Elastic Network Virtualization

Max Alaluna Nuno Neves Fernando M. V. Ramos
LaSIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
malaluna@alunos.fc.ul.pt, nuno@di.fc.ul.pt, fvramos@ciencias.ulisboa.pt

***Abstract*—Network virtualization allows multiple tenant networks to coexist on a shared infrastructure. Core to its realization is the embedding of virtual networks onto the underlying substrate. Existing approaches are not suitable for cloud environments as they lack a fundamental requirement: elasticity. To address this issue we explore the capacity of flexibly changing the topology of a virtual network by proposing an embedding solution that adds elasticity to the tenant's virtual infrastructures. For this purpose, we introduce four primitives to tenants' virtual networks – including scale in and scale out – and propose new algorithms to materialize them. The main challenge is to enable these new services while maximizing resource efficiency and without impacting service quality. Instead of further improving existing online embedding algorithms – always limited by the inability to predict future demand – we follow a different approach. Specifically, we leverage network migration for our embedding procedures and to introduce a new reconfiguration primitive for the infrastructure provider. As migration introduces network churn, our solution uses this technique judiciously, to limit the impact to running services. Our solution improves on network efficiency over the state-of-the-art, while reducing the migration footprint by at least one order of magnitude.**

***Index Terms*—virtual network embedding, network virtualization**

I. Introduction

The emergence of cloud services has fundamentally changed the nature of computing. By outsourcing computing to the cloud, businesses are relieved from the burden of operating and maintaining an infrastructure, while being provided with the flexibility and elasticity required to respond to the dynamic demand for their services. Until recently, the pay-as-you-go model of the cloud has been restricted to computing services (renting VMs) and storage resources (renting storage space and access). This model falls short on providing the required guarantees for modern services, however. Certain applications, such as MapReduce or High-performance computing (HPC), have strict networking constraints, for instance with respect to latency and throughput, that traditional cloud services fail to guarantee [18], [27], [30]. The unpredictable network performance of cloud services negatively affects user workloads and increases tenant's costs [5], [27], [36], hindering deployment of several classes of applications in the cloud [5].

Some efforts have been made to improve cloud services with network guarantees [5], [16]. First, *SecondNet* [16] proposed the Virtual Data Center abstraction, providing tenants with bandwidth guarantees for pairs of VMs, and a data center network virtualization architecture to materialize it. As this solution provides a dense connectivity that makes it difficult to multiplex multiple tenants on the underlying infrastructure,

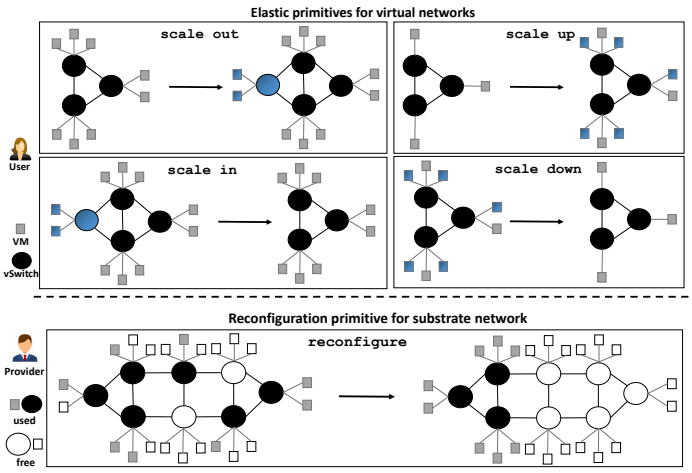


Figure 1: ElasticVN primitives

Oktoptus [5] enhanced cloud offerings with two new virtual network abstractions: the virtual cluster, geared for data-intensive traffic, and the virtual oversubscribed cluster, good for applications featuring local communication patterns. Other solutions have improved over these works by considering more advanced bandwidth allocations [20], [26] and additional guarantees, such as bounded packet delay [19].

Limitations of the state-of-the-art. While improving the situation, these solutions are limited to simple network abstractions that leave aside many typical workloads. Whereas for traditional enterprise workloads a flat L2 networking service may suffice, modern analytics workloads typically demand for L3 routing, and large web services often require hierarchical network topologies with multiple tiers. The experience from production-level environments (as reported in [22]) confirms this scenario: as deployments mature, tenants migrate to these more complicated workloads. *This makes a case for offering tenants virtual networks with **arbitrary** topologies.*

A second requirement when a tenant migrates its workloads to the cloud is **elasticity**: allowing re-scaling of the virtual networks. This is useful to enable tenants to automatically respond to "black friday" events. For instance, by expanding the network with new nodes (scale-out), or by increasing the capacity of specific nodes to serve more VMs (scale-up). Or, conversely, to scale-down to save costs when service demand is low. Unfortunately, approaches that provide scaling of virtual networks consider a limited set of topologies (e.g., [10], [33]), using specialized embedding approaches that do not work for arbitrary topologies. Conversely, the approaches that solve the Virtual Network Embedding (VNE) problem and thus enable

arbitrary topologies [6], [35] are static and do not allow the virtual network to evolve. The first work that considers both requirements, enabling elasticity in virtual networks, was recently proposed by Michel et al. [24]. However, the solution introduces high network overhead and relatively low efficiency, as will be made clear later.

Our contribution. In this paper we introduce four network primitives to user VNs which provide the necessary elasticity to a cloud environment. These primitives are depicted in Figure 1. Each node represents a virtual switch that connects multiple VMs. The first two primitives (top of the figure) enable network expansion (scale-out and scale-up), and the latter two (middle of Figure 1) represent network contraction (scale-in and scale-down). We detail these primitives in section IV-A.

A strawman approach to the problem of extending the network would consist in mapping the new nodes with traditional VNE algorithms while retaining the existing mappings. The problem is that this results in resources being fragmented across the substrate network. As we show in Section V, this leads to inefficiencies and negatively impacts application performance. An alternative would be to investigate new VNE heuristics for the problem. *We note, however, that **any** practical solution to the online VNE problem has a fundamental limitation: its lacks knowledge about future requests*. In practice, VN requests are not known in advance: they arrive dynamically and stay in the network for an arbitrary period of time.

As such, we follow an alternative approach: to consider network migration jointly with embedding. This is enabled by modern techniques that allow migration *of a network* of virtual machines with little to no impact to application or system performance [14], [15]. This enables – for the first time – the remapping of *running* virtual networks, and thus gives hope to overcome the limitation of traditional VNE approaches. We are, however, informed that migration, if not performed judiciously, can inadvertently overload the network, and result in application performance degradation [21], [28]. This is effectively one of the drawbacks of the state-of-the-art solution [24]. As such, we opt not to perform a complete VN remapping as response to a scale-out request. A complete remap potentially requires several network elements to be migrated, resulting in an undesirable high level of churn. By contrast, our solution, `ElasticVN`, migrates virtual network elements selectively. The key idea is to migrate VN elements only when it is estimated this will have positive impact in terms of substrate network efficiency (by making better use of resources) and VN performance (by reducing path lengths).

