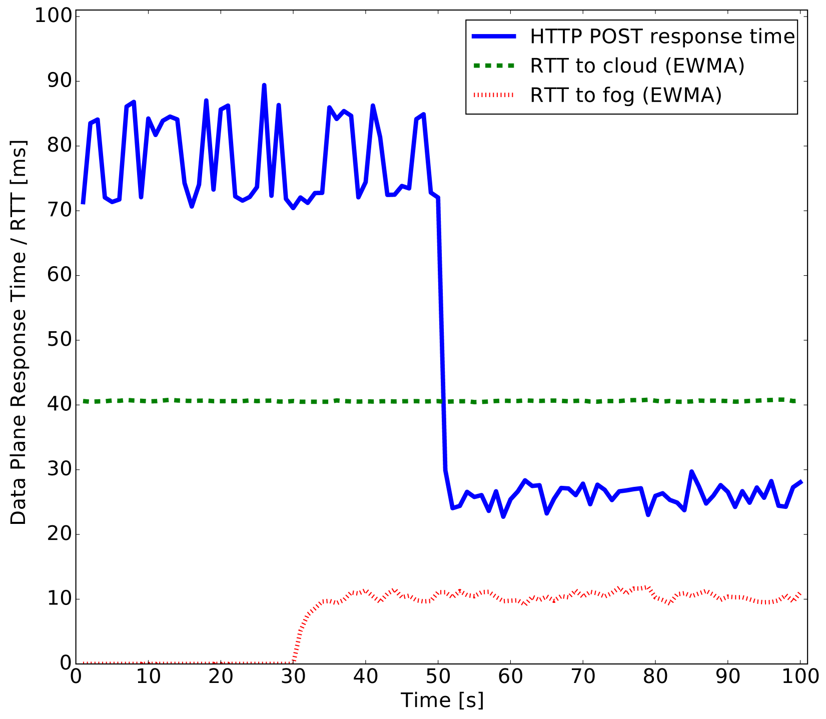

FIGURE 8 Temporal evolution of the transport data plane response time for HTTP POST requests and corresponding measured RTT values (EWMA with weight $\alpha = 0.5$).

In Fig. 8, the temporal evolution of the transport data plane response time for HTTP POST requests is represented by the solid line, while the network-level EWMA of the RTT is represented by the dashed line for the cloud domain and by the dotted line for the fog domain. At the beginning, the HTTP traffic is sent towards the server in the cloud domain, with a quite steady RTT moving average of about 40 ms. From $t = 0$ s to $t = 50$ s, the traffic actually reaches the cloud domain (located in Belgium), and the fluctuations in the measured response time are mainly caused by application-level delays. Meanwhile, at $t = 30$ s the periodic ping measurement detects that the fog node has become available, with a RTT moving average of about 10 ms, significantly lower than the RTT measured toward the cloud. After a resource/service discovery period, assumed to be completed at $t = 50$ s, the transport domain SDN controller steers the traffic coming from the IoT endpoint toward the fog domain, achieving overall better latency performances. This validates the correct behavior of the transport domain control plane from the functional point of view. The difference between the RTT values and HTTP POST response times is due to the additional overhead included in the HTTP POST transaction with respect to a simple ping packet.

8.5 | End-to-end service deployment validation

As a final validation, we measured the actual end-to-end service deployment time across the multi-domain scenarios in Figs. 5 and 6. For this analysis it is worth reminding that the service deployment response time is due to the response time of the management plane, consisting of the VIMs orchestrating the service implementation via the NBI, the delay in the network control plane, implemented by the SDN controllers, i.e. the IoTC/ONOS/Ryu platforms in this specific test bed, and the data plane latency required by the data traveling the network once the SFC is set up.

TABLE 5 Average end-to-end service deployment time, for different QoS features and cloud or fog domain scenarios.

