Toward dynamic virtualized network services in telecom operator networks

Towards Dynamic Virtualized Network Services in Telecom Operator Networks

Ivano Cerrato[a], Alex Palesandro[a], Fulvio Risso[a,*], Marc Suñé[b], Vinicio Vercellone[c], Hagen Woesner[b]

[a]*Department of Computer and Control Engineering, Politecnico di Torino, Torino, Italy*
[b]*Berlin Institute for Software Defined Networks GmbH, Berlin, Germany*
[c]*Telecom Italia lab, Torino, Italy*

Abstract

NFV and SDN are nowadays seen as a solid opportunity by telecom operators to reduce costs while at the same time providing new and better services. Recently, the Unify project proposed a multi-layered architecture that, leveraging different levels of abstraction, can orchestrate and deploy generic network services on the physical infrastructure of the telecom operator. In this paper, we exploit such an architecture to deliver end-to-end generic services in presence of multiple concurring players (e.g. network operator, end-users), leveraging a new simple data model. Particularly, we propose a description-based approach allowing to deploy agile, implementation-independent and high-level network services over a distributed set of resources. The resulting data model can abstract generic services, including both middlebox-based (e.g., firewalls, NATs, etc.) and traditional LAN-based ones (e.g., a bittorrent client). Finally, two distinct prototypes, originated by different design principles, are implemented in order to validate our proposal with the aim of demonstrating the adaptability of our approach to different contexts.

Keywords: NFV, SDN, service graph, forwarding graph, virtual network functions, network orchestration

1. Introduction

The way network services are delivered has dramatically changed in the last few years thanks to the Network Functions Virtualization (NFV) paradigm, which allows network services to experiment the same degree of flexibility and agility already available in the cloud computing world. In fact, NFV proposes to transform the network functions that today run on dedicated appliances (e.g., firewall, WAN accelerator) into a set of software images that can be consolidated into high-volume standard servers, hence replacing dedicated middleboxes with virtual machines implementing those Virtual Network Functions (VNFs). Thanks to their software-based nature, VNFs could be potentially deployed on any node with computing capabilities located everywhere in the network, ranging from the home gateway installed in the customer premises till to the data center servers [1, 2].

NFV is mainly seen as a technology targeting network operators, which can exploit the power of the IT virtualization (e.g., cloud and datacenters) to deliver network services with unprecedented agility and efficiency and at the same time achieve a reduction of OPEX and CAPEX. However, also end users (e.g., xDSL customers) can benefit from NFV, as this would enable them to customize the set of services that are active on their Internet connection. But, while NFV currently focuses mostly on middlebox-based applications (e.g., NAT, firewall), end users are probably more oriented to services based on traditional network facilities (e.g., L2 broadcast domains), which receive less consideration in the NFV world.

Motivated by the growing interest, e.g., of telecom operators, in extending the functionalities of Customer Premise Equipments (CPEs) in order to deliver new and improved services to the users [3] [4] [5] [6], this paper presents a solution that is oriented to deliver *generic* network services that can be selected by *multiple players*. Particularly, our proposal enables also the *dynamic* instantiation of *per-user* network services on the large infrastructure of the telecom operators, possibly starting from the home gateway till the data center, as depicted in Figure 1. Our solution enables several players (e.g., telecom operator, end users, etc.) to cooperatively define the network services; moreover, it is general enough to support both traditional middlebox functions as well as traditional host-based network services. For example, a customer can define its own network service by asking for a transparent firewall and a Bittorrent client, while the network operator complements those applications by instantiating a DHCP and a NAT service[1].

In our solution, the entire network infrastructure is controlled by a service logic that performs the identification of the user that is connecting to the network itself, following the approach proposed in [7]. Upon a successful identification, the proper set of network functions chosen by the user is instantiated in one of the nodes (possibly, even the home gateway) available on the telecom operator network, and the physical infrastructure is configured to deliver the user traffic to the above set of VNFs.

The paper describes the service-oriented layered architecture to achieve those objectives, modeled after the one proposed by

*Corresponding author. Email address: fulvio.risso@polito.it.

[1] In this paper we assume that end users are only enabled to select VNFs *trusted* by the operator. The case in which they can deploy *untrusted* VNFs (e.g., implemented by the end users themselves) would in fact open security issues that are beyond the scope of this work.

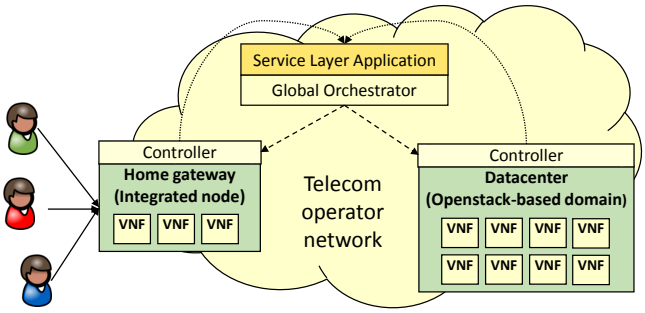


Figure 1: Deployment of virtual network functions on the telecom operator network.

the Unify project [8, 9], and a possible set of data models that are used to describe and instantiate the requested network services starting from an high-level and user-friendly view of the service. The high-level description is then converted into a set of primitives (e.g., virtual machines, virtual links) that are actually used to instantiate the service on the physical infrastructure. Moreover, it presents two possible implementations of the nodes of the infrastructure layer on which the service is actually deployed. Particularly, we explored two solutions that are based on different technologies, with different requirements in terms of hardware resources. The first is based on the OpenStack open-source framework and it is more appropriate to be integrated in (existing) cloud environments; the second exploits mostly dedicated software and it is more oriented to the domestic/embedded segment (e.g., resource-constrained CPEs).

The remainder of this paper is structured as follows. Section 2 provides an overview of the related works, while Section 3 introduces an architecture to deploy general network services across the whole network under the control of the telecom operator (as shown in Figure 1). Section 4 details some formalisms expressing the service to be deployed, which are then exploited to solve the challenges arising from our use case, as discussed in Section 5. Section 6 details the preliminary implementation of the architecture, which is then validated in Section 7, both in terms of functionalities and performance. Finally, Section 8 concludes the paper and provides some plans for the future.

2. Related work

Three FP7 EU-funded projects focusing on the integration of the NFV and SDN concepts (UNIFY [10], T-NOVA [11] and SECURED [12]) started recently. Particularly, the first one aims at delivering an end-to-end service that can be deployed everywhere in the telecom operator network, starting from the points of presence at the edge of the network, till to the datacenter, by exploiting SDN and NFV technologies. Similarly, T-NOVA proposes an equivalent approach that puts more emphasis on the target of providing a uniform platform for third-party VNF developers, while SECURED aims at offloading personal security applications into a programmable device at the edge of the network. An SDN-based service-oriented architecture has been proposed also in [13, 14], which enables to deliver middlebox-based services to user devices, leveraging Service Function Chaining and SDN concepts.

From the industry side, the IETF Service Function Chaining (SFC) [15] working group aims at defining a model to describe and instantiate network services, which includes an abstract set of VNFs, their connections and ordering relations, provided with a set of QoS constraints. Similarly, the European Telecommunications Standards Institute (ETSI) started the Industry Specification Group for NFV [16], which aims at developing the required standards and sharing their experiences of NFV development and early implementation. Based on the ETSI proposal, the Open Platform for NFV (OPNFV) project [17] aims at accelerating the evolution of NFV by defining an open source reference platform for Virtual Network Functions.

The problem of CPE virtualization, which represents one of the possible use cases of our architecture, is investigated in several papers (e.g., [3, 4, 5, 6]); however they have a more limited scope as do not foresee the possibility to instantiate the service across an highly distributed infrastructure and focus on more technological-oriented aspects.

Finally, the OpenStack [18] community is aware of the possibility to use that framework to deliver NFV services as well, as shown by the new requirements targeting of traffic steering and SFC primitives [19, 20]; in fact, we rely on some preliminary implementation of those functions [21] in order to build our prototype.

3. General architecture

Our reference architecture to deliver network services across the telecom operator network, which follows closely the one proposed by the ETSI NFV working group [16], was defined in the FP7 UNIFY project [8, 9] and it is shown in Figure 2. As evident from the picture, it allows the deployment of network services through three main portions, namely the *service layer*, the *orchestration layer* and the *infrastructure layer*.

3.1. Service layer

The *service layer* represents the upper level component of our system and enables different players to define their own network services. Although our service layer includes some use-case specific functions such as user identification (detailed in Section 6.1), we introduce here the general concept of a generic service description expressed in an high level formalism called *service graph* (Section 4.1), which enables the definition of generic services (expressed independently by each player) and their potential interactions.

The service graph could be provided accompanied with several non-functional parameters. Particularly, we envision a set of *Key Quality Indicators (KQIs)* that specify requirements such as the maximum latency allowed between two VNFs, or the maximum latency that can be introduced by the entire service. We also foresee the definition of a list of high-level policies to be taken into account during the deployment of the service. An example of such policies could be the requirement of deploying the service in a specific country because of legal reasons.

