This work has been accepted for publication in 2018 IEEE Conference on Network Softwarization (Netsoft 2018).
 ISBN: 978-1-5386-4633-5

# Scaling and Placing Bidirectional Services with Stateful Virtual and Physical Network Functions

Sevil Dräxler, Stefan Schneider, Holger Karl
Paderborn University, Paderborn, Germany
{sevil.draexler, stefan.schneider, holger.karl}@uni-paderborn.de

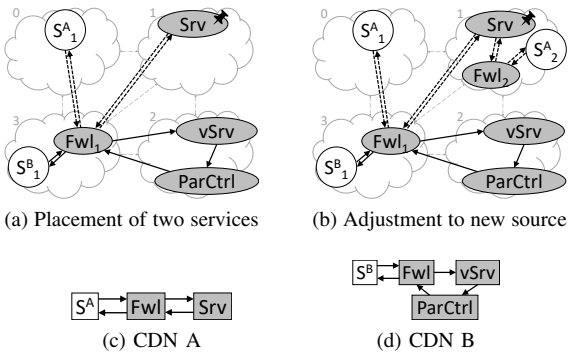
***Abstract*—Network function virtualization requires scaling and placement, deciding the number and the location of function instances. Current approaches are limited in flexibility and practical applicability. Specifically, we study dynamic, single-step, joint scaling and placement of network services with bidirectional flows traversing Physical or Virtual Network Functions (VNFs) and returning to their sources. We develop models to support stateful components and legacy network functions with fixed locations in these network services as well as the possibility of reusing VNFs across network services. We formalize the problem of jointly scaling and placing such network services as a mixed-integer linear program (MILP). We show that this problem is NP-complete and also present a heuristic algorithm to find good solutions in short time. In an extensive evaluation with realistic scenarios, we investigate the capabilities of the two approaches.**

## I. Introduction

Complex network services running on top of softwarized networks, consist of multiple Virtual Network Functions (VNFs), legacy Physical Network Functions (PNFs), and application components. These services can be bidirectional, e.g., processing requests and sending back responses to the users. The orchestration of such network services is an ongoing challenge in network function virtualization (NFV).

Fig. 1a shows an example orchestration scenario where two Content Delivery Networks (CDN), realized by network services A and B, have been deployed. Each network service has its own user group, represented by *sources* $S^A_1$ and $S^B_1$. In network service A, user requests go through a stateful firewall towards a content delivery server, deployed as a PNF in node 1; content returns to the users through the same firewall. Network service B is a virtualized version of this CDN and, additionally, requires the network flows to go through a parental control function before returning to the users through the firewall. In this example, all servers expect similar functionality from the firewall. Therefore, the requests can be mapped to the same instance of the stateful firewall to reduce deployment costs, requiring sufficient resources at the firewall.

If service A decides to expand its coverage to additional users in a different geographical location (Fig. 1b), a new instance of the firewall might need to be instantiated in a suitable location. The placement and scaling for both services need to be recalculated (taking the existing deployments into account) to find the optimal number of instances required for the functions and their optimal location (where "optimal" can be defined per-scenario).




(a) Placement of two services (b) Adjustment to new source

(c) CDN A (d) CDN B

Fig. 1. (a) Embedding of two CDN network services sharing a stateful firewall. (b) As a new source appears, the embedding is adjusted. (c) and (d) show the structure of the network services.

In such cases, deciding placement without considering scaling possibilities or scaling the service without considering placement constraints can leave the network and the services in suboptimal states. To prevent this, we focus on *jointly* optimizing placement and scaling. As in a related problem formulation from our previous work [1], we use the term *service template* for flexible descriptors that specify the *structure* of a network service. E.g., Fig. 1c and 1d show the service structures of the previous example. Service templates include type of the involved components (VNF or PNF), their inter-connections, and the resource requirements of each component relative to the load it needs to handle. Service templates are scaled to the required number of instances for each VNF and the resources required for each instance, jointly with placing instances into the network. The main driving factor of this process is the location of *sources* and the load originating from them for each service. Sources represent, e.g., the users of a service or sensors collecting data.

This problem has been investigated from different points of view (Section II). However, the following three important aspects have not been considered so far, which constitute the **three main contributions** of our paper.

**First**, as shown in Fig. 1, network services typically consist of *upstream* flows (from source to, e.g., servers of service functions) and *downstream* flows (vice versa) [2], resulting in *bidirectional* service templates. Bidirectional services, despite their relevance, have so far been overlooked in placement and scaling models. To address this, we present approaches for

joint scaling and placement of bidirectional network services.

In bidirectional network services, stateful VNFs may require upstream and corresponding downstream flows to pass through the identical instance [3]. In this case, the corresponding paths need to be created with extra care. E.g., in Fig. 1b, the content delivery server must forward the flows originating from sources $S_1^A$ and $S_2^A$ to the instance of the stateful firewall that has already seen the corresponding upstream flow and can pass it onwards to the right users. Our model ensures the correctness of path creation for flows traversing stateful VNFs.

Using existing solutions that cannot handle bidirectional network services with stateful components, upstream and downstream parts of such services have to be mapped to the network separately. Moreover, to ensure that a certain stateful component is traversed by the same flows in both directions, the services might need to be divided into even smaller parts with fixed endpoints. For example, the service in Fig. 1d should be broken into: $S^B \rightarrow$Fwl$\rightarrow$vSrv, vSrv$\rightarrow$ParCtrl$\rightarrow$Fwl, and Fwl$\rightarrow S^B$. Dividing templates into sub-templates and embedding them sequentially in separate steps requires extra effort compared to our solution of embedding bidirectional templates in a single step. More importantly, in each step of such a sequential process, the requirements of the next steps cannot be considered, possibly resulting in more resource consumption, higher delay, and even capacity violations.

**Second**, service owners may want to reuse some VNFs across multiple services they submit to a service platform (like in Fig. 1). To support these use cases, our joint placement and scaling model allows reusing instances of VNFs across multiple services, if service owners so choose.

**Third**, for the foreseeable future, both physical and virtual network functions have to be supported. PNFs are placed in inalterable locations in the network with fixed sets of resources, like the content delivery server pinned to node 1 in Fig. 1. Our model can handle services that are composed of VNFs as well as PNFs, which is another less-investigated topic in the placement and scaling area.

This paper is structured as follows. We first position our work among related approaches (Section II). Then, we describe our model for a multi-objective optimization problem for **joint scaling and placement** of **bidirectional** network services composed of **virtual or physical** network functions, which may include **stateful** components, as well as components with a **fixed location** (Section III) and prove it to be NP-complete (Section III-E). We formalize the problem as a multi-objective optimization problem (Section IV). We introduce a heuristic (Section V) that can find good solutions quickly and evaluate the quality and runtime of our solutions (Section VI).

