JASPER: Joint Optimization of Scaling, Placement, and Routing of Virtual Network Services

Sevil Dräxler, Holger Karl, and Zoltán Ádám Mann

Abstract—To adapt to continuously changing workloads in networks, components of the running network services may need to be replicated (*scaling* the network service) and allocated to physical resources (*placement*) dynamically, also necessitating dynamic re-routing of flows between service components. In this paper, we propose JASPER, a fully automated approach to jointly optimizing scaling, placement, and routing for complex network services, consisting of multiple (virtualized) components. JASPER handles multiple network services that share the same substrate network; services can be dynamically added or removed and dynamic workload changes are handled. Our approach lets service designers specify their services on a high level of abstraction using *service templates*. JASPER automatically makes scaling, placement and routing decisions, enabling quick reaction to changes. We formalize the problem, analyze its complexity, and develop two algorithms to solve it. Extensive empirical results show the applicability and effectiveness of the proposed approach.

Index Terms—Cloud computing services, mathematical optimization, orchestration, virtual networks.

I. INTRODUCTION

Network services and applications like video streaming and online gaming, consist of different service components, including (virtual) network functions, application servers, data bases, etc. Typically, several of these network services are hosted on top of wide-area networks, serving the continuously changing demands of their users. The need for efficient and automatic deployment, scaling, and path selection methods for the network services has led to paradigms like network softwarization, including software-defined networking (SDN) and network function virtualization (NFV).

SDN and NFV provide the required control and orchestration mechanisms to drive the network services through their life-cycle. Today, network services are placed and deployed in the network based on fixed, pre-defined descriptors [1] that contain the number of required instances for each service component and the exact resource demands. More flexibility can be achieved by specifying auto-scaling thresholds for

metrics of interest. Once such a threshold is reached, the affected network services should be modified, e.g., scaled.

To react to addition and removal of network services, fluctuations in the request load of a network service, or to serve new user groups in a new location, (i) the network services can be scaled out or in by adding or removing instances of service components, respectively, (ii) the placement of service components and the amount of resources allocated to them can be modified, and (iii) the network flows between the service components can be re-routed through different, more suitable paths.

Given this large number of degrees of freedom for finding the best adaptation, deciding scaling, placement, and routing independently can result in sub-optimal decisions for the network and the running services. Consider a service platform provider hosting a dynamically changing set of network services, where each network service serves dynamically changing user groups that produce dynamically changing data rates. Trade-offs among the conflicting goals of network services and platform operators can be highly non-trivial, for example, Placing a compute-intensive service component on a node with limited resources near the *source* of requests (e.g., the location of users, content servers, etc.) minimizes latency but placing it on a more powerful node further away in the network minimizes processing time.

To deal with these challenges, we propose JASPER, a comprehensive approach for the **J**oint optimiz**A**tion of **S**caling, **P**lac**E**ment, and **R**outing of virtual network services. In JASPER, each network service is described by a *service template*, containing information about the components of the network service, the interconnections between the components, and the resource requirements of the components. Both the resource requirements and the outgoing data rates of a component are specified as *functions of the incoming data rates*.

The input to the problem we are tackling comprises service templates, location and data rate of the *sources* of each network service, and the topology and available resources of the underlying network. Our optimization approach takes care of the rest: based on the location and current data rate of the sources, in a single step, the templates are scaled by replicating service components as necessary, the placement of components on network nodes is determined, and data flows are routed along network paths. Node and link capacity constraints of the network are automatically taken into account. We optimize the solution along multiple objectives, including minimizing

S. Dräxler and H. Karl are with Paderborn University, Paderborn, Germany.

Z. A. Mann is with University of Duisburg-Essen, Essen, Germany.

resource usage, minimizing latency, and minimizing deployment adaptation costs.

Our main contributions are as follows:

- We formalize *template embedding* as a joint optimization problem for scaling, placing, and routing service templates in the network.
- We prove the NP-hardness of the problem.
- We present two algorithms for solving the problem, one based on mixed integer programming, the other a custom heuristic.
- We evaluate both algorithms in detail to determine their strengths and weaknesses.

With the proposed approach, service providers obtain a flexible way to define network services on a high level of abstraction while service platform providers obtain powerful methods to optimize the scaling and placement of multiple services in a single step, fully automatically.

The main novelty of our approach is the flexibility that it offers in several aspects:

- The number of instances of the service components does not have to be pre-defined but is automatically determined and adjusted as necessary.
- Scaling decisions are not based on simple and rigid local rules, but rather on the global system state, thus also taking into account the interplay between scaling decisions for different service components.
- The resource needs of the service components do not have to be pre-defined. Instead, they are specified as functions of the input data rate of the component. This way, the resource allotment of the components can be flexibly adjusted as needed.
- Not only chains of service components are supported, but also more complicated graph structures. Moreover, flows between components can be split among multiple paths.
- Optimization is carried out with respect to multiple metrics, including the minimization of node and link overloads, the minimization of resource consumption, and the minimization of deployment modification costs.

Although some existing approaches possess some of these characteristics, we are not aware of any existing approach that would offer the same level of flexibility as JASPER.

The flexible problem formulation results in a complex optimization problem, in which scaling, placement, and routing decisions are taken in a single step for all – already deployed or newly requested – network services. Efficiently solving such a complex optimization problem is not trivial. The empirical results demonstrate that our proposed algorithms can be used to tackle this problem.

The rest of the paper is organized as follows. In Section II, we give an overview of related work. Section III presents a high-level overview of our approach and Section IV describes the details of our model and assumptions. We discuss the complexity of template embedding in Section V and formulate the problem as a mixed integer programming model in Section VI. We present a heuristic solution in Section VII and the evaluation results of our solutions in Section VIII, before concluding the paper in Section IX.

II. RELATED WORK

Our solution can be applied in different contexts, e.g., (distributed) cloud computing and Network Function Virtualization (NFV). In this section, after an analysis of related approaches from a theoretical point of view, we give an overview of related work in the cloud computing and NFV contexts. The major difference among the existing work in these two fields is usually the abstraction level considered for the substrate network and the resulting assumptions for the model. In particular, in the cloud computing context, embedding is typically done on top of physical machines in data centers, while in the NFV context, embedding is done on top of geographically distributed points of presence.

A. Virtual network embedding problem

The combination of the placement and path selection sub-problems of template embedding is similar to the Virtual Network Embedding (VNE) problem. Both deal with mapping virtual nodes and virtual links of a graph into another graph. VNE does not include the scaling step that is an important part of JASPER. Fischer et al. [2] have published a survey of different approaches to VNE, including static and dynamic VNE algorithms.

In contrast to static VNE solutions that consider the initial mapping process only, in this paper we also deal with optimizing and modifying already embedded templates. Some VNE solutions, for example, Houidi et al. [3], can modify the mapping in reaction to node or link failures. The modifications in their work, however, are limited to recalculating the location for the embedded virtual network, i.e., migrating some of the nodes and changing the corresponding paths among them. In addition to these modifications, our approach can also modify the *structure* of the graph to be embedded by adding or removing nodes and links if necessary.

B. Cloud computing context

The related problem in cloud environments is typically formulated as resource allocation for individual components, i.e., scaling and placing instances of virtual machines on top of physical machines while adhering to capacity constraints [4], [5]. The communication among different virtual machines is usually left out or considered only in a limited sense [6]. Among cloud computing approaches, even the ones that do consider the communication among virtual machines [7], [8], [9], [10] do not include routing decisions whereas JASPER also includes routing.

Relevant to the placement sub-problem of template embedding, Bellavista et al. [11] focus on the technical issues of deploying flexible cloud infrastructure, including network-aware placement of multiple virtual machines in virtual data centers. Wang et al. [12] study the dynamic scaling and placement problem for network services in cloud data centers, aiming at reducing costs. These papers also do not address routing. Moreover, our approach of specifying resource consumption as a function of input data rates allows a much more realistic modeling of the resource needs of service components than the

constant resource needs assumed by the existing approaches in this context.

Keller et al. [13] consider an approach similar to our template embedding problem in the context of distributed cloud computing. Our terminology is partly based on their work but there are important differences in the assumptions and the models that make our approach stronger and more flexible than their solutions. In contrast to their model, where the number of users determines the number of required instances, the deciding factor in our work is the *data rate* originating from different source components. Moreover, we do not enforce strict scaling restrictions for components as done in their work. (For example, their method needs as input the exact number of instances of a back-end server that is required behind a front-end server.) Finally, we use a more sophisticated multi-objective optimization approach where different metrics like CPU and memory load of network nodes, data rate on network links, and latency of embedded templates are considered.

C. Network function virtualization context

The placement and routing problems are also relevant in the field of Network Function Virtualization (NFV). In the NFV context, the *forwarding graphs* of network services composed of multiple virtual network functions (VNFs) are mapped into the network. Herrera et al. [14] have published an analysis of existing solutions for placing network services as part of a survey on resource allocation in NFV. Addis et al. [15] study the properties and complexity of this problem, comparing the VNE-based and routing and location-based formulations.

Kuo et al. [16] consider the joint placement and routing problem, focusing on maximizing the number of admitted network service embedding requests. Gupta et al. [17] consider placement and routing of (linear) chains of virtual network functions focusing on minimizing the used network link capacity. Ahvar et al. [18] propose a solution to this problem, with the assumption that the VNFs can be re-used among different flows. Their objective is to find the optimal number of VNFs for all requests and to minimize the costs for the provider. Another similar approach that considers re-using components is proposed by Bari et al. [19]. Kebbache et al. [20] aim at solving this problem in an efficient way that can scale with the size of the underlying infrastructure and the embedded network services. They measure the efficiency of their algorithms with respect to run time, acceptance rate, and costs. Another attempt to solve this problem in an efficient and scalable way has been made by Luizelli et al. [21], focusing on minimizing resource allocation. In comparison to all these approaches, we consider a more comprehensive optimization objective, trying to minimize the delay for network services, the number of added or removed instances, resource consumption, as well as overload of resources.