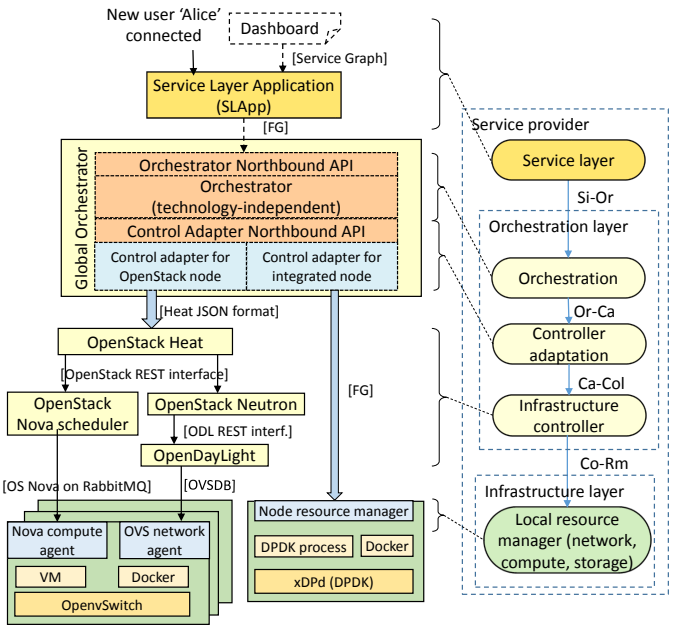


Figure 2: Overall view of the system, including the two implementations of the infrastructure layer.

Given the above inputs, possibly facilitated by some graphical tools that allow different players to select and build the desired service, the service layer should be able to translate the service graph specification into an orchestration-oriented formalism, namely the *forwarding graph* (Section 4.2). This new representation provides a more precise view of the service to be deployed, both in terms of computing and network resources, namely VNFs and interconnections among them, always conserving KQIs and policies imposed by the player that defined the service.

As depicted in Figure 2, the service layer includes a component that implements the service logic, identified with the service layer application (SLApp) block in the picture. The service layer could also export an API that enables other components to notify the occurrence of some specific events in the lower layers of the architecture. The SLApp module could react to these events in order to implement the service logic required by the specific use case. An example of such events may be a new user device (e.g., a smartphone) that connects to the network, which could trigger the deployment of a new service graph or an update of an existing one.

Finally, the service northbound interface enables also cloud-like services, as well as it offers to 3rd-party providers (e.g., content-providers) the possibility to deploy services in the operator infrastructure, orchestrating resources on demand and being billed for their utilization in a pay-per-use fashion.

3.2. Orchestration layer

The *orchestration layer* sits below the service layer and it is responsible of two important phases in the deployment of a service.

First, it manipulates the forwarding graph in order to allow its deployment on the infrastructure, adapting the service definition to the infrastructure-level capabilities, which may require the deployment of new VNFs for specific purposes, as well as the consolidation of several VNFs into a single one (Section 4.2). Second, the orchestration layer implements the scheduler that is in charge of deciding where to instantiate the requested service. The scheduling could be based on different classes of parameters: *(i)* information describing the VNF, such as the CPU and the memory required; *(ii)* high-level policies and KQIs provided with the forwarding graph; *(iii)* resources available on the physical infrastructure, such as the presence of a specific hardware accelerator on a certain node, as well as the current load of the nodes themselves.

According to Figure 2, the orchestration layer is composed of three different logical sub-layers. First, the *orchestration* sub-layer implements the forwarding graph transformation and scheduling in a technology-independent approach, without dealing with details related to the particular infrastructure, which is under the responsibility of the infrastructure layer. The next component, called *controller adaptation* sub-layer, implements the technology-dependent logic that is in charge of translating the forwarding graph into the proper set of calls for the northbound API of the different *infrastructure controllers*, which correspond to the bottom part of the orchestration layer. Infrastructure controllers are in charge of applying the above commands to the nodes operating on the physical network; the set of commands needed to actually deploy a service is called *infrastructure graph* (Section 4.3) and, being infrastructure-specific, changes according to the physical node/domain that will host the service that is going to be instantiated. The infrastructure controller should also be able to identify the occurrence of some events in that layer (e.g., unknown traffic arrives at a given node), and to notify it to the upper layers of the architecture. As shown in Figure 2, different nodes/domains require different infrastructure controllers (in fact, each resource has its own controller), which in turn require many *control adapters* in the controller adaptation sub-layer.

Having in mind the heterogeneity (e.g., core and edge technologies) and size of the telecom operator network, it is evident how the global orchestrator, which sits on top of many resources, is critical in terms of performance and scalability of the entire system. For this reason, according to the picture, the global orchestrator has syntactically identical northbound and southbound interfaces (in fact, it receives a forwarding graph from the service layer, and it is able to provide a forwarding graph to the next component), which paves the way for a hierarchy of orchestrators in our architecture. This would enable the deployment of a forwarding graph across multiple administrative domains in which the lower level orchestrators expose only some information to the upper level counterparts, which allows the architecture to potentially support a huge number of physical resources in the infrastructure layer. Although such a hierarchical orchestration layer is an important aspect of our architecture, it is out of the scope of this paper and it is not considered in the implementation detailed in Section 6.2.

3.3. Infrastructure layer

The *infrastructure layer* sits below the orchestration layer and contains all the physical resources that will actually host the deployed service. It includes different nodes (or domains), each one having its own infrastructure controller; the global orchestrator can potentially schedule the forwarding graph on each one of these nodes. Given the heterogeneity of modern networks, we envision the possibility of having multiple nodes implemented with different technologies; in particular, we consider two classes of infrastructure resources.

The first class consists in cloud-computing domains such as the *OpenStack-based domain* in Figure 1, referencing one of most popular cloud management toolkits, each one consisting of a cluster of physical machines managed by a single infrastructure controller. The second class of resources is instead completely detached by traditional cloud-computing environments, representing nodes such as the future generation of home-gateways hosted in the end users' homes. One of such a node, called *integrated node*, is shown in the bottom-left part of Figure 1 and consists of a single physical machine that is provided mostly with software written from scratch. Moreover, the infrastructure controller is integrated in the same machine hosting the required service.

The infrastructure layer does not implement any logic (e.g., packet forwarding, packet processing) by itself; in fact, it is completely configurable, and each operation must be defined with the deployment of the proper forwarding graph. This makes our architecture extremely flexible, since it is able to implement whatever service and use case defined in the service layer.

4. Data models

This section details the data abstractions that are used to model and then deploy the network services on the physical infrastructure. Our data models are inspired by the NFV ETSI standard [16], which proposes a service model composed of "*functional blocks*" connected together to flexibly realize the desired service. In order to meet the objectives described in the introduction, we instantiated the ETSI abstract model in multiple flavors according to the details needed in the different layers. All those flavors are inspired by the objective of integrating the functional component description of network services and their topology, together with the possibility to model also existing services provided by cloud computing.

4.1. Service graph

The **service graph (SG)** is a high level representation of the service that includes both aspects related to the infrastructure (e.g., which network functions implement the service, how they are interconnected among each other) and to the configuration of these network functions (e.g., network layer information, policies, etc.). Our SG is defined with the set of basic elements shown in Figure 3 and described in the following of this section. These building blocks were selected among the most common elements that we expect are needed to define network services.

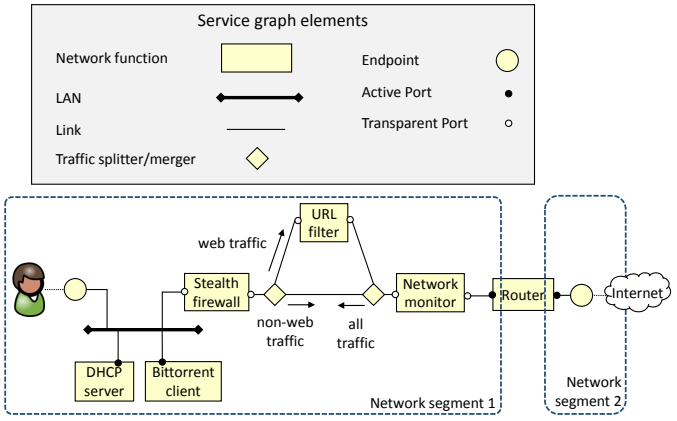


Figure 3: Service graph: basic elements and example.

The **network function** (NF) is a functional block that may be later translated into one (or more) VNF images or to a dedicated hardware component. Each network function is associated with a template (Section 4.4) describing the function itself in terms of memory and processing requirements, required processor architecture (e.g., x86-64), number and type of ports, etc.

The **active port** defines the attaching point of a NF that needs to be configured with a network-level address (e.g., IPv4), either dynamic or static. Packets directed to that port are forwarded by the infrastructure based on the link-layer address of the port itself.

The **transparent port** defines the attaching point of a NF whose associated (virtual) NIC does not require any network-level address. If traffic has to be delivered to that port, the network infrastructure has to "guide" packets to it through traffic steering elements, since the natural forwarding of the data based on link-layer addresses does not consider those ports.

The **local area network (LAN)** represents the (logical) broadcast communication medium. The availability of this primitive facilitates the creation of complex services that include not only transparent VNFs, but also traditional host-based services that are usually designed in terms of LANs and hosts.

The **point-to-point link** defines the logical wiring among different components and can be used to connect two VNFs together, to connect a port to a LAN, and more.

The **traffic splitter/merger** is a functional block that allows to split the traffic based on a given set of rules, or to merge the traffic coming from different links. For instance, it is used in Figure 3 to redirect only the outgoing web traffic toward an URL filter, while the rest does not cross that NF.

Finally, the **endpoint** represents the external attaching point of the SG. It can be either a logical entity or a specific port (e.g., a physical/virtual NIC, a network tunnel endpoint), active on a given node of the physical infrastructure. An endpoint can be used to attach the SG to the Internet, to an end user device, but also to the endpoint of another service graph, if several of them have to be cascaded in order to create a more complex service. Each endpoint is associated with an *identifier* and an optional *ingress matching rule*, which are required for the SGs attaching rules to operate (detailed in Section 4.1.1).

