How to orchestrate a distributed OpenStack

David Haja[1], Marton Szabo[1], Mark Szalay[1], Adam Nagy[1], Andras Kern[2], Laszlo Toka[1,3], Balazs Sonkoly[1,4]
[1]Budapest University of Technology and Economics, [2]Ericsson Research, Hungary
[3]MTA-BME Information Systems Research Group, [4]MTA-BME Network Softwarization Research Group
{david.haja, marton.szabo, mark.szalay, adam.nagy, sonkoly, toka}@tmit.bme.hu, andras.kern@ericsson.com

***Abstract*—We see two important trends in ICT nowadays: the backend of online applications and services are moving to the cloud, and for delay-sensitive ones the cloud is being extended with fogs. The reason for these phenomena is most importantly economic, but there are other benefits too: fast service creation, flexible reconfigurability, and portability. The management and orchestration of these services are currently separated to at least two layers: virtual infrastructure managers (VIMs) and network controllers operate their own domains, it should consist of compute or network resources, while handling services with cross-domain deployment is done by an upper-level orchestrator. In this paper we show the slight modification of OpenStack, the mainstream VIM today, which enables it to manage a distributed cloud-fog infrastructure. While our solution alleviates the need for running OpenStack controllers in the lightweight edge, it takes into account network aspects that are extremely important in a resource setup with remote fogs. We propose and analyze an online resource orchestration algorithm, we describe the OpenStack-based implementation aspects and we also show large-scale simulation results on the performance of our algorithm.**

I. Introduction

Cloud-edge (or fog) computing [1] and mobile edge computing (MEC) [2] are novel concepts extending traditional cloud computing by deploying compute resources closer to customers and end devices. This approach, closely integrated with carrier-networks, enables several future 5G applications and network services, such as novel Industry 4.0 use-cases, Tactile Internet, or remote driving [3]. Edge resources provide execution environments close to users in terms of latency (e.g., in mobile base stations). By these means, on the one hand, customers' devices can offload computational tasks to this environment instead of consuming their local resources. On the other hand, latency critical functions can also be offloaded from central clouds to the edge enabling critical machine type communication which is required by various envisioned services.

A dedicated component, namely the resource orchestrator (RO), is in charge of finding the proper placement of software components realizing the service. Following ETSI's terminologies on Network Function Virtualization (NFV) [4], the software modules composing the network service are referred to as Virtual Network Functions (VNFs). RO can be considered as a component encompassing orchestration related tasks, and in ETSI's architecture it appears both in the Virtual Infrastructure Manager (VIM) and in the NFV Orchestrator (NFVO). In general, RO assigns VNFs composing the service to compute resources and also allocates paths between connected VNFs.

A novel RO (or a hierarchy of ROs) which can efficiently manage underlying resources in cloud-edge/mobile edge/distributed cloud computing environments is an inevitable future component with challenging tasks. It must be able to jointly handle compute and network resources in a tightly integrated framework and it must be aware of network characteristics besides compute capabilities. Furthermore, the requested network services have to be created on-the-fly within seconds. Two different design approaches can be applied to achieve such features. On the one hand, on top of VIMs and network controllers, a higher level orchestrator, i.e., the NFVO, can be added which is able to integrate different resource domains. This solution results in a hierarchy of ROs and the cooperation of VIMs and NFVO yielding larger deployment time and the need for strictly defined external APIs. However, multi-provider scenarios require this approach [5]. On the other hand, the VIM itself can be extended with network awareness and with the detailed view on network resources. With such an upgrade, the additional NFVO becomes unnecessary for single-provider setups where resources belong to the same operator and by these means, the orchestration and deployment time can be reduced significantly.

Today's most widely deployed open-source VIM is OpenStack, a cloud computing platform, used as the *operating system* of both private and public clouds. OpenStack provides Infrastructure as a Service (IaaS) and it is responsible for the management of large pools of compute, storage and networking resources.

In this paper, we propose a novel extension to OpenStack which makes it a network aware resource orchestrator. More specifically, we define and implement the key building blocks which enable efficient and dynamic orchestration of compute and network resources over distributed cloud environments belonging to a single provider. Our goals are fourfold. First, we give a general model for cloud-fog infrastructure. Second, we propose and analyze an online embedding algorithm, named DARK, making use of greedy heuristics, which solves a generalized version of Virtual Network Embedding (VNE). Third, we implement a proof-of-concept prototype as an extension to OpenStack. And finally, we evaluate the concept via large-scale simulations to confirm the scalability of our approach.

