This work has been accepted for publication in 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID).
 Electronic ISBN: 978-1-5090-6611-7.
S. Drãxler, H. Karl and Z. Á. Mann, "Joint Optimization of Scaling and Placement of Virtual Network Services," 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Madrid, 2017, pp. 365-370. doi: 10.1109/CCGRID.2017.25
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7973722&isnumber=7973673

Joint Optimization of Scaling and Placement of Virtual Network Services

Sevil Dräxler, Holger Karl
Paderborn University, Paderborn, Germany

Zoltán Ádám Mann
University of Duisburg-Essen, Essen, Germany

***Abstract*—Management of complex network services requires flexible and efficient service provisioning as well as optimized handling of continuous changes in the workload of the service. To adapt to changes in the demand, service components need to be replicated (*scaling*) and allocated to physical resources (*placement*) dynamically. In this paper, we propose a fully automated approach to the joint optimization problem of scaling and placement, 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.**

I. Introduction

Large-scale cloud and data center networks are typically hosting several network services (e.g., video streaming) composed of different (virtualized) components, serving a continuously changing demand. For managing these services, fast, flexible, and automatic deployment and scaling mechanisms are required, which has led to concepts like network function virtualization [1].

Such technologies provide the basic mechanisms to flexibly adapt to changing demands, such as the addition and removal of services or fluctuations in the resource demand of a service. In particular, (i) services can be scaled by adding or removing instances of service components, (ii) the placement and resource allocation of service components can be modified, and (iii) network flows between the service components can be re-routed through other network paths.

Having so many degrees of freedom also means a huge search space so that finding the best adaptation requires a complex strategy. Consider a provider hosting a dynamically changing set of services, where each service serves a dynamically changing set of request and data sources, each with dynamically changing data rates. Trade-offs between the conflicting goals can be highly non-trivial, for example:

- Placing a data processing component on a node with limited resources near the source, thus minimizing latency, versus placing it on a more powerful node further away in the network, thus minimizing processing time.
- Letting a single instance of a processing component serve multiple sources, thus minimizing compute resource consumption, versus using dedicated instances near the sources, thus minimizing network load.
- Changing the current configuration to a better one that will hopefully pay off in the long run, versus keeping the current configuration, thus avoiding reconfiguration costs.
- Fulfilling the resource requirements of one service versus the requirements of another service.

Given the complexity of this optimization problem and the pace with which changes in the demand may occur, automation is indispensable. Today, however, only very limited computer support is available for some partial aspects (e.g., scaling without placement, or placement without scaling).

We argue that a more comprehensive approach is necessary. In our proposed solution, each service is described by a *service template*, containing information about the components of the 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*. These functions can be specified by the service developers or determined using service profiling methods [2]. Service developers can focus their attention on building services from components, without having to worry about the instantiation and placement of the components.

Text

Our optimization approach takes care of the rest: based on the location and current data rate of the data sources, the templates are scaled by replicating service components as necessary, the placement of components on physical nodes is determined, and data flows are routed along network paths. Node and link capacity constraints are automatically taken into account, and the solution is optimized along multiple objectives, including minimization of resource usage, minimization of latency, and minimization of deployment adaptation costs.

Our main contributions can be summarized as follows:

- Formalizing the *template embedding* process as a joint optimization problem for scaling and placing service templates in the network, where demands of service components are determined as a function of the incoming data rate to each instance.
- Proving the NP-hardness of the problem.
- Presenting two algorithms for solving the template embedding problem, one based on mixed integer programming, the other a custom heuristic.
- Detailed evaluation of both algorithms to determine their strengths and weaknesses.

With the proposed approach, service developers obtain a flexible way to define services on a high level of abstraction while providers obtain powerful methods to optimize the

scaling and placement of multiple services in a single step, fully automatically.

II. Previous work

Template embedding has similar properties to virtual network embedding. Both problems deal with mapping virtual nodes and virtual links of a graph into another graph. Fischer et al. [3] have published a survey of different approaches to this problem, including static and dynamic embedding algorithms.

In contrast to static virtual network embedding solutions, in this paper we also deal with optimizing and modifying already embedded templates, in addition to the initial mapping process. Our approach is different from dynamic solutions as well. For example, Houidi et al. [4] present a mixed integer program that can modify the mapping in reaction to node or link failures. The modifications, 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 relocating the embedded nodes and links, our approach can also determine and modify the *structure* in the template (the graph to be embedded) by adding/removing nodes and links if necessary.

