2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing

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*—The management of complex network services requires flexible and efficient service provisioning as well as optimized handling of continuous changes in the workload of the services. 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. 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; in particular, (i) services can be scaled by adding or removing instances of service components, (ii) resource allocation of service components can be modified, and (iii) network flows between the service components can be re-routed.

Having so many degrees of freedom also means an enormous search space so that finding the best adaptation requires a complex strategy. Moreover, 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* (e.g., a service component generating data flows or service users generating requests), thus minimizing latency, versus placing it on a more powerful node further away in the network, thus minimizing processing time.

Given the complexity of this optimization problem and the pace of the changes in demand, automation is indispensable. Existing solutions typically focus on a partial solution, 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 among them, and their resource demands. Both the resource demands 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 on building services from components, without having to worry about the instantiation and placement of the components.

Our optimization approach takes care of the rest: based on the location and current data rate of the 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 taken into account and the solution is optimized along multiple objectives, including minimization of resource usage, latency, and deployment adaptation costs.

In this paper, we formalize the *template embedding* process as a joint optimization problem for scaling and placing service templates, where demands of service components are determined as a function of the incoming data rate to each instance. We present two algorithms for solving the template embedding problem, one based on mixed integer programming, the other a custom heuristic, and evaluate 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

Similar to the virtual network embedding (VNE) problem [3], template embedding deals with mapping virtual nodes and virtual links of a graph into another graph. Unlike static VNE solutions, in this paper we also deal with optimizing already embedded templates, besides the initial embedding. In addition to relocating the embedded nodes and links, presented in dynamic VNE solutions, our approach can also determine and modify the *structure* of the graph to be embedded by adding/removing nodes and links.

This problem has recently gained importance also in the field of Network Function Virtualization (NFV), where services composed of multiple virtual network functions are mapped into the network. Several solutions [4], [5], [6] have been proposed for this problem. In contrast to existing solutions, our approach can be used for initial placement as well as adapting existing placements. Moreover, our approach determines both the structure of the service and its mapping to the network in one single step, aiming for a global optimum.

Keller et al. [7] formulate the template embedding problem similar to our approach. Our assumptions and terminology are

978-1-5090-6611-7/17 $31.00 © 2017 IEEE
DOI 10.1109/CCGRID.2017.25

partly based on their work, but there are important differences that make our approach stronger and more flexible than their solution. Their templates describe strict *scaling restrictions* on components; e.g., the front-end server of an application needs exactly m instances of the back-end server. The number of application users then determines the number of instances required for each component, based on the scaling restrictions. We determine the number of required instances for each component based on the *data rate* (e.g., requests or bits per second) from different sources on different network nodes, without scaling restriction, providing a more fine-grained control. Moreover, they assign pre-defined resource demands to components. We determine the demands during the template embedding process as a function of the input data rate. Finally, we use a more sophisticated multi-objective optimization approach where different metrics are considered, providing 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 within capacity constraints are also typical features of such problem formulations [8]; however, communication among virtual machines is typically not taken into account or considered only in a rudimentary way [9], [10], [11], without including routing decisions. Moreover, our approach of specifying resource consumption as a function of input data rates allows a much more realistic demand modeling 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.

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 (*template*), a concrete and deployable instantiation of the service (*overlay*), and the actual network (*substrate network*). We use different names and notations to distinguish among these graphs.

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 CPU and memory capacities, $\text{cap}_{\text{cpu}}(v)$ and $\text{cap}_{\text{mem}}(v)$[1]. We assume every node has forwarding 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 its internal communications can be done with unlimited data rate and negligible delay.

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

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

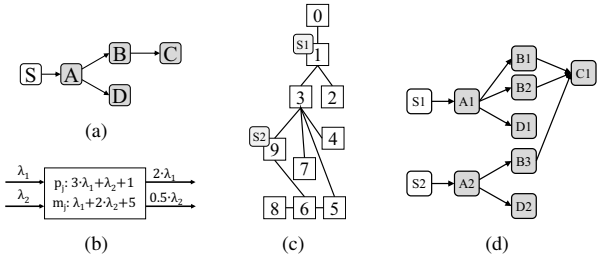


Fig. 1. Some examples: (a) a template, (b) a component, (c) a substrate network with bi-directional links, including two sources for the template, and (d) a corresponding overlay

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 types 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 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. Sources and Overlays

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 may represent populations of users, sensors, or any other component that can generate network flows.

