Hindawi
Mobile Information Systems
Volume 2018, Article ID 9570506, 6 pages
https://doi.org/10.1155/2018/9570506

Research Article

The Impact of Container Virtualization on Network Performance of IoT Devices

Kyungwoon Lee, Youngpil Kim, and Chuck Yoo

College of Informatics, Korea University, Seoul, Republic of Korea

Correspondence should be addressed to Chuck Yoo; chuckyoo@os.korea.ac.kr

Received 15 December 2017; Accepted 20 March 2018; Published 2 May 2018

Academic Editor: Jeongyeup Paek

Container-based virtualization offers advantages such as high performance, resource efficiency, and agile environment. These advantages make Internet of Things (IoT) device management easy. Although container-based virtualization has already been introduced to IoT devices, the different network modes of containers and their performance issues have not been addressed. Since the network performance is an important factor in IoT, the analysis of the container network performance is essential. In this study, we analyze the network performance of containers on an IoT device, Raspberry Pi 3. The results show that the network performance of containers is lower than that of the native Linux, with an average performance difference of 6% and 18% for TCP and UDP, respectively. In addition, the network performance of containers varies depending on the network mode. When a single container runs, bridge mode achieves higher performance than host mode by 25% while host mode shows better performance than bridge mode by 45% in the multicontainer environment.

1. Introduction

The Internet of Things (IoT) has been adopted by diverse industries such as transportation, smart home, and national utilities [1]. As the number of IoT devices increases [2], the exponential growth in the volume of IoT devices poses challenges in terms of device management [3]. Although several management techniques [4] have been proposed for automatic configuration, software upgrade, and fault management, they predominantly rely on management protocols [3]. As a result, it is difficult to manage a large number of IoT devices which do not support management protocols.

Celesti et al. [5] and Mulfari et al. [6] proposed applying container-based virtualization [7] to IoT devices for resolving the management issue. Container-based virtualization allows multiple execution environments (containers) to run concurrently on an operating system (OS) and simplifies the packaging, distribution, installation, and execution of IoT devices. For example, software updates for enhanced security or functionalities in IoT devices can be accomplished simply by replacing old container images with the new container images. Furthermore, container orchestration tools such as Kubernetes [8] and Docker Swarm [9] can automate container operations.

In addition, container-based virtualization in IoT devices enable to run different IoT applications simultaneously, which improves resource utilization. For example, temperature and motion sensing can be performed concurrently by running the tasks in different containers. Because each container has an independent root file system and libraries, a container has its own execution environment while sharing computing resources in an IoT device with multiple containers. Furthermore, IoT platforms [10, 11] provide different APIs and functionalities, and so IoT applications are developed in target IoT platforms. With containers, although IoT applications are developed on different IoT platforms, they can run on an IoT device by operating multiple IoT platforms in different containers simultaneously.

To utilize such advantages of containers, previous studies [5, 6, 12, 13] deployed containers to IoT devices and conducted performance evaluation to compare the performance of containers with the native environment. However, they did not conduct detailed network performance analysis of

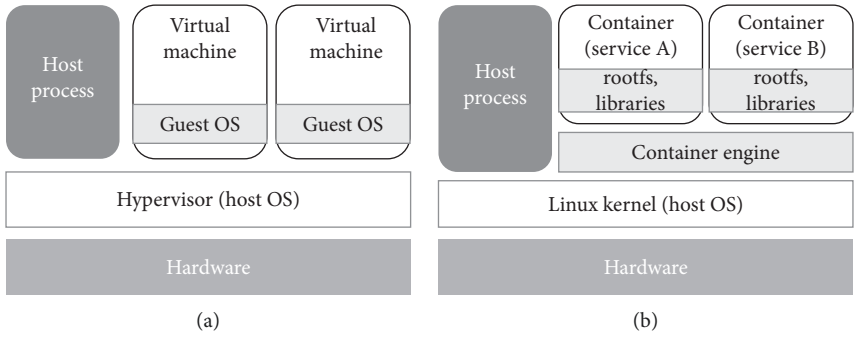

Figure 1: Comparison between server virtualization (a) and container-based virtualization (b).

containers on IoT devices. This paper is a major extension of our previous work presented in [14] and focuses on the network performance of containers on IoT devices. Note that container-based virtualization provides two different network modes: bridge mode and host mode. Because each mode goes through a different path in the network stack of the shared OS kernel, they exhibit different performances and behaviors. However, the difference between the network modes of containers has not been addressed in previous studies. Second, when multiple containers with different IoT platforms run on an IoT device, resource contention can arise. Because the physical resources in IoT devices are not as powerful as those of servers, management of resource contention is an important issue that has to be addressed.