This problem has recently gained importance also in the field of Network Function Virtualization (NFV), where services that are composed of multiple virtual network functions are mapped into the network. Several optimization approaches [5], [6], [7], [8] and heuristic algorithms [9], [10] have been proposed for this problem. Our template embedding approach has two important differences to these solutions: First, our approach can be used for initial placement as well as scaling and adapting existing placements. Second, in our approach, both the structure of the service and its mapping to the network are determined in one single step, based on the requirements of the service and current state of network resources, thus aiming for a global optimum.

Keller et al. [11] formulate the template embedding problem similar to our approach. Our assumptions and terminology are partly based on their work, but there are important differences that make our approach stronger and more flexible than the solution presented in that paper. Their templates describe strict *scaling restrictions* for each component; e.g., in a two-tier application, the front-end server needs exactly m instances of the back-end server. They consider the sources of traffic to be users distributed over the network. The number of users then determines the number of instances required for each component, based on the scaling restrictions. We drop the scaling restrictions as well as the discrete modeling of the users. Instead, we determine the number of required instances for each template component based on the *data rate* (e.g., requests or bits per second) coming from different source components on different locations in the network, which allows a more fine-grained control of the scaling process.

Moreover, Keller et al. assign pre-defined resource demands to components and express the compute capacity of network nodes as the maximal number of instances they can host. We model the CPU and memory demands of each instance of a component during the template embedding process as a function of the data rate it actually handles. Substrate network nodes in our model can host instances of different components as long as the capacity is not exceeded, thus allowing more flexibility in the placement. Finally, the optimization objective in their model is to minimize the total number of instances for embedded templates. 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. For these reasons, we believe that our problem formulation provides a more realistic model for optimizing virtual network services.

Another related area is the allocation of virtual machines to physical machines in cloud data centers. Scaling and placing instances while obeying capacity constraints are also typical features of such problem formulations [12]; however, communication among virtual machines is typically not taken into account or considered only in a rudimentary way [13]. Although some of those approaches account for the communication among virtual machines through the network [14], [15], [16], they do not include routing decisions. 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 existing approaches.

We do not impose any limitation on the type of components used. Therefore, our solution is applicable in different contexts, e.g., NFV, (distributed) cloud computing, and data center network management.

TABLE I
Notations Used for Graphs in the Model

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

III. Problem model

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

Informally, the problem we are addressing is as follows: given a set of services with their templates and sources, we want to optimally embed the services into the network.

A. Substrate Network

We model the *substrate network* as a directed graph $G_{\text{sub}}=(V, L)$. Each *node* $v \in V$ is associated with a CPU

capacity $\text{cap}_{\text{cpu}}(v)$ and a memory capacity $\text{cap}_{\text{mem}}(v)$[1]. Moreover, we assume every node has routing capabilities and can forward traffic to its neighboring nodes.[2] Each *link* $l \in L$ is associated with a maximum data rate $b(l)$ and a delay $d(l)$. For each node v, we assume the internal communications (e.g., the communications inside a data center) can be done with unlimited data rate and negligible delay.

B. Templates

The substrate network hosts a set $\mathcal{T}$ of network services. We define the structure of each 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 service and specify the way they should be connected to each other to deliver the desired service. Fig. 1(a) shows an example template.

A template component $j \in C_T$ has a set $\text{In}(j)$ of inputs and a set $\text{Out}(j)$ of outputs. Its resource consumption depends on data rates of the flows entering the component. We characterize this using a pair of functions $p_j, m_j : \mathbb{R}_{\geq 0}^{|\text{In}(j)|} \rightarrow \mathbb{R}_{\geq 0}$, where p_j is the CPU load and m_j is the required memory size of component j. These functions should typically account for the data rate of the flows entering the component as well as a fixed consumption value at idle times. Data rates of the outputs are determined as a function of data rates of the inputs specified as $r_j : \mathbb{R}_{\geq 0}^{|\text{In}(j)|} \rightarrow \mathbb{R}_{\geq 0}^{|\text{Out}(j)|}$. Fig. 1(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.

