Noname manuscript No.
(will be inserted by the editor)

Performance Evaluation of an HTTP Proxy Implemented as a Virtual Network Function

Rodrigo S. V. Eiras · Rodrigo S. Couto · Marcelo G. Rubinstein

The final publication is available at Springer via
https://doi.org/10.1007/s12243-021-00886-4

Abstract Network functions virtualization (NFV) is an important approach in the telecommunications industry. One of the main features of NFV is the execution of network functions in software rather than using specific hardware. These functions can run on virtualization platforms, which can increase service elasticity and reduce infrastructure costs. However, virtualization imposes performance penalties, which can severely impact NFV services. In this work, we analyze this performance impact when the virtualized network function is an HTTP proxy. We then compare two virtualization solutions, i.e., KVM and Docker, under different configurations. Our results show that Docker containers yield performance close to that of native Linux for HTTP proxies since Docker does not employ a hypervisor. We show that KVM incurs a severe performance penalty, which a paravirtualization approach can reduce. We also evaluate how much load balancing in Docker can improve the performance of virtual proxies. We show that, for our scenario, two parallel proxies significantly improve performance. However, we observe a negative impact when increasing the number of proxies since they interfere with each other.

Keywords Network Functions Virtualization · Docker · KVM · HTTP Proxy

Rodrigo S. V. Eiras and Marcelo G. Rubinstein
Universidade do Estado do Rio de Janeiro - FEN/DETEL/PEL
E-mail: rubi@uerj.br; ORCID: 0000-0002-2509-4010
Present address of Rodrigo S. V. Eiras
IesBrazil Consulting and Services - IT Section
E-mail: rodrigo.eiras@iesbrazil.com.br; ORCID: 0000-0002-6867-9502

Rodrigo S. Couto
Universidade Federal do Rio de Janeiro - COPPE/PEE/GTA
E-mail: rodrigo@gta.ufrj.br; ORCID: 0000-0002-6921-7756

This work is based on our paper published in Portuguese in the Proceedings of the XXXVI Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC) available at
`https://sol.sbc.org.br/index.php/sbrc/article/view/2402/2366`

1 Introduction

Traditionally, network functions such as firewalls, routers, and proxies are attached to dedicated devices. It turns network management less flexible and with lower expansion capacity. Network functions virtualization (NFV), in which general-purpose servers host network functions, is an alternative that offers a solution to these problems. NFV has been attracting attention in the telecommunications industry due to the possibility of reducing operational and maintenance costs. Accordingly, the European Telecommunication Standard Institute (ETSI) [15] has published a series of specifications for a standardized NFV architecture.

One barrier to the adoption of NFV is the performance of virtual network functions (VNFs) [25]. This is because, in a virtualized environment, the computational instructions performed on the guest operating system (OS) installed on a virtual machine need to be managed by a hypervisor [10]. The hypervisor guarantees isolation between virtual machines but imposes a performance penalty. Hence, applications that run on virtual machines tend to exhibit lower performance than applications run in nonvirtualized environments. Moreover, each virtual machine created in a standard virtualization environment requires OS installation and the reservation of resources, such as memory. Under certain circumstances, this can lead to a waste of resources when the virtual machine is idle. A performance gain can be achieved through paravirtualization since the virtual machine is modified to access the physical computer hardware directly. However, the virtual machine parameters, such as the amount of memory and the number of virtual processors, still need to be configured before VNF instantiation.

Experiments conducted in [17] show that, for the same hardware configuration, software-based routers running Linux can achieve a forwarding packet rate of 1.2 M packets/s, whereas in a Xen [4] virtualized environment, the forwarding packet rate is approximately 0.2 M packets/s. Thus, applications that demand a high forwarding rate have performance problems when executed in standard virtualization environments. This shows the challenge of implementing network functions as VNFs. A solution to this problem is the adoption of lightweight virtualization, also known as container-based virtualization. In this type of virtualization, isolation is applied only at the process level instead of at the level of the whole OS. Because containers do not suffer from hypervisor overhead, we can expect a performance improvement of the virtualized applications. Hence, it is critical to evaluate whether we can virtualize a network function and which type of virtualization solution is suitable. The use of containers can be interesting for VNFs because an application's performance using a container can be close to that in a native Linux system due to the absence of a hypervisor layer.

A proxy is one type of function that can be run as a VNF. A hypertext transfer protocol (HTTP) proxy is responsible for responding to HTTP requests on behalf of servers. Thus, it is possible, for example, to filter requests, allowing access only to certain server pages. We can also reduce the response

time of a request and the access link utilization when the proxy is associated with a cache. However, according to [30], proxy services are sensitive to processor (CPU) usage, which can become a drawback when using virtualization. Due to this possibility and the fact that proxies are among the most used network functions on the Internet, we evaluate the corresponding performance.