The rest of the paper is organized as follows. In Sec. II, a brief summary on the related work is given. Sec. III introduces our model. Sec. IV is devoted to our novel online embedding algorithm. Our proof-of-concept prototype is described in Sec. V. Sec. VI summarizes our main findings gained from

the large-scale simulations. And finally, Sec. VII concludes.

II. Related work

As NFV technology matures, multiple NFV orchestration solutions have emerged; we list here the most important ones. The OpenSource MANO (OSM) [6] project, hosted by ETSI, targets delivering an open source management and orchestration (MANO) framework that is aligned with ETSI's NFV reference architecture. The project ONAP (Open Network Automation Platform) [7] aims to enable end-to-end service provisioning and orchestration across multiple domains using NFV and SDN technologies combined in a common platform. The NFV-related components of ONAP are conform with ETSI's reference model: all the key elements, such as NFVO, VNFM and VIM can be found in their architecture. The goal of the CORD (Central Office Re-architected as a Datacenter) [8] project is to combine NFV, SDN and commodity datacenter environments to bring the agility of the clouds to the Telco Central Office. Another platform, Cloudify [9], was originally designed to drive orchestration and application deployment in clouds, then the platform was later expanded, thus the Telecom Edition of the Cloudify platform emerged. This combines the NFVO and VNFM functionalities of the MANO architecture.

In OpenStack, *nova-scheduler* is responsible for managing the computational resources on the hypervisor of each physical host. The applied simple placement strategy called filter scheduler [10] is affected by several limitations. For example, the sequential processing of virtual machine requests makes it impossible to define more complex placement constraints that affect more than one instance. Much effort have been made to make *nova-scheduler* more efficient in terms of network-awareness. In [11] an extension is presented that enables a network-aware placement of instances by taking into account bandwidth constraints to and from nodes by keeping track of host-local network resource allocation. In [12] a new filtering step is proposed, that takes into account the actual load (CPU, network I/O, RAM) of the physical node. Authors of [13] discuss the extensions required to introduce a network-aware scheduler: the solution aims to optimize the VM placement from a networking perspective, which is essential for the efficient deployment of VNF service graphs. They used OpenDayLight (ODL) [14] to collect network topology information and to configure traffic steering. The solution tries to minimize bandwidth utilization of the physical links, however the definition of an efficient scheduling algorithm was not in the scope of their work.

VNE (or more exactly, a generalized version of VNE) is the process that maps multiple service graphs (SGs) [15] composed of different VNFs to a common physical infrastructure, represented by the Resource Graph (RG); VNE is known to be $\mathcal{NP}$-hard [16], which means finding the optimal solution cannot be done within reasonable time in case of large input, e.g., many services to be deployed in a large infrastructure. Two different approaches exist to solve the VNE problem: i) exact solutions that find the optimum but these can be applied to limited scale problems only, ii) approximation-based algorithms that trade the optimal solution for better runtime. [17] summarizes many solutions for both.

III. Distributed cloud infrastructures

In this section we summarize two different approaches that can be considered for the orchestration of a distributed cloud system. After that, we describe our resource topology model.

A. Orchestration methods in a distributed cloud environment

Two different solutions can be considered when orchestrating a distributed cloud. In the first case, each computing cluster has its own VIM with the associated RO that manages the resources locally. In order to implement inter-domain resource orchestration functionality, a higher level entity is necessary (e.g., an NFVO), which connects the underlying remote domains in a common view. This results in an orchestration hierarchy which increases the deployment time. The other possible way is to integrate the orchestration logic into the VIM, thus eliminating the need for the upper level orchestrator. It is feasible only for single provider use-cases where the resources are owned by the same operator. In this case, the architecture is simpler, the VIM handles all the internal and external topology information, but only homogeneous technologies can be managed this way. In the first case, the architecture is more complex, well-defined interface definitions are required between the components. The underlying domains may hide some important information about the real features when the abstraction level is not refined enough, but it supports different technology and administrative domains. In our paper, we focus on the second case, thus we provide a novel extension for OpenStack's scheduling algorithm.

