Micro-service Based Network Management for Distributed Applications

Rafael de Jesus Martins, Rodolfo B. Hecht, Ederson Ribas Machado, Jéferson Campos Nobre, Juliano Araujo Wickboldt, and Lisandro Zambenedetti Granville

Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, RS, Brazil
{rjmartins, jcnobre, jwickboldt, granville}@inf.ufrgs.br
{rodolfo.hecht, ederson.machado}@ufrgs.br

Abstract. Computer networks and their services have become increasingly dynamic with the introduction of concepts such as Network Functions Virtualization (NFV) and cloud computing. To understand and configure such a complex network to their best interests, users must use several management tools that they are not necessarily familiar with. In this paper, we present an architecture that provides network management for distributed applications as easy-to-use micro-services. Our solution is based on container virtualization technologies to offer, to the user, the maximum benefit through minimal cost. We present a proof-of-concept for our architecture through a use case. Our results show that acceptable overhead is added when deploying a solution for a distributed application, and negligible overhead is added by the management tools when the user application is under heavy stress.

Keywords: Network Management · Micro-services

1 Introduction

Conventional computer networks are relatively static in terms of physical structure with respect to network topology, functionality, and protocols. These networks extensively employ physical middleboxes to perform key network functions, such as routing, firewalling, and load balancing. The administration of such networks relies on network management solutions, based for instance on SNMP, NETconf, and Netflow, that are typically implemented through distributed architectures, following the static structure of the underlying managed network.

The adoption of plentiful middleboxes in a network increases the overall number of devices to be managed, and because middleboxes are implemented with proprietary hardware, the inclusion of new network functions, including their proper configuration and maintenance, often requires a manual, thus costly, effort from the network operator. To address these middleboxes limitations, Network Functions Virtualization (NFV) is an emerging technology that relies on virtualization to implement and deploy Virtual Network Functions (VNFs) [5]. By decoupling the proprietary hardware from the associated software, NFV

enables functions to be run on top of commodity hardware, reducing operational costs and increasing network dynamicity and scalability. To achieve that, NFV is often realized with Virtual Machines (VMs), or, recently, with emerging lightweight virtualization technologies based on containers [3]. When compared to VMs, VNFs materialized through container virtualization can be deployed faster and more efficiently [6]. Containers can create and replicate customized environments, offering isolation for running applications. Because of its enhanced performance, Docker [13] has been largely adopted in industry and academia. Nevertheless, the capacity to deploy, manage, and orchestrate NFV container-based application in network environments that are heterogeneous and dynamic remains an open challenge [15].

The dynamic nature of future networks and the ephemeral function virtualization that follows along present new challenges and opportunities for network management [7]. Likewise, the ever-growing infrastructures based on the cloud-fog-edge paradigm is inherently dynamic with respect to hosted services [11]. Moreover, the distributed nature of the cloud paradigm can be leveraged by distributed applications [4]. Cloud applications can enjoy this synergy by being designed through a micro-service paradigm, in which the application is decomposed in smaller interconnected functions [1]. As of now, the burden to deploy, configure, and monitor network management tools for any new application is on its owner, usually, and is not a light one to carry out. Picking the right management tools for each application and guaranteeing their correct functioning throughout scaling events, for example, can become a more difficult task than providing the application itself. Service providers, tenants, and end-users of cloud computing could therefore benefit from the automation of these management tasks.

In this paper, we present an architecture designed to provide network management for distributed applications as micro-services. In our architecture, a tenant or customer can pick network management tools for the network infrastructure serving one or more applications of interest when deploying those applications, and the desired tools are then deployed and configured automatically, transparently to the user. By using container virtualization to deploy the desired management tools, minimal overhead is added to the operation. To assess the feasibility of our proposal, we present an use case for an implementation of our architecture. Our results show that the deployment of a solution with the user application and the necessary network management tools adds an acceptable overhead when compared to the deployment of the user application by itself. A performance analysis of the user application under stress also indicates that negligible overhead is added by the management modules, thus being a valuable tool to understand and manage distributed applications when it is most needed.

