End-to-end service orchestration across SDN and cloud computing domains

End-to-End Service Orchestration across SDN and Cloud Computing Domains

Roberto Bonafiglia, Gabriele Castellano, Ivano Cerrato, Fulvio Risso
Politecnico di Torino, Dept. of Computer and Control Engineering, Torino, Italy

***Abstract*—This paper presents an open-source orchestration framework that deploys end-to-end services across OpenStack-managed data centers and SDN networks controlled either by ONOS or OpenDaylight. The proposed framework improves existing software in two directions. First, it exploits SDN domains not only to implement traffic steering, but also to execute selected network functions (e.g., NAT). Second, it can deploy a service by partitioning the original service graph into multiple subgraphs, each one instantiated in a different domain, dynamically connected by means of traffic steering rules and parameters (e.g. VLAN IDs) negotiated at run-time.**

I. Introduction

End-to-end service deployment of Network Functions Virtualization (NFV) services usually involves two levels of orchestrators [1], [2]. As shown in Figure 1, an *overarching orchestrator* (OO) sits on top of many possible heterogeneous technological domains and receives the end-to-end service request as a service graph, which defines the involved VNFs and their interconnections. This component is responsible of *(i)* selecting the domain(s) involved in the service deployment (e.g., where NFs have to be executed), *(ii)* deciding the network parameters to be used to create the proper traffic steering links among the domains, and *(iii)* creating the service *subgraphs* to be actually instantiated in above domains. The bottom orchestration level includes a set of *Domain Orchestrators* (DO), each one handling a specific technological domain and interacting with the infrastructure controller (e.g., the OpenStack [3] cloud toolkit in data centers, the ONOS [4] or OpenDaylight (ODL) [5] controller in SDN networks) to actually instantiate the service subgraph in the underlying infrastructure. In addition, DOs export a summary of the computing and networking characteristics of the domain, used by the OO to execute its own tasks. Notably, DOs simplify the integration of existing infrastructure controllers in the orchestration framework, because any possible missing feature is implemented in the DO itself while the infrastructure controllers are kept unchanged.

Existing orchestration frameworks (e.g., [6]) present the following limitations. First, they exploit SDN domains only to create network paths, neglecting the fact that SDN controllers can actually host many applications (e.g., firewall, NAT) that program the underlying network devices according to their own logic. Second, they do not take care of automatically configuring the inter-domain traffic steering to interconnect portions of the service graph deployed on different domains. For instance, this would require to properly characterize subgraphs endpoints (called Service Access Point, or SAPs) with

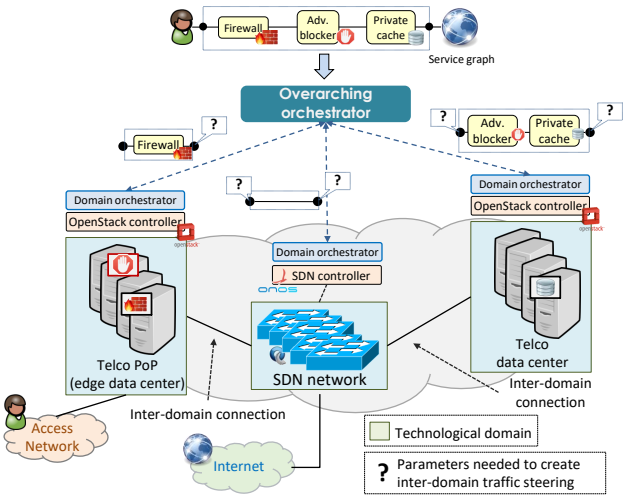

Fig. 1. Service graph deployment in a multi-domain environment.

the proper network parameters, thus replacing the question marks in the subgraphs shown in Figure 1 with the proper information such as VLAN IDs, GRE keys and more, based on the capabilities of the underlying infrastructure.

