Flexible Support of VNF Placement Functions in OpenStack

Simon Oechsner, Andreas Ripke
NEC Laboratories Europe
Kurfürsten-Anlage 36, 69115 Heidelberg
Germany

Abstract—**In the context of virtualization and cloud computing, a currently interesting topic is the movement of network functions into the cloud. Since these Virtualized Network Functions have high performance and availability requirements, the placement and resilient deployment of these functions are important issues. In this paper, we present a real-world mechanism implemented for OpenStack that supports arbitrary placement functions for performance optimization and resilience provisioning. In addition, we provide an example of how this mechanism can be used in practice.**

I. Introduction

Virtualization has become one of the strongest drivers for change in the IT landscape in the recent past. Starting with server virtualization and giving economical incentives for the intensive and multi-tenant use of data centers, its influence on the movement of infrastructure 'to the cloud' has been tremendous.

By now, the benefits of virtualization of IT infrastructure are visible enough to make the softwarization of historically hardware-implemented components feasible and the focus of research. Where in the past network elements such as switches or routers were expensive pieces of equipment due to the implementation of much of their logic in hardware, the trend now goes to Software-Defined Networking (SDN) [1]. One of the main propositions of SDN is the separation of the control plane from the data plane, allowing for cheaper hardware on the one hand, and a more flexible and programmable network management by a software controller on the other.

Following this development, other network functions that in the past needed specialized and thus expensive hardware to satisfy performance requirements are planned to be virtualized in the near future. This is termed Network Function Virtualization (NFV) and holds not only big economic promise for telecommunication operators, but also changes completely the way network functions are developed and deployed.

The premise of NFV is that network functions (NFs) such as firewalls or mobile core network functions can be flexibly instantiated and placed on COTS hardware, allowing for on-demand usage of resources and adaptation to different demand situations. While this value proposition is shared with other virtualization use-cases, the movement of NFs and other critical IT infrastructure to a virtualized environment faces harder challenges due to the performance and availability requirements of these functions.

As an example, some functions from the area of voice telecommunication use to have a '5 nines' requirement, i.e., an availability of 99.999%. As well, a chain of these network functions might need to fulfill strict performance requirements such as end-to-end delay and throughput. The orchestration part of the virtual infrastructure management needs to take into account these requirements when deploying components such as Virtualized Network Functions (VNF) or other critical service components. The ETSI ISG on NFV has for example identified this need with respect to resilience [2]. It is a currently on-going research issue to extend or replace the existing orchestration mechanisms implemented for less demanding use-cases to be suitable for this new environment.

One of the core functions concerned is the placement of virtual machine (VM) or NF instances. In many cases, VM placement is understood as the mapping of the requirements of a VM, such as number of CPU cores, memory, or line card speed, to a physical host that can supply the according resources. This kind of placement function is typically provided by existing cloud management systems, and suffices for many applications run in a virtualized environment.

However, for performance-critical distributed infrastructure, an additional type of placement becomes important, namely the placement of a set of instances in relation to each other. High delays or low throughput between instances due to a non-optimal placement in different parts of the physical infrastructure is bound to have a serious negative impact on application performance. This kind of placement is less considered in currently existing cloud management solutions, although approaches for optimization algorithms exist.

The orchestration challenge we are focusing on in this paper is providing support for this latter type of VM and thus VNF placement in an automated and transparent fashion, both for resilience patterns as well as for performance optimization. Thus, critical infrastructure components should be instantiated in a way that ensures high availability or performance, while not demanding a large degree of manual planning and intervention from the service operator.

In this context, this paper describes a practical solution and case study for placing a network function, i.e., a signaling component, in a resilient fashion in the open source cloud environment OpenStack[1]. The paper thus aims to contribute to a general and practice-oriented scheme for placing virtualized functions in infrastructures.

The contribution is twofold: we will present a mechanism to organize the infrastructure based on an adapted zone concept, allowing a flexible degree of freedom for placement algorithms. We also provide an example showing how to use this scheme with OpenStack.

The paper is organized as follows: Section II describes use-cases where infrastructure-aware VM placement is beneficial, as well as approaches from literature implementing such a placement function and concepts from existing cloud management systems related to this topic. Section III contains our proposed solution to support infrastructure-aware placement in OpenStack, while Section IV provides details about our selected use-case and the example placement function based on it. Finally, we provide a summary and conclusions of the paper in Section V.