Enabling tenants to scale their networks over time leads to resource fragmentation, a phenomenon we show in Section V. Fragmentation leads to an increase in path lengths in the virtual networks, as resources belonging to the same virtual network are mapped across distant regions of the substrate. As a result, communication between adjacent nodes (in the virtual topology) needs to traverse additional switches, increasing latency and bandwidth variability between virtual machines. This has a negative impact in job completion times and overall application performance [32]. In addition, physical resources

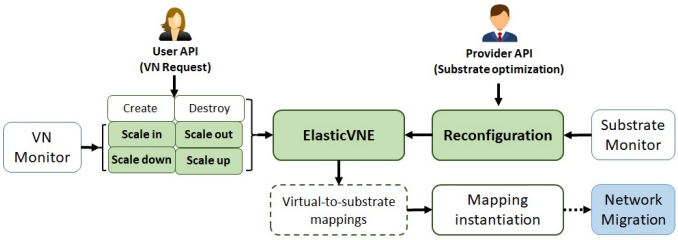


Figure 2: Elastic VN system

increasingly become unusable with a decline of the virtual network acceptance ratio, leading to a decrease in revenue for the infrastructure provider, even though the physical network technically still has sufficient capacity to accommodate a request. To mitigate the effects of resource fragmentation, we extend the infrastructure provider API with a *reconfiguration* primitive (bottom of Figure 1). The idea is to explicitly remap virtual networks, in order to optimize network performance and resource usage. To implement this primitive, we again make (judicious) use of network migration [14], [15].

Figure 2 illustrates the entire system, centered in our contributions. The current state of network virtualization is shown in white, and our extensions are shown in green. The user API is extended to allows users not only to create and destroy virtual networks, but also to re-scale them. This procedure can be either on-demand, or be triggered automatically by a VN monitor that is pre-configured by the user (e.g., when customer requests are above a certain threshold the VN can be scaled up in a pre-configured manner). The VN requests are then embedded using `ElasticVNE`, resulting in a mapping of the virtual to the substrate network. The provider can also interfere, triggering a reconfiguration (on-demand or by monitoring key metrics, such as average path lengths). After the mappings are defined, the system implements them in the substrate. When specific elements need to be migrated, the system runs the network migration tool [14], [15] (beyond the scope of this paper, thus shown in blue).

To evaluate our solution we have performed large scale simulations considering arbitrary virtual and substrate topologies. Our results show that ElasticVNE improves acceptance ratios over [24] by 20% for new requests and reduces path lengths by over 3x. Importantly, these results are achieved with a very small migration footprint. Our solution requires 10x less migrations and reduces migration time by 100x over the state-of-the-art [24]. Our results go further in concluding that an *unrestricted use of migration is harmful* for network efficiency. Network reconfiguration is shown to further increase the gains in terms of path length reduction, ultimately improving user application performance.

To summarize, we make the following contributions:

- We present and motivate primitives that allow tenants to re-scale their networks, and a reconfiguration primitive to assist providers.
- We propose new algorithms to materialize these primitives. One key novelty is that we leverage network migration techniques recently made available.

- Our large scale simulations show that our solution achieves higher acceptance ratios and reduced path lengths over the state-of-the-art, limiting the migration footprint, thus avoiding excessive network churn.

II. Motivating use cases

We start this section by presenting the rationale for a new requirement in modern virtual environments: arbitrary network topologies. In this context, we then motivate the need for elastic virtual networks with four use cases: (i) Scale out to satisfy costumers' demand; (ii) Scale in to save costs; (iii) Rapid deployment; and (iv) NFV service chaining.

Arbitrary topologies. Users increasingly want to tailor the virtual network topology to their applications. For instance, a streaming video application provider may require a tree to distribute streaming to a group of receivers. By contrast, certain applications contain centralized services for which a hub-and-spoke topology is more suitable. In a virtualized environment, as virtual networks grow, the complexity and size of the logical networks tend to grow steadily, alongside the number of hypervisors. The experience from production environments confirms these diverse use cases. As an example, in the typical VMware NSX/NVP deployment virtual networks have hundreds of VMs attached to a number of logical switches interconnected by a few logical logical routers, complemented with ACLs [22].

Scale-out and scale-up. A network that is built to provide a fixed level of service capacity will occasionally be overwhelmed by peak loads that occur on rare but important occasions, such as on a "black friday" or on the Christmas week. The key advantage of cloud solutions is mostly their auto-scaling ability, namely to automatically scale-up (increasing the capacity of a resource) and scale-out (adding instances of a resource). This enables users to satisfy increasing demands without turning away customers.

Scale-in and scale-down. On average, however, services tend to be operating at far below the maximum capacity required for peak loads. As such, the ability to scale-down (reducing the capacity of a resource) and scale-in (removing instances of a resource) is fundamental to save costs and achieve complete elasticity.

Deployment and testing. Distributed system designers typically want to run experiments under a variety of topologies to explore how their new protocol performs in different settings. In addition, testing how distributed applications respond to scale up and scale down events is usually required before actual deployment.

Service chaining. The elasticity of NFV has been recently exploited in 5G networks [29] and in data centers [11]. For instance, 5G networks are centered around providing isolated network slices featuring high throughput and low latency to its customers. Each slice contains network services composed of virtual network functions. To adapt to traffic loads in a network slice, a single network function, or even an entire service chain, may need to be scaled-out and scaled-in dynamically by adding or removing virtual nodes [29].

III. Abstracting the Network

This section explains how we abstract the infrastructure, capturing the fundamental characteristics of the elements composing the substrate network and the user virtual networks requests.

Substrate network. We model the substrate network as a weighted undirected graph $G^S = (N^S, E^S, A^S_N, A^S_E)$, where $n^S \in N^S$ represents the nodes and $e^S \in E^S$ are the edges (or links) connecting them. A node is a network element capable of forwarding packets. Nodes and edges are characterized by attributes A^S_N and A^S_E, respectively.

In our setting, a physical/virtual machine contains a software switch n^S to interconnect the local compute elements (e.g., containers) to the data center infrastructure. Since they share the existing resources, we employ attribute $cpu(n^S)$ to aggregate the total CPU capacity available in the substrate machine for network tasks and for user application processing. An edge connecting two switches is characterized by its bandwidth capacity (attribute $bw(e^S) > 0$) and its average latency ($lat(e^S) > 0$). Including the latter is important as inter-cloud links typically have much higher latency when compared to internal data center connections (we have empirically confirmed this, as we explain in Section V). In summary, the node and edge attributes are, respectively, $A^S_N = \{cpu(n^S) : n^S \in N^S\}$ and $A^S_E = \{(bw(e^S), lat(e^S)) : e^S \in E^S\}$.