C. Overlays and sources

A template specifies the types of components and the connections among them as well as their resource demands depending on the load. A specific, deployable instantiation of a service can be derived by scaling its 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 service component might be required. To model this, for each 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 a data rate. Such a tuple means that an instance of source component j generates a data or request flow from v with rate λ. Sources represent populations of users, sensors, or any other component that can generate flows to be processed by the corresponding service.

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 corresponding template. To be able to make the required number of instances for each component, we assume the components are stateless or a state management system is in place to handle the state upon adding or removing instances. 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.

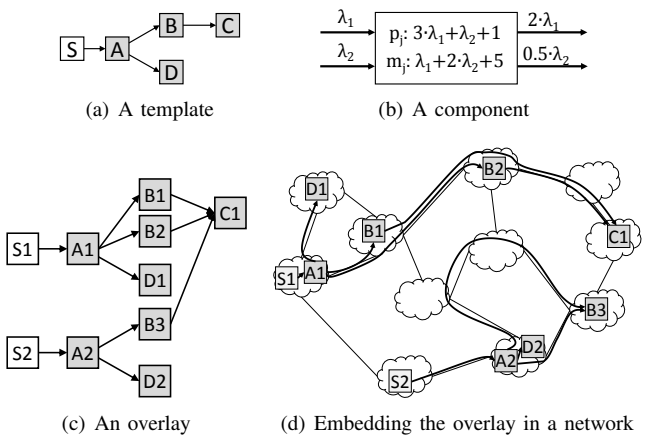


Fig. 1. Some examples: (a) a template, (b) a component, (c) an overlay corresponding to the template, and (d) a mapping of the overlay into a substrate network. The links of the substrate network are bi-directional.

Fig. 1(c) shows an example overlay corresponding to the template in Fig. 1(a). An overlay might include multiple instances of a specific template component: e.g., B1, B2, and B3 all correspond to 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 $G_{\text{OL}}(T)$ must be mapped to the substrate network by a feasible mapping P_T. We define the mapping as a pair of functions: $P_T = \left(P_T^{(I)}, P_T^{(E)}\right)$.

$P_T^{(I)} : I_{\text{OL}} \rightarrow V$ 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

[1]This can be easily extended to other types of resource.

[2]Capacities can be 0, e.g., to represent conventional switches.

instances by a single instance and thus save the idle resource consumption of one instance.

$P_T^{(E)} : E_{\text{OL}} \to \mathcal{F}$ 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 $e \in E_{\text{OL}}$ 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. 1(d) shows a possible mapping of the overlay of Fig. 1(c) to an example substrate network, based on the predefined allocation 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.

E. Objectives

The *system state* consists of the overlays and their mapping on the substrate network, which can be changed by our template embedding algorithm.

A valid system state must respect all capacity constraints: for each node v, the total resource needs of the instances mapped to v must be within its capacity (for both CPU and memory), and for each link l, the sum of the flows 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 state: for example, a valid system state (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 state σ, our primary objective is to find a new state σ', in which the number of constraint violations is minimal (ideally, zero). For this, we assume violating node and link capacity constraints are equally undesired.

There are a number of further, secondary objectives, which can be used as tie-breaker to choose from system states 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 of amounts of capacity constraint violations, for each resource type (CPU, memory, bandwidth)
- Total resource consumption of all instances across all overlays, for each resource type (CPU, memory, bandwidth)

Higher values for these metrics result in higher costs for the system or in lower customer satisfaction, so our objective is to minimize these values. Therefore, our aim is to select a new state σ' from the set of states with minimal number of constraint violations that is Pareto-optimal with respect to these secondary metrics.

IV. Complexity

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

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 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, 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.