II. Background

Three topics are of major interest in the context of this paper: resilient deployment of components, placement of virtual machines (VMs) and the zone concept as understood by cloud management systems. Therefore, in the following we provide necessary background information and explanations as well as relevant references on these points.

A. Resilient Network Function Deployment

Remus [3] is a scheme for providing high availability of any kind of virtual machine by using continuous, asynchronous replication of state changes between the active and the backup instance of a redundant instance pair. The backup is only activated if the protected instance fails, otherwise it is just a copy of the active VM state. The state replication is done using checkpointing, where the protected instance buffers output at a checkpoint until the backup has acknowledged the state synchronization, at which point it is released. In the meantime, speculative execution can continue on the protected VM.

The system discussed in [3] was configured with a dedicated direct Ethernet crossover connection or independent switches. If such a scheme should be implemented in a data center architecture, a corresponding low-delay and high-throughput connection would be one of the requirements, illustrating the need for placing the protected active and the backup instance close to each other in the infrastructure.

An extension of the approach of Remus for replication across a WAN environment is presented in [4]. The envisioned use-case is for replicated virtual machines to be placed in separate data centers for disaster recovery. As the approach is based on Remus, it also uses the transparent VM cloning and take-over mechanism that provides a general resilience pattern for any kind of VM. For checking the state of the instances and to avoid network problems between the instances to become a (incorrect) reason for failover, a separate watchdog is used as an arbiter that checks connectivity to both instances, using a quorum algorithm to decide about the failover state. Failover is realized on network layer, it is proposed to use BGP updates for this (in contrast to ARP for Remus). While the solution works over a WAN link with 1 Gbps and a latency of 5 ms, the impact the connectivity between the instances has on the performance of such a system in general is highlighted as well.

A resilient deployment of both signaling and data SIP instances using message replaying is evaluated in [5]. It increases resilience by providing redundant instances coupled with high-availability daemons and assigns the reachable IP address to the main daemon. This daemon replicates the signaling traffic to the backup daemon, which forwards it to the backup server instance, thus implementing state synchronization. Again, the requirements for the synchronization imply an intelligent placement of the two instances.

B. Placement of Network Functions

Placing network functions in a cloud infrastructure basically means placing the VMs containing the implementations of these functions. Typical cloud management systems, such as Amazon AWS[2], OpenStack or VMWare[3], come with orchestration functionality that handles this placement.

However, this placement mainly focuses on finding a physical host that has the required resources to run the VM, e.g., number of cpu cores, memory or line card speed. The placement of a set of VMs in relation to each other is typically not considered in detail, although affinity and anti-affinity rules might be recognized. An affinity rule would express that VMs need to be started on the same host, while anti-affinity requires that VMs are instantiated on separate machines.

While it can be imagined that many conditions could be expressed with this kind of rules if they could refer to different levels (e.g., same or different host, rack, fault domain, ...), generating an expression for a precise placement of a large set of VMs would be complex and difficult to debug and audit.

An example for the necessity to provide a fine-grained and tightly controlled placement of instances to physical machines is provided by [6]. Here, an algorithm for the placement of redundant VMs for a given application is proposed. It assigns individual VMs to specific physical hosts in order to minimize the number of hosts needed to guarantee a certain level of redundancy. For this type of algorithm, standard affinity and anti-affinity rules do not suffice.

Another example that more importantly also shows the need for taking into account the placement of sets of VMs in relation to each other is given in [7]. Here, a placement strategy is presented that aims at improving traffic flows in a data center by taking into account traffic rates between VMs when placing them. Based on real data center traffic traces, it identifies the optimization potential offered by placing high-traffic VM pairs close together, and bases its VM and host clustering algorithm on this observation. The basic idea is to instantiate a cluster of VMs with high intra-cluster traffic on the same host cluster with low intra-cluster route costs. Again, to deploy

[1]http://www.openstack.org/

[2]http://aws.amazon.com/

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

such a placement algorithm a fine-grained addressing of hosts is necessary.

As a continuation of the work of [7], [8] describes a placement algorithm that takes an application description, complete with availability and connection constraints, and a datacenter description and translates these into tree models. The algorithm then places VMs on physical machines by first grouping VMs and placing groups on server clusters, then checking if the individual VM demands, such as memory or cpu requirements, are met.

