Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

A Quantitative Cross-Comparison of Container Networking Technologies for Virtualized Service Infrastructures in Local Computing Environments

Awais Aziz Shah[1,2] | Giuseppe Piro*[1,2] | Luigi Alfredo Grieco[1,2] | Gennaro Boggia[1,2]

[1]Department of Electrical and Information Engineering (DEI), Politecnico di Bari, Italy
[2]Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), Italy

Correspondence
*Giuseppe Piro, Department of Electrical and Information Engineering (DEI), Politecnico di Bari, Italy.
Email: giuseppe.piro@poliba.it

Present Address
Department of Electrical and Information Engineering (DEI), Politecnico di Bari, Italy

Summary

Container networking is emerging as a game-changer paradigm for the deployment of virtualized service infrastructures in a faster and reliable way. Nevertheless, Small and Medium Enterprises are still skeptical to revise their business in this direction because of the absence of deep studies showing its effectiveness in real deployments leveraging local computing environments. To bridge this gap, this paper presents a quantitative cross-comparison of cutting-edge technologies for container networking (including Docker as a container engine, Docker Swarm and Kubernetes as orchestrators, bare-metal and OpenStack cloud as deployment platform), properly integrated to realize a virtualized service infrastructure within a commercial workstation. Initial experimental tests are conducted to identify the most suitable combination of technologies for high-load environments, where many clients contact the virtualized service infrastructure to download files of large size. Obtained results demonstrate that the combination of Docker and Kubernetes generally ensures better performance on the bare-metal deployment platform, thus emerging as mature and effective solutions to be used by Small and Medium Enterprises. Finally, the behavior of the identified virtualized service infrastructure is also evaluated in a smart farm use case, where containers are in charge of processing images provided by mobile drones for monitoring purposes. Also, in this case, the conducted study highlights the promising capability offered by container networking in real deployments, exploiting local computing environments.

KEYWORDS:
Container Networking, Virtualized Service Infrastructures, Local Computing Environment, Experimental Analysis.

1 | INTRODUCTION

As well known, virtualization gives the opportunity to optimize the usage of hardware resources and to conceive advanced and isolated services through a common (frequently distributed) platform[1,2]. Since decades, virtualization was achieved by using Virtual Machines, that are emulator of computers having their own kernels, managed by a hypervisor system[3,4]. More recently, instead, a novel virtualization technology, namely container, is gaining momentum. Different from Virtual Machines, containers act as high-level applications running on top of the same Operating System [5–7], thus offering quick startup time, rapid application loading, less memory space requirement, a swift recovery from failure, and portability (e.g., build once and run anywhere). Furthermore, thanks to the *container networking* paradigm, containers can interact with each other, while paving the road to a new way to conceive applications with less cost and reduced time-to-market and revising the industry business [8–10].

At the time of this writing, Tech Giants (like VMWare, Xen, Microsoft, Amazon, and Google) already offer subscription-based solutions for deploying virtualized service infrastructures in the cloud [11]. On the other hand, Small and Medium Enterprises (SMEs) would like to implement their own virtualized service infrastructures into local computing environments (apart the deep control and personalization of implemented functionalities, it would erase heavy subscription fees related to the usage commercial clouds) [12,13]. Theoretically, this is possible thanks to the presence of a number of open-source technologies enabling container networking [14–41]. Nevertheless, SMEs are still skeptical about the usage because of the following two main reasons. First, the effective usage of container networking requires the selection and the joint integration of container engines (i.e, the technology that effectively implements the container), orchestrator (i.e, the technology that is responsible for managing, scheduling, and deploying individual containers for applications within the cluster, while offering load balancing and service discovery functionalities), and many other supporting tools that make possible their implementation and usability in specific platforms. Unfortunately, this task cannot be successfully and quickly achieved by SMEs with limited technical skills and revenue to spend on research and development activities [42]. Second, the scientific literature either investigated the behavior and the performance of containers against Virtual Machines or the technologies enabling container networking separately (for more details, see the summary of the state of the art discussed in Section 2). Thus, there are no contributions that address a quantitative investigation of the joint integration of containerization technologies in local computing environments, along with a clear description of the pros and cons characterizing the popular solutions available today.

To bridge this gap, this paper presents an experimental cross-comparison of cutting-edge technologies for container networking, properly integrated to realize a virtualized service infrastructure in local computing environments. Specifically, a centralized orchestrator is configured to offer service discovery and load balancing functionalities (i.e., management of clients' requests and their distribution to available containers). At the same time, some containers are deployed to expose resources and advanced services to remote clients. By considering the main outcomes of a qualitative analysis of containerization technologies presented (by the same authors of this work) in [43], the set of technologies investigated herein includes: (1) Docker as the container engine, (2) Docker Swarm and Kubernetes as orchestrators with load balancing and service discovery capabilities, (3) bare-metal and OpenStack cloud as deployment platforms and (4) Docker-compose, Docker-Machine, Kubeadm, and Flannel as supporting tools for scheduling and deployment functionalities. Due to the possible combinations between the selected orchestrator technologies (i.e., Docker Swarm and Kubernetes) and deployment platforms (i.e., bare-metal and OpenStack cloud), four different experimental testbeds have been implemented within a commercial workstation having computing capabilities that are comparable to those available in most of SMEs realities.

Initial experimental tests are conducted to identify the most suitable combination of technologies for high-load environments (i.e., when the whole system is in change of managing a high traffic load), where many clients contact the virtualized service infrastructure to download files of large size. Clients' requests are generated through the Poisson statistics from a laptop, connected to the aforementioned virtualized service infrastructures through a real-world network. Moreover, different Key Performance Indicators (KPIs), that include CPU utilization, memory footprint, network load, connection delay, and request completion time, are measured by assuming an average number of requests per unit of time equal to 5 and 10 requests/minute. Obtained results demonstrate that the integration of Docker and Kubernetes on the bare-metal deployment platform provides better performance in terms of percentage of CPU used by containers, distribution of the network load over the time and among the deployed containers, connection delay, and request completion time, while registering a slight (but still acceptable) increment of the memory footprint.

To provide further insight, the behavior of the more performant technologies is also evaluated in a more complex scenario of a smart farm use case. Differently from the previous case, a variable number of drones flying in a smart farm is now emulated on two laptops, connected to the virtualized service infrastructure by means of two different wireless access points. Drones generate livestock pictures with a Poisson statistic and deliver them to the virtualized service infrastructure. The service orchestrator forwards these pictures to available containers, which will recognize the type and the number of animals within the pictures through a machine learning-based image processing elaboration. The outcome of this processing is finally delivered to a remote server for monitoring purposes. This new campaign of experimental tests remarks that the behavior of the virtualized service infrastructure is not drastically influenced by the presence of mobile users: in any case, the service orchestrator is able to properly forward users' requests to available containers, while guaranteeing a uniform balancing of computing tasks. The execution of heavy tasks inevitably brings to higher computing and memory requirements, while reducing the overall traffic load. Nevertheless, the tests fully confirm that the combination of Docker and Kubernetes on the bare-metal deployment platform represents a suitable solution for effectively exploiting container networking capabilities in real deployments with local (hence limited) computing capabilities.

The rest of the paper is organized as follows: Section 2 presents background on container networking and reviews the state of the art. Section 3 deeply describes the technologies taken into the account in this study and illustrates the implemented experimental testbeds. The quantitative cross-comparisons, between the identified technologies for container networking are presented in Section 4. Finally, Section 5 concludes the work and draws future research activities.

2 | BACKGROUND ON CONTAINER NETWORKING

Virtualization generally refers to the implementation of an additional layer above the host Operating System, aiming at providing a separate environment for running applications with virtually allocated resources[14]. In the past, a popular hypervisor-based approach was adopted for virtualization, known as Virtual Machines. A Virtual Machine represents an emulated computer within a virtualization environment, having its own guest Operating System and kernel, and works above the host Operating System[11]. More recently, instead, containers emerged as a game-changer in the virtualization context. They can implement services and networking functionalities at the application layer of a given Operating System. Since containers emotively share the same Operating System, they generally ensure better usage of hardware resources through virtualization[14]. At the same time, containers embrace their own binaries, libraries, and runtime component, provide portability and agility (i.e., once built, they run anywhere), and introduce a new disruptive way to design and deploy future applications and services[11]. The comparison between containers and Virtual Machines were investigated in many works, including[4,6,11,15–37 44–50]. The majority of these studies compare CPU and memory usage, disk input/output, execution time, and network load of both virtualization technologies and clearly highlight that containers perform better or equal to Virtual Machines. According to the container networking paradigm, containers can interact with each other, while offering the opportunity to deploy novel applications over distributed and virtualized environments[51]. To reach this goal, however, it is necessary to integrate technologies implementing container engine, orchestrator, load balancer, and service discovery tools within a specific deployment platform.

Regarding the deployment platform, two main solutions are adopted today: bare-metal and OpenStack cloud. Bare-metal refers to the conventional physical system, where available hardware resources (e.g., compute, storage, and other resources) are managed by the computer's main Operating System. Here, all the resources are acquired by a single user[52]. During the latest years, instead, OpenStack emerged as a leading open-source solution for the setup of both small and large scale cloud operating environments[7]. It provides virtualized resources and workspace to multiple independent users[44 49]. Surely, containers can be deployed within both bare-metal and OpenStack deployment platforms[53–56]. However, the work discussed in[38,39] remark that containers deployed in OpenStack cloud load quicker as compared to bare-metal systems. However, there are no other contributions that investigate the performance of containers deployed on bare-metal and OpenStack cloud in other directions.

In the container networking context, the deployment of scalable applications over multiple nodes is achieved through an orchestrator. Specifically, the orchestrator automates and controls many tasks, such as provisioning and deployment of containers, redundancy, and availability of containers, scaling, and removing containers to spread application load evenly across host infrastructure. The role of orchestrator is very important in large and dynamic environments. Surveys conducted on this topic report that there is a need for research activities to evaluate the impact of technologies implementing orchestration functionalities in the container networking context[40,41].

The horizontal distribution of traffic across multiple containers in a cluster is carried out by a load balancer. This task is necessary to prevent containers from getting overloaded and ensure service availability. At the time of this writing,[4] is the only contribution presenting some technological details related to the orchestrator used in their test. Also in this case, however, the contribution does not discuss a comparison among different technologies offering the same functionalities.

A qualitative cross-comparison of emerging technologies for container networking (including container engines, orchestrators, load balancers, and service discovery tools) have been performed by the same authors of this work in[43]. The results of the comparison highlight that Docker is a powerful emerging container engine that works on multiple platforms, and that Kubernetes and Docker Swarm are the open-source orchestration technologies that come up with built-in scheduler, load balancer, and service discovery functionalities.

TABLE 1 Summary of the investigated state of the art.

Reference Papers	Container resource utilization	Comparison with Virtual Machine	Network performance	Orchestration	Implementation in the Cloud	Implementation on bare-metal	load balancer
4						✓	
11	✓		✓		✓		
15		✓	✓			✓	
16,18,37	✓	✓				✓	
17,19,20,26,31 – 35	✓	✓	✓			✓	
21		✓				✓	
22,36	✓			✓		✓	
23,24,57	✓					✓	
27		✓					
28,58	✓	✓					
29						✓	
30	✓	✓	✓		✓		
38					✓	✓	
39	✓	✓			✓	✓	
40				✓			
41	✓			✓	✓		
59,60	✓		✓				
61	✓			✓			
62	✓				✓		
63	✓						
This work	✓		✓	✓	✓	✓	✓

Table 1 summarizes the topics covered by the current scientific literature, including the analysis of container resource utilization, comparison with Virtual Machine, network performance, orchestration, implementation in the cloud and bare-metal, and load balancer implementation. It emerges that at the time of this writing, to the best of the Author's knowledge, there are no contributions that implement the integration of state of the art enabling technologies for container networking. For this reason, there is a need for a quantitative cross-comparison of cutting-edge technologies for container engine, and orchestrators with load balancing and service discovery features deployed on multiple deployment platforms.

3 | INTEGRATION OF CUTTING-EDGE TECHNOLOGIES ENABLING THE CONTAINER NETWORKING PARADIGM

This section presents a detailed description of the developed *virtualized service infrastructure* based on container networking. Starting from the main outcomes of the qualitative cross comparison of container networking technologies presented in[43], Docker has been selected as the reference container engine, whereas Docker Swarm and Kubernetes are considered as possible orchestrators offering load balancing and service discovery capabilities. Note that the integration of these technologies requires the usage of additional supporting tools (i.e., Virtualbox, Docker-compose, Docker-Machine, Kubeadm, and Flannel), as discussed below. The whole virtualized service infrastructure, instead, has been realized through bare-metal and OpenStack cloud deployment platforms.

Without loss of generality, the developed scenario embraces virtual machines. One of them hosts the orchestrator and a container application. The rest of the virtual machines, instead, implements container applications only. The virtualized service infrastructure is able to receive multiple requests issued through the HTTP protocol. In particular, the former request is handled by the container placed within the virtual machine hosting the orchestrator. In case of multiple requests, instead, the orchestrator activates the load balancing functionality to distribute them across the rest of the available containers.