The work in [14] presents a performance evaluation of a single HTTP proxy implemented as a VNF. The evaluation considers two different open-source virtualization platforms: Kernel-based Virtual Machine (KVM) [22], a full virtualization platform by default, and Docker [12], which implements lightweight virtualization using containers. The results show that Docker containers yield performance close to that of native Linux, representing a good alternative to KVM. This work extends [14] by evaluating the performance of several simultaneous proxy instances. This evaluation is essential since, in production networks, the number of proxies must vary according to the demand (e.g., as a function of the number of received requests). We can observe that using two proxy instances improves performance compared with that achieved using only one proxy. In this case, Docker containers come even closer to native Linux in terms of performance. The results also show that in our scenario, the performance improvement is not significant for more than two proxies. In addition, this work evaluates the performance of KVM paravirtualization machines. The aim of this evaluation is verify whether paravirtualization is a suitable alternative to Docker when flexibility and isolation are more significant concerns. The results show that when a performance drop is acceptable, paravirtualization in KVM can be considered, with the performance being approximately 50% better than that achieved using KVM with full virtualization.

This work is organized as follows. Section 2 describes related work. Section 3 characterizes the virtualization types used in this work. Section 4 addresses the performance evaluation. Finally, Section 5 presents conclusions and future work.

2 Related Work

There are several studies evaluating the performance of web proxies and virtualization solutions considering various applications/functions. Kim *et al.* [19] evaluate how big data transfers impact the performance of a web proxy with a cache. The authors assess metrics such as latency and throughput in a campus area network, focusing only on cases that can generate bad behavior of the cache or service failures. The results show that large concurrent objects considerably degrade the cache system performance even when the number of transmitting computers is small. To solve this problem, the authors propose a peer-to-peer cooperative web caching scheme. They show that this scheme performs the task of caching and delivery of large objects in an efficient and cost-effective way. However, this work does not evaluate the performance of the proxy system implemented as a VNF.

Regarding virtualization, solutions not applied to specific network functions are evaluated by [3,13,33,16]. Bachiega *et al.* [3] present a survey of the literature on the performance evaluation of containers. They identify the performance evaluation methods (i.e., analytical modeling, simulation, and measurement) and workloads used in several papers from Springer, IEEE, ACM, and Scopus databases related to performance evaluation and container virtualization. Challenges of the area and future work are also taken into account. The analysis shows that a performance evaluation can rely on benchmarks for CPUs, memory, databases, networks, scientific applications, and virtualization tools. They also observe that there is still little research related to the performance evaluation of containers.

Dua *et al.* [13] present a theoretical analysis of solutions based on hypervisors and containers. Regarding containers, the authors evaluate Linux Containers (LXC), Docker, Warden Container, Google lmctfy, and OpenVZ. They compare characteristics such as isolation, data storage, and network communications. The main focus is on containers and their capacity to host applications. They conclude that Docker is the most relevant solution for PaaS (Platform as a Service) implementors, and OpenVZ is the most secure one though it needs a modified kernel. The analysis is relevant, but they do not evaluate performance metrics focusing on NFV. Yang and Lan [33] propose a performance evaluation model based on queuing theory for virtual machines in KVM. This model can calculate the performance of a virtual server by taking into account practical data produced during the server use. They evaluate parameters that can impact virtual machine performance, such as processing speed, floating-point operations, and file copy rate. The results inspire an index for comparing solutions in terms of efficiency in scenarios involving web and database servers. However, Yang and Lan do not consider the use of containers. Felter *et al.* [16] evaluate database applications, MySQL and Redis, in KVM and Docker. They use metrics such as memory and CPU utilization, transaction throughput, and network latency. The results show that container virtualization improves performance. However, they show that some Docker functions, such as NAT (Network Address Translation), introduces overheads that may impact database performance. Although Felter *et al.* provide insightful conclusions, database applications have different requirements than network functions do. Hence, their study is not necessarily applicable to NFV services.

Other studies, such as [5,18,7], investigate the application of virtualization solutions to network functions. Bondan *et al.* [5] analyze a VNF using ClickOS [23] and OSv [20], based on standard virtualization, and CoreOS [11], a lightweight virtualization solution. The authors evaluate the boot time required to create and initialize new VNFs on containers and virtual machines. They also assess memory usage and response time for these VNFs. They show that ClickOS and CoreOS outperform OSv. Bondan *et al.* state that ClickOS has better boot and response times than CoreOS. However, CoreOS requires less memory due to its lightweight virtualization. Their work uses a simple VNF that only forwards traffic. In contrast, our analysis uses an actual proxy

implementation. Additionally, we analyze the effect of load balancing between containers, while [5] considers only one container running the VNF. Heideker and Kamienski [18] present NFV and software-defined networking (SDN) as possible solutions to improve flexibility and quality of service in smart city and Internet of Things scenarios. NFV is used to solve the elasticity problem related to public Internet access in large cities. The authors present a VNF for NAT management in a smart city. In the evaluations, KVM and Xen are used as standard virtualization platforms, and LXC is used as a lightweight virtualization platform. Their work evaluates performance metrics in the NAT VNF, such as the average throughput, response time, and resource utilization. In their experiments, LXC performs better than KVM. For example, the average throughput for LXC is approximately 80% higher than that for KVM. The work is related to NFV infrastructure, but the focus is on NAT instead of HTTP proxies.

