Resource Allocation Algorithm for Distributed Cloud Environments

Marton Szabo*, David Haja†, Mark Szalay‡
Department of Telecommunications and Media Informatics
Budapest University of Technology and Economics, Hungary
*szabo.marton@tmit.bme.hu, †haja.david@tmit.bme.hu, ‡mark.szalay@tmit.bme.hu

Abstract

In this paper we present a novel on-line NFV (Network Function Virtualization) orchestration algorithm for edge computing infrastructure providers that operate in a heterogeneous cloud environment. The goal of our algorithm is to minimize the usage of computing resources which are offered by a public cloud provider (e.g. Amazon Web Services), while fulfilling the required networking related constraints (latency, bandwidth) of the services to be deployed. We propose a reference network architecture which acts as a test environment for the evaluation of our algorithm. During the measurements, we compare our results to the optimal solution provided by an ILP-based solver.

Index Terms

orchestration; network algorithm; heterogeneous cloud, fog computing, cloud computing

I. INTRODUCTION

In the field of telecommunications many new emerging trends can be observed. For example, IoT (Internet of Things) aims to make traditional devices smart and connected to the Internet; this is a major trend already present nowadays. In the field of transportation, we can also find many exciting new solutions (e.g. self driving cars, autonomous drones, etc.). The appearance of 5G networks is expected to enable even more revolutionary services to be built [1]. These can be for example the tactile Internet and on-line augmented reality applications, where the low response time is a crucial prerequisite. These services require not only the evolution of the radio interface, but also necessitates certain modifications in the topology of the back-haul network in order to serve the large number of new devices and the network traffic generated by them, and provide near real-time response times.

Today's widely deployed telecommunications networks are not flexible enough to fulfill these expected new challenges. For example, running network functions are currently binded to the special purpose hardware elements located in the core of the network (e.g. firewalls, carrier grade NAT platforms), which means unbearably high latency for most of the new applications. The NFV (Network Function Virtualization) concept aims to overcome this challenge [2], [3]: virtualization of the network functions makes it possible to run these services on general purpose hardware (e.g. x86 based servers with high compute capacity), thus removing the limitations coming from the physical location of the devices.

Furthermore, by extending the traditional cloud concept with compute nodes at edge of the network - often called Mobile Edge Computing (MEC) [4] - using together with the high amount of resources in the core data centers enables many new applications for the service providers. This way, it is possible to run certain network functions near to the end-users with very low latency guaranteed, while other components of a service - that are not so sensitive to latency - can be deployed in core data centers instead of placing those in the limited capacity edge nodes [5]. One service consists of elementary functions connected with each other in a given order. This is called Service Function Chaining (SFC), and its model defines different requirements to the underlying network and virtualization environments (required CPU, RAM, storage, constraints on bandwidth, latency between nodes) [6]. The process that maps multiple service graphs (SGs) composed of different virtual network functions (VNFs) to a common physical infrastructure, represented