Given the possible combinations of orchestrator technologies and deployment platforms, four different experimental testbeds have been realized:

- **Testbed 1: integration of Docker and Docker Swarm on bare-metal**. In the bare-metal deployment platform, the Docker-Machine supporting tool is used to deploy the whole virtualized service infrastructure. First of all, four virtual machines are created through Virtualbox. According to the Docker Swarm technology, one of these virtual machines has been configured as the Swarm Manager, which represents the orchestrator running service discovery and load balancing functionalities. The setup of the Swarm Manager implies the creation of the Docker Bridge and the Docker API Proxy System. The former provides a communication bus for interconnected containers. The latter defines a unified interface between the real-world network and the virtualized service infrastructure. A join-token provided by the Swarm Manager is used by other virtual machines for establishing a connection with the Swarm Manager and the Docker API Proxy system. After joining the Swarm Manager, the other three virtual machines are configured as slave nodes, namely Swarm Workers. From this moment on, Swarm Manager and Swarm Workers can also interact with each other through the Docker Bridge. At the same time, Swarm Manager and Swarm Workers can communicate with the external real-world network through the Docker API Proxy System, as depicted in Figure 1.

  On the Swarm Manager, a YAML file is used to describe the structure of the virtual environment to be deployed and the services to be executed in each container. Then, the Docker-compose supporting tool is used to launch the container application on each virtual machine.

  Now, a client can issue its request, that will be delivered to the developed virtualized service infrastructure through the real-world network. Figure 1 explains the resulting communication pattern. First, the client request is received by the Swarm Manager via the Docker API Proxy System (step 1). Then, the Swarm Manager distributes the incoming requests among available containers. The distribution follows a round-robin approach, starting from the container available within the nodes hosting the Swarm Manager. The example reported in Figure 1 shows that the request is forwarded to the container installed on the first Swarm Worker through the Docker bridge (step 2). Then, the Swarm Worker answers to the given client by sending back the requested content through the Docker API Proxy System (step 3). Finally, the Swarm Worker updates its status with the Orchestrator on the Docker Bridge (step 4).

- **Testbed 2: integration of Docker and Kubernetes on bare-metal**. This testbed still uses the bare-metal deployment platform. The virtualized service infrastructure depicted in Figure 2 highlights the presence of four virtual machines created through Virtualbox. Docker and Kubernetes packages are installed into each virtual machine.

  The Kubernetes cluster embraces two kinds of nodes: Kube Master and Kube Worker. The former one is the orchestrator, which implements service discovery and traffic load functionalities. The latter one, instead, refers to the generic node

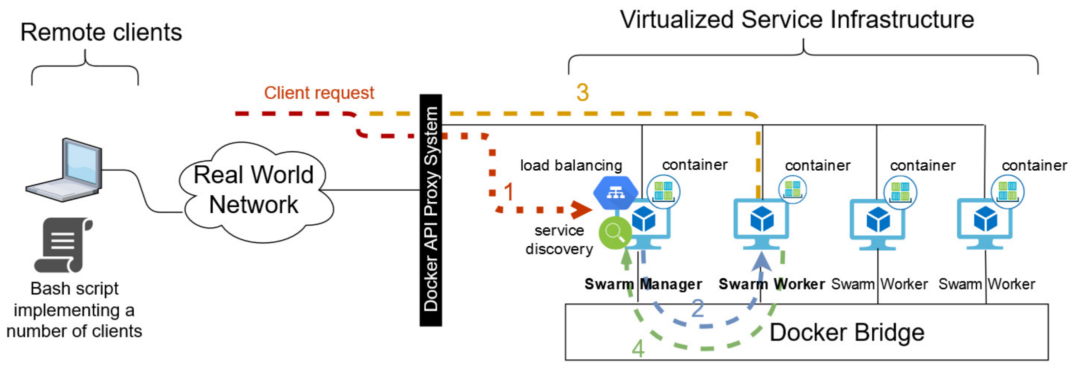


FIGURE 1 Testbed 1: integration of Docker and Docker Swarm on bare-metal.

controlled by the orchestrator and exposing a service or a resource. On the other side, the Flannel supporting tool is properly configured to create an overlay network that interconnects the nodes belonging to the Kubernetes cluster. The Kube Manager is declared and a Kube Proxy System is created, which manages the communication with the client. The join-token returned by the Kube Manager is used by the other virtual machines for establishing a connection with the orchestrator. From now onwards, the Kube Workers are connected with the Kube Master, Kube API Proxy System for communication with the client, and the Flannel overlay network for exchanging internal information.

A YAML file is used to describe the structure of the virtual environment to be deployed and the services to be executed in each container. Then, the deployment of containers is created and exposed in the cluster through Kubernetes.

Now, the virtualized service infrastructure is ready to handle the client requests. Figure 2 describes the resulting communication pattern: First, the client request is received by the Kube Master via the Kube API Proxy System (step 1). Then, the Kube Master distributes the incoming requests among available containers. The distribution exploits the round-robin approach, which starts by delivering the first client request to the container installed on the same machine of the Kube Master. The example shown here demonstrates that the request is forwarded to the container deployed on the first Kube Worker through the Flannel overlay network (step 2). Then, the Kube Worker answers to the given client by sending back the requested content through the Docker API Proxy system (step 3) Finally, the Kube Worker exchange its state with the Kube Master on the Flannel overlay network (step 4).

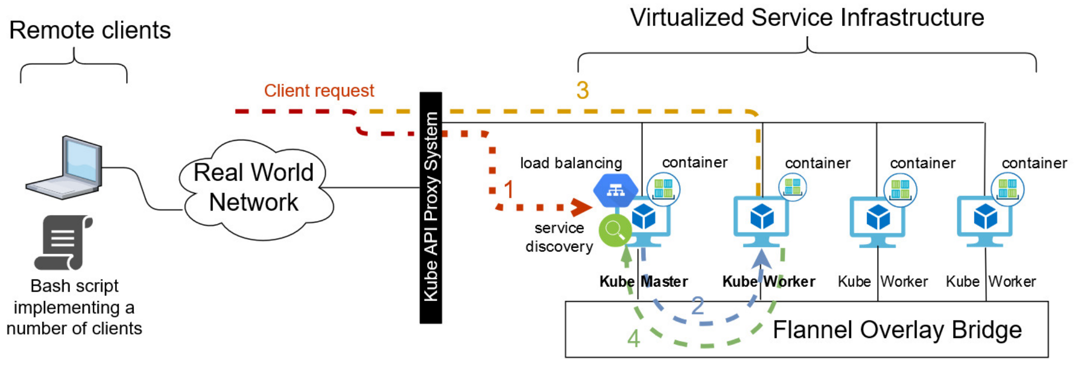


FIGURE 2 Testbed2: integration of Docker and Kubernetes on bare-metal.

- **Testbed 3: integration of Docker and Docker Swarm on OpenStack cloud**. This testbed is based on a different deployment platform, which is an OpenStack cloud. It firstly requires the configuration of key components of the OpenStack cloud, that are: 1) Nova, the primary computing engine behind OpenStack), 2) Swift, a storage system for objects and files, 3) Cinder, a block storage component that allows access to specific locations on a disk drive, 4) Neutron, the entity providing the networking capability to the overall virtualized environment, 5) Keystone, the security manager, and 6) Glance, the component providing virtual machine images. Once installed on the physical machine, OpenStack cloud can be initially managed through its graphical user interface or command-line instructions. Different from the bare-metal deployment platform, OpenStack cloud does not require the Virtualbox tool. It is able to autonomously create virtual machines, which are simply referred to as *instances*.

By default, OpenStack cloud creates a virtual network infrastructure made up of private and public networks. All the instances are installed within the private network. They have their private IP addresses and can communicate with the external real-word network through a virtual router running the Network Address Translation (NAT) protocol.

The integration of Docker and Docker Swarm into the private network of the OpenStack cloud is achieved by means of the same approach already explained for the Testbed 1. One of the OpenStack cloud instances is designed as the Swarm Manager. The setup of the Swarm Manager implies, as expected, the creation of both Docker Bridge and Docker API Proxy System which handles the client communication. The join-token returned by Swarm Manager is used by other virtual machines for establishing a connection with the Swarm Manager and the Docker API Proxy system. The other three virtual machines are configured as slave nodes, namely Swarm Workers.

From this moment on, Swarm Manager and Swarm Workers can interact with each other through the Docker Bridge and with the client through the Docker API Proxy System, within the private network created in the OpenStack cloud, as depicted in Figure 3. The Docker-compose supporting tool has been used to launch a container application in each virtual machine. Now, the client requests can access to the virtualized service infrastructure. The resulting communication process has been presented in Figure 3. The Swarm Manager receives the client request via Docker API Proxy System (step 1). Likewise, Testbed 1, the Swarm Manager distributes the incoming requests to the available containers according to the round-robin technique. In this example, the request is forwarded to the container installed on the first Swarm Worker through the Docker bridge (step 2). Then, the Swarm Worker answers to the given client by sending back the requested content of the container through the Docker API Proxy System (step 3). Finally, the Swarm Worker updates its status with the Swarm Manager through the Docker bridge (step 4).

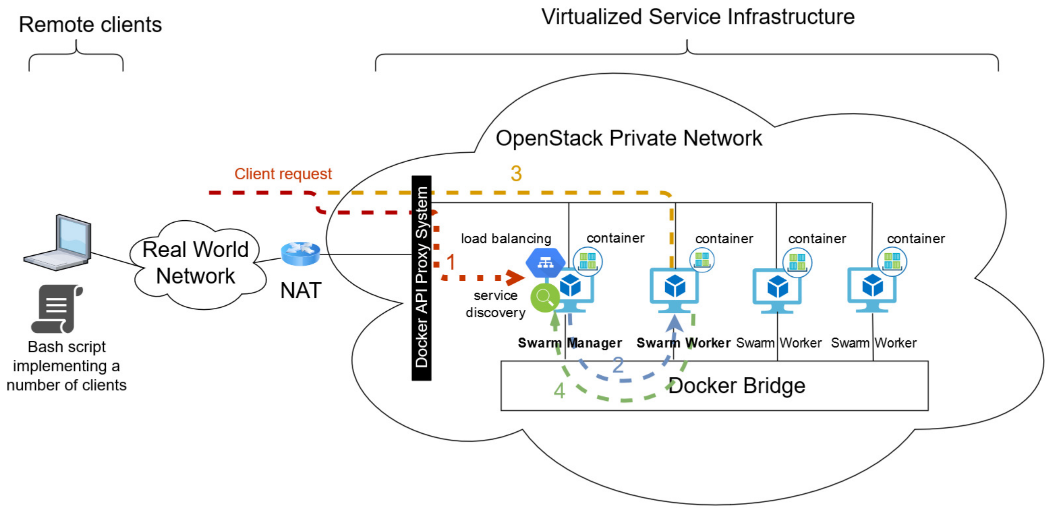


FIGURE 3 Testbed 3: integration of Docker and Docker Swarm on OpenStack cloud.

- **Testbed 4: integration of Docker and Kubernetes on OpenStack cloud**. In this final deployment, the OpenStack cloud is used to create the four instances within the private network. Then, Kube Manager and Kube Workers are configured as already discussed for the Testbed 2.

  Indeed, the Kube Manager is declared by means of the kubeadm tool. As mentioned before, a join-token be used by the other virtual machines for establishing a connection with the orchestrator, within the private network of the OpenStack cloud. Within the Kube Manager, a YAML file is used to describe the structure of the virtual environment to be deployed and the services to be executed in each container. The Flannel supporting tool is properly configured to create an overlay network that interconnects the nodes belonging to the Kubernetes cluster. Also, in this case, a virtual router is implemented with NAT rules that connect the virtualized service infrastructure with the real-world network.

Then, Kubernetes creates and exposes the deployment of the containers in the cluster. From this moment on, the Kube Manager and Kube Workers are connected to the Kube API Proxy System for communication with the client and the Flannel overlay network for exchanging information.

Now the client can make a request, that will be delivered to virtualized service infrastructure. As seen in Figure. 4, the client's request is handled in the same pattern by the Orchestrator as already discussed in Testbed 2.

Now, the client requests can access to the virtualized service infrastructure. The resulting communication process has been presented in Figure 3. The Swarm Manager receives the client request via Docker API Proxy System (step 1). Likewise, Testbed 1, the Swarm Manager distributes the incoming requests to the available containers based on the round-robin technique. In this example, the request is forwarded to the container installed on the first Swarm Worker through the Docker bridge (step 2). Then, the Swarm Worker answers to the given client by sending back the requested content of the container through the Docker API Proxy System (step 3). Finally, the Swarm Worker updates its status with the Swarm Manager through the Docker bridge (step 4).