The remainder of the paper is organized as follows. In Section II, we present the background and related work. Then, we describe the proposed architecture in Section III. In Section IV, we explain how network management architectures are mirrored in our solution templates. Then, we present a use case and discuss the results regarding the architecture's implementation in Section V. Finally, in Section VII, we present our conclusions and future work.

2 Background and Related Work

The present article proposes micro-service based network management for distributed applications. Thus, it is necessary to present background on micro-service and container as well as some particularities of the chosen implementation. Besides that, we discuss some related works.

Container is a set of one or more processes organized separately from the system. All files required for the execution of such processes are provided by a separate image. In practice, containers are portable and consistent throughout the migration between development, testing and production environments. The containers are light and start very quickly [14]. **Docker** is an example of open platform for lightweight container virtualization platform which exploit improvements in kernel-level namespace support in Linux. These namespaces provide isolation between the host and the container as well as among different containers. Docker is aided by a set of tools and workflows which can be used by developers to deploy and manage containers [8].

Micro-service is an architectural style largely based on decoupled autonomous services that can be developed, deployed and operated independently of each other. Micro-services lead to various challenges in relation to team organization, development practices and infrastructure [12]. In this context, the container architecture proves to be a feasible implementation of micro-services. This tendency to use micro-services architecture has been shown to be allied to the structure of containers in at least 5 essential reasons [1]: to reduce complexity using small services, to scale, remove and deploy parts of the system easily, to improve the flexibility of using different structures and tools, to increase overall scalability, and to improve the resilience of the system.

Ciuffoletti [2] proposed the automated specification and implementation of a monitoring infrastructure in a container-based distributed system. In this work, a simple monitoring infrastructure model was defined to provide an interface between the user and the cloud management system. This model defined a monitoring infrastructure, comprising multiple instances of two basic components, one for measurement and one for data distribution. A proof of concept demonstration was described through the Docker hub, and consisted of two multi-threaded Java applications that implement the two basic components. The reference architecture of the monitoring subsystem is composed of two entities: one that manages data, one type of proxy, another that produces data.

Jaramillo et al. [8] presented a case study to discuss how Docker can effectively help leverage the micro-services architecture with an actual working model. Our work differs from the above by meeting the challenge of observability, *i.e.*, microservices architecture needs a way to visualize the health status of all services in the system to quickly locate and respond to any problem that occurs. The architecture we present, deploy containers with network management tools associated with micro-services available in another set of containers where the applications are installed. Thus, this scheme can monitor, register and manage the network of containers of the main applications avoiding their failures, or even tracking the reason for failures through their records.

Jha *et al.* [9] carried a study on the performance evaluation of Docker containers that perform a heterogeneous set of micro-services at the same time. This study concludes that running multiple micro-services within a container is also a viable deployment option, as it gives comparable (sometimes better) performance than the baseline, except for running multiple similar types of micro-services.

Lv *et al.* [10] proposed a machine learning-based container scheduling strategy for micro-services architecture to adjust the number of containers accurately and quickly in real time, specially when the service load suddenly fluctuates. Data obtained from experiments are used to train a random forest regression model for the prediction of the required service containers in the next time window. By adjusting the number of containers to balance the load pressure of the services, the proposed algorithm saves significant time comparing to traditional algorithms as well as other machine learning algorithms.

3 Network Management as a Micro-service

The aforementioned popularization of cloud computing services and similar distributed computing architectures has enabled the growth of distributed applications. Applications and macro-services can be modularized in lower-level independent services, which are themselves interconnected in a way to provide the application with high-level functionality. For example, the service for a Wordpress Web page can be split in a service running a Web server application and another service running the required database; in another example, a critical service can be replicated among several geo-distributed computing nodes, adding load-balancing and fault-tolerance capabilities to the service with ease. Clients of cloud computing services can then leverage the scalability and robustness offered by the cloud infrastructure transparently, while maintaining their focus on the application itself.