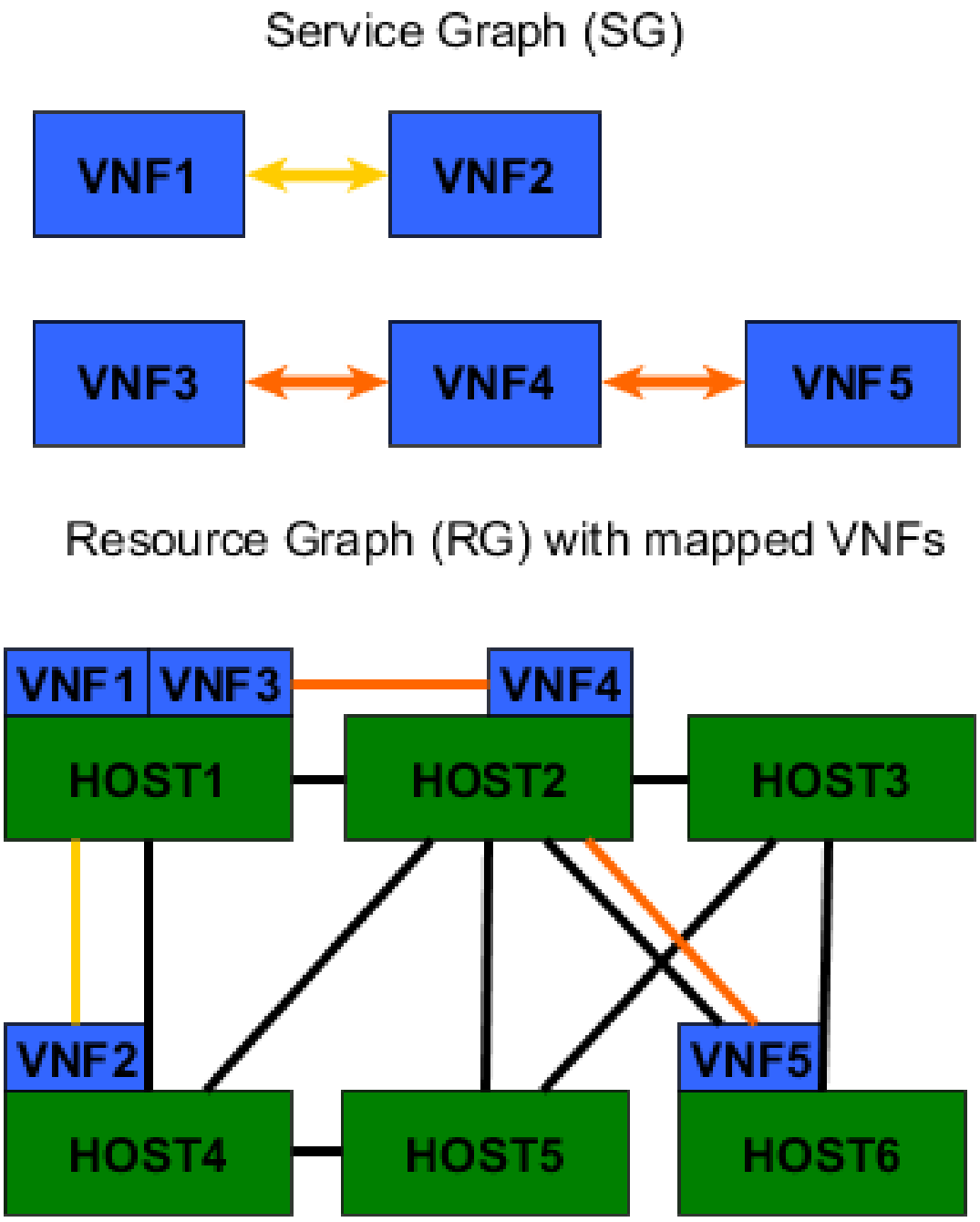

Figure 1. Mapping Service Function Chain to the infrastructure

by the Resource Graph (RG), is called Virtual Network Embedding (VNE). An example of the placement of NFs and logical connections to the nodes and links of the physical infrastructure is shown in Fig. 1.

By applying the previously described technologies together with dynamically reconfigurable, software-based networks (Software Defined Networks, SDN), limitations caused by the current rigid network architectures can be eliminated, thus making the introduction of the new-generation network services possible. These can be for example self-driving cars and industrial robots controlled from the cloud, edge

content caching, and smart city applications.

An interesting aspect of the future's 5G networks is the resource sharing between different service providers, which would enable their users to be served independently of their actual physical location (e.g. in case of roaming). In such a multi-provider cloud environment the goal of the participants is to utilize their own infrastructure the most efficiently, thus minimizing the expenses caused by using external resources. In this paper, we propose a new online resource orchestration algorithm which finds proper placement for the network functions of online services while minimizing the costs to be paid for external resources taken at third-party infrastructure providers. In order to evaluate our algorithm, we implemented a framework where we tested its performance in various simulation scenarios. We compare our results to an ILP-provided optimal offline solution.

The paper is organized as follows: Sec. II overviews the related work. Sec. III describes our reference architecture and the basics of fog computing. In Sec. IV we define the problem in the form of an ILP to be solved. Our online heuristic algorithm is explained in Sec. V. Performance measurements are evaluated in Sec. VI. We conclude our work in Sec. VII.

II. RELATED WORK

Virtual Network Embedding (VNE) is known to be NP-hard [7], 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, ii) approximation-based algorithms that trade the optimal solution for better runtime. [8] summarizes many of the possible solutions to the VNE problem.

Several approaches use Integer Linear Programming (ILP) to solve the VNE problem. In [9] the authors implemented an ILP formula to minimize the cost of embedding in terms of edge costs while maximizing the acceptance ratio. Reconfiguration of the existing mapping by enabling VNF migrations formed as MILP (Mixed ILP) were studied in [10].

Many different approaches solve the VNE problem with heuristic algorithms. Most of them perform the mapping in two steps: node mapping stage and edge mapping stage, thus physical nodes that have been selected to host neighboring network functions in the node mapping state may be multiple hops away from each other. Many algorithms aim to solve this problem by minimizing link utilization, e.g., [11], [12]. Authors of [13] proposed a hybrid algorithm, which first solves a relaxation of the original problem by using linear programming in polynomial time. Then they use deterministic and randomized rounding techniques on the solution of the linear program to approximate the values of the variables in the original MILP.

A decomposing mapping algorithm proposed in [14] aims to minimize the mapping cost by making a selection of the available decompositions during the node mapping stage.

III. HETEROGENEOUS NETWORK MODEL

In this section, we describe the new networking concept which is expected to be able to cope with the explosive increase of the new IoT endpoints and high bandwidth applications like real-time virtual reality. After that, we present our own network model, that can be used to evaluate different service creation methods over the new architecture.

A. Fog computing

By heterogeneous cloud, we mean a complex system that extends the traditional cloud computing paradigm with adding computing capacity close to the end users. These resources are distributed in the service provider's network, for example co-located with an Internet PoP (Point of Presence). This new approach makes possible to serve the users at the edge of the network rather than routing over the whole

Internet backbone to data centers in the core. This ensures latency reduction and bandwidth savings in the backbone, thus better QoS (Quality of Service) can be provided.