## II. Related Work

We build our model upon the joint scaling and placement (JointSP) model from of our previous work [1]. We extend the unidirectional service model of JointSP to support network services with several bidirectional flows starting from each source. We refer to the model in this paper as the *Bidirectional JointSP (B-JointSP)*. Unlike JointSP, our model supports (1) network services with stateful instances, (2) services composed of VNFs and (legacy) PNFs at fixed locations, and (3) reusing service components across network services.

From a theoretical point of view, another related problem is Virtual Network Embedding (VNE): nodes and edges of a virtual network are mapped to nodes and paths in the substrate network, respectively. Similar to our problem, VNE is also an NP-hard problem [4]. The most important difference of VNE to B-JointSP concerns scaling. In VNE, virtual networks have a fixed size and structure. Therefore, the number of required virtual nodes and their interconnections have to be determined in a separate step. We perform scaling and placement in a single step. This way, we take characteristics of the substrate network and the sources into account leading to better solutions. Moreover, our approach considers changing data rates of a flow, resource requirements depending on the load, and latencies between instances. These considerations are usually not taken into account by VNE approaches.

In several related models regarding placement of network functions [5], [6], multiple requests for an instance of the same VNF can be mapped to the same node, which is similar to the possible reuse of VNFs in our model. Most of the related models allow fixing the location of start and end points of services. In addition to that, in B-JointSP, instances of any intermediate VNF can be fixed as well, e.g., to model legacy network functions. The placement model from Moens and De Turck [7] supports hybrid networks, which partly consist of dedicated physical hardware, similar to our model that supports combinations of VNFs and PNFs in services.

Most of the existing models consider scaling [8], [9] and placement [10], [11], [12] separately. Ghaznavi et al. [13] focus on optimizing existing embeddings while minimizing the overhead of modifications, which is also considered in JointSP and B-JointSP. Similar to our model, Mijumbi et al. [14] also consider online modifications of existing embeddings. However, they assume that VNF instances are already placed in the network and requests for these VNFs are then mapped to the instances.

While our work shares similarities with many of these papers, it is unique in considering bidirectional flows and combining them with other aspects, so far only considered in isolation in other work.

## III. Model and Problem

Our goal is to embed a set of network services in a *substrate network*. Once embedded, these network services will be serving flows that possibly consist of upstream and downstream traffic. We describe each network service by a *flexible service template*. Based on the data rate resulting from different flows in multiple *source* locations, each service template is scaled to create an *overlay*. Overlays include all required instances per service component as well as their location in the substrate network, their required resources and their ingoing and outgoing data rates. In this section, we describe how we model each of these entities. Parts of these definitions, e.g., basic definition of templates and components,

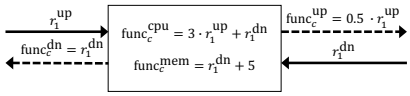



Fig. 2. Resource demands and data rates of an example component

are similar to the JointSP model [1], which we include here for completeness.

### A. Substrate Network

The substrate network $G_{\text{sub}}{=}(V, L)$ is a connected directed graph. Each *node* $v \in V$ is annotated with CPU and memory capacities, $\text{cap}_{\text{cpu}}(v)$ and $\text{cap}_{\text{mem}}(v)$. Each *link* $l \in L$ is associated with a maximum data rate $\text{cap}_{\text{dr}}(l)$ and a delay $d(l)$. Other types of resources can easily be modeled in a similar fashion. We assume any connections inside a network node can be realized with unlimited link capacity and zero delay (e.g., if a network node represents a cluster of machines).

### B. Service Templates

A service template $T = (C_T, A_T)$ describes a network service. *Components* $c \in C_T$ represent the VNFs or PNFs of the network service and the directed *arcs* $A_T$ between the components specify the structure of the service. $\mathcal{T}$ is the set of all active service templates in a network. Fig. 1c and 1d show example templates for bidirectional services.

Each component $c \in C_T$ has separate (possibly empty) vectors of ingoing connection points (briefly, inputs) for upstream and downstream traffic, as well as (possibly empty) vectors of outgoing connection points (briefly, outputs) for upstream and downstream traffic.

The resource consumption for each component $c$ depends on the data rates at the upstream and downstream *inputs* and is defined by functions $\text{func}_c^{\text{cpu}}, \text{func}_c^{\text{mem}}$, representing the required amount of CPU and memory, respectively. Similarly, the outgoing data rates are relative to the incoming data rates and are specified by vectors of functions $\text{func}_c^{\text{up}}, \text{func}_c^{\text{dn}}$ for the upstream and downstream outputs, respectively. If there are multiple outputs in one direction, the traversing flows can be split across these outputs as defined by these functions. Fig. 2 shows example functions for a component that receives an expected data rate of $r_1^{\text{up}}$ and $r_1^{\text{dn}}$ on its upstream and downstream inputs. The functions define the resource demands and outgoing data rates of the component using the ingoing data rates.

To correctly define bidirectional services and distinguish upstream and downstream traffic of the flows, we need to associate roles to components in a service. First, each service has a mandatory single *source* component (SRC, for short), e.g., $\text{S}^{\text{A}}$ and $\text{S}^{\text{B}}$ in Fig. 1c,1d. Each instance of a source component (e.g., representing different populations of users in different geographic locations) has a fixed location and a given outgoing data rate for each flow starting at this source. Source components do not consume resources. Second, some components in a service can have the role of an *END* component (e.g., Srv and vSrv in Fig. 1c,1d). ENDs change the direction of traffic from upstream to downstream in a bidirectional service template. They have only upstream inputs and downstream outputs. Each bidirectional service template has at least one END component. Third, components that are neither SRC nor END are called *intermediate* components (INT, for short). These components send out flows received at an upstream (downstream) input through an upstream (downstream) output. These roles are assigned to each component by the owner of the service, who can define when the upstream traffic turns into downstream traffic.

A service template $T$ is scaled and placed in the substrate network according to the location and the data rate of flows produced by the sources at their respective locations. For every source instance location $v \in V$, we assume we are given the set of flows together with their data rates $(f, r_f) \in F_{T,v}$. $S_T = \{F_{T,v} | v \text{ a source location of } T\}$ collects this information (which typically changes over time).

Additionally, components can be specified as *stateful*, indicating that their instances maintain some internal state for each traversing flow. If both upstream and downstream traffic of a flow traverse an instance of a stateful component, they have to traverse *exactly the same instance* in both directions. This is not required for upstream and downstream traffic of a flow traversing the same *stateless* component; in that case, the traffic may be routed over different instances. When traversing a component, each flow is handled by exactly one instance of that component. In this way, no state inconsistency can occur for flows that traverse a stateful component only in one direction. For load balancing, different flows may be assigned to different instances of a component.