QoS feature	Cloud scenario	Fog scenario
Delay sensitive	532.3 ms	511.8 ms
Loss sensitive	554.0 ms	530.1 ms

In this case we measured the time needed for the user's request containing the intent-based service specification to reach the VIMs in the different domains, the generation of data in the IoT domain, its transmission through the transport domain to the destination server in the cloud/fog domain, and the final acknowledgment. We did not include the time needed to actually program the network control plane that was already presented in Table 4.

The measured average values, computed over 100 samples and shown in Table 5, are about half a second, the most of it due to the service management plane (orchestration, intent-based request set and processing, etc.) with just about 10% related to the network data plane latency. Nonetheless, the reduced network latency is evident when the service is "re-routed" to the fog domain. This is very important because, for all the data posted after the service set-up, the network delay would be the only component (the time needed by the management plane being needed just at set-up) and therefore they would experience an improvement in response time of almost 100%.

9 | CONCLUSION

In this paper we proposed a reference architecture, inspired by the ETSI MANO framework, and an intent-based NBI for end-to-end service management across multiple technological domains. In particular, we considered the use case of a software-defined IoT infrastructure that "produces" relevant data that are processed and "consumed" at a set of cloud-based services. The IoT domain is connected to the cloud by a generic transport network. The IoT test bed, the transport network and the cloud infrastructure are SDN-enabled with specific and technology-dependent implementations of SDN controllers. An overarching orchestration service is also assumed that exploits abstractions to implement Service Function Chains that span across the domains with a unified northbound Interface based on the JSON syntax and service oriented abstractions.

The manuscript reports the validation results that demonstrate the feasibility of the approach and the potentials of the NBI applied in real environments over a heterogeneous OpenFlow/IoT SDN test bed. The latency values measured at both data and control/management planes allowed us to get a first insight to the performance levels of the overall system, resulting in reasonable response times for service setup and QoS requirement satisfaction. Scalability tests on the ONOS-based VIM also gave promising results. The use case reported here represents a working example of a more general approach to properly define high-level interfaces and develop the related control and management components to unify orchestration capabilities across multiple SDN/NFV domains.

As future directions, we intend to test performance when multiple IoT networks are managed by the same controller, implementing a load balancing strategy. Also the case of integrating IoT networks using different technologies, such as LoRa, will be investigated. We also plan to generalize the proposed intent-based NBI in order to encompass different service scenarios that may involve multiple domains, such as 5G network slicing or multi-access edge computing. Finally, we are also developing an original mathematical formulation of the intent mapping problem and an intent specification interpreter based on natural language.

ACKNOWLEDGMENT

This work was partially funded by: project "GAUChO - A Green Adaptive Fog Computing and Networking Architecture," funded by the Italian Ministry of Education, University and Research (MIUR) under the "PRIN 2015" program, grant no. 2015YPXH4W_004; the COST Action CA15104-IRACON; the Horizon 2020 grant no. 644090: "EuroCPS - European Network of competencies and platforms for Enabling SME from any sector building Innovative CPS products to sustain demand for European manufacturing."

References

1. Soares J, Goncalves C, Parreira B, et al. Toward a telco cloud environment for service functions. *IEEE Communications Magazine* 2015; 53(2): 98-106.

2. Peterson L, Al-Shabibi A, Anshutz T, et al. Central office re-architected as a data center. *IEEE Communications Magazine* 2016; 54(10): 96-101.

3. Mijumbi R, Serrat J, Gorricho JL, Bouten N, De Turck F, Boutaba R. Network Function Virtualization: State-of-the-Art and Research Challenges. *IEEE Communications Surveys Tutorials* 2016; 18(1): 236-262.

4. Hu F, Hao Q, Bao K. A Survey on Software-Defined Network and OpenFlow: From Concept to Implementation. *IEEE Communications Surveys Tutorials* 2014; 16(4): 2181-2206.

5. Callegati F, Cerroni W, Contoli C, Cardone R, Nocentini M, Manzalini A. SDN for dynamic NFV deployment. *IEEE Communications Magazine* 2016; 54(10): 89-95.