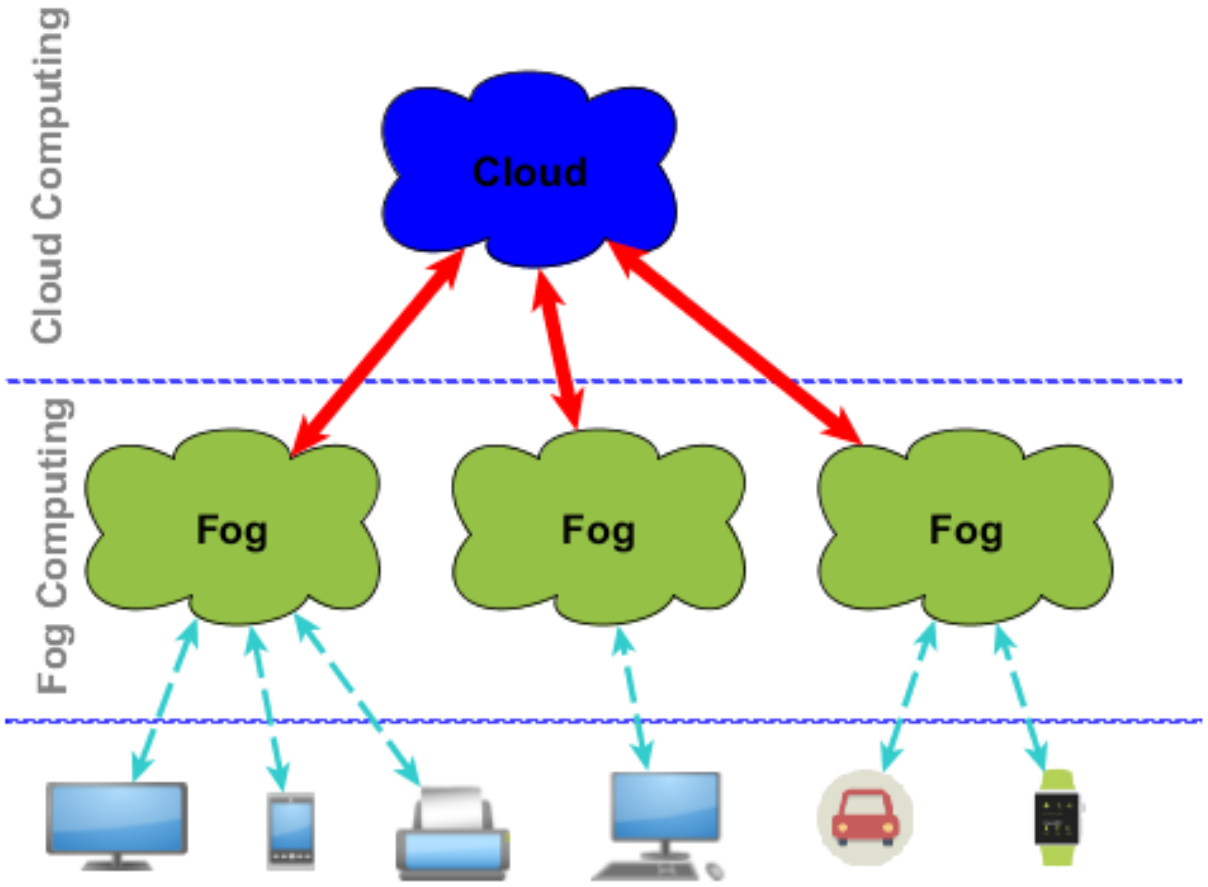

Figure 2. The architecture of fog computing

This architecture consists of geographically distributed small data centers (Fogs), and it is called as Fog Computing in the relevant literature [15], [16]. The architecture [17] can be divided into three layers as depicted in Fig. 2. The top layer includes the central clouds that can be the ISP's own private cloud or a public cloud provider's (e. g. Amazon Web Services, Microsoft Azure) infrastructure. The Fogs are located in the middle layer: this layer contains less computing capacity compared to the previous layer, but can host applications with strict requirement on response time. While having limited resources, Fogs can be used to enhance the performance of the end devices, or to offload the hardware intensive tasks from them, thus ensuring better battery lifetime. The bottom layer hosts the end-devices that consume the compute and network resources of the ISP. The devices usually connect to the network via wireless interface. Further features of the end equipment are location-independence, limited hardware resources and large quantity.

Fog Computing also defines an ideal architecture for one of today's most important emerging paradigm, which is the IoT (Internet of Things) [18], [19]. In IoT, sensor devices in the bottom layer usually monitor different environmental variables, then send the measurements to a central entity located in the network. This entity may send control messages to the devices in order to change their state, then aggregates and transmits the data to an other processing unit that for example stores the data and creates statistics. This other unit may be suitable to be placed in the central cloud infrastructure. Other Fog Computing use-cases

can be found in the white-paper issued by the Open Fog Consortium [20].

B. Reference network

Based on the previously described considerations we created our own network model for the three-tier Fog Computing architecture: a network consists of a given number of Fogs and Central Clouds. Each fog contains a random number of servers with given computing capabilities (CPU, RAM and storage) and two gateway nodes. Each Fog 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 Fog the nodes are connected in a full mesh topology. We assume that bandwidth is not the bottleneck therein, because the nodes that belong to the same Fog are located close to each other and the blocking-less feature can be provided by choosing the right data center topology. The Fogs and the central data centers are connected with each other via the core network. The central cloud can be hosted by a public infrastructure provider (e.g. Amazon Web Services, Microsoft Azure) or the ISP's own data center. A topology may contain any number of central clouds. We assume that they have unlimited compute, storage and memory capacity, but the service provider needs to pay a fee for the consumed resources. Fig. 3 shows an example topology with four fog clusters and one central cloud node. The SAPs represent the connection points to the network. The core network in the center is responsible for the interconnection of the other parts of the topology.

IV. PROBLEM STATEMENT

We search for a solution to the following problem: How can we deploy service chains to a previously described heterogeneous cloud environment in a cost effective way? Let us assume that we are an ISP with Fogs scattered around our network with given computing capabilities. Furthermore, we have access to one or more public cloud provider's infrastructure through the Internet. Because of economic reasons, it may be beneficial to have a contract with more than one provider, thus better prices can be achieved for the allocation. We expose our network to the end-users, who can then initiate SFC deployments with various QoS requirements. In this case, an efficient algorithm, which allocates physical resources to the components of the SFC, is necessary. The goal of the algorithm is to minimize the cost to be payed for consuming external resources by fulfilling the QoS requirements and other constraints that are dictated by the capabilities of the network. Finding an optimal solution is known to be NP-hard as this problem can be treated as a generalized version of the previously described VNE problem (arbitrary resource cost assigned to each node).

Equations 1-8 describe the problem as an ILP in the following section. The notations we use in the formal problem description are summarized in Table I. We provide the intuitive meaning of each line of the ILP in the following.

$$\forall i \in V_s : \sum_{u \in V_r} x_u^i = 1 \tag{1}$$

$\forall (i,j) \in E_s, \forall u \in V_r :$

$$\sum_{v:(u \to v) \in V_r} y_{u,v}^{i,j} - \sum_{w:(w \to u) \in V_r} y_{w,u}^{i,j} = x_u^i - x_u^j \tag{2}$$

$$\forall u \in V_r, \forall i \in V_s : \sum_{i \in V_s} x_u^i \vec{r_i} \leq \vec{\rho_u} \tag{3}$$

$$\forall (i,j) \in E_s : \sum_{(u,v) \in E_r} y_{u,v}^{i,j} \delta_{u,v} \leq d^{i,j} \tag{4}$$