Liu et al. [22] consider the dynamic placement and adjustment of services, similar to the joint placement and scaling approach in this paper. However, their model uses different assumptions than ours, e.g., considering a single user per service function chain, which can only be linear. Also, unlike our approach, they use pre-calculated paths in the network and do not consider the latency of paths for routing.
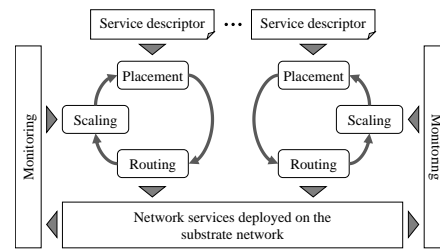

Fig. 1. Conventional network service life-cycle, from descriptors to running services

In our work, the exact structure of the network service does not have to be fixed in the deployment request. In a previous work [23], we have studied another type of flexibility in the network service structure, namely, the case where the components are specified with a partial order and can be re-ordered. JASPER is based on the assumption that the *order* of traversing the service components is fixed and given, however, the number of instances for each component and the amount of resources allocated to each component can be adapted dynamically.

Several other solutions [24], [25], [26], [27] have been proposed for placement, scaling, and path selection problems for network services. Our template embedding approach has two important differences compared to these solutions. First, our approach can be used for initial placement of a newly requested service as well as scaling and adapting existing embeddings. Second, in our approach, the structure of the service and mapping of the service components to network nodes and the optimal routing are determined in one single step, based on the requirements of the service and current state of network resources, searching for a global optimum.

A preliminary version of this work was presented at the CCGrid 2017 conference [28]. Compared to the conference version, this paper contains the proof of NP-hardness, more detailed explanation of the problem model and the devised algorithms, and a more detailed evaluation and discussion of the practical applicability of the proposed approach.

III. APPROACH OVERVIEW

In typical management and orchestration frameworks [1], service providers need to submit exact descriptors of their network service structure, resource demands, and expected traffic from sources to a service management and orchestration system (Fig. 1). Based on the descriptors, placement, scaling, and routing decisions are made for each network service, independently from one another.

JASPER makes several changes to this approach, partly with respect to the description of the network services (Section III-A) and partly with respect to handling the scaling, placement, and routing decision processes (Section III-B).

A. Templates instead of over-specified descriptors

Deploying several instances of service components might be necessary in different scenarios, e.g., for load balancing purposes or for instantiating new instances closer to the new
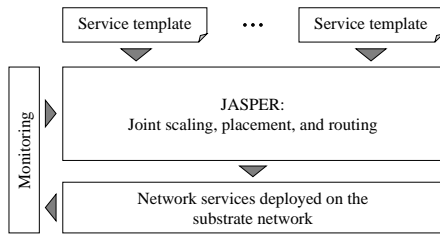
Fig. 2. Network service management and orchestration using JASPER

users in case of a delay-sensitive application. In existing cloud and NFV orchestration solutions, auto-scaling rules are defined in service descriptors, based on certain thresholds and using strict minimum and maximum values for the number of instances that can be created. Because of the limited precision and flexibility of typical descriptors, we base our approach on so-called *service templates*. Using service templates, service providers are required to specify neither the *exact* resource demands (e.g., memory or CPU) of service components nor the required number of instances of each component.

The service template describes the components of the network service and their required interconnections on an abstract level, without deployment details. Moreover, it gives the resource demands of the network service as a function of the load:

- The required computational capacity (e.g., CPU and memory) is described for each service component as a function of the input data rate. This can be used to calculate the network node capacity required to host the service component.
- The amount of traffic leaving each service component towards other components is specified as a function of the data rate that enters the component. This can be used to calculate the link capacity required to host the traffic flowing between any two interconnected instance.

In addition to the service templates, service providers may include the expected traffic originating from the sources of the network service in the request to embed a service template. As the traffic is constantly changing, the current traffic needs to be monitored and fed back to the template embedding process, to keep the network service in an optimal state. In this way, depending on the location and data rate of the sources of the network service, resource requirements are calculated dynamically, based on the given functions, eliminating the risk of over- or under-estimating the resource demands. Based on the functions describing the dependency of resource requirements and outgoing data rates on incoming data rates, it is also possible to reason about possible changes to the deployment and their impact, which is a pre-requisite for effective optimization. The specific functions highly depend on the type and implementation of the service component and can be derived, for example, based on historical usage data or by automatic service profiling methods [29].

B. Joint, single-step scaling, placement, and routing

As shown in Fig. 1, in typical management and orchestration frameworks [1], based on the description of the network

service and the state of the network's resources, the requested number of instances for each service component are computed and then placed and deployed with the requested amount of resources, in an appropriate location. After path selection and instantiation of the network service, the running instances are monitored and re-scaled and re-placed based on pre-defined scaling rules if required. Deciding the number of required instances for each service component, the amount of resources allocated to each component, and the optimal paths selected for routing the network service flows are, however, highly interdependent problems, which cannot be solved optimally using such independent management and orchestration steps.

Our approach, illustrated in Fig. 2, changes the way network service life-cycle is handled, by combining scaling, placement, and routing steps into a joint decision process. Depending on the location and data rate of the sources,

- each service template is scaled out into an overlay with the necessary number of instances required for each service component;
- each component instance is mapped to a network node and is allocated the required amount of resources on that node;
- the connections among component instances are mapped to flows along network links, carrying the data rate.

JASPER is an integrated approach in multiple dimensions: (i) scaling, placement, and routing decisions are made in a single optimization step; (ii) all services that are to be placed in the same substrate network are considered together; (iii) newly requested and already deployed services are optimized jointly. This way, a global optimum can be achieved. The main difference in handling newly requested and already deployed services is that for already deployed services, changes to the existing deployment may incur costs which have to be captured and optimized, thus adding one more optimization objective to the problem.

IV. PROBLEM MODEL

In this section, we formalize our model and define the problem we are tackling. Our model uses three different graphs for representing (i) the generic network service structure, (ii) a concrete and deployable instantiation of the network service, and (iii) the actual network. We use different names and notations to distinguish among these graphs (Table I).

Informally, the problem we address is as follows: given a substrate network, a set of – newly requested or already existing – network services with their templates, and the source(s) for the services in the network along with the traffic originating from them, we want to optimally embed the network services into the network.

A. Substrate network

We model the *substrate network* as a directed graph $G_{sub}=(V, L)$. Each *node* $v \in V$ is associated with a CPU capacity $cap_{cpu}(v)$ and a memory capacity $cap_{mem}(v)$ (this can be easily extended to other types of resources). Moreover, we assume that every node has routing capabilities and can

TABLE I
NOTATIONS USED FOR GRAPHS IN THE MODEL

| Graph	Symbol	Name	Annotations
Template G_{tmpl}	$j \in C_T$ $a \in A_T$	Component Arc	$\text{In}(j)$, $\text{Out}(j)$, p_j, m_j, r_j
Overlay G_{OL}	$i \in I_{\text{OL}}$ $e \in E_{\text{OL}}$	Instance Edge	$c(i)$, $P_T^{(I)}(i)$ $P_T^{(E)}(e)$
Network G_{sub}	$v \in V$ $l \in L$	Node Link	$\text{cap}_{\text{cpu}}(v)$, $\text{cap}_{\text{mem}}(v)$ $b(l)$, $d(l)$

Fig. 3. Some examples: (a) a template, (b) a component, (c) an overlay corresponding to the template, and (d) a mapping of the overlay into a substrate network. The links of the substrate network are bi-directional.

forward traffic to its neighboring nodes.[1] Each *link* $l \in L$ is associated with a maximum data rate $b(l)$ and a propagation delay $d(l)$. For each node v, we assume that the internal communications (e.g., communication inside a data center) can be done with unlimited data rate and negligible delay.

B. Templates

The substrate network has to host a set $\mathcal{T}$ of network services. We define the structure of each network service $T \in \mathcal{T}$ using a *template*, which is a directed acyclic graph $G_{\text{tmpl}}(T) = (C_T, A_T)$. We refer to the nodes and edges of the template graph as *components* and *arcs*, respectively. They define the type of components required in the network service and specify the way they should be connected to each other to deliver the desired functionality. Fig. 3(a) shows an example template.

A template component $j \in C_T$ has an ordered set of inputs, denoted as $\text{In}(j)$, and an ordered set of outputs, denoted as $\text{Out}(j)$. Its resource consumption depends on the data rates of the flows entering the component. We characterize this using a pair of functions $p_j, m_j : \mathbb{R}_{\geq 0}^{|\text{In}(j)|} \to \mathbb{R}_{\geq 0}$, where p_j is the CPU load and m_j is the required memory size of component j, depending on the data rate of the incoming flows. These functions typically account for resource consumption due to processing the input data flows as well as fixed, baseline consumption (even when idle). Similarly, data rates of the outputs of the component are determined as a function of the data rates on the inputs, specified as $r_j : \mathbb{R}_{\geq 0}^{|\text{In}(j)|} \to \mathbb{R}_{\geq 0}^{|\text{Out}(j)|}$. Fig. 3(b) shows examples for functions p_j, m_j, r_j that define the resource demands and output data rates of an example component.

Each arc in A_T connects an output of a component to an input of another component.

Source components are special components in the template: they have no inputs, a single output with unspecified data rate, and zero resource consumption. In the example of Fig. 3(a), S is a source component, whereas the others are normal processing components.