Figure 3 provides a SG example with three NFs connected to a LAN, featuring both active (e.g., the DHCP server and the bittorrent machine, which need to be configured with IP addresses) and transparent ports (the stealth firewall). The outgoing traffic exiting from the stealth firewall is received by a splitter/merge block, which redirects the web traffic to an URL filter and from here to a network monitor, while the non-web traffic travels directly from the stealth firewall to the network monitor. Finally, the entire traffic is sent to a router before exiting from the service graph. In the opposite direction, the traffic splitter/merger on the right will send *all* the traffic coming from Internet to the stealth firewall, without sending anything to the URL filter as this block needs to operate only on the outbound traffic.

As cited above, the SG also includes aspects related to the configuration of the NFs, which represent important service-layer parameters to be defined together with the service topology, and that can be used by the control/management plane of the network infrastructure to properly configure the service. In particular, this information includes network aspects such as the IP addresses assigned to the active ports of the VNFs, as well as VNF-specific configurations, such as the filtering rules for a firewall.

The SG can potentially be inspected to assess formal properties on the above configuration parameters; for example, the service may be analyzed to check if the IP addresses assigned to the VNFs active ports are coherent among each others. To facilitate this work, the SG defines the **network segment**, which is the set of LANs, links and ports that are either directly connected or that can be reached through a NF by traversing only its transparent ports. Hence, it corresponds to an extension of the broadcast domain, as in our case data-link frames can traverse also NFs (through their transparent ports), and it can be used to check that all the addresses (assigned to the active ports) of the same network segment belong to the same IP subnetwork. As shown in the picture, a network segment can be extended outside of the SG; for instance, if no L3 device there exists between an end user terminal and the graph endpoint, the network segment also includes the user device.

4.1.1. Cascading service graphs

As introduced above, the SG endpoints are associated with some parameters that are used to connect SGs together (*cascading graphs*). Particularly, the *identifier* is the foundation of the SG attaching rules, as only the endpoints with the same identifier (shown with the same color in Figure 4) can be attached together. Instead, the optional *ingress matching rule* defines which traffic is allowed to *enter* into the graph through that particular endpoint, e.g., only the packets with a specific source MAC address.

The rules that define how to connect several graphs together change according to both the number of graphs to be connected and the presence of an ingress matching rule on their endpoints. While the case for two endpoints directly connected looks straightforward, the problem with three or more endpoints is more complex. Figure 4 shows two examples in which three endpoints must be connected together. In the first case,

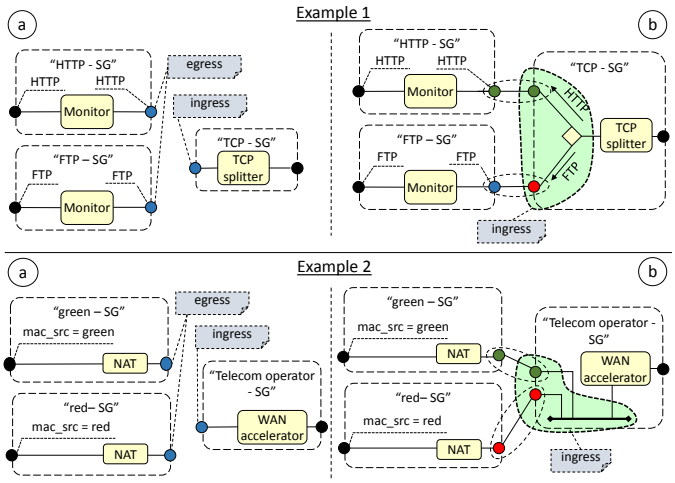


Figure 4: Cascading SGs.

two egress endpoints are associated with an ingress matching rule that specifies which traffic must *enter* into the graph through that endpoint. This ingress matching rule must be used, in case of traffic going from the right to the left, to deliver the desired packets to the correct graph, notably HTTP traffic to the "HTTP-SG" and FTP traffic to the "FTP-SG". This is achieved by transforming the ingress endpoint of the "TCP-SG" into the set of components enclosed in the green shape of Example 1(b), namely a traffic splitter/merger module attached with many new endpoints, each one connected to a different graph. This way, the common "TCP-SG" will be able to dispatch the packet answers to the proper graph.

The second example in Figure 4 shows the case in which the egress endpoints are not associated with any ingress matching rule, which makes it impossible to determine the right destination for the packets on the return path, as a traffic splitter/merger module cannot be used in the "telecom operator-SG" to properly dispatch the traffic among them. In this case, the ingress endpoint of the common "telecom operator-SG" is transformed into a LAN connected to several new endpoints, each one dedicated to the connection with a single other graph. This way, thanks to the MAC-based forwarding guaranteed by the LAN, the "telecom operator-SG" can dispatch the return packets to the proper graph, based on the MAC destination address of the packet itself.

4.2. Forwarding graph and lowering process

The SG provides an high level formalism to define network services, but it is not adequate to be deployed on the physical infrastructure of the network because it does not include all the details that are needed by the service to operate. Hence, it must be translated into a more resource-oriented representation, namely the **forwarding graph (FG)**, through a **lowering process** that resembles to the intermediate steps implemented in software compilers. The FG can be seen as a generalization of the OpenFlow data model that specifies also the functions that have to process the traffic into the node, in addition to define the (virtual) ports the traffic has to be sent to.

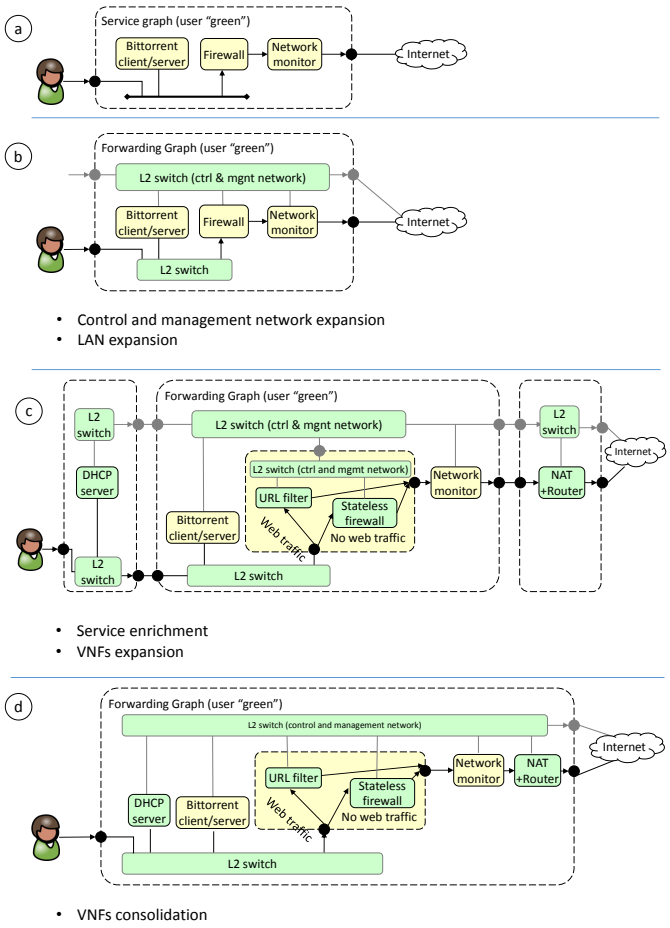


Figure 5: From the SG to the FG: the *lowering process*.

The different steps of the lowering process are shown in Figure 5 and discussed in the following.

The **control and management network expansion** enriches the service with the "control and management network", which may be used to properly configure the VNFs of the graph. In fact, most NFs require a specific vNIC dedicated to control/-management operations; although this may be an unnecessary detail for the player requiring the service, those network connections have to be present in order to allow the service to operate. In this step, the control network is created as a LAN and all the VNFs that actually have vNICs identified as *control interfaces* in their template (Section 4.4) are attached to it automatically. An example of this step is evident by a comparison between Figure 5(a) and Figure 5(b), in which a control/-management network consisting of a L2 switch VNF has been added to the graph, although more complex forms for the control network (e.g., including also other VNFs such as a firewall) can be defined as well.

The **LAN expansion** translates the abstract LAN element defined in the SG into a proper (set of) VNFs that emulate the broadcast communication medium, e.g., a software bridge or an openflow switch with an associated controller implementing the L2 learning mechanism. This step is shown in Figure 5(b) where the LAN is replaced with a software bridge.

```
"forwarding-graph" : {
  "id" : "abcd123",
  "flow-rules" :  [
    {
      "flow-space" : {
        "port"  : "endpoint:1",
      },
      "action" : {
        "type" : "forward",
        "function" : "stateless_firewall:1"
      }
    },
    {
      "flow-space" : {
        "port" : "stateless_firewall:2",
        "tcp_src" : "80"
      },
      "action" : {
        "type" : "forward",
        "function" : "URLfilter:1"
      }
    },
    .......
  ]
}
```

Figure 6: Excerpt of a forwarding graph.

The **service enrichment** requires that the graph is analyzed and enriched with those functions that have not been inserted in the SG, but that are required for the correct implementation of the service. An example is shown in Figure 5(c), where the graph analysis determines that the network segment connected to the user does not include any DHCP server, nor routing and NAT functionalities; in this case the proper set of VNFs are added automatically.

The **VNFs expansion** can replace a VNF with other equivalent VNFs, properly connected in a way to implement the required service. As an example, the firewall in Figure 5(b) is replaced in Figure 5(c) by a subgraph composed of a URL filter only operating on the web traffic, while the non-web traffic is delivered to a stateless firewall. As evident, the ports of the "original" VNF are now the endpoints of the new subgraph, which also has a control network dedicated to the new VNFs. Moreover, these new VNFs are in turn associated with a template, and can be recursively expanded in further subgraphs; this is equivalent to the "*recursive functional blocks*" concept provided in the ETSI standard [16], which may trigger further optimization passes.