Virtual network. Similar to the substrate, VNs are modeled as weighted undirected graphs $G^V = (N^V, E^V, A^V_N, A^V_E)$, where N^V is the set of virtual nodes, E^V is the set of virtual edges (or links), and A^V_N / A^V_E correspond to the virtual nodes and edges attributes, respectively. The $cpu(n^V)$ attribute of a node n^V represents a requirement associated to the virtual switch and its locally connected virtual hosts (containers in our virtualization platform). The attribute $cpu(n^V)$ is thus the sum of all necessary processing capacity (for network and applications tasks). For the virtual edges, $bw(e^V)$ and $lat(e^V)$ represent the required bandwidth and maximum acceptable latency. In summary, the two sets of attributes are: $A^V_N = \{cpu(n^V) : n^V \in N^V\}$ and $A^V_E = \{(bw(e^V), lat(e^V)) : e^V \in E^V\}$.

Virtual Network Request (VNR). VNRs are composed of a description of the virtual network graph plus two additional parameters: $VNR = (G^V, id, Type)$. These include a unique identifier for the request, $VNR.id$, and the type of request, $VNR.Type$, namely: (i) an arrival, for a new VNR; (ii) a departure, when the virtual network ends operation and its resources can be reclaimed; (iii) an upgrade, when the user asks for an increase on the resources associated with a VNR already in operation, including scale-out and scale-up; (iv) a downgrade, when the user requests for a decrease on the resources consumed by an existing VNR, including scale-in and scale-down.

IV. Elastic Virtual Networks

In this section we present new network primitives for tenants' virtual networks and for the operator's substrate

infrastructure, and the algorithms we propose to materialize them.

A. Elastic VN primitives

Figure 1 (top) illustrates four re-scaling primitives, to scale virtual networks both horizontally (out/in) and vertically (up/down):

- **scale-out**: to add new network elements (e.g., switches) to a running virtual network (VN);
- **scale-in**: to remove network elements from a running VN;
- **scale-up**: to increase the capacity of existing resources of a VN (e.g., to connect new VMs to a switch); and
- **scale-down**: to decrease the capacity of existing resources of a VN.

In addition, we present a reconfiguration primitive (bottom of Figure 1) for operators to be able to increase the resource efficiency of the substrate:

- **reconfigure**: to remap existing virtual networks in order to improve resource usage and performance.

As will be made clear in the next section, to their materialization the substrate we consider is capable of migrating network elements using existing techniques [12], [13].

B. Elastic VNE algorithms

We divide our solutions to materialize these primitives in two sets: user-driven and provider-driven. The first corresponds to the algorithms that respond to user requests (left side of Figure 2), including launching new VNs, and removing, upgrading (scale-up and scale-out), or downgrading (scale-down and scale-in) running VNs. The second corresponds to the algorithms required for substrate reconfiguration (top right of the same figure), triggered by actions from the provider side. Some of the algorithms we present next are common to both sets, as will be made clear.

User-driven embedding. Our approach derives an embedding for the virtual network requests (VNRs) that arrive. The goal is to find the most appropriate mapping for the virtual nodes and edges, taking into consideration the requirements stated by the user (which appear in the form of attributes). When the available resources are not enough to fulfill the requirements, namely in the case of new requests or upgrades, mapping fails and an error is signaled.

Algorithm 1, *elasticEmbedding()*, responds to user requests. It resorts to a few global variables to keep information about the execution (see Table I). Some are also used by other algorithms. The algorithm implements an infinitive loop that receives and processes every VN request that reaches the system. Four types of requests are addressed: *arrival*, *departure*, *upgrade*, and *downgrade*.

The procedure starts by waiting for the next virtual network request vnr (line 2). In case it is a new VNR, it attempts to find a suitable embedding (line 4) by calling *getMap()*. This procedure is presented later as Algorithm 3. If successful, the mapping is stored in the *Nets* structure (line 6), and the *vnr* is deployed on the substrate (line 7). The function `getMap()` also modifies the global variables that hold information about the residual resources (R_N and R_E), reflecting the consumption of CPU and bandwidth[1]. Otherwise, an error is signaled (line 9).

Name	Description
G^S	substrate network specification (see Section III)
$Nets$	a set that keeps the embedding for the currently deployed requests. Each element is a pair (v, m), where v is a VNR and m a mapping of the virtual network to the substrate. Initialized: $Nets \leftarrow \emptyset$
R_N	a set that keeps for each substrate node the residual CPU, i.e., the amount of CPU that has not been consumed. Initialized: $R_N \leftarrow \{rcpu(n^S) = cpu(n^S) : n^S \in N^S\}$
R_E	a set that keeps for each substrate edge the residual bandwidth, i.e., the amount of bandwidth that has not been consumed. Initialized: $R_E \leftarrow \{rbw(n^S) = bw(e^S) : e^S \in E^S\}$

Table I: Global variables employed by the algorithms.

Algorithm 1: elasticEmbedding()

Input: G^S, $Nets$, R_N, R_E

```
1  while (true) do
2      vnr ← waitForVNR();
3      if (vnr.Type == arrival) then
4          map ← getMap(vnr, G^S, R_N, R_E, ∅, ∅);
5          if (map ≠ ∅) then
6              addVNR(vnr, map, Nets);
7              deploy(vnr, map, G^S, ∅, ∅);
8          else
9              error(vnr.id, "Error mapping new VNR");
10     else
11         (v, m) ← getVNR(vnr.id, Nets);
12         if (vnr.Type == departure) then
13             terminate(v, m, G^S, R_N, R_E);
14             updateNodeLinkResources(G^S, R_N, R_E, m);
15             delVNR(vnr.id, Nets);
16         else
17             if (vnr.Type == upgrade) then
18                 map ← getMap(vnr, G^S, R_N, R_E, v, m);
19             else /* vnr.Type==downgrade */
20                 map ← getDwMap(vnr, G^S, R_N, R_E, v, m);
21             if (map ≠ ∅) then
22                 delVNR(vnr.id, Nets);
23                 addVNR(vnr, map, Nets);
24                 deploy(vnr, map, G^S, v, m);
25             else
26                 error(vnr.id, "Could not upgrade VN");
```

The other types of requests are related to changes to running VNs. Therefore, we start by obtaining information about the existing VNR and corresponding mapping (v, n) by consulting the $Nets$ set (line 11). In case of departure, the virtual network is terminated (line 13), the corresponding nodes and edge resources (R_N and R_E) are freed (line 14), and the set $Nets$ is updated (line 15). For upgrades and downgrades, variable *map* gets the replacement mapping by calling *getMap()* (line 18)

[1]For each node n^V in map, which is mapped onto the substrate node n^S, the corresponding residual $rcpu(n^S) \in R_N$ becomes $rcpu(n^S) \leftarrow rcpu(n^S) - cpu(n^V)$. In a similar manner, the residual bandwidth of the substrate edges is modified in R_E.

and *getDwMap()* (line 20), Algorithms 3 and 6, respectively. If the request can be fulfilled, the previous VN is removed from *Nets*, and the new mapping is stored in this set (Lines 22-23). Then, the necessary changes are applied to the substrate: to add or remove nodes and links, and/or to increase or decrease their capacity. In some cases, network migration may be performed, as will be made clear in Algorithm 4.