B. Topology model

Based on the previously described considerations we created our own network model for the three-tier Edge Computing architecture: a network consists of a given number of edge clusters and central clouds. Each cluster contains a pre-defined number of servers with given computing capabilities (CPU, RAM and storage) and two gateway nodes. Each of them has a SAP (Service Access Point) attached to it via the SAP-Gateway. The SAP works as a connection point to the network. The end devices can consume the remote resources through this interface (e.g., a mobile base station). Within a cluster the nodes are connected in a full mesh topology. The clusters and the central data centers are connected with each other via the core network. A topology may contain any number of central clouds and we assume that they have unlimited compute, storage and memory capacity. The model is able to capture use-cases when the central cloud resources are owned by another operator (e.g., Amazon). Then the service provider has to pay a fee for consumed resources according to a cost model. In the rest of the paper, we assume a single provider using only its own resources. The topology is shown in Fig. 1.

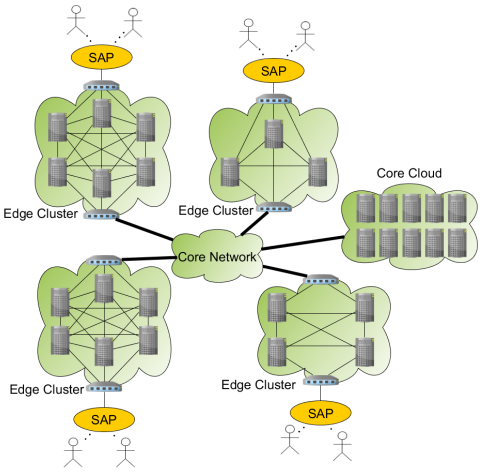

Figure 1. Our resource model for cloud-fog infrastructures

Algorithm 1 Service graph mapping to resource graph

```
1:  running ← copy(RG)
2:  mapped_vnodes ← ∅
3:  map_list ←ORDERSUBCHAINS(SG)
4:  mapped_vnodes.insert(ϱ_s.first)
5:  rollback_level = 0
6:  for all (u, v, link) ∈ map_list do
7:      if (u, v) ∈ mapped_vnodes then
8:          success ← MAPVIRTUALLINK(link)
9:      else if u ∉ ϱ_s then
10:         success ← MAPVNF(u, v, link)
11:     else                          ▷ This means actual_element is a SAP
12:         success ← MAPVLINK2SAP(u, v, link)
13:     end if
14:     if ¬success and rb_level ≥ max_rb then
15:         success ← MIGRATINGEDGE2CORE(cable, u, v)
16:     else
17:         success ← ROLLBACK(u, v, link)
18:         rollback_level+ = 1
19:     end if
20:     if success then
21:         mapped_vnodes.insert(v)
22:     end if
23: end for
```

IV. DARK: OUR ORCHESTRATION ALGORITHM

In this section we propose a novel orchestration algorithm, called DARK, that aims to cope with the new challenges of distributed cloud architectures. VNF migration is an important feature of our approach, which gives option to migrate a given set of network functions to the cloud, thus freeing up network resources in the edge layer in order to serve more latency and bandwidth sensitive service deployment requests. The main steps of our algorithm, taking into account VNF migration costs, are described in the following, and pseudo-code is provided in Alg. 1 and Alg. 2.

A. Service Graph preprocessing

In the first step of our algorithm, we calculate the order of execution. The ORDERSUBCHAINS method splits the incoming service request to the list of triplets containing the links and their connected nodes, i.e., the VNFs. The method starts with the first available SAP and collects all the neighboring nodes and connected edges, then appends the link with the strictest bandwidth requirement to the list together with its endpoints. After that it collects the available nodes and edges that became reachable via the new node.

B. VNF mapping

The next step is the mapping of the service requests to the physical infrastructure. The MAP method iterates through the previously ordered list of edges. Depending on the status of the nodes connected by the link, three different cases are possible. If both ends have already been allocated to a computing resource previously, then only a suitable path for the virtual edge needs to be found. To achieve this we run a Dijkstra algorithm between the hosts in the physical topology.