The **VNFs consolidation** analyzes the FG looking for redundant functions, possibly optimizing the graph. For instance, Figure 5(d) shows an example in which two software bridges connected together are replaced with a single software bridge instance with the proper set of ports, hence limiting the resources required to implement the LANs on the physical infrastructure.

The **endpoints translation** converts the graph endpoints in either physical ports of the node on which the graph will be deployed, virtual ports (e.g., GRE tunnels) that connect to another graph running in a different physical server, or endpoints of another FG, if many graphs running on the same server must be

connected together.

Finally, the **flow-rules definition** concludes the lowering process. In particular, (*i*) the connections among the VNFs, (*ii*) the traffic steering rules defined by the SG traffic splitter/merger, and (*iii*) the ingress matching rules associated with the endpoints, are translated into a sequence of "flowspace/action" pairs (Figure 6). The flow space includes all the fields defined by Openflow 1.3 [22] (although new fields can be defined), while the action can be a forwarding rule either to a physical or a virtual port.

As a final remark, the FG does not specify all low level details such as the physical node on which the service will be deployed, as well as the reference to the precise physical/virtual NICs needed by the VNF to operate, which are replaced by generic VNF entry/exit points, as shown in Figure 6. The final translation from abstract to actual VNF ports will be carried out in the next step.

4.3. Infrastructure graph and reconciliation process

The **infrastructure graph (IG)** is the final representation of the service to be deployed, which is semantically, but not syntactically, equivalent to the FG. The IG is obtained through the **reconciliation process**, which maps the FG on the resources available in the infrastructure layer, and it consists of the sequence of commands to be executed on the physical infrastructure in order to properly deploy and connect together all the required VNFs.

This process takes into account that some of the VNFs in the FG can be implemented through some modules (both software and hardware) already available on the node on which the graph is going to be deployed, instead of being implemented with the image specified in the template. For example, if the node is equipped with a virtual switch (vSwitch) that supports also the backward learning algorithm such as OpenvSwitch, all the L2 switch VNFs in the FG are removed and those functions are carried out through the vSwitch itself, as shown in the right portion of Figure 7. Instead, as depicted in the left of Figure 7, if the node features a pure Openflow vSwitch (such as xDPd), all the VNFs specified in the FG will be implemented by instantiating the proper images, either VMs implementing the L2 bridging process or through a set of coordinated VMs implementing an OpenFlow switch controlled by an OpenFlow controller with the proper virtual LAN emulation software. Obviously, other mappings between the VNFs and the resources available on the node are possible, according to the specific capabilities of the infrastructure layer; for instance we can obtain a different IG (hence a different number of deployed VNFs) starting from the same FG. The mapping of a VNF on a specific resource is possible thanks to the definition of a set of standard VNF names that uniquely identify a precise network function, such as the name "`L2 switch`" that identifies a LAN emulation component; this allows the reconciliation module to recognize the function needed and, possibly replace it with a more appropriate implementation.

After the reconciliation process, the final IG is converted into the commands (e.g., shell script, protocol messages) required to actually instantiate the graph, such as retrieve the VM image

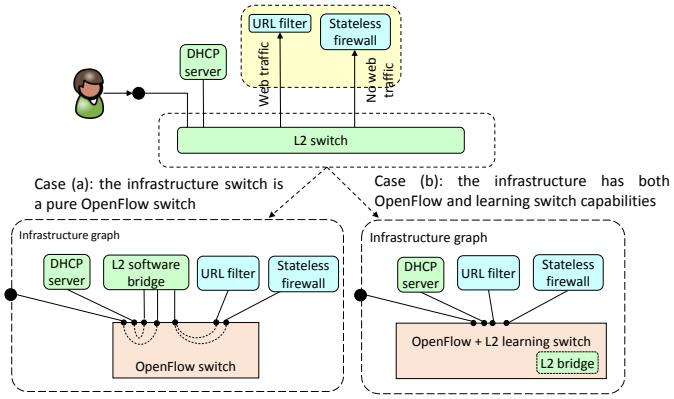


Figure 7: Example of the output of the reconciliation process when mapping a L2 switch functionality in case of two different types of infrastructure nodes.

and start it up, create OpenFlow rules, and more. Particularly, the flow rules defining the links of the FG, i.e., the connections among the VNFs, are properly translated according to the technology used by the physical node to implement the graph. For example, if the physical node interconnects the VNFs through an Openflow vSwitch, each flow rule is converted in a number of Openflow `flowmod` messages, hence combining together SDN and NFV concepts. However, other flavors of the infrastructure layer could implement these connections through other technologies, such as GRE tunnels or VLAN tags. A similar process is applied to VNFs, as VM images are retrieved and started using commands that depend on the technology implementing the VNFs (e.g., virtual machine, Docker container, etc.).

4.4. Network function template

Each VNF is associated with a template that describes the VNF itself in terms of both physical characteristics (e.g., CPU and memory requirements) and possible infrastructure-level configurations (e.g., how many vNICs can be configured); an example of such a template is provided in Figure 8.

The template contains some information related to the hardware required to execute the VNF, such as the amount of memory and CPU as well as the CPU instruction set. Moreover, the boolean element `expandable` indicates if the VNF consists of a single image, or if it is actually a subgraph composed of several VNFs connected together. In the former case, the `uri` element refers to the image of the VNF, while in the latter it refers to a graph description that must replace the original VNF in the FG. In the case of *non-expandable* VNF, the template also specifies the image type such as KVM-compatible VM, Docker container, and more; for instance, the firewall described in Figure 8 is implemented as a single KVM-based virtual machine.

Moreover, the template provides a description of the ports of the VNF, each one associated with several parameters. In particular, the `label` specifies the purpose of that port, and it is useful in the definition of the SG, since it helps to properly connect the VNF with the other components of the service (e.g., the *external* port of the firewall should be connected towards the Internet, while the *internal* ones should be connected towards the

```
"network-function" : {
  "name" : "firewall",
  "expandable": false,
  "uri": "http://myvnfs.com/images/7701f",
  "vnf-type" : "kvm-virtual-machine",
  "memory-requirements": 4096,
  "cpu-requirements": {
    "platform-type": "x86-64",
    "cores-number": 1
  },
  "ports": [
    {
      "label": "control",
      "cardinality" : "1",
      "ipv4-config": "DHCP"
    }
    {
      "label": "external",
      "cardinality": "1",
      "ipv4-config": "none"
    },
    {
      "label": "internal",
      "cardinality": "1-N",
      "ipv4-config": "none"
    }
  ]
}
```

Figure 8: Example of a VNF template.

users). The label could assume any value, which is meaningful only in the context of the VNF. The parameter `ipv4-config`, instead, indicates if the port cannot be associated with an IPv4 address (`none`), or if it can be statically (`static`) or dynamically (`DHCP`) configured. Finally, `cardinality` specifies the number of ports of a certain type; for instance, the VNF of the example has one *control* port, one *external* port, and at least one *internal* port (in fact, it has a variable number of *internal* ports, which can be selected during the definition of the SG).

5. The validation use case: user-defined network services

Section 3 and Section 4 respectively provide a general overview of the architecture and a description of the associated data-models; those concepts could be used in different use cases involving multiple players in defining completely virtualized services.

In order to provide a concrete use case to validate our data models, we selected a challenging scenario in which *end users*, such as xDSL customers, can define their own service graphs to be deployed on the telecom operator infrastructure. Particularly, an end user's SG can only operate on the traffic of that particular end user, i.e., on the packets he sends/receives through his terminal device. Vice versa, the telecom operator can define a SG that includes some VNFs that should operate on all the packets flowing through the network; hence, this SG must be shared among all the end users connected to the telecom operator infrastructure.

Our use case presents some interesting challenges that can be solved through the multi-layer architecture and the data-models presented so far. These challenges, together with their solutions, are summarized in Table 1.

First, the service layer must be able to recognize when a new end user attaches to the network, and then to authenticate the user himself. The API exported by the service layer to the orchestration layer (Section 3.1) could be exploited in our use case just for this purpose.

Note that, since the infrastructure layer does not implement any (processing and forwarding) logic by itself, the authentication mechanism requires the deployment of a specific graph that only receives traffic belonging to unauthenticated users, and which includes some VNFs implementing the user authentication. This could be implemented by means of the SG formalism detailed in Section 4.1, together with the VNF template (Section 4.4).

Second, after the user is authenticated, the service layer must retrieve his SG and then connect it to the telecom operator graph in a way so that the user traffic, in addition of being processed by the service defined by the user himself, is also processed by the VNFs selected by the telecom operator. Notably, the telecom operator graph should be shared among different users, in order to reduce the amount of resources required by the service. The interconnection of two graphs in cascade can be realized by exploiting the graph endpoints elements provided by the SG formalism, as detailed in Section 4.1.1.

Third, the user SG must be completed with some rules to inject, in the graph itself, all the traffic coming from/going towards the end user terminal, so that the service defined by an end user (only) operates on the packet belonging to the user himself. Also this challenge has been solved thanks to the graph endpoints (Section 4.1), which can be associated with rules identifying the traffic that should enter into the graph through a particular endpoint.

Finally, the service layer must require (at the lower layers of the architecture) to deploy the user graph; this operation may require the creation of some tunnels on the network infrastructure so that the user traffic is brought from the network entry point to the graph entry point, which could have been deployed everywhere on the physical infrastructure. The multi-layer architecture proposed in Section 3 ensures the deployment of all the SGs defined at the service layer, regardless of the particular services described by the graphs themselves. In fact, both the lowering process that transforms the SG in FG (Section 4.2), as well as the instructions provided to the infrastructure components through the IG (Section 4.3), are generic enough and can model all the possible services defined at the service layer.