Troubleshooting distributed applications malfunctioning, however, is rarely an easy task. In the previous Wordpress example, a slow response time from the server could be due to a problem in the web server, in the database, or in the communication between them. In this case, the application owner may be well-equipped to monitor how each service is running, but understanding the communication between them could require the installation and configuration of additional network management tools. Similarly, other network management features desired by the application owners, such as security in the form of a firewall or a deep packet inspector, would have to be deployed and configured on top of the application by its owner.

In order to aid application owners with little expertise in the network management discipline, we introduce an architecture that offers the deployment and configuration of such management tools as a micro-service for the user. Our proposal is that application owners should only specify the network management features desired, in addition to the application itself, and would have the low-level work performed by our architecture. Additionally, in order to comply with containerization and micro-services paradigms, management tools should be de-

ployed by the platform only when required, and isolation between application containers and management containers should be maintained whenever possible. With these requirements in mind, we design an architecture that is able to provide network management as a micro-service for distributed application owners. Our architecture and its functioning are presented in Figure 1.

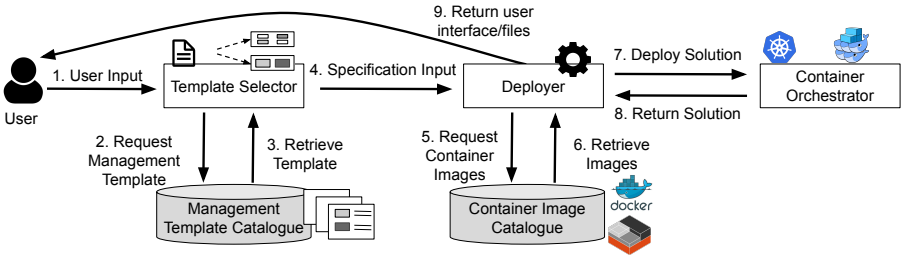


Fig. 1: Proposed architecture, including modules and their interaction.

The **User** of our architecture is the owner of a distributed application who wants to add one or more network management features effortlessly. Their input to the system is composed of a relationship of their regular application and the management features they expect for each service. The user input is passed through the Template Selector module.

The **Template Selector** is a module responsible for interpreting the user expectation regarding network management, and translating them in a relationship between the application modules and the required network management tools, which is mapped through a template. The Template Selector can be implemented to work with different abstraction levels, from low-level configuration parameters to high-level intents. In this work, we implement this module to work with low-level configuration parameters as a proof-of-concept, but the module can be expanded to include different abstraction levels with no impact in the remainder of the architecture. In our architecture, if a user's application is composed of a Web server and a database, for example, and they inform their need to monitor the latency from the Web server to their database, the Template Selector will map the request in a template that includes One-Way Ping (OWPing) and OWAMP, client and server for determining one-way latency, to be deployed and configured alongside the Web server and the database, respectively. To achieve that, the Template Selector will query the **Management Template Catalogue** for a template that fits these requirements. If such a template is available in the catalogue, a complete deployment specification containing both the user's and the management's applications will be provided. Additionally to the management tools required, every available template is also composed of a central monitoring container; when needed, this container also offers users an interface to interact with the management features they previously asked for. The following section covers network management templates specificities in-depth.

The **Deployer** receives the deployment specification and processes it, determining how containers should be distributed, which namespaces must be shared (and by which containers), and any other network configuration needed for the solution to be deployed. When the solution is produced, the Deployer queries the **Container Image Catalogue** for the images needed. Users can provide their own application images when needed, but images for some prominent network management tools are already pre-configured in the system. When the solution is ready to be instantiated, the Deployer triggers the **Container Orchestrator** to deploy the containers. Open-source platforms such as Kubernetes and Docker Swarm are example of container orchestrators, allowing the management and orchestration of Docker Engine clusters. Any additional configuration necessary is performed by the Deployer before returning a user interface to the user.