This paper overcomes the above limitations by proposing an orchestration framework that *(i)* can *transparently instantiate NFs wherever they are available (e.g., either on cloud computing or SDN domains)*, and *(ii)* that enables the OO to enrich the service subgraphs with *information needed for DOs to automatically set up the inter-domain traffic steering*.

Particularly, the paper presents the OO, and details the architecture and implementation of two open source DOs that deploy service graphs in vanilla SDN-based and OpenStack-based domains, namely in SDN networks and data centers. Notably, other domains can be integrated in our orchestration framework as well, provided that the proper DO is created and executed on top of the companion infrastructure controller.

The remainder of this paper is the following. Section II details the domain characteristics exported by DOs and shows how this is used by the OO to execute its own tasks. The OpenStack domain orchestrator is then presented in Section III, while Section IV details the SDN domain orchestrator. Evaluation results are provided in Section V, and finally Section VI concludes the paper.

II. Multi-domain orchestration

In a multi-domain infrastructure, the OO receives from each DO the characteristics of the domain under their responsibility, such as the capabilities and available resources in terms of computing and networking. When a service is requested through its northbound interface, the OO *(i)* selects the best domain(s) that have to be involved in the service deployment, *(ii)* creates the service subgraphs for each of those domains based on information associated by domain themselves, and *(iii)* pushes the resulting subgraphs to the selected DOs (Figure 1). This section presents first *what* and *how* is exported by DOs, and the operations carried out by the OO to implement the service.

A. Exported domain information

Each DO exports a summary of the computing and networking characteristics of the controlled domain according to a specific data model (available at [7]) that has been derived from the YANG [8] templates defined by OpenConfig [9].

From the point of view of computing, we export for each domain the list of NFs it is able to implement (e.g., firewall, NAT). For example, a NF can be a software bundle available in the ONOS controller in case of SDN domain, a specific VM image present in the domain VM repository in case of data center, and more. Notably, DOs advertise neither how NFs are implemented, nor the resources required for their execution. This enables the OO to schedule a given NF on any domain advertising the capability to execute such a function, being it a data center or an SDN network.

From the point of view of networking, DOs export a description of the domain as a "big-switch" with a set of *boundary* interfaces, whose attributes are used by the OO to decide the parameters needed to set up the inter-domain traffic steering, which need to be coordinated among the two domains that terminate the connection. First, the DO advertises whether the selected boundary interface is directly connected with another domain (and, if so, who), with an access network, or the Internet. Second, it advertises a set of *inter-domain traffic steering technologies*, which indicate the ability of the domain to classify incoming traffic based on specific patterns (e.g., VLAN ID, GRE tunnel key), and modify outgoing traffic in the same way (e.g., send packets as encapsulated in a specific tunnel, or tagged with a given VLAN ID, and more). Each inter-domain traffic steering technology is then associated with a list of *labels* (e.g., VLAN ID, GRE key) that are still available and can then be exploited to identify new types of traffic. Finally, other parameters associated with interfaces are inherited from the OpenConfig model, e.g., their Ethernet/IP configuration.

B. Overarching orchestrator

The overarching orchestrator deploys service graphs that consist of service access points (*SAPs*), NFs and (logical) links, as shown at the top of Figure 2. A SAP represents an entry/exit point of traffic into/from the service graph; it may be associated with a specific traffic classifier (i.e., that selects