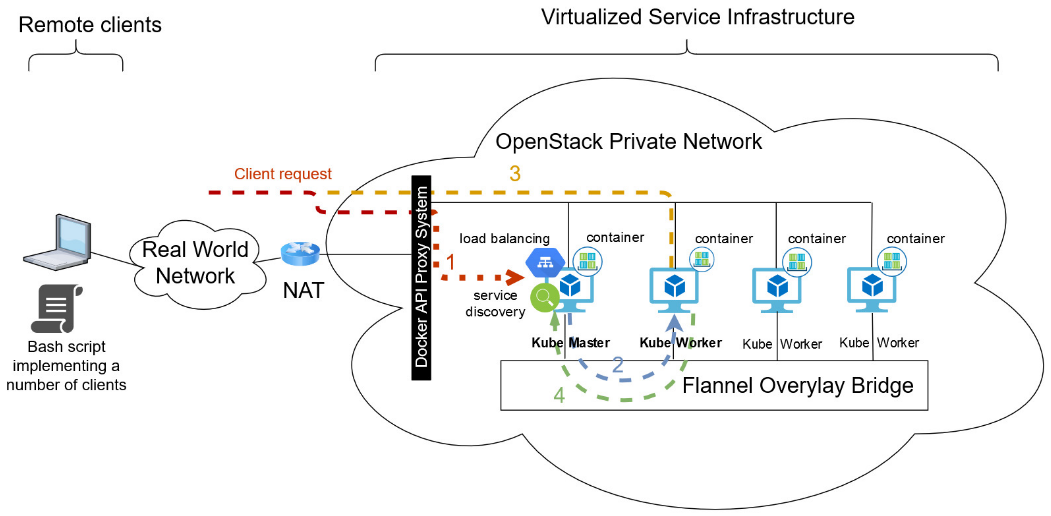


FIGURE 4 Testbed 4: integration of Docker and Kubernetes on OpenStack cloud.

4 | CROSS COMPARISON AND PERFORMANCE ASSESSMENT

This Section presents a cross-comparison and performance assessment of the reviewed and integrated cutting-edge technologies enabling the container networking paradigm. Specifically, two campaigns of experimental tests are discussed below. First, the behavior of four Testbeds described in Section 3 is investigated in a high-load environment, where many clients contact the virtualized service infrastructure to download files of large size. This study is useful to identify the most suitable combination of technologies ensuring better performance in computing environments typically available for Small and Medium Enterprises. Second, the performance of the identified virtualized service infrastructure is also evaluated in a smart farm use case, where containers are in charge of processing images provided by mobile drones for monitoring purposes. In this case, the analysis would highlight the promising capabilities offered by container networking in real deployments, exploiting local computing environments.

The computing architecture adopted in both tests is a commercial workstation with Intel® Xeon(R) E5-16200 CPU (made up of 8 cores working at 3.60 GHz each), 16 GHz RAM, and Ubuntu 18.04.2 LTS Operating System. Of course, the proposed implementation can be safely and easily extended to develop more complex scenarios, by using machines with higher computing capabilities.

Moreover, the conducted analysis considered two groups of KPIs, simply referred to as *infrastructure KPIs* and *end-user KPIs*. Infrastructure KPIs are introduced for describing the behavior of integrated technologies within the virtualization platform. They include:

- **CPU utilization**: it refers to the percentage of CPU utilized by the container. It has been monitored by running bash scripts on each virtual machine for gathering data through the *Docker Stats* command, which returns a live data stream of the containers.
- **Memory footprint**: it represents the amount of main memory consumed by the container, measured in MB. Similar to the previous case, it was collected by executing bash scripts containing on each virtual machine using the same *Docker Stats* command.
- **Network load**: it reports the average amount of data sent by the container in a unit of time. Indeed, it is expressed in terms of Mbps. It was measured by running bash scripts inside the containers for monitoring data flow on the internal bridge of the container, using the *brctl* command.

On the other hand, the end-users KPIs are defined to evaluate the quality of service experienced by the clients willing to retrieve the video file exposed by the virtualized service infrastructure. They include:

- **Connection delay**: it is the amount of time required to establish the connection between the client and the virtualized service infrastructure at the transport level. It was measured in minutes by running a bash script running at the client-side, which exploits its TCP features.
- **Request completion time**: it refers to the amount of time required to download the whole video file. It was measured in minutes by running a bash script running at the client-side.

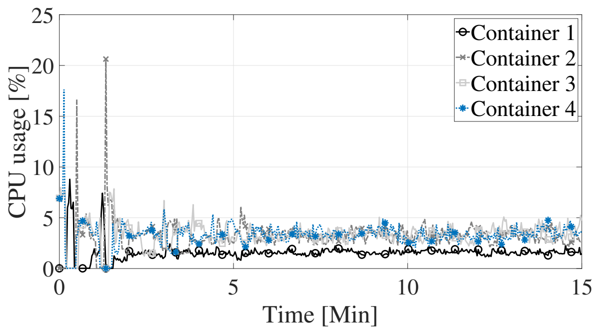
4.1 | Cross-comparison in a high-load environment

In order to compare the behavior of the four Testbeds described in Section 3 in a high-load environment (i.e., when the whole system is in change of managing a high traffic load), the virtualized service infrastructure is configured to expose resources to remote clients. In particular, to verify the right capability of the load balancing functionality to distribute incoming requests among available containers, each container was configured to host the same data content, which represents a video file of 13 minutes (i.e., a trailer taken from YouTube), encoded at an average rate of 1.45 Mbps. A laptop, connected to the virtualized service infrastructure through the real-world network, is used to emulate many remote clients willing to retrieve the aforementioned video content. In this regard, a Python script is used to generate client requests, and the communication between clients and remote containers is established through the HTTP protocol. To test the impact of the traffic load on system performance, requests are generated according to the Poisson statistics, where the average number of requests per minute, λ, is set to 5 and 10. Finally, while each test lasts 20 minutes, the results are extracted from an intermediate observation interval of 15 minutes.

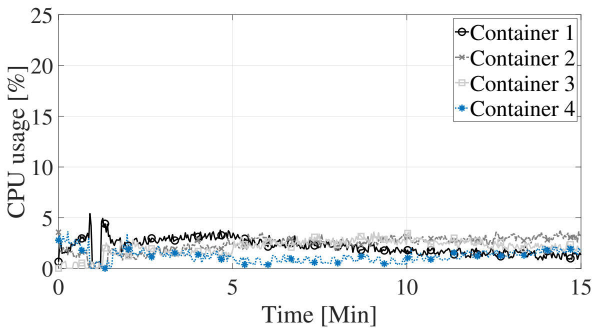
4.1.1 | CPU utilization

Figures 5 and 6 show the CPU usage of different containers deployed within the four investigated testbeds when the average number of client requests is set to 5 and 10 per minute, respectively. In all the cases, it emerges that the CPU usage registered by the available containers is almost similar during the time. This demonstrates the ability of all the selected technologies, and in particular of the load balancing functionalities implemented by the orchestrator, to offer a fair distribution of tasks within the whole virtualized service infrastructure, independently from the traffic load.

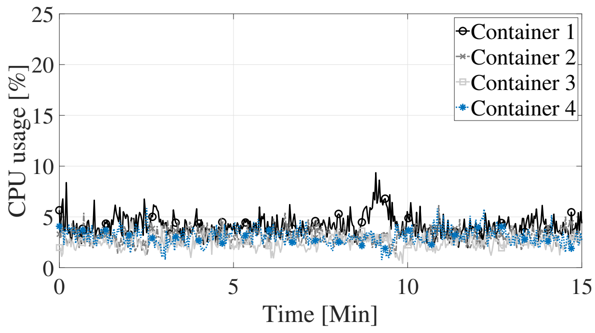
However, to better investigate the different behavior of developed testbeds, the cumulative distribution function of the CPU utilization values measured for all the containers belonging to a given testbed is reported in Figure 7. From the statistical perspective, it emerges that the integration of the Kubernetes orchestrator within the bare-metal deployment platform registers the lowest CPU utilization. On the contrary, the virtualized service infrastructure exploiting the Docker Swarm orchestrator and using the OpenStack deployment platform achieves the worst performance. In the OpenStack cloud, several components (previously mentioned in Testbed3) are involved in providing the virtual computing, storage, and networking resources to the



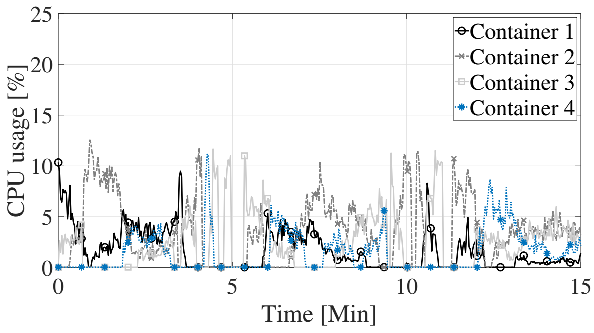
(a) Testbed 1



(b) Testbed 2

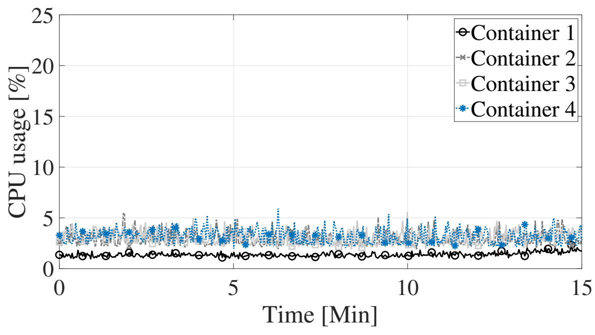


(c) Testbed 3

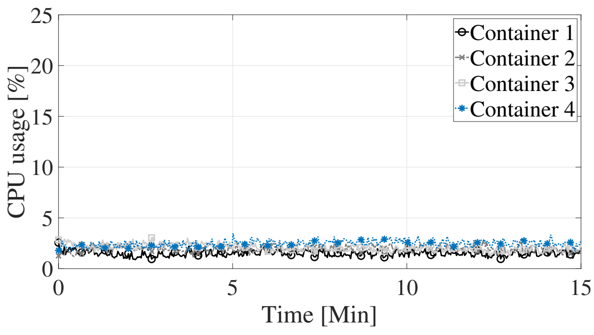


(d) Testbed 4

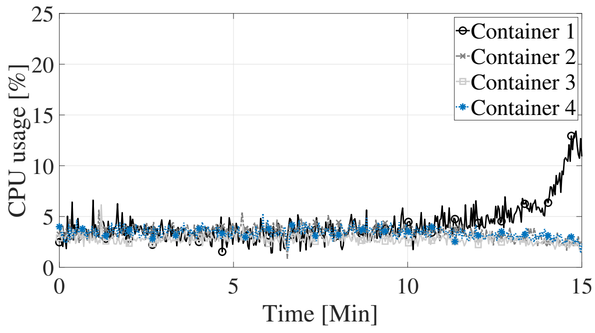
FIGURE 5 CPU utilization measured when $\lambda = 5$ requests/minutes.



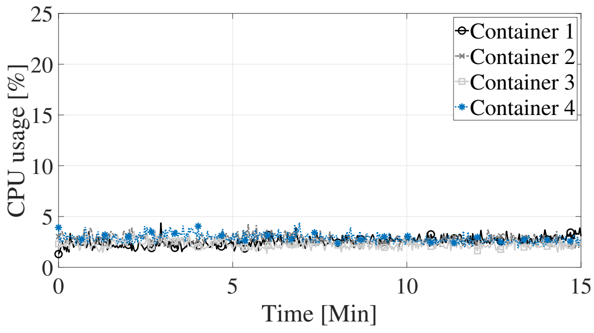
(a) Testbed 1



(b) Testbed 2

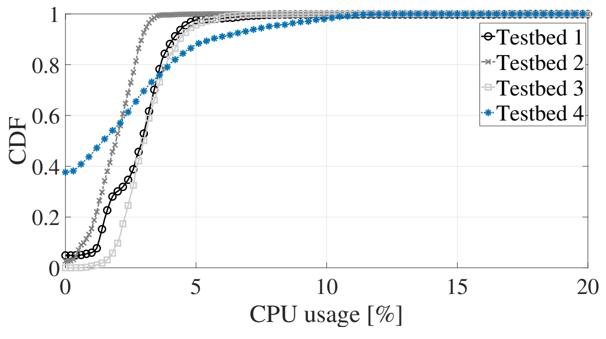


(c) Testbed 3

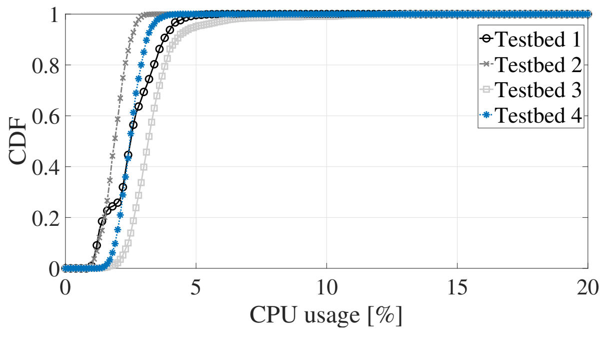


(d) Testbed 4

FIGURE 6 CPU utilization measured when $\lambda = 10$ requests/minutes.



(a) λ=5 requests/minute



(b) λ=10 requests/minute

FIGURE 7 Cumulative distribution function of CPU utilization measurements.

instances running into the considered deployment platform. This additional computational overhead justifies the higher CPU usage registered by the testbed leveraging the OpenStack cloud deployment platform.