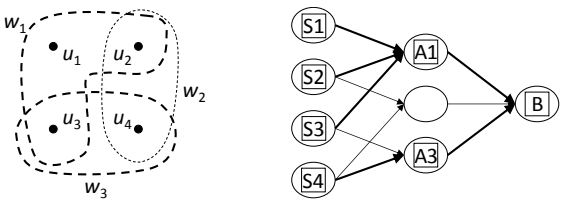
From this instance of Set Covering, an instance of the Template Embedding problem is created as follows. The substrate network consists of nodes $V = \{s_1, \ldots, s_{|U|}\} \cup \{a_1, \ldots, a_{|\mathcal{W}|}\} \cup \{b\}$, where each s_i represents an element of U and each element a_j represents 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: $\text{cap}_{\text{cpu}}(s_i) = \text{cap}_{\text{mem}}(s_i) = 0$ for each $i \in [1, |U|]$, $\text{cap}_{\text{cpu}}(a_j) = 0$ and $\text{cap}_{\text{mem}}(a_j) = 1$ for each $j \in [1, |\mathcal{W}|]$, and $\text{cap}_{\text{cpu}}(b) = 1$ and $\text{cap}_{\text{mem}}(b) = 0$. 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

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

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



(a) An instance of Set Covering ($k = 2$) and a solution (thick lines)

(b) The generated instance of Template Embedding and the corresponding solution

Fig. 2. An example for the proof of Theorem 1

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. 2 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 0 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.

TABLE II
VARIABLES

Name	Domain	Definition
$x_{j,v}$	$\{0,1\}$	1 iff an instance of component $j \in \mathcal{C}$ 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
$\Lambda_{j,v}$	$\mathbb{R}_{\geq 0}^{\lvert\text{In}(j)\rvert}$	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
$\Lambda'_{j,v}$	$\mathbb{R}_{\geq 0}^{\lvert\text{Out}(j)\rvert}$	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-allocation over all nodes
ψ_{mem}	$\mathbb{R}_{\geq 0}$	Maximum memory over-allocation over all nodes
ψ_{dr}	$\mathbb{R}_{\geq 0}$	Maximum capacity over-allocation 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 $x_{j,v} \neq x^*_{j,v}$

V. 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 an appropriate solver can be used to solve this formulation, 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 that the variables must fulfill. $\mathcal{C} = \bigcup_{T \in \mathcal{T}} C_T$ 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 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.

The *problem inputs* are the substrate network, the set of service templates, and the set of sources, as described in Section III. Additionally, information about existing instances should also be taken into account during the decision process.

For this, we define $x^*_{j,v}(\forall j \in \mathcal{C}, v \in V)$ as a constant given as part of the problem input. If there is a previously mapped instance of j on node v in the network, $x^*_{j,v}$ is 1, otherwise it is 0.

A. Constraints

Here we define the sets of constraints that enforce the required rules to optimize the template embedding process.

Mapping consistency rules:

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

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

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

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

$$\forall j \in \mathcal{C}, \forall v \in V: \quad x_{j,v} - x^*_{j,v} \leq \delta_{j,v} \tag{5}$$

$$\forall j \in \mathcal{C}, \forall v \in V: \quad x^*_{j,v} - x_{j,v} \leq \delta_{j,v} \tag{6}$$

Flow and data rate rules:

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

$$\forall j \in \mathcal{C}, \forall v \in V, k \in [1, |\text{In}(j)|]: \quad (\Lambda_{j,v})_k = \sum_{a \text{ ends in input } k \text{ of } j, v' \in V} y_{a,v',v} \tag{8}$$

$$\forall j \in \mathcal{C}, \forall v \in V, k \in [1, |\text{Out}(j)|]: \quad (\Lambda'_{j,v})_k = \sum_{a \text{ starts in output } k \text{ of } j, v' \in V} y_{a,v,v'} \tag{9}$$

$$\forall a \in \mathcal{A}, \forall v, v_1, v_2 \in V: \quad \sum_{vv' \in L} z_{a,v_1,v_2,vv'} - \sum_{v'v \in L} z_{a,v_1,v_2,v'v} = \begin{cases} 0 & \text{if } v \neq v_1 \text{ and } v \neq v_2 \\ y_{a,v_1,v_2} & \text{if } v = v_1 \text{ and } v_1 \neq v_2 \\ 0 & \text{if } v = v_1 = v_2 \end{cases} \tag{10}$$

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

Calculation of resource consumption:

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

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

Capacity constraints:

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

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

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

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

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

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

To illustrate the interplay of these constraints, we assume we need to optimize the embedding shown in Fig. 1(d). 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 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-allocation of node and link capacities are indicated correctly, and collect the maximum value of over-allocation for each resource type. This maximum value is used in the objective function described in Section V-B, which drives the decisions based on the constraints.