4 Network Management Architectures as Instantiable Templates

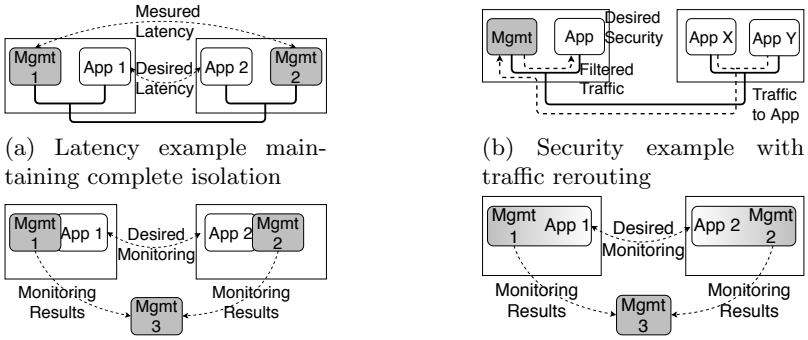
Network management is a discipline that includes, for example, network configuration, fault analysis, performance monitoring, and security assurance. The management of complex networks is thus not to be solved by a single tool but rather by the careful selection and combination of network management software. Their diverse purpose means that network management tools greatly differ with respect to their computational and architectural requirements, *e.g.*, a misplaced firewall is a useless firewall. A typical network management architecture is composed by management agents, that interact directly with managed devices to collect management information and oftentimes set configuration parameters, and a central management entity (*e.g.* an SNMP manager) that monitors and acts on the data collected by agents. Being a distributed application itself, the network management architecture can also be organized as a set of micro-services.

To fulfill user expectations regarding network management features, the appropriate set of tools must be chosen. Since requirements for management tools differ from each other, careful thought is required in their deployment. In this context, our architecture introduces *instantiable templates* for network management architectures. An instantiable template contains the information required so that management containers can be deployed alongside the user application containers, their positioning, and any other configuration needed to realize the management function correctly. Experts can develop new templates as needed, and the new templates can be fed to the architecture's template catalogue.

An important aspect that differentiate templates is with respect to container isolation. Container isolation between user application and network management tools should be maintained whenever possible. This is realized by the definition of a complete and exclusive set of namespaces for each container deployed, providing resource isolation between processes and containers running in a single system. Usually, in the micro-service paradigm, complete isolation between containers is a welcome feature. However, by carefully breaching certain isolation aspects between specific management tools and the user application they are

expected to manage, we can leverage the benefits of the micro-service paradigm while performing the network management deployment needed to realize users' desired features. In this case, two or more containers will share a subset of namespaces, allowing the network management tool to properly perform its function.

To illustrate the different template compositions, consider the following 4 common network management tasks that our system can realize (Figure 2):



(a) Latency example maintaining complete isolation

(b) Security example with traffic rerouting

(c) Monitoring example with partial namespace sharing

(d) Configuration example with full namespace sharing

Fig. 2: Set of instantiable templates, from least to most intrusive.

(a) Monitoring the latency between two containers of the user application: for latency sensitive applications, deploying instances of a tool such as OWAMP, one at each of the hosts where the application containers reside, and connecting them through the same local network as their application counterparts can be enough to provide the intended latency measures. Complete isolation between management and user application is maintained, in this situation.

(b) Securing an application through the use of a firewall: the incoming traffic must be routed through the firewall. Only the firewall position in the network is relevant to the deployment of this solution, and thus containers remain isolated. However, traffic must be rerouted and potentially modified through the new firewall function, which could affect the application performance.

(c) Monitoring all network traffic between two containers of an application: NetFlow or similar tools can be used to realize the desired monitoring. However, deploying the containerized agent tool in its own network namespace would be useless, since it would be only monitoring itself. Instead, they must be deployed in the same network namespaces (and host, therefore) of the user applications, so it can correctly perform the desired function. A collector that centralizes monitoring agents data must also be included in the template, but does not require any special isolation or positioning configuration relative to the application containers.

