Introducing Network-Aware Scheduling Capabilities in OpenStack

Introducing Network-Aware Scheduling Capabilities in OpenStack

Francesco Lucrezia*, Guido Marchetto*, Fulvio Risso*, Vinicio Vercellone†
*Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy
†Telecom Italia, Torino, Italy

***Abstract*—This paper motivates and describes the introduction of network-aware scheduling capabilities in OpenStack, the open-source reference framework for creating public and private clouds. This feature represents the key for properly supporting the Network Function Virtualization paradigm, particularly when the physical infrastructure features servers distributed across a geographical region. This paper also describes the modifications required to the compute and network components, Nova and Neutron, and the integration of a network controller into the cloud infrastructure, which is in charge of feeding the network-aware scheduler with the actual network topology.**

Keywords*—*OpenStack, network-aware scheduling, NFV, SDN

I. Introduction

Network-Function-Virtualization (NFV) and Software-Defined-Network (SDN) are emerging as a solid opportunity for the ISPs to reduce costs while at the same time providing better and/or new services. The possibility to flexibly manage and configure highly-available and scalable network services is attractive and the adoption of such technologies is gaining momentum. In particular, NFV proposes to transform the network functions (e.g., NAT, firewall, etc) that today are running on proprietary equipment into a set of software images that could be installed on general purpose hardware, hence becoming Virtual Network Functions (VNFs) and leveraging high-volume standard servers (e.g., x86-based blades) and computing/storage virtualization. A further step is the adoption of such virtualized entities to create arbitrary graphs of VNFs, so that also complex network services could be delivered with unprecedented agility and efficiency, with a huge impact in terms of costs, reliability and complexity of the network. The SDN paradigm, instead, permits a centralised control over the network and a more flexible way to hook the virtual instances manager with the network manager.

In this scenario, a significant problem is to make as easy and inexpensive as possible for service providers the transition from the old to this new concept of networks. This can be done only by trying to take advantage of the general purpose software tools that companies already run on general purpose hardware to provide those new services. In this context OpenStack [1] comes into play as it represents the reference open-source project for creating and managing public and private clouds, although its scope is limited to the datacenter environment and therefore it does not provide all the functionalities required in other deployment scenarios, such as when compute nodes are scattered across a distributed network infrastructure. In fact, the "vanilla" version of OpenStack has three main limitations when applied for our context. First, it does not take the **network topology** into account, assuming simply that the network is fast enough to support the workload, which is not always the case when computing resources are distributed across the entire infrastructure of the telecom operator, from the Customer Premise Equipment (CPE) located at the edge of the network and often connected with slow links (e.g., xDSL) to the main data center. Second, it does not consider the **relationships among different network functions**, e.g., as service chains are (by definition) sequences of possibly dependent VNFs, often traversed by huge amount of traffic (I/O bound workloads). Third, it does not consider the necessity to support **arbitrary traffic paths** between the network functions belonging to each service graph, which requires the availability of generic traffic steering primitives.

In our context scenario, where physical links may have limited bandwidth and servers are far away from each other, a much more careful consideration of network resources is essential. Instead, the OpenStack scheduler, which in charge of properly determining the placement of virtual machines (VMs) through the *Filter&Weight* algorithm, schedules VNFs (actually, VMs) without considering either possible interaction among them or network-related parameters characterizing the underlying infrastructure. We argue that knowing the physical network resources (e.g., network topology, link capacities and load conditions), VM placement can be optimized also from a networking perspective, e.g., in a way that minimizes the load over physical cables and reduce the network latency.