Algorithm 2: reconfiguration()

Input: G^S, $Nets$, R_N, R_E

1 $NetsO \leftarrow$ orderVNs($Nets$);
2 $i \leftarrow 0$;
3 **forall** $((vnrO, mapO) \in NetsO)$ **do**
4 $\quad tmpRN \leftarrow R_N$; $tmpRE \leftarrow R_E$;
5 $\quad R_N \leftarrow$ removeVNodes(R_N, $NetsO_i$);
6 $\quad R_E \leftarrow$ removeVEdges(R_E, $NetsO_i$);
7 $\quad mNew \leftarrow$ getMap($vnrO_i, G^S, R_N, R_E, 0, 0$);
8 $\quad$ **if** $(mNew \neq \emptyset)$ **then**
9 $\quad\quad Nets \leftarrow (Nets$ - $NetsO_i) \cup (vnrO_i, mNew)$;
10 $\quad\quad$ deploy($vnrO_i$, $mapO_i$ $G^S, vnrO_i, mNew$);
11 $\quad$ **else**
12 $\quad\quad R_N \leftarrow tmpRN$; $R_E \leftarrow tmpRE$;
13 $\quad$ i++;

Provider-driven embedding. As VNs arrive and depart, are updated or downgraded, substrate resources become fragmented. As a consequence, nodes of a VN tend to be located further apart, increasing the length of the substrate paths (in terms of number of hops) that map the virtual edges. This negatively impacts communication latencies, and reduces efficiencies as more substrate links are used. To address these problems, we provide a *reconfiguration* primitive to substrate providers. This enables VN redeployment with the goal of improving mapping efficiencies globally, possibly by migrating selected nodes. Algorithm 2 materializes this primitive.

We first initialize the $NetsO$ set (line 1) with the running VNs ordered by decreasing network sizes (number of virtual nodes). The average path length, $avgPL(v, m)$, is used to break ties (higher values first). We compute this metric as follows. For each pair $(v, m) \in Nets$, where v is a VNR and m is the current mapping, $avgPL(v, m)$ equals:

$$\frac{\parallel \{e^S : \forall e^V \rightarrow \{e^S\},\ e^V \in E^V \wedge e^S \in E^S\} \parallel}{\parallel \{e^V : \forall e^V \in E^V\} \parallel} \qquad (1)$$

where $\parallel . \parallel$ counts the elements in a set, and $e^V \rightarrow \{e^S\}$ represents edge e^V having been mapped onto a path composed of a group of substrate edges $\{e^S\}$. The formula thus outputs the average substrate path length: the ratio of the number of substrate edges used to map all virtual edges of a specific virtual network. Ideally, $avgPL() = 1$, with each virtual edge mapped to a single substrate edge. In practice, multiple substrate edges may have to be employed, and $avgPL() > 1$.

The algorithm enters in a loop, storing first a copy of the original residual resource sets (lines 4), necessary to enable rollback in case of an unsuccessful reconfiguration. The residual resources are then updated as if the VNs selected for reconfiguration were evicted from the substrate (releasing the corresponding CPU and bandwidth) (lines 5-6). Next, we try to remap the VN (line 7). If successful, we update the $Nets$ structure with the new mapping, and redeploy the VNR in the substrate which, again, may include migration of certain nodes (lines 8-10). Otherwise, we rollback (lines 11-12).

Algorithm 3: getMap()

Input: vnr, G^S, R_N, R_E, $vnrP$, $mapP$
Output: map /* node & link mappings */

1 $map \leftarrow \emptyset$;
2 $map.N \leftarrow$ nodeMap($vnr, G^S, R_N, R_E, vnrP, mapP$);
3 **if** $(map.N == \emptyset)$ **then**
4 $\quad$ **return** $\emptyset$;
5 **else**
6 $\quad map.L \leftarrow$
$\quad$ linkMap($vnr, G^S, R_N, R_E, map.N, vnrP, mapP$);
7 $\quad$ **if** $(map.L == \emptyset)$ **then**
8 $\quad\quad$ **return** $\emptyset$;
9 $\quad$ **else**
10 $\quad\quad$ updateResources($R_N, map, mapP$);
11 $\quad\quad$ **return** map;

Embedding general procedures. Next, we present the mapping procedures used by the elastic and reconfiguration algorithms. Algorithm 3 is employed to perform VN embedding, and is called for both new VNRs, for reconfiguration, or to scale out a VN. Since this problem is NP-hard, we resort to a heuristic where nodes are mapped first, followed by mapping of the virtual edges. The goals are to maximize the overall acceptance ratio of users' requests, fulfilling all requirements, and to maximize efficiency (e.g., by minimizing the number of substrate links used).

The procedure starts by initializing the *map* variable, which will store the node (*map.N*) and link (*map.L*) mappings (line 1). Then, node mapping is performed, for each virtual node to be assigned a specific substrate node (line 2). If successful, follows link embedding, to map one substrate path for each virtual edge (line 6). If both operations succeed, the residual resources are updated (line 10). Note that as parameters of the *nodeMap()* and *linkMap()* procedures (to be presented as Algorithms 4 and 5) we include information on the present mappings (*vnrP* and *mapP*), alongside the new request (*vnr*), to accommodate upgrade, downgrade, and reconfiguration.