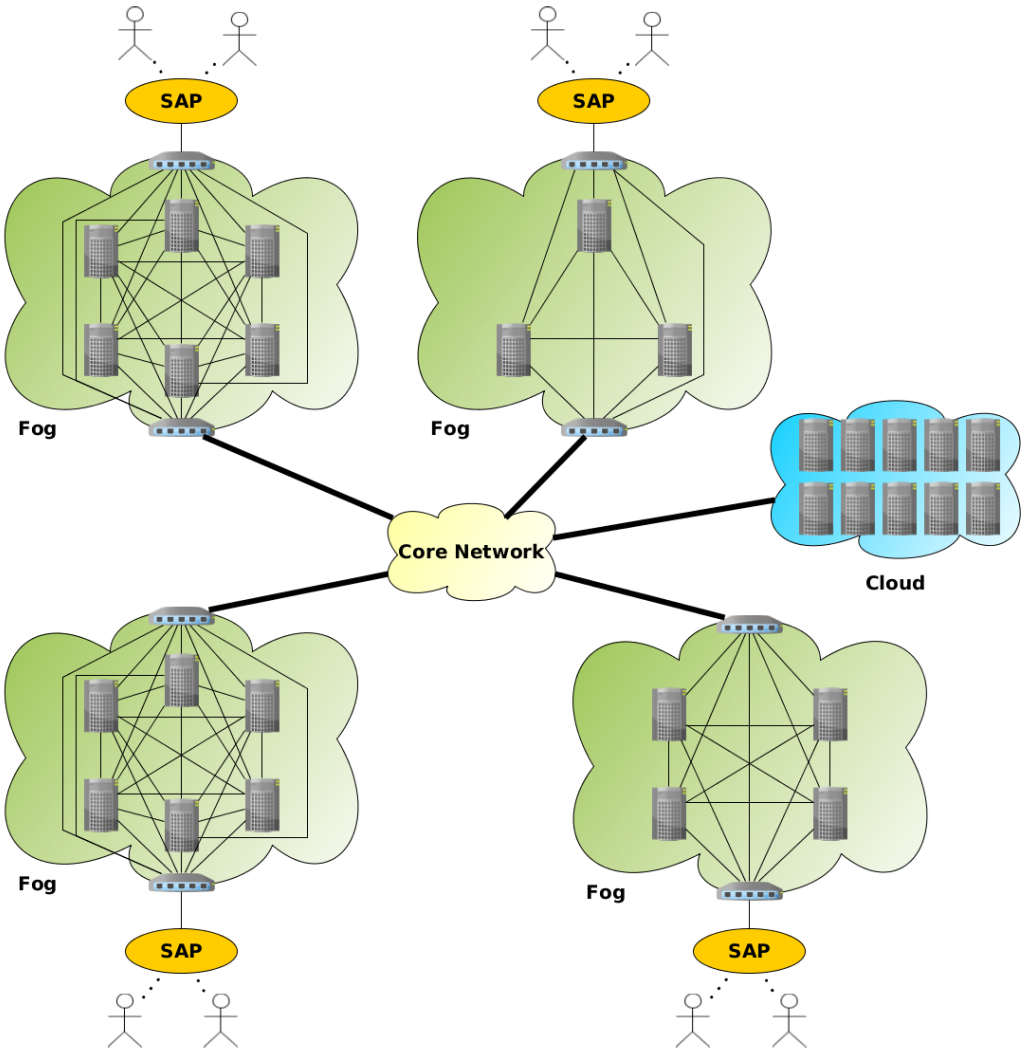

Figure 3. Example to the modeled architecture

$$\forall (u,v) \in E_r : \sum_{(i,j) \in E_s} y_{u,v}^{i,j} b^{i,j} \leq \beta_{u,v} \tag{5}$$

$$\forall i \in \varrho_s, : x_i^i = 1 \tag{6}$$

$$\min \sum_{u \in V_r} \sum_{i \in V_s} x_u^i c_u^i \tag{7}$$

As the result of the mapping, each NF is assigned to exactly one physical node (1). The flow constraint is given by (2). The total amount of resources required by the NFs mapped to a given node cannot exceed the resources available at the node (3). The total delay on the physical path of a SG link cannot exceed the requested delay between the two NFs (4). Similarly, the total bandwidth of virtual links mapped to the same physical link cannot be greater than the available bandwidth (5). The SG SAPs are mapped to the RG SAPs with the same ID (6). The objective function of the optimization is to minimize the external resource costs (7). The cost of deploying a VNF is calculated based on the formula:

Table I
NOTATIONS USED

Notation	Description
V_s, E_s	NFs and links of the service graph
V_r, E_r	Nodes and links of the resource graph
$(i,j) \in E_s$	SG link between NF i and j
$(u,v) \in E_r$	RG link between node u and v
$x_u^i : i \in V_s, u \in V_r$	1 if NF i has been mapped to node u
$y_{u,v}^{i,j}$	1 if (u,v) is in the physical path of (i,j)
$\vec{r_i}$	Resources required by NF i
$\vec{\rho_j}$	Available resources in Node j
$\delta_{u,v}$	Delay of physical link (u,v)
$d^{i,j}$	Maximal delay between NF i and j
$\beta_{u,v}$	Available bandwidth on (u,v) link
$b^{i,j}$	Required bandwidth between NF i and j
$\varrho_s \subset V_s$	SAPs in the SG
$\varrho_r \subset V_r$	SAPs in the RG
$\varrho_s \subseteq \varrho_r$	SG sAPs can be found in the RG also
c_u^i	The cost of running NF i on Node u
$\gamma \subset V_r$	Core Cloud nodes

$$c_u^i = \alpha_u * CPU^i + \beta_u * RAM^i + \gamma_u * BW^i + \delta_u * STR^i, \tag{8}$$

where α, β, γ and δ are the cost parameters of the physical node. The CPU, RAM, BW and STR are the resources requested by the NF, where BW is the sum bandwidth on all links connected to the NF, and STR is the allocated storage. In that case, when a node belongs to a fog, we set the parameters to 0, thus ensuring that using our own infrastructure does not imply any cost. We entered the problem to an ILP solver by customizing the program that was used in [21] with our own cost calculation method.