B. Optimization objective

We formalize the optimization objective based on the goals defined in Section III-E:

$$\begin{aligned} \text{minimize} \quad & M_1 \cdot \Big(\sum_{v \in V} (\omega_{v,\text{cpu}} + \omega_{v,\text{mem}}) + \sum_{l \in L} \omega_l \Big) \\ & + M_2 \cdot \Big(\sum_{\substack{a \in \mathcal{A} \\ v,v' \in V \\ l \in L}} (d(l) \cdot \zeta_{a,v,v',l}) + \sum_{\substack{j \in \mathcal{C} \\ v \in V}} \delta_{j,v} \Big) \\ & + \psi_{\text{cpu}} + \psi_{\text{mem}} + \psi_{\text{dr}} + \sum_{\substack{j \in \mathcal{C} \\ v \in V}} (\varrho_{j,v} + \mu_{j,v}) + \sum_{\substack{a \in \mathcal{A} \\ v,v' \in V \\ l \in L}} z_{a,v,v',l} \end{aligned} \tag{20}$$

By assigning sufficiently large values to constants M_1 and M_2, we can achieve the following goals with the given priorities (1 being the highest priority):

1) Number of capacity constraint violations over all nodes and links is minimized.
2) Template arcs are mapped to network paths with minimum latency. Moreover, 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 III-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 optimizing an existing embedding. However, when used for initial embedding, the term $\sum_{j\in\mathcal{C},v\in V}\delta_{j,v}$ should be removed from the objective function to ensure that the decision is not biased towards an embedding with fewer instances, in case 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 $j \in \mathcal{C}$, 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.

VI. 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 works for non-linear functions p_j, m_j, and r_j as well.

The heuristic, shown in Algorithm 1, 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 removes as many edges as possible (lines 3–6) and when this is not possible anymore, 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 increase the existing flows (lines 17–19), and if this is not sufficient, creates further instances and flows (lines 20–23).

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,Φ,$\lambda' - \lambda$)

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 and the one leading 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 time complexity of the best known algorithms for this purpose are worse than quadratic with respect to the size of the graph [17]; 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 is used, which runs in linear time. It should 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 CREATEINSTANCEANDFLOW 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

Algorithm 2 Auxiliary methods of the heuristic

```
1: /* Decrease the flows on the edges in E by Δλ in total */
2: procedure DECREASE(E,Δλ)
3:     sort E in non-decreasing order of flow data rate
4:     for all e ∈ E while flow data rate λ(e) ≤ Δλ do
5:         Δλ := Δλ − λ(e)
6:         remove e
7:     if Δλ > 0 then
8:         let e be the next edge
9:         reduce flow of e by a factor of (λ(e) − Δλ)/λ(e)
10: /* Increase the flows in Φ leaving output k of instance i by Δλ
    in total */
11: procedure INCREASE(i,k,Φ,Δλ)
12:     for all arc (c(i), j) leaving output k of c(i) do
13:         if ∄i′ ∈ I_OL with c(i′) = j and ii′ ∈ E_OL then
14:             φ := CREATEINSTANCEANDFLOW(j, i, Δλ)
15:             Δλ := Δλ − (data rate of φ)
16:             Φ := Φ ∪ {φ}
17:     for all φ ∈ Φ do
18:         d := INCRFLOW(φ,Δλ)
19:         Δλ := Δλ − d
20:     while Δλ > 0 do
21:         (c(i), j): random arc leaving output k of c(i)
22:         φ := CREATEINSTANCEANDFLOW(j, i, Δλ)
23:         Δλ := Δλ − (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 ∈ V do
27:         create temporary instance i′ of j on v
28:         φ: flow of data rate 0 from i to i′
29:         INCRFLOW(φ,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:     β1 := maximum flow based on cap_CPU(v′)
37:     β2 := maximum flow based on cap_mem(v′)
38:     d := min(d, β1, β2)
39:     P: v ⇝ v′ path of high bandwidth (b) and low latency
40:     increase φ by min(b, d) along P
```

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 above comparison relation.