In this study, we first deploy container-based virtualization on IoT device, Raspberry Pi 3, using the Docker [15] open-source container engine. Then, we analyze the network performance of containers as follows: (1) we observe the performance of a single container according to the network mode, (2) evaluate the performance when multiple containers run concurrently, and (3) investigate performance bottlenecks through system-level profiling. This analysis is carried out with the objective of identifying the performance issues of containers.

2. Background: Container-Based Virtualization

Container-based virtualization provides isolation between multiple containers running on a general-purpose OS such as Linux (called the host OS). In contrast to server virtualization that requires separate OS instances for virtual machines, multiple containers share the same OS kernel (Figure 1). Each container has an independent root file system and safely shared system binaries and libraries. Based on the lightweight architecture of container-based virtualization, containers offer several advantages over virtual machines such as high performance, resource efficiency, and agile environment. As a result of these advantages, containers have been adopted in the IT industry in areas such as cloud data centers, mobile systems, and networks [16–18].

Containers use two key techniques to provide isolated execution environments: namespace and cGroup. Namespace is used for isolation between execution environments that include the process trees, network, user IDs, and file systems of containers. Linux offers six types of namespaces (mnt, pid, net, pic, uts, and user) that can allocate an independent space for a container and manage the space individually. cGroup manages resources such as CPU, memory, network, and file I/O for containers. With cGroup, the resource usage of containers can be controlled by assigning specific bounds to containers. In addition, a specific amount of resources for a container can be allocated to prevent resource contention between containers running concurrently.

3. IoT Container Network Mode Analysis

3.1. Containers on IoT Devices. In container-based virtualization, as depicted in Figure 1(b), a container engine running on the Linux kernel allows users to create/remove container images and run/configure containers using namespace. In addition, the container engine can allocate resources such as CPU time and memory space for each container through cGroup. Each container runs as a user-level process in the host OS (Linux), which is the same as a host process. However, containers have independent root file system whereas host processes share the root file system of the host OS.

An example of IoT devices running containers is gateway devices [12]. Gateway devices receive data collected from sensor devices using a network protocol such as 6LoWPAN and CoAP and deliver the data to a server in a cloud data center using HTTP. Thus, two different containers can run on a gateway device: one (e.g., service A) communicating using CoAP and the other (e.g., service B) reporting status information of devices through HTTP. By running a service in each container, the gateway can perform colocated but distinguished services.

3.2. Network Mode. When a container is created by a container engine, the container engine configures the network mode of the container. First, bridge mode is the default network mode that utilizes the network bridge of Linux. The bridge is a Link Layer device that forwards traffic between networks based on MAC addresses and is therefore also referred to as Layer 2 device. It makes forwarding decisions based on the tables of MAC addresses that it builds by learning which hosts are connected to each network. Containers in bridge mode have a virtual network interface, eth0, which has independent MAC and IP addresses, as depicted in Figure 2(a). By assigning virtual network interfaces, bridge mode provides an independent network stack to each container for an isolated network environment. The virtual network interfaces of containers are connected to

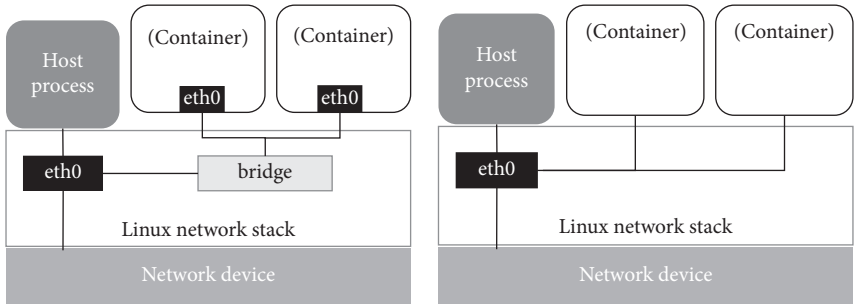

Figure 2: Network mode: bridge mode (a), host mode (b).

a network bridge in the host OS, and incoming packets are forwarded in the bridge based on their MAC addresses. Because every packet coming/outgoing to/from containers is forwarded by the bridge, the network performance of containers can be degraded when the bridge is overloaded.