V. OUR ONLINE ALGORITHM

In this section we propose a novel orchestration algorithm that aims to minimize resource usage from the external core clouds, thus utilizing our own infrastructure in the most efficient way. NF 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 fogs in order to serve more latency and bandwidth sensitive requests. We introduce the $migrationcost$ attribute in (9):

$$migrationcost^i = \min \lambda_u * STR^i, u \in \gamma, \tag{9}$$

as there is a cost penalty to be payed for moving NF_i to the cloud from one of the fogs. The rationale is that a migration process requires redundant resources to be allocated caused by the duplicating the state of the network function to be transfered to the cloud. We derive this cost from the migration coefficient of a node (λ) and the allocated memory to NF^i. The main steps of our algorithm taking into account migration costs are described in the following, and pseudo-code is provided in Alg. 1 and 2.

A. SG 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.

Complexity: We iterate over all of the virtual links while we collect the neighbors in the service graph, and store them in a sorted list. Each of the service graphs are connected, so the following inequality will be satisfied: $(V_s \backslash \varrho_s) \leq E_s$
The computational complexity of ORDERSUBCHAINS() is:

$$\mathcal{O}\left(|E_s|\right).$$

B. VNF mapping

The next step step is the mapping of the service requests to the physical infrastructure. The MAP method iterates trough 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 bandwidth 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 NF. 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 NF 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 NF, 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 an other 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 NFs to the central cloud, thus freeing up resources in the fogs. The migration process is described in the next section.

Complexity: In MAPVNF function the orchestration algorithm maps a Service Graph node and an adjacent link together. First the algorithm filters the RG nodes by hardware capacity. In this step the orchestrator examine all of the physical nodes, so its complexity is: $\mathcal{O}\left(|V_r|\right)$. After that we remove the physical nodes, which do not complete the network requirements with the node selected in the previous mapping. In this step we use Dijkstra algorithm to calculate the path and length between compatible physical nodes and the previous node. Its complexity is: $\mathcal{O}\left(|V_r|(|E_r| + |V_r|log|V_r|)\right)$.
The final list is sorted based on the resource cost of hosting the actual VNF. The complexity of weighting and sorting is: $\mathcal{O}\left(|V_r| + |V_r|log|V_r|\right)$.
Summarizing,the computational complexity of MAPVNF() can be estimated from above with:

$$\mathcal{O}\left(|V_r| + |V_r|(|E_r| + |V_r|log|V_r|) + |V_r| + |V_r|log|V_r|\right).$$

When the algorithm doing the rollback, the max_rb parameter gives a limitation for the number of attempts with a given VNF, and also restricts the level of the backtracking. Therefore the ROLLBACK function complexity can be estimated from above with: $\mathcal{O}\left(max\_rb^{max\_rb}\right)$.

Algorithm 1 Service graph mapping to resource graph

$running \leftarrow$ copy(RG)
$mapped\_vnodes \leftarrow \emptyset$
$map_list \leftarrow$ORDERSUBCHAINS(SG)
$mapped\_vnodes.insert(\varrho_s.first)$
$rollback\_level$ = 0
for all $(u, v, link) \in map\_list$ **do**
    if $(u, v) \in mapped\_vnodes$ **then**
        $success \leftarrow$ MAPVIRTUALLINK($link$)
    else if $u \notin \varrho_s$ **then**
        $success \leftarrow$ MAPVNF($u, v, link$)
    else ▷ This means actual_element is a SAP
        $success \leftarrow$ MAPVLINK2SAP($u, v, link$)
    end if
    if $\neg success$ **and** $rb\_level \geq max\_rb$ **then**
        $success \leftarrow$ MIGRATING($cable, u, v$)
    else
        $success \leftarrow$ ROLLBACK($u, v, link$)
        $rollback\_level+ = 1$
    end if
    if $success$ **then**
        $mapped\_vnodes.insert(v)$
    end if
end fordone

Algorithm 2 VNF migration

procedure MIGRATING($migratables, u, v$)
    $mig\_vnf\_try = 0$
    for migratable_vnf ∈ migratables **do**
        if ISBIGGER($migratable\_vnf$,u) **then**
            if mig_vnf_try < max_vnf) **then**
                $poss\_nodes \leftarrow$ GETCOMPNODES($migratable\_vnf$)
                $mig\_try = 0$
                for n ∈ poss_nodes **do**
                    if mig_try < max_try **then**
                        $backup\_rg \leftarrow$ GETRUNNINGRG()
                        $backup\_sgs \leftarrow$ GETMAPPEDSGS()
                        $success \leftarrow$ TRYMIGRATE(n,$migratable\_vnf$,u,v)
                        if $success$ **then**
                            return True
                        else
                            RESETRG($backup\_rg$)
                            RESETSGS($backup\_sgs$)
                        end if
                        $mig\_try+ = 1$
                    end if
                end for
                $mig\_vnf\_try+ = 1$
            else
                return False
            end if
        end if
    end for
    return False
end procedure

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 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.

Complexity: The migration is limited by two integers: max_vnf and max_try. max_vnf value gives the maximum number of the VNFs which will be tried to migrate. The value of max_try limits the migration attempt for one VNF. A migration attempt is nothing more than increasing the old resource values, decreasing the new ones and check if everything is okay.

The computational complexity of MIGRATING() is:

$$\mathcal{O}\left(max\_vnf\ max\_try\right).$$

VI. Measurement Results

We have run measurements to demonstrate how our algorithm works in different topology setups with various requests. We had six scenarios each of them contained different number of fogs between 1 and 20. During our simulations we posted the requests (the service graphs) one by one, till the first failure occurred. We indicated failure when there was no way to deploy a service graph completely to the remaining available resource set.

First of all we compared our algorithm's efficiency with and without the migration function. In the comparison we examined how many CPUs our algorithm can deploy to the same resource set. It is easy to see that migrating is mostly used for cost efficiency, because if we turn off this feature then our algorithm has two options for deploying NFs. First option: our algorithm deploys cloud compatible NFs directly in the cloud. This is obviously not cost efficient because we have to pay for allocated resources to the third-party from the beginning. Second option: we want our algorithm to be cost efficient, so it puts all the NFs in the fogs. It is definitely cheap for us, however our fog resources will quickly exhaust.