C. Overlays and sources

A template specifies the types of components and the connections among them as well as their resource demands

---

¹Capacities can be 0, e.g., to represent conventional switches by 0 CPU capacity or an end device by 0 forwarding capacity.
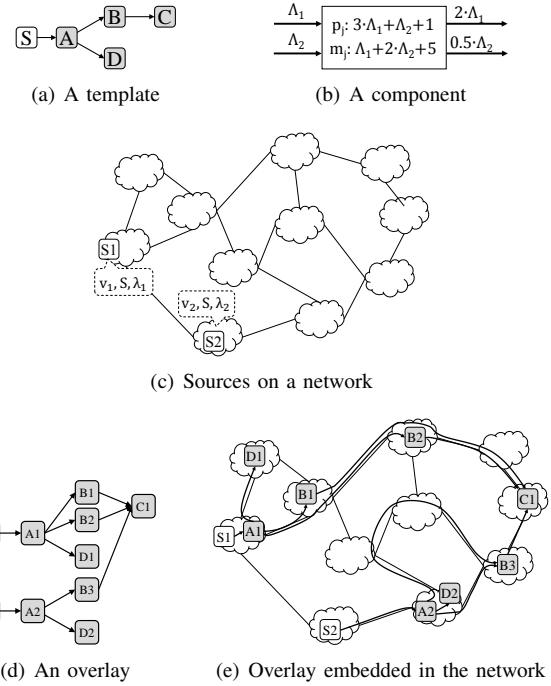
---

depending on the load. A specific, deployable instantiation of a network service can be derived by scaling its template, i.e., creating the necessary number of instances for each component and linking the instances with each other according to the requirements of the template. Depending on data rates of the service flows and the locations in the network where the flows start, different numbers of instances for each component might be required. To model this, for each network service T, we define a set of *sources* $S(T)$. The members of $S(T)$ are tuples of the form (v, j, λ), where $v \in V$ is a node of the substrate network, $j \in C_T$ is a source component, and $\lambda \in \mathbb{R}_+$ is the corresponding data rate assigned to the output of this source component. Such a tuple means that an instance of source component j generates a flow from node v with rate λ. Sources may represent populations of users, sensors, or any other component that can generate flows to be processed by the corresponding network service. Fig. 3(c) shows two example sources for the template of Fig. 3(a), located on different nodes of the substrate network. The naming of the concrete instances of components in this figure follows the convention that the first letter identifies the corresponding component in the template, e.g., S1 and S2 represent concrete instances of the source component S from the template.

An *overlay* is the outcome of scaling the template based on the associated sources. An overlay OL stemming from template T is described by a directed acyclic graph $G_{\text{OL}}(T) = (I_{\text{OL}}, E_{\text{OL}})$. Each component *instance* $i \in I_{\text{OL}}$ corresponds to a component $c(i) \in C_T$ of the underlying template. Each $i \in I_{\text{OL}}$ has the same characteristics (inputs, outputs, resource consumption characteristics) as $c(i)$. Moreover, if there is an edge from an output of an instance i_1 to an

input of instance i_2 in the overlay, then there must be a corresponding arc from the corresponding output of $c(i_1)$ to the corresponding input of $c(i_2)$ in the template. This ensures that the edge structure of the overlay is in line with the structural requirements of the network service, represented by the arcs in the template.

To be able to create the required number of instances for each component, we assume either that the components are stateless or that a state management system is in place to handle state redistribution upon adding or removing instances. In this way, requests can be freely routed to any instance of a component. Alternatively, additional details can be added to the model, for example, to make sure that the flows belonging to a certain session are routed to the right instance of stateful components that have stored the corresponding state information.

Fig. 3(d) shows an example overlay corresponding to the template in Fig. 3(a). The naming of the instances follows the same convention as described for sources, e.g., A1 and A2 are instances of component A.

An overlay might include multiple instances of a specific template component, e.g., B1, B2, and B3 all are instances of component B. An output of an instance can be connected to the input of multiple instances of the same component, like the output of A1 is connected to the inputs of B1 and B2. In a case like that, B1 and B2 share the data rate calculated for the connection between components A and B. Similarly, outputs of multiple instances in the overlay can be connected to the input of the same instance, like the input of C1 is connected to the output of B1, B2, and B3, in which case the input data rate for C1 is the sum of the output data rates of B1, B2, and B3.

D. Mapping on the substrate network

Each overlay $G_{OL}(T)$ must be mapped to the substrate network by a feasible mapping P_T. We define the mapping as a pair of functions $P_T = \left(P_T^{(I)}, P_T^{(E)} \right)$.

$P_T^{(I)} : I_{OL} \rightarrow V$ maps each instance in the overlay to a node in the substrate network. We make the simplifying assumption that two instances of the same component cannot be mapped to the same node. The rationale behind this assumption is that in this case it would be more efficient to replace the two instances by a single instance and thus save the idle resource consumption of one instance.[2]

$P_T^{(E)} : E_{OL} \rightarrow \mathcal{F}$ maps each edge in the overlay to a flow in the substrate network; $\mathcal{F}$ is the set of possible flows in G_{sub}. We assume the flows are splittable, i.e., can be routed over multiple paths between the corresponding endpoints in the substrate network.

The two functions must be compatible: if $e \in E_{OL}$ is an edge from an instance i_1 to an instance i_2, then $P_T^{(E)}(e)$ must be a flow with start node $P_T^{(I)}(i_1)$ and end node $P_T^{(I)}(i_2)$. Moreover, $P_T^{(I)}$ must map the instances of source components in accordance with the sources in $S(T)$, mapping an instance

corresponding to source component j to node v if and only if $\exists (v, j, \lambda) \in S(T)$.

The binding of instances of source components to sources determines the outgoing data rate of these instances. As the overlay graphs are acyclic, the data rate $\lambda(e)$ on each further overlay edge e can be determined based on the input data rates and the r_j functions of the underlying components, considering the instances in a topological order. The data rates, in turn, determine the resource needs of the instances.

Fig. 3(e) shows a possible mapping of the overlay of Fig. 3(d) to an example substrate network, based on the pre-defined location of S1 and S2 in the network. Note that it is possible to map two communicating instances to the same node, like A2 and D2 in the example. In this case, the edge between them can be realized inside the node, without using any links. The flow between A2 and B3 is an example of a split flow that is routed over two different paths in the substrate network.

Note that Fig. 3(e) shows only a single overlay mapped to the substrate network for the sake of clarity. In general, JASPER can embed several overlays corresponding to different network services into a substrate network.

E. Objectives

The *system configuration* consists of the overlays and their mapping on the substrate network. A new system configuration can be computed by an appropriate algorithm for the template embedding problem.

A valid system configuration must respect all capacity constraints: for each node v, the total resource needs of the instances mapped to v must be within its capacity concerning both CPU and memory, and for each link l, the sum of the flow values going through l must be within its maximum data rate. However, it is also possible that some of those constraints are violated in a given system configuration: for example, a valid system configuration (i.e., one without any violations) may become invalid because the data rate of a source has increased, because of a temporary peak in resource needs, or a failure in the substrate network. Therefore, given a current system configuration σ, our primary objective is to find a new system configuration σ', in which the *number of constraint violations is minimal* (ideally, zero). For this, we assume that violating node CPU, memory, and link capacity constraints is equally undesirable.

There are a number of further, secondary objectives, which can be used as tie-breaker to choose from system configurations that have the same number of constraint violations:

- Total delay of all edges across all overlays
- Number of instance addition/removal operations required to transition from σ to σ'
- Maximum amounts of capacity constraint violations, for each resource type (CPU, memory, link capacity)
- Total resource consumption of all instances across all overlays, for each resource type (CPU, memory, link capacity)

Higher values for these metrics result in higher costs for the system or in lower customer satisfaction, so our objective is to

---

[2]This simplification is mostly a technicality to simplify the problem write-up and could be extended if necessary.

minimize these values. Therefore, our aim is to select a new system configuration σ' from the set of system configurations with minimal number of constraint violations that is Pareto-optimal with respect to these secondary metrics.

F. Problem formulation summary

Our aim is to handle the scaling, placement, and routing for newly requested network services as well as already deployed network services. Taking this into account, the Template Embedding problem can be summarized as follows:

The inputs are the substrate network, a template for each network service, and the location and data rate of the sources for each network service. Additionally, for the already deployed network services, an overlay and its mapping onto the substrate network are given.

The outputs for the newly requested network services are the overlays and their mappings onto the substrate network. For the already deployed network services the modified overlays and their modified mappings onto the substrate network are calculated.

Scaling is performed while creating the overlay from the template, while placement and routing are performed when the instances and edges of the overlay are mapped onto the substrate network.

A further important detail concerns the relationship between different network services. The creation of the overlay from the template and its mapping onto the substrate network are defined for each network service separately; however, they share the same substrate network. The objectives defined in Sec. IV-E relate to the whole network including all network services, aiming for a global optimum and potentially resulting in trade-offs among the network services. A further connection among different network services may arise if they share the same component type. In this case, it is also possible that the corresponding overlay instances are realized by the same instance.

V. COMPLEXITY

Theorem 1. *For an instance of the Template Embedding problem as defined in Section IV, deciding whether a solution with no violations exists is NP-complete in the strong sense*[3].

Proof. It is clear that the problem is in NP: a possible witness for the positive answer is a solution – i.e., a set of overlays and their embedding into the substrate network – with 0 violations. The witness has polynomial size and can be verified in polynomial time wrt. to the input size.