The first part is termed Structural Aware Planner (SAP), while the second is the Demand Aware Planner (DAP). This is an important distinction, since typically, cloud infrastructures provide solutions for the latter (i.e., selecting adequate physical hosts for the requirements of an instance), but not for the former (i.e., placing related groups of instances according to their interconnection requirements). In order to avoid introducing new abbreviations without need, we will in the following use SAP for (Infra-)Structural Aware Planning and DAP for Demand Aware Planning as well to distinguish where necessary which kind of placement planning we refer to.

The algorithm of [8] provides a good example for a placement algorithm that is using the full range of individual as well as subgroups of hosts from which to draw on for its final result, but at the price of tightly integrating both of these steps. Also, it finally either needs to do the actual deployment itself or still needs to communicate its result to a cloud management system, such as OpenStack.

In [9], a deployment function is described that accepts as input an abstract service description including placement and resilience requirements. Here, the delay between redundant NF instances is noted to be of particular interest, since state synchronization may take place between them.

The aim of the deployment function is to generate a cloud deployment in OpenStack, i.e., generating a Heat template to instantiate a described service. In addition, the use of availability zones is proposed to allow for the placement of VNFs. However, it does not cover the establishment or practical use of these zones in order to allow for a flexible positioning of the network functions. This issue is addressed in the work presented here.

The described use-case of [9], a SIP PBX example, is the one we base our own example scenario on. It shows that the informed placement of VM instances in relation to each other does not only play a role for traffic optimization, as is the focus in [7] and [8], but also for placing redundant instances of components with the goal of increasing resilience.

Thus, we aim to make the placement process more flexible and modular by providing a real-world solution for designating parts of the infrastructure in which to place instances, and using this designation to communicate between the different optimization steps. The same designation can be used after any step for the actual commands of the cloud infrastructure framework, i.e., OpenStack. This enables an easy replacement or development of the different algorithms.

We utilize this placement mechanism in combination with a resilience pattern mapped to OpenStack in order to provide a first step towards an automatic deployment of resilient components in this cloud infrastructure framework.

C. Zone concepts

The concept of zones and groups of resources appears in different cloud management and orchestration systems. In the following, we will describe how it is interpreted in these environments to clarify the terms used and how our concept relates to existing ones.

1) Amazon AWS: In Amazon's cloud service Amazon Web Services (AWS), a user can select to start their VM instances in *regions*, and, within a region, in different *availability zones*. Regions are actual geographical regions, e.g., U.S. West Coast or Europe, and are completely isolated from each other. Availability zones within a region are also isolated, but interconnected via low-latency links according to Amazon. The recommendation is to use different availability zones if redundant instances are to be started to provide fault isolation.

Both of these constructs are pre-configured and defined by Amazon, and are offered via the AWS API to the cloud user. In addition, a user can define *placement groups* to launch their instances in, with all machines in one placement group being on the same, low-latency 10 Gbps network. Thus, placement groups cannot span across availability zones. This means that using placement groups can be helpful by trying to avoid network performance bottlenecks, but is of little use for resilience. In addition, the same caveat applies as to basic affinity rules, i.e., using placement groups can only yield a rather coarse-grained placement.

2) OpenStack: In OpenStack, four different related concepts are defined: *cells*, *regions*, *availability zones* and *host aggregates*. Cells and regions are both used to differentiate between different sites of a cloud deployment. Cells share the same API endpoint from a client perspective, while regions have different API endpoints and do not cooperate. More relevant to our discussion are the availability zones and host aggregates. Both are configured by the cloud provider before a deployment takes place.

An availability zone in OpenStack is a logically separated group of resources in the infrastructure, with the primary purpose of being used for isolation of components and assuring redundancy. An example would be having an availability zone for each independent power supply domain. Following this logic, the general idea is that any given physical machine can only be part of one availability zone at a time (although details of this seem to be under discussion still). The selection of an availability zone to start an instance in is explicit, and thus the configured zones are to be made visible to a user.

In contrast, a host aggregate serves the purpose of grouping resources that have some attribute in common, e.g., amount of memory or number of cpu cores. It allows a cloud user to start instances on machines that have desired attributes. This happens through defining a number of key-value pairs for the flavors of the user's instances which are then matched to the defined attributes of the host aggregates. Thus, the machine selection is implicit from the viewpoint of the users, since the configured host aggregates themselves are not visible to them. A host can be part of more than one host aggregate, since it may match more than one grouping criterion. This makes host aggregates a natural candidate for our approach, as we will show in the next section.