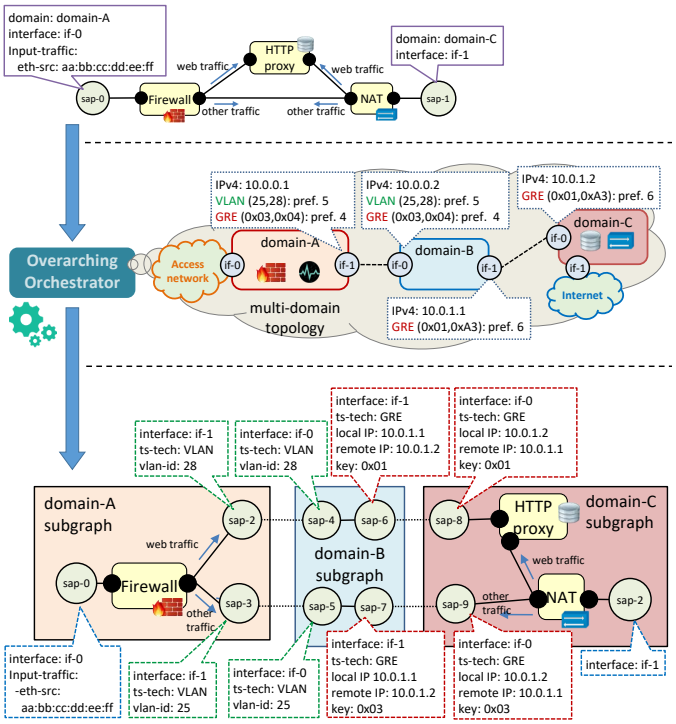

Fig. 2. Placement and splitting of a service graph: sub-graphs are interconnected through traffic steering technologies exported by each domain.

packets that have to enter in the given service graph) and with a specific domain boundary interface that corresponds, e.g., to the entry point of those packets in the multi-domain infrastructure. Links can be associated with constraints on the traffic that has to transit on that specific connection.

In order to deploy service graphs[1], the OO operates on a virtual view of the underlying infrastructure, built using information exported by DOs and consisting in a set of interconnected domains, each one associated with the available NFs, and with the proper characterization of the inter-domain connections (e.g., available VLAN IDs). With this information, the OO selects the domain(s) that will actually implement the required NFs, links and SAPs. To this purpose, we exploit a greedy approach inspired by the hierarchical routing, which minimizes the distance between two NFs/SAPs directly connected in the service graph, in terms of domains to be traversed. Notably, a NF can be executed in all domains that advertise the possibility of implementing that specific NF. Hence, when a given NF (e.g., a firewall) is needed, the OO is enabled to select any domain in which that NF is available, regardless of the fact that such a domain is a data center implementing the NF as a VM, or an SDN network implementing it with an SDN application executed in the controller. Moreover, some SAPs are already associated with specific domain interfaces, and then must be scheduled on that specific domain.

[1]We implemented the steps described in this section in the FROG orchestrator, whose source code is available at [10]. We also developed an open source library to manage service graphs, which is available at [11].

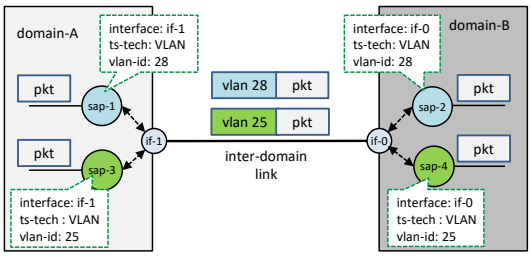

Fig. 3. Inter-domain traffic steering based on endpoints parameters.

As shown in Figure 2, once domains involved in the deployement of NFs have been selected, the OO creates one subgraph per each domain, which includes the NFs and SAPs assigned to that domain and, possibly, new SAPs not present in the "original" service graph. These are originated by links that have been split because connecting NFs (or SAPs) assigned to different domains. Notably, some domains (e.g., `domain-B` in Figure 2) are only used to create network paths between NFs/SAPs that are actually instantiated in other domains; a service subgraph is generated for these domains as well, which just include links between (new) SAPs.

The two SAPs originated from the split of a link must be associated with parameters to be used by DOs to create inter-domain links between them, thus recreating the connection described in the service graph. Then, as shown in Figure 2, the OO associates new SAPs with specific domain boundary interfaces, and with a specific inter-domain traffic steering technology and label (e.g., GRE tunnel based on the key `0x01`) that is available in both interfaces to be connected.