6. Prototype implementation

This section presents the preliminary implementation of the architecture introduced in Section 3, detailing its components and the engineering choices that have been made in order to create the prototype.

6.1. The service layer

Our service layer logic is strictly related to our use case, in which the end users can define generic services to be applied

Table 1: Challenges of the considered use case and related solutions.

	Challenge	Solution
#1	The SLApp recognizes when a new user is connected	API exported by the service layer to be notified of the occurrence of some events
#2	New user's authentication (the infrastructure layer does not implement any processing and forwarding logic by itself)	The SG formalism and the VNF template used to define a graph that includes VNFs implementing the user authentication
#3	Interconnection of several user SGs to a common telecom operator graph	Graph endpoints defined in the SG formalism
#4	A user SG only operates on the traffic belonging to that particular user	Graph endpoint associated with *ingress matching rules* identifying the traffic allowed to flow through the endpoint itself
#5	Each service is implemented on the physical infrastructure	The lowering process and the formalisms (SG, FG, IG) are generic enough to support all the possible services required by the users

to their own traffic, while the operator can define a service operating on all the packets flowing through the network. Our implementation of the SLApp delegates specific tasks to different OpenStack modules, some of which have been properly extended. In particular, **Horizon**, the OpenStack dashboard, is now able to provide to the end users a graphical interface allowing them to express (out of band) the service they expect from the network, using the building blocks depicted in Figure 3. **Keystone**, the token-based authentication mechanism for users and permissions management, is now able to store the user profile, which contains user's specific information such as the description of his own SG. Finally, **Swift**, the OpenStack object storage, has been used to store the VNF templates. Notably, those modules are present also in case of the integrated node implementation (Section 6.3), as the service layer is independent from the actual infrastructure layer.

At boot time, the SLApp asks the orchestration layer to instantiate two graphs, the telecom operator graph and the authentication graphs (Section 6.1.1), which are deployed on one of the available infrastructure nodes. In addition, it configures the network nodes with the proper rules to detect when a new flow is created on the network (e.g., a new user terminal attaches to an edge node), which in turns triggers a call of the proper event handler in the SLAapp that forces an update of the authentication graph, so that it can properly handle the traffic generated from the new connected device. In fact, the SLApp has been notified about the source MAC address of the new packets, which can be used to uniquely identify all the traffic belonging to the new device. This enables the SLApp to update the authentication graph with a new endpoint representing the entry point of the user traffic in the network, and which is associated with an ingress matching rule expressed on the specific MAC address; this way, the new packets can be provided to the authentication graph, wherever the graph itself has been deployed. Finally, the updated graph is passed to the orchestration layer, which takes care of applying the modifications on the physical infrastructure.

A successful user authentication through the authentication graph triggers the instantiation of the SG associated with the user himself. This is achieved by the SLApp, which retrieves the proper SG description from the user profile repository, connects it to the telecom operator-defined SG such as in the example shown in the right part of Figure 4, and starts the lowering process aimed at creating the FG. In particular, the SLApp executes the "*control and management network expansion*" and the "*LAN expansion*" as shown in Figure 5(b), and the "*service enrichment*" step. In our case, the latter consists in adding a DHCP server and a VNF implementing the NAT and router functionalities, in case these VNFs have not been included by the end user during the definition of his own service.

Before being provided to the orchestration layer, the user-side endpoint of the user SG is associated with *(i)* an ingress matching rule expressed on the MAC address of the user device, so that only the packets belonging to that user are delivered to his graph, and *(ii)* the entry point (i.e., the actual port on the physical edge node where the user connects to) of such a traffic in the telecom operator network. This way, the orchestration layer will be able to configure the network in order to bring the user traffic from its entry point into the network to the node on which the graph is deployed.

Finally, the SLApp also keeps track of user sessions in order to allow a user to connect multiple concurrent devices to his own SG. In particular, when a user already logged in attaches to the network with a new device, the SLApp: *(i)* retrieves the FG already created from the orchestration layer; *(ii)* extends it by adding a new endpoint associated with an ingress matching rule operating on all the traffic coming from the new device, and representing the entry point of such packets in the network; *(iii)* sends the new FG to the orchestration layer.

6.1.1. Authentication graph

The *authentication graph* is used to authenticate an end user when he connects to the telecom operator network with a new device, and it is automatically deployed by the service layer when the infrastructure starts, hence turning a service-agnostic infrastructure into a network that implements our use case.

The authentication SG (shown at the top of Figure 9) consists of a LAN connected to a VNF that takes care of authenticating the users, a DNS server and a DHCP server. However, the VNF implementing the users authentication is in fact another SG made up of three VNFs: an Openflow switch, an Openflow controller and a web captive portal. The resulting FG, completed with the control and management network, is shown at the bottom of Figure 9; as evident, only the control network is connected to the Internet while all the user traffic is kept local.

When an unauthenticated user connects to the network, his traffic is brought to the authentication graph. In particular, the DHCP server returns an initial IP address to the user, which is

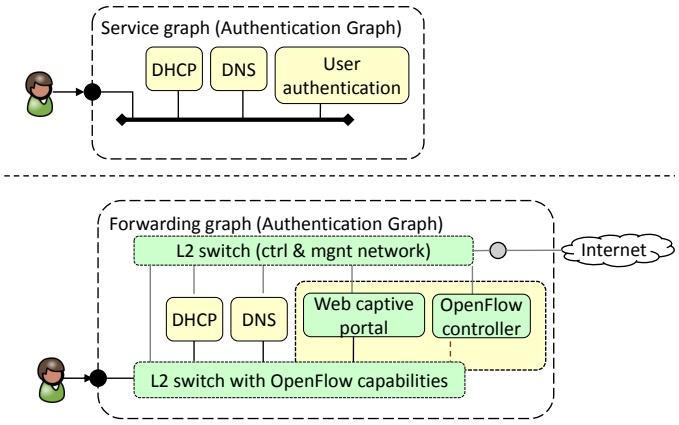


Figure 9: Authentication SG and FG.

able to generate traffic. All DNS queries are resolved by the DNS server into the proper IP addresses, but all the web requests are redirected to the captive portal; in fact, the HTTP traffic entering into the Openflow switch is sent to the Openflow controller, which modifies the original MAC and IP destination addresses with those of the web captive portal and then sends back the packet to the Openflow switch, which will deliver it to the captive portal. This VNF provides a HTTP `302 temporary redirect` message to the user in order to notify the client of the redirection and avoiding wrong caching, then a login page is shown. After the (successful) user authentication, the web captive portal contacts the SLApp through the control network and triggers the deployment of the SG associated with that user on the infrastructure.

6.2. Global orchestrator

The **global orchestrator** implements the first two levels of the orchestration layer depicted in Figure 2 and consists of a technology-dependent and a technology-independent part.

The technology-independent part receives the FG from the service layer (through its northbound interface) and it executes the following operations as defined in the lowering process (Section 4.2). First, for each VNF specified in the graph, it retrieves the corresponding VNF template from the OpenStack Swift service. In case the template is actually a subgraph composed by other VNFs (Section 4.4), it executes the "*VNFs expansion*" step (Figure 5(c)) and retrieves the description of the new VNFs that, in turn, could be recursively expanded in further subgraphs. The "*VNFs consolidation*" step follows, possibly consolidating multiple function instances as shown in Figure 5(d). Finally, the "*flow-rules definition*" step creates a sequence of "flow-space/action" pairs describing how to steer the traffic within the graph.

At this point, the global orchestrator schedules the FG on the proper node(s) of the physical infrastructure. Although the general model presented in Section 3.2 supports a scheduling based on parameters such as CPU and memory requirements of the VNFs, KQIs (e.g., maximum latency, expected throughput) and high level policies, the current implementation simply instantiates the entire FG on the same node used as a network entry point for the traffic to be injected into the graph itself. The resulting FG is then provided to the proper control adapter, chosen based on the type of infrastructure node (i.e., OpenStack-based node or integrated node) that has been selected by the scheduler and that has to execute the FG. These adapters take care of translating the FG into the formalism accepted by the proper *infrastructure controller*, which is in charge of sending the commands to the infrastructure layer. Moreover, they convert the abstract endpoints of the graph (i.e., the ones that have not yet been translated into physical ports e.g., by the service layer) into physical ports of the node; finally, if needed, they instruct the infrastructure controller to create the required GRE tunnels. Tunnels can be used to connect together graphs that have been instantiated on two different nodes (*graph cascading*), or to connect two portions of the same graph that have been deployed on different nodes (*graph splitting*). In our use case, a tunnel is required to bring the user's traffic from the edge node he is connected to towards the node where the graph has been instantiated, as well as to bring the traffic generated by unknown user terminals to the authentication graph.

As a final remark, the global orchestrator supports the update of existing graphs. In fact, when it receives a FG from the service layer, it checks if this graph has already been deployed; in this case, both FGs (the one deployed and the new one) are provided to the proper control adapter, which sends to the infrastructure controller either the difference between the two graphs in case of integrated node, or both FGs in case of OpenStack-based node, as that implementation will be able to automatically identify the differences thanks to the OpenStack Heat module.

6.3. The integrated node

The **integrated node** [23] [24] is the first flavor of our infrastructure layer and it consists of a single physical machine running mostly ad hoc software, whose overall architecture is shown in Figure 10. In this implementation, the infrastructure controller is *integrated* on the same server running the VNFs. Multiple integrated nodes are possible on the infrastructure and must be coordinated by the global orchestrator.

The integrated node receives the FG through the northbound (REST) interface of the **node resource manager**, which is the component that takes care of instantiating the graph on the node itself; this requires to execute the reconciliation process in order to obtain the IG, to start the proper VNF images (downloaded from a **VNFs repository**) and to configure the proper traffic steering among the VNFs. Particularly, the last two operations are executed through two specific modules of the node resource manager, namely the **compute controller** and the **network controller**.