4.1.2 | Memory footprint

Figures 8 and 9 show the amount of memory occupied by the different containers deployed within the four investigated testbeds when the average number of client requests is set to 5 and 10 requests per minute, respectively. Apart from the different memory footprint experienced by each testbed, it is very important to highlight these two considerations. First, when the bare-metal deployment platform is used, the virtual machine hosting orchestrator and container (that is the first one in the presented implementations) experience a higher memory usage. Second, the memory footprint slightly grows during the first half of the experiment, when the number of parallel requests is increasing. This behavior is more evident when the average number of requests per minute is set to 10. Here, in fact, the increment of the traffic load brings to higher memory consumption.

The cumulative distribution functions of the memory footprint values measured for each testbed are depicted in Figure 10. Results demonstrate that the lowest memory footprint is registered by the integration of Docker Swarm orchestrator into the OpenStack deployment platform. On the contrary, the adoption of Kubernetes within the bare-metal deployment platform consumes the highest amount of memory. These results reverse the considerations discussed for the CPU utilization: the testbed registering the highest CPU utilization ensures the lowest memory footprint, whereas the testbed registering the lowest CPU utilization achieves the highest memory footprint. Indeed, the containers deployed with Kubernetes on the bare-metal platform perform their tasks by utilizing more memory and less CPU than the technologies available on other testbeds.

4.1.3 | Network load

The average amount of traffic generated by each container in a unit of time, when the average number of client requests is set to 5 and 10 requests per minute, is reported in Figures 11 and 12, respectively. In the case of $\lambda = 5$ requests per minute, the deployments with Docker Swarm produces consistently low throughput. On the contrary, the deployment of Kubernetes as the orchestrator achieves high throughput, thus ensuring that the requests are generally completed in a lower amount of time.

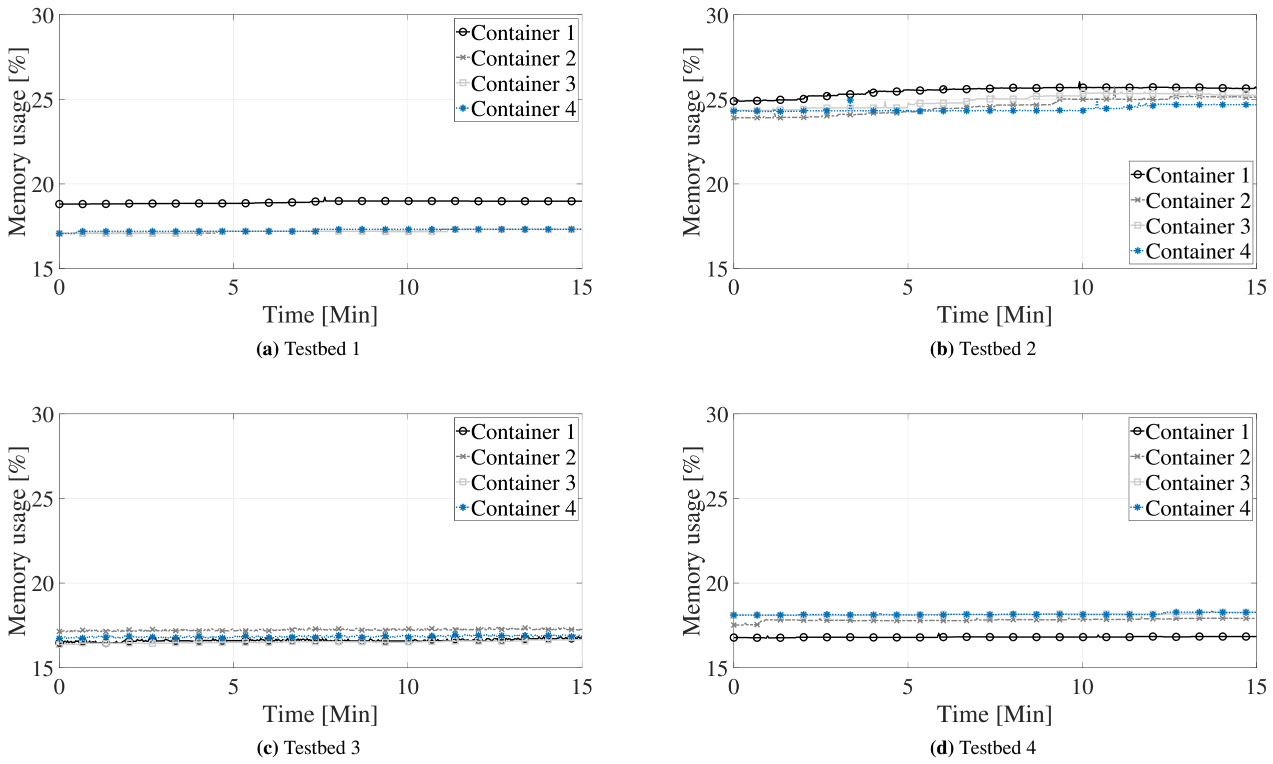


FIGURE 8 Memory footprint measured when $\lambda = 5$ requests/minutes.

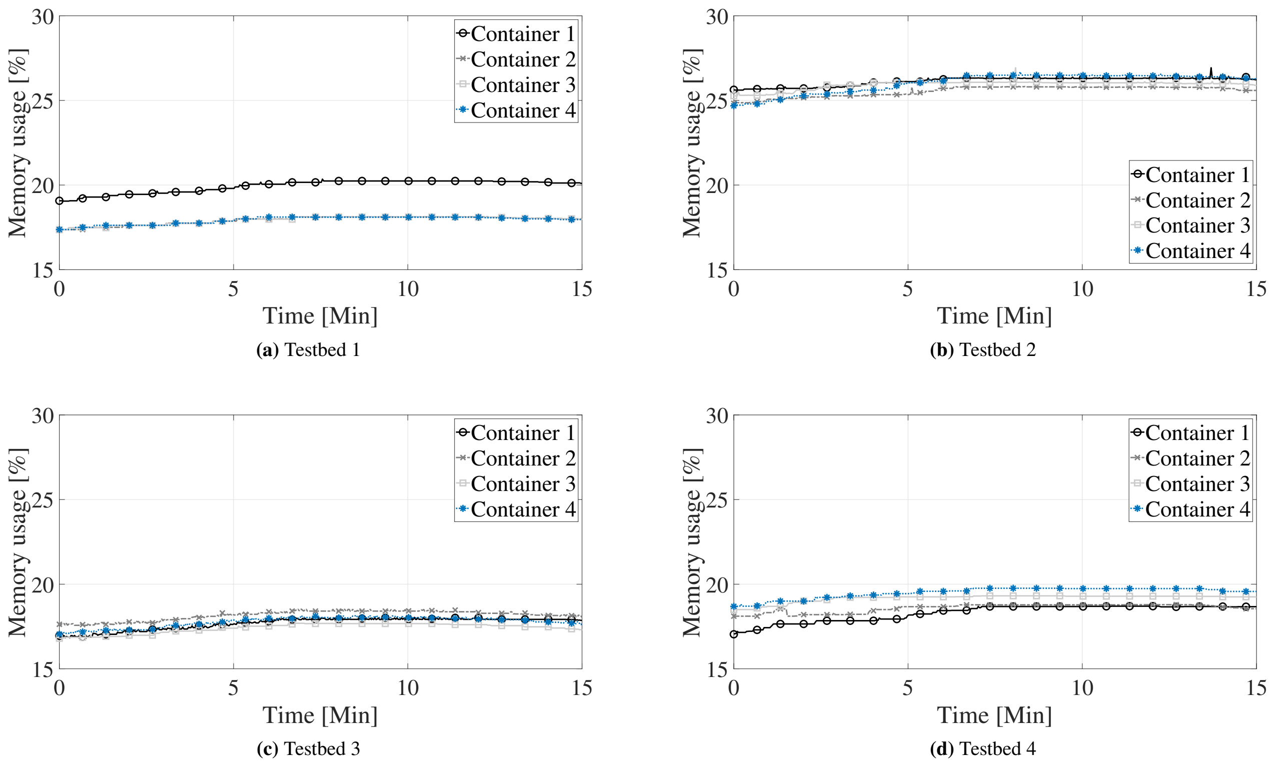
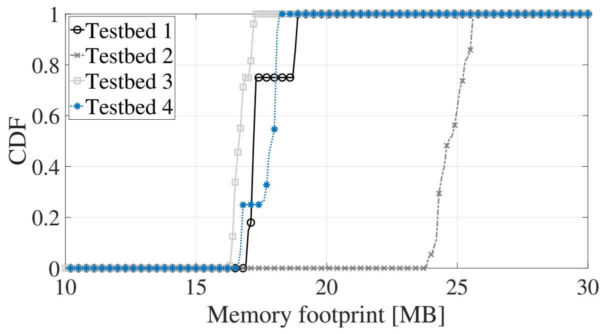
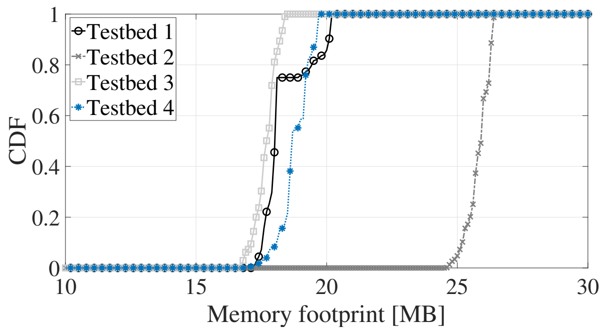


FIGURE 9 Memory footprint measured when $\lambda = 10$ requests/minutes.



(a) λ=5 requests/minute



(b) λ=10 requests/minute

FIGURE 10 Cumulative distribution function of memory footprint measurements.

The cumulative distribution functions of network load measurements are reported in Figure 13. The results show that high network throughput is achieved by the integration of Kubernetes on the bare-metal deployment platform. On the opposite side, the deployment of Docker Swarm on the bare-metal platform produced lower throughput. This behavior is more clearer when the average number of requests per minute is set to 10. As anticipated before, OpenStack builds on several components that provide the virtualized network resources to the instances inside the OpenStack cloud. It can be the reason behind the average network throughput in the case of deployments on OpenStack.

4.1.4 | Connection delay and request completion time

To conclude the cross-comparison, the KPIs introduced to evaluate the impact of developed testbeds on the quality of experience registered by remote clients are discussed below.

As the first set of results, Figures 14 and 15 show the connection delay measured when the average number of client requests is set to 5 and 10 requests per minute, respectively. As expected, different requests experience different connection delays, ranging from hundreds of milliseconds to a few seconds. Connection delays tend to increase with the traffic load, because of the increment of both traffic and tasks the virtualized service infrastructure handles. However, what clearly emerges from the reported curves is that the adoption of OpenStack as a deployment platform always provides higher connection delays. The delayed responses from OpenStack cloud is mainly due to the presence of NAT, which introduces an additional communication latency to the message exchange between clients and the virtualized service infrastructure.

More specifically, the integration of Docker Swarm within the OpenStack deployment platform registers higher connection delays. Whereas, clients connected to a virtualized service infrastructure embracing the Kubernetes orchestrator and adopting the bare-metal deployment platform generally registers lower connection delays.

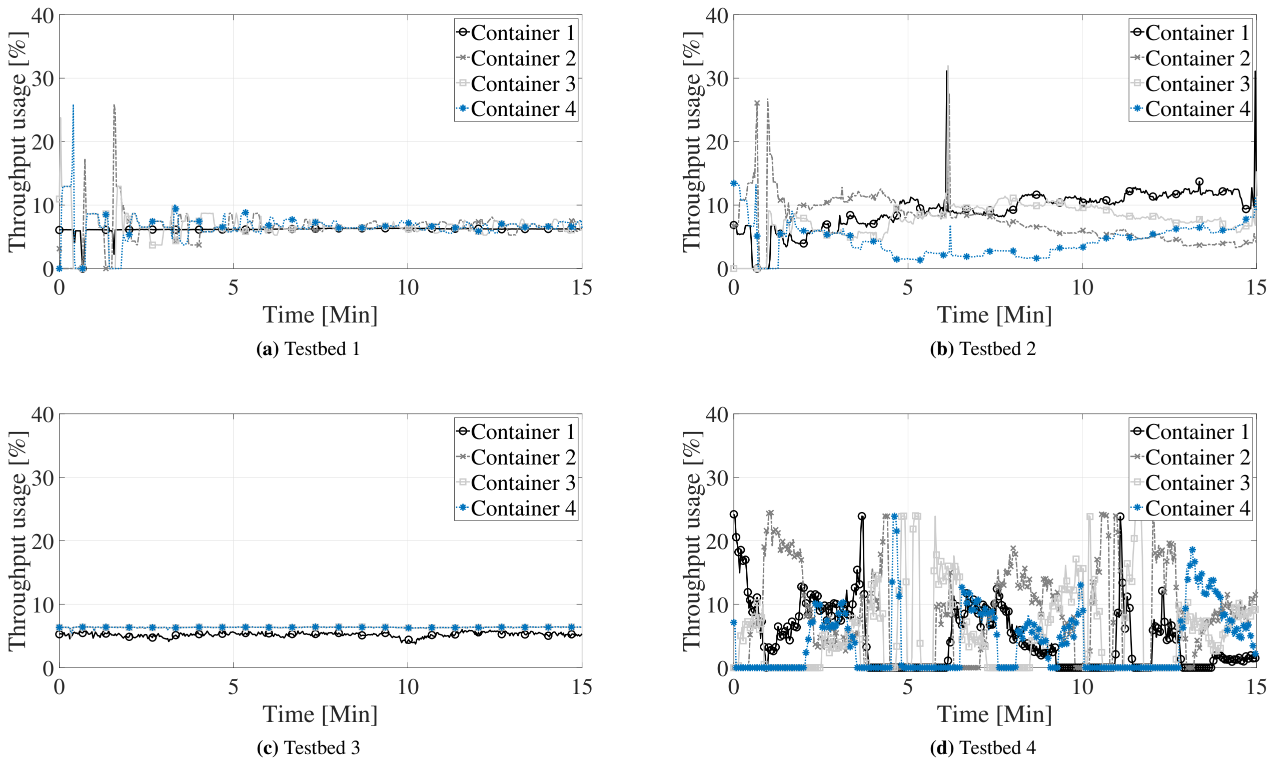


FIGURE 11 Network load measured when $\lambda = 5$ requests/minutes.