A directed arc $a \in A_T$ connects exactly one output of a component to exactly one input of another component in the service template. To distinguish and correctly transmit flows over the right inputs and outputs, we assign each arc either upstream or downstream direction. Upstream (downstream) arcs can only connect upstream (downstream) outputs to upstream (downstream) inputs. Each arc $a$ is annotated with a delay bound $d_{\text{max}}(a)$, specifying the maximum delay that can be tolerated between the corresponding components.[1]

### C. Overlays

Based on sources $S_T$ and the substrate network $G_{\text{sub}}$, an overlay $G_{\text{OL}}(T) = (I_{\text{OL}}, E_{\text{OL}})$ for each service template $T$ is created. An overlay contains at least one *instance* $i \in I_{\text{OL}}$ for each component $\text{Comp}(i) \in C_T$ and at least one directed *edge* $e \in I_{\text{OL}}$ per arc $\text{Arc}(e) \in A_T$. Fig. 1 shows example overlays of the templates shown in Fig. 1c and 1d embedded in the substrate network. After the services have been scaled in Fig. 1b, there are two instances of the source component $\text{S}^{\text{A}}$ and one instance of $\text{S}^{\text{B}}$. Accordingly, component Fwl is instantiated twice.

[1]The delay bound can alternatively be defined for the round-trip delay. Bounding it per-arc, however, allows a more fine-grained control.

### D. Problem Formulation

B-JointSP is the problem of finding the optimal embedding for a set of service templates in the substrate network. Various optimization goals are conceivable, for example minimizing the following metrics individually or in combination:

- Over-subscription of resources
- Number of instances that should be added or removed
- Total resource consumption of network services
- Total delay for embedded network services

For service template embedding, the following inputs are required:

- Substrate network
- A set of (bidirectional) service templates, possibly with stateful components, possibly sharing some components
- For each template, a set of flows and sources
- A set of instances pinned to fixed locations (optional)
- A previous embedding of the service templates to reflect scenario changes (optional)

B-JointSP can be applied either for the initial embedding of network services into an empty network or for adjusting an existing embedding. Adjustment is needed if there is a change in any of the inputs, e.g., if templates or source locations are added/removed, flow data rates change, or network capacities change. When embedding a service template, each instance in its overlay is mapped to a node and each edge to a path in the substrate network, respecting capacity constraints. We assume that at most one instance per component can be mapped to each network node. If necessary, rather than placing another instance of an existing component at one node, our model scales up the existing instance in that location or scales it out and places the new instance in another location.

Our model is flexible with respect to priorities for the optimization metrics. For example, over-subscription of resources could be forbidden or allowed; over-subscription avoids rejecting requests and improves utilization but jeopardizes service metrics. During the embedding process, one of our objectives is to minimize the maximum over-subscription over all node and link resources. Alternatively, strict limits for the amount of over-subscription can be set.

If multiple service templates include the same component $c$ (e.g., specified using the same identifier), instances of $c$ can be reused in the overlays of these service templates (if this is undesirable, one can easily create a copy of $c$ with another identifier). In this case, the traffic belonging to each network service has to be separated to ensure correct forwarding of their flows. Therefore, we adapt each reused component $c$, by replicating its in- and outputs and adjusting functions $\text{func}_c^{\text{cpu}}, \text{func}_c^{\text{mem}}, \text{func}_c^{\text{up}}, \text{func}_c^{\text{dn}}$. Each overlay can use its own in- and outputs. Fig. 3 shows how the CPU, memory, and data rate functions of the example component from Fig. 1a are adapted to allow two service templates to share this component. The constant values of the combined functions remain unchanged after adaptation. In this way, reusing the component results in a lower idle resource consumption than the case where separate components are used for each template.

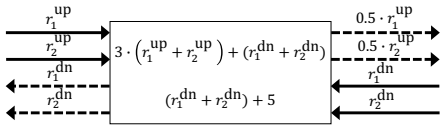



Fig. 3. Resource demands and data rates of the example component Fig. 1, adapted to be shared between two templates

### E. Problem Complexity

Using polynomial-time reduction, we show that for an instance of the B-JointSP problem as defined in Section III-D, it is an NP-complete problem to decide if a solution exists where the over-subscription of (node and link) resources is zero. Based on the given load and resource capacities, it is possible to check in polynomial time in the size of the problem input whether an embedding of a set of templates results in any over-subscription. The size of the solution is also polynomial in the size of the input, so the problem is in NP.

Our problem is an extension of the JointSP problem, which has been proven to be NP-complete [1], [15]. We show a reduction of JointSP to B-JointSP, proving it NP-hard. Given an instance of JointSP, we construct an instance of B-JointSP as follows. As JointSP only considers unidirectional service templates, for every template component in JointSP, we consider the inputs/outputs as upstream inputs/outputs. Similarly, we consider every template arc as an upstream arc. JointSP does not include any delay bound for arcs, so for every arc we set the maximum delay to infinity. In JointSP, every instance of a source component $c$ at node $v$ with data rate $r$ is specified as $(v, c, r)$. There are no stateful instances in JointSP, so the flows from sources can be distributed freely over different instances of each component.

To create a corresponding scaling and load balancing behavior in the B-JointSP, we transform every such source instance of every template $T$ into a source instance with $M$ flows, $F_{T,v} = \{(f_1, r/M), (f_2, r/M), \cdots, (f_M, r/M)\}$. $M$ is a sufficiently large number to create data rate values with the desired precision. E.g., if an implementation of JointSP supports values with 2 digits after the decimal point for data rates of overlay edges, we translate each source instance with data rate $r$ into $r/0.01$ flows starting from this source, each flow having a data rate of $0.01$. We assume all input parameters are rational numbers; hence, there is a limited number of digits after the decimal point.

Using the remaining input parameters directly as provided for JointSP, we now have a complete instance of the B-JointSP problem. If we have a solution for a JointSP problem instance with no violation of capacity constraints, then B-JointSP also has a corresponding solution without over-subscription. Similarly, combining the data rates of different flows from each source instance into a joint data rate, a solution with no over-subscription found for B-JointSP is also a solution with no violations for JointSP.