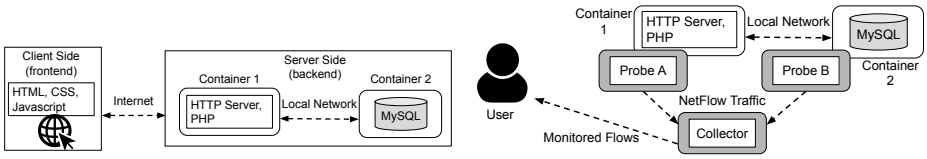
(d) Monitoring and configuring parameters of two containers of an application: this can be done through SNMP, for example, implicating a more intrusive namespace sharing between containers, since SNMP must access network, mount, and other information that would otherwise be isolated. Both application and management containers thus reside in the same set of namespaces, isolated from other systems but not from each other.

5 Use Case: Flow Monitoring for a Distributed Application

In this section, we discuss a use case for a proof-of-concept of our architecture and evaluate its results. The proposed scenario is described in Subsection 5.1 and results are discussed in Subsection 5.2.

5.1 Scenario Description

We consider the use case of a user that needs to run a simple Web application. The back-end of their application is deployed in a distributed architecture, where a Web server runs as a micro-service and responds to clients requests, and a database runs as a separate micro-service, as shown in Figure 3a. When needed, the Web server operates on the database over the network, since both services do not necessarily run on the same host. We use a simple Wordpress instance running over Apache for the Web server micro-service, and the database micro-service is deployed with MySQL. In our scenario, the user informs their desire to monitor existing network flows in both of their services.



(a) Web server and database running as separate micro-services

(b) Proposed solution. NetFlow captures system traffic for each container

Fig. 3: Proposed use case, and solution realized by our architecture.

Our architecture interprets the user's input, and identifies the need to deploy a management architecture based on NetFlow to meet the user's requirements. Based on the selected template, three containers for management will be deployed along the user application (Figure 3b). Two *fprobe* containers will each monitor one of the user's containers, and report their monitoring to a centralized collector. As per the selected template, the network namespaces for the application containers will be shared with the *fprobe* containers. The collector also

offers a processed list of network flows to the user; an experienced user can also directly interact with the management tools and logs, and perform themselves any in-depth analysis they so wish. The proposed solution is shown in Figure 3b.

5.2 Performance Evaluation

Our proof-of-concept is analyzed with respect to two performance aspects. First, we measure the time elapsed to provision of network management containers and their configuration, in comparison to having the user application being deployed by itself. Some overhead here is thus expected, albeit acceptable, since this is a one-time only cost over the life cycle of the application. Second, we must measure the network and computational overhead for having the management tools running alongside the user application. Since users are probably interested in solving bottleneck issues in their system, minimal overhead must be added by our solution for this bottleneck not to be any further narrowed.

Regarding the deployment time, the user application by itself and the complete solution have been deployed 35 times each. To assess the scalability of our solution, the experiment is extended to include an increasing number of application replicas, *i.e.*, multiple instances of the described application, which could be used by an user for fault tolerance or load balancing, for example. Figure 4 presents the time results for all cases. In this test, images for both user application and management solution are available in a local Container Image Catalogue, thus minimizing the network bandwidth effect of having the Deployer download remotely.