On the other hand, host mode does not forward packets through the bridge, but processes the packets in the same network stack of the host OS. In Figure 2b, packets of containers are handled in the same manner as packets of host processes. Containers in host mode share the same MAC and IP addresses, and packets are delivered using port numbers. Because packets of containers and host processes are handled in the same network stack of the host OS, host mode does not take advantage of containers, which is isolation. For example, services having the same port number in different containers cannot run concurrently. The reason is that incoming packets cannot be delivered to corresponding containers when they have the same port number. Therefore, containers in the host mode can run services concurrently only when the services have different port numbers.

4. Experimental Results

We evaluate the network performance of containers with different network modes in three steps. First, we compare the network performance of a container in the default network mode (bridge mode) with that of a native user process to observe the performance overhead of a container, using TCP and UDP. Then, we measure the performance difference of bridge and host modes and investigate the reason through system-level profiling when a single container runs on an IoT device. Third, we increase the number of containers running concurrently from one to four and measure the aggregated throughput to see the performance overhead of running multiple containers.

Evaluation is conducted based on Docker container engine, on a Raspberry Pi 3 board (ARMv7 quad core, 1 GB RAM) running Ubuntu 16.04 LTS (kernel version 4.4.38). Our evaluation server running Ubuntu 16.04 LTS (kernel version 4.8.0) is equipped with Intel i7-3930K@3.2 GHz (hexa-core), 16 GB RAM, and 500 GB SSD to evaluate the network performance of Raspberry Pi 3 running containers. The server and Raspberry Pi 3 communicate with each other using a wireless access point, and the maximum throughput of the network is 20 Mbps, as in the hardware specification of Raspberry Pi 3 (Raspberry pi 3 supports 802.11 b/g/n wireless network and 802.11g is utilized in our experiment).

Netperf benchmark [19] is used to measure the network performance. Containers on Raspberry Pi 3 run a Netperf client that continuously transmits the specific size of packets while a Netperf server on the evaluation server measures the throughput. We configure the Netperf client on Raspberry Pi 3 to transmit small (64 B) and large (1024 B) packets, respectively, to observe the different behavior of containers depending on the packet size. Then, we run the Netperf client for 60 seconds and measure the average throughput.

4.1. Performance Comparison with Host OS. First, we evaluate the network performance of a container with the default network mode on a Raspberry Pi 3 compared to the native Linux for different packet sizes (64 B and 1024 B). Figure 3 shows the network throughput of a container and the native Linux in TCP and UDP. The container achieves lower performance than the nonvirtualized native process by an average of 6% and 18% for TCP and UDP, respectively.

To investigate the reason for the low performance of the container, we collect system parameters such as the number of CPU cycles consumed and the number of page faults using a Linux profiling tool, perf [20]. Through the system profiling, we find that the reason of the low performance is two-fold: additional CPU cycles for I/O processing and limited memory capacity. First, the container consumes more CPU cycles than the native Linux by two times. This is because containers require extra CPU cycles for I/O operations which Linux network bridge intervenes. This increases I/O latency and reduces available CPU cycles for the entire system [21]. Second, the container results in frequent page faults (13 page faults/s) compared to the native Linux (2 page faults/s) because of the limited memory capacity of containers (1 GB in our experiments). As a result of these overheads, containers have difficulty in achieving high performance closed to the native Linux. Even though the network performance of a container is not as high as the native Linux, for small (64 B) and large (1024 B) packets, the container exhibits almost the similar performance in TCP.

However, the throughput changes with the packet size for UDP; this is different from TCP, which achieves similar