Traffic steering is implemented with a (pure) Openflow DPDK-enabled datapath, based on the **extensible Data-Path deamon (xDPd)** [25]. xDPd supports the dynamic creation of several Openflow switches, called Logical Switch Instances (LSIs); each LSI can be connected to physical interfaces of the node, to VNFs, and to other LSIs. A different LSI (called `tenant-LSI`) is dedicated to steer the traffic among the VNFs

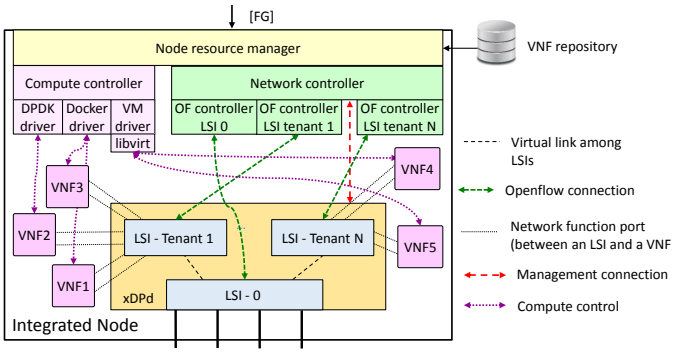


Figure 10: Logical architecture of the integrated node.

of a specific graph, while the `LSI-0` is in charge of classifying the traffic coming from the network (or from other graphs) and of delivering it to the proper `tenant-LSI`. The `LSI-0` is the only one allowed to access the physical interfaces, and the traffic flowing from one `tenant-LSI` to another has to transit through the `LSI-0` as well. Since LSIs are *pure* Openflow switches, the reconciliation process described in Section 4.3 cannot remove the L2Switch VNFs, which are then implemented using the proper software images because of the unavailability of the backward learning algorithm in xDPd.

When a FG description (either a new one or an update of an existing FG) is received by the node resource manager, this module: *(i)* retrieves a software image for each VNF required and installs it; *(ii)* instantiates a `tenant-LSI` on xDPd and connects it to the `LSI-0` and to the proper VNFs; *(iii)* creates a new OpenFlow controller associated to the tenant-LSI that is in charge of inserting the required forwarding rules (i.e., traffic steering), which sets up the proper connections among VNFs as required by the FG. In particular, the FG rules that define the paths among VNFs (and physical ports) originate two sequences of Openflow `flowmod` messages: one to be sent to the `LSI-0`, so that it knows how to steer traffic among the graphs deployed on the node and the physical ports; the other used to drive the `tenant-LSI`, so that it can properly steer the packets among the VNFs of a specific graph.

When a packet enters into the `LSI-0` and cannot be forwarded to any `tenant-LSI`, it is delivered to the `LSI-0` controller using the Openflow `packet in` message; at this point the Openflow controller notifies the service layer of the presence of a new user terminal, which will react by creating the proper network setup (e.g., tunnels) to redirect that traffic to the authentication graph.

The integrated node supports three flavors of VNFs: DPDK processes [26], Docker containers [27], and VMs. While the former type provides better performance (in fact, an LSI exchanges packets with DPDK VNFs with a zero-copy mechanism), Docker containers and VMs guarantee better isolation among VNFs, as well as they allow to limit CPU and memory usage. Data exchange between LSIs and Docker containers/VMs takes place through the `KNI` virtual interface available with the DPDK framework.

Finally, the architecture of the integrated node can support

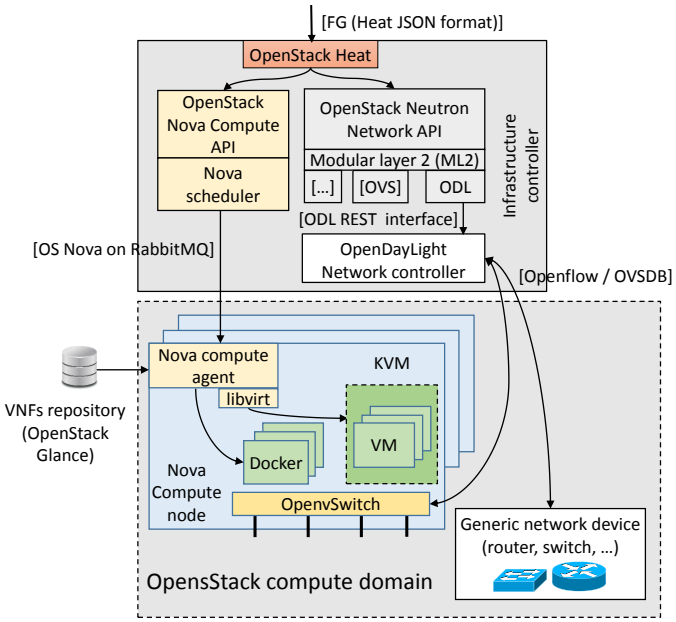


Figure 11: OpenStack-based node.

a network-aware scheduling algorithm, which has the potential to optimize the location of each VNF based on the I/O connections of the VNF itself. This allows for example to start two cascaded VNFs on two cores of the same physical CPU in order to keep the traffic within the same NUMA node, or to allocate a VNF on the CPU that is connected to the NIC used to send the packet out to the network. This is possible because the node resource manager, which takes care of both deploying the VNFs and configuring the vSwitch to properly steer the traffic among them, receives the entire FG from the upper layer, which describes both the VNFs to be executed and the connections among them.

6.4. The OpenStack-based node

The **OpenStack-based node** is the second flavor of our infrastructure layer and it consists of a cluster of servers within the same OpenStack domain. As shown in Figure 11, all the physical machines of the cluster are managed by a single infrastructure controller, which is composed of a number of OpenStack modules and a SDN controller. Multiple OpenStack-based nodes are possible on the infrastructure and must be coordinated by the global orchestrator.

OpenStack [18] is a widespread cloud toolkit used for managing cloud resources (network, storage, compute) in datacenters; hence, its support in our architecture represents an interesting choice because of the possibility to deploy our services in an existing (and widely deployed) environment. However, since OpenStack was designed to support the deployment of *cloud* services, several modifications have been made to support FGs (hence *network* services) as well.

As depicted in Figure 11, our OpenStack-based node exploits the following components: *(i)* **Nova**, the compute service; *(ii)* **Neutron**, the network service; *(iii)* **Heat**, the orchestration layer

and *(iv)* **Glance**, the VM images repository. Openstack is able to start VMs by interacting with a wide range of different hypervisors (e.g. KVM, Xen, VMware); moreover, in order to properly steer the traffic between the several servers under its control our prototype integrates also the **OpenDaylight (ODL)** [28] SDN controller. As evident from the picture, Heat, Nova scheduler, Nova API, Neutron and ODL compose the infrastructure controller, while each physical machine executing the VNFs is a Nova compute node, which runs a Nova compute agent, the **OpenvSwitch** (OVS) [29] softswitch and the **KVM hypervisor**.

When the global orchestrator decides to deploy a FG in an OpenStack-based node, the proper control adapter translates the FG description into the format supported by Heat. To be used in our prototype, Heat has been extended in order to support the `flow-rule` primitive, which describes how to steer the traffic between the ports of the VNFs composing a graph. This primitive provides an interface similar to the OpenFlow 1.3 `flowmod`; however, it allows the traffic steering between virtual ports without knowing in advance the physical server on which the respective VNFs will be scheduled. As soon as Heat receives the FG, it performs a reconciliation step that removes, from the graph itself, all the VNFs implementing the L2 switch, since this functionality will be mapped on the OVS instances running on the physical servers. In fact, OVS is able both to forward traffic based on traffic steering rules, as well as to execute the MAC learning algorithm. After this translation, the FG is decomposed into a set of calls to Nova and Neutron.

For the compute part, Nova receives a sequence of commands for each VNF of the graph in order to deploy and start the VNF itself; at this point, the Nova scheduler *(i)* selects the physical server on which the VNF must be deployed using the standard OpenStack "filter & weight" algorithm[2], *(ii)* sends the proper command to the Nova compute instance on the selected node, which in turn *(iii)* retrieves the VNF image from Glance and finally *(iv)* starts the VM[3]. It is worth noting that Nova scheduler has two limitations: *(i)* it schedules a VNF as soon as it receives the command from Heat; *(ii)* it does not have any information on the paths among the VNFs in the graph. As a consequence, the FG could be split on the available compute nodes without taking into account the paths among the VNFs, clearly resulting in suboptimal performance.

For the networking part, when Heat detects that all the VNFs (i.e., VMs) are started, Heat sends a `flow-rule` at a time to Neutron, which takes care of creating the proper connections among these VNFs. Similarly to Heat, also Neutron has been extended to support the `flow-rule` primitive[4]. When Neutron receives a `flow-rule`, it retrieves the network topology from ODL and then creates the proper Openflow `flowmod` messages required to steer the traffic on the physical infrastructure. At this point, the `flowmods` are provided to ODL, which sends them to the proper switches; note that these switches could be either inside a Nova compute node, or physical switches used to interconnect several servers, in case the VNFs have been instantiated by the Nova scheduler on many compute nodes.

In addition to the components described so far, each OpenStack deployment also includes the **network node**, which is a particular server running some services such as a NAT, a DHCP server, a load balancer and a router; moreover, by default it is crossed by all the traffic entering/leaving the cluster of servers.

Similarly to the integrated node, also the OpenStack-based node notifies the service layer when a new user terminal connects to the node itself in order to allow the system to redirect that traffic to the authentication graph.