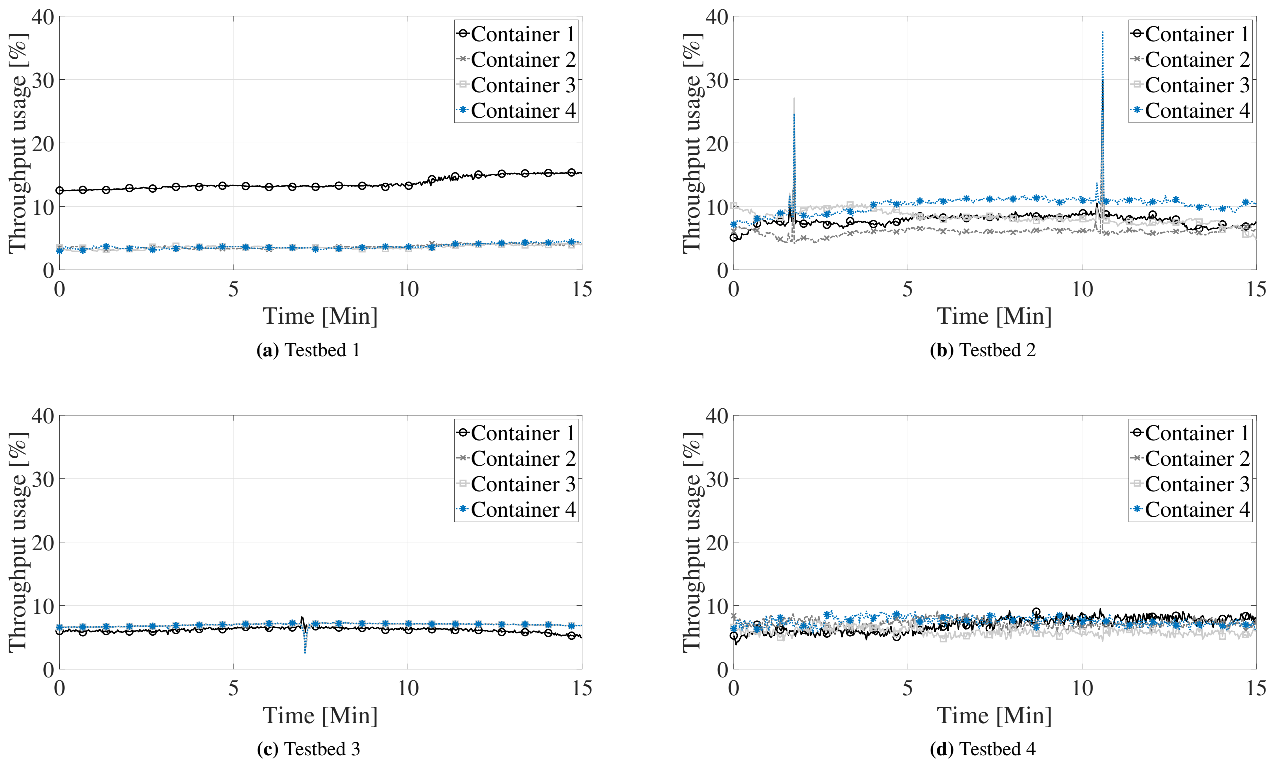
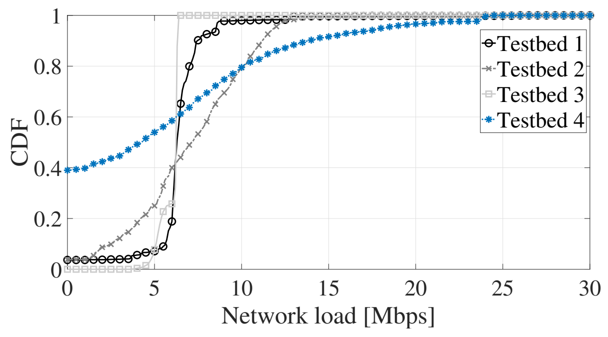
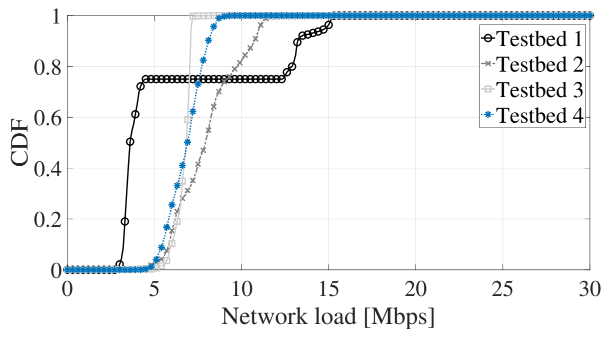


FIGURE 12 Network load measured when $\lambda = 10$ requests/minutes.

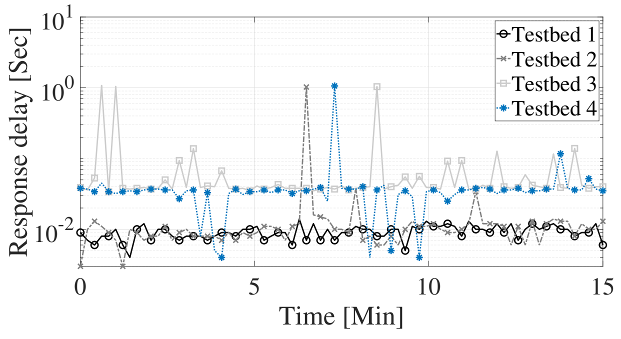


(a) λ=5 requests/minute

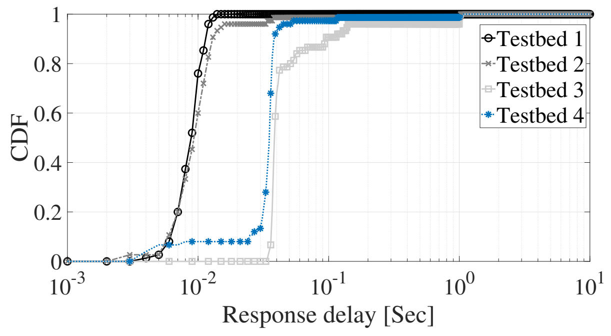


(b) λ=10 requests/minute

FIGURE 13 Cumulative distribution function of network load measurements.

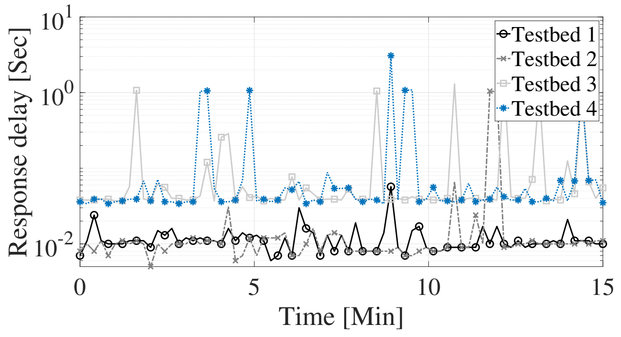


(a) Measured values during the time

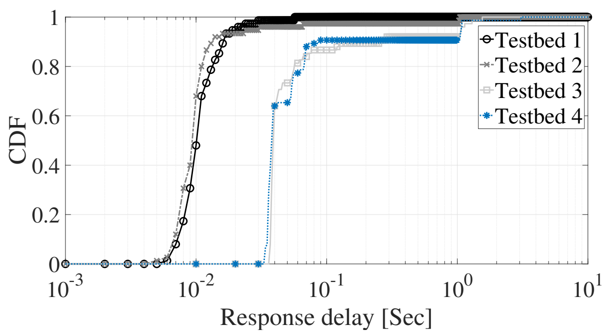


(b) Cumulative distribution function

FIGURE 14 Connection delays measured when λ = 5 requests/minutes.



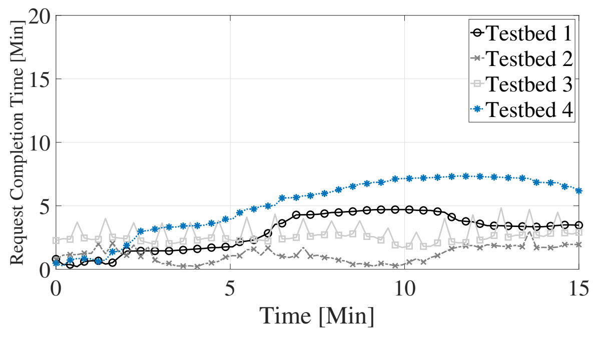
(a) Measured values during the time



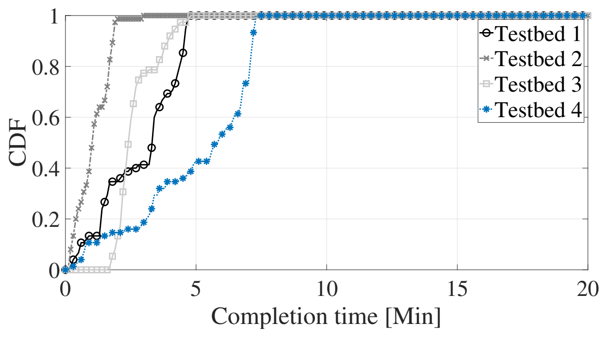
(b) Cumulative distribution function

FIGURE 15 Connection delays measured when $\lambda = 10$ requests/minutes.

The request completion time measured when the average number of client requests is set to 5 and 10 requests per minute are reported in Figures 16 and 17, respectively. In line with previous results, the request completion time tends to increase in the first half of the experimental tests, because of the increment of the number of concurrent requests. The higher the number of active requests, in fact, the higher the number of tasks executed by the components belonging to the virtualized service infrastructure. That, in turn, provokes the introduction of additional latencies. Here, the impact of the traffic load is tremendous: on the average, the request completion time registers an increment of about 10 minutes. In any case, however, the results demonstrate that the deployment with Kubernetes and bare-metal achieves better results. On the contrary, Docker Swarm within the OpenStack deployment platform registers the worst behavior.

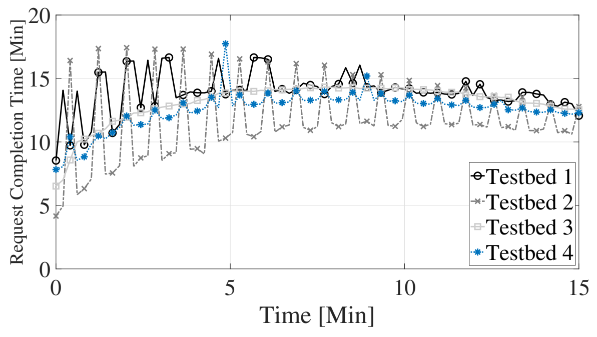


(a) Measured values during the time

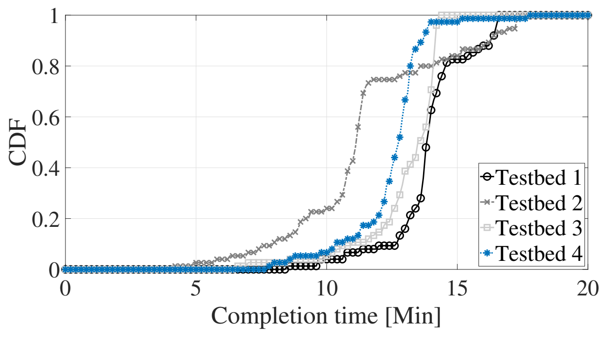


(b) Cumulative distribution function

FIGURE 16 Request completion time measured when $\lambda = 5$ requests/minutes.



(a) Measured values during the time



(b) Cumulative distribution function

FIGURE 17 Request completion time measured when $\lambda = 10$ requests/minutes.

4.1.5 | Final considerations emerging from the analysis of the high-load environment

The results obtained from the first campaign of experimental tests firstly demonstrate that the integration of cutting-edge technologies for container networking is already feasible for hardware platforms whose computing capabilities are much more limited with respect to those available in the cloud. This would encourage SMEs to use these technologies in their business, services, and applications.