To establish NP-hardness, we show a reduction from the Set Covering problem (which is known to be NP-complete in the strong sense [30]) to the Template Embedding problem. An input of the Set Covering problem consists of a finite set U, a finite family $\mathcal{W}$ of subsets of U such that their union is U,

---

[3]NP-complete in the strong sense means that the problem remains NP-complete even if the numbers appearing in it are constrained between polynomial bounds. Under the P≠NP assumption, this precludes even the existence of a pseudo-polynomial algorithm – i.e., an algorithm the runtime of which is polynomial if restricted to problem instances with polynomially bounded numbers.
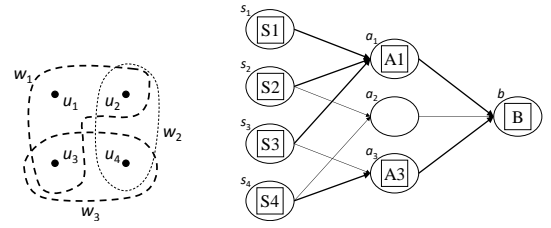
(a) An instance of Set Covering ($k = 2$) and a solution (thick lines)

(b) The generated instance of Template Embedding and the corresponding solution (placed instances and thick links)

Fig. 4. An example for the proof of Theorem 1. The elements $u_1 \ldots, u_4$ are represented by nodes $s_1, \ldots, s_4$, the sets w_1, w_2, w_3 by nodes a_1, a_2, a_3.

and a number $k \in \mathbb{N}$. The aim is to decide whether there is a subset $\mathcal{Z} \subseteq \mathcal{W}$ with cardinality at most k such that the union of the sets in $\mathcal{Z}$ is still U.

From this instance of Set Covering, an instance of the Template Embedding problem is created as follows. The substrate network consists of nodes $V = \{s_1, \ldots, s_{|U|}\} \cup \{a_1, \ldots, a_{|\mathcal{W}|}\} \cup \{b\}$, where each s_i represents an element of U and each element a_j represents an element of $\mathcal{W}$. There is a link from s_i to a_j if and only if the element of U represented by s_i is a member of the set represented by a_j. Furthermore, there is a link from each a_j to b. The capacities of the nodes are as follows: $\mathrm{cap}_{\mathrm{cpu}}(s_i) = \mathrm{cap}_{\mathrm{mem}}(s_i) = 0$ for each $i \in [1, |U|]$, $\mathrm{cap}_{\mathrm{cpu}}(a_j) = 0$ and $\mathrm{cap}_{\mathrm{mem}}(a_j) = 1$ for each $j \in [1, |\mathcal{W}|]$, and $\mathrm{cap}_{\mathrm{cpu}}(b) = 1$ and $\mathrm{cap}_{\mathrm{mem}}(b) = 0$. For each link, its maximum data rate is 1, its delay is 0.

There is a single template consisting of a source component S and two further components A and B, and two arcs (S, A) and (A, B). Component A has one input and one output, its resource consumption as a function of the input data rate λ is given by $p_A(\lambda) = 0$ and $m_A(\lambda) = 1$; its output data rate is given by $r_A(\lambda) = 1$. Component B has one input and no output, its resource consumption as a function of the input data rate λ is given by

$$p_B(\lambda) = \begin{cases} 1, & \text{if } \lambda \le k, \\ 2, & \text{otherwise,} \end{cases}$$

and $m_B(\lambda) = 0$. In each s_i, there is a source corresponding to an instance of S with data rate $\lambda = 1$.

Suppose first that the original instance of Set Covering is solvable, i.e., there is a subset $\mathcal{Z} \subseteq \mathcal{W}$ with cardinality at most k such that the union of the sets in $\mathcal{Z}$ is U. In this case, the generated instance of the Template Embedding problem can also be solved without any violations, as follows (see Fig. 4 for an example). Each s_i must of course host an instance of S. In each a_i corresponding to an element of $\mathcal{Z}$, an instance of A is created. Since the union of the sets in $\mathcal{Z}$ is U, each s_i has an outgoing link to at least one a_j hosting an instance of A, which can be selected as the target of the traffic leaving the source in s_i through the link (s_i, a_j). Further, a single instance of B is created in node b and each instance of A is connected to B through the (a_j, b) link. Since the number of instances of A is at most k, each emitting traffic with data rate 1, the CPU requirement of the instance of B is 1, so that it fits on b,

and hence we obtained a solution to the Template Embedding problem with no violation.

Now assume that the generated instance of the Template Embedding problem is solvable without violations. Then, we can construct a solution of the original instance of Set Covering, as we show next. In a solution of the generated instance of the Template Embedding problem, each s_i must host an instance of S and there is no other instance of S. Instances of A can only be hosted by a_j nodes because of the memory requirement, and an instance of B can only be hosted in b because of the CPU requirement. We define $\mathcal{Z}$ to contain those elements of $\mathcal{W}$ for which the corresponding node a_j hosts an instance of A. Since each source generates traffic that must be consumed by an instance of A and there is a path (actually, a link) from s_i to a_j only if the set corresponding to a_j contains the element corresponding to s_i, it follows that the sets in $\mathcal{Z}$ cover all elements of U. Moreover, since the instance of B must fit on b and each instance of A generates traffic with data rate 1, it follows that the number of instances of A is at most k and hence $|\mathcal{Z}| \leq k$, thus $\mathcal{Z}$ is a solution of the original Set Covering problem.

Since all numbers in the generated instance of the Template Embedding problem are constants, this reduction shows that the Template Embedding problem is indeed NP-hard in the strong sense. □

As a consequence, we can neither expect a polynomial or even pseudo-polynomial algorithm for solving the problem exactly nor a fully polynomial-time approximation scheme, under standard assumptions of complexity theory.

VI. MIXED INTEGER PROGRAMMING APPROACH

In this section, we provide a mixed integer programming (MIP) formulation of the problem. On one hand, this serves as a further formalization of the problem; on the other hand, under suitable assumptions (to be detailed in Section VI-C) an appropriate solver can be used to solve the mixed integer program, yielding an algorithm for the problem.

Based on the assumption that two instances of the same component cannot be mapped to a node, instances can be identified by the corresponding component and the hosting node. This is the basis for our choice of variables, which are explained in more detail in Table II.

We use the following notations for formalizing the constraints and objectives. $C=\bigcup_{T \in \mathcal{T}} C_T$ denotes the set of all components, $\mathcal{A}=\bigcup_{T \in \mathcal{T}} A_T$ the set of all arcs, and $\mathcal{S}=\bigcup_{T \in \mathcal{T}} S(T)$ the set of all sources across all network services that we want to map to the network. M, M_1, and M_2 denote sufficiently large constants. $(\Lambda_{j,v})_k$ denotes the kth component of the vector $\Lambda_{j,v}$. $\underline{0}$ denotes a zero vector of appropriate length.

Information about existing instances should also be taken into account during the decision process. For this, we define $x^*_{j,v} (\forall j \in C, v \in V)$ as a constant given as part of the problem input. If there is a previously mapped instance of component j on node v in the network, $x^*_{j,v}$ is 1, otherwise it is 0.

TABLE II
VARIABLES

Name	Domain	Definition
$x_{j,v}$	$\{0,1\}$	1 iff an instance of component $j \in C$ is mapped to node $v \in V$
$y_{a,v,v'}$	$\mathbb{R}_{\geq 0}$	If $a \in A_T$ is an arc from an output of $j' \in C_T$ to an input of $j' \in C_T$, an instance of j is mapped on $v \in V$, and an instance of j' is mapped on $v' \in V$, then $y_{a,v,v'}$ is the data rate of the corresponding flow from v to v'; otherwise it is 0
$z_{a,v,v',l}$	$\mathbb{R}_{\geq 0}$	If $a \in A_T$ is an arc from an output of $j \in C_T$ to an input of $j' \in C_T$, an instance of j is mapped on $v \in V$, and an instance of j' is mapped on $v' \in V$, then $z_{a,v,v',l}$ is the data rate of the corresponding flow from v to v' that goes through link $l \in L$; otherwise it is 0
$\Lambda_{j,v}$	$\mathbb{R}_{\geq 0}^{|\text{In}(j)|}$	Vector of data rates on the inputs of the instance of component $j \in C_T$ on node $v \in V$, or an all-zero vector if no such instance is mapped on v
$\Lambda'_{j,v}$	$\mathbb{R}_{\geq 0}^{|\text{Out}(j)|}$	Vector of data rates on the outputs of the instance of component $j \in C_T$ on node $v \in V$, or an all-zero vector if no such instance is mapped on v
$\varrho_{j,v}$	$\mathbb{R}_{\geq 0}$	CPU requirement of the instance of component $j \in C_T$ on node $v \in V$, or zero if no such instance is mapped on v
$\mu_{j,v}$	$\mathbb{R}_{\geq 0}$	Memory requirement of the instance of component $j \in C_T$ on node $v \in V$, or zero if no such instance is mapped on v
$\omega_{v,\text{cpu}}$	$\{0,1\}$	1 iff the CPU capacity of node $v \in V$ is exceeded
$\omega_{v,\text{mem}}$	$\{0,1\}$	1 iff the memory capacity of node $v \in V$ is exceeded
ω_l	$\{0,1\}$	1 iff the maximum data rate of link $l \in L$ is exceeded
ψ_{cpu}	$\mathbb{R}_{\geq 0}$	Maximum CPU over-subscription over all nodes
ψ_{mem}	$\mathbb{R}_{\geq 0}$	Maximum memory over-subscription over all nodes
ψ_{dr}	$\mathbb{R}_{\geq 0}$	Maximum capacity over-subscription over all links
$\zeta_{a,v,v',l}$	$\{0,1\}$	1 iff $z_{a,v,v',l} > 0$
$\delta_{j,v}$	$\{0,1\}$	1 iff $x_{j,v} \neq x^*_{j,v}$

A. Constraints

Here we define the sets of constraints that enforce the required properties of the template embedding process.