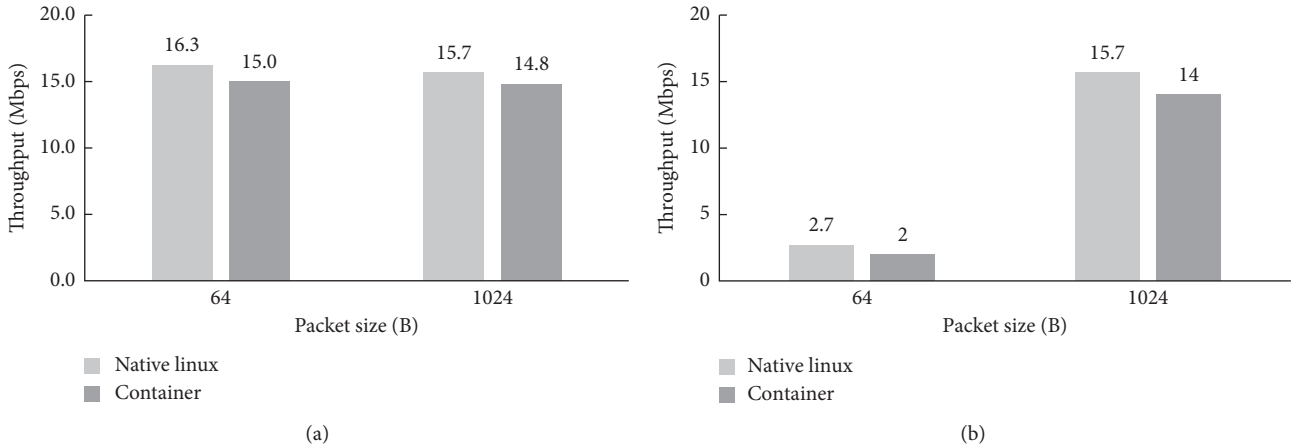

Figure 3: Network throughput of native process and a container in TCP (a) and UDP (b).

performance for different packet sizes. Our profiling results in Table 1 demonstrate that the number of CPU cycles and the number of context switches are much higher in UDP than TCP for 64 B packets (by a factor of 1.5 and 2, resp.). For 1024 B packets, all parameters of TCP and UDP are almost the same. This is due to the packet transmission method of the Netperf client. For TCP, the Netperf client allocates 16 KB of memory for the packet transmission by default and continuously transmits the allocated packets. On the other hand, for UDP, the Netperf clients allocate memory only for 32 packets by default. As a result, a system call for a 64 B packet transmission in TCP can deliver 256 packets at once while only 32 packets are delivered in UDP. So, UDP consumes more CPU cycles to send the same amount of packets than TCP. This difference results in the low performance of UDP.

Table 1 also shows that the number of context switches for 1024 B in TCP is higher than that for 64 B. The reason of high context switches for 1024 B is the frequent CPU migration of the Netperf client. For TCP, the number of CPU migration per second for 1024 B is four which is the double of that for 64 B. The reason is that the Netperf client transmitting large TCP packets (1024 B) consumes less CPU time compared to small TCP packets (64 B). For example, when 16 KB of memory is copied to the kernel networking stack, the number of packet header required for 64 B is larger than that for 1024 B. As a result, the Netperf client transmitting 1024 B packets is scheduled out easily and moved to another CPU core for load balancing.

Table 1: System-level statistics for different protocols and packet sizes.

Protocol	TCP		UDP	
Packet size	64 B	1024 B	64 B	1024 B
CPU cycles ($\times 10^6$)	985	570	1,597	510
Context switching	224	343	580	341
Page faults	13	5	6	6
Instructions/cycle	0.51	0.42	0.48	0.45

4.2. Performance Comparison with Host Mode. We measure the network throughput of a single container in different network modes to find out the performance difference between bridge and host modes. Figure 4 shows that bridge mode achieves higher performance than host mode for TCP by an average of 25%.

To investigate their difference at the system level, we measure the system parameters when a container transmits 64 B packets using TCP. In this case, there is a performance difference of 3 Mbps between bridge and host. From Table 2, host mode incurs a 50% higher rate of context switching than in bridge mode whereas other parameters are similar. This means that as network packets in host mode are handled, host mode requires more context switching than bridge mode. This frequent context switching degrades the network performance of a container by up to 25%.

On the other hand, in UDP, bridge mode achieves the same performance as host mode for 64 B packets while showing higher performance than that of host mode by 8% for 1024 B packets. This is because processing 64 B UDP packets consumes a large amount of CPU cycles with frequent context switches (Table 1) for both of bridge and host, which causes performance degradation independent of network modes. Since processing 1024 B UDP packets consumes less CPU cycles than 64 B, bridge mode is able to achieve higher performance than host mode by utilizing available CPU cycles.

In addition, we measure the aggregated throughput of multiple containers when the number of colocated containers varies from one to four. Figure 5 depicts that the aggregated network throughput of the containers differs between the network modes when the number of containers transmitting 64 B TCP packets increases. The aggregated throughput of bridge and host modes increases as the number of containers increases up to three and decreases when four containers run concurrently. This is because Raspberry Pi 3 is equipped with quad core CPU, and CPU contention arises when four containers and interrupt handlers for incoming packets compete for CPU.