VII. Evaluation

We implemented the presented algorithms in the form of a C++ program. For solving the MILP, Gurobi Optimizer 7.0.1[3]

[3]http://www.gurobi.com/

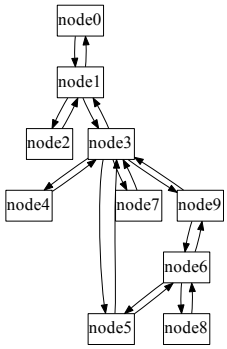


Fig. 3. Example substrate network

was used. For substrate networks, we used benchmarks for the Virtual Network Mapping Problem[4] from Inführ and Raidl [18]. As service templates, we use examples from IETF's Service Function Chaining Use Cases [19].

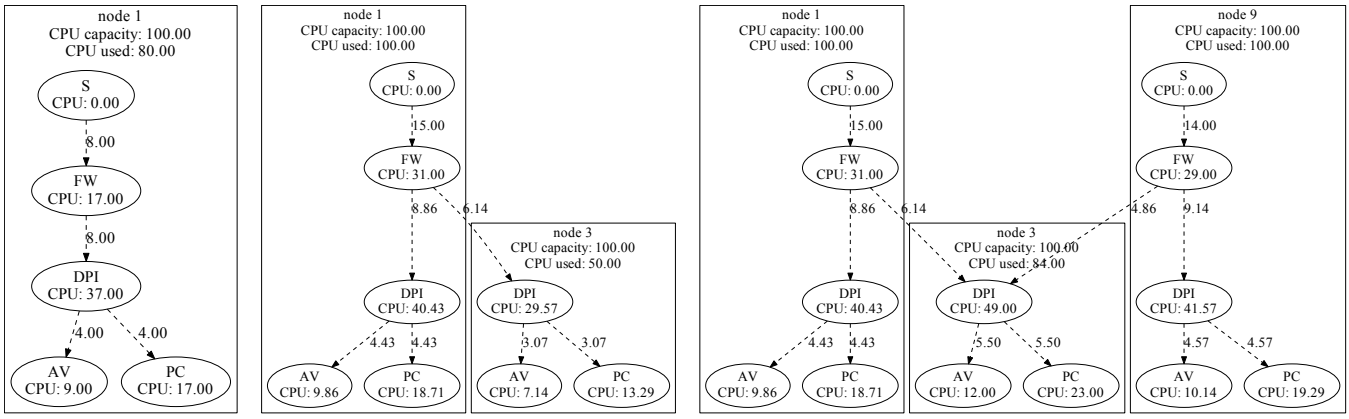
First, we illustrate our approach on a small substrate network of 10 nodes and 20 arcs (see Fig. 3) in which the CPU and memory capacity of each node is taken to be 100. In this network, a service consisting of a source (S), a firewall (FW), a deep packet inspection (DPI), an anti-virus (AV), and a parental 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. 4(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. 4(b)). 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, and 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. 4(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.

We consider a substrate network with 20 nodes and 44 arcs, in which multiple services are deployed. Each service is a virtual content delivery network for video streaming, consisting of a streaming server, a DPI, a video optimizer, and a cache. The number of concurrently active services varies from 0 to 4, the number of sources varies from 0 to 20. Fig. 5 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

[4]https://www.ac.tuwien.ac.at/files/resources/instances/vnmp



(a) Initial embedding (b) Result of increased source data rate (c) Result of the emergence of a second source

Fig. 4. Illustrative example (memory values not shown for better readability)

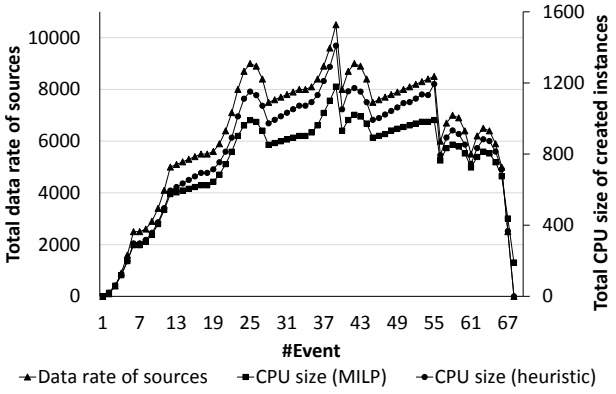


Fig. 5. Temporal development of the demand and the allocated capacity in a complex scenario

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 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. This is because in the MILP algorithm, the optimal location for all required instances can be determined at the same time respective to the location of the sources, resulting 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 through 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. The corresponding plots have been omitted because of space constraints.