TABLE I
PARAMETERS

| Symbol | Definition |
|---|---|
| $v \in V$, $l \in L$ | Substrate network nodes and links |
| $\text{cap}_{\text{cpu}}(v)$, $\text{cap}_{\text{mem}}(v)$ | CPU and memory capacity of node $v$ |
| $\text{cap}_{\text{dr}}(l)$, $d(l)$ | Capacity and delay of link $l$ |
| $c \in C_T$, $a \in A_T$ | Components and arcs of template $T$ |
| $n^{\text{up}}_{\text{in}}(c)$, $n^{\text{dn}}_{\text{in}}(c)$ | Number of upstream and downstream inputs of component $c$ |
| $n^{\text{up}}_{\text{out}}(c)$, $n^{\text{dn}}_{\text{out}}(c)$ | Number of upstream and downstream outputs of component $c$ |
| $\text{func}^{\text{cpu}}_c$, $\text{func}^{\text{mem}}_c$ | Functions for CPU and memory demands of component $c$ |
| $\text{func}^{\text{up}}_c$, $\text{func}^{\text{dn}}_c$ | Functions for upstream and downstream outgoing data rates from component $c$ |
| $\text{src}(a)$, $\text{dst}(a)$ | Component where arc $a$ begins and ends |
| $d_{\max}(a)$ | Maximum delay for arc $a$ |
| $F_{T,v} \in S_T$ | Source flows of template $T$ at node $v$ |
| $(f, r_f) \in F_{T,v}$ | Flow $f$ with data rate $r$ |
| $(c, v) \in X$ | An instance of component $c$ that is fixed to node $v$ |
| $x^*_{c,v}$ | Equals 1 iff an existing instance of component $c$ is available at node $v$ |
| $i \in I_{\text{OL}}$, $e \in E_{\text{OL}}$ | Instances and edges of overlay |
| $\mathcal{T}$ | All templates to be embedded |
| $\mathcal{C} = \bigcup_{T \in \mathcal{T}} C_T$ | All components from templates in $\mathcal{T}$ |
| $\mathcal{C}_{\text{SRC}}$, $\mathcal{C}_{\text{INT}}$, $\mathcal{C}_{\text{END}} \subset \mathcal{C}$ | All SRC, INT and END components |
| $\mathcal{C}_{\text{fixed}}$, $\mathcal{C}_{\text{state}} \subset \mathcal{C}$ | All fixed and stateful components |
| $\mathcal{A} = \bigcup_{T \in \mathcal{T}} A_T$ | All arcs of templates in $\mathcal{T}$ |
| $\mathcal{A}_{\text{up}}$, $\mathcal{A}_{\text{dn}} \subset \mathcal{A}$ | All upstream and downstream arcs |
| $\mathcal{S} = \bigcup_{T \in \mathcal{T}} S_T$ | All sources of templates in $\mathcal{T}$ |
| $\mathcal{F}$ | All flows from all sources of all templates |

TABLE II
VARIABLES

| Variable | Definition |
|---|---|
| $x_{c,v}$ | 1 iff an instance of component $c$ is mapped to node $v$ |
| $\delta_{c,v}$ | 1 iff $x_{c,v} \neq x^*_{c,v}$, i.e., an instance of component $c$ is added or removed at node $v$ |
| $\text{cpu}_{c,v}$, $\text{mem}_{c,v}$ | CPU/memory demand of the instance of component $c$ at node $v$, or 0 (if no such instance exists) |
| $t^{\text{up}}_{c,v,f}$, $t^{\text{dn}}_{c,v,f}$ | 1 iff upstream/downstream traffic of a flow $f$ traverses an instance of component $c$ at node $v$ |
| $\text{in}^{\text{up}}_{c,v,f}$, $\text{in}^{\text{dn}}_{c,v,f}$ | Vector of data rates at inputs of the instance of component $c$ at node $v$, corresponding to flow $f$, or an all-zero vector |
| $\text{out}^{\text{up}}_{c,v,f}$, $\text{out}^{\text{dn}}_{c,v,f}$ | Vector of data rates at outputs of the instance of component $c$ at node $v$, corresponding to flow $f$, or an all-zero vector |
| $e_{a,v,v',f}$ | 1 iff for an arc $a$, an overlay edge between nodes $v$ and $v'$ corresponding to flow $f$ is created |
| $z_{a,v,v',l}$ | Data rate on link $l$ corresponding to an arc $a$ that connects an instance of component $c$ at node $v$ to an instance of component $c'$ at node $v'$, or 0 |
| $\zeta_{a,v,v',l}$ | 1 iff $z_{a,v,v',l} > 0$ |
| $\psi_{\text{cpu}}$, $\psi_{\text{mem}}$, $\psi_{\text{dr}}$ | Maximum CPU/memory/data rate over-subscription |

The reduction can be performed in polynomial time in the size of the input, so B-JointSP is an NP-hard problem. Together with the fact that it is in NP, it follows that our problem is an NP-complete problem.

## IV. Mixed-Integer Program Formulation

In this section, we present a mixed-integer programming (MIP) formulation for B-JointSP, which is partly inspired by the MIP formulation of JointSP [1]. All constraints are linear or can be linearized. The problem is a mixed-integer linear programming (MILP) if functions $p_c, m_c, r^{\text{up}}_c, r^{\text{dn}}_c$ are linear for each component $c$. The formulation, however, also works with non-linear functions. Given the inputs described in Section III-D, the following MIP can be used to find optimal solutions to B-JointSP. Table I summarizes the input parameters. Decision variables are presented in Table II. $M$ represents a constant that is sufficiently large, used in the so-called Big-M formulations.

### A. Constraints

Fixed components and sources with their corresponding flow data rates are assigned to their pre-defined locations:

$$\forall v \in V, \forall c \in \mathcal{C}_{\text{SRC}} : x_{c,v} = \begin{cases} 1 & \text{if } \exists F_{T,v} \in \mathcal{S}, c \in C_T \\ 0 & \text{else} \end{cases} \tag{1}$$

$$\forall v \in V, \forall c \in \mathcal{C}_{\text{fixed}} : x_{c,v} = \begin{cases} 1 & \text{if } \exists (c, v) \in X \\ 0 & \text{else} \end{cases} \tag{2}$$

$$\forall v \in V, \forall f \in \mathcal{F}, \forall c \in \mathcal{C}_{\text{SRC}}, \forall T \in \mathcal{T} :$$
$$\text{out}^{\text{up}}_{c,v,f} = \begin{cases} r_f & \text{if } \exists (f, r_f) \in F_{T,v} \\ 0 & \text{else} \end{cases} \tag{3}$$

When going from one solution to another, we track the added/removed instances on each node (Const. 4). If an instance of a component is created on a node, the right number of upstream (Const. 5,7) and downstream (Const. 6,8) inputs and outputs should be created on each instance (we represent the $k$-th element of a vector $w$ by $(w)_k$).

$$\forall c \in \mathcal{C}, \forall v \in V : \delta_{c,v} = \begin{cases} x_{c,v} & \text{if } x^*_{c,v} = 0 \\ 1 - x_{c,v} & \text{if } x^*_{c,v} = 1 \end{cases} \tag{4}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n^{\text{up}}_{\text{in}}(c)] : (\text{in}^{\text{up}}_{c,v,f})_k \leq M \cdot x_{c,v} \tag{5}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n^{\text{dn}}_{\text{in}}(c)] : (\text{in}^{\text{dn}}_{c,v,f})_k \leq M \cdot x_{c,v} \tag{6}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n^{\text{up}}_{\text{out}}(c)] : (\text{out}^{\text{up}}_{c,v,f})_k \leq M \cdot x_{c,v} \tag{7}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n^{\text{dn}}_{\text{out}}(c)] : (\text{out}^{\text{dn}}_{c,v,f})_k \leq M \cdot x_{c,v} \tag{8}$$