Algorithm 4 maps virtual nodes to the substrate. The main idea is to rank nodes in the substrate favoring those (i) with more resources available (to balance load), (ii) located closer to substrate nodes that map VN neighbors (to reduce path lengths), and (iii) that minimize migration cost, in case of upgrades and reconfigurations. After initializing the structure $nMap$ that maintains the mappings, an auxiliary variable G^{aux} keeps a copy of the original substrate graph (lines 1-2). Next, the virtual nodes of vnr are stored in ascending order, using as metric the resources required, according with Equation 2 (line 3).

$$scoreV(n^V) = cpu(n^V) \times \sum_{\forall e^V \succ n^V} bw(e^V) \qquad (2)$$

Algorithm 4: nodeMap()

Input: vnr, G^S, R_N, R_E, $vnrP$, $mapP$
Output: $nMap$ /* node mappings */

```
1  nMap ← ∅;
2  G^aux ← G^S;
3  virN ← virScore(vnr);
4  forall (n^V ∈ virN) do
5      virtualNodeMapped ← false;
6      subN ← subScore(n^V, G^aux, vnr, vnrP, mapP);
7      forall (n^S ∈ subN) do
8          cpuV ← cpu(n^V);
9          if isMapped(n^V, n^S, mapP) then
10             cpuV ← cpu(n^V) − getCPU(n^V, vnrP)
11         if (cpuV ⩽ R_N(n^S)) then
12             nMap ← nMap ∪ (n^V, n^S);
13             G^aux ← G^aux − n^S;
14             virtualNodeMapped ← true;
15             break;
16     if (virtualNodeMapped == false) then
17         return ∅;
18 return nMap;
```

where $e^V \succ n^V$ means edge e^V is connected to n^V. The formula basically takes into consideration the CPU and bandwidth requested for the virtual node, returning a higher score for nodes that require more resources. Processing thus starts from the less demanding nodes.

The procedure then loops to embed each virtual node. Variable *virtualNodeMapped* indicates whether a successful mapping was found for this node, and thus is initialized with *false* (line 5). Function *subScore()* is then used to order the substrate nodes in G^{aux} (line 5), using Equations 3 and 4 to compute the ranking.

$$baseScore(n^S) = \frac{\frac{R_N(n^S)}{cpu(n^S)} \times \sum_{\forall e^S \succ n^S} \frac{R_E(e^S)}{bw(e^S)}}{avgDist2Neighbors(n^S, \bar{N}^V)} \quad (3)$$

$$scoreS(n^S) = \frac{baseScore(n^S)}{migrationCost(n^V, n^S)} \quad (4)$$

The first equation attempts to increase the acceptance ratio while minimizing the consumption of substrate links. The value of $baseScore$ is higher for nodes that have a larger share of resources available: $R_N(n^S)/cpu(n^S)$ is the percentage of available CPU at node n^S; and $R_E(e^S)/bw(e^S)$ is the proportion of available bandwidth of a link e^S ending at n^S ($e^S \succ n^S$ has an equivalent meaning to the one above). In addition, it penalizes substrate nodes proportionally to their distance from the virtual nodes already mapped. For this purpose function *avgDist2Neighbors()* computes the average hop distance between n^S and $\bar{N}^S$, which represents the set of substrate nodes where the neighbors of n^V already mapped are placed.

The second equation determines the final score. For new VNRs, it is equal to the $baseScore$. For upgrades and reconfigurations, however, there is a penalty based on the estimate of the cost of migration. Function *migrationCost()* thus returns 1 if node n^V is already mapped into n^S, or if it is a new virtual node. Otherwise, it has a value, larger than 1, proportional to the estimated time necessary to perform the migration. To compute this penalty time, we consider the network latency and bandwidth between the source and destination nodes, and the amount of information required to be moved (i.e., the footprint of virtual hosts and switch). The substrate nodes are placed in the $subN$ set in descending order, ensuring that the nodes with best scores are considered first (line 6).

The loop that follows attempts to map virtual node n^V. First, it adjusts the required CPU (lines 8-10). This is needed to accommodate scale up requests, as the virtual node may already be mapped in the substrate node under consideration n^S. In this case, it is only necessary to provision the additional CPU requested. If enough residual CPU is available, it stores the new mapping (line 12), and updates the auxiliary variable (line 13). By removing n^S from G^{aux} the algorithm guarantees that a substrate node does not map more than one virtual node from the same VN. This decreases the impact of substrate failures to the VN. If embedding fails, the algorithm returns 0 (lines 16-17).

Algorithm 5: linkMap()

Input: vnr, G^S, R_N, R_E, $nMap$, $vnrP$, $mapP$, `maxP`
Output: $lMap$ /* link mappings */

```
1  lMap ← ∅;
2  forall (e^V ∈ vnr.G^V.E^V) do
3      candMap ← ∅;
4      totalBw ← 0;
5      paths ←
         getPaths(e^V, vnr, G^S, R_E, nMap, vnrP, mapP, maxP);
6      foreach (p ∈ paths) do
7          if (lat(e^V) ≥ getLatency(p, G^S)) then
8              bwp ← getMinBandwidth(p, R_E);
9              totalBw ← totalBw + bwp;
10             candMap ← candMap ∪ (e^V, bwp, p);
11     if (totalBw ≥ bw(e^V)) then
12         forall (mp ∈ candMap) do
13             bw(mp) ← ⌈(bw(mp)/totalBw) * bw(e^V)⌉;
14         lMap ← lMap ∪ candMap;
15     else
16         return ∅;
17 return lMap;
```

Algorithm 5 finds a mapping between virtual edges and substrate paths. Each edge is processed individually, searching for a suitable path between the two substrate nodes that embedded its virtual endpoints. The approach is flexible, enabling either single or multiple paths embeddings by adjusting the input `maxP`.

After initializing the link mappings set $lMap$ (line 1) the algorithm enters in a loop to embed each virtual edge. First, it initializes the set that stores the candidate mapping and a

variable that holds the total bandwidth of all candidate paths. Then, it obtains this set of `maxP` paths to connect the two substrate nodes (line 5). In our implementation, we resort to the K-edge disjoint shortest path algorithm to find these paths, using as edge weights the inverse of the residual bandwidths. This ensures that when "distance" is minimized, the algorithm picks the paths that have more bandwidth available.

Then, each candidate path p is evaluated to check if it fulfills the latency requirements (line 7). For this purpose, the *getLatency()* function returns the overall path latency (the sum of latencies of each individual substrate link that forms the path). Each of the paths that fulfills the requirement is stored in $candMap$ (lines 8-10), jointly with its available bandwidth. This is calculated with the *getBandwidth()* function, that returns the bandwidth of the bottleneck link in p. The set will have at most $MaxP$ paths, the constant that defines the degree of multipathing (when set to 1, a single path is used).

Finally, we define how much traffic goes through each path, ensuring that together they provide the requested edge bandwidth (lines 11-13). If enough bandwidth is available in the candidate paths, we update the bandwidth in every path to an amount proportional to their maximum capacity, therefore distributing the load. Then, if successful, the set of paths is added to the link mappings set (line 14).