1) Mapping consistency rules:

$$\forall (v,j,\lambda) \in \mathcal{S}: \qquad x_{j,v} = 1 \qquad (1)$$

$$\forall (v,j,\lambda) \in \mathcal{S}: \qquad \Lambda'_{j,v} = \lambda \qquad (2)$$

$$\forall j \in C, \forall v \in V, k \in [1, |\text{In}(j)|]: \qquad (\Lambda_{j,v})_k \leq M \cdot x_{j,v} \quad (3)$$

$$\forall j \in C, \forall v \in V, k \in [1, |\text{Out}(j)|]: \qquad (\Lambda'_{j,v})_k \leq M \cdot x_{j,v} \quad (4)$$

$$\forall j \in C, \forall v \in V: \qquad x_{j,v} - x^*_{j,v} \leq \delta_{j,v} \qquad (5)$$

$$\forall j \in C, \forall v \in V: \qquad x^*_{j,v} - x_{j,v} \leq \delta_{j,v} \qquad (6)$$

Constraints (1) and (2) enforce that the placement respectively the output data rate of source component instances are in line with the tuples specified in $\mathcal{S}$. Constraint (3) guarantees the consistency between the variables $\Lambda_{j,v}$ and $x_{j,v}$: if $\Lambda_{j,v}$ has a positive component, then $x_{j,v}$ must be 1, i.e., only an existing component instance can process the incoming flow. Constraint (3) is analogous for the outgoing flows, represented by the $\Lambda'_{j,v}$ variables. Constraints (5) and (6) together ensure that $\delta_{j,v} = 1$ if and only if $x_{j,v} \neq x^*_{j,v}$.

2) Flow and data rate rules:

$\forall j \in C, j$ not a source component, $\forall v \in V :$
$$\Lambda'_{j,v} = r_j(\Lambda_{j,v}) - (1 - x_{j,v}) \cdot r_j(\underline{0}) \quad (7)$$

$\forall j \in C, \forall v \in V, k \in [1, |\text{In}(j)|]:$

$$(\Lambda_{j,v})_k = \sum_{a \text{ ends in input } k \text{ of } j, v' \in V} y_{a,v',v} \quad (8)$$

$\forall j \in C, \forall v \in V, k \in [1, |\text{Out}(j)|]:$

$$(\Lambda'_{j,v})_k = \sum_{a \text{ starts in output } k \text{ of } j, v' \in V} y_{a,v,v'} \quad (9)$$

$\forall a \in \mathcal{A}, \forall v, v_1, v_2 \in V:$

$$\sum_{vv' \in L} z_{a,v_1,v_2,vv'} - \sum_{v'v \in L} z_{a,v_1,v_2,v'v} =$$

$$= \begin{cases} 0 & \text{if } v \neq v_1 \text{ and } v \neq v_2 \\ y_{a,v_1,v_2} & \text{if } v = v_1 \text{ and } v_1 \neq v_2 \\ 0 & \text{if } v = v_1 = v_2 \end{cases} \quad (10)$$

$\forall a \in \mathcal{A}, \forall v, v' \in V, \forall l \in L: \quad z_{a,v,v',l} \leq M \cdot \zeta_{a,v,v',l} \quad (11)$

Constraint (7) computes the data rate on the outputs of a processing component instance based on the data rates on its inputs and the r_j function of the underlying component. The constraint is formulated in such a way that for $x_{j,v} = 1$, $\Lambda'_{j,v} = r_j(\Lambda_{j,v})$, whereas for $x_{j,v} = 0$ (in which case also $\Lambda_{j,v} = 0$ because of Constraint (3)), also $\Lambda'_{j,v} = 0$ so that there is no contradiction with Constraint (4). Constraint (8) computes the data rate on the inputs of a component instance as the sum of the data rates on the links ending in that input. Similarly, Constraint (9) ensures that the data rate on the outputs of a component instance is distributed on the links starting in that output. Constraint (10) is the flow conservation rule, also ensuring the right data rate of each flow, thus connecting the $z_{a,v,v',l}$ variables (flow values on individual links) and the $y_{a,v,v'}$ variables (flow data rate). Constraint (11) sets the $\zeta_{a,v,v',l}$ variables (on the basis of the $z_{a,v,v',l}$ variables), so that they can be used later on in the objective function (Section VI-B).

3) Calculation of resource consumption:

$\forall j \in C, \forall v \in V: \quad \varrho_{j,v} = p_j(\Lambda_{j,v}) - (1 - x_{j,v}) \cdot p_j(\underline{0}) \quad (12)$

$\forall j \in C, \forall v \in V: \quad \mu_{j,v} = m_j(\Lambda_{j,v}) - (1 - x_{j,v}) \cdot m_j(\underline{0}) \quad (13)$

Constraints (12) and (13) calculate CPU respectively memory consumption of each component instance based on the p_j and m_j functions of the underlying component[4]. The logic here is analogous to that of Constraint (7).

4) Capacity constraints:

$\forall v \in V: \quad \sum_{j \in C} \varrho_{j,v} \leq \text{cap}_{\text{cpu}}(v) + M \cdot \omega_{v,\text{cpu}} \quad (14)$

$\forall v \in V: \quad \sum_{j \in C} \varrho_{j,v} - \text{cap}_{\text{cpu}}(v) \leq \psi_{\text{cpu}} \quad (15)$

$\forall v \in V: \quad \sum_{j \in C} \mu_{j,v} \leq \text{cap}_{\text{mem}}(v) + M \cdot \omega_{v,\text{mem}} \quad (16)$

$\forall v \in V: \quad \sum_{j \in C} \mu_{j,v} - \text{cap}_{\text{mem}}(v) \leq \psi_{\text{mem}} \quad (17)$

$\forall l \in L: \quad \sum_{a \in \mathcal{A}; v, v' \in V} z_{a,v,v',l} \leq b(l) + M \cdot \omega_l \quad (18)$

$\forall l \in L: \quad \sum_{a \in \mathcal{A}; v, v' \in V} z_{a,v,v',l} - b(l) \leq \psi_{\text{dr}} \quad (19)$

[4]Adding more resource types would be reflected by adding corresponding constraints here.

The aim of these constraints is to set the ω and ψ variables (based on the already defined ϱ, μ and z variables), which will be used in the objective function (Section VI-B). Constraint (14) ensures that $\omega_{v,\text{cpu}}$ will be 1 if the CPU capacity of node v is overloaded, while Constraint (15) ensures that ψ_{cpu} will be at least as high as the amount of CPU overload of any node (the appearance of ψ_{cpu} in the objective function will guarantee that it will be exactly the maximum amount of CPU overload and not higher than that). Constraints (16), (17) do the same for memory overloads and Constraints (18), (19) do the same for the overload of link capacity.

5) Interplay of the constraints:
To illustrate the interplay of the constraints, we assume that we need to optimize the embedding shown in Fig. 3(e). Constraints (1) and (2) ensure that instances of the source component, i.e., S1 and S2, are embedded and their output data rates are set correctly. Constraint (9) ensures that these data rates are then handed out as flows that can only end up in instances of A. These flows are mapped to network links and instances of A are assigned input data rates using Constraints (10) and (8), respectively. That being set, Constraint (3) marks the instances A1 and A2 as embedded, and Constraint (7) sets their output data rates using the respective r_j function. In a similar way, the rest of the components are instantiated and embedded in the network.

Constraints (5) and (6) ensure that the $\delta_{j,v}$ variables are set correctly. Constraints (12) and (13) compute the resource consumption of each instance based on the input data rates and the corresponding p_j and m_j functions. Constraints (14)–(19) make sure that over-subscription of node and link capacities are captured correctly, and collect the maximum value of over-subscription for each resource type. This maximum value is used in the objective function described in Section VI-B, which drives the decisions based on the constraints.

B. Optimization objective

We formalize the optimization objective based on the goals defined in Section IV-E as follows:

$$\text{minimize} \quad M_1 \cdot \left(\sum_{v \in V} (\omega_{v,\text{cpu}} + \omega_{v,\text{mem}}) + \sum_{l \in L} \omega_l \right) +$$

$$+ M_2 \cdot \left(\sum_{\substack{a \in \mathcal{A} \\ v,v' \in V \\ l \in L}} (d(l) \cdot \zeta_{a,v,v',l}) + \sum_{\substack{j \in C \\ v \in V}} \delta_{j,v} \right) +$$

$$+ \psi_{\text{cpu}} + \psi_{\text{mem}} + \psi_{\text{dr}} + \sum_{\substack{j \in C \\ v \in V}} (\varrho_{j,v} + \mu_{j,v}) + \sum_{\substack{a \in \mathcal{A} \\ v,v' \in V \\ l \in L}} z_{a,v,v',l} \quad (20)$$

By assigning sufficiently large values to M_1 and M_2, we can achieve the following goals with the given priorities (in decreasing order): 1) Number of capacity constraint violations over all nodes and links is minimized. 2) Template arcs are mapped to network paths in such a way that their total latency is minimized. Moreover, the number of instances that need to be started/stopped is minimized. 3) The maximum value for capacity constraint violations over all nodes and links is minimized. Also, overlay instances and the edges among

them are created in a way that their resource consumption is minimized.

The objective function is in line with the objectives defined in Section IV-E. The primary objective is to minimize the number of constraint violations; a sufficiently large M_1 ensures that a decrease in the first term of the objective function has larger impact than any change in the other terms. Moreover, the resulting solution σ' will be Pareto-optimal with respect to the other, secondary metrics: otherwise, there would be another solution σ'' that is as good as σ' according to each secondary metric and strictly better than σ' in at least one secondary metric, but then, σ'' would lead to a lower overall value of the objective function.