6. Rosa RV, Santos MAS, Rothenberg CE. MD2-NFV: The case for multi-domain distributed network functions virtualization. In: *2015 International Conference and Workshops on Networked Systems (NetSys)*; 2015: 1-5.

7. Phemius K, Bouet M, Leguay J. DISCO: Distributed multi-domain SDN controllers. In: *2014 IEEE Network Operations and Management Symposium (NOMS)*; 2014: 1-4.

8. Sonkoly B, Czentye J, Szabo R, et al. Multi-Domain Service Orchestration Over Networks and Clouds: A Unified Approach. In: *2015 ACM SIGCOMM Conference*; 2015: 377–378.

9. Clemm A, Ciavaglia L, Zambenedetti Granville L. Clarifying the Concepts of Intent and Policy. Internet-Draft draft-clemm-nmrg-dist-intent-01, Internet Engineering Task Force; : 2018. Work in Progress.

10. Cerroni W, Buratti C, Cerboni S, et al. Intent-based management and orchestration of heterogeneous OpenFlow/IoT SDN domains. In: *2017 IEEE Conference on Network Softwarization (NetSoft)*; 2017: 1-9.

11. Lenrow D. Intent: Don't Tell Me What to Do! (Tell Me What You Want). https://www.sdxcentral.com/articles/contributed/network-intent-summit-perspective-david-lenrow/2015/02/; 2015.

12. Intent NBI - Definition and Principles. https://www.opennetworking.org/sdn-resources/technical-library; 2016.

13. Cohen R, Barabash K, Rochwerger B, et al. An intent-based approach for network virtualization. In: *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM)*; 2013: 42-50.

14. Han Y, Li J, Hoang D, Yoo J, Hong JW. An intent-based network virtualization platform for SDN. In: *2016 12th International Conference on Network and Service Management (CNSM)*; 2016: 353-358.

15. Pham M, Hoang DB. SDN applications - The intent-based Northbound Interface realisation for extended applications. In: *2016 IEEE NetSoft Conference and Workshops (NetSoft)*; 2016: 372-377.

16. Kang J, Lee J, Nagendra V, Banerjee S. LMS: Label Management Service for intent-driven Cloud Management. In: *2017 IFIP/IEEE International Symposium on Integrated Network and Service Management (IM)*; 2017: 177-185.

17. Subramanya T, Riggio R, Rasheed T. Intent-based mobile backhauling for 5G networks. In: *2016 12th International Conference on Network and Service Management (CNSM)*; 2016: 348-352.

18. Arezoumand S, Dzeparoska K, Bannazadeh H, Leon-Garcia A. MD-IDN: Multi-domain intent-driven networking in software-defined infrastructures. In: *2017 13th International Conference on Network and Service Management (CNSM)*; 2017: 1-7.

19. OpenDayLight Group Based Policy User Guide. http://docs.opendaylight.org/en/stable-nitrogen/user-guide/group-based-policy-user-g
.

20. OpenDayLight Network Intent Composition (NIC) User Guide. http://docs.opendaylight.org/en/stable-nitrogen/user-guide/network-int
.

21. ONOS Intent Framework. https://wiki.onosproject.org/display/ONOS/Intent+Framework; .

22. Rubio-Loyola J, Serrat J, Charalambides M, Flegkas P, Pavlou G, Lafuente AL. Using linear temporal model checking for goal-oriented policy refinement frameworks. In: *Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'05)*; 2005: 181-190.

23. Halpern JM, Pignataro C. Service Function Chaining (SFC) Architecture. IETF RFC 7665; 2015

24. Quinn P, Elzur U, Pignataro C. Network Service Header (NSH). IETF RFC 8300; 2018

25. Boucadair M. Service Function Chaining (SFC) Control Plane Components and Requirements. Internet-Draft draft-ietf-sfc-control-plane-08, Internet Engineering Task Force; : 2016. Work in Progress.