An *overlay* is the outcome of scaling the template based on the location and data rate of its 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 template. To be able to create 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) 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(d) 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: $P_T = \left(P_T^{(I)}, P_T^{(E)}\right)$.

$P_T^{(I)} : I_{\text{OL}} \to V$ maps each instance in the overlay to a node in the substrate network. We assume that two instances of the same component cannot be mapped to the same node, as 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.

$P_T^{(E)} : E_{\text{OL}} \to \mathcal{F}$ 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.

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 an instance of 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 demands of the instances.

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, data rate)
- Total resource consumption of all instances across all overlays, for each resource type (CPU, memory, data rate)

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.

F. Complexity

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

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 have used a reduction from the Set Covering problem to the Template Embedding problem. We omit the details of this proof because of space limitations.

Due to the complexity of the problem, 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.

IV. Solution

We solve the template embedding problem using two approaches: using mixed integer programming and a heuristic algorithm. In this section, we give an overview of these approaches.

TABLE I

Name	Domain	Definition
$x_{j,v}$	$\{0,1\}$	1 iff an instance of $j \in \mathcal{C}$ 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 data rate of the 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 data rate of the 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 $j \in C_T$ on node $v \in V$, or an all-zero vector if no such instance exists 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 $j \in C_T$ on node $v \in V$, or an all-zero vector if no such instance exists on v
$\varrho_{j,v}$	$\mathbb{R}_{\geq 0}$	CPU requirement of the instance of $j \in C_T$ 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 exists 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 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. Mixed Integer Programming Approach

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, explained in more detail in Table I.

$\mathcal{C} = \bigcup_{T \in \mathcal{T}} C_T$ is 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. 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. Additionally, information about existing instances should be taken into account: $x^*_{j,v} (\forall j \in \mathcal{C}, v \in V)$ is 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.

Using the following sets of constraints, we enforce the required rules to optimize the template embedding process.

a) Mapping consistency rules:

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

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

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

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

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

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

b) Flow and data rate rules:

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

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

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

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

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

c) Calculation of resource consumption:

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

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

d) Capacity constraints:

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

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

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

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

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

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

e) Objective function:

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

This can be used for initial embedding of service templates as well as optimizing an existing embedding. In the first case, 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 embeddings with fewer instances.

If all functions p_j, m_j, and r_j are linear, then we obtain a mixed-integer linear program (MILP), which can be solved by appropriate solvers.

Algorithm 1 Main procedure of the heuristic algorithm

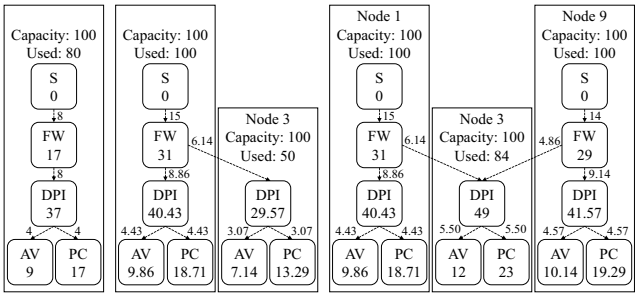
1: **if** $\exists G_{\text{OL}}(T)$ with $T \notin \mathcal{T}$ **then**
2: remove $G_{\text{OL}}(T)$
3: **for all** $T \in \mathcal{T}$ **do**
4: **if** $\nexists G_{\text{OL}}(T)$ **then**
5: create empty overlay $G_{\text{OL}}(T)$
6: **for all** $(v, j, \lambda) \in S(T)$ **do**
7: **if** $\nexists i \in I_{\text{OL}}$ 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,$\lambda' - \lambda$)

B. Heuristic Approach

The heuristic shown in Algorithm 1 is not guaranteed to find an optimal solution but is much faster than the mixed integer programming approach. It also has the advantage that it works for non-linear functions p_j, m_j, and r_j, as well.