This paper investigates to what extent OpenStack needs to be modified to support the deployment of network service graphs in the NFV context. In particular, we discuss the extensions required to introduce a network-aware scheduler, i.e., with the capability to optimize the VM placement from a networking perspective, which is essential for the efficient deploying of VNF service graphs. As an additional feature, we also extend OpenStack with traffic steering primitives, which are used to instruct the traffic to follow arbitrary paths among VNF, allowing packets to traverse network functions without the need of explicit IP routing rules. Traffic-steering is necessary to implement arbitrary service graphs featuring transparent VNFs, i.e., applications that act as a pass-through component in the service graph, such as a transparent firewall, a network monitor, and more, and to allow the traffic exiting from one VM to be split and sent toward different paths (e.g., web traffic to a first VNF, while the rest to a second VNF). All these results have been achieved by modifying the two core components of OpenStack, namely the compute part (a.k.a., Nova) and the network part (Neutron); in addition, we also require the deployment of a separate network controller — OpenDaylight (ODL) [2] in the specific case — to collect network-related parameters, such as the network topology, and communicate them to the scheduler.

The rest of the paper is organized as follow: Section II

describes the modification applied to OpenStack, in Section III the experimental validation is presented, while related and future work are discussed in Section IV and V respectively.

II. Extended OpenStack Architecture

A. Enabling a Network-Aware Scheduling

The location of the several information required by a network-aware scheduler to operate is shown in Figure 1.

Briefly, we need to feed the OpenStack scheduler with the service graph, which is composed by a set of VMs (item (*i*) in Figure 1) and the logical interconnections between them (item (*ii*) in Figure 1). However, this information is either missing, or scattered across different OpenStack components. With respect to the missing information, Nova schedules one VM at a time and does not have the notion of "group of VMs", which is required in order to be able to activate the scheduling task only when all the components of the service graph have been communicated to OpenStack. This requires the Nova API to be extended with a new function that defines a group of VMs, and forces Nova to call the scheduler only when all the VMs belonging to that service graph are known. With respect to the information that is present in OpenStack but is not known by Nova, we can cite the logical interconnections between VMs that are known only by Neutron, which requires the definition of a new function that allow Nova to get access to this information.

In addition to the service graph, the scheduler needs to know the network topology and status (item (*iii*) in Figure 1), which is known by the ODL network controller[1]. This requires the definition of a function that reads the network topology from ODL, and a last component (item (*iv*) in Figure 1) that maps the location of each softswitch returned by ODL to the proper server it is running in.

Given those requirements, OpenStack had to be modified in several places, as depicted in Figure 2, where the circles represent the components that had to be extended or modified in order to support our use-case. The internal calls sequence is depicted in Figure 3 with numbers as identifiers of each call. In the following section we will refer to such calls with the proper identifiers.

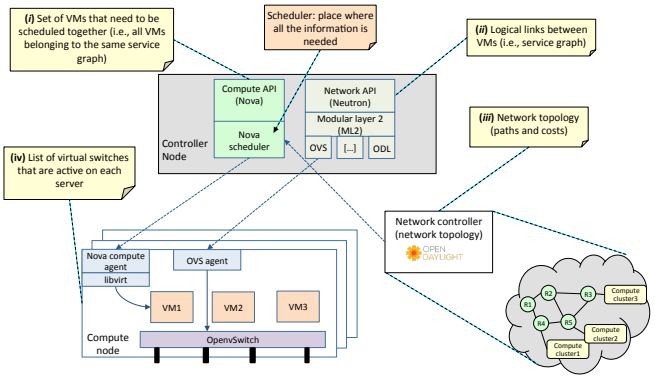

Fig. 1: Location of the information required by a network-aware scheduler in the OpenStack architecture.