If one of the end nodes is not in *mapped_vnodes* yet and it is not a SAP either, then the node needs to be mapped. In the MAPVNF method the program tries to find a suitable place for the VNF. First, it filters the available physical nodes based on computing resources, and after that it checks if the candidate is reachable from the previous node via any sequence of edges. If the path does not satisfy the latency requirement, or any of the edges does not have enough bandwidth, then the node is removed from the list of candidates. When the list of compatible nodes is available, they are sorted based on the resource cost of hosting the actual VNF. After the host node is determined, the link can also be mapped with the previously seen method. In that case, when the actual element is a SAP, then the algorithm calculates the path with the lowest latency, where the required bandwidth is available on all edges. If the path fulfills the latency requirement between the previous mapped VNF and the SAP, then the link mapping is performed.

It may occur, that one of the steps above fails. For example, none of the nodes have enough resource to host a given VNF, or the network related requirements cannot be met. In that case, the algorithm tries to step back to a previous state. This step is performed by the ROLLBACK method. In order to ensure better runtime, limiting the number of rollback steps may be necessary. We can do that by setting the *max_rollback* constant to an appropriate value. The ROLLBACK method restores the state when the previous VNF was mapped, then chooses another candidate from the list of the suitable nodes, and continues the mapping from the modified state. If the number of rollbacks exceeds the limit, then the algorithm tries to migrate one or more already mapped VNFs to the central cloud, thus freeing up resources in the edge. The migration process is described in the next section.

C. Migrating VNFs

The MIGRATING method is responsible for the migration of the already mapped VNFs to free compute resources and map the actual VNF. The three arguments are the list of non-delay-sensitive network functions (*migratables*), actual VNF needed to implement (u) and the previous VNF (v) which is connected to the actual VNF and already mapped to a physical node. The migration procedure is the following.

First, it iterates through the list containing migratable functions and checks the compute constraints using ISBIGGER

Algorithm 2 VNF migration

```
1: procedure MIGRATING(migratables, u, v)
2:     mig_vnf_try = 0
3:     for migratable_vnf ∈ migratables do
4:         if ISBIGGER(migratable_vnf,u) then
5:             if mig_vnf_try < max_vnf) then
6:                 poss_nodes ← GETCOMPNODES(migratable_vnf)
7:                 mig_try = 0
8:                 for n ∈ poss_nodes do
9:                     if mig_try < max_try then
10:                        backup_rg ← GETRUNNINGRG()
11:                        backup_sgs ← GETMAPPEDSGS()
12:                        success ← TRYMIGRATE(n,migratable_vnf,u,v)
13:                        if success then
14:                            return True
15:                        else
16:                            RESETRG(backup_rg)
17:                            RESETSGS(backup_sgs)
18:                        end if
19:                        mig_try+ = 1
20:                    end if
21:                end for
22:                mig_vnf_try+ = 1
23:            else
24:                return False
25:            end if
26:        end if
27:    end for
28:    return False
29: end procedure
```

method. If the migratable VNF reserves more compute resource (CPU, RAM, and storage) than the requirements of the actual VNF, then the actual VNF would be mapped to physical node if it did not contain that migratable VNF. If it is true, the GETCOMPNODES method returns the physical nodes which contain sufficient free resources for the migratable VNF. The second part of the method iterates through these possible physical nodes.

Secondly, the TRYMIGRATE method tries to execute the migration process, which means the migratable VNF is removed from the original physical node and placed in the possible target node, in addition the connected *vlinks* are reconfigured to use another physical link path between the VNFs. Furthermore the method maps the actual VNF to the original server where the migratable one was moved from, and determines the physical links which will implement the virtual link between the actual and the previous VNF. So far only the compute constraints of the VNFs have been checked, however, the network requirements can fail during the mapping.

Finally, the method checks in each iteration if the migration was successful. If the remapped physical paths fulfill the corresponding virtual link requirements, then the TRYMIGRATE method returns true, thus the procedure of migrating was successful. Otherwise, in RESETRG and RESETSGS we restore the previous state of the resource and the already mapped service graphs, and continue with the next migration option from the list. Instead of checking all the possible migration options (migratable VNFs and possible physical nodes), in order to reduce runtime we only test a sufficient amount of them. This iteration number can be controlled by defining the value of *max_try* and *max_vnf* environment values.

The computational complexity of the algorithm is polynomial (for further details, see [18]).

V. Implementation - Proof of concept prototype

As previously presented, our DARK algorithm gives us a service graph mapping to our internal topology, which defines the exact resources where the virtual service components should be executed. As a proof of concept, we substituted the default *nova-scheduler* of OpenStack with our own algorithm, in order to support network-aware VNF placement. This prototype is also able to run automated measurements to determine the physical network characteristics. In this section, we describe the required improvements that we have made.