The algorithm starts by checking that each service has a corresponding overlay and each overlay corresponds to a service (lines 1–5). If templates have arrived or left since the last invocation of the algorithm, the corresponding overlay is created or removed at this point. Next, the mapping of the sources is checked (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, the output data rate of the corresponding source component instance is updated; if a source disappeared, the corresponding source component instance is removed. Finally, to propagate the changes of the sources to the processing instances, we iterate over all instances and ensure that the new output data rates, determined by the new input data rates, are discharged correctly by outgoing flows (lines 12–24). For this, it is important to consider the instances in 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.

DECREASE removes as many edges as possible and when this is not possible anymore, it reduces the next flow on each link by the same factor to achieve the required reduction. INCREASE first checks if new instances need to be created to be consistent with the template, then tries to increase the existing flows, and if this is not sufficient, creates further instances and flows.



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

Fig. 2. Embedding of a template into an example substrate network, showing CPU values (memory values omitted for readability) and overlay data rates

V. Evaluation

First, we illustrate our approach on a small substrate network of 10 nodes and 20 arcs (Fig. 1(c)) in which the CPU and memory capacity of each node is 100. We consider a template consisting of a source (S), a firewall (FW), a deep packet inspection (DPI), an anti-virus (AV), and a parental control (PC) component. Initially, there is a single source in node 1 with a moderate data rate. As a result, our MILP-based algorithm[3] deploys all components on node 1 (Fig. 2(a)).

Subsequently, 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 places the newly created instances on node 3 (Fig. 2(b)). Later on, a second source emerges for the same service on node 9. The algorithm decides to create new 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 is routed to the existing DPI, AV, and PC instances on node 3 because node 3 still has sufficient free capacity (Fig. 2(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 more complex scenarios.

We consider a substrate network with 20 nodes and 44 arcs[4], in which multiple templates are embedded[5]. Each template corresponds to 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 embedded templates varies from 0 to 4, the number of sources varies from 0 to 20. Fig. 3 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

[3]Solved using Gurobi Optimizer 7.0.1 (http://www.gurobi.com)

[4]Network instances from benchmarks for the Virtual Network Mapping Problem (https://www.ac.tuwien.ac.at/files/resources/instances/vnmp)

[5]Templates based on examples from IETF's Service Function Chaining Use Cases in Mobile Networks (draft-ietf-sfc-use-case-mobility-07)

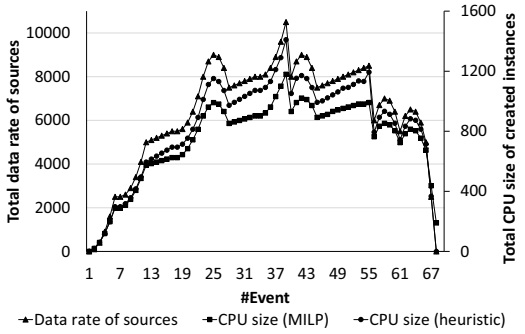


Fig. 3. Temporal development of the demand and the allocated capacity in a complex scenario

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.

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, based on 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, scalability of the MILP approach is limited. By increasing the data rates of sources on this substrate network, the timeout of 60 seconds we had set for our simulations is reached, giving solutions with unacceptable optimality gaps. For bigger substrate networks, the performance of the algorithm further deteriorates, up to the point where it cannot be run anymore because of memory problems. In contrast, the execution time of the heuristic algorithm remains very low even for very large substrate networks. For example, with 1000 nodes and 2530 arcs, the execution time is still below 20 milliseconds, suiting industrial problem sizes as well.

VI. Conclusion

We have presented a fully automatic approach to scale and place multiple virtual network services on a common substrate network. Besides formally defining this NP-hard problem, 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. 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. To show the applicability of our solutions in NFV context, we are working on integrating our solution approaches into SONATA's open-source orchestrator [12]. Our algorithms can be added to the system as service-specific management plugins, reading the required information from the virtual network function descriptors used in SONATA.

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 Horizon 2020 research and innovation programme under grant 731678 (RestAssured).

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] "SONATA project," http://sonata-nfv.eu, date accessed: 2017-01-30.