Moreover, the aggregated throughput of bridge mode degrades by 31% compared to host mode when four

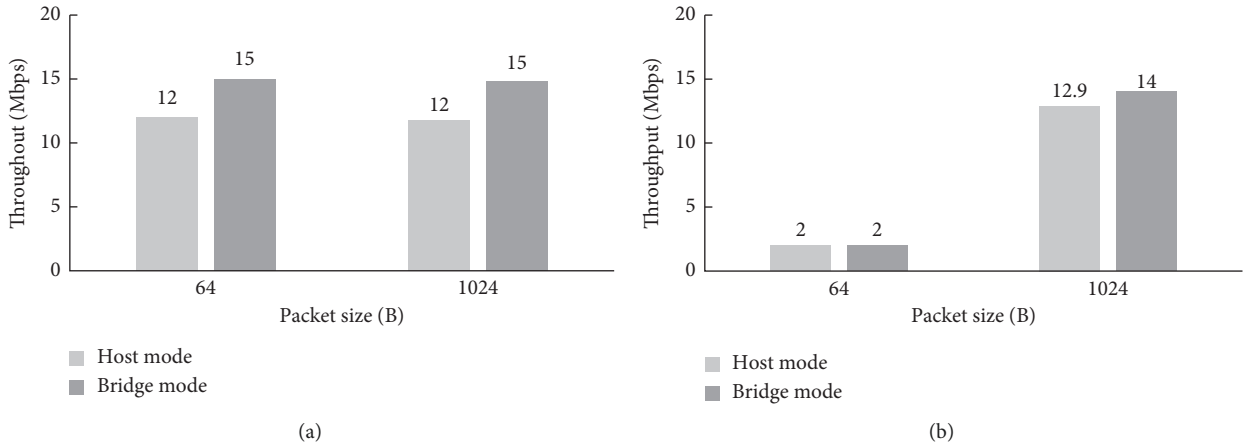

Figure 4: Network performance by different network modes in TCP (a) and UDP (b).

Table 2: System-level statistics in different network modes (protocol: TCP, packet size: 64 B).

Network mode	Bridge mode	Host mode
CPU cycles ($\times 10^6$)	972	1,050
Context switching	224	337
Page faults	13	14
Instructions/cycle	0.51	0.49

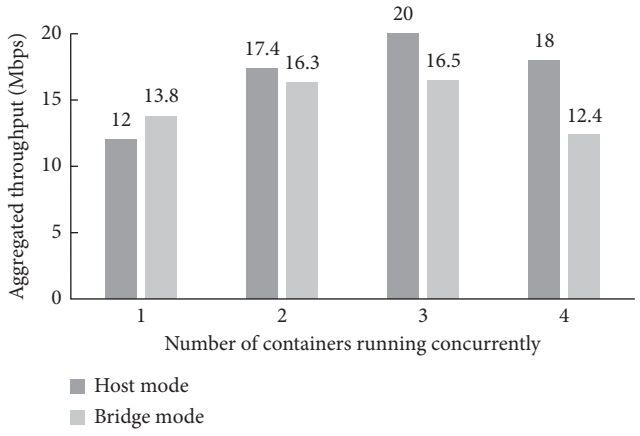

Figure 5: Aggregated throughput by number of containers.

containers are running. The reason of this performance degradation of bridge mode is that interrupt-related processing (called *bottom half*) is disabled in order to protect shared data [22] when packet transmission is performed in network interfaces. Because bridge mode assigns a virtual network interface to each container as in Figure 2, there are five network interfaces including the bridge when four containers are running while host mode does not require additional network interfaces except the actual network interface of the host OS. The additional network interfaces in bridge mode cause more *bottom half* disabled compared to host mode when transmitting packets. This brings the processing latency in network and degrades the aggregated throughput when multiple containers are running. Therefore, the network performance of bridge mode decreases compared to host mode as the number of containers increases above three.

5. Conclusion

Container-based virtualization offers several advantages, such as efficient resource utilization and scalable device management. In this paper, we present the results of analysis on the network performance of containers for IoT devices. This paper evaluates the network performance of containers on a widely used IoT device, Raspberry Pi 3. The performance evaluation results show that the network performance of containers running on an IoT device is lower than that of the native Linux by 12% in average. We also compare the network performance for different container network modes, which was not carried out on previous studies.