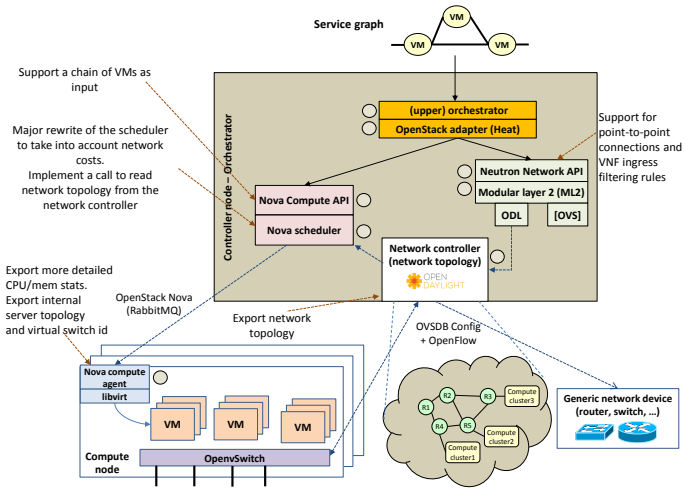

Fig. 2: Overall OpenStack architecture: modified components.

B. OpenStack call sequence

When the OpenStack orchestrator, Heat, receives a call for a chain deployment (label 1 of Figure 3), a set of API are called for Nova and Neutron. In particular, the orchestrator requests Nova to deploy and start each VNF of the chain (2, 4). Nova will retain the requests until the last one is received. A new API has been introduced in Nova and called by Heat to specify constraints such as minimum amount of bandwidth required in the connection between two VMs of the chain (5). Once the Nova module has the set of VMs associated to their ports, it can recreate the original graph retrieving from Neutron the way virtual machines have to be interconnected leveraging, in particular, on the call `get_flowrules` (6). This way Nova scheduler has a model of the complete service chain that has to be deployed comprehensive of VMs, links and associated constraints. At this point, it is needed to provide to the scheduler a map of how the physical servers are connected and the links capacities. In order to do so, a call to ODL to retrieve the physical topology is done with the provided API (8); the topology is composed by nodes (virtual or physical switches) and links with associated bandwidth capacities. The Nova scheduler translates and compresses all the information into a cost matrix in which the elements represent the min-cost path values (in terms of bandwidth or number of hops) between each softswitch residing in the different Nova compute nodes[2]. The identifiers of these virtual switches are retrieved from the Nova-agents. With all these information Nova is now ready to solve the scheduling problem. From the *Filter&Weight* algorithm of the legacy OpenStack we kept the first phase of filtering in order to have as input of the problem only machines compliant (in terms of type of OS, virtualization flavors and possibly policy-rules) with the particular VMs that are going to be scheduled.

The new Nova scheduler defines an objective function that minimizes the traffic load on the physical links, which is

[1]It is worth mentioning that the standard OpenStack installation does not require a distinct network controller; in this case, Neutron will control only software switches in the compute node, which are connected together through a mesh of GRE tunnels.

[2]Physical network devices operating in the network are not relevant for the Nova scheduler, as this component needs to know the *cost* between a VM to another, no matter how the physical path looks like.

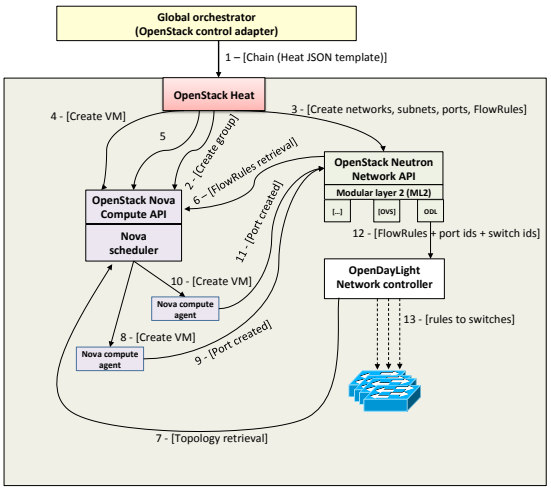

Fig. 3: OpenStack internal calls sequence.

weighted inversely to the bandwidth capacity of the links (i.e. $w_l = k/C_l$ where k is a constant and C_l the link capacity). In this paper, the definition of an efficient algorithm for the Nova scheduler was not among the objectives, hence we use a brute force enumeration algorithm in which all the possible feasible solution are checked in order to have the optimum allocation. Clearly, for a deployment in a real scenario a meta-heuristic needs to be exploited and its complexity analyzed in order to have a solution that can cope with hundreds of variables involved. For example, the algorithms presented in [3] seem to be a good starting point for a real implementation in OpenStack. This is left as a future work.