The data rate of upstream/downstream traffic of flows traversing an instance of a component determines the data rate of the upstream/downstream traffic that leaves that instance (Const. 9/10). $\underline{0}$ denotes an all-zero vector of appropriate length. As instances of END components (Section III-B) can only have upstream inputs and downstream outputs, they are reflected by a special rule (Const. 11). We keep track of the instances of components that upstream or downstream traffic of each flow traverses (Const. 12–15). This is required to make

sure that each flow traverses exactly the same instance of a stateful component in both directions (Const. 16).

$$\forall c{\in}\mathcal{C}, \forall v{\in}V, \forall f{\in}\mathcal{F}:$$

$$\text{if } c{\in}\mathcal{C}_{\text{INT}}: \text{out}^{\text{up}}_{c,v,f}{=}\text{func}^{\text{up}}_{c}(\text{in}^{\text{up}}_{c,v,f}){-}(1{-}x_{c,v}) \cdot \text{func}^{\text{up}}_{c}(\underline{0}) \tag{9}$$

$$\text{if } c{\in}\mathcal{C}_{\text{INT}}: \text{out}^{\text{dn}}_{c,v,f}{=}\text{func}^{\text{dn}}_{c}(\text{in}^{\text{dn}}_{c,v,f}){-}(1{-}x_{c,v}) \cdot \text{func}^{\text{dn}}_{c}(\underline{0}) \tag{10}$$

$$\text{if } c{\in}\mathcal{C}_{\text{END}}: \text{out}^{\text{dn}}_{c,v,f}{=}\text{func}^{\text{dn}}_{c}(\text{in}^{\text{up}}_{c,v,f}){-}(1{-}x_{c,v}) \cdot \text{func}^{\text{dn}}_{c}(\underline{0}) \tag{11}$$

$$M \cdot t^{\text{up}}_{c,v,f} \geq \sum_{k\in[1,n^{\text{up}}_{\text{in}}(c)]} (\text{in}^{\text{up}}_{c,v,f})_k + \sum_{k\in[1,n^{\text{up}}_{\text{out}}(c)]} (\text{out}^{\text{up}}_{c,v,f})_k \tag{12}$$

$$t^{\text{up}}_{c,v,f} \leq M \cdot \sum_{k\in[1,n^{\text{up}}_{\text{in}}(c)]} (\text{in}^{\text{up}}_{c,v,f})_k + \sum_{k\in[1,n^{\text{up}}_{\text{out}}(c)]} (\text{out}^{\text{up}}_{c,v,f})_k \tag{13}$$

$$M \cdot t^{\text{dn}}_{c,v,f} \geq \sum_{k\in[1,n^{\text{dn}}_{\text{in}}(c)]} (\text{in}^{\text{dn}}_{c,v,f})_k + \sum_{k\in[1,n^{\text{dn}}_{\text{out}}(c)]} (\text{out}^{\text{dn}}_{c,v,f})_k \tag{14}$$

$$t^{\text{dn}}_{c,v,f} \leq M \cdot \sum_{k\in[1,n^{\text{dn}}_{\text{in}}(c)]} (\text{in}^{\text{dn}}_{c,v,f})_k + \sum_{k\in[1,n^{\text{dn}}_{\text{out}}(c)]} (\text{out}^{\text{dn}}_{c,v,f})_k \tag{15}$$

$$\text{if } c{\in}\mathcal{C}_{\text{state}}: t^{\text{up}}_{c,v,f}{=}t^{\text{dn}}_{c,v,f} \tag{16}$$

We ensure that a single flow is not divided over multiple instances of a component (Section III-B). For this, if a flow traverses an instance at node $v$, we allow the flow to enter and exit this instance using exactly one edge for its respective arc (Const. 18,20). Instances of END components again need special treatment (Const. 17). For each edge created this way, the data rate is also calculated on the corresponding input/output of its source/destination instances (Const. 19,21).

$$\forall a{\in}\mathcal{A}, \forall v{\in}V, \forall f{\in}\mathcal{F}:$$

$$\text{if } \text{src}(a){\in}\mathcal{C}_{\text{END}}: \sum_{v'\in V} e_{a,v,v',f}{=}t^{\text{up}}_{c,v,f} \tag{17}$$

$$\text{if } a{\in}\mathcal{A}_{\text{up}} \text{ from output } k \text{ of } \text{src}(a) \text{ to input } k' \text{ of } \text{dst}(a):$$

$$\sum_{v'\in V} e_{a,v,v',f}{=}t^{\text{up}}_{c,v,f} \tag{18}$$

$$\sum_{v'\in V} (\text{out}^{\text{up}}_{\text{src}(a),v',f})_k \cdot e_{a,v',v,f}{=}(\text{in}^{\text{up}}_{\text{dst}(a),v,f})_{k'} \tag{19}$$

$$\text{if } a{\in}\mathcal{A}_{\text{dn}} \text{ from output } k \text{ of } \text{src}(a) \text{ to input } k' \text{ of } \text{dst}(a):$$

$$\sum_{v'\in V} e_{a,v,v',f}{=}t^{\text{dn}}_{c,v,f} \tag{20}$$

$$\sum_{v'\in V} (\text{out}^{\text{dn}}_{\text{src}(a),v',f})_k \cdot e_{a,v',v,f}{=}(\text{in}^{\text{dn}}_{\text{dst}(a),v,f})_{k'} \tag{21}$$

Data rates of individual flows over created edges are then mapped to links in the substrate network; we ensure flow conservation over the path(s) they take (Const. 22–25). During path creation, the total delay of the links corresponding to an edge cannot exceed the maximum delay specified for the corresponding arc (Const. 26). We prevent an overlay edge being mapped to a path with a loop (Const. 27).

$$\forall a{\in}\mathcal{A}, a \text{ starts at output } k \text{ of } \text{src}(a), \forall v, v_1, v_2{\in}V:$$