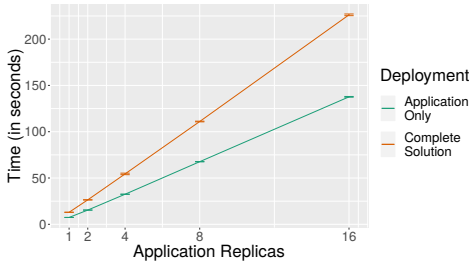
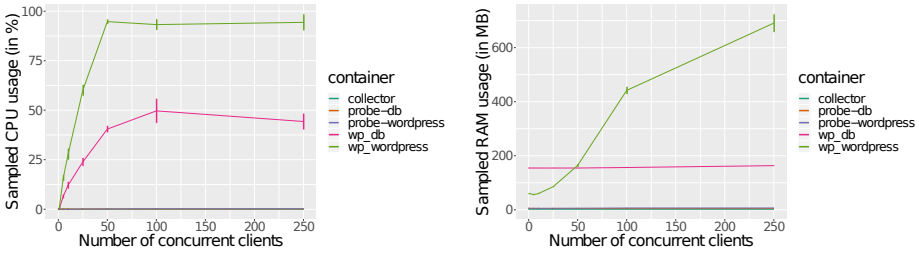


Fig. 4: Deployment time for deploying user application by itself, and with management tools included by our solution.

The results presented in Figure 4 indicate that deploying the user application by itself (*i.e.,* without additional replicas) takes on average 7.41 s to be fulfilled, whereas deploying the complete solution takes on average 12.99 s. Deploying the complete solution therefore results in a 75.3% overhead to the deployment time for a single replica, and proportionally less overhead is added when more replicas are included (64.34% for 16 replicas). The almost imperceptible error bars (for 95% confidence) also indicate the low variance observed throughout the

experiment. Although there is a noticeable overhead added, we argue that the time for deployment is a one-time cost for the user, thus offset by the benefits offered by our solution.

Another important analysis is regarding the overhead introduced for when the user application is being stressed. In this case, to assess the network and computational impact of management tools we instantiate an increasing number of clients that perform requests to the Web server. In order to evaluate how the system fares under different load levels, the number of concurrent clients increases from 0 to 250, and the interval between each client requests are chosen randomly from 0 to 2 seconds. We monitor the increase in CPU and RAM usage by all containers in our deployment, and the total traffic generated by the user application and by the management tools deployed.



(a) CPU usage for each container

(b) RAM usage for each container

Fig. 5: Computational overhead for the system under increasing stress.

Figure 5a shows the results regarding the CPU usage. It is noticeable that the CPU usage for the Web server container (wp_wordpress) explodes from the start, reaching its maximum from 50 concurrent clients onwards, and taking as much CPU resource as possible in order to process all client requests. A similar trend is observed for the database container (wp_db), albeit the maximum CPU used by it is closer to the 60% mark, and occurs at the 100 clients mark. The two results are expected, since the increase in demand for the Web server will rapidly make it consume all available resources; a fraction of these requests will also trigger some operation to the database, thus resulting in an increase for it as well. Most importantly, it is noticeable that the CPU usage by all the management containers (collector, probe-db, probe-wordpress) remains negligible and stable throughout the experiment. This result is important because it indicates that the network management solution can be useful to the application owner when they need it the most, without burdening the system performance itself.

Results with respect to the RAM usage are presented in Figure 5b. The results show for the most part a similar trend to what was observed regarding the CPU usage. RAM usage by the Web server container quickly outgrows all the others combined in order to fulfill all the clients requests. The memory used

by the database in this case is stable throughout the experiment. As with the CPU, the most important result is that the RAM usage by all the management containers is negligible and stable, regardless the number of concurrent clients.

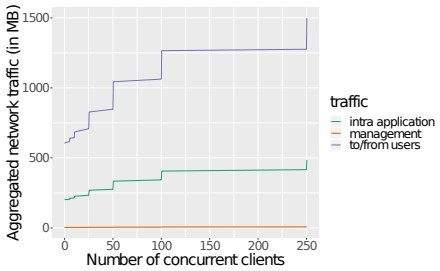


Fig. 6: Accumulated traffic between application containers, application and users, and network management tools.