This mixed integer program can be used for initial embedding of service templates as well as for optimizing existing embeddings. However, for the initial embedding of newly requested network services, the term $\sum_{j \in C, v \in V} \delta_{j,v}$ should be removed from the objective function because it would introduce an unwanted bias towards embeddings with fewer instances, although it is possible that having more instances can decrease the overall cost of the solution.

C. Solving the mixed integer program

All our constraints are linear equations and linear inequalities, and also the objective function is linear. Hence, if the functions p_j, m_j, and r_j are linear for all $j \in C$, then we obtain a mixed-integer linear program (MILP), which can be solved by appropriate solvers. For non-linear functions, a piecewise linear approximation may make it possible to use MILP solvers to obtain good (although not necessarily optimal) solutions.

VII. HEURISTIC APPROACH

Now we present a heuristic algorithm that is not guaranteed to find an optimal solution but is much faster than the mixed integer programming approach. Moreover, it has the advantage that it does not require the functions p_j, m_j, and r_j to be linear.

The heuristic constructs the new solution from the existing one by means of a series of small local changes.[5] While doing so, it has to be ensured that (i) the instantiation of source components is in line with the given data sources, (ii) the data flows produced by each instance are routed to appropriate instances, and (iii) capacity constraints are satisfied as much as possible. This can be achieved by iterating through the instances of each overlay once in a topological order, possibly creating new instances on the fly if necessary. Note that this may indeed be necessary, for example, if a new data source appeared or the output data rate of a data source increased. A topological order of a directed graph $G = (V, E)$ is an order of its vertices as $v_1, v_2, \ldots, v_n$ such that for each edge $v_i v_j \in E$, $i < j$ holds; in other words, each edge points "forward." For any directed acyclic graph, a topological order can be determined in linear time. In each step, the algorithm aims at economical use of resources, e.g., by only creating

[5] Also the placement of a new service is done with a series of small local changes, creating component instances one by one.

Algorithm 1 Main procedure of the heuristic algorithm

1: **if** $\exists G_{\text{OL}}(T)$ with $T \notin \mathcal{T}$ **then**
2:      remove $G_{\text{OL}}(T)$
3: **for all** $T \in \mathcal{T}$ **do**
4:      **if** $\nexists G_{\text{OL}}(T)$ **then**
5:          create empty overlay $G_{\text{OL}}(T)$
6:      **for all** $(v, j, \lambda) \in S(T)$ **do**
7:          **if** $\nexists i \in I_{\text{OL}}$ with $c(i) = j$ and $P_T^{(I)}(i) = v$ **then**
8:              create $i \in I_{\text{OL}}$ with $c(i) = j$ and $P_T^{(I)}(i) = v$
9:          set output data rate of i to λ
10:     **if** $\exists i \in I_{\text{OL}}$, where $c(i)$ is a source component but $\nexists (P_T^{(I)}(i), c(i), \lambda) \in S(T)$ for any λ **then**
11:         remove i
12:     **for all** $i \in I_{\text{OL}}$ in topological order **do**
13:         **if** all input data rates of i are 0 **then**
14:             remove i and go to next iteration
15:         compute output data rates of i
16:         **for all** output k of i **do**
17:             Φ: set of flows currently leaving output k
18:             λ: sum of the data rates of the flows in Φ
19:             λ': new data rate on output k
20:             **if** $\lambda' < \lambda$ **then**
21:                 $\mathcal{E}$: set of edges leaving output k
22:                 DECREASE($\mathcal{E}, \lambda - \lambda'$)
23:             **else if** $\lambda' > \lambda$ **then**
24:                 INCREASE($i, k, \Phi, \lambda' - \lambda$)

---

new instances if necessary, deleting unneeded instances, or preferring short paths.

The heuristic is shown in Algorithm 1. It starts by checking that each service has a corresponding overlay and each overlay corresponds to a service (lines 1–5). If a new service has been started or an existing service has been stopped since the last invocation of the algorithm, the corresponding overlay is created or removed at this point.

Next, the mapping of the sources and source components is checked and updated if necessary (lines 6–11): if a new source emerged, an instance of the corresponding source component is created; if the data rate of a source changed, then the output data rate of the corresponding source component instance is updated; if a source disappeared, then the corresponding source component instance is removed.

Finally, to propagate the changes of the sources to the processing instances, we need to iterate over all instances and ensure that the new output data rates, which are determined by the new input data rates, are discharged correctly by outgoing flows (lines 12–24). For this purpose, it is important to consider the instances in a topological order (according to the overlay) so that when an instance is dealt with, its incoming flows have already been updated. If a change in the outgoing flows is necessary, then the INCREASE or DECREASE procedures are called.

The auxiliary subroutines are detailed in Algorithm 2. DECREASE first removes as many edges as possible (lines 3–6); when a further decrease is necessary but no more edges can be removed, it reduces the next flow on each link by the same factor to achieve the required reduction (lines 7–9). INCREASE first checks if new instances need to be created to be consistent with the template (lines 12–16), then tries to

---

Algorithm 2 Auxiliary methods of the heuristic

1: /* *Decrease the flows on the edges in $\mathcal{E}$ by $\Delta\lambda$ in total* */
2: **procedure** DECREASE($\mathcal{E},\Delta\lambda$)
3:     sort $\mathcal{E}$ in non-decreasing order of flow data rate
4:     **for all** $e \in \mathcal{E}$ while flow data rate $\lambda(e) \leq \Delta\lambda$ **do**
5:         $\Delta\lambda := \Delta\lambda - \lambda(e)$
6:         remove e
7:     **if** $\Delta\lambda > 0$ **then**
8:         let e be the next edge
9:         reduce flow of e by a factor of $(\lambda(e) - \Delta\lambda)/\lambda(e)$
10: /* *Increase the flows in Φ leaving output k of instance i by $\Delta\lambda$ in total* */
11: **procedure** INCREASE($i,k,\Phi,\Delta\lambda$)
12:     **for all** arc $(c(i), j)$ leaving output k of $c(i)$ **do**
13:         **if** $\nexists i' \in I_{OL}$ with $c(i') = j$ and $ii' \in E_{OL}$ **then**
14:             $\varphi :=$ CREATEINSTANCEANDFLOW($j, i, \Delta\lambda$)
15:             $\Delta\lambda := \Delta\lambda -$ (data rate of φ)
16:             $\Phi := \Phi \cup \{\varphi\}$
17:     **for all** $\varphi \in \Phi$ **do**
18:         $d :=$ INCRFLOW($\varphi,\Delta\lambda$)
19:         $\Delta\lambda := \Delta\lambda - d$
20:     **while** $\Delta\lambda > 0$ **do**
21:         $(c(i), j)$: random arc leaving output k of $c(i)$
22:         $\varphi :=$ CREATEINSTANCEANDFLOW($j, i, \Delta\lambda$)
23:         $\Delta\lambda := \Delta\lambda -$ (data rate of φ)
24: /* *Create an instance of component j with flow from instance i of high data rate (capped at cutoff)* */
25: **procedure** CREATEINSTANCEANDFLOW($j,i,$cutoff)
26:     **for all** $v \in V$ **do**
27:         create temporary instance i' of j on v
28:         φ: flow of data rate 0 from i to i'
29:         INCRFLOW($\varphi,$cutoff)
30:         remove i' and φ
31:     create instance of j on node resulting in best flow
32: /* *Increase flow data rate by at most d* */
33: **procedure** INCRFLOW(φ,d)
34:     $v :=$ start node of φ
35:     $v' :=$ end node of φ
36:     $\beta_1 :=$ maximum flow based on $cap_{CPU}(v')$
37:     $\beta_2 :=$ maximum flow based on $cap_{mem}(v')$
38:     $d := \min(d, \beta_1, \beta_2)$
39:     P: $v \rightsquigarrow v'$ path of high bandwidth (b) and low latency
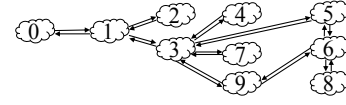40:   increase φ by $\min(b, d)$ along P

Fig. 5. Example substrate network

be noted that split flows can still be created if INCRFLOW is run multiple times for a flow.

When improving a flow and when selecting from multiple possible flows, the INCRFLOW and CREATEINSTANCEAND-FLOW routines must strike a balance between flow data rate and the increase in overall delay of the solution. Our strategy for comparing two possible flows is to first compare their data rates and compare their latencies only if there is a tie. This strategy is used in line 31 to select the best flow. The rationale is that selecting flows with high data rate leads to a small number of instances to be created. However, we also employ a cutoff mechanism: flow data rates above the cutoff (the increase in data rate that we want to achieve) do not add more value and are hence regarded to be equal to the cutoff value. This increases the likelihood of a tie, so that the tie-breaking method of preferring lower latencies is also important. An analogous strategy is used in line 39 to compare paths: the primary criterion is to prefer paths with higher bandwidth – up to the given cutoff d – and, in case of a tie, to prefer paths with lower latency. For finding the best path, a modified best-first-search is used, in which the nodes to be visited are stored in a priority queue, where priority is defined in accordance with the comparison relation described above.

VIII. EVALUATION

We implemented the presented algorithms in the form of a C++ program. For solving the MILP, Gurobi Optimizer 7.0.1[6] was used. For substrate networks, we used benchmarks for the Virtual Network Mapping Problem[7] from Inführ and Raidl [33]. As service templates, we used examples from IETF's Service Function Chaining Use Cases [34].

A. An example

First, we illustrate our approach on a small substrate network of 10 nodes and 20 links (see Fig. 5) in which the CPU and memory capacity of each node is both 100. In this network, a service consisting of a source (S), a firewall (FW), a deep packet inspection (DPI) component, an anti-virus (AV) component, and a parental control (PC) component is deployed. Initially, there is a single source in node 1 with a moderate data rate. As a result, our algorithm deploys all components of the service in node 1 (see Fig. 6(a)).

