Assessing the Value of Containers for NFVs: A Detailed Network Performance Study

Jakob Struye*, Bart Spinnewyn*, Kathleen Spaey*, Kristiaan Bonjean†, Steven Latré*
*IDLab - Dept. of Mathematics and Computer Science - University of Antwerp - imec
Middelheimlaan 1, 2020 Antwerp, Belgium, firstname.lastname@uantwerpen.be
†Newtec, Laarstraat 5, 9100 Sint-Niklaas, Belgium

***Abstract*—Since its introduction in 2012, telecommunications operators have been applying the Network Function Virtualization principle to their core infrastructure, leading to more agile and cost-efficient deployments. While these Virtualized Network Functions (VNFs) are traditionally implemented using Virtual Machines (VMs), efforts are starting to shift to containerized VNF implementations, further improving agility and cost-efficiency. Furthermore, telecom applications often require extreme networking performance in terms of throughput and latency. While research has shown that containers outperform VMs on this front, it is currently unclear how the choice of container provider influences network performance. In this paper we compare the networking performance of Linux container implementations Docker, rkt and LXC. Throughput and latency are evaluated for single-host *host*, *bridge* (or *NAT*) and *macvlan* network configurations. This is, to the best of our knowledge, the first comparison featuring all three major Linux container implementations. We show that LXC performs best, with Docker and rkt showing throughputs of respectively up to $35\,\%$ and $58\,\%$ lower. Of the considered networking implementations, the macvlan network performs best. While it experiences a significant performance degradation when many containers are chained together, a single container using macvlan can outperform even a bare metal implementation when enough CPU resources are available.**

I. Introduction

While the concept of Linux containers has been around for almost 20 years, it has only gained massive traction in the computer science world over the past few years. Since the introduction of the Docker container in 2013, containers have been widely adopted. According to monitoring service company Datadog, $10\,\%$ of the systems on which their software is installed were running Docker in June 2016 [1]. Containers are often considered to be an alternative to Virtual Machines (VMs). While VMs create full virtualized environments including kernel, containers take a more lightweight approach, reusing the host machine's kernel and libraries. With this approach, resources are used more efficiently due to overhead reduction [2][3][4]. As an effect containers spin up faster and more can be deployed on one machine compared to VMs.

One application of containers is building Virtualized Network Functions (VNFs). With Network Function Virtualization (NFV), physical middleboxes such as firewalls and load balancers are replaced by software implementations running on general purpose hardware. Scaling up a deployment using traditional middleboxes is slow and expensive as it requires buying additional specialized hardware. In an NFV-based deployment, operators can simply deploy additional VNFs quickly. While traditionally implemented using VMs, a recent shift to container-based VNFs leads to even faster deployment and even more efficient resource usage. Telecommunications companies (telcos) have deployed large-scale NFV-based infrastructures [5]. These VNFs are usually required to have a consistently low latency. In addition, higher per-instance throughput leads to cheaper deployments.

As with VMs, multiple container providers are active on the container market. A survey performed in April-May 2016 by DevOps.com and ClusterHQ provides some insight into the usage of different container providers [6]. The most popular container provider by far is Docker, used by $94\,\%$ of 218 respondents. LXC and rkt are used by a significantly lower yet non-negligible $15\,\%$ and $10\,\%$ of respondents. All other providers combined are also only used by $5\,\%$. It is currently unclear how significantly the choice of container provider influences network performance. Each provider offers their own implementation of several networking solutions. Choosing a less performant container provider would lead to more expensive deployments.

In this paper we compare the networking performance of Docker, rkt and LXC containers as a means to study the performance of future NFV environments. We analyze throughput and latency for the three most common networking solutions for containers: *host*, *bridge* (or *NAT*) and *macvlan* networking. We compare the performance when sending small and large packets using one or two container hosts. While existing research does compare the performance of these networking solutions, experiments usually consider only one container provider. This paper instead contributes a detailed comparison of all three major container providers' network performance, providing insight into their relative performance.

The outline of this paper is as follows. We summarize related work in Section II. In Section III we provide a brief overview of how containers work. Section IV contains a review of how container networking may be implemented. In Section V we present our experiments, along with the results in Section VI. Finally Section VII concludes this paper.

978-3-901882-98-2

II. Related Work

Claassen et al. compared the performance of the bridge, macvlan and ipvlan networks using Docker [7]. They deployed 1 to 128 container pairs, where one container in the pair sent a continuous TCP flow to the other. This was performed with all pairs on one machine with all traffic going via a router. Only the TCP throughput is reported in the paper. With all containers on one host, macvlan and ipvlan showed similar throughput, outperforming the bridge network by a factor 2.5 to 3. Next, Anderson et al. investigated the usability of different Docker container networking implementations for NFV applications [3]. The authors compared the performance of multiple network implementations including bridge networks and macvlan. The authors gathered numerous metrics by sending traffic through a chain of containers using `Netperf`. Compared to a host network implementation, the additional latency of a packet passing through macvlan and bridge networks was $1.0\,\%$ and $3.2\,\%$ respectively. Round-trip times for the bridge network were $16.7\,\%$ higher than with the macvlan network. Overall macvlan was found to be the most performant solution.