In the context of 5G networks, Chang *et al.* [7] consider the installation of Open5GCore, an SDN-enabled Evolved Packet Core (EPC), on physical machines, KVM hypervisors, and Docker containers. They also present an experimental performance evaluation using different types of traffic. The main results show that KVM and Docker perform as well as physical machines when lightweight traffic, such as VoIP traffic, is sent. However, the performance gap becomes significant when the traffic is heavy, for example, in high-volume data transmissions. Despite its better performance, Chang *et al.* conclude that Docker may lead to security issues. These issues happen since Docker must work on a privileged mode to allow all capabilities needed by Open5GCore. Again, the authors do not evaluate HTTP proxy solutions.

We highlight the main characteristics of the related work in Table 1. It is worth noting that, despite Kim *et al.* [19] and Yang and Lan [33], most of the related work considers lightweight virtualization. Furthermore, we can state that this virtualization type is suitable for different applications, from database management systems to network functions. Our work follows this direction, showing that Docker is practical to HTTP proxies and outperforms a standard virtualization solution. Although NFV is a subject widely studied in academia and the telecommunications industry, we are not aware of any studies focusing on virtualized HTTP proxies. To address this gap, this work evaluates the performance of two virtualization solutions, KVM and Docker, when used in an HTTP proxy.

3 Virtualization Solutions

Virtualization solutions multiplex access to the physical hardware, allowing multiple virtual instances to run on the same machine. These solutions, conceived initially for datacenters, can run VNFs that need network hardware multiplexing. In this work, we classify these solutions into standard virtualization, which can be full virtualization or paravirtualization, and lightweight virtualization, also known as container-based virtualization.

Table 1 Main characteristics of related work.

Work	Virtualization Type	Application/Function
Kim *et al.* [19]	-	HTTP Proxy
Bachiega *et al.* [3]	Lightweight	Several (Survey)
Dua *et al.* [13]	Standard and Lightweight	-
Yang and Lan [33]	Standard	HTTP and MySQL
Felter *et al.* [16]	Standard and Lightweight	MySQL and Redis
Bondan *et al.* [5]	Standard and Lightweight	Packet Forwarding
Heideker and Kamienski [18]	Standard and Lightweight	NAT
Chang *et al.* [7]	Standard and Lightweight	VoIP, Video, and FTP
This work	Standard and Lightweight	HTTP Proxy

3.1 Standard Virtualization

Standard virtualization solutions multiplex access to the hardware by virtual machines. To do this, they implement a software layer named a hypervisor, which lies between the hardware of the physical machine and the virtual machines. The hypervisor controls access to the hardware and is responsible for delivering the virtual machine abstractions to the host OS, providing isolation between the virtual machines. In this way, different virtual machines can implement different OSes, referred to as guest OSes. In short, each virtual machine has its own virtual hardware abstraction and its own OS, which is responsible for controlling the devices necessary for its operation, such as CPUs, memories, and hard disks.

Depending on the virtualization project's needs, the virtualization implementation can be total, referred to as full virtualization, or partial, referred to as paravirtualization. These two types of virtualization differ with respect to the calls that pass through the hypervisor, as detailed below. Regardless of the virtualization solution, the environment must be carefully studied and correctly dimensioned since these solutions increase the complexity and modify the dynamics of infrastructure operation.

3.1.1 Full Virtualization

In full virtualization, the hypervisor intercepts virtual machine instructions before they reach the hardware. For example, if a virtual machine needs to send a network packet, the hypervisor must intercept the message before copying it to the network interface card. This interception guarantees isolation without the need to modify the OS of the virtual machine [4]. Various full virtualization solutions are available, such as VMware ESXi [32], Citrix XenServer [8], Microsoft Hyper-V [24], and Oracle VirtualBox [27].

The main drawback of using full virtualization in NFV is the performance bottleneck imposed by the hypervisor on the virtual machines. All the virtual machine's networking operations first need to pass through the hypervisor layer before reaching the hardware. Consequently, applications running in a virtualized environment may experience a performance drop compared to applications that do not use virtualization [17].

3.1.2 Paravirtualization

The concept of paravirtualization is similar to the principle of full virtualization. A hypervisor runs on a physical machine, allowing the instantiation of guest virtual machines. The difference lies in the fact that the guest OS of a virtual machine is aware of its virtualization. Consequently, it can make calls directly to the hardware, thus improving performance. OS changes are implemented as an alternative to the hypervisor's intercepts, reducing the processing overhead. Hence, some privileged instructions are executed directly on the hardware, performing I/O and protected memory operations [2].