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

$$\begin{cases} 0 & \text{if } v{\neq}v_1, v{\neq}v_2 \\ 0 & \text{if } v{=}v_1{=}v_2 \\ \sum_{f\in\mathcal{F}} e_{a,v_1,v_2,f} \cdot (\text{out}^{\text{up}}_{\text{src}(a),v_1,f})_k & \text{if } v{=}v_1, v_1{\neq}v_2, a{\in}\mathcal{A}_{\text{up}} \\ \sum_{f\in\mathcal{F}} e_{a,v_1,v_2,f} \cdot (\text{out}^{\text{dn}}_{\text{src}(a),v_1,f})_k & \text{if } v{=}v_1, v_1{\neq}v_2, a{\in}\mathcal{A}_{\text{dn}} \end{cases} \tag{22}$$

$$\forall l{\in}L: z_{a,v_1,v_2,l} \leq M \cdot \zeta_{a,v_1,v_2,l} \tag{23}$$

$$\forall l{\in}L: \zeta_{a,v_1,v_2,l} \leq M \cdot z_{a,v_1,v_2,l} \tag{24}$$

$$\forall l{\in}L: \zeta_{a,v_1,v_2,l} \leq \sum_{f\in\mathcal{F}} e_{a,v_1,v_2,f} \tag{25}$$

$$\sum_{l\in L} \zeta_{a,v_1,v_2,l} \cdot d(l) \leq d_{\max}(a) \tag{26}$$

$$\forall v'v''{\in}L, \text{if } v''v'{\in}L: \zeta_{a,v_1,v_2,v'v''} + \zeta_{a,v_1,v_2,v''v'} \leq 1 \tag{27}$$

The combined data rate of the flows on upstream and downstream inputs of each created instance determines the resource demands of that instance (Const. 28, 29).

$$\forall c{\in}\mathcal{C}, \forall v{\in}V:$$

$$\text{cpu}_{c,v}{=}\text{func}^{\text{cpu}}_{c}\Big(\sum_{f\in\mathcal{F}} \text{in}^{\text{up+dn}}_{c,v,f}\Big){-}(1{-}x_{c,v}) \cdot \text{func}^{\text{cpu}}_{c}(\underline{0}) \tag{28}$$

$$\text{mem}_{c,v}{=}\text{func}^{\text{mem}}_{c}\Big(\sum_{f\in\mathcal{F}} \text{in}^{\text{up+dn}}_{c,v,f}\Big){-}(1{-}x_{c,v}) \cdot \text{func}^{\text{mem}}_{c}(\underline{0}) \tag{29}$$

We also keep track of the maximum over-subscription of node and link resources to minimize or bound it if necessary (Const. 30–32).

$$\forall c{\in}\mathcal{C}, \forall v{\in}V: \sum_{c\in\mathcal{C}} \text{cpu}_{c,v} - \text{cap}_{\text{cpu}}(v) \leq \psi_{\text{cpu}} \tag{30}$$

$$\forall c{\in}\mathcal{C}, \forall v{\in}V: \sum_{c\in\mathcal{C}} \text{mem}_{c,v} - \text{cap}_{\text{mem}}(v) \leq \psi_{\text{mem}} \tag{31}$$

$$\forall l{\in}L: \sum_{a\in\mathcal{A},v,v'\in V} z_{a,v,v',l} - \text{cap}_{\text{dr}}(l) \leq \psi_{\text{dr}} \tag{32}$$

### *B. Objectives*

Based on the problem formulation in Section III-D, we define the following objective functions for the MIP:

- $\text{obj}_1$: Minimize maximum resource over-subscription

$$\text{min. } \psi_{\text{cpu}}{+}\psi_{\text{mem}}{+}\psi_{\text{dr}}$$

- $\text{obj}_2$: Minimize the number of added/removed instances

$$\text{min. } \sum_{j\in\mathcal{C},v\in V} \delta_{j,v}$$

- $\text{obj}_3$: Minimize total resource consumption

$$\text{min. } \sum_{j\in\mathcal{C},v\in V} (\text{cpu}_{j,v}{+}\text{mem}_{j,v}){+} \sum_{a\in\mathcal{A},v,v'\in V,l\in L} z_{a,v,v',l}$$

- $\text{obj}_4$: Minimize total delay

$$\text{min. } \sum_{a\in\mathcal{A},v,v'\in V,l\in L} d(l) \cdot \zeta_{a,v,v',l}$$

In practice, jointly optimizing all four objectives is necessary to serve the requirements of service and platform owners. Therefore, we define the following lexicographical combination of the four objectives:

$$\text{minimize } w_1 \cdot \text{obj}_1{+}w_2 \cdot \text{obj}_2{+}w_3 \cdot \text{obj}_3{+}w_4 \cdot \text{obj}_4$$

Weights $w_1, \ldots, w_4$ should be selected such that objectives' ranges do not overlap and each objective has a clear priority. The actual weights depend on the use case (Section VI).

**Algorithm 1** Embedding procedure

```
1: remove overlays of previous templates ∉ 𝒯
2: add empty overlays for new templates ∈ 𝒯
3: for all template T ∈ 𝒯 do
4:     add/remove/update source instances and flows
5:     update fixed instances
6:     set current direction to upstream
7:     for all instance i in topological order do
8:         if i is not used and not fixed/source then
9:             remove i and continue with the next instance
10:        if i is an instance of an end component then
11:            set current direction to downstream
12:        for all output k of i in current direction do
13:            get arc a and flows leaving k
14:            if a ∉ A_T then
15:                continue with next output
16:            update mapping of flows to edges
```

## V. Heuristic Approach

Resource demands of network services and capacity of the substrate network change frequently, requiring quick reactions. We present a heuristic that finds good solutions for B-JointSP quickly, either from scratch or adapting an existing solution. It consists of initialization, sequential embedding, and iterative improvement.

### A. Initialization

During initialization, the heuristic computes the shortest paths between all pairs of nodes in the substrate network based on the Floyd-Warshall algorithm. We assign each link $l$ a weight $w(l) = 1/(\text{cap}_{\text{dr}}(l) + 1/d(l))$.[2] In this way, paths with low delay, consisting of links with high capacity are favored. As flows are mapped to these precomputed paths when embedding a service template, the load on the links increases, possibly resulting in over-subscription. Iterative improvements then aim at reducing or avoiding over-subscription.

If multiple templates are provided as input, they are sorted to start with embedding the *heaviest* template. The weight of each template is a rough estimation of its expected total resource demand, calculated assuming all of its flows are mapped to a single instance per component and to a single link between the components.

### B. Embedding Procedure

After initialization, the heuristic creates an initial solution using the embedding procedure shown in Algorithm 1. The general workflow of this procedure is similar to the embedding procedure of JointSP [1], but we have substantially modified the details to support bidirectional flows, stateful and legacy components, and instances shared among different overlays.

At the start, empty overlays are added for new templates and overlays of old templates (with $T \notin \mathcal{T}$) are removed (Lines 1–2). The templates in set $\mathcal{T}$ are embedded sequentially in the order determined during initialization (Line 3). When embedding a template, first, its source instances are updated to be consistent with the sources $S_T$ by adding or removing source instances or updating the flows leaving these sources (Line 4). Fixed instances are updated similarly (Line 5).