26. Davoli G, Cerroni W, Contoli C, Foresta F, Callegati F. Implementation of service function chaining control plane through OpenFlow. In: *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*; 2017: 1-4.

27. Bera S, Misra S, Vasilakos AV. Software-Defined Networking for Internet of Things: A Survey. *IEEE Internet of Things Journal* 2017; 4(6): 1994-2008.

28. Luo T, Tan HP, Quek TQS. Sensor OpenFlow: Enabling Software-Defined Wireless Sensor Networks. *IEEE Communications Letters* 2012; 16(11): 1896-1899.

29. Zeng D, Miyazaki T, Guo S, Tsukahara T, Kitamichi J, Hayashi T. Evolution of Software-Defined Sensor Networks. In: *2013 IEEE Conference on Mobile Ad-hoc and Sensor Networks (MSN)*; 2013: 410-413.

30. Galluccio L, Milardo S, Morabito G, Palazzo S. SDN-WISE: Design, prototyping and experimentation of a stateful SDN solution for WIreless SEnsor networks. In: *2015 IEEE Conference on Computer Communications (INFOCOM)*; 2015: 513-521.

31. Baddeley M, Nejabati R, Oikonomou G, Sooriyabandara M, Simeonidou D. Evolving SDN for Low-Power IoT Networks. In: *2018 IEEE Conference on Network Softwarization (NetSoft)*; 2018: 71-79.

32. Oteafy SMA, Hassanein HS. Towards a global IoT: Resource re-utilization in WSNs. In: *2012 International Conference on Computing, Networking and Communications (ICNC)*; 2012: 617-622.

33. Bera S, Misra S, Roy SK, Obaidat MS. Soft-WSN: Software-Defined WSN Management System for IoT Applications. *IEEE Systems Journal* 2016. to be published.

34. Buratti C, Stajkic A, Gardasevic G, et al. Testing Protocols for the Internet of Things on the EuWIn Platform. *IEEE Internet of Things Journal* 2016; 3(1): 124-133.

35. Network Functions Virtualisation (NFV); Management and Orchestration. http://www.etsi.org/technologies-clusters/technologies/nfv; 2014.

36. The Frog 4.0 Project. https://github.com/netgroup-polito/frog4; .

37. ONOS: Open Network Operating System. http://onosproject.org; .

38. OpenStack: Open Source Cloud Computing Software. https://www.openstack.org; .

39. Jain S, Kumar A, Mandal S, et al. B4: Experience with a Globally-deployed Software Defined WAN. *SIGCOMM Comput. Commun. Rev.* 2013; 43(4): 3–14.

40. What is Software-Defined WAN (or SD-WAN or SDWAN)?. https://www.sdxcentral.com/sd-wan/definitions/software-defined-sdn-wa
.

41. Jain R, Khondoker R. Security Analysis of SDN WAN Applications–B4 and IWAN. In: Khondoker R., ed. *SDN and NFV Security*. Vol. 30 of *Lecture Notes in Networks and Systems*. Cham, Switzerland: Springer. 2018 (pp. 111–127).

42. Mininet. http://mininet.org; 2017.

43. Abrignani MD, Buratti C, Dardari D, et al. The EuWIn Testbed for 802.15.4/Zigbee Networks: From the Simulation to the Real World. In: *ISWCS 2013; The Tenth International Symposium on Wireless Communication Systems*; 2013: 1-5.

44. D22.1 - Definition of EuWIn@CNIT/Bologna testbed interfaces and preliminary plan of activities. In: NoE-Newcom♯. ; 2013.

45. Ryu SDN Framework. https://osrg.github.io/ryu/; 2017.

46. Callegati F, Cerroni W, Contoli C, Foresta F. Performance of Intent-based Virtualized Network Infrastructure Management. In: NONE ., ed. *Proceedings of IEEE ICC 2017, Paris, France*; 2017.

□