In this work, we use the KVM [22] hypervisor because it is an open-source tool that supports both paravirtualization and full virtualization and is integrated with the Linux kernel. In our environment, VirtIO implements paravirtualized network interfaces [31]. VirtIO is a framework designed to accelerate I/O instructions in a virtualized KVM environment. In brief, VirtIO provides drivers and shared queues between the guest OS and the host OS. These queues can be directly accessed by virtual machines, decreasing the overhead imposed by the hypervisor [26].

3.2 Lightweight Virtualization

Lightweight virtualization offers a different level of virtualization and isolation than standard virtualization. Unlike in full virtualization and paravirtualization, containers are used to implement isolation between process-level applications in the host OS, thus avoiding hypervisor overhead. Hence, the containers share the same OS installed on the physical machine.

Because the containers share the host OS kernel, lightweight virtualization provides advantages such as the ability to quickly provision large numbers of application instances. This is possible because these instances do not require virtual hard disk images, unlike in standard virtualization approaches. However, since the containers share the same OS, it is impossible to run different OS kernels on the same physical machine. Another drawback when implementing container technology is the isolation method provided by these solutions. For example, a security problem in the kernel can affect all containers on a given physical machine [6]. In this work, we use Docker [12] to implement containers because it is an open-source tool that is widely employed in the software development industry and because NFV implementations use this solution.

4 Performance Evaluation

We analyze the processing time of HTTP requests using a testbed with three physical machines. Figure 1 shows the testbed, composed of one PC with HTTP clients, one with an HTTP server, and one intermediate machine. The

three physical machines used in our testbed have the same hardware. Each one has a quad-core Intel Xeon 3.2 GHz CPU, 16 GB of RAM, and four Gigabit Ethernet interfaces. The intermediate machine is directly connected to the other two using Ethernet cables.

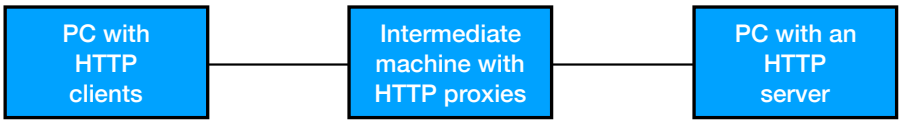

Fig. 1 Testbed topology.

The clients run in Docker containers, while the server runs natively in Linux. These two machines communicate through an intermediate one, which hosts the HTTP proxies. Depending on the experimental configuration, these proxies can run in Docker containers, on KVM virtual machines, or on the physical machine. All virtual and physical machines and containers run 64-bit Ubuntu Linux 14.04 LTS. On the intermediate machine, each VM has 4 GB of RAM and a quad-core virtual CPU. The maximum RAM allowed to a Docker container is also 4 GB, and it can use up to four CPU cores. We use this simple scenario to provide greater control over the experiments carried out.

We run KVM virtual machines using a configuration based on full virtualization and another one in which the network interfaces use VirtIO drivers. This latter uses a paravirtualization approach. For Docker evaluation, we also use two configurations, which differ in their network implementation. In the first configuration used in our experiments, which we call Docker-NAT, Docker forwards packets to containers using NAT. The second configuration is called Docker-Routing, in which static routing rules are employed to forward packets.

For each configuration, our experiment consists of generating HTTP requests from a given number of parallel clients and evaluating the processing time. The processing time is defined here as the time elapsed between sending the request and receiving its response. The HTTP server runs Apache 2 since Apache is the most widely used open-source HTTP server. The proxies run Squid 3.0 since it is a popular open-source proxy. For the generation of HTTP requests and the evaluation of processing time, we employ the Apache JMeter 3.1 tool [1]. For each configuration, we perform ten replicate experiments, and we present the results using confidence intervals at the 95% confidence level. However, for the majority of the results, the confidence intervals are imperceptible.

In an experiment, each client sends to the server 1000 simultaneous HTTP requests for the same object of 1024 kbytes in size. We choose this size to simplify our experiments, since the conclusions presented in this work are also valid for larger sizes [14]. After the reception of all server replies, we evaluate the average processing time. We employ the Linux Stress tool with 12 CPU-intensive processes to induce a processing load in the intermediate machine during the experiment. We choose this tool since it is already available in

Ubuntu Linux. By applying this tool, we aim to evaluate the performance in saturated environments. We perform the experiments in two parts. The purpose of the first part is to evaluate the processing time for different virtualization techniques. In the second part, we focus on Docker containers, evaluating the processing time behavior as the number of parallel proxies increases.

4.1 Comparison of Virtualization Techniques

In this first part, we consider ten parallel clients and evaluate the performance when there are one or two proxies in the intermediate machine. Each proxy is a VM or a Docker container. For the native Linux evaluation, each proxy is a different Squid process listening on a different TCP port. The purpose of using two proxies is to check whether we can improve the processing time by parallelizing the proxy execution.