Finally, we analyze the results with respect to the traffic generated in the scenario, as presented in Figure 6. Network traffic has been divided in three groups for this analysis: *intra application*, which is the traffic between the Web server and the database; *to/from users*, which is the Web server communication with the users requesting its service; and *management*, which is all traffic generated or consumed by any of the three management containers. Results are shown in terms of total traffic being exchanged by each group, and some values do not start at 0 because traffic have been exchanged prior to the start of the experiment. The increase in the traffic to/from users is the most prominent, which is a straightforward result for the increase in clients throughout the experiment. As a secondary result, Web server and database communication increases, although not as much as in the clients case. Regarding the management overhead, it is noticeable how little impact is added to the overall communication of the system, with the traffic remaining close to 0 throughout the experiment. Paired with the previous result regarding CPU and RAM, we can conclude that the system's performance (*i.e.*, the user application) is not affected by the management tools deployed even when under stress. Thus, our solution is fit to assist application owners in dealing with malfunctions of their system.

6 Conclusions and Future Work

Network management plays an important role in the success of new, dynamic network paradigms. Therefore, the need to automate management solutions and offer them as an easy-to-use service to users is pivotal. In the case of distributed applications, understanding how modules communicate over the network can be the difference between making or breaking an application, but determining the correct tools for each case requires some network knowledge one might not have.

In this paper, we presented an architecture that can easily deploy network management tools for distributed applications. Through our architecture, application owners can indicate what type of management features they expect for each module of their application, and the selection and configuration of management tools is performed automatically. Because our solution is designed using the micro-service paradigm, negligible run-time overhead is added to the user application, and a low-cost overhead in deployment time is offset by the benefits our solution offers. In future work, we expect to further enrich our architecture, with the development of the other modules, and the improvement of the ones that are already in place.

References

1. Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., Steinder, M.: Performance evaluation of microservices architectures using containers. In: IEEE International Symposium on Network Computing and Applications, pp. 27–34 (2015)
2. Ciuffoletti, A.: Automated deployment of a microservice-based monitoring infrastructure. Procedia Computer Science **68**, 163–172 (2015)
3. Cziva, R., Pezaros, D.P.: Container network functions: Bringing nfv to the network edge. IEEE Communications Magazine **55**(6), 24–31 (2017)
4. Dikaiakos, M.D., Katsaros, D., Mehra, P., Pallis, G., Vakali, A.: Cloud computing: Distributed internet computing for it and scientific research. IEEE Internet computing **13**(5), 10–13 (2009)
5. ETSI: NFV ISG White Paper #3: Network Operator Perspectives on Industry Progress. In: SDN and OpenFlow World Congress. Frankfurt, Germany (2013)
6. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and linux containers. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 171–172 (2015)
7. Han, B., Gopalakrishnan, V., Ji, L., Lee, S.: NFV: Challenges and opportunities for innovations. IEEE Communications Magazine **53**(2), 90–97 (2015)
8. Jaramillo, D., Nguyen, D.V., Smart, R.: Leveraging microservices architecture by using docker technology. In: SoutheastCon 2016, pp. 1–5. IEEE (2016)
9. Jha, D.N., Garg, S., Jayaraman, P.P., Buyya, R., Li, Z., Ranjan, R.: A holistic evaluation of docker containers for interfering microservices. In: 2018 IEEE International Conference on Services Computing (SCC), pp. 33–40. IEEE (2018)
10. Lv, J., Wei, M., Yu, Y.: A container scheduling strategy based on machine learning in microservice architecture. In: IEEE International Conference on Services Computing (SCC), pp. 65–71. IEEE (2019)
11. Mahmud, R., Kotagiri, R., Buyya, R.: Fog computing: A taxonomy, survey and future directions. In: Internet of everything, pp. 103–130. Springer (2018)
12. Mayer, B., Weinreich, R.: A dashboard for microservice monitoring and management. In: 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), pp. 66–69. IEEE (2017)
13. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux Journal **2014**(239), 2 (2014)
14. Sill, A.: The design and architecture of microservices. IEEE Cloud Computing **3**(5), 76–80 (2016)
15. Yi, B., Wang, X., Li, K., k. Das, S., Huang, M.: A comprehensive survey of network function virtualization. Computer Networks **133**, 212 – 262 (2018)