Algorithm 6: getDwMap()

Input: vnr, G^S, R_N, R_E, $vnrP$, $mapP$
Output: map /* node & link mappings */

1 **forall** $(n^V \in vnrP.G^V.N^V)$ **do**
2 $cpuP \leftarrow cpu(n^V)$;
3 $cpu \leftarrow getCPU(n^V, vnr)$;
4 **if** $(cpu \neq 0)$ **then**
5 $map.N \leftarrow map.N \cup (n^V, getNS(n^V, mapP.N))$;
6 **forall** $(e^V \in vnrP.G^V.E^V)$ **do**
7 $bwP \leftarrow bw(e^V)$;
8 $bw \leftarrow getBW(e^V, vnr)$;
9 **if** $(bw \neq 0)$ **then**
10 $candMap \leftarrow getMapEdge(e^V, mapP.L)$;
11 **foreach** $(mp \in candMap)$ **do**
12 $bw(mp) \leftarrow bw(mp) * [bw/bwP]$;
13 $map.L \leftarrow map.L \cup candMap$;
14 updateNodeLinkResources($G^S, R_N, R_E, map, mapP$);
15 **return** map;

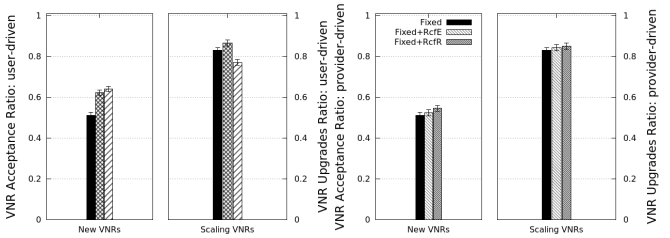
The last procedure is Algorithm 6, necessary to downgrade (scale in and/or down) a previously deployed VNR (recall Algorithm 1). The procedure first adds to $map.N$ the nodes that were not removed from the previous mapping (lines 1-5), and follows a similar procedure for the edges (lines 6-13). It starts by obtaining, for each node, the used CPU resources in the existing embedding ($cpuP$) and the new request (cpu). Function *getCPU()* outputs 0 if the virtual node n^V was removed from the vnr graph (scaled in). Nodes that remain in the graph are thus added to $map.N$. Function *getNS()* outputs the node n^S where n^V was – and will remain – mapped.

For the edges, a similar procedure takes place. The present (line 7) and new (line 8) virtual edge bandwidths are obtained (*getBW()* returns 0 if the edge has been eliminated). Then, the bandwidth associated to each of the paths used to map the virtual edge is updated in the required proportion, in a manner similar to Algorithm 5. The procedure terminates by updating the residual sets (R_N and R_E), releasing the unnecessary resources (line 14).

V. Evaluation

In this section we aim to answer two main questions. Does ElasticVN improve the efficiency and acceptance ratio over the state-of-the-art approaches? At what cost with respect to migration footprint?

Experimental setup. For our experiments we prepared the typical setup to evaluate VNE work [9]. We have extended the VNE simulator developed in [2] to simulate the dynamic arrival of VNRs to the system, and have used the GT-ITM tool [37] to generate the substrate and virtual networks. We have employed the Waxman model to link nodes with a probability of 50% [25].



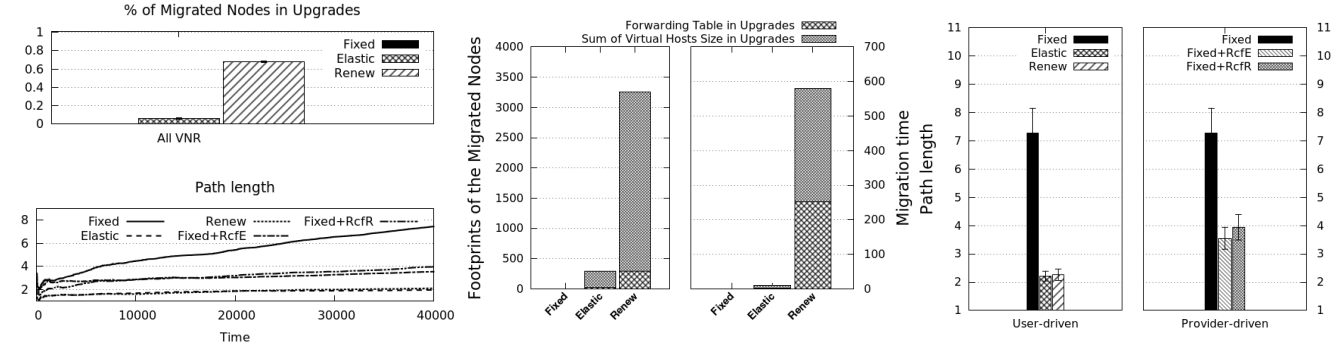
(a) Acceptance ratio: user-driven. (b) Accept. ratio: provider-driven

Figure 3: VNR acceptance ratios.

Substrate networks have a total of 100 nodes. As this solution is integrated into a real network virtualization platform[2], we have set up the simulation parameters with realistic values for this setting. Towards this goal, we have measured, for several consecutive days, the link bandwidth and delays in the target environment. The results were consistently in the hundreds of Mbps. As such, we have set the bandwidth of substrate links – $bw(n^S)$ – as a random variable uniformly distributed between 500 and 1000. The CPU resources – $cpu(n^S)$ – are uniformly distributed between 50 and 100. Given the diversity of the networks considered, we have set up some links (intra-domain) with 1 unit delay, and others (inter-domain) with 20 units delay. Again, these values were set based on our empirical analysis.

VNRs have a number of virtual nodes uniformly distributed between 5 and 20. As the nodes in our setting are switches connecting several virtual hosts (specifically, containers), we have set the node footprint to be uniformly distributed between 25 and 50 for the forwarding table, and between 250 and 500 for the sum of VM/container storage. The first values are based on the forwarding table size of switches [23]. Containers' sizes are also on this range [1], and we assume each virtual switch

[2]We omit its reference to respect the double-blind review process.

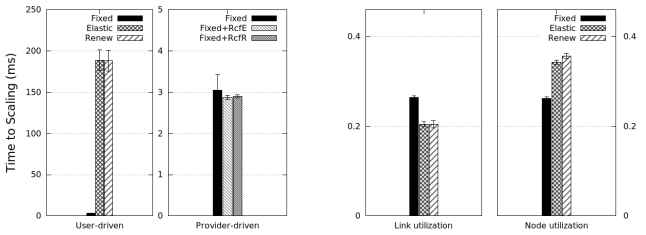


(a) Ratio of migrated nodes (top) and path length (bottom)

(b) Migration cost

(c) Path lengths

Figure 4: Migration costs, path lenghts.



(a) Time to re-scale VNs

(b) Resource usage

Figure 5: Time to re-scale and substrate utilization.