As shown in Figure 3, each DO can configure its own domain (e.g., by instantiating specific flow rules) so that packets sent through a specific SAP are properly manipulated (e.g., encapsulated in a specific GRE tunnel) before being sent towards the next domain through a specific interface. Similarly, this information enables DOs to recognize the traffic received from a specific interface and a specific encapsulation as entering from a given SAP. Notably, packets should be tagged/encapsulated just before being sent out of the domain, while the tag/encapsulation should be removed just after the packet is classified in the next domain.

III. OpenStack Domain Orchestrator

This section presents our OpenStack Domain Orchestrator (OS-DO) that enables the instantiation of service (sub)graphs in cloud computing environments controlled by the vanilla OpenStack controller. Source code is available at [10].

A. OpenStack overview

As shown in Figure 4, an OpenStack domain includes the OpenStack infrastructure controller that manages: *(i)* high-volume servers, called *compute nodes*, hosting VMs (or containers); *(ii)* a network node hosting the basic networking services and representing the entry/exit point for traffic in/from the data center network; *(iii)* an optional SDN controller, which is mandatory in our scenario; *(iv)* other helper services

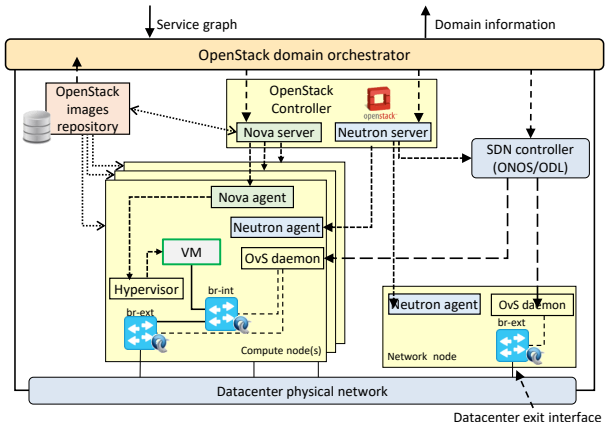

Fig. 4. Architecture of the OpenStack domain.

such as the VM images repository. Each compute node includes two OpenFlow-enabled virtual switch instances (Open vSwitch [12] in our scenario): `br-int`, connected to all the VM ports running in that specific compute node, and `br-ex`, connected to the physical ports of the server. Servers are connected through a physical network that it is not necessarily under the control of the OpenStack controller.

This paper focuses on the two main components of the OpenStack controller. Nova, the former, takes care of handling the lifecycle of VMs (e.g., start/stop) in the compute nodes, while Neutron, the latter, is responsible for managing the vSwitches (e.g., create ports, instantiate flow rules), in our case through the SDN controller. Particularly, Neutron programs the forwarding tables of the vSwitches in order to create *virtual networks* between VMs ports, which implement a broadcast LAN and may include basic services (e.g., DHCP and routing) deployed on the network node.

B. Discovering and exporting domain information

One of the tasks of the OS-DO is to advertise a summary of the computing and networking characteristics of the underlying domain, which includes the supported NFs (taken from the OpenStack VMs repository) and information about the boundary interfaces. Boundary interfaces are virtual/physical interfaces of the network node, which are responsible for connecting the OpenStack domain to the rest of the world and hence handling incoming/outgoing traffic.

The list of boundary interfaces and the associated parameters (e.g., next domain, available inter-domain traffic steering technologies and labels, etc.) is loaded at bootstrap and exported both at the system bootstrapping and each time a change is detected (e.g., an inter-domain traffic steering technology cannot be used anymore, an interface has been shut down).

C. Deploying service graphs

When receiving a service (sub)graph to be deployed, the OS-DO first checks that the service satisfies specific constraints (that depend on the limitations described in Section III-D), and that the required NFs and inter-domain traffic steering parameters are actually available.

If the graph is valid, the OS-DO interacts with Neutron to define the NFs ports and create one virtual network for each link of the service graph; each virtual network will then be attached to the two NFs ports connected by the link itself. In case of link between a NF port and a SAP, the virtual network is connected only to the NF, because the vanilla OpenStack is not able to attach domain boundary interfaces to such a network. At this point, the OS-DO interacts with Nova in order to start the required NFs; Neutron creates automatically the NF ports defined before, connects them to the `br-int` switch in the compute node(s) where NFs are deployed, and finally instantiates the flow rules needed to implement the required virtual networks.