At the same time, the integration of Docker engine and Kubernetes orchestrator within the bare-metal deployment platform emerges as the most suitable solution because it ensures better performance in terms of CPU usage of containers, distribution of the network load during the time and among the deployed containers, connection delays, and request completion time, while registering a slight (but still acceptable) increment of the memory footprint. On the contrary, the integration of the Docker engine and Docker Swarm orchestrator within the OpenStack deployment platform, instead, generally achieves lower performance levels.

As a final consideration, it is important to note that computing tasks significantly influences the amount of energy consumed by involved servers. In fact, the higher is the weight of computing tasks, the higher is the amount of consumed energy. Thus, the evaluation of the CPU consumption gives an idea of the amount of energy consumed by containers integrated within the developed virtualized service infrastructure. In this sense, the conducted study demonstrates that all the reviewed service orchestrators can uniformly distribute users' requests among available containers, thus guaranteeing a very good balancing of the energy consumption among the key component of the system involved in computing tasks.

4.2 | Performance assessment in a smart farm scenario

The goal of this Section is to investigate the performance of the selected combination of technologies in a more complex scenario, referring to a smart farm use case. The resulting architecture is depicted in Figure 18. It embraces many drones flying in a smart farm, able to take pictures with their on-board camera and deliver them to the remote virtualized service infrastructure for monitoring purposes. The drones are emulated through two laptops, connected to the virtualized service infrastructure by means of two different wireless access points. Note that the number of drones emulated in each laptop changes over the time and the reference scripts are properly configured to emulate the movement of drones among the two available network attachment points. This way, the proposed campaign of experimental tests is also able to evaluate the ability of the whole system to properly work also in the presence of mobile users. During the tests, each drone randomly selects a livestock image from a local storage according to the Poisson statistic. Indeed, the number of messages delivered by each drone to the remote virtualized service infrastructure in a unit of time, λ, is set to $\lambda = 10$ requests/minute and $\lambda = 20$ requests/minute. The service orchestrator forwards the received pictures to available containers, which implements an object recognition application developed by using python and TensorFlow trained models. Here, the application identifies the type and the number of animals recognized in the image and share these outcomes with another server for monitoring purposes.

In line with the previous campaign of experimental tests, also in this case, the study evaluates the impact that the execution of heavy computing tasks, generated in different network load conditions, to the usage of computing, memory, and communication capabilities of the virtualized service infrastructure. This time, however, tests last 45 minutes.

4.2.1 | CPU utilization

Figure 19 shows the CPU usage registered by the four containers deployed in the considered virtualized service infrastructure. Independently from the average number of requests generated by every single drone, it is possible to observe that the usage of the CPU follows a bursty behavior: computing tasks are highly used just during the elaboration of pictures. Contrarily from the previous campaign of tests, this time higher CPU usage is due to the complex image processing task performed by the containers. In any case, however, results fully confirm the ability of the orchestrator to uniformly distribute incoming requests among available containers. As expected, the cumulative distribution functions reported in Figure 20 highlight that the higher the number of requests generated by each drone in a unit of time, the higher is the registered CPU usage. But, in all the considered configurations, containers never glut their computing resources. This important result demonstrates, once again, how the development of a virtualized service infrastructure is feasible also in local computing environments.

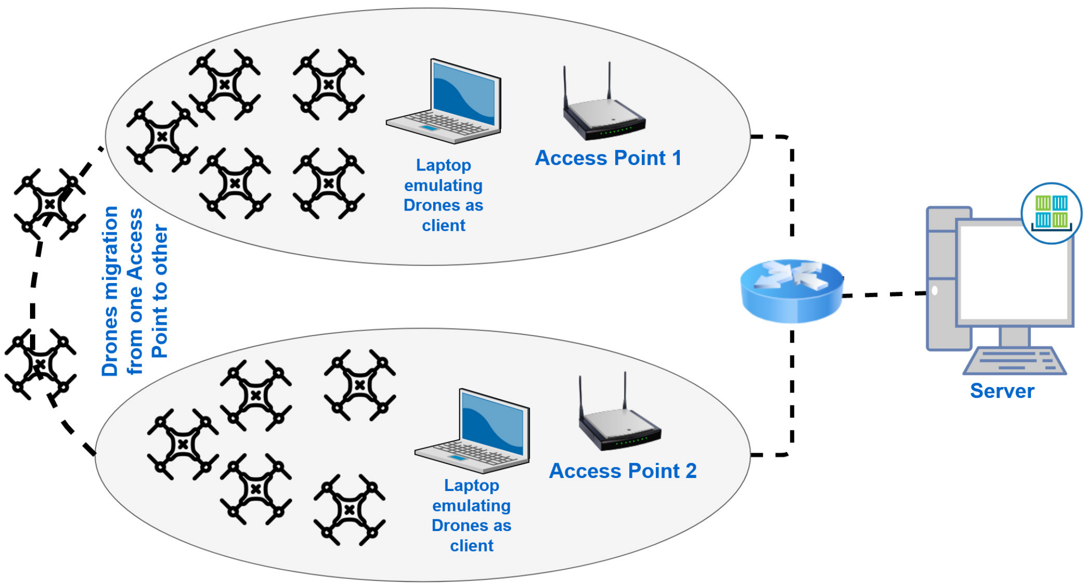

Laptop emulating Drones as client
Access Point 1
Drones migration from one Access Point to other
Server
Laptop emulating Drones as client
Access Point 2

FIGURE 18 The virtualized service infrastructure evaluated in the smart farm use case.

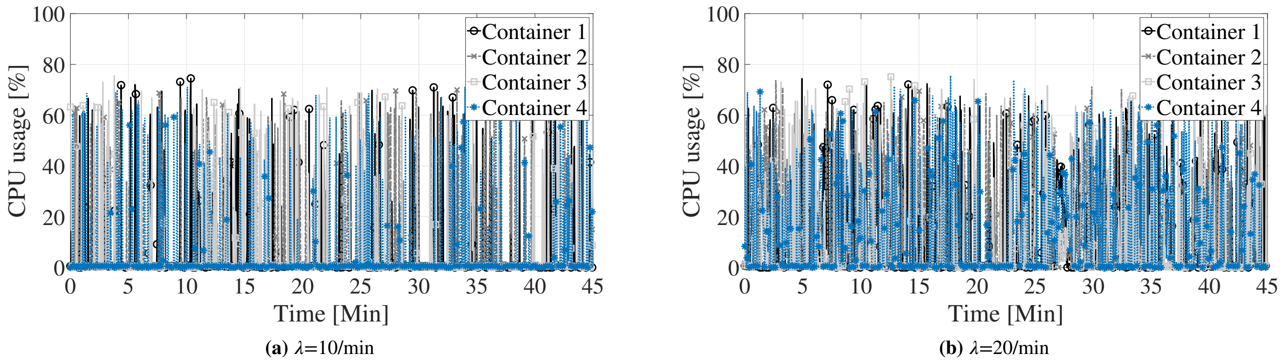


FIGURE 19 CPU usage measured during the emulation of the smart farm use case.

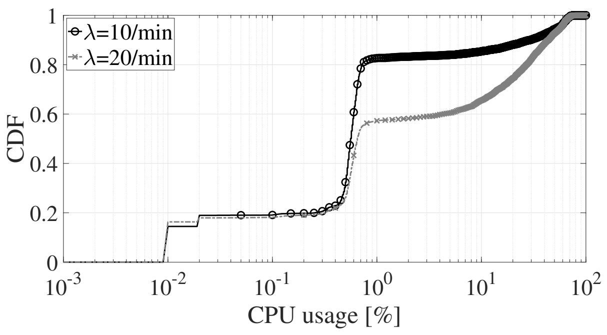


FIGURE 20 Cumulative distribution function of CPU usage measured during the emulation of the smart farm use case.

4.2.2 | Memory Footprint

The image processing tasks implemented by containers require a higher amount of memory, as reported in Figure 21. The single container occupies 1.75 GB just for storing the machine learning-based application for image processing. This is the reason, during the tests, about 2 GB of RAM is consumed by each operating container. Also for the memory footprint, the whole system ensures a good balancing among the four available containers. Moreover, since the machine learning-based application always remains loaded on the RAM, the curves reported in Figure 21 do not present a bursty behavior. Figure 22 shows the cumulative distribution function of the memory footprint measured during the emulation of the smart farm use case. Here, it is possible to observe that the higher average number of requests per unit of time introduces a slight increment of the amount of consumed memory. Containers, in fact, are involved in a higher number of tasks, thus requiring more memory.

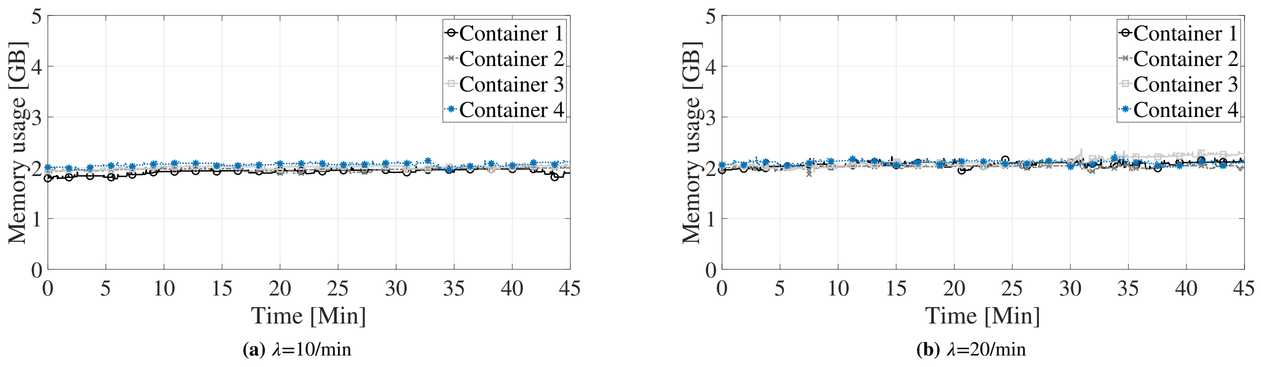


FIGURE 21 Memory footprint measured during the emulation of the smart farm use case.

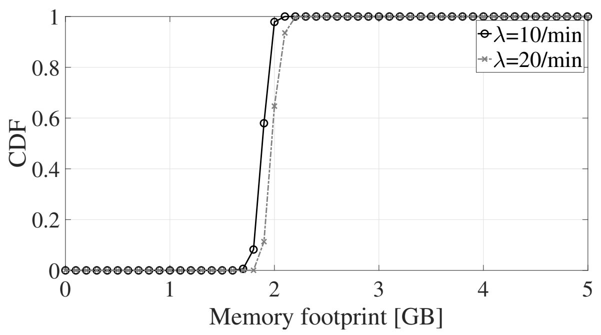
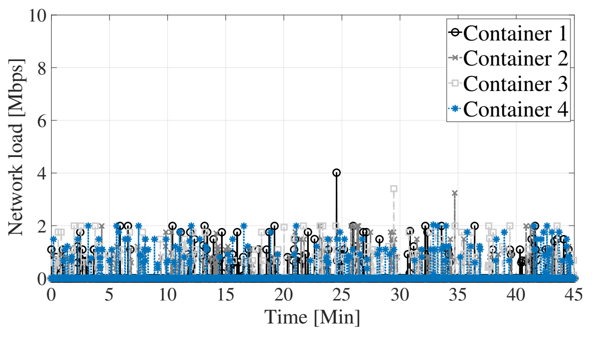


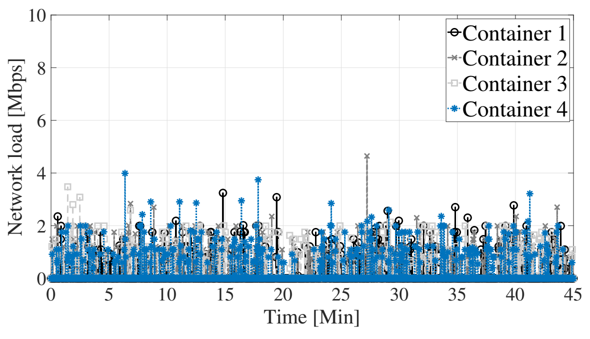
FIGURE 22 Cumulative distribution function of the memory footprint measured during the emulation of the smart farm use case.

4.2.3 | Network load

In the smart farm use case, the amount of data delivered by the network is just due to the transmission of pictures and provisioning of the resulting process. For this reason, the traffic load reported in Figure 23 presents a bursty behavior and does not achieve very high values. According to the cumulative distribution functions of the network load reported in Figure 24, it is possible to observe that the higher the number of messages generated by the drones, the higher the measured amount of traffic managed within the virtualized service infrastructure. From one hand, this result confirms what has already been presented before. From another hand, it shows how the smart farm use case requires less communication capabilities with respect to the scenario investigated in the initial campaign of experimental tests.



(a) λ=10/min



(b) λ=20/min

FIGURE 23 Network load measured during the emulation of the smart farm use case.