Subsequently, the data rate of the source increases. As a result, the resource demand of the processing components of the service increases so that they do not fit onto node 1 anymore. Our algorithm automatically re-scales the service by duplicating the DPI, AV, and PC components and automatically places the newly created instances on a nearby node, namely node 3 (see Fig. 6(b)).

increase the existing flows (lines 17–19). If this is not sufficient to achieve the necessary increase, it creates further instances and flows (lines 20–23).

In the CREATEINSTANCEANDFLOW procedure (called by INCREASE to create a new instance of a component together with a flow from an existing instance), all nodes of the substrate network are temporarily tried for hosting the new instance. The candidate that leads to the best flow is selected (lines 26–31). Finally, the INCRFLOW procedure (called by both INCREASE and CREATEINSTANCEANDFLOW) increases the data rate of a flow along a new path (lines 34–40).

As can be seen, we avoid computing maximum flows. This is because the running time of the best known algorithms for this purpose are worse than quadratic with respect to the size of the graph [31]. Since these subroutines are run many times, the high time complexity would be problematic for large substrate networks. Instead, each run of INCRFLOW increases a flow only along one new path. For finding the path, a modified best-first-search [32] is used, which runs in linear time. It should

(a) Initial embedding (b) Result of increased source data rate (c) Result of the emergence of a second source
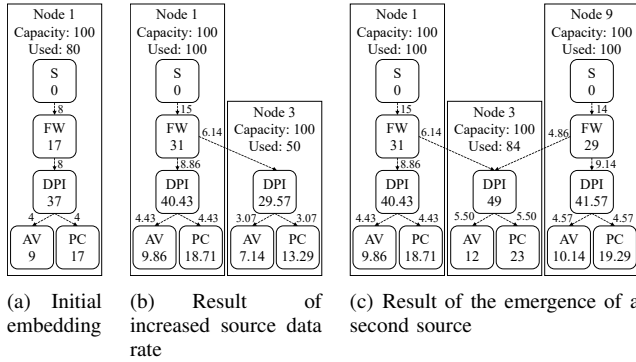
Fig. 6. Illustrative example: Each component is shown with its CPU demand (memory values not shown for better readability)
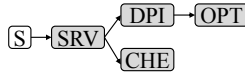

Fig. 7. Service template, with sources of data (S), streaming server (SRV), deep packet inspector (DPI), video optimizer (OPT), and cache (CHE)

Later on, a second source emerges for the same service on node 9. The algorithm automatically decides to create new processing component instances on node 9 to process as much as possible of the traffic of the new source locally. The excess traffic from the new FW instance that cannot be processed locally due to capacity constraints is routed to the existing DPI, AV, and PC instances on node 3 because node 3 still has sufficient free capacity (see Fig. 6(c)).

Already this small example shows the difficult trade-offs that template embedding involves. Next, we show that our approach is capable of handling also much more complex scenarios.

B. Comparison of the algorithms

We consider a substrate network with 20 nodes and 44 links, in which multiple services are deployed. Each service is a
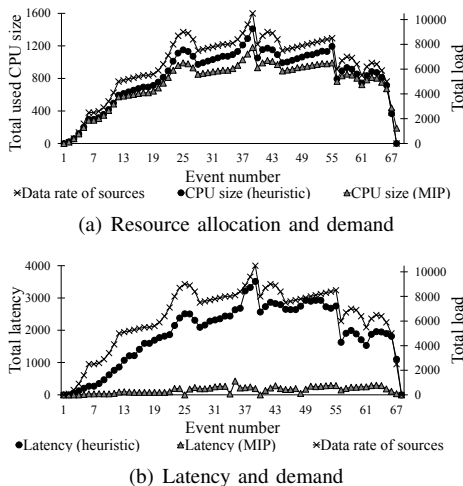

(a) Resource allocation and demand

(b) Latency and demand

Fig. 8. Temporal development of the demand, the allocated capacity, and the total latency in a complex scenario

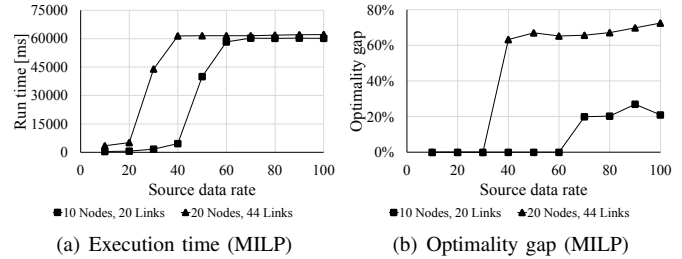(a) Execution time (MILP) (b) Optimality gap (MILP)

Fig. 9. Scalability of the presented algorithms

virtual content delivery network for video streaming, consisting of a streaming server, a DPI, a video optimizer, and a cache. The service template is shown in Fig. 7. The number of concurrently active services varies from 0 to 4, the number of sources varies from 0 to 20. Fig. 8(a) shows how the total data rate of the sources (as a metric of the demand) and the total CPU size of the created instances (as a metric of the allocated processing capacity) change through re-optimization after each event. An event is the emergence or disappearance of a service, the emergence or disappearance of a source, or the change of the data rate of a source. As can be seen, the allocated capacity using both the heuristic and the MILP algorithms follow the demand very closely, meaning that our algorithms are successful in scaling the service in both directions to quickly react to changes in the demand.

Regarding total data rate and total latency of the overlay edges, the MILP algorithm performs better than the heuristic algorithm. For example, Fig. 8(b) shows the total latency over all paths created for the template in this scenario[8]. The reason for this difference is that in the MILP algorithm, the optimal location for all required instances can be determined at the same time. This results in shorter distances between the source and the instances. The heuristic algorithm, however, needs to create instances one by one, resulting in larger data rates traveling over larger distances in the substrate network.

In this scenario, to handle the peak demand, a total of 127 instances are created using the MILP algorithms, while the heuristic algorithm creates 261 instances.

C. Scalability

Since the template embedding problem is NP-hard, it is foreseeable that the scalability of the MILP solver will be limited. In order to test this, we gradually increase the source data rate of the service from our first experiment, leading to an increasing number of instances; moreover, we also consider substrate networks of increasing size. In each case, the MILP solver is run with a time limit of 60 seconds, meaning that the solution process stops at (roughly) 60 seconds with the best solution and the best lower bound that the solver found until that time. The measurements were performed on a machine with Intel Core i5-4210U CPU @ 1.70GHz and 8GB RAM.

---

[8]In Fig. 8(b), in the high-load area between event 20 and 50, some problem instances are too complex to be solved within the 60 seconds time limit we have set for the optimizer. This results in solutions with zero latency, as no paths are created.

Fig. 9(a) shows the execution time of the MILP algorithm for different data rates and substrate network sizes, while Fig. 9(b) shows the corresponding gap between the found solution and the lower bound. For a small network with 10 nodes and 20 links, the algorithm computes optimal results for the lower half of source data rate values, and even for larger source data rates, the optimality gap is quite low (around 20 %), meaning that the results are almost optimal. However, for a bigger substrate network with 20 nodes and 44 links, the solver reaches the time limit for much smaller source data rate and also the optimality gap is much bigger. For even bigger substrate networks, the performance of the algorithm further deteriorates, up to the point where it cannot be run anymore because of memory problems. The large sensitivity to the size of the substrate network is not surprising, given that the number of variables of the MILP is cubic in the size of the substrate network.

In contrast, the execution time of the heuristic algorithm remains very low even for the largest substrate networks: for 1000 nodes and 2530 links, the execution time is still below 20 milliseconds, rendering the heuristic practical for real-world problem sizes as well.

D. Analysis

To gain further insight into the inter-dependencies between the input and output parameters of the Template Embedding problem, we experimented with different problem sizes and different levels of resource availability. For this, we used the video streaming template from Section VIII-B (Fig. 7) with a single source injecting a total data rate of 1000 units into the network, from a node selected randomly and uniformly. We used three substrate networks: (i) 200 nodes and 472 links, (ii) 500 nodes and 1288 links, (iii) 1000 nodes and 2530 links. We created low-resource and high-resource configurations as follows[9]:

- CPU and memory capacities of each node were selected randomly and uniformly from the range [1,5] for low capacity and [10,50] for high capacity.
- Data rate of each link was selected randomly and uniformly from the range [50,100] for low capacity and [500,1000] for high capacity.

We ran the heuristic algorithm 100 times on each setup (low/high node capacity and low/high link capacity) and each substrate network. Fig. 10 shows some aggregated results, with confidence intervals at 95% confidence level.

As shown in Fig. 10(a), the algorithm adapts the amount of used data rate to the amount of available link capacity. On all three networks, in both setups with low link capacity, links are carrying considerably less data rate than in the setups with high link capacity. This figure also shows that the algorithm uses more link capacity for embedding the same template as the network gets larger, increasing the total available link capacity in the network. With low link capacity, the algorithm

---

[9]These ranges were selected based on the amount of resources required to embed the template with exactly one instance per component, as an estimation of the required resources for handling the input data rate.