A. Network status measurement

Since currently OpenStack does not provide any network related metrics, it cannot take them into account during the orchestration process. To solve this problem we implemented a measurement method which is conform with our previously introduced network model. Our tool is based on VMTP [19] which is a data path performance measurement module for OpenStack. It performs automatic measurements between the different virtual networks, but can also be used to benchmark native hosts. It connects to the given nodes via SSH, executes the measurements by using the selected protocols (TCP, UDP, ICMP), then returns the result to the management server.

Each OpenStack compute host in our reference cloud is configured to belong to one custom Availability Zone. An Availability Zone may represent an edge cluster or a central cloud according to our terminology. As we assume that the servers that are located in the same cluster are deployed physically close to each other (e.g., in the same rack), it is enough to measure the latency and bandwidth values between 1-1 selected servers in each zone. By applying this method we can construct the delay and bandwidth matrices that describe the parameters of the underlying physical network.

Furthermore, through the OpenStack API we also collect the available compute node resources (CPU, RAM, storage) from each hypervisor. From the previously gathered information, we can build up the Resource Graph, that contains the compute resources extended with the networking related features.

B. OpenStack scheduler algorithm and modifications

OpenStack's physical resource orchestrator component is Nova, which uses its filter scheduler for filtering and weighting to make informed decisions on which compute node a new instance should be created. During the virtual machine placement *nova-scheduler* iterates over all compute nodes, evaluates each of them against a set of filters. The list of resulting host is sorted by the administrator-defined weights. This default filtering operation cannot deploy properly our virtual services because there is no standard filter class that tackles the network resources (delay, bandwidth) between infrastructure nodes. With Nova API it is possible to deploy a virtual machine on a manually specified host. In our prototype we use this possibility for executing our VNFs on the host given by our resource orchestrator algorithm.

The next step in the prototype's workflow is ensuring correct traffic steering. OpenStack's Pike release officially supports

Figure 2. Service Graph model of the generated services

network traffic steering with Neutron port chains provided by the *networking-sfc* [20] module. Our code uses Neutron API to create an ingress and an egress port for each virtual machine. These ports are grouped into port pairs by the owner virtual machine. The port pairs are grouped into Neutron port pair groups by the virtual link connections. A port chain consists of a set of Neutron port pair groups to define the sequence of service functions. These Neutron objects make it possible to deploy our traffic steering model for service chaining that uses only Neutron ports.

C. Challenges and limitations of OpenStack

If the delay is too high between the controller and the compute nodes, then the controller cannot execute the commands properly on compute nodes. We conducted several experiments with emulated delay between the controller and the compute nodes. We could successfully deploy virtual machines even with extreme, 10 seconds, delay value, naturally the spawning method took couple of minutes till we got our VM in active state. Consequently we assume that OpenStack compute service is able to work in a distributed environment which has relatively high delay among the nodes.

We used OpenStack's latest (at the time of writing this paper) release, i.e., Pike, because the official Neutron package support for the service function chaining first appeared in that one. Our orchestrator algorithm considers the possibility of migrating VNFs between compute nodes, therefore we implemented the migration API calls in our prototype code. Migrating VMs between compute nodes is not trivial, if there are different types of CPUs in the compute nodes. In this case we have to make modifications in Nova's configuration files to avoid any issues. Note that to the best of our knowledge there are no solutions to live-migrate VMs between different OpenStack clouds, hence the importance of federating computing clouds and fogs under one VIM. Furthermore, as only one controller is needed in this case, our proposed setup does not take hardware resources for controlling purposes from the limited capacity edge compute nodes.

VI. Large-scale simulation results

We have run measurements in a simulator to demonstrate how our algorithm performs in different topology setups with various requests. The simulated three topology setups contained different number of edge nodes, more precisely 10, 50, 100. We determined three types of network service (*NS1, NS2, NS3*) and each of them contained a service access point, and two VNFs in a logical service chain shown in Fig. 2. The differences between them were in the latency constraints of the virtual links, which are shown in Table I.