For the experiments in this first part, we evaluate the average processing time for HTTP requests when the proxy cache is either enabled or disabled. When the proxy has a cache enabled, it stores the objects received in past replies. Hence, it avoids contacting the server when requesting the same object multiple times, thus improving the processing time. The cache is generally disabled when the proxy is employed only to filter web content. We then present the results considering these two cache configurations.

4.1.1 Average Processing Time Without Caches

Figure 2 shows the average processing time for each configuration, considering all ten clients. For each virtualization solution, this figure also shows the results for one and two proxies. It is apparent that when paravirtualization is employed in KVM, its performance improves relative to the full virtualization approach. However, paravirtualized VMs still require a higher processing time than Docker. We also observe that all Docker configurations exhibit performance closer to that of native Linux, which acts as a lower bound on the processing time. This is because Docker operation incurs less overhead than KVM due to the absence of a hypervisor.

Figure 2 also shows that Docker-NAT has a slightly longer processing time than Docker-Routing. This difference occurs since Docker-NAT attaches the physical and virtual network interfaces to a software bridge, which adds a small overhead. Table 2 shows the CPU and memory utilization in our experiment for these two configurations. To evaluate these values more accurately, we disable the Linux Stress tool and send requests using only one client. The results in Table 2 show that Docker-NAT uses more CPU resources than Docker-Routing, which explains its longer processing time. Table 2 also indicates that Docker-Routing uses more memory since it needs to store the containers' IP addresses. Section 4.2 builds on this last conclusion.

Finally, Figure 2 shows that, for all configurations, the average processing time drops when two proxies are employed. In the case of Docker containers,

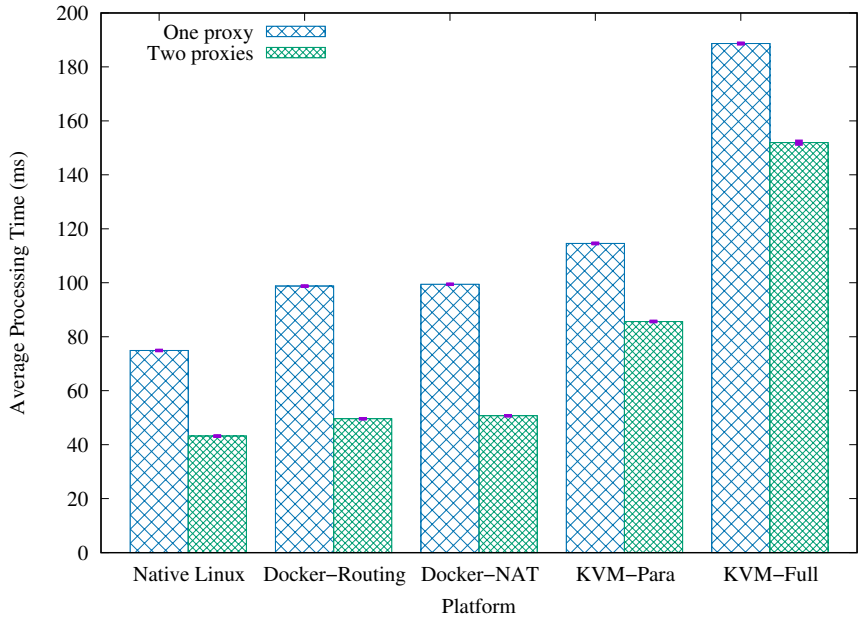

Fig. 2 Average processing time for 10 clients when caches are disabled. The results include confidence intervals at the 95% confidence level. However, these intervals are tiny and not easily visible in this figure.

Table 2 CPU and memory utilization in Docker.

Configuration	CPU (%)	Memory (%)
Docker-Routing	40.75 ± 0.02	3.750 ± 0.001
Docker-NAT	42.67 ± 0.02	1.000 ± 0.001

the performance becomes very close to that of native Linux. These results also show that if the requirements for a web application are replaced with stricter ones, we can instantiate more VMs or containers to achieve shorter processing times. Similarly, if the number of web requests increases, we can create more VMs and containers to reduce the processing time.

The reaction to new requirements or higher loads is closely related to the creation of new VMs or new containers. To compare Docker and KVM in this case, we measure the time needed to instantiate a VM or a container. We instantiate ten VMs or ten containers and evaluate the average time to complete this operation. We perform ten replicates of this experiment, measuring their average and confidence intervals at the 95% confidence level. In the experiments, it is assumed that the physical machine disk already has the required VM or container images. Table 3 presents the obtained results, which show that we can instantiate Docker containers in much less time than KVM VMs. This is because a container needs to maintain less state information than a VM because it shares various resources with the physical host [5]. For exam-

ple, each VM needs a virtual disk to store its OS, while a container shares the same OS as its physical host.

Table 3 Time to instantiate a VM or a container.