All other instances can be placed and scaled dynamically to react to the current demand (Lines 6–16). The embedding procedure processes the instances sequentially in topological order such that each instance is processed at most once in upstream and once in downstream direction. The topological order depends on the structure of the template and can be derived by following its arcs. In bidirectional templates, first, the upstream direction is considered. Once an instance of an end component is reached, the direction is switched to downstream as flows return towards their sources.

When processing an instance, the heuristic computes the outgoing flows and their data rates at each output based on the ingoing flows in the current direction. As each output $k$ belongs to a separate arc $a$, the heuristic considers the outputs sequentially, mapping the flows leaving $k$ to edges along $a$. If $a$ belongs to another template (reusing the same instance, Fig. 3), the output is skipped. Otherwise, the mapping of the outgoing flows to edges along $a$ is updated as follows: First, the mapping of flows that no longer leave output $k$ (but previously did) is removed. The data rate of other flows that are already mapped to edges is updated.

Next, new flows that were not previously mapped are assigned to edges. The heuristic assigns flows in random order, each to exactly one edge. To determine the best edge, all nodes with enough remaining resources to handle the flow and within maximum delay $d_{\max}(a)$ from the location of the current instance are considered as candidates. Among the candidates, the heuristic chooses the one reachable with the shortest path. The flow is mapped to the edge going to the chosen node. If not yet existing, the heuristic creates the corresponding edge and destination instance. If no candidate nodes with enough free resources exist within $d_{\max}(a)$ from the current location, the node with lowest over-subscription is chosen.

A special case is when $a$ is a downstream arc back to a stateful component. In this case, each flow has to return to the exact same instance of the component that it already traversed in upstream direction. Hence, each flow is mapped to the edge returning to the corresponding stateful instance. After all flows are mapped to edges, the heuristic removes any unused edges leaving $k$, i.e., edges without any flows mapped to them.

### C. Iterative Improvement

While the embedding procedure ensures correct embeddings and tries to optimize the embedding for the objectives defined in Section III-D, its sequential processing can lead to suboptimal results. Especially when embedding bidirectional templates with stateful components, it can unnecessarily oversubscribe nodes. As flows return to the same stateful instances they traversed in upstream direction, the resource consumption of these stateful instances increases. However, as this increase in resource consumption cannot easily be anticipated, it can

[2] If $d(l) = 0$, we assume $w(l) = 0$.

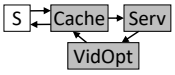



Fig. 4. Video streaming template used for evaluation

lead to over-subscription – even if there are still available resources at neighboring nodes.

Therefore, the overlay of each template is modified iteratively based on tabu search [16] by picking a random instance that is neither source nor fixed and declaring it as *tabu*. The overlay is then reset and recreated using the embedding procedure again but disallowing to place the tabu instance at the same location as before. This leads to a different distribution of instances and the load, possibly decreasing or avoiding over-subscription.

The heuristic modifies each of the overlays in multiple iterations while maintaining three different solutions. In this way, we omit unsuccessful modifications but also enable exploring slightly worse solutions without losing the best found solution. Hence, the tabu-based improvement scheme allows to overcome local optima while reusing the efficient embedding procedure to ensure correctness.

## VI. Evaluation

In this section, we evaluate the performance of the MIP and the heuristic with respect to multiple metrics. As the heuristic achieves close-to-optimal results in a significantly shorter time than the MIP, we use the heuristic for further analyzing the features of our solutions in larger scenarios.

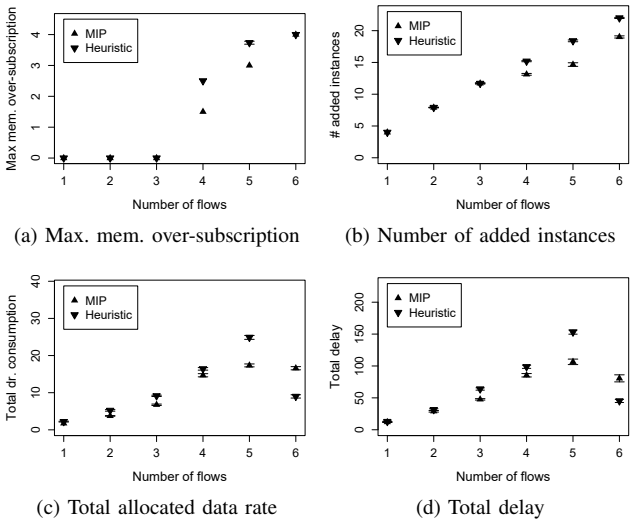
### A. Evaluation Setup

We implemented the MIP and the heuristic in Python 3.5 and used the Gurobi Optimizer, v 7.0.2. All simulations were performed on machines with Intel Xeon E5-2695 v3 CPUs running at 2.30 GHz and using GNU Parallel [17] to automatically assign jobs to available cores. For reproducibility, our evaluation code is publicly available [18].

Our evaluation is based on the Abilene data set [19], representing a real backbone network. Due to the long runtime of the MIP, for comparing the performance of the MIP and the heuristic, we only considered the western half of the network, consisting of 6 nodes and 14 directed links with uniform capacities. Additionally, we evaluated the heuristic using much larger networks with hundreds of nodes. We calculated the link delay $d(l)$ for each link $l$ based on the distances between the geographical locations of the nodes.

Based on common NFV use cases [20], we chose a video streaming network service as the bidirectional template to be embedded (Fig. 4), in which users (represented by source component S) request videos from a cache or server. Before streaming videos from the cache, they are transcoded by a video optimizer, which reduces their data rate by 50% [21].

Using the Abilene network and the video streaming template, we created different problem instances with 1 to 6 flows leaving sources at randomly varying locations in the network. All MIP results for 1 to 4 flows have 0% optimality gap; results




(a) Max. mem. over-subscription (b) Number of added instances

(c) Total allocated data rate (d) Total delay

Fig. 5. Comparison of the MIP's and heuristic's results for increasing load.

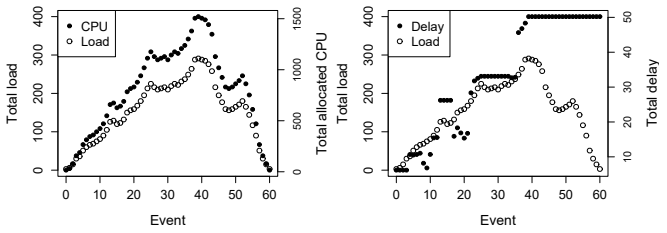
for 5 and 6 flows have an average gap of 6.9% with some outliers with at most 22% gap. We used the MIP and heuristic to create initial embeddings on an empty network, minimizing the lexicographical combination of the four objectives $\text{obj}_1$–$\text{obj}_4$ (high to low priority), as defined in Section IV-B.