III. Zone Tree Mechanism

We showed in Section II that a range of placement algorithms exists, with output 'resolution' ranging from placing specific VMs on individual hosts as well as placing groups of VMs within a cluster of physical machines. Especially this latter case offers a degree of freedom for subsequent placement steps. This is particularly important if the placement algorithm does not meticulously make sure that a selected host actually has the required resources available.

Thus, the basic issue we aim to solve in the context of a cloud platform such as OpenStack is to be able to flexibly assign virtual machines to a host or a group of hosts with a placement function, while keeping this aforementioned degree of freedom as much as possible for the subsequent steps of the orchestration.

To give an example, if the placement function determines that two VMs should be started in separate racks (but within the same fault domain), then the cloud orchestrator should have the freedom to choose specific physical hosts within these racks. These two steps correspond to the SAP and DAP components described in [8].

This issue could also be solved by merging all the placement and orchestration steps and optimizing within this combined algorithm, as done in [8]. However, for the sake of modularisation of different optimization algorithms, it would be helpful to be able to flexibly denote subgroups of machines and to exchange these common denominations between the different parts of the orchestrator and between the orchestrator and the cloud infrastructure manager, e.g., OpenStack Nova. Here, we focus on a scheme to exchange and re-use placement information between different functions, and not on how this information is used or how conflicts between different placement algorithms should be resolved.

A natural candidate for such a scheme is to organize the infrastructure logically in the form of a zone tree. The root of this tree is a node containing the complete infrastructure, and its leaves are the physical hosts. The intermediate levels denote different degrees of aggregation, e.g., racks, fault domains, data centers, etc. The actual tree for any given installation thus depends on the topology of the physical infrastructure. A small example of such a tree is shown in Figure 1.

To map this tree structure and the flexible addressing of leaves and subtrees it offers to orchestration, we propose to use the host aggregate concept of OpenStack. It allows to place hosts in more than one aggregate, which is also necessary for creating a tree structure since any leaf is part of several scoped subtrees (except for the trivial case, a tree of height $h = 1$). We implement the tree by creating a host aggregate for each of the tree nodes (using `nova aggregate-create`, we will provide the other main commands in the same fashion in the following), adding a *zone* attribute to each of these aggregates (`nova aggregate-set-metadata`).

This attribute contains as a value the scoped 'address' of this node in the tree, very similar to the scoped domain addresses

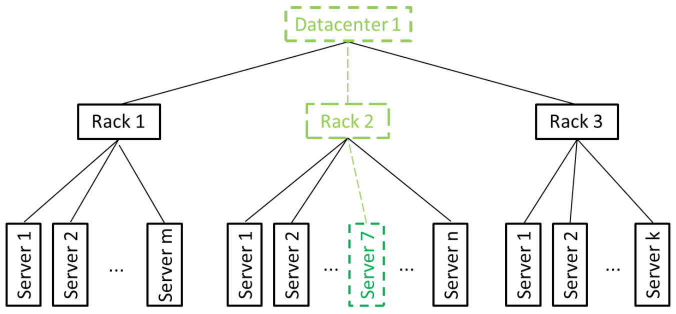

Fig. 1. Example tree with three levels: Datacenter, rack and server

known from DNS. For example, the highlighted node in the tree of Figure 1 would have the value *1.2.7*, i.e., it is physical server *7* located in datacenter *1*, rack *2* (although we use a numbered format here, each node can in principle be named using a string). Intermediate nodes would have only a prefix of such a 'complete' address, e.g., *zone=1.2* denotes the complete rack *2* in datacenter *1*.

Each Nova compute node is then added to all host aggregates representing subtrees it belongs to (`nova aggregate-add-host`). In the case of our example node, it would be added to three host aggregates, namely the three with the attributes *zone=1*, *zone=1.2* and *zone=1.2.7*, respectively.

In addition, for each instance that is to be placed using this tree structure, a number of flavors with the same new *zone* attribute containing the placement information needs to be created (`nova flavor-create` and `nova flavor-key`, respectively). For each combination of a node in the tree and pre-existing flavor, a flavor is defined that uses the same configuration (memory, number of cpu cores, disk space, ...), but that uses as a value for this zone key the node address in the tree.