Virtualization Solution	Time (s)
KVM	52.00 ± 12.05
Docker	4.00 ± 0.15

4.1.2 Average Processing Time With Caches

We also evaluate the average processing time for Squid 3.0 proxies with a cache. In this case, we expect a reduction in the average processing time compared to the values obtained in Section 4.1.1. The aim of this experiment is to verify whether the corresponding comparison of the virtualization techniques leads to the same conclusions as in Section 4.1.1. When a cache is enabled in Squid 3.0, the HTTP server replies to the first request for a given object, and a copy of this object is stored locally by the proxy. This copy is then used to serve future requests for this object, which are handled directly by the proxy, thus reducing the response time. The cache in Squid 3.0 has two important configuration parameters: the maximum object size and the total memory size. The first defines the maximum size of an object that can be stored in the cache, which we set to 1024 kbytes, as recommended by the Squid 3.0 documentation to avoid performance degradation. The second parameter is the amount of RAM dedicated to the proxy, for which we use the default value of 260 Mbytes. We configure the other parameters of Squid 3.0 with their default values.

Figure 3 shows the average processing time for HTTP requests with one and two proxy instances when the caches are enabled. As in the previous experiment presented in Section 4.1.1, Docker containers exhibit performance closer to that of native Linux, especially with two proxy instances. With caches, the performance of paravirtualized KVM is more similar to that of Docker than it is when caches are disabled. A comparison between Figures 2 and 3 also shows that, regardless of the virtualization technique, a cache can considerably improve an HTTP proxy's performance.

4.2 Behavior With Multiple Docker Containers

Sections 4.1.1 and 4.1.2 compare different virtualization solutions when executing proxies as VNFs. The results show that Docker containers achieve performance close to that of native Linux. The results also show that the use of two proxies improves load balancing, bringing the Docker performance even closer to that of native Linux. Since Docker achieves the best performance among the virtualization technologies evaluated in this work, this next experiment analyzes in more detail its ability to perform load balancing. To this end,

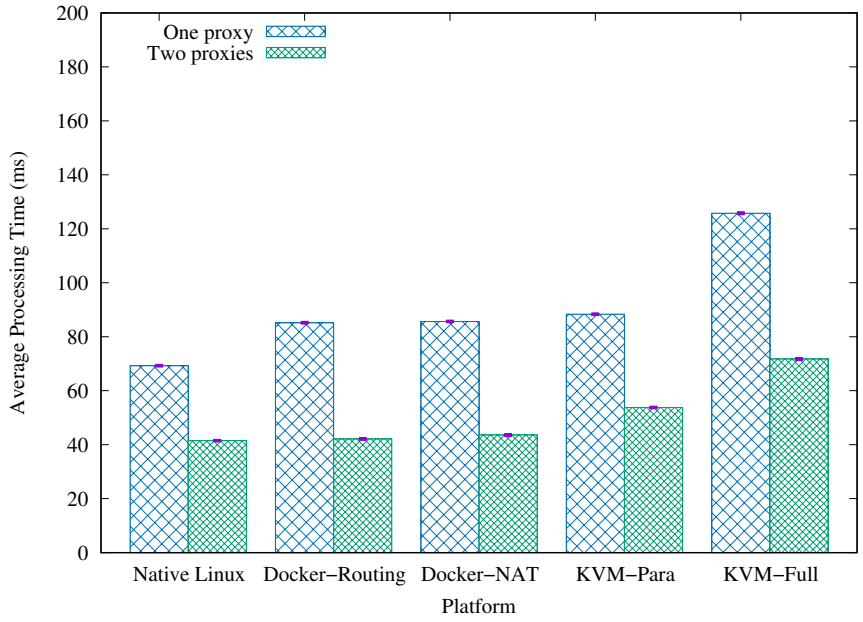

Fig. 3 Average processing time for 10 clients when caches are enabled. The results include confidence intervals at the 95% confidence level. However, these intervals are tiny and not easily visible in this figure.

we create 64 client containers to request files over HTTP, each one with the same behavior as in the previous experiments. This number of clients makes it possible to vary the number of proxies by powers of two and ensure that they receive an equal number of clients. As in the previous experiments, the files have a size of 1024 kbytes. The proxies each have a cache since this is the configuration with the best results in the earlier experiments.

Each client container runs an Apache JMeter instance to send requests. The intermediate machine, which hosts the proxies, runs up to 32 containers with Squid 3.0. This number is close to the number of Docker containers supported by the employed hardware; at 38 instances, the containers begin to refuse connections with a "no route to destination" message. We vary the number of proxies by activating the desired number of containers on the intermediate machine. We equally split the traffic from the 64 clients among the proxies. For example, when the number of proxies is four, we activate four containers on the intermediate machine, and each container thus receives traffic from 16 clients.

Figure 4 shows the average processing time for 64 clients as a function of the number of proxies on a log scale. Note that the response time is higher than the results in Sections 4.1.1 and 4.1.2 since more clients are employed in this experiment (i.e., 64 instead of 10). It can be seen that load balancing with two instances results in a noticeable improvement in the processing time (i.e.,

approximately 70 ms) compared with the case of only one instance *proxy*. The experiments in Sections 4.1.1 and 4.1.2 also show this behavior.