### B. Performance and Runtime of MIP and Heuristic

Considering $\text{obj}_1$, both approaches can completely avoid memory over-subscription for 1 to 3 flows by distributing the load over different instances and nodes. For problem instances with more flows, the available memory no longer suffices and is over-subscribed relative to the number of flows (Fig. 5a). While being in the same order of magnitude, the heuristic creates embeddings with significantly higher maximum memory over-subscription compared to the optimal results of the MIP (up to 67% higher average). The results are similar for CPU resources whereas link capacities are never over-subscribed.

These observations are similar for the other three objectives: The heuristic can closely approximate the optimal results of the MIP with just small deviations in all objectives (Fig. 5). An exception occurs for 6 flows: The heuristic cannot find solutions with as few instances as the MIP (worsening $\text{obj}_2$), but these instances are useful to create shorter paths between the instances with lower allocated data rate and shorter delay than the MIP (improving $\text{obj}_3$ and $\text{obj}_4$).

Where the MIP needs minutes to hours, the heuristic finds solutions in milliseconds to seconds. For example, the worst case runtimes for problem instances with 4 flows were 137.1 hours (MIP) compared to 6.7 s (heuristic). Using this advantage of the heuristic, we further evaluated the performance of our solutions solving additional problem instances using the heuristic. We used the largest substrate network in the SNDlib library [19], consisting of 161 nodes and 664 directed links, and with 10 sources and 30 flows. We ran simulations starting with an empty network, embedding the template of Fig. 4, and adapting the embeddings based on the increasing and




(a) Changes to the allocated CPU based on the changes in load

(b) Changes to the delay based on the changes in load

Fig. 6. Analysis of the solutions for a large problem instance with a series of events that change the overall load of the network.

decreasing load in the form of 60 events. For these problems, the heuristic found embeddings in 42.31 s in average.

Fig. 6a shows the total allocated CPU over these events, based on the total load from all flows at all source locations. Resource allocation clearly adapts to the load; with increasing load, more CPU is allocated to the service components and when the load decreases, the allocated resources are decreased. We observed a similar trend for the amount of allocated memory and link capacity. Fig. 6b shows an important consequence of the choice of priorities for the four objectives in our simulations. For us, the highest priority after minimizing the over-subscription of resource was minimizing the number of added and removed instances to limit of the costs and overheads associated with starting and stopping VNFs. Minimizing the delay had the lowest priority among the objectives, as our approaches ensure that the delay remains within the maximum delay boundary defined in each template. Looking at the total delay for the embedded template, it can be observed that after the peak load around event 40, although the load decreases, the delay does not decrease. Adhering to the priority of minimizing the number of added/removed instances, the existing instances (possibly placed farther from the sources) are still used with reduced traffic amounts rather than removing or migrating them.

## VII. Conclusion

B-JointSP is a comprehensive, flexible, and realistic model for NFV, for jointly scaling and placing complex network services taking the downstream direction of flows returning to their sources into account as well as stateful VNFs, reusing network functions across network services, and legacy network functions. The formulated MIP can be used for optimal scaling and placement of templates in small substrate networks. For larger networks, our heuristic can find close-to-optimal solutions within seconds. In practice, this short runtime allows quick adaptation of embeddings to ongoing load fluctuations in the network. Our evaluations showed that the heuristic approximates the MIP's solution quality and can even outperform it with respect to some metrics when optimizing multiple objectives. Depending on the priorities of the service or platform providers, our solutions can be adjusted to deliver the required results. We have tested a configuration where over-subscription of resources is minimized, the amount of allocated resources adapts exactly to the laod, and the number of added or removed instances are kept as low as possible to avoid unnecessary costs in a frequently changing load setup.

## Acknowledgments

This work has been partially supported by the SONATA project, funded by the European Commission under grant number 671517 through the Horizon 2020 and 5G-PPP programs, the 5GTANGO project, funded by the European Commission under grant number 761493 through the Horizon 2020 and 5G-PPP programs, and the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

## References

[1] S. Dräxler, H. Karl, and Z. A. Mann, "Joint optimization of scaling and placement of virtual network services," in *IEEE/ACM CCGrid*, 2017.

[2] T. Nadeau and Q. Quinn, "Problem statement for service function chaining," IETF, Internet Request for Comments RFC 7498, 2015.

[3] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro, "Service function chaining use cases in mobile networks," IETF Draft draft-ietf-sfc-use-case-mobility-07, 2016.

[4] 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.

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

[6] M. Savi, M. Tornatore, and G. Verticale, "Impact of processing costs on service chain placement in network functions virtualization," in *IEEE NFV-SDN*, 2015.

[7] H. Moens and F. De Turck, "VNF-P: A model for efficient placement of virtualized network functions," in *IEEE CNSM*, 2014, pp. 418–423.

[8] P. Chuprikov, S. Nikolenko, and K. Kogan, "On demand elastic capacity planning for service auto-scaling," in *IEEE INFOCOM*, 2016.

[9] C. Fuerst, S. Schmid, L. Suresh, and P. Costa, "Kraken: Online and elastic resource reservations for multi-tenant datacenters," in *IEEE INFOCOM*, 2016.

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

[11] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan, "Optimal virtual network function placement in multi-cloud service function chaining architecture," *Computer Communications*, vol. 102, pp. 1–16, 2017.

[12] M. T. Beck and J. F. Botero, "Scalable and coordinated allocation of service function chains," *Computer Communications*, vol. 102, pp. 78–88, 2017.

[13] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba, "Elastic virtual network function placement," in *IEEE CloudNet*, 2015, pp. 255–260.

[14] R. Mijumbi, J. Serrat, J. Gorricho, N. Bouten, F. De Turck, and S. Davy, "Design and evaluation of algorithms for mapping and scheduling of virtual network functions," in *IEEE NetSoft*, 2015.

[15] S. Dräxler, H. Karl, and Z. A. Mann, "JASPER: Joint Optimization of Scaling, Placement, and Routing of Virtual Network Services," 2017. [Online]. Available: https://arxiv.org/abs/1711.10839

[16] F. Glover, "Tabu search – part I," *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989.

[17] O. Tange, "GNU Parallel - the command-line power tool," *The USENIX Magazine*, vol. 36, no. 1, pp. 42–47, 2011.

[18] "Implementation of the MIP and the heuristic." [Online]. Available: https://github.com/CN-UPB/B-JointSP

[19] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly, "SNDlib 1.0 – Survivable Network Design Library," in *ENOG INOC*, 2007.

[20] ETSI NFV ISG, "Network functions virtualisation (NFV): Use cases," Group Specification ETSI GS NFV 001 V1.1.1, 2013.

[21] Tellabs, "Mobile video optimization concept and benefits," White Paper, 2011, https://goo.gl/RLkWFX (Accessed Dec. 2017).