Notation	Algorithm description
Fixed	Scale the VN while fixing the embedded nodes to their substrate nodes, and re-mapping only the new nodes. The state-of-the-art approach for scaling [24].
Elastic	Scale the VN considering our ElasticVNE heuristic.
Renew	Scale the VN by re-mapping all nodes.
Fixed+Rcf^E	Using the *Fixed* baseline, and applying periodic reconfiguration using our ElasticVNE heuristic.
Fixed+Rcf^R	Using the *Fixed* baseline, and applying periodic reconfiguration by re-mapping all nodes. The state-of-the-art approach for reconfiguration [24].

Table II: VNR configurations that were evaluated in the experiments.

to support tens of containers, as is common. In addition, pairs of virtual nodes are connected with a Waxman topology with probability 50%. The CPU of the virtual nodes are uniformly distributed between 10 and 20, and the bandwidth of virtual links between 200 and 400. We assume that VNR arrivals are modeled as a Poisson process with an average rate of 4 VNRs per 100 time units. Each VNR has an exponentially distributed lifetime (Dur) with an average of 1000 time units.

At each 10 VNRs arrivals, the simulator chooses 75% of the embedded VNs to be scaled-in or scaled-out, with 50% probability each. Each VNR extends (scales out) or reduces (scales in) the number of nodes by 30%. For the reconfiguration experiments, we set reconfiguration after 100 events. We have used as baseline embedding algorithm the one proposed in [34]. For link embedding we evaluated both shortest path (SP) and k-edge disjoint SP with k=2. For space reasons we present only the results using the former, but our conclusions generalize. We emphasize, however, that other embedding algorithms could be used – what we aim to evaluate is the use of network migration to assist VN scaling and reconfiguration, not the underlying embedding algorithm used.

We set up 5 experiments (shown in Table II), each considering the same substrate and VNR topologies. We ran every experiment 10 times, for 50k time units each, so on average around 2000 VNRs were simulated per run. The order of arrival and the capacity requirements of each VNR are kept the same for all configurations in Table II, ensuring that they solve equivalent problems.

User-driven results. Figure 3a presents the acceptance ratio for new VNRs and for scaling requests. For *new* requests, with our algorithm `elastic` the acceptance ratio improves around 20% over `fixed` (the state-of-the-art [24]), and is very close to the `renew` approach. Figure 5b illustrates the reason for these improvements: a better use of resources. While node utilization is increased because more requests are accepted, we reduce link utilization by shortening path lengths (shown next). For re-scaling requests, although the acceptance ratio for `fixed` is already high, `elastic` improves by close to 4%. Interestingly, the results for `renew` are worse. This may seem counter intuitive at first, but the reason is illuminating: *an excessive use of migration is harmful*. We found that link mapping failures increase after node mappings that demand migration, as it becomes harder to find substrate paths available due to the high churn.

Importantly, the results obtained with our `elastic` algorithms are achieved with a relatively small migration footprint. This can be observed in Figure 4a (top). With `elastic`, only around 6% of nodes are migrated, which is a figure more than 10x smaller than with the use of `renew`, the solution without migration restrictions. As a result, the migration cost, both in terms of footprint (which translate in number of bytes exchanged) and migration time, is reduced by two orders of magnitude (Figure 4b). The heuristics that use network migration also drastically reduce path lengths. As shown in Figure 4c, paths are shortened by over 3.2x. Figure 4a shows a single representative run, where it is clear paths lengths

are kept consistently small with `elastic`. This reduction is translated in improved virtual network performance (e.g., lower latencies) and better resource usage.

Finally, Figure 5a shows the average time to re-scale VNRs. While the results are similar for the algorithms that consider network migration (`elastic` and `renew`), they increase re-scaling time by over 150ms compared to the `fixed` baseline. The reason is that the latter only considers the new nodes for re-mapping, whereas the others consider *the entire virtual network* in the decision process. Importantly, this delay is reasonable in a realistic environment. For instance, empirical results obtained from a virtualization platform [4] have shown that a running time on the tens or hundreds of miliseconds for the embedding algorithms represents an insignificant fraction of the overall provisioning time (less than 3% in the worst case).

The main conclusion is that the improvements obtained by our `elastic` solution, in both acceptance ratio and path lengths, are achieved with a judicious use of migration and acceptable re-scaling times.

Provider-driven results. Considering reconfiguration alone, we found that the acceptance ratio improves over the `fixed` baseline (Figure 3b), but only very slightly, either with `Rcf^E` (our reconfiguration mechanism that considers migration cost) or with `Rcf^R` [24] (no restrictions for migration). On the other hand, Figure 4c shows that reconfiguration is able to decrease path lengths significantly. Both algorithms more than halve the average path length. Provider-driven reconfiguration is anyway less effective when compared to the user-driven elastic algorithms. This is mainly due to the reconfiguration frequency. While the former is triggered only periodically, the elastic algorithms effectively trigger a reconfiguration for *every* scaling request.

VI. Related work

Virtual network embedding. One of the algorithms most commonly used to address the VNE problem was proposed by Yu et al. [35]. The authors proposed a greedy approach for node mapping and an innovative link mapping approach. As the original problem is NP-hard, the authors considered a substrate with the ability of path splitting (multi-path). As a result, they have shown the problem could be solved as a Multi-Commodity Flow (MCF) problem, for which efficient algorithms exist. Chowdhury et al. [6] have afterwards proposed algorithms that introduced a better coordination between the node and the link mapping phases, improving acceptance ratios. While the literature on the VNE problem is vast [9], few works have considered elasticity. We address these later.

Cloud virtual networks. Some efforts have been made to improve cloud services with network guarantees. SecondNet [17] proposed the Virtual Data Center abstraction, providing tenants with bandwidth guarantees for pairs of VMs. Oktopus [5] enhanced cloud offerings with two abstractions: the virtual cluster and the virtual oversubscribed cluster. Faircloud [26] and EyeQ [20] went further by consider work conserving bandwidth allocations, and Jang et al. [19] providing bounded packet delay. These solutions provide only a limited set of topologies, and use specialized embedding approaches. Modern virtualization solutions [7], [8], [22] improve by enabling *arbitrary* virtual networks. This is achieved by simulating the entire network topology at the edge (mainly at the source host). By contrast, and similar to other virtualization platforms [3], we address the generic VNE problem. We are thus not limited to network nodes at the edge (e.g., our solution is capable of integrating emerging programmable networking hardware). More fundamentally, none of these solutions considers the introduction of elastic primitives to virtual networks.