Fig. 4 shows how many virtual CPU our algorithm can deploy with migrating and without migrating them if we want to keep our cost efficiency. Each bar on the figure represents the average number of

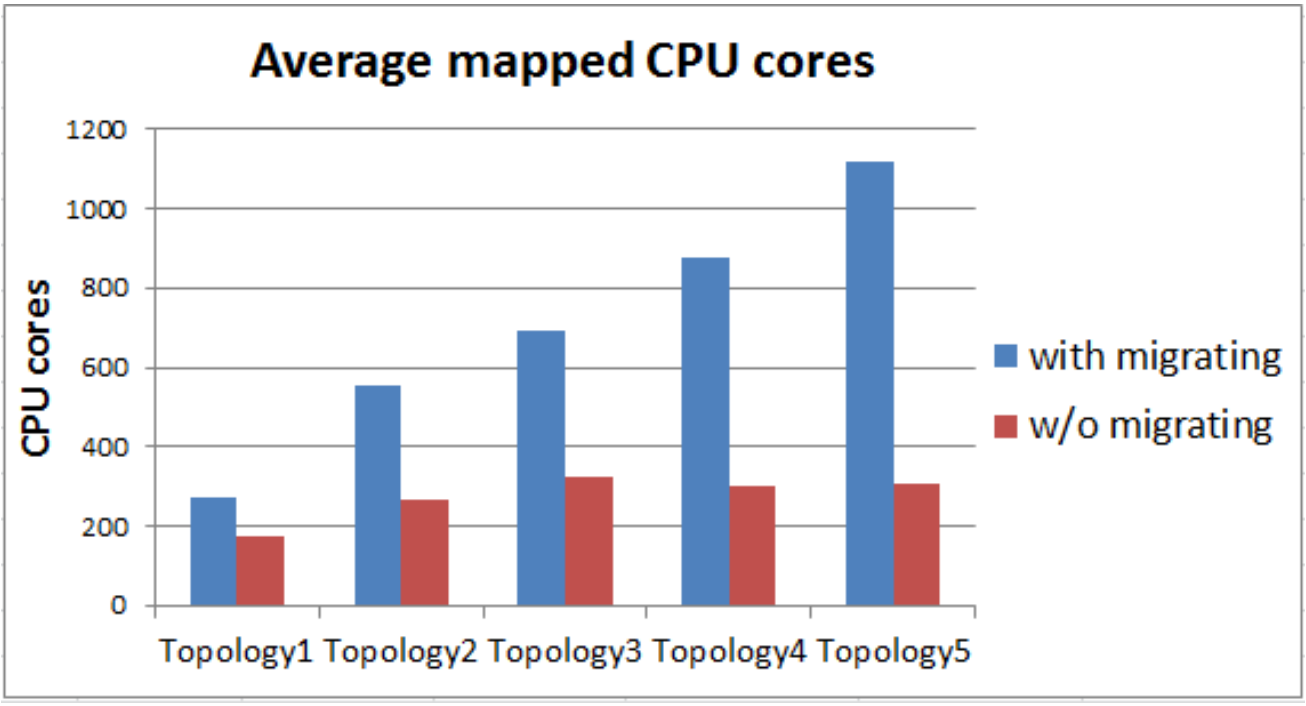

Figure 4. CPUs mapped with and without migration

mapped virtual CPUs on the quoted topologies. As you can see we deploy more virtual CPUs when the migration feature is turned on.

Next we wanted to examine what the cost difference between our algorithm and an optimal solution is. The optimal solution comes from the offline algorithm, which solves the previously defined ILP problem. Both algorithms were used with the same input, which contained the same resource graph and the same set of request graphs. We executed our algorithm and collected all the successfully deployed requests till the first failure. After that we executed the offline algorithm with the same resource graph and the collected request set. Both algorithms return with a number that defines the price of the request set on the given topology. We compared these outputs and plotted those in a scatter plot shown in Fig. 5: each point represents a cost difference for a given request sequence. It is noticeable how the request randomization affects the deployed CPU number. It is also remarkable how our algorithm scales: with the increase of the number of Fogs the difference between the two solutions does not go higher than 20%.

In Fig. 6 we depict the distribution of cost differences. The x axis represents the cost difference (in percentage) between the offline and our algorithm. The optimal solution contains only the price of the allocated resource in the third-party environment (cloud). In our solution we calculate a migration cost as well. It is remarkable how the migration cost affects our final price. If the migration cost is higher than the resource allocation cost, then it is worthy to put a cloud compatible NF directly to the cloud, so we can avoid high migration cost.

Finally we compared the execution times for both algorithms. The result is represented in Fig. 7. The offline algorithm's execution time is increasing exponentially with the number of Fogs, however our algorithm execution time grows only linearly.

VII. Conclusion

In this work we examined how future networks should handle IoT services and their deployment. We made network topology examples that could be used to run such services. For handling the service creation on these topologies we created an online algorithm that can deploy services cost efficiently and that is