Our evaluation results show that bridge mode achieves higher performance than host mode when a container runs alone. On the contrary, when multiple containers running concurrently, bridge mode shows performance degradation. These results indicate that bridge mode should be configured for a single container while offering host mode for multiple containers to achieve high network performance. As future work, we plan to explore the diverse network environments such as overlay network that connects multiple containers on different IoT devices.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the SW Starlab support program (IITP-2018-2015-0-00280) and supervised by the IITP (Institute for Information & Communications Technology Promotion). This work was also partly supported by IITP grant funded by the Korea government (MSIT) (no.

2015-0-00288, Research of Network Virtualization Platform and Service for SDN 2.0 Realization).

References

[1] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (IoT): a vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1645–1660, 2013.

[2] IDC, "The Internet of Things: Data from Embedded Systems Will Account for 10% of the Digital Universe by 2020," IDC, Framingham, MA, USA, 2014, https://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm.

[3] S. K. Datta and C. Bonnet, "A lightweight framework for efficient M2M device management in oneM2M architecture," in *Proceedings of Recent Advances in Internet of Things (RIoT), 2015 International Conference on IEEE*, pp. 1–6, Singapore, April 2015.

[4] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: a survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.

[5] A. Celesti, D. Mulfari, M. Fazio, M. Villari, and A. Puliafito, "Exploring container virtualization in IoT clouds," in *Proceedings of Smart Computing (SMARTCOMP), 2016 IEEE International Conference on IEEE*, pp. 1–6, St. Louis, MO, USA, March 2016.

[6] D. Mulfari, M. Fazio, A. Celesti, M. Villari, and A. Puliafito, "Design of an IoT cloud system for container virtualization on smart objects," in *Proceedings of European Conference on Service-Oriented and Cloud Computing*, Springer, pp. 33–47, Taormina, Italy, September 2015.

[7] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 275–287, 2007.

[8] The Linux Foundation, Kubernetes, San Francisco, CA, USA, https://kubernetes.io.

[9] Docker Inc., Docker Swarm, Docker Inc., San Francisco, CA, USA, https://docs.docker.com/engine/swarm/.

[10] OpenIoT Consortium, Openiot: https://github.com/OpenIotOrg/openiot/wiki.

[11] KaaIoT Technologies, Kaa IoT Development Platform: https://www.kaaproject.org.

[12] A. Krylovskiy, "Internet of things gateways meet Linux containers: performance evaluation and discussion," in *Proceedings of Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on IEEE*, pp. 222–227, Milan, Italy, December 2015.

[13] R. Morabito, "A performance evaluation of container technologies on internet of things devices," in *Proceedings of Computer Communications Workshops (INFOCOM WKSHPS), 2016 IEEE Conference on IEEE*, pp. 999-1000, San Francisco, CA, USA, April 2016.

[14] K. Lee, H. Kim, B. Kim, and C. Yoo, "Analysis on network performance of container virtualization on IoT devices," in *Proceedings of 2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 35–37, Jeju Island, Republic of Korea, October 2017.

[15] Docker Inc., Docker, Docker Inc., San Francisco, CA, USA, https://www.docker.com/.

[16] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *Proceedings of the Tenth European Conference on Computer Systems*, ACM, p. 18, New York, NY, USA, 2015.

[17] L. Xu, G. Li, C. Li, W. Sun, W. Chen, and Z. Wang, "Condroid: a container-based virtualization solution adapted for android devices," in *Proceedings of Mobile Cloud Computing, Services, and Engineering (MobileCloud), 2015 3rd IEEE International Conference on IEEE*, pp. 81–88, San Francisco, CA, USA, March 2015.

[18] D. Drutskoy, E. Keller, and J. Rexford, "Scalable network virtualization in software-defined networks," *IEEE Internet Computing*, vol. 17, no. 2, pp. 20–27, 2013.

[19] R. Jones, "NetPerf: A Network Performance Benchmark," *Information Networks Division*, Hewlett-Packard Company, Palo Alto, CA, USA, 1996.

[20] Arnaldo Carvalho de Melo, "The New Linux perf tools," in *Proceedings of the Linux Kongress*, Nuremberg, Germany, September 2010.

[21] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proceedings of Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on IEEE*, pp. 171-172, Philadelphia, PA, USA, March 2015.

[22] *Robert Love, Bottom Halves and Deferring Work Linux Kernel Development*, Addison-Wesley Professional, Boston, MA, USA, 3rd edition, 2010.