VN elasticity and substrate reconfiguration. Some related work has considered the possibility of scaling virtual networks. However, most work targets specific topologies. For instance, both [10] and [33] consider virtual cluster topologies only. Zhani et al. [38] consider arbitrary topologies, but do not allow scale in and scale out (i.e., add/remove nodes). While these works consider VM migration in the substrate to enhance efficiency, none considers the migration of a *network* of VMs, limiting their scope. The closest work to ours is that of Michel et al. [24]. Its authors are the first to consider network migration-assisted scaling of arbitrary virtual networks. As we have shown in Section V, our work improves over it by improving network efficiency, increasing accepting ratio by over 20%, with a much lower overhead, reducing the network migration requirement by at least one order of magnitude.

Network Migration. Wang et. al. [31] where amongst the first to propose a management primitive that enables network migration. Their solution, VROOM, allows virtual routers to move between physical routers, without requiring changes to the logical topology. Recent network migration techniques [12], [14] have built upon this work to offer the ability to migrate entire virtual networks, including virtual machines, routers, and links, with negligible downtime. These works do not consider elasticity of network virtualization, and are thus orthogonal to our work. But we leverage these techniques to materialize the virtual network and reconfiguration primitives proposed in this paper.

VII. Conclusion

In this paper, we presented and motivated new primitives to scale virtual networks and to reconfigure the underlying substrate, bringing elasticity to virtual networking. The key novelty of the algorithms we proposed was the (judicious) use of network migration, enabling network reconfiguration to assist in improving resource efficiency. Our simulations have shown that our solution achieves higher acceptance ratios and smaller path lengths when compared to the state-of-the-art, while drastically limiting the migration footprint.

Acknowledgment

We would like to thank the anonymous reviewers for their feedback, which helped improve the paper. This work was supported by FCT through funding of the uPVN project, ref. PTDC/CCI-INF/30340/2017, and LASIGE Research Unit, ref. UID/CEC/00408/2019.

REFERENCES

[1] Docker Repositoty. https://hub.docker.com.

[2] ViNE-Yard. http://www.mosharaf.com/ViNE-Yard.tar.gz.

[3] A. Al-Shabibi et al. Openvirtex: Make your virtual sdns programmable. HotSDN '14.

[4] M. Alaluna et al. Secure multi-cloud network virtualization. *Computer Networks*, 161:45 – 60, 2019.

[5] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.

[6] M. Chowdhury, M. R. Rahman, and R. Boutaba. Vineyard: Virtual network embedding algorithms with coordinated node and link mapping. *IEEE/ACM Trans. Netw.*, 20(1):206–219, Feb. 2012.

[7] M. Dalton, D. Schultz, J. Adriaens, A. Arefin, A. Gupta, B. Fahs, D. Rubinstein, E. C. Zermeno, E. Rubow, J. A. Docauer, J. Alpert, J. Ai, J. Olson, K. DeCabooter, M. de Kruijf, N. Hua, N. Lewis, N. Kasinadhuni, R. Crepaldi, S. Krishnan, S. Venkata, Y. Richter, U. Naik, and A. Vahdat. Andromeda: Performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[8] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure accelerated networking: Smartnics in the public cloud. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, 2018.

[9] A. Fischer et al. Virtual network embedding: A survey. *IEEE Communications Surveys & Tutorials*, 15(4), 2013.

[10] C. Fuerst, S. Schmid, L. Suresh, and P. Costa. Kraken: Online and elastic resource reservations for multi-tenant datacenters. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[11] A. Gember-Jacobson et al. Opennf: Enabling innovation in network function control. SIGCOMM '14.

[12] S. Ghorbani et al. Transparent, live migration of a software-defined network. SOCC'14, pages 3:1–3:14, NY, USA, 2014. ACM.

[13] S. Ghorbani et al. Coconut: Seamless scale-out of network elements. EuroSys'17, 2017.

[14] S. Ghorbani and P. B. Godfrey. Coconut: Seamless scale-out of network elements. EuroSys '17.

[15] S. Ghorbani, C. Schlesinger, M. Monaco, E. Keller, M. Caesar, J. Rexford, and D. Walker. Transparent, live migration of a software-defined network. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 3:1–3:14, New York, NY, USA, 2014. ACM.

[16] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 15:1–15:12, New York, NY, USA, 2010. ACM.

[17] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, CoNEXT '10, 2010.

[18] Q. He, S. Zhou, B. Kobler, D. Duffy, and T. McGlynn. Case study for running hpc applications in public clouds. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, 2010.

[19] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. SIGCOMM '15.

[20] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim. Eyeq: Practical network performance isolation at the edge. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, 2013.

[21] C. Jo, Y. Cho, and B. Egger. A machine learning approach to live migration modeling. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, 2017.

[22] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 203–216, Berkeley, CA, USA, 2014. USENIX Association.

[23] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 15–28, New York, NY, USA, 2017. ACM.

[24] O. Michel, E. Keller, and F. Ramos. Network defragmentation in virtualized data centers. *Sixth IEEE International Conference on Software Defined Systems (SDS)*, 06 2019.

[25] M. Naldi. Connectivity of Waxman Topology Models. *Computer Communications*, 29(1):24–31, Dec. 2005.

[26] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, 2012.

[27] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010.

[28] V. Shrivastava, P. Zerfos, K. Lee, H. Jamjoom, Y. Liu, and S. Banerjee. Application-aware virtual machine migration in data centers. In *2011 Proceedings IEEE INFOCOM*, 2011.

[29] C. Sun et al. Enabling nfv elasticity control with optimized flow migration. *IEEE JSAC*, 36(10), 2018.

[30] G. Wang and T. S. E. Ng. The impact of virtualization on network performance of amazon ec2 data center. In *2010 Proceedings IEEE INFOCOM*, 2010.

[31] Y. Wang et al. Virtual routers on the move: Live router migration as a network-management primitive. SIGCOMM'08, pg 231–242, NY, USA.

[32] C. Wilson et al. Better never than late: Meeting deadlines in datacenter networks. SIGCOMM '11.

[33] L. Yu and Z. Cai. Dynamic scaling of virtual clusters with bandwidth guarantee in cloud datacenters. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[34] M. Yu et al. Rethinking virtual network embedding: Substrate support for path splitting and migration. *SIGCOMM Comput Commun Rev*, 38(2):17–29, Mar. 2008.

[35] M. Yu, Y. Yi, J. Rexford, and M. Chiang. Rethinking virtual network embedding: Substrate support for path splitting and migration. *SIGCOMM Comput. Commun. Rev.*, 38(2):17–29, Mar. 2008.

[36] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, 2008.

[37] E. W. Zegura et al. How to Model an Internetwork. In *IEEE INFOCOM*, pages 594–602, March 1996.

[38] M. F. Zhani, Q. Zhang, G. Simona, and R. Boutaba. VDC Planner: Dynamic migration-aware Virtual Data Center embedding for clouds. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 18–25, May 2013.