Two papers did include a brief comparison of Docker and LXC networking performance as part of a more general comparison. Morabito et al. gathered throughput and latency measurements for bridge networks using `Netperf` [8]. The authors compared the containers' performance to a bare metal implementation. Both Docker and LXC saturated the $10\,\mathrm{Gbps}$ link using TCP traffic, but saw a throughput decrease of respectively $42.97\,\%$ and $42.14\,\%$ using UDP. TCP latency increased by $19.36\,\%$ and $17.35\,\%$ for Docker and LXC, while UDP latency increased by $12.13\,\%$ and $10.82\,\%$. This is a first indication that LXC is slightly more performant than Docker. Kozhirbayev and Sinnott performed similar measurements, also using `Netperf` [9]. Docker's throughput was $7.5\,\%$ (UDP) to $10.0\,\%$ (TCP) lower than with a bare metal implementation. With LXC this loss increased to $24.9\,\%$ to $25.8\,\%$. The similar `iPerf` tool led to very different results, with TCP throughput decreases of $43.9\,\%$ for Docker and only $18.4\,\%$ for LXC. The authors suspect that the TCP buffer size may be responsible for the diverging results between the two tools. We note that in both papers the authors only investigated one networking solution with one packet size and one container per host.

Many papers have compared the network performance of multiple networking solutions for one container provider, often also including a comparison with VMs or bare metal. Apart from some limited comparisons covered above, none compare the performance of different container providers. As such, we are, to the best of our knowledge, the first to contribute a detailed comparison between all three major container providers. With host, bridge and macvlan networking, we consider the three main single-host networking solutions for containers.

III. Linux Containers

The type of *operating-system-level virtualization* we focus on is known as *Linux containers* or simply *containers*. This technology is based on two kernel features: *namespaces* and *cgroups*. Namespaces are a method of limiting what resources are visible to a set of processes [10][11]. Each set of resources has a namespace type associated with it. The most important one for our research is the `network` type. Every `network` namespace has its own separate network stack. Processes within the namespace can only see that network stack, and have no access to other namespaces' network devices. Cgroups on the other hand handle resource guaranteeing, limiting, throttling and accounting. These are again divided into separate *subsystems* or *controllers*. Each subsystem has control over one type of resources. For example, the `memory` cgroup can track and limit memory usage. Overall, namespaces limit what resources a process can see, and cgroups limit how much of those resources a process can use. Using these features, the system at large can be hidden from a set of processes. Those processes are then running in a container. We consider the following three container implementations:

1) LXC: The oldest of the modern Linux container implementations, having its original release in August 2008, soon after crucial parts of Linux namespaces and cgroups were added to the kernel [12]. LXC containers are mainly intended to be used as full-fledged machines, rather than running only one application. While the LXC API provides a simple way to download container templates for many popular Linux distributions, there is no convenient way to obtain a container with a specific application pre-installed.

2) Docker: By far the most popular implementation of modern Linux containers [13]. For many, the name Docker is synonymous with containers. Since its initial release in May 2013, Docker has grown to be an ecosystem with many components for building, running, managing, storing and scaling container-based applications [14]. Docker focuses on a one-app-per-container design. To enable this, there are a number of online registries providing pre-built application images, of which users can upload their own. While Docker initially used LXC as the underlying driver for the containers, a custom driver named *libcontainer* was added as the default driver in March 2014 [15].

3) rkt: The third implementation we will consider is rkt, initially released in December 2014 by CoreOS, a company best known for its container-focused Linux distribution *Container Linux* [16]. The CoreOS team created rkt as a reaction to Docker becoming a full-fledged platform, rather than a building block for containerization easily integrated with other tools. Docker images can be converted automatically to a rkt-compatible image format through a tool called docker2aci [17].

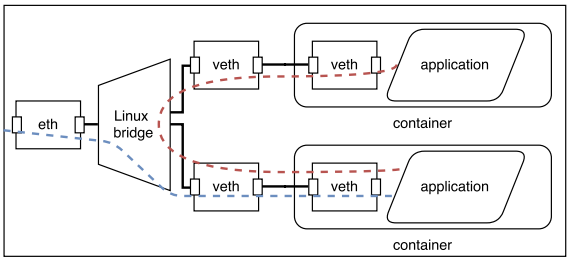

Fig. 1. Bridge network with traffic flows

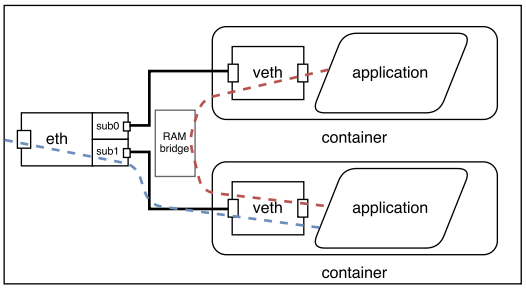

Fig. 2. Macvlan network with traffic flows

IV. Container Networking

As VNFs are in essence packet handling applications, their networking performance is paramount. All major container providers implement multiple networking solutions with different levels of isolation, flexibility and performance. In this section we cover the three most common solutions.