C. Traffic steering

Heat has been also extended to support the new Neutron primitive **FlowRule** (3), which denotes the logical link between two VMs. The FlowRule describes how to steer the traffic between the ports of the VNFs, which is required to support *traffic splitting* among different network functions, e.g., when the traffic exiting a first VNF (e.g., a software router) has to be directed party to a second VNF (e.g., a stateful firewall for web traffic), partly to a third VNF (e.g., a stateless firewall for the rest)[3]. This primitive provides an interface similar to the OpenFlow 1.0 **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. In this way, only when Nova module has scheduled the VMs and the involved ports attached to the virtual switches become `ACTIVE` (10, 12), the FlowRules are passed to the network controller (13), which has been modified to accomplish the task of translating such rules into FlowMods and to instantiate them into the switches along the minimum-cost paths between servers where the VMs are allocated (14). To accomplish this, Neutron has to pass the identifiers of the switches chosen by Nova after the scheduling process so that ODL can determine the end-points of each FlowRule. Note that the switches involved could be either inside a Nova compute node or physical switches used to interconnect several servers, but only those inside the Nova-agents could be the end-points of the FlowRules.

Neutron has been modified from the northbound interface to the mechanism driver of ODL to support the new RESTful API URL and the basic CRUD (create-read-update-delete) methods to permit the user or other components to manage the FlowRules. Other than a new URL, this piece of code takes care of providing a definition of the request via a XML schema or a JSON structure (shown in listing 1) as well as instructing the below component to redirect each request to the proper corresponding entry point in the Modular Layer 2 (ML-2) plugin component. The implication of the network controller automatically enables traffic steering along VNFs belonging to the same chain with simple OpenFlow rules derived from the FlowRule abstraction.

```
POST http://neutronIP:neutronPort/...

{
  "flowrule":
  {
    "id": "<tenant_id><unique_name>",
    "hardTimeout": "0",
    "priority": "50",
    "ingressPort": "<VM_port_id_1>",
    "etherType": "0x800",
    "tpSrc": "80",
    "protocol": "TCP",
    "actions": "OUTPUT=<VM_port_id_2>"
  }
}
```

Listing 1: JSON request example performed to the *FlowRule* Neutron API extension

D. Minor modifications

In order to create an environment where our scheduler can be actually deployed and tested, OpenStack (in its IceHouce release) requires some further modifications. First of all, (*i*) since a VNF may be a transparent function that hence needs transparent ports (i.e. network ports without the TCP/IP stack enabled), we introduced an emulation of this "port type" by exploiting the above described traffic-steering feature. Furthermore, (*ii*) we removed the constraint for all the traffic entering and leaving its domain to pass through the Network Node, the traffic concentrator point used in OpenStack. This is crucial in a distributed environment where a frugal usage of the bandwidth available on physical interconnections is essential. This feature is obtained because deployed chain has its own forwarding plan defined and instantiated through ODL instead of Neutron.

III. Experimental validation

A. Testbed setup

Figure 4 shows the testbed used to validate our network-aware scheduler, which is composed by four physical server, one hosting the orchestrator (OpenStack server and OpenDaylight controller) and the remaining acting as compute nodes; in addition, two workstations were used as traffic generator and receiver. All the servers were equipped with 32 GB RAM, 500 GB Hard drive, Intel i7-3770 @ 3.40 GHz CPU (four cores plus hyper-threading) and 64 bits Ubuntu 12.04 server OS, kernel 3.11.0-26-generic. Virtualization were provided by the standard KVM hypervisor. All the physical connections were

[3]Traditional service chains without traffic splitting can be supported by the "vanilla" Openstack, without the need for traffic steering extensions.