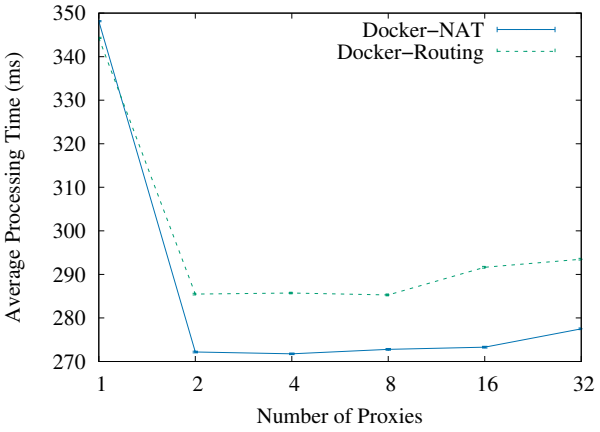

Fig. 4 Average processing time for 64 clients and with caches enabled as a function of the number of proxies. The results include confidence intervals at the 95% confidence level. However, these intervals are tiny and not easily visible in this figure.

Figure 4 also shows that, for one proxy, the processing time for the NAT configuration is higher than that for the routing configuration. This behavior is also observed in Sections 4.1.1 and 4.1.2. However, for 64 clients and a higher number of proxies (i.e., more than two proxies), this situation is inverted. This is because Docker-Routing uses more memory than Docker-NAT, as shown previously in Table 2. Hence, we need to store more routes as the number of clients increases. Because the routing table is the same for all intermediate machine containers, the time taken when accessing this table increases. Finally, Figure 4 shows that the performance does not improve with more than two proxies. Additionally, at eight proxies, the processing time starts to increase. Docker isolation problems explain this behavior since all containers share CPU and memory and thus can interfere with each other [9].

5 Conclusion

NFV replaces dedicated equipment with virtual network functions, increasing infrastructure flexibility as well as reducing costs. Nevertheless, the performance of virtual network functions should be carefully analyzed. In this work, we have analyzed the performance of virtualized HTTP proxies using different virtualization technologies. Our results allow us to conclude that Docker containers enable network performance close to that of native Linux when executing HTTP proxies.

Despite the better performance of Docker, standard virtualization solutions, such as KVM, offer a higher level of isolation. Hence, these solutions

should be employed when more isolation is needed. In this work, we have evaluated a paravirtualization solution for KVM, named VirtIO. The results show that this solution significantly improves a proxy's processing time, but not enough to outperform Docker. Thus, in situations where standard virtualization is necessary, it is possible to use paravirtualization techniques to improve performance. As noted in [28], the performance of VirtIO depends on how much of the hardware is dedicated to virtual machines; that is, the more resources are accessed directly by the virtual machines, the better the performance.

If an HTTP proxy needs a high packet processing rate, as in the context of large-scale Internet providers, Docker and KVM alone can fail to meet the performance requirements. In this case, hardware-based solutions should be introduced. For example, field programmable gate arrays (FPGAs) can be used to implement protocols in hardware [29]. However, hardware-based solutions lack Docker and KVM's flexibility since it is necessary to consider the specific hardware when designing network functions.

This work has also analyzed the extent to which load balancing can improve the performance of virtual proxies. The results show that, for all virtualization solutions, instantiating two proxy instances improves performance compared to the scenario with only one proxy. In this case, Docker containers achieve performance even closer to that of native Linux. Finally, we analyze the Docker load balancing capacity as a function of the number of proxies for up to 32 instances. With more than two proxies, the average processing time is not reduced, and thus, there is no performance improvement. Moreover, increasing the number of containers increases memory consumption, which can lead to a performance decrease.

As future work, it will be interesting to evaluate the scalability of Docker considering a complete NFV solution using, for example, orchestration tools such as Kubernetes [21]. It will also be important to extend this work by assessing how Docker's isolation can affect the security of VNFs.

References

1. Apache: Jmeter 3.1 (2020). Http://jmeter.apache.org
2. Babu, A., M, H., Martin, J.P., Cherian, S., Sastri, Y.: System performance evaluation of para virtualization, container virtualization and full virtualization using Xen, OpenVZ and XenServer. In: International Conference on Advances in Computing and Communications, pp. 247–250 (2014)
3. Bachiega, N.G., Souza, P.S.L., Bruschi, S.M., do R. S. de Souza, S.: Container-based performance evaluation: A survey and challenges. In: IEEE International Conference on Cloud Engineering, pp. 398–403 (2018)
4. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: ACM Symposium on Operating Systems Principles (SOSP), p. 164–177 (2003)
5. Bondan, L., dos Santos, C.R.P., Granville, L.Z.: Comparing virtualization solutions for NFV deployment: A network management perspective. In: IEEE Symposium on Computers and Communication (ISCC), pp. 669–674 (2016)
6. Bui, T.: Analysis of Docker security. Tech. rep., Aalto University School of Science, Finland (2015). Http://arxiv.org/abs/1501.02967