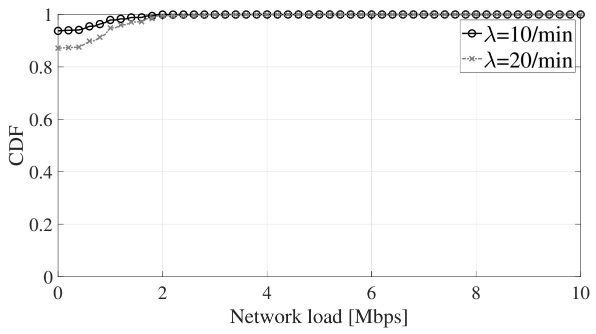


FIGURE 24 Cumulative distribution function of the network load measured during the emulation of the smart farm use case.

4.2.4 | Connection delay and request completion time

To further investigate the performance of the investigated virtualized service infrastructure in the smart farm use case, the connection delay experienced by all the drones during the service provisioning is reported in Figure 25. Once again, results demonstrate that the virtualized service infrastructure promises a proactive response to clients. The cumulative distribution function of the connection delays measured in different traffic load conditions is reported in Figure 26. As expected, when the number of requests generated by drones increases, the average connection delay increases accordingly. But, in any case, it remains below 0.1 s with a probability higher than 97%.

On the other hand, the request completion time, that in this campaign of experimental tests represents the amount of time required to complete the complex image processing task, ranges from a few second to tens of seconds, depending on the amount of task load addressed by the virtualized service infrastructure (see Figure 27). Inevitably, and in line with all the afore discussed comments, the higher the task load, the higher the task completion time. The cumulative distribution functions reported in Figure 28, however, highlight that the identification of the type and the number of animals from the picture provided by flying drones is less than 20 s with a very high probability, even in a very complex scenario. This supports a final consideration: the limited computing resources adopted in the experimental tests are enough to implement complex services with acceptable levels of quality of service.

4.2.5 | Final considerations emerging from the analysis of the smart farm scenario

The results obtained from the second campaign of experimental tests can be finally used to formulated these additional comments. First, the behavior of the virtualized service infrastructure is not drastically influenced by the users' mobility: the service orchestrator is able to properly forward the requests to the available containers, independently from the origin of the requests. Based on the previous considerations, it is possible to confirm that the combination of Docker (i.e., the container engine) and Kubernetes (i.e., the service orchestrator with load-balancing functionalities) ensures better performance on the bare-metal

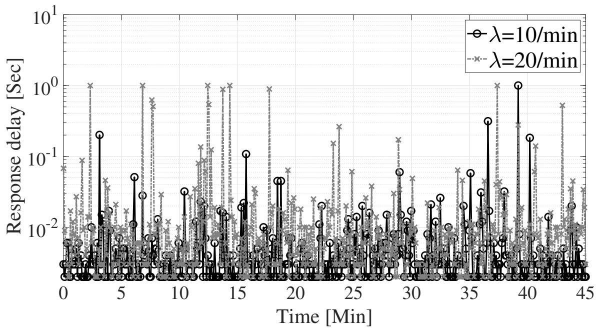


FIGURE 25 Connection delay measured during the emulation of the smart farm use case.

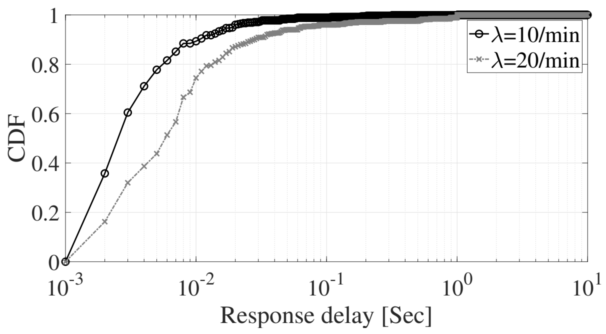


FIGURE 26 Cumulative distribution function of the conneciton delay measured during the emulation of the smart farm use case.

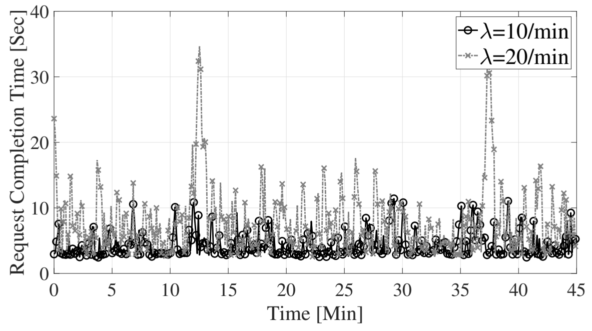


FIGURE 27 Request completion time measured during the emulation of the smart farm use case.

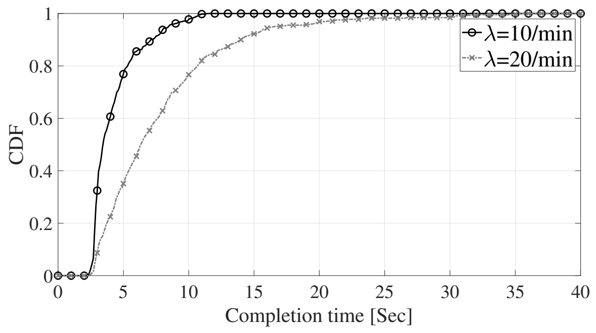


FIGURE 28 Cumulative distribution function of the request completion measured during the emulation of the smart farm use case.

deployment platform also in a more complex scenario with mobile users. Differently from the previous scenario, the smart farm use case, characterized by the execution of heavy tasks, requires higher computing and memory capabilities, as well as less communication bandwidth. Specifically, the higher CPU usage also translates into higher energy consumption (as already discussed in the previous section). But, despite what was explicitly declared from the performance assessment, the overall analysis clearly demonstrates that deployed infrastructure represents a suitable solution for effectively exploiting container networking capabilities in real deployments with local (i.e., limited) computing capabilities.

Of course, the study presented in this paper can be generalized for implementing many other real-life scenarios, such as drones' communication in a mission-critical scenario[64], as well as for smart mobility applications in smart cities (i.e., where containers can effectively perform on the server-side to ensure quick computations of data with high efficiency and very minimal delay)[65].

5 | CONCLUSION

This paper presented a quantitative cross-comparison of cutting-edge technologies for container networking, properly integrated to realize a virtualized service infrastructure in local computing environments. In particular, the set of investigated technologies are Docker as container engine, Docker Swarm and Kubernetes as orchestrators with load balancing and service discovery capabilities, bare-metal and OpenStack cloud as deployment platforms, and Docker-compose, Docker-Machine, Kubeadm, and Flannel as supporting tools for scheduling and deployment functionalities. First, the behavior of the four developed testbeds has been investigated through experimental tests in a high-load scenario, to evaluate the ability of the virtualized service infrastructure to provide answers to multiple requests received from the remote real-world network, as well as of reporting pros and cons of the considered technologies.The conducted study revealed that the integration of Docker engine and Kubernetes orchestrator within the bare-metal deployment platform ensures better performance. Then, the performance of the most suitable technologies was evaluated in a smart farm use case, integrating mobile drones and complex image processing tasks. The outcome of the evaluation demonstrated that the behavior of the virtualized service infrastructure is not drastically influenced by the users' mobility, the execution of heavy tasks generally requires higher computing and memory capabilities, while still guaranteeing acceptable levels of quality of service in real deployments with local (i.e., limited) computing capabilities. Indeed, the results of the cross comparison presented in this paper would facilitate the Small and Medium Enterprises in the selection of technologies for container networking and encourage their adoption for revising business, services, and applications. Future research activities will significantly extend the proposed study by further investigating energy and security issues in more complex and dynamic environments, also in the presence of heterogeneous and coexisting services.

6 | ACKNOWLEDGMENTS

This work was mainly supported by the Apulia Region (Italy) Research project INTENTO (36A49H6) and partially supported by the PRIN project no. 2017NS9FEY entitled "Realtime Control of 5G Wireless Networks: Taming the Complexity of Future Transmission and Computation Challenges" funded by the Italian MIUR.

References

1. Borangiu T, Trentesaux D, Thomas A, Leitão P, Barata J. Digital transformation of manufacturing through cloud services and resource virtualization. *Computers in Industry* 2019; 108: 150-162.

2. Hughes A, Awad A. Quantifying Performance Determinism in Virtualized Mixed-Criticality Systems. In: *2019 IEEE 22nd International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE; 2019: 181–184.

3. Rodríguez-Haro F, Freitag F, Navarro L, et al. A summary of virtualization techniques. *Procedia Technology* 2012; 3: 267–272.

4. Joy AM. Performance comparison between linux containers and virtual machines. In: *2015 International Conference on Advances in Computer Engineering and Applications*. IEEE; 2015: 342–346.

5. Bhimani J, Yang Z, Leeser M, Mi N. Accelerating big data applications using lightweight virtualization framework on enterprise cloud. In: *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE; 2017: 1–7.

6. Tadesse SS, Chiasserini CF, Malandrino F. Characterizing the power cost of virtualization environments. *Transactions on Emerging Telecommunications Technologies* 2018; 29(8): e3462.

7. Kominos CG, Seyvet N, Vandikas K. Bare-metal, virtual machines and containers in OpenStack. In: *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE; 2017: 36–43.

8. Pahl C, Lee B. Containers and clusters for edge cloud architectures–a technology review. In: *2015 3rd international conference on future internet of things and cloud*. IEEE; 2015: 379–386.

9. Farris I, Taleb T, Flinck H, Iera A. Providing ultra-short latency to user-centric 5G applications at the mobile network edge. *Transactions on Emerging Telecommunications Technologies* 2018; 29(4): e3169.

10. Silva Barbosa dFE, Mendonça Júnior dFF, Dias KL. A platform for cloudification of network and applications in the Internet of Vehicles. *Transactions on Emerging Telecommunications Technologies* 2020; 31(5): e3961.

11. Kozhirbayev Z, Sinnott RO. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems* 2017; 68: 175–182.

12. Andriyanto A, Doss R, Yustianto P. Adopting SOA and Microservices for Inter-enterprise Architecture in SME Communities. In: *2019 International Conference on Electrical, Electronics and Information Engineering (ICEEIE)*. 6. . IEEE; 2019: 282–287.

13. Attaran M, Woods J. Cloud computing technology: improving small business performance using the Internet. *Journal of Small Business & Entrepreneurship* 2019; 31(6): 495–519.

14. Xavier MG, Neves MV, Rossi FD, Ferreto TC, Lange T, De Rose CA. Performance evaluation of container-based virtualization for high performance computing environments. In: *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE; 2013: 233–240.

15. Bonafiglia R, Cerrato I, Ciaccia F, Nemirovsky M, Risso F. Assessing the performance of virtualization technologies for NFV: A preliminary benchmarking. In: *2015 Fourth European Workshop on Software Defined Networks*. IEEE; 2015: 67–72.

16. Adufu T, Choi J, Kim Y. Is container-based technology a winner for high performance scientific applications?. In: *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. IEEE; 2015: 507–510.

17. Anderson J, Hu H, Agarwal U, Lowery C, Li H, Apon A. Performance considerations of network functions virtualization using containers. In: *2016 International Conference on Computing, Networking and Communications (ICNC)*. IEEE; 2016: 1–7.

18. Chung MT, Quang-Hung N, Nguyen MT, Thoai N. Using docker in high performance computing applications. In: *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*. IEEE; 2016: 52–57.

19. Shirinbab S, Lundberg L, Casalicchio E. Performance Comparision between Scaling of Virtual Machines and Containers using Cassandra NoSQL Database. *Cloud Computing* 2019: 103.

20. Felter W, Ferreira A, Rajamony R, Rubio J. An updated performance comparison of virtual machines and linux containers. In: *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE; 2015: 171–172.

21. Spoiala CC, Calinciuc A, Turcu CO, Filote C. Performance comparison of a webrtc server on docker versus virtual machine. In: *2016 International Conference on Development and Application Systems (DAS)*. IEEE; 2016: 295–298.

22. Higgins J, Holmes V, Venters C. Orchestrating docker containers in the HPC environment. In: *International Conference on High Performance Computing*. Springer; 2015: 506–513.

23. Casalicchio E, Perciballi V. Measuring docker performance: What a mess!!!. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM; 2017: 11–16.

24. Preeth E, Mulerickal FJP, Paul B, Sastri Y. Evaluation of Docker containers based on hardware utilization. In: *2015 International Conference on Control Communication & Computing India (ICCC)*. IEEE; 2015: 697–700.

25. Ruan B, Huang H, Wu S, Jin H. A performance study of containers in cloud environment. In: *Asia-Pacific Services Computing Conference*. Springer; 2016: 343–356.

26. Li Z, Kihl M, Lu Q, Andersson JA. Performance overhead comparison between hypervisor and container based virtualization. In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE; 2017: 955–962.

27. Afolabi I, Taleb T, Samdanis K, Ksentini A, Flinck H. Network slicing and softwarization: A survey on principles, enabling technologies, and solutions. *IEEE Communications Surveys & Tutorials* 2018; 20(3): 2429–2453.

28. Morabito R, Kjällman J, Komu M. Hypervisors vs. lightweight virtualization: a performance comparison. In: *2015 IEEE International Conference on Cloud Engineering*. IEEE; 2015: 386–393.