The OS-DO has to create (*i*) all the links connecting a NF with a SAP and (*ii*) the inter-domain traffic steering on its own; this is done by interacting directly with the SDN controller (ODL in our case) because the vanilla OpenStack (i.e., Neutron) offers limited possibilities to control the in-bound/outbound traffic of the data center (e.g., only through IP addresses), which is not enough to set up the inter-domain traffic steering. To create the link between a NF port and a SAP (which is associated with a domain boundary interface in the network node, and with inter-domain traffic steering information), the OS-DO first interacts with ODL to get the `br-int` switch connected to the NF. Then, through ODL, it creates a GRE tunnel between such a vSwitch and the network node, and sets up the flow rules in order to actually create the connection. At this point, the OS-DO inserts in the network node also the flow rules needed to properly tag/encapsulate outgoing traffic and to classify incoming packets, as required by the inter-domain traffic steering parameters associated with the SAPs.

Finally, by default OpenStack checks that the traffic exiting from a VM port has, as a source address, the MAC addressed assigned to the port itself, in order to avoid address spoofing in VMs. However, in case the VM implements a transparent NF (e.g., network monitor), it sends out traffic generated by other components, and therefore with a source MAC address different from that of the VM port. Then, when creating NF ports in Neutron, the OS-DO configures such a module in order to disable the checks on the above ports.

D. Limitations

Vanilla OpenStack does not complex graphs that require to split the traffic between different NFs (e.g., the web traffic exiting from a firewall has to go to the HTTP proxy, while the rest goes directly to the Internet, as shown in the graph of Figure 2), and neither asymmetric graphs (e.g., traffic exiting from the firewall goes to the HTTP proxy, but not vice versa). In fact, since vanilla Neutron only connects VM ports to virtual LANs, we are forced to use these virtual networks to implement links, resulting in the impossibility to finely split network traffic and to have asymmetric connections. Overcoming this limitation requires a set of deep modifications to OpenStack, as shown in [13], resulting in the impossibility to rely on *vanilla* controllers.

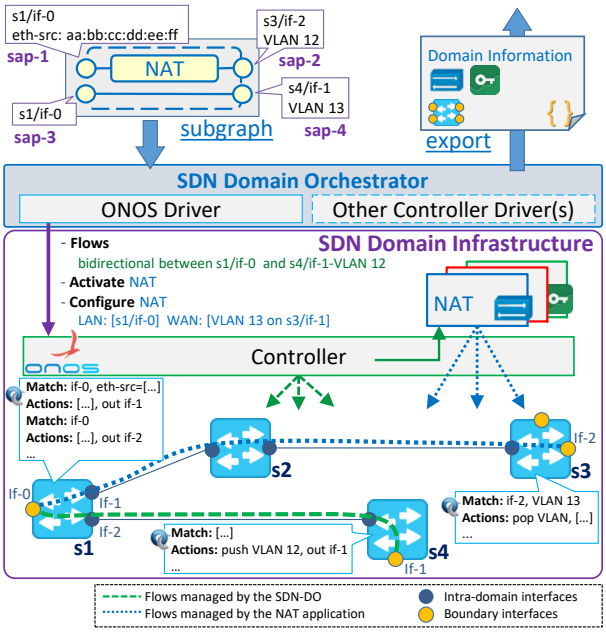

Fig. 5. Service graph deployment in an SDN domain.

IV. SDN Domain Orchestrator

This section details our SDN Domain Orchestrator (SDN-DO) [10] that sits on top of a network domain consisting of OpenFlow switches under the responsibility of a vanilla SDN controller, allowing an OO to instantiate service graphs that include both NFs and links.