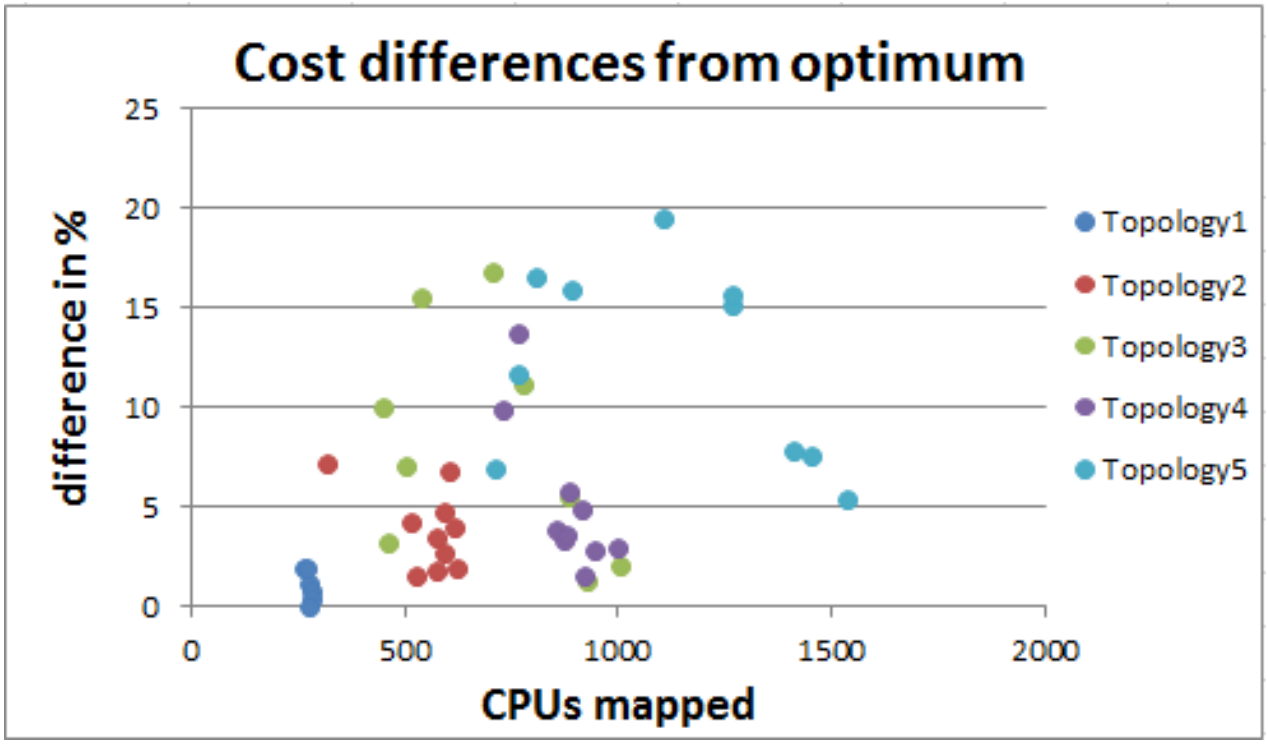

Figure 5. Differences of cost between the online and offline algorithms

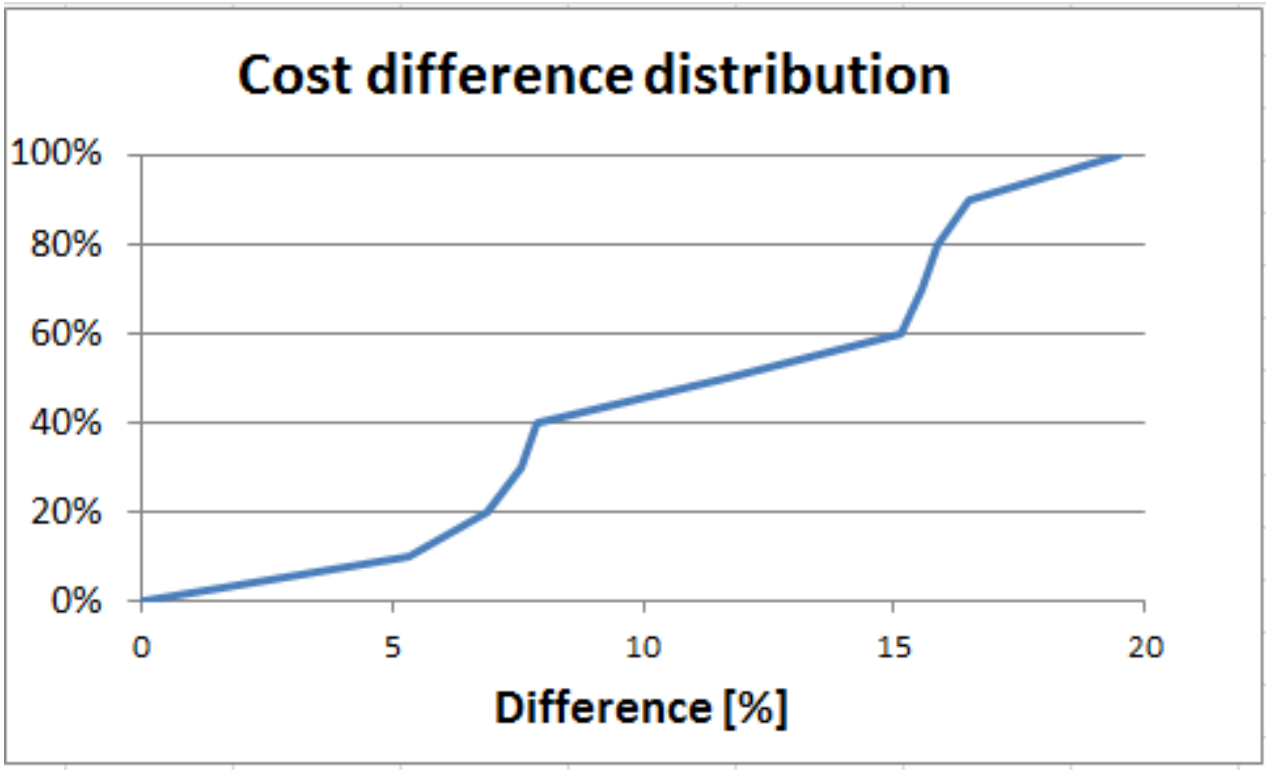

Figure 6. Cost difference distribution

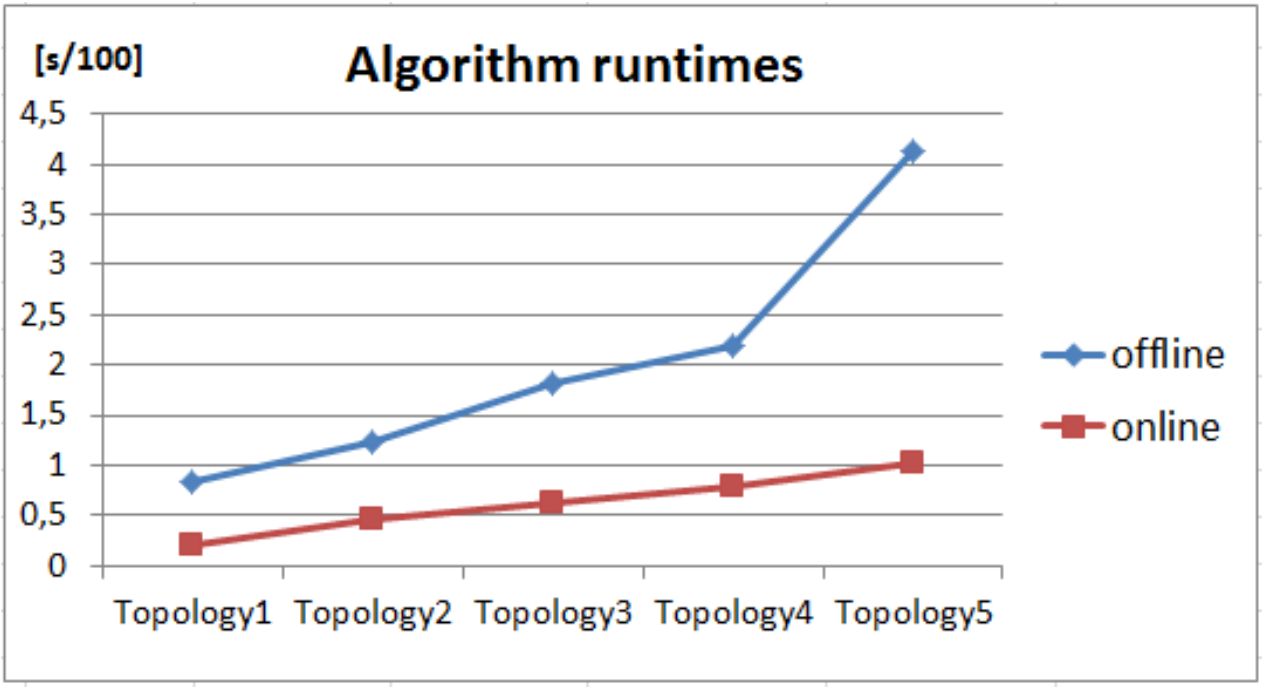

Figure 7. Runtimes of the algorithms

also able to handle networking requirements of the services. Our algorithm runs in polynomial time, thus scales as well, while it provides close-to-optimal orchestration results.

REFERENCES

[1] N. Panwar, S. Sharma, and A. K. Singh, "A Survey on 5G," *Phys. Commun.*, vol. 18, no. P2, pp. 64–84, Mar. 2016. [Online]. Available: http://dx.doi.org/10.1016/j.phycom.2015.10.006

[2] ETSI, "White Paper: Network Functions Virtualisation (NFV)," 2013. [Online]. Available: http://portal.etsi.org/nfv/nfv_white_paper2.pdf

[3] ETSI GS NFV-PER 001, Dec 2014, *Network Functions Virtualisation (NFV); Architectural Framework*, Mentve:http://www.etsi.org/deliver/etsi_gs/NFV-PER/001_099/001/01.01.02_60/gs_NFV-PER001v010102p.pdf, ETSI, 2014 Dec.

[4] G. Brown, "Mobil edge computing use cases and deployment options," Tech. Rep., 2016.03.01. [Online]. Available: https://www.juniper.net/assets/us/en/local/pdf/whitepapers/2000642-en.pdf

[5] Z. B. Pavel Mach, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys and Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[6] J. Halpern and C. Pignataro, "Service Function Chaining (SFC) Architecture," IETF RFC 7665, Oct. 2015.

[7] 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 – 7th International Network Optimization Conference.

[8] M. T. B. H. d. M. Andreas Fischer, Juan Felipe Botero and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Communications Surveys and Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[9] W. B. A. D. Z. Ines Houidi, Wajdi Louati, "Virtual network provisioning across multiple substrate networks," *Computer Networks*, vol. 55, no. 4, pp. 1011–1023, 2011.