6.5. Discussion: Openstack-based node vs. integrated node

Table 2 summarizes the main differences between the integrated node and the OpenStack-based node. As shown, the main advantage of the latter is its capability to deploy SGs in an existing cloud environment (albeit with some modifications), which facilitates the introduction of those services in telecom operator networks; in fact, operators often already have OpenStack instances active in their data centers. However, this very important advantage is balanced by severe limitations compared to the integrated node.

First, OpenStack does not allow the service layer to have the complete control of the service chain, as each OpenStack domain is connected to the Internet through the *network node* (Section 6.4) and all the packets towards/from the Internet are forced to traverse the network services running in this component (e.g., NAT and router), even if the SG does not include those functions.

Second, the OpenStack-based node cannot optimize the placement of the VNFs based on their layout in the FG, e.g., possibly instantiating two consecutive VNF within the same graph on the same physical server. This is due to the fact that the OpenStack scheduler, implemented in Nova, is unaware of the overall layout of the FG: this information is only received by Heat and it is not passed down to the Nova scheduler. To make things worse, the Nova scheduler is invoked individually per each VNF that has to be scheduled and therefore it is unable to implement even a simple optimization such as scheduling all the VNFs of the same FG on the same server. This problem is not present in the integrated node as it receives the entire FG, hence it has all the information related to the required VNFs and the paths among them, hence potentially scheduling the VNFs on the available CPUs by considering their position in the graph, with the obvious advantages in terms of overall performance.

[2]This algorithm acts as follows: first, all the Nova compute nodes that are not able to run a VM are filtered (e.g., because the VM requires an hypervisor that is not available on the node). Then, a weight is associated with each one of the remaining servers, and the one with higher weight is selected to run the VM. The weights are calculated by considering the resources available on the machine.

[3]Note that no modification has been required by Nova compute in order to support the deployment of the FGs.

[4]The `flow-rule` is functional equivalent to the Neutron official traffic steering extension [19]. However, it has not been used in our prototype because: *(i)* it was not available when this prototype was created (July 2014); *(ii)* it does not support ports that do not belong to a VM.

Table 2: Integrated node vs OpenStack-based node.

	Integrated node	**OpenStack-based node**
Compatible with existing cloud environments	No	Yes
Complete control of the FG	Yes	No (due to the network node)
Support to smart scheduling of the FG	Possible	Requires many changes to the OpenStack internals
Type of VNFs	Docker containers, DPDK processes, VMs	VMs, Docker containers (not completely supported)

Third, the integrated node shows better memory consumption compared to the OpenStack-based node, whose components have not been created for resource-constrained environments as this is unlikely to occur in (almost resource-unlimited) data centers.

Finally, the OpenStack-based node supports only VM-based VNFs, as the initial support for Docker containers appears still rather primitive. Vice versa, the integrated node supports VNFs implemented as Docker containers, VMs and DPDK processes, which seems to suggest the possibility to write more lightweight and efficient VNFs, particularly with respect with the potentially better I/O capabilities, which represent a fundamental difference from VNF and traditional VM-based services.

7. Prototype validation

To validate the architecture described in the paper, we carried out several tests aimed at both testing the functionalities implemented, and to measure the performance of the infrastructure layer in terms of throughput, latency introduced and resource required. The tests were repeated both with an infrastructure layer consisting of a single integrated node, as well as in case of an OpenStack cluster composed by two compute nodes. Note that, in the latter case the graphs are split so that the VNFs are distributed between the two physical servers.

7.1. Service overview

The FGs deployed in the tests are shown in Figure 12; according to our use case, these graphs include the authentication graph used to authenticate new end users connected to the network, and the telecom operator graph, which provides connectivity to the Internet and that is crossed by the traffic generated from/going towards all the end users. The control network of this telecom operator graph also includes a firewall, so that only the authorized entities (e.g., the telecom operator itself) can control and configure the deployed VNFs.

The end user graph provides an example of traffic steering, since it requires that the web traffic is delivered to a traffic monitor and then to a firewall that blocks the HTTP GET towards specific URLs, while the other packets simply traverse a second traffic monitor VNF. Thanks to the control interface of the traffic monitors we are able to observe the packets flowing through the specific VNF, and hence to validate the correct behavior of the traffic steering mechanism.

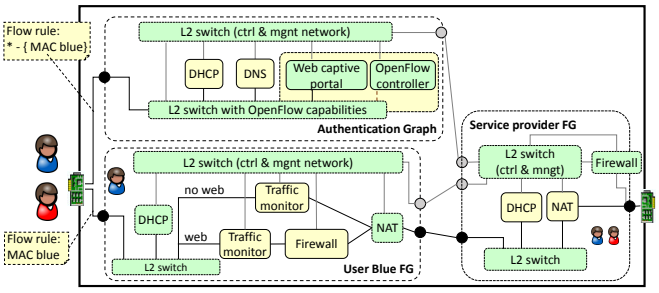


Figure 12: Use case scenario.

During the tests carried out on the OpenStack-based node, the VNFs are implemented as VMs running on the KVM hypervisor. In contrast, in the tests with the integrated node, the firewall is implemented as a DPDK process[5], while all the others VNFs are implemented as Docker containers. In particular, both the VMs and the Docker containers run an Ubuntu operating system, and the VNFs are implemented through *standard* Linux tools (e.g., `iptables`).

As a final remark, according to our use case and the current implementation of the architecture, the end users are directly connected to the node on which their graphs are deployed.

7.2. Performance evaluation

This section shows the tests executed in order to measure the performance of the preliminary implementation of our architecture.

During the tests, a machine is dedicated to the execution of the service layer (i.e., SLApp, Keystone and Horizon) and the global orchestrator; it is equipped with 16 GB RAM, 500GB HD, Intel i7-2620M @ 2.7 GHz (one core plus hyperthreading) and OSX 10.9.5, Darwin Kernel Version 13.4.0, 64 bit, which is the same for both the infrastructure nodes.

The infrastructure layer is implemented on a set of servers with 32 GB RAM, 500GB HD, Intel i7-3770 @ 3.40 GHz CPU (four cores plus hyperthreading) and Ubuntu 12.04 server OS, kernel 3.11.0-26-generic, 64 bits. In case of the integrated node, one of those machines executes all the software. In case of OpenStack-based node, a first machine hosts the infrastructure controller (Heat, Nova scheduler, Nova API, Neutron and ODL)

[5]Note that this firewall is a quite simple single thread process based on the `libpcre` regular expression engine, which drops all the packets matching specific regular expressions.

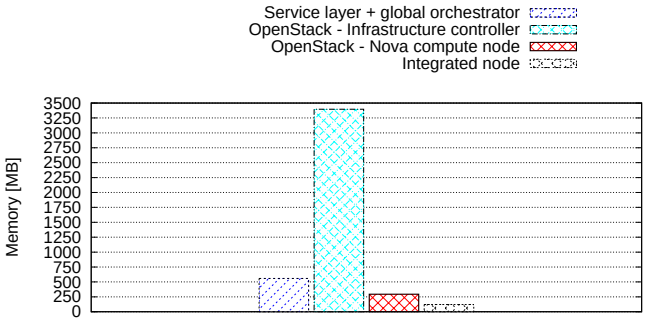


Figure 13: Memory consumption.

and the network node, while two other machines are dedicated to the implementation of two Nova compute nodes (which are connected through a Gigabit Ethernet link).

The memory required by the different components of the system is reported in Figure 13, in which the consumption related to the Nova compute node and to the integrated node has been measured without any VNF deployed. As shown, the infrastructure controller for the OpenStack-based node is the heaviest component, while the requirements of the integrated node, which is almost based on ad hoc modules, is quite reduced.

According to Figure 14, we repeated the tests in the following conditions: *(i)* user device and server directly connected using a gigabit Ethernet link; *(ii)* user devices connected, through a gigabit Ethernet network, to the node on which the graphs are deployed, which is in turn connected to the server through a second gigabit Ethernet link. Moreover, as node running the VNFs, we used: the integrated node, an OpenStack-based node with a single server, and an OpenStack-based node consisting of two servers connected with a gigabit Ethernet link.

The first test carried out aims at measuring the latency introduced by the deployed services (Figure 12); in particular, the user device(s) sends 100 `ping` towards the server, and the results were averaged and reported in Figure 14(a). Figure 14(b) shows instead the results of the second test executed, aimed at measuring the throughput obtained during the download of a file of 512 MB from the server; the download has been done using the Linux tool `wget`, which uses the HTTP protocol. Hence, according to the user graph, while the ping is not handled by the firewall, this VNF is instead involved during the file transfer.

As expected, the deployment of a SG on the network does not come for free, since the numbers obtained are reduced with respect to the case in which no service is instantiated between the user device and the server. However, as evident from the figure, this penalty is limited when the graph is deployed on an OpenStack-based node consisting of a single server, both in case a single user graph is instantiated (in addition to the authentication and the telecom operator graphs) and in case of two (identical) user graphs. In this last condition, the test has been executed with both the users pinging/transferring the file at the same time, and the results have been averaged in the table.

Instead, when the graphs are scheduled in an OpenStack cluster of two nodes, performance are worse in both the types of test; for this reason, the measurements have not been repeated with two users connected to the node. The low performance are a consequence of the fact that the standard scheduling algorithm implemented in OpenStack scheduled the user VNFs in a way so that each packet crosses four times the link between the two compute nodes. This confirms the necessity of the introduction, in the Nova scheduler, of an algorithm that schedules the VNFs on the physical servers according to their interconnections in the graph[6].

Surprisingly, results obtained with the integrated node are extremely low, unless the entire graph is deployed on a single server. We are currently investigating the reason for this poor performance, although we suspect they are related to the packet exchange mechanism between the vSwitch and the Docker containers (currently based on the DPDK KNI ports), which should be carefully optimized.