As shown in Figure 5, the SDN-DO executes its own tasks (e.g., retrieves the list of NFs to be exported, implements the received service subgraph) by interacting with the SDN controller through a specific driver, which exploits the vanilla REST API exported by the controller itself. At the time of writing, a complete driver for ONOS (Falcon, Goldeneye, Hummingbird and Ibis releases) has been developed, while a partial driver for OpenDaylight (Hydrogen, Helium and Lithium releases) is available, which lacks of the possibility of interacting with the controller to manage NFs; then, in this case the domain can just be used to set up network paths.

A. Exploiting SDN applications as NFs in service graphs

The proposed SDN-DO exploits the possibility offered by widespread SDN controllers to dynamically deploy and run applications in the form of software bundles. In fact these bundles, which implement the logic that decides which OpenFlow rules to deploy in the switches, can implement network applications such as NAT, L4 firewall and more, namely NFs usually executed as VMs running in cloud computing domains.

However, the following main differences exist between NFs implemented as SDN applications (hence software bundles), and NFs implemented as VMs. First, SDN applications usually just process the first packet(s) of a flow, then install specific rules in the underlying switches so that the next packets are directly processed by the switches themselves, while VMs

reside on the data path and hence receive (and process) all packets explicitly. Second, while VMs have in fact virtual ports that can be connected among each other through, e.g., a vSwitch, software bundles do not have ports, then it is not trivial to guarantee that flow rules instantiated by a NF B only operate on traffic already processed by flow rules installed by a NF A, in case B follows A in the service graph. Then, the current version of the SDN-DO prototype only supports graphs with up to one chained NF; support for complex services is a future work[2].

Unfortunately, not all the bundles available in the SDN controller may be used as NFs. For instance, some of them may not actually implement any NF (e.g., the bundle that discovers the network topology) while others, although implementing NFs, may not accept the configuration of specific parameters such as the subset of traffic on which they have to operate, thus preventing the SDN-DO to properly setup graph links. In other words, SDN applications must be extended to be compatible with our architecture; the SDN-DO (details in Section IV-B) looks at specific additional information in the *Project Object Model (POM)* of each ONOS bundle to detect suitable applications.

Finally, SDN controllers usually do not support multiple instances of the same bundle running at the same time, preventing the SDN-DO to deploy the same NFs as part of different graphs; then multi-tenancy, if desired, has to be managed by the application itself and written, with the proper syntax, in the POM file as well.

B. Discovering and exporting domain information

One of the tasks of the SDN-DO is to export information about the domain boundary interfaces and the list of NFs available in the domain. The former information is managed in the same way as the OS-DO. For the latter, we created a new *"app-monitor"* ONOS bundle [14] that retrieves the list of available NFs. In particular, *app-monitor* uses a specific ONOS API in order to intercept the following events: *(i) bundle installed* - a new application is available, which may be used to implement a NF; *(ii) bundle removed* - the application is no longer available and cannot be used anymore to implement NFs; *(iii) bundle activated* - the application is running; however, given that ONOS does not support multiple instances of the same application, that application is no longer available for future services (hence, the SDN-DO must not longer advertise that capability), unless it explicitly supports multi-tenancy; *(iv) bundle deactivated* - the application is no longer used, hence it is available again. Each time the status of a NF bundle changes, the SDN-DO updates the exported information and notifies the OO accordingly.

C. Deploying service graphs

When receiving a service (sub)graph, the SDN-DO first validates the service request by checking the availability of the requested NFs in ONOS and the validity of the parameters associated with the SAPs. Then, it interacts with the network controller in order to start the proper NFs. Graphs links and inter-domain traffic steering information are managed in different ways depending on the fact that the link is between two SAPs, or between a SAP and a NF port.

In the former case (e.g., the connection between `SAP-3` and `SAP-4` in Figure 5), the SDN-DO, through the SDN controller, directly instantiates the flow rules to setup the connection between the endpoints. In addition, it instantiates the flow rules needed to implement the inter-domain traffic steering, i.e., to properly tag/encapsulate the packets before sending them out of the domain, and to classify incoming packets and recognize the SAP they "belong" to.