We generated three scenarios of service request sequences. Each of them included different percentages of NS1, NS2 and NS3 which can be seen in Table II. We used these service request sequences as inputs for the proposed algorithm for mapping them into the generated topology models. During our simulations we posted each network service (included in a service sequence) one by one, and counted the number of deployed service requests in which the mapping process fulfills all the compute and network constraints. We indicated failure when there was no way to deploy a service request to the remaining available resource set, or in the case of *nova-scheduler* algorithm, if the deployment did not satisfy the service's network requirements. Each simulation contained 1000 service requests and we differed three type of topology with 10, 50, 100 edge nodes.

Table I
Network and compute requirements in request sets

		NS1	NS2	NS3
VLink1	delay (ms)	8 - 10		80 - 150
	bandwidth (Mbps)	100 - 500		
VLink2	delay (ms)	8 -10	80 - 150	
	bandwidth (Mbps)	100 - 500		
VNF1 & VNF2	CPU	1 - 4		
	RAM (GB)	1 - 8		
	Storage (GB)	10-100		

In Fig. 3 we demonstrate our obtained simulation results. The incoming service requests arrived continuously one-by-one while after every mapping we logged the number of successfully mapped services. The examined strategies included our proposed orchestration algorithm, the simulated OpenStack's *nova-scheduler* and an extended version of the latter called *nova smart*, which takes into account the end-to-end delay constraints.

The simulated OpenStack's *nova-scheduler* has significantly lower performance than others, in each scenario, on each topology. This is not a surprising result since the original scheduler operation does not take into account the network requirements of the services, therefore after each iteration the mapping solution will meet the constraints only randomly. The more edge nodes are available, the greater the chance *nova-scheduler* selects from those that do not meet the network constraints. That is why it provides a performance close to zero in case of 50 and 100 edge nodes (green lines in Fig. 3).

In case of Scenario 1, when the required services do not contain many delay-sensitive virtual links, the proposed orchestration algorithm DARK has obvious advantage over the other solutions: in the absence of free resources on a node, it migrates VNFs to another server. When the services contain more delay-sensitive virtual links, then the migration

Table II
Measurement scenarios

	NS1 (%)	NS2 (%)	NS3 (%)
Scenario1	10	10	80
Scenario2	80	10	10
Scenario3	10	80	10

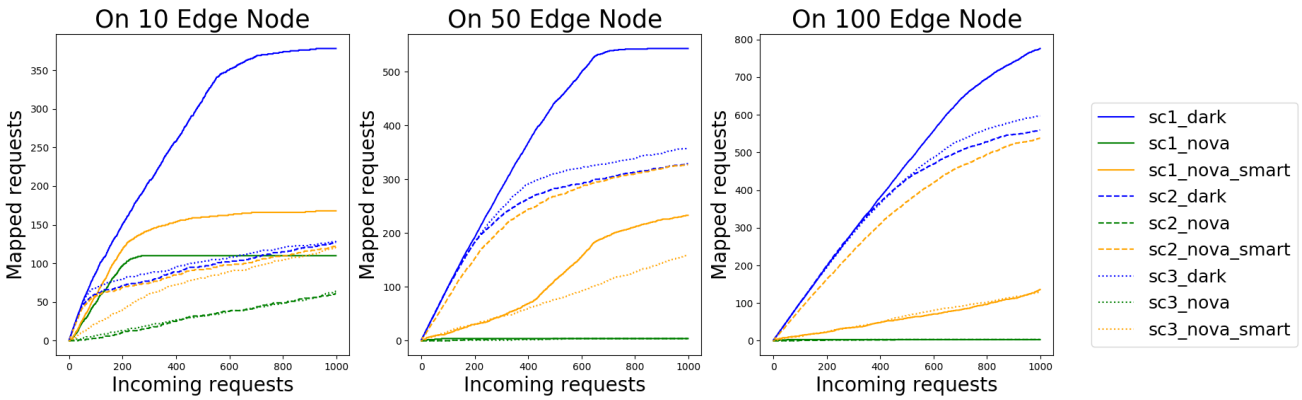

Figure 3. Number of mapped service requests depending on incoming requests

of VNFs is rarely possible anymore. As a consequence, DARK is not able to free compute resources in order to offer enough room for incoming services. That is why the performance difference is smaller between DARK and *nova-scheduler* in case of Scenarios 2 and 3, where the network services contain more delay-sensitive links than in Scenario 1.