Thus, the number of flavors in the OpenStack deployment is multiplied by the number of tree nodes. While this might be a potential overhead (this factor might be in the range of 10^4 for a datacenter with a number of hosts in the same range), each flavor is only generated once and basically only consists of a database entry. Therefore, we currently do not foresee this being a prohibiting performance issue.

Finally, the zone name configuration and the infrastructure topology information needs to be made available to placement algorithms. However, since this information should be relatively stable, it should be sufficient to provide this in the form of a description document, such as for example an XML-formatted document. As an additional benefit, this information together with a mapping of this naming scheme to the actual hardware should make it easier for the cloud infrastructure provider to generate an audit trail if necessary for the provisioning of critical infrastructure, without directly disclosing too much of its infrastructure details to customers.

With the configuration described above, any deployment function or simple placement algorithm acting before the actual OpenStack orchestration can flexibly deliver placement information to the OpenStack orchestration by selecting the according flavor to be started and generating the according command or Heat template. Since OpenStack maps the flavor to a host aggregate with the according value for the *zone* key, the instance will be started in the subtree defined by the value of that attribute. If the subtree is not a leaf, OpenStack will retain the degree of freedom to choose one of the nodes contained in it according to its own optimization criteria.

This illustrates already how an additional placement algorithm can communicate its requirements to the standard OpenStack orchestration. However, the underlying zone tree concept allows for a more general use. It can be easily imagined how different placement and orchestration steps can exchange placement information, e.g., also the OpenStack orchestration could be exchanged for a different mechanism and still reuse this information. However, in the described form no changes to the OpenStack code are necessary, all steps described above are simply a specific configuration of an OpenStack deployment.

IV. Example Placement Algorithm

In order to show the feasibility of our modular placement approach, we implemented a simple placement function that covers the SAP part described in Section II, and tested it in a small testbed with an OpenStack Icehouse deployment.

The considered specific use-case is to place a redundant active-backup pair of VMs, i.e., a placement for a simple resilience pattern such as described in [3], [5] or [9]. The requirement stemming from this use-case is to aim to place pairs of instances close to each other with respect to end-to-end delay, but far enough apart to guarantee a certain target level of availability. The corresponding DAP functionality is then to be provided by OpenStack, i.e., the standard *nova-scheduler*.

The algorithm used for the placement function is a heuristic that takes as an input the availability of the individual components of the physical infrastructure, as well as the delay between them. We assume that this kind of information is available from the infrastructure provider, which is in line with the either implicit or explicit assumption made by all placement optimization algorithms.

A second input is the required maximum delay between the redundant instances and the minimum availability of the joint component. We envision this to come from the SLA and service description of the service operator.

The infrastructure itself is modeled as a tree, similar to the approach from [8], and based also on the tree-based configuration of the OpenStack deployment as described in the last section. For each of the interior nodes of the tree, i.e., the roots of the subtrees, the maximum delay between two leaves in the according subtree is calculated, as well as the minimum availability. This basically constitutes a worst-case assumption for the placement of the instance pair within this subtree.

Then, a filter is applied to identify subtrees in which the requirements for the instance pair are met. The largest of these subtrees is selected in order to provide the OpenStack nova-scheduler the maximum degree of freedom. An example is shown in Figure 2, for a required maximum delay between instances of $10\,\mathrm{ms}$ and a minimum availability of $a = 0.99$. The output of the algorithm that is communicated to the cloud management system as parameters of a Heat template are two of the children of this maximal subtree, one for placing each instance (which are generally roots of subtrees themselves). This output basically takes the format of the specific flavors for the selected zones to be inserted in the template, such as described in the previous subsection. The rest of the template is currently planned to be pre-formatted with all the other necessary instructions for the component deployment, just defining the flavors of the instances as parameters.