29. Manco F, Lupu C, Schmidt F, et al. My VM is Lighter (and Safer) than your Container. In: *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM; 2017: 218–233.

30. Sharma P, Chaufournier L, Shenoy P, Tay Y. Containers and virtual machines at scale: A comparative study. In: *Proceedings of the 17th International Middleware Conference*. ACM; 2016: 1.

31. Acharya A, Fanguède J, Paolino M, Raho D. A Performance Benchmarking Analysis of Hypervisors Containers and Unikernels on ARMv8 and x86 CPUs. In: *2018 European Conference on Networks and Communications (EuCNC)*. IEEE; 2018: 282–9.

32. Struye J, Spinnewyn B, Spaey K, Bonjean K, Latré S. Assessing the value of containers for NFVs: A detailed network performance study. In: *2017 13th International Conference on Network and Service Management (CNSM)*. IEEE; 2017: 1–7.

33. Ramalho F, Neto A. Virtualization at the network edge: A performance comparison. In: *2016 IEEE 17th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. IEEE; 2016: 1–6.

34. Weerasinghe J, Abel F, Hagleitner C, Herkersdorf A. Disaggregated fpgas: Network performance comparison against bare-metal servers, virtual machines and linux containers. In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE; 2016: 9–17.

35. Barik RK, Lenka RK, Rao KR, Ghose D. Performance analysis of virtual machines and containers in cloud computing. In: *2016 International Conference on Computing, Communication and Automation (ICCCA)*. IEEE; 2016: 1204–1210.

36. Bhimani J, Yang J, Yang Z, et al. Understanding performance of I/O intensive containerized applications for NVMe SSDs. In: *2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*. IEEE; 2016: 1–8.

37. Seo KT, Hwang HS, Moon IY, Kwon OY, Kim BJ. Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters* 2014; 66(105-111): 2.

38. Yamato Y. OpenStack hypervisor, container and baremetal servers performance comparison. *IEICE Communications Express* 2015; 4(7): 228–232.

39. Yamato Y. Performance-aware server architecture recommendation and automatic performance verification technology on IaaS cloud. *Service Oriented Computing and Applications* 2017; 11(2): 121–135.

40. Pahl C, Brogi A, Soldani J, Jamshidi P. Cloud container technologies: a state-of-the-art review. *IEEE Transactions on Cloud Computing* 2017.

41. Truyen E, Van Landuyt D, Lagaisse B, Joosen W. Performance overhead of container orchestration frameworks for management of multi-tenant database deployments. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM; 2019: 156–159.

42. Chan CM, Teoh SY, Yeow A, Pan G. Agility in responding to disruptive digital innovation: Case study of an SME. *Information Systems Journal* 2019; 29(2): 436–455.

43. Shah AA, Piro G, Grieco LA, Boggia G. A Qualitative Cross-Comparison of Emerging Technologies for Software-Defined Systems. In: *2019 Sixth International Conference on Software Defined Systems (SDS)*. IEEE; 2019: 138–145.

44. Sefraoui O, Aissaoui M, Eleuldj M. OpenStack: toward an open-source solution for cloud computing. *International Journal of Computer Applications* 2012; 55(3): 38–42.

45. Kumar R, Gupta N, Charu S, Jain K, Jangir SK. Open source solution for cloud computing platform using OpenStack. *International Journal of Computer Science and Mobile Computing* 2014; 3(5): 89–98.

46. Wen X, Gu G, Li Q, Gao Y, Zhang X. Comparison of open-source cloud management platforms: OpenStack and OpenNebula. In: *2012 9th International Conference on Fuzzy Systems and Knowledge Discovery*. IEEE; 2012: 2457–2461.

47. Rosado T, Bernardino J. An overview of openstack architecture. In: *Proceedings of the 18th International Database Engineering & Applications Symposium*. ACM; 2014: 366–367.

48. Yamato Y, Nishizawa Y, Muroi M, Tanaka K. Development of resource management server for production IaaS services based on OpenStack. *Journal of Information Processing* 2015; 23(1): 58–66.

49. Cherrueau RA, Lebre A, Pertin D, Wuhib F, Soares JM. Edge computing resource management system: a critical building block! initiating the debate via openstack. In: *{USENIX} Workshop on Hot Topics in Edge Computing (HotEdge 18).* ; 2018.

50. Yanagawa T. Openstack-based next-generation cloud resource management. *Fujitsu Sci. Tech. J* 2015; 51(2): 62–65.

51. Alfonso dC, Calatrava A, Moltó G. Container-based virtual elastic clusters. *Journal of Systems and Software* 2017; 127: 1–11.

52. Yeoman S. How secure are bare metal servers?. *Network Security* 2019; 2019(2): 16–17.

53. Cacciatore K, Czarkowski P, Dake S, et al. Exploring opportunities: Containers and openstack. *OpenStack White Paper* 2015; 19.

54. Merlino G, Dautov R, Distefano S, Bruneo D. Enabling Workload Engineering in Edge, Fog, and Cloud Computing through OpenStack-based Middleware. *ACM Transactions on Internet Technology (TOIT)* 2019; 19(2): 28.

55. Calinciuc A, Spoiala CC, Turcu CO, Filote C. Openstack and docker: building a high-performance iaas platform for interactive social media applications. In: *2016 International Conference on Development and Application Systems (DAS)*. IEEE; 2016: 287–290.

56. Lingayat A, Singh A, Naik V, Badre RR, Gupta AK. Horizon, a Web-Based User Interface for Managing Services in OpenStack: An Introspection. In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. IEEE; 2018: 1–6.

57. Xu P, Shi S, Chu X. Performance evaluation of deep learning tools in Docker containers. In: *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE; 2017: 395–403.

58. Di Tommaso P, Palumbo E, Chatzou M, Prieto P, Heuer ML, Notredame C. The impact of Docker containers on the performance of genomic pipelines. *PeerJ* 2015; 3: e1273.

59. Morabito R. A performance evaluation of container technologies on Internet of Things devices. In: *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE; 2016: 999–1000.

60. Morabito R. Virtualization on internet of things edge devices with container technologies: a performance evaluation. *IEEE Access* 2017; 5: 8835–8850.

61. Morabito R, Farris I, Iera A, Taleb T. Evaluating performance of containerized IoT services for clustered devices at the network edge. *IEEE Internet of Things Journal* 2017; 4(4): 1019–1030.

62. Saha P, Beltre A, Uminski P, Govindaraju M. Evaluation of docker containers for scientific workloads in the cloud. In: *Proceedings of the Practice and Experience on Advanced Research Computing*. ACM; 2018: 11.

63. Bachiega NG, Souza PS, Bruschi SM, Souza dSdR. Container-Based Performance Evaluation: A Survey and Challenges. In: *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE; 2018: 398–403.

64. Al-Turjman F. A novel approach for drones positioning in mission critical applications. *Transactions on Emerging Telecommunications Technologies* 2019.

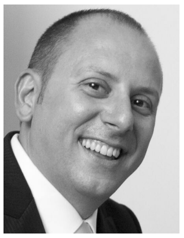
65. Al-Turjman F. Smart-city medium access for smart mobility applications in Internet of Things. *Transactions on Emerging Telecommunications Technologies* 2019: e3723.

AUTHOR BIOGRAPHY

Awais Aziz Shah received his bachelor's and master's degree in software engineering from IIUI and COMSATS University, Islamabad, Pakistan in 2009 and 2013, respectively. Currently, he is pursuing Ph.D. in Electrical and Information Engineering at Politecnico di Bari, Italy on an international student scholarship. He has served as a lecturer with the department of computer science at the University of Lahore and Superior University Lahore from 2016 to 2018. He worked as a freelance and affiliate software engineer with True-Meridian and 1ton technologies, Islamabad, Pakistan from 2008 to 2016. During his software engineering tenure, he has worked on agile software development, specifically test-driven development as a software engineer and researcher. He is a reviewer of several prestigious journals. His research interests include software-Defined Networks, network function virtualization, Internet of things, and test-driven development.

Giuseppe Piro has been an Assistant Professor in Telecommunication at Politecnico di Bari since November 2018. In March 2018, he held the habilitation as "Associate Professor" in Telecommunications Engineering, according to the National Scientific Habilitation procedure (ASN 2016-2018). He received a first level degree and a second level degree (both cum laude) in Telecommunications Engineering from "Politecnico di Bari", Italy, in 2006 and 2008, respectively. He received the Ph.D. degree in Electronic Engineering from "Politecnico di Bari", Italy, on March 2012. His main research interests include secure Internet of Things and Industry 4.0, 5G and B5G systems, data-centric and programmable architectures for the Future Internet, nano-networks, Internet models and network measurements. His research activity is documented in more than 100 peer-reviewed international journals and conference papers, accounting for more than 4200 citations and a H-index of 25 (Scholar Google). At the time of this writing, he is the local investigator of the PRIN project no. 2017NS9FEY entitled “Realtime Control of 5G Wireless Networks: Taming the Complexity of Future Transmission and Computation Challenges”. Moreover, he is involved in the European EU H2020 GUARD project and in the European Space Agency (ESA) project funded under the contract no. 4000129810/20/NL/CLP. He is also involved in Italian MIUR PON projects (Pico&Pro, FURTHER, AGREED, RAFAEL) and in Apulia Region (Italy) Research project INTENTO. He founded 5G-aisimulator, LTE-Sim, and NANO-SIM projects and is a developer of Network Simulator 3. In the past, he was involved in EU H2020 projects, like FANTASTIC-5G, BONVOYAGE, and symbIoTe, in the "Apulia Israel joint Accelerator (AIJA)" project, and in the Italian MISE project entitled "Pre-commercial trials of 5G technology using spectrum in the 3.6 GHz-3.8 GHz range" - Area Milano (bando MISE), coordinated by Vodafone. He is also regularly involved as member of the TPC of many prestigious international conferences. Currently, he serves as Associate Editor for Sensors journal (MDPI), Internet Technology Letter (Wiley) and Wireless Communications and Mobile Computing journal (Hindawi).

L. Alfredo Grieco received the Dr. Eng. degree (with honors) in electronic engineering from "Politecnico di Bari," Bari, Italy, in Oct. 1999 and the Ph.D. degree in information engineering from "Università di Lecce," Lecce, Italy, on December 2003. From Jan. 2005 to Oct. 2014, he held an Assistant Prof. position at the "DEI - Politecnico di Bari". From March to June 2009, he has been a Visiting Researcher with INRIA (Sophia Antipolis, France), working on the topic of Internet measurements. From Oct. to Nov. 2013, he has been a Visiting Researcher with LAAS-CNRS (Toulouse, France) working on Information Centric Networking design of M2M systems. From Nov. 2014 to Dec. 2018, he has been an Associate Professor in Telecommunications at Politecnico di Bari (DEI). Since Dec. 2018, he is a Full Professor in Telecommunications at Politecnico di Bari (DEI). He authored around 200 scientific papers published in venues of great renown that gained more than 10000 citations. His current research interests include: Industrial Internet of Things, Information Centric Networking, and Nano-communications. He is the Founder Editor in Chief of the Internet Technology Letters Journal (Wiley) and served as EiC of the Transactions on Emerging Telecommunications Technologies (Wiley) from mid-2016 to 2019. He also serves as associate editor the IEEE Transactions on Vehicular Technology journal (for which he has been awarded as top associate editor in 2012, 2017, and 2020). He has been constantly involved as Technical Program Committee member of many prestigious conferences. Within the Internet Engineering Task Force (Internet Research Task Force), he contributed (as author of RFC 7554) new standard protocols for industrial IoT applications (new standard architectures for tomorrow ICN-IoT systems). From Jan. 2019, he is Founding Member and Subarea-Chair of the IEEE SIG on Intelligent Internet Edge. In 2020 he has been nominated scientific coordinator of the IoT4.0 Lab.



Gennaro Boggia Boggia received, with honors, the Dr. Eng. and Ph.D. degrees in electronics engineering, both from the Politecnico di Bari, Bari, Italy, in July 1997 and March 2001, respectively. Since September 2002, he has been with the Department of Electrical and Information Engineering, Politecnico di Bari, where he is currently a Full Professor. From May 1999 to December 1999, he was a Visiting Researcher with the TILab, TelecomItalia Lab, Torino, Italy, where he was involved in the study of the core network for the evolution of Third-Generation (3G) cellular systems. In 2007, he was a Visiting Researcher at FTW, Vienna, Austria, where he was involved in activities on passive and active traffic monitoring in cellular networks. He has authored or coauthored more than 170 papers in international journals or conference proceedings, gaining more than 6800 citations. He is active in IETF and IEEE working groups; within them, he contributed to RFC 7476 and RFC 7945 in the context one Information Centric Networking. He is also regularly involved as a Member of the Technical Program Committee of many prestigious international conferences. His research interests include the fields of Wireless Networking, Cellular Communication, Internet of Things (IoT), Network Security, Security in IoT, Information Centric Networking (ICN), Protocol stacks for industrial applications, Internet measurements, and Network Performance Evaluation. Dr. Boggia is currently an Associate Editor for the ETT Wiley Journal, and the Springer Wireless Networks journal.

□