8. Conclusion and future works

This paper presents a network orchestration architecture that, starting from the service required by multiple players (e.g., end users, telecom operator), takes care of instantiating it on the physical infrastructure of the network, by exploiting the opportunities offered by the Network Functions Virtualization (NFV) and Software Defined Networking (SDN) paradigms.

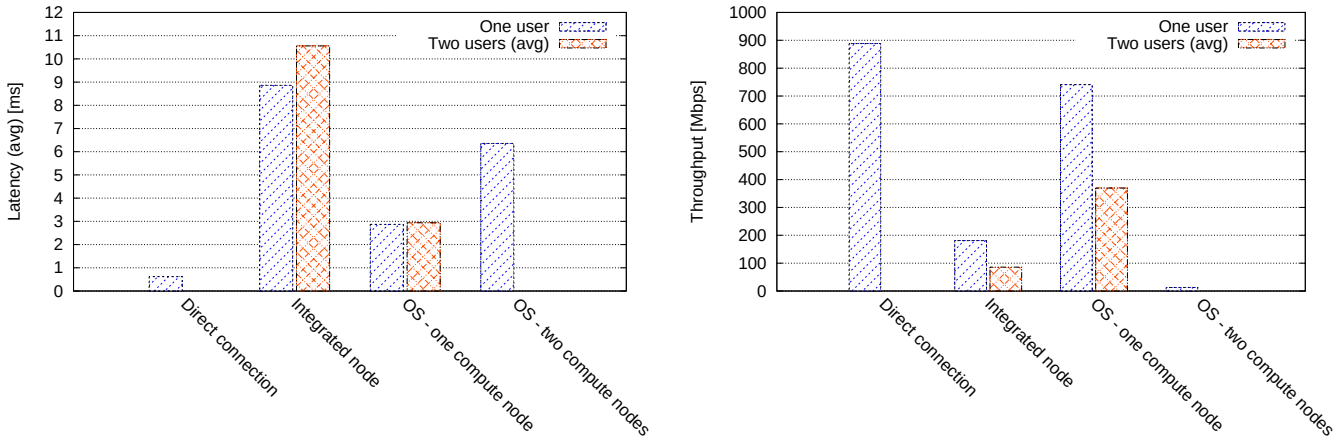
The contribution of this paper is twofold. First, we proposed a new formalism, called *service graph (SG)*, to flexibly model end-to-end network services. The SG data-model describes how to deliver flexible network services, leveraging existing elements and the traffic steering primitives introduced by NFV/SFC. It is worth noting that this SG definition is completely compliant with NFV principles of abstract description of a service, but enriches its traditional expressiveness to model legacy networks and services.

The second contribution is made by the introduction of the *forwarding graph (FG)* and the "lowering process" that leads to the deployment of an optimized service. This translation process is capable to adapt the service delivering to available resources of the underlying infrastructure; moreover, it is also able to detect specific capabilities of selected nodes adapting the infrastructure graph obtained as output.

In order to validate our model, we implemented two prototypes of nodes for the physical infrastructure: the *integrated node* and the *OpenStack-based node*. While the former consists of a single server mainly based on ad hoc components, the latter is implemented as a cluster of server orchestrated by the an extended version of the OpenStack framework. Experimental results showed that, while the integrated node has low requirements in terms of memory, its performance are overcome by the OpenStack-based node in almost all the tests carried out.

It is worth pointing out that the modifications proposed to the "vanilla" OpenStack were designed by avoiding to change existing API or disrupt former primitive behavior provided by the

[6]Note that the same algorithm should also be implemented in the global orchestrator, which schedules the VNFs on the proper nodes of the infrastructure layer.



(a) (b)

Figure 14: Performance of the infrastructure layer: (a) ping; (b) file transfer.

platform. Therefore, those add-ons can be silently integrated in a previous installation, transparently enriching the network capabilities of an OpenStack domain.

As a plan for the future, we foresee two different challenges to be pursued in order to let this architecture to properly scale to the telecom operator network size. First, the proposal of an algorithm to implement a network-aware scheduling, capable of deploying VNFs on the physical infrastructure by considering the paths expressed into the graph.

Second, the definition of a hierarchical orchestration layer through the whole telecom operator network. This would allow the deployment of a FG across multiple administrative domains, in which the lower level orchestrators expose only some information to the upper level counterparts. This scenario is perfectly compatible with our architecture and will be the object of further analysis; in fact, the global orchestrator presented in the paper has syntactically identical northbound and southbound interfaces, and hence a hierarchy of orchestrators is possible.

As a final remark, the configuration parameters for the network functions, as well as the possibility of assessing formal properties on them, are out of the scope of this paper and will be investigated in our future work.

Acknowledgment

This work was conducted within the framework of the FP7 UNIFY project, which is partially funded by the Commission of the European Union. Study sponsors had no role in writing this report. The views expressed do not necessarily represent the views of the authors employers, the UNIFY project, or the Commission of the European Union. The authors wish also to thank Fabio Mignini, Matteo Tiengo and Roberto Bonafiglia for their contribution in the development of the prototypes.

References

[1] European Telecommunication Standards Institute (ETSI), Network Functions Virtualization (NFV) - white paper.

[2] European Telecommunication Standards Institute (ETSI), Network Functions Virtualization; use cases, URL `http://www.etsi.org/deliver/etsi_gs/NFV/001_099/001/01.01.01_60/gs_NFV001v010101p.pdf`.

[3] Berlin Institute for Software Defined Network, Deutsche Telekom Innovation Labs, Your home in your pocket, URL `https://www.opennetworking.org/images/stories/sdn-solution-showcase/BISDN-demo.pdf`.

[4] J. Soares, M. Dias, J. Carapinha, B. Parreira, S. Sargento, Cloud4nfv: A platform for virtual network functions, in: CLOUDNET'14, 2014, pp. 288–293.

[5] F. Sanchez, D. Brazewell, Tethered linux cpe for ip service delivery, in: Proceedings of the First IEEE Conference on Network Softwarization (NetSoft 2015), 2015.

[6] Z. Bronstein, E. Shraga, Nfv virtualisation of the home environment, in: Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th, IEEE, 2014, pp. 899–904.

[7] F. Risso, I. Cerrato, Customizing data-plane processing in edge routers, in: Proceedings of the First European Workshop on Software Defined Networking (EWSDN), 2012, pp. 114–120. doi:10.1109/EWSDN.2012.14.

[8] P. Sköldström, B. Sonkoly, A. Gulyás, F. Németh, M. Kind, F.-J. Westphal, W. John, J. Garay, E. Jacob, D. Jocha, J. Elek, R. Szabó, W. Tavernier, G. Agapiou, A. Manzalini, M. Rost, N. Sarrar, S. Schmid, Towards unified programmability of cloud and carrier infrastructure, in: Proceedings of the Third European Workshop on Software Defined Networking (EWSDN), 2014, pp. 55–60.

[9] A. Császár, W. John, M. Kind, C. Meirosu, G. Pongrácz, D. Staessens, A. Takács, F.-J. Westphal, Unifying cloud and carrier network: Eu fp7 project unify, in: Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC '13), UCC '13, IEEE Computer Society, 2013, pp. 452–457.

[10] Unify: unifying cloud and carrier network (2013). URL `http://www.fp7-unify.eu/`

[11] T-nova: Network functions as a service over virtualised infrastructure), URL `http://www.t-nova.eu/` (2014).

[12] The SECURED project (SECURity at the network EDge)), URL `http://www.secured-fp7.eu/` (2013).

[13] J. Rückert, J. Blendin, N. Leymann, G. Schyguda, D. Hausheer, Software-defined network service chaining, in: EWSDN 2014, 2014.

[14] J. Blendin, J. Rückert, N. Leymann, G. Schyguda, D. Hausheer, Position paper: Software-defined network service chaining, in: Proceedings of the

Third European Workshop on Software Defined Networking (EWSDN 2014), 2014.

[15] Internet Engineering Task Force (IETF), Service Functions Chaining (SFC) working group (2014).
URL https://datatracker.ietf.org/wg/sfc/documents/

[16] European Telecommunication Standards Institute (ETSI), Network Functions Virtualization Industry Specification Group, URL http://portal.etsi.org/NFV.

[17] OPNFV, OPNFV - an open platform to accelerate nfv., URL http://www.opnfv.org (2015).

[18] Openstack, URL http://www.openstack.org/.

[19] Openstack blueprints: Neutron services insertion, chaining, and steering, URL https://blueprints.launchpad.net/neutron/ (2014).

[20] Openstack blueprints: Neutron service chaining specification, URL https://review.openstack.org/#/c/93524/ (2014).

[21] F. Lucrezia, G. Marchetto, F. Risso, V. Vercellone, Introducing network-aware scheduling capabilities in openstack, in: Proceedings of the First IEEE Conference on Network Softwarization (Netsoft 2015), 2015.

[22] Openflow 1.3, URL https://www.opennetworking.org.

[23] Unify Consortium, D5.2 universal node interfaces and software architecture, URL https://www.fp7-unify.eu/files/fp7-unify-eu-docs/Results/Deliverables/UNIFY-WP5-D5.2-Universal%20node%20interfaces%20and%20software%20architecture.pdf (2014).

[24] I. Cerrato, T. Jungel, A. Palesandro, F. Risso, M. Suñé, H. Woesner, User-specific network service functions in an sdn-enabled network node, in: EWSDN 2014, 2014.

[25] xdpd, URL http://www.xdpd.org.

[26] Dpdk, URL http://dpdk.org.

[27] Docker, URL http://www.docker.com.

[28] Opendaylight, URL http://www.opendaylight.org/.

[29] B. Pfaff, J. Pettit, T. Koponen, K. Amidon, M. Casado, S. Shenker, Extending networking into the virtualization layer, in: Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII), 2009.