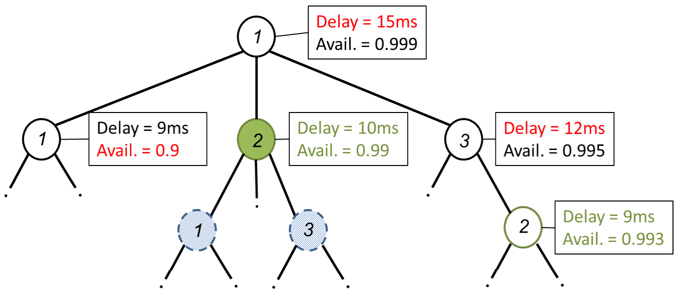

Fig. 2. Illustration of the placement algorithm. Both nodes 1.2 and 1.3.2 comply with the delay and availability requirements, but since 1.2 offers a higher degree of freedom, it is preferred and two of its children are returned (hatched nodes, 1.2.1 and 1.2.3).

This example algorithm can of course still lead to a case where instances are ordered to be started on hosts or group of hosts where the demanded resources are simply not available, despite our efforts to increase the set of candidate hosts offered to OpenStack for its scheduler. In this case, the cloud manager would reply negatively to the provisioning request, necessitating the algorithm to select a different set of host groups. This loop, which also appears in [8], would then need to be continued until either a valid host selection can be made, or if no solution can be found.

V. Conclusions and Future Work

In this paper, we discussed the topic of VM placement and its relevance in the context of NFV deployment. We showed that existing approaches and implementations for infrastructure-aware instance placement all point to the necessity of being able to communicate this placement to an actual cloud management system. We consequentially described a mechanism how to achieve this for the open source system OpenStack, and provided a small example of a placement function and how it can use this mechanism.

We consider this work just as a first stepping stone for a more complete framework for NFV provisioning. Apart from specialized and sophisticated placement functions, a support for more automated deployment of resilience patterns in cloud management systems is another building block to be developed.

Acknowledgment

The research presented in this paper has been funded by the European Commission in the context of the Research Framework Program Seven (FP7) project SECCRIT (Grant Agreement No. 312758).

References

[1] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," Open Networking Foundation, White paper, April 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf

[2] ETSI Industry Specification Group (ISG) Network Functions Virtualisation (NFV), "ETSI GS NFV-REL 001: Network Functions Virtualisation (NFV); Resiliency Requirements," ETSI ISG NFV, Tech. Rep., January 2015. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/NFV-REL/001_099/001/01.01.01_60/gs_NFV-REL001v010101p.pdf

[3] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: High availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 161–174. [Online]. Available: https://www.usenix.org/legacy/event/nsdi08/tech/full_papers/cully/cully.pdf

[4] S. Rajagopalan, B. Cully, R. O'Connor, and A. Warfield, "SecondSite: Disaster Tolerance As a Service," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: ACM, 2012, pp. 97–108. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151039

[5] G. Kambourakis, D. Geneiatakis, S. Gritzalis, A. Dagiouklas, C. Lambrinoudakis, S. Ehlert, and J. Fiedler, "High availability for SIP: Solutions and real-time measurement performance evaluation," *International Journal of Disaster Recovery and Business Continuity*, vol. 1, no. 1, February 2010. [Online]. Available: http://www.sersc.org/journals/IJDRBC/vol1_no1_2010/2.pdf

[6] F. Machida, M. Kawato, and Y. Maeno, "Redundant virtual machine placement for fault-tolerant consolidated server clusters," in *IEEE Network Operations and Management Symposium (NOMS)*. Osaka, Japan: IEEE, April 2010, pp. 32–39, iSBN 978-1-4244-5366-5. [Online]. Available: http://csc.csudh.edu/btang/seminar/papers/05488431.pdf

[7] X. Meng, V. Pappas, and L. Zhang, "Improving the scalability of data center networks with traffic-aware virtual machine placement," in *Proceedings of the 29th Conference on Information Communications (INFOCOM'10)*. Piscataway, NJ, USA: IEEE Press, 2010, pp. 1154–1162. [Online]. Available: http://dl.acm.org/citation.cfm?id=1833515.1833690

[8] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, and I. Whalley, "Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement," in *2011 IEEE International Conference on Services Computing (SCC)*. Washington, DC, U.S.A.: IEEE, July 2011.

[9] R. Bless, M. Schöller, A. Ripke, and M. Stiemerling, "Resilient deployment of virtual network functions," in *5th International Workshop on Reliable Networks Design and Modeling*, RNDM. RNDM, 09 2013. [Online]. Available: https://www.seccrit.eu/upload/Bless-RNDM-2013.pdf