Instead, if a link connects a SAP to a NF (e.g., the connection between `SAP-2` and the NAT in Figure 5), the SDN-DO configures the application (using the ONOS *Network Configuration Service* [15]) with information about the traffic on which it has to operate, which is derived by the parameters associated with the SAP itself. For instance, the above NAT is configured to operate on specific traffic coming from interfaces `s1/if-0` and `s3/if-2` (i.e., with a specific source MAC address in the former case, with VLAN_ID 12 in the latter), to tag traffic transmitted on `s3/if-2` with the VLAN_ID 12, and to untag traffic tagged with VLAN_12 and received from such an interface. Hence, in this case the inter-domain traffic steering is handled directly by the ONOS application that has to be aware of these parameters, while in case of the OS-DO, VMs ignore completely how they are connected with the rest of the graph.

V. Validation

We validated the proposed orchestration framework by instantiating a service graph consisting of a NAT between an host and a public server, both attached to an SDN network.

Tests have been repeated in two configurations. Initially, (Figure 6(a)), the SDN domain has no available NFs, hence it is exploited only for traffic steering and the NF is deployed as a VM on OpenStack; inter-domain traffic is delivered by setting up the proper GRE tunnels. Second (Figure 6(b)), the SDN-DO exports the capability of running the NAT as a bundle on top of the ONOS controller, hence the OO selects the SDN domain also to execute the NF. In this way, the traffic exchanged between `Host` and `Server` only traverses the SDN domain and the OpenStack domain is not involved at all in the service deployment.

In both cases, we measured the time needed by the orchestration framework to deploy and start the service; results are shown in Figure 7, which breaks the total deployment time in the several steps of the process. As expected, the major contribution to the service deployment is given by the VM startup, while the activation of the SDN bundle is almost immediate. As shown in the picture, we also measured the time between the end of the service deployment from the point of view of the overarching orchestrator, and the time in which `Host` and `Server` were able to communicate. In case of

[2]A similar problem may exist between flow rules instantiated by NFs and flow rules installed by the SDN-DO, e.g., in order to set up the inter-domain traffic steering, as described later in this section.

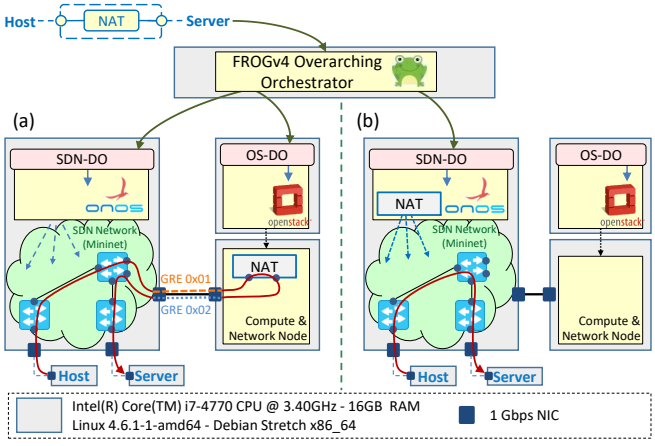

Fig. 6. Validation scenario.

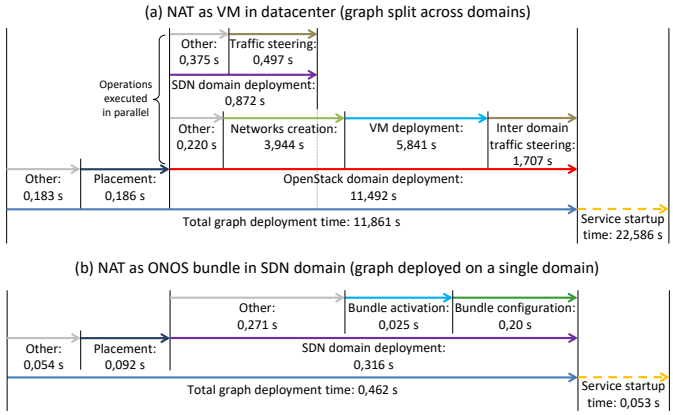

Fig. 7. Time required to deploy the requested service graph (horizontal axis not in scale).