Finally, one would ask why *nova smart* performs better in Scenario 2 as we increase the number of edge nodes. *Nova smart* filters the nodes based on the end-to-end delay constraint of the network service. Since in Scenario 2 most of the service links are delay-sensitive, those servers will be kept by the filtering from which the later random selection will be mostly successful. Interestingly, in such cases this simple smartening of the *nova-scheduler* is enough to reach high efficiency.

VII. Conclusion

We are in the middle of a transformation process in which online applications are now containerized and run in virtual machines somewhere in the cloud instead of deploying code on bare metal in-house. But in the near future, the progress continues, and the phenomenon of fog and edge computing will widely spread, and together with the advanced wireless radio technology of 5G and the plethora of smart devices and sensors of the Internet of Things, today's cloud will descend and surround us eventually providing ubiquitous computing. We prepare for this vision by proposing a service orchestration algorithm, which we call DARK, in order for an infrastructure provider to be able to quickly map incoming requests for virtualized service to physical resources. Our algorithm was designed with the cloud-edge infrastructure topology in mind. As a second contribution, we take the most widely applied VIM of today, OpenStack, and investigate how to make it compliant with the future's requirements. We propose to federate all the geographically scattered compute islands of a provider under one controller in oder to save on resource usage and to enable VM migration to remote nodes. We implemented and successfully tested our OpenStack-based prototype. Third, with numerical evaluation we compared DARK and an advanced *nova-scheduler*, showing that although DARK's performance is outstanding, depending on the delay-tolerance of services, a simpler solution might perform well enough.

Acknowledgement

This research was supported by H2020-ICT-2014 project 5GEx (grant agreement no. 671636), which is partially funded by the European Commission. Project no. PD 121201 has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the PD_16 funding scheme.

References

[1] C. C. Byers, "Architectural imperatives for fog computing: Use cases, requirements, and architectural techniques for fog-enabled iot networks," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 14–20, 2017.
[2] P. Mach *et al.*, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys and Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.
[3] P. Schulz *et al.*, "Latency Critical IoT Applications in 5G: Perspective on the Design of Radio Interface and Network Architecture," *IEEE Communications Magazine*, vol. 55, no. 2, pp. 70–78, 2017.
[4] ETSI, "White Paper: Network Functions Virtualisation (NFV)," 2013. [Online]. Available: http://portal.etsi.org/nfv/nfv_white_paper2.pdf
[5] B. Gero *et al.*, "The Orchestration in 5G Exchange - a Multi-Provider NFV Framework for 5G Services," in *IEEE NFV-SDN*, 2017.
[6] ETSI Open Source MANO. [Online]. Available: https://osm.etsi.org/
[7] ONAP. [Online]. Available: https://www.onap.org
[8] CORD. [Online]. Available: https://opencord.org/
[9] Cloudify. [Online]. Available: https://cloudify.co/
[10] OpenStack Nova Filter Scheduler. [Online]. Available: https://docs.openstack.org/nova/latest/user/filter-scheduler.html
[11] M. Scharf *et al.*, "Network-aware instance scheduling in openstack," in *IEEE ICCCN*, 2015.
[12] S. Sahasrabudhe *et al.*, "Improved filter-weight algorithm for utilization-aware resource scheduling in openstack," in *IEEE ICIP*, 2015.
[13] F. Lucrezia *et al.*, "Introducing network-aware scheduling capabilities in openstack," in *IEEE NetSoft*, 2015.
[14] OpenDayLight. [Online]. Available: https://www.opendaylight.org/
[15] B. Nemeth *et al.*, "Efficient Service Graph Embedding: A Practical Approach," in *IEEE NFV-SDN*, 2016.
[16] E. Amaldi *et al.*, "On the computational complexity of the virtual network embedding problem," *Electronic Notes in Discrete Mathematics*, vol. 52, pp. 213 – 220, 2016, iNOC 2015.
[17] A. Fischer *et al.*, "Virtual network embedding: A survey," *IEEE Communications Surveys and Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.
[18] M. Szabo *et al.*, "Resource Allocation Algorithm for Distributed Cloud Environments," Tech. Rep., 2018. [Online]. Available: https://sb.tmit.bme.hu/mediawiki/index.php/DARK
[19] VMTP. [Online]. Available: http://vmtp.readthedocs.io
[20] OpenStack Service Function Chaining. [Online]. Available: https://docs.openstack.org/newton/networking-guide/config-sfc.html