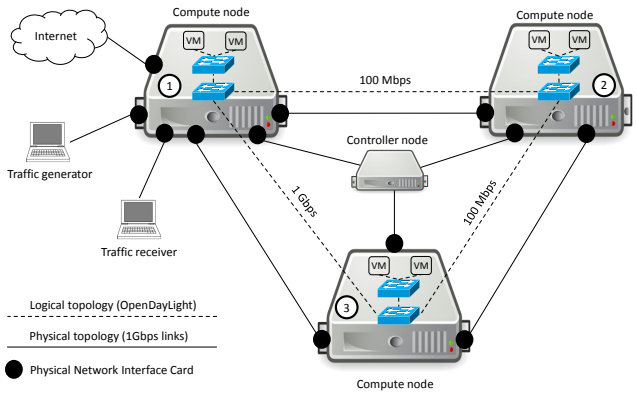

Fig. 4: Testbed: physical and logical setup.

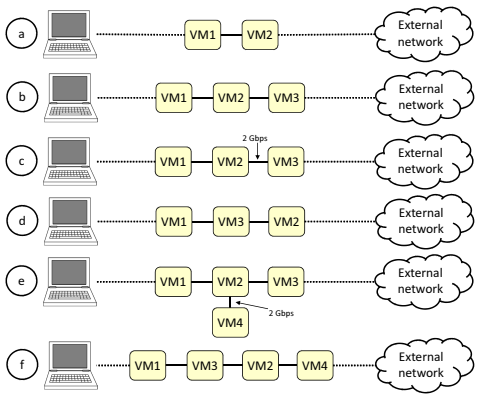

Fig. 5: Chains of VMs used for testing.

point-to-point Gigabit Ethernet links; furthermore compute servers were logically connected through a full mesh network made by three point-to-point GRE tunnels; the network seen by the OpenFlow OpenDayLight controller is shown with dashed lines. In order to demonstrate the advantages of our scheduler, the bandwidth of two (logical) links was artificially lowered to 100Mbps.

B. Test conditions

Since our test aim at validating the advantages of the new scheduler, we choose a rather conventional setup for the parameters that do have a major impact on the results. Particularly, we allocated fully virtualized virtual machines (VMs), with Ubuntu 12.04 server-64 bits as guest operating system. Each VM was configured with two virtual interface cards bridged with the standard `linuxbridge` tool, configured to forward the traffic received from a NIC to the the other as fast as possible. Finally, we configured OpenStack to limit to two the number of virtual machines that can be executed in each compute node.

To validate the new architecture, we asked OpenStack to create and schedule the six different chains of VMs shown in Figure 5 under different settings. First, we used the standard OpenStack scheduler (the *Filter&Weight*) declined in two different configurations: *distribute*, which distributes VMs across all physical servers, or *consolidate*, which needs to reach the 100% load on a given server before allocating a VM on the next server. As evident, neither configuration take the network topology or the shape of the service chain into consideration, as the weighing process analyzes only the amount of memory available on each server. VMs are scheduled according to their identifier, starting with VM1 first.

Finally, the network was configured to force the test traffic to traverse all the VMs of each chain in the proper order, starting from the workstation acting as traffic generator toward the traffic receiver. Latency was measured by issuing `ping` requests between the traffic generator and receiver, while the throughput of the chain was measured with `iperf`, configured to generate TCP traffic.

C. Results

Results, shown in Table I, confirm that in the *Filter&Weight* scheduler algorithm (in both configurations) the sequence of requests is crucial when choosing the location where virtual machines will be deployed. In particular, taking a look at scenarios (b) and (d) that features the same set of VMs although chained (and then scheduled) in a different order, it is evident the standard OpenStack scheduler (in both configuration) allocates the VMs in the same way, despite the different logical connections among the VMs of the chain. Instead, our scheduler optimizes the placement of the VMs taking into account both the connections among VMs and the characteristics of the network, hence (*i*) allocating consecutive VMs on the same server (VM1 and VM2 in case (b), VM1 and VM3 in case (d)) and (*ii*) favoring Server3, connected with a 1Gbps link, against Server2, connected with a 100Mbps link. In this particular example, this allows our traffic generators to measure a throughput that is one order of magnitude higher than the original scheduler.