7. Chang, H.C., Qiu, B.J., Chen, J.C., Tan, T.J., Ho, P.F., Chiu, C.H., Lin, B.S.P.: Empirical experience and experimental evaluation of Open5GCore over hypervisor and container. Wireless Communications and Mobile Computing **2018** (2018)
8. Citrix: Citrix hypervisor (2020). Https://www.citrix.com/products/citrix-hypervisor/
9. Combe, T., Martin, A., Pietro, R.D.: To Docker or not to Docker: A security perspective. IEEE Cloud Computing **3**(5), 54–62 (2016)
10. Couto, R.S., Campista, M.E.M., Costa, L.H.M.K.: Network resource control for Xen-based virtualized software routers. Computer Networks **64**, 71–88 (2014)
11. Docker: CoreOS: Open source projects for Linux containers (2020). Https://coreos.com
12. Docker: Docker - build, ship and run any app, anywhere (2020). Https://www.docker.com
13. Dua, R., Raja, A.R., Kakadia, D.: Virtualization vs containerization to support PaaS. In: IEEE International Conference on Cloud Engineering (IC2E), pp. 610–614 (2014)
14. Eiras, R.S.V., Couto, R.S., Rubinstein, M.G.: Performance evaluation of a virtualized HTTP proxy using KVM and Docker. In: International Conference on the Network of the Future (NoF), pp. 1–5 (2016)
15. ETSI, N.: Network functions virtualisation (NFV) architectural framework. ETSI GS NFV **2**(2), V1 (2013)
16. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. Tech. rep., IBM (2015). IBM Research Report - RC25482 (AUS1407-001)
17. Fernandes, N.C., Moreira, M.D.D., Moraes, I.M., Ferraz, L.H.G., Couto, R.S., Carvalho, H.E.T., Campista, M.E.M., Costa, L.H.M.K., Duarte, O.C.M.B.: Virtual networks: Isolation, performance, and trends. Annals of Telecommunications **66**(5-6), 339–355 (2011)
18. Heideker, A., Kamienski, C.: Managing elasticity in an NFV-based IaaS environment. In: Workshop of IEEE Latin-American Conference on Communications (2017)
19. Kim, H.C., Lee, D., Chon, K., Jang, B., Kwon, T., Choi, Y.: Performance impact of large file transfer on web proxy caching: A case study in a high bandwidth campus network environment. Journal of Communications and Networks **52**, 52–66 (2010)
20. Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., Zolotarov, V.: OSv — optimizing the operating system for virtual machines. In: USENIX Annual Technical Conference (ATC), pp. 61–72 (2014)
21. Kubernetes: Kubernetes (2020). Https://kubernetes.io
22. KVM: Kernel-based virtual machine (2020). Https://www.linux-kvm.org/page/Main_Page
23. Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., , Honda, M.: ClickOS and the art of network function virtualization. In: USENIX Symposium on Networked Systems Design and Implementation (NSDI), pp. 459–473 (2014)
24. Microsoft: Microsoft Hyper-V (2017). Https://msdn.microsoft.com/pt-br/library/hh831531(v=ws.11).aspx
25. Mijumbi, R., Serrat, J., Gorricho, J.L., Bouten, N., Turck, F.D., Boutaba, R.: Network function virtualization: State-of-the-art and research challenges. IEEE Communications Surveys and Tutorials **18**(1), 236–262 (2016)
26. Nakajima, Y., Masutani, H., Takahashi, H.: High-performance vNIC framework for hypervisor-based NFV with userspace vswitch. In: Fourth European Workshop on Software Defined Networks, pp. 43–48 (2015)
27. Oracle: Virtualbox (2020). Https://www.virtualbox.org
28. Rasmusson, L., Corcoran, D.: Performance overhead of KVM on Linux 3.9 on ARM cortex-a15. ACM SIGBED Review **11**(2), 32–38 (2014)
29. Shantharama, P., Thyagaturu, A.S., Reisslein, M.: Hardware-accelerated platforms and infrastructures for network functions: A survey of enabling technologies and research studies. IEEE Access **8**, 132021–132085 (2020)
30. Squid: Squid proxy (2020). Http://www.squid-cache.org/Intro
31. VirtIO: Virtio - paravirtualized drivers for KVM/Linux (2020). Http://www.linux-kvm.org/page/Virtio
32. VMware: Vmware - official site (2020). Https://www.vmware.com/
33. Yang, J., Lan, Y.: A performance evaluation model for virtual servers in KVM-based virtualized system. In: IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity), pp. 66–71 (2015)

Acknowledgements This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. It was also supported by CNPq, FAPERJ, and FAPESP Grant 15/24494-8.