[10] A. F. Gregor Schaffrath, Stefan Schmid, "Optimizing long-lived cloudnets with migrations," in *UCC '12, ACM*. ACM, 2012, pp. 99–106.

[11] W. G. Hongyan Cui and J. Liu, "A virtual network embedding algorithm based on virtual topology connection feature," *Wireless Personal Multimedia Communications*, 2013.

[12] Z. Z. Gang Wang and X. W. Zhaoming Lu, Yi Tong, "A virtual network embedding algorithm based on mapping tree," *Communications and Information Technologies*, 2013.

[13] M. R. R. Mosharaf Chowdhury and R. Boutaba, "Vineyard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Transactions on Networking*, vol. 20, no. 1, pp. 206–219, 2012.

[14] S. S. et al, "Network service chaining with optimized network function embedding supporting service decompositions," *Computer Networks*, vol. 93, no. 3, pp. 492–505, 2015.

[15] Q. L. Shanhe Yi, Cheng Li, "A survey of fog computing: Concepts, applications and issues," in *Mobildata '15, ACM*. ACM, 2015, pp. 37–42.

[16] L. R.-M. Luis M. Vaquero, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 5, pp. 27–32, 2014.
[17] 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.
[18] R. B. Amir Vahid Dastjerdi, "Fog computing: Helping the internet of things realize its potential," *IEEE Computer*, vol. 49, no. 8, pp. 112–116, 2016.
[19] J. Z. S. A. Flavio Bonomi, Rodolfo Milito, "Fog computing and its role in the internet of things," *MCC '12*, pp. 13–16, 2012.
[20] O. Consortium, "Openfog reference architecture for fog computing," Tech. Rep., 2017.02.08. [Online]. Available: \url{https://www.openfogconsortium.org/wp-content/uploads/OpenFog_Reference_Architecture_2_09_17-FINAL.pdf}
[21] B. Németh, B. Sonkoly, M. Rost, and S. Schmid, "Efficient Service Graph Embedding: A Practical Approach," in *Second IEEE International Workshop on Orchestration for Software Defined Infrastructures (O4SDI @ IEEE NFV-SDN 2016)*, Nov 2016.