Our scheduler is also able to satisfy possible requirements in terms of minimum bandwidth among VMs. For instance, scenarios (c) and (e) impose some limitations about the bandwidth that must be available between VMs, which forces the two involved VMs to be allocated on the same server. In fact, the requested bandwidth (2Gbps) is higher than the speed of any physical link, hence the only choice consists in allocating both VMs on the same server where the (virtual) link speed is assumed to be the bandwidth of the memory (25.6GBps in our servers).

Our network aware scheduler shows a possible drawback when analyzing the time needed to take its decision, which increases dramatically as the number of compute nodes and the size of the network increases, as shown in Table II. This suggests that the brute force algorithm should be replaced with a more lightweight heuristics in our future work, which looks more appropriate to solve a problem of class NP.

IV. Related Work

The necessity of a network-aware scheduler has been discussed in the OpenStack community [4], but no work has been done ever since the implementation of the blueprint is marked as “deferred”. An example of extended OpenStack architecture for a dynamic resource allocation scheme can be found in [5], where the authors provide an extensible set of management objectives among which the system can switch at runtime during the process of resource allocation for interactive and computationally intensive applications. However, they do

	Algorithm	Server #1	Server #2	Server #3	Sched. time [s]	Thr [Mbps]	Lat. [ms]
a	Consolidate	VM1, VM2			0.242	840	1.16
	Distribute	VM1	VM2		0.236	82	1.89
	Network-aware	VM1, VM2			0.0008	840	1.16
b	Consolidate	VM1, VM2	VM3		0.310	79	2.19
	Distribute	VM1	VM2	VM3	0.325	60	3.50
	Network-aware	VM1, VM2		VM3	0.0011	748	2.14
c	Consolidate	VM1, VM2	VM3		0.312	79	2.20
	Distribute	VM1	VM2	VM3	0.323	60	3.51
	Network-aware	VM1		VM2, VM3	0.0014	760	2.15
d	Consolidate	VM1, VM2	VM3		0.312	77	2.41
	Distribute	VM1	VM2	VM3	0.320	58	3.50
	Network-aware	VM1, VM3		VM2	0.0011	750	2.12
e	Consolidate	VM1, VM2	VM3, VM4		0.392	76	2.57
	Distribute	VM1, VM4	VM2	VM3	0.399	59	4.02
	Network-aware	VM1, VM3		VM2, VM4	0.69	750	2.23
f	Consolidate	VM1, VM2	VM3, VM4		0.396	57	2.38
	Distribute	VM1, VM4	VM2	VM3	0.402	47	3.48
	Network-aware	VM1, VM3		VM2, VM4	0.011	683	2.20
Reference - No virtual machines between client and server						864	0.458

TABLE I: Comparing different scheduling algorithms: VM locations, traffic throughput and latency.

# of VMs (# of links)	# of compute nodes 1	2	3	4
1(0)	0.00041	0.00043	0.00043	0.00044
2(1)	0.00047	0.00049	0.00060	0.00073
2(2)	0.00048	0.00053	0.00067	0.00080
3(2)	-	0.0010	0.0016	0.0078
3(3)	-	0.008	0.019	0.19
3(6)	-	0.003	0.54	59
4(3)	-	0.0017	0.0092	0.031
4(4)	-	0.017	0.26	1.5
5(4)	-	-	0.47	15
5(5)	-	-	5.1	75

TABLE II: Processing time of the "brute force" scheduler (seconds).