TABLE I
PERFORMANCE AT RUN-TIME OF THE DEPLOYED SERVICES.

	Throughput [Mbit/s]	Avg Latency [ms]
NAT in VM	838	1.041
NAT in ONOS	962	0.465

NAT implemented in a VM, this time is higher because the application can start only after that the bootstrapping of the guest operating system. Instead, in case of ONOS bundle, the application starts immediately.

In both the considered scenarios, we measured the end-to-end latency introduced by the service (using the `ping` command) and the throughput (using the `iperf3` tool to generate TCP traffic). As expected, the throughput is higher when the NAT is deployed in the SDN domain (Table I). In fact, in this case the traffic is kept local to the SDN network, and the NAT is actually implemented as a set of OpenFlow rules installed in the switches (only the first packet of a flow is processed by the application in the controller).

VI. Conclusion

This paper presents an open source orchestration framework based on two layers of orchestrators, which is capable of deploying end-to-end services across vanilla SDN and cloud computing domains. The presented architecture includes a DO sitting on top of each specific infrastructure domain, which exports (*i*) the list of NFs (e.g., firewall, NAT) available in the domain itself and (*ii*) the information associated with the domain boundary interfaces. Our architecture exploits the former information to allow the OO to transparently instantiate NFs both on cloud computing domains and in SDN networks, hence enabling SDN domains to provide richer services that go beyond traditional traffic steering. Instead, the latter are used to enrich the service (sub)graphs with the data required to set up automatically the inter-domain traffic steering, hence enabling to setup highly dynamic services.

The paper details also the implementation of two different DOs: one instantiates services in OpenStack-based cloud environments, the other interacts either with ONOS or OpenDaylight to deploy traffic steering in the SDN network and (in case of ONOS) to execute NFs in the form of software bundles. Other infrastructure domains can be integrated in our framework, provided that the proper DO is created.

References

[1] I. Cerrato, A. Palesandro, F. Risso, M. Su, V. Vercellone, and H. Woesner, "Toward dynamic virtualized network services in telecom operator networks," *Computer Networks*, vol. 92, Part 2, pp. 380 – 395, 2015, software Defined Networks and Virtualization.

[2] B. Sonkoly, J. Czentye, R. Szabo, D. Jocha, J. Elek, S. Sahhaf, W. Tavernier, and F. Risso, "Multi-domain service orchestration over networks and clouds: a unified approach," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 377–378, 2015.

[3] OpenStack. https://wiki.openstack.org/.

[4] ONOS - Open Network Operating System. http://onosproject.org/.

[5] Open Daylight. https://wiki.opendaylight.org/.

[6] G. A. Carella and T. Magedanz, "Open baton: A framework for virtual network function management and orchestration for emerging software-based 5g networks," *Newsletter*, vol. 2016, 2015.

[7] Netgroup @polito. Domain information library. https://github.com/netgroup-polito/domain-information-library.

[8] YANG - a data modeling language for the network configuration protocol (netconf). https://tools.ietf.org/html/rfc6020.

[9] Openconfig. http://www.openconfig.net.

[10] Netgroup @polito. The frog v.4. https://github.com/netgroup-polito/frog4.

[11] ——. Network function - forwarding graph library. [Online]. Available: https://github.com/netgroup-polito/nffg-library

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

[13] F. Lucrezia, G. Marchetto, F. Risso, and V. Vercellone, "Introducing network-aware scheduling capabilities in openstack," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE, 2015, pp. 1–5.

[14] Netgroup @polito. ONOS applications. https://github.com/netgroup-polito/onos-applications.

[15] The network configuration service. https://wiki.onosproject.org/display/ONOS15/The+Network+Configuration+Service.