(a) Total used data rate
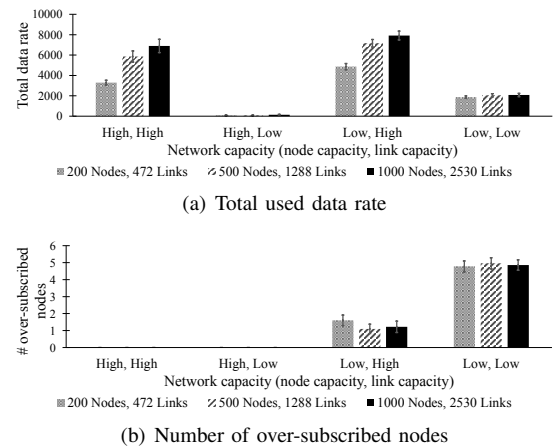

(b) Number of over-subscribed nodes

Fig. 10. Impact of different node and link capacities

concentrates the instances on as few nodes as possible; therefore, the majority of the load remains inside nodes, instead of traveling across the network. Obviously, this can result in over-loaded nodes if the node capacities are not enough. Fig. 10(b) shows the number of network nodes with over-subscribed CPU capacities. In the setups with high node capacity, no CPU over-subscription is noticed. With low node capacity and low link capacity, the instances are concentrated in fewer nodes resulting in more node over-subscription. With low node capacity, even with high link capacity, over-subscription could not be avoided in our experiments. However, fewer nodes were affected, as the the data rates could be distributed more freely across the network.

IX. CONCLUSIONS

We have presented JASPER, a fully automatic approach to scale, place, and route multiple virtual network services on a common substrate network. JASPER can be used for both the initial allocation of newly requested services and the adaptation of existing services to changes in the demand. Besides formally defining the problem and proving its NP-hardness, we developed two algorithms for it, an MILP-based one and a custom constructive heuristic. Empiric tests have shown how our approach finds a balance between conflicting requirements and ensures that the allocated capacity quickly follows changes in the demand. Moreover, we have shown that our solutions can adapt the embedding to the amount of available resources in the network, e.g., using less link capacity and concentrating the instances on less nodes when links have a low capacity and nodes have enough capacity.

The MILP-based algorithm gives optimal or near-optimal results for relatively small substrate network graphs, making it suitable for, e.g., calculations on top of a geographically distributed network where each node represents a data center. The heuristic remains very fast for even the largest networks that were tested. Overall, the tests gave evidence to the feasi-bility of our approach, which makes it possible (i) for service developers to specify services at a high level of abstraction and (ii) for providers to quickly re-optimize the system state after changes.

References

[1] ETSI NFV ISG, "GS NFV-MAN 001 V1.1.1 Network Function Virtualisation (NFV); Management and Orchestration," Dec. 2014.

[2] A. Fischer, J. F. Botero, M. T. Beck, H. de Meer, and X. Hesselbach, "Virtual Network Embedding: A Survey," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, pp. 1888–1906, 2013.

[3] I. Houidi, W. Louati, and D. Zeghlache, "Exact Multi-Objective Virtual Network Embedding in Cloud Environments," *The Computer Journal*, vol. 58, no. 3, pp. 403–415, 2015.

[4] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[5] Z. A. Mann, "Interplay of virtual machine selection and virtual machine placement," in *Proceedings of the 5th European Conference on Service-Oriented and Cloud Computing*, 2016, pp. 137–151.

[6] ——, "Allocation of virtual machines in cloud data centers – a survey of problem models and optimization algorithms," *ACM Computing Surveys*, vol. 48, no. 1, 2015.

[7] D. M. Divakaran and M. Gurusamy, "Towards flexible guarantees in clouds: Adaptive bandwidth allocation and pricing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 6, pp. 1754–1764, 2015.

[8] E. Ahvar, S. Ahvar, N. Crespi, J. Garcia-Alfaro, and Z. Á. Mann, "NACER: a network-aware cost-efficient resource allocation method for processing-intensive tasks in distributed clouds," in *Proceedings of the 14th IEEE International Symposium on Network Computing and Applications*, 2015, pp. 90–97.

[9] M. Alicherry and T. Lakshman, "Optimizing data access latencies in cloud systems by intelligent virtual machine placement," in *Proceedings of IEEE Infocom*, 2013, pp. 647–655.

[10] E. Ahvar, S. Ahvar, Z. A. Mann, N. Crespi, J. Garcia-Alfaro, and R. Glitho, "CACEV: a cost and carbon emission-efficient virtual machine placement method for green distributed clouds," in *IEEE 13th International Conference on Services Computing*, 2016, pp. 275–282.

[11] P. Bellavista, F. Callegati, W. Cerroni, C. Contoli, A. Corradi, L. Foschini, A. Pernafini, and G. Santandrea, "Virtual network function embedding in real cloud environments," *Computer Networks*, vol. 93, pp. 506–517, dec 2015.

[12] X. Wang, C. Wu, F. Le, A. Liu, Z. Li, and F. Lau, "Online VNF scaling in datacenters," in *IEEE International Conference on Cloud Computing, CLOUD*. IEEE, jun 2017, pp. 140–147.

[13] M. Keller, C. Robbert, and H. Karl, "Template Embedding: Using Application Architecture to Allocate Resources in Distributed Clouds," in *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, 2014.

[14] J. G. Herrera and J. F. Botero, "Resource allocation in NFV: A comprehensive survey," *IEEE Transactions on Network and Service Management*, vol. 13, no. 3, pp. 518–532, 2016.

[15] B. Addis, G. Carello, F. D. Bettin, and M. Gao, "On a virtual network function placement and routing problem: properties and formulations," halshs-01643064, 2017.

[16] T. W. Kuo, B. H. Liou, K. C. J. Lin, and M. J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *IEEE INFOCOM*, 2016.

[17] A. Gupta, M. F. Habib, P. Chowdhury, M. Tornatore, and B. Mukherjee, "Joint virtual network function placement and routing of traffic in operator networks," *UC Davis, Davis, CA, USA, Tech. Rep*, 2015.

[18] S. Ahvar, H. P. Phyu, and R. Glitho, "CCVP: Cost-efficient Centrality-based VNF Placement and Chaining Algorithm for Network Service Provisioning," in *IEEE NetSoft*. IEEE, jul 2017, pp. 1–9.

[19] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte, "Orchestrating virtualized network functions," *IEEE TNSM*, vol. 13, no. 4, pp. 725–739, 2016.

[20] S. Khebbache, M. Hadji, and D. Zeghlache, "Virtualized network functions chaining and routing algorithms," *Computer Networks*, vol. 114, pp. 95–110, feb 2017.

[21] M. C. Luizelli, W. L. da Costa Cordeiro, L. S. Buriol, and L. P. Gaspary, "A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining," *Computer Communications*, vol. 102, pp. 67–77, apr 2017.

[22] J. Liu, W. Lu, F. Zhou, P. Lu, and Z. Zhu, "On dynamic service function chain deployment and readjustment," *IEEE Transactions on Network and Service Management*, vol. 14, no. 3, pp. 543–553, Sept 2017.

[23] S. Dräxler and H. Karl, "Specification, composition, and placement of network services with flexible structures," *International Journal of Network Management*, vol. 27, no. 2, pp. e1963–n/a, 2017, e1963 nem.1963.

[24] S. Mehraghdam and H. Karl, "Placement of Services with Flexible Structures Specified by a YANG Data Model," in *IEEE 2nd Conference on Network Softwarization (NetSoft)*, 2016.

[25] S. Sahhaf, W. Tavernier, D. Colle, and M. Pickavet, "Network Service Chaining with Efficient Network Function Mapping Based on Service Decompositions," in *IEEE 1st Conference on Network Softwarization (NetSoft)*, April 2015.

[26] H. Moens and F. De Turck, "VNF-P: A Model for Efficient Placement of Virtualized Network Functions," in *IEEE 10th Conference on Network and Service Management (CNSM)*, 2014.

[27] M. Savi, M. Tornatore, and G. Verticale, "Impact of Processing Costs on Service Chain Placement in Network Functions Virtualization," in *IEEE 1st Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, 2015.

[28] S. Dräxler, H. Karl, and Z. Á. Mann, "Joint optimization of scaling and placement of virtual network services," in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017)*, 2017, pp. 365–370.

[29] M. Peuster and H. Karl, "Understand Your Chains: Towards Performance Profile-based Network Service Management," in *Proceeding of the Fifth European Workshop on Software Defined Networks*. IEEE, 2016.

[30] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. E. Miller and J. W. Thatcher, Eds., 1972, pp. 85–103.

[31] D. S. Hochbaum, "The pseudoflow algorithm: A new algorithm for the maximum-flow problem," *Operations Research*, vol. 56, no. 4, pp. 992–1009, 2008.

[32] R. E. Korf, "Linear-space best-first search," *Artificial Intelligence*, vol. 62, no. 1, pp. 41–78, 1993.

[33] J. Inführ and G. R. Raidl, "Solving the virtual network mapping problem with construction heuristics, local search and variable neighborhood descent," in *Proceedings of the 13th European Conference on Evolutionary Computation in Combinatorial Optimization*, 2013, pp. 250–261.

[34] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Q. Fu, Q. Sun, C. Pham, C. Huang, J. Zhu, and P. He, "Service Function Chaining (SFC) General Use Cases," Work in progress, IETF Secretariat, Internet-Draft draft-liu-sfc-use-cases-08, Sep. 2014.

Sevil Dräxler (née Mehraghdam) received the M.Sc. degree in computer science from Paderborn University, Germany in 2014. She is currently a PhD student in Computer Networks research group in Paderborn University. Her main research interests are placement and scaling of virtualized composed services in the field of network function virtualization and software-defined networking.

Holger Karl heads the Computer Networks research group in Paderborn University. He has two main research interests; the first one is advanced wireless network, e.g., cooperative diversity techniques and resource management in factory-floor automation. His second interest is future Internet, specifically the design and architecture of protocol stacks and unifying concepts like SDN and NFV across different scenario types.

Zoltán Ádám Mann received the MSc and PhD degrees in Computer Science from Budapest University of Technology and Economics in 2001 and 2005, respectively. He is currently senior researcher at paluno – The Ruhr Institute for Software Technology, University of Duisburg-Essen. His research interests include optimization problems and algorithms in cloud computing.