not cope with NFV deployment, which is instead the central topic in [6], [7], [8] and [9]. In [6], the authors propose modifications to the Nova-scheduler of OpenStack to solve a joint optimization problem of the NFV Infrastructure resources under multiple stakeholder policies. They introduce constraints for the NFV deployment related to QoS, fault-tolerance and network topology redundancy, but they do not discuss the interaction between OpenStack and the network controller. [7] proposes an architecture for NFV by leveraging on a set of OpenFlow enabled physical switches programmed and managed by a centralized controller, which provides an interface for the deployment of fine-grained traffic steering policies. In our implementation, the flow rules for steering come from the cloud controller through Neutron which in turn delegates the instantiation to the network controller. [8] presents an end-to-end platform for NFV that lays on OpenStack and OpenDaylight as our extension does. However, they focus on the high-level description of the framework rather than on the identification of the implementation and the technical issues existing in OpenStack which hinder the capability of NFV deployment. Finally, [9] probably offers the most complete framework for NFV orchestration and deployment: the authors describe an orchestration layer for virtual middleboxes which provides efficient and correct composition in the presence of dynamic scaling via software-defined networking mechanisms. However their implementation is not based on OpenStack and they do not cope with a distributed cloud environment, which is the target scenario for the algorithms presented in [3].

V. Conclusion and Future Work

This paper proposes an extended architecture for NFV and network-aware scheduling in OpenStack. In particular, we extended the already available API of Nova for a bulk scheduling of VMs belonging to the same user chain and introduced a new primitive in Neutron for traffic-steering. This is crucial in an operator network, where xDSL users are connected through a link with limited bandwidth, especially if some compute nodes are located at the network edges, in which our network-aware scheduler shows its main advantages compared to the standard *Filter&Weight* algorithm.

In our scheduler we used a brute force approach to obtain the optimum VM placement. However, the implementation of a meta-heuristic algorithm is necessary to provide a sub-optimal solution in a reasonable amount of time when the number of VMs and servers increases. Other future works concern the capability to support more complex service graphs and a scheduling algorithm that takes into account policy-constraints in order to define administration rules to determine the VM placement. Finally, the network controller of OpenStack should be redesigned to support awareness of dynamic changes in the underlying network.

Acknowledgment

The authors wish to thank M. Tiengo for his work on this subject during his MSc thesis, and the rest of the NFV team that contributed to achieve this result: A. Palesandro, F. Mignini, R. Bonafiglia, M. Migliore, I. Cerrato.

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.

References

[1] "Openstack," http://www.openstack.org/.

[2] "OpenDaylight," http://www.opendaylight.org/.

[3] M. Alicherry and T. Lakshman, "Network aware resource allocation in distributed clouds," in *INFOCOM, 2012 Proceedings IEEE*, March 2012, pp. 963–971.

[4] "Network-aware scheduler in OpenStack," https://blueprints.launchpad.net/nova/+spec/network-aware-scheduler/.

[5] F. Wuhib, R. Stadler, and H. Lindgren, "Dynamic resource allocation with management objectives: Implementation for an openstack cloud," in *Proceedings of the 8th International Conference on Network and Service Management*, ser. CNSM '12. Laxenburg, Austria, Austria: International Federation for Information Processing, 2013, pp. 309–315. [Online]. Available: http://dl.acm.org/citation.cfm?id=2499406.2499456

[6] M. Yoshida, W. Shen, T. Kawabata, K. Minato, and W. Imajuku, "Morsa: A multi-objective resource scheduling algorithm for nfv infrastructure," in *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, Sept 2014, pp. 1–6.

[7] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, and M. Tatipamula, "Steering: A software-defined networking for inline service chaining," in *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, Oct 2013, pp. 1–10.

[8] J. Soares, M. Dias, J. Carapinha, B. Parreira, and S. Sargento, "Cloud4nfv: A platform for virtual network functions," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*, Oct 2014, pp. 288–293.

[9] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella, "Stratos: A network-aware orchestration layer for virtual middleboxes in clouds," in *Open Networking Summit*, April 2013.