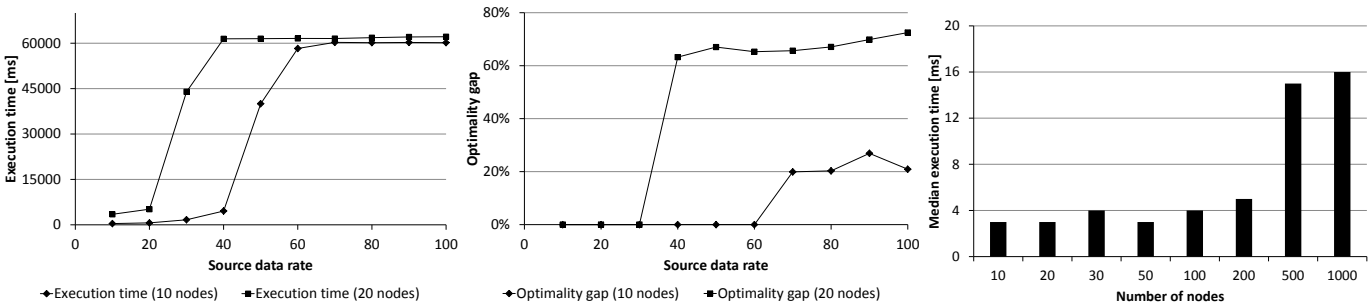
Since the template embedding problem is NP-hard, it is foreseeable that the scalability of the MILP solver will be limited. In order to test this, we gradually increase the source data rate of the service from our first experiment, leading to an increasing number of instances; moreover, we also consider 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.

Fig. 6(a) shows the execution time of the MILP algorithm for different data rates and substrate network sizes, while Fig. 6(b) shows the corresponding gap between the found solution and lower bound. As can be seen, for a small network with 10 nodes and 20 arcs, 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 arcs, the timeout is reached 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, as shown in Fig. 6(c), the execution time of the heuristic algorithm remains very low even for the largest substrate networks: for 1000 nodes and 2530 arcs, the execution time is still below 20 milliseconds, rendering the heuristic practical for industrial problem sizes as well.

VIII. Conclusions and Future Work

We have presented a fully automatic approach to scale and place multiple virtual network services on a common substrate network. Besides formally defining the problem and proving its NP-hardness, we developed two algorithms for it, an MILP-



(a) Execution time of the MILP algorithm (b) Optimality gap of the MILP algorithm (c) Execution time of the heuristic algorithm

Fig. 6. Scalability of the presented algorithms

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. The MILP-based algorithm gives optimal or near-optimal results for relatively small networks, whereas the heuristic remains very fast for even the largest networks that were tested. Overall, the tests gave evidence to the feasibility 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 reoptimize the system state after changes.

Promising future research directions include, beside further algorithmic enhancements to the presented algorithms, the consideration of queuing incoming requests in the service components and the investigation of the effects of cyclic service templates.

ACKNOWLEDGMENT

This work has been performed in the context of the SONATA project, funded by the European Commission under Grant number 671517 through the Horizon 2020 and 5G-PPP programs. This work is partially supported by the German Research Foundation (DFG) within the Collaborative Research Center On-The-Fly Computing (SFB 901) and the International Graduate School "Dynamic Intelligent Systems".

The work of Z. Á. Mann was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947) and the European Union's 7th Framework Programme (FP7/2007-2013) under grant agreement 610802 (CloudWave).

REFERENCES

[1] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. D. Turck, and R. Boutaba, "Network function virtualization: State-of-the-art and research challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[2] 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.

[3] 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.

[4] 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.

[5] S. Mehraghdam, M. Keller, and H. Karl, "Specifying and Placing Chains of Virtual Network Functions," in *IEEE 3rd International Conference on Cloud Networking (CloudNet)*, 2014.

[6] 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.

[7] 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.

[8] 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.

[9] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and Evaluation of Algorithms for Mapping and Scheduling of Virtual Network Functions," in *IEEE 1st Conference on Network Softwarization (NetSoft)*, 2015.

[10] M. T. Beck and J. F. Botero, "Coordinated Allocation of Service Function Chains," in *IEEE Global Communications Conference*, 2015.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.