1) Host Network: The host network is by far the simplest solution. The container simply uses the host machine's network stack. As a container can overhear network traffic of other containers using the host network and even of the host itself, this solution clearly offers no network isolation. Performance-wise, host networks incur no virtualization overhead.

2) Bridge Network: A regular Linux bridge is created on the host [18]. Every time a new container joins the bridge network a new pair of virtual ethernet (veth) interfaces is created, with one end connected to the bridge, and the other to the container. All interfaces except for its veth interface are hidden from the container using `network` namespaces. The container is assigned a private IP address, useful for communication between colocated containers. Packets between such containers pass through four veth interfaces and a bridge, as seen in Fig. 1. Outgoing traffic is masqueraded with the host's IP address. To make a container's port easily addressable from outside, the port has to be *forwarded* or *exposed*. This can be achieved through a number of `iptables` NAT rules. Docker and rkt can configure these automatically, while LXC requires manual configuration.

3) Macvlan: This option uses the kernel's macvlan technology, initially introduced in 2007 [18]. Macvlan creates multiple virtual sub-interfaces on a physical interface. Each of these sub-interfaces is assigned a random MAC address and an IP address in the physical interface's subnet. Containers are addressable from outside the host through this IP address. While macvlan supports four different modes, we only consider the *bridge* mode. This mode creates a *pseudo bridge* between the different sub-interfaces. Packets sent between sub-interfaces are simply copied in-memory, and never leave the host machine. The traffic flows for this mode are shown in Fig. 2. The other modes do not allow inter-container communication or require specialized hardware.

We note that the Data Plane Development Kit (DPDK) libraries [19] could significantly increase performance of the networks we consider [20][21]. As there are, to the best of our knowledge, no DPDK-specific optimizations for the container technologies we consider, we do not incorporate DPDK into our experiments.

V. Experimental Setup

In this section we describe the experimental setup. Our network performance tests are based on a basic containerized packet processing VNF. Note that *processor* always refers to this VNF in this paper. We measure the performance both when packets pass through a single processor, and when they traverse multiple processors in sequence. The latter is performed both with all processors on one physical machine, and distributed across two machines on the same subnet. When two machines are involved, the packets are *ping-ponged* between the machines. In all tests we send UDP traffic to determine the attainable Ethernet packets per second (pps) throughput. All tests are repeated for small (28 B) and large (1454 B) UDP payload sizes. The processor is implemented using `socat`, which listens for UDP traffic on a port and forwards all received UDP payloads in new datagrams. Table I provides an overview of the used devices and software versions. Fig. 3 shows the setup. Packets originate at the Ixia chassis, are sent to host A, possibly *ping-ponged* between host A and host B 10 times and finally sent from host A back to the Ixia chassis.

We performed experiments with one processor on one host, twenty processors on one host and twenty-one processors alternating between two hosts. We performed every experiment with small and large packet sizes, for every combination of container implementation and network solution, along with reference bare metal experiments. Each of these 60 arrangements was run for 3 minutes. We consider the following metrics:

- **Average packets per second** (pps): We measure the throughput in pps instead of in payload bytes. As many VNFs only read packet headers and ignore payloads, the total packet length only has a minor influence on packet processing time, making pps a more useful metric.

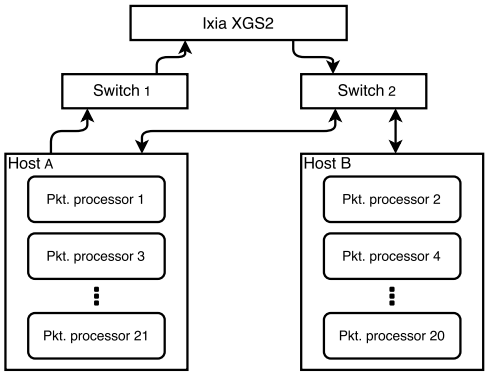

Fig. 3. Network performance setup

TABLE I
VERSION AND MODEL INFORMATION

Operating system (CentOS)	7.3.1611
Kernel	3.10.0-514.10.2.el7
Docker	17.03.0
rkt	1.21.0
LXC	2.0.7
Container hosts	HP ProLiant BL460c Gen9 Servers [22]
CPU	Intel® Xeon® E5-2640 (2.4 GHz, 10 cores, hyper-threading disabled)
Memory	64 GB
Packet generator	Ixia XGS2 chassis [23]
Network interfaces	1 Gbps

- **Store-forward latency**: The interval between the final byte of a packet leaving the Ixia chassis and the first byte of that packet again arriving at the chassis. We report the minimum, maximum and average values.

VI. RESULTS DESCRIPTION

A. Average packets per second

We first analyze the average pps rate as reported by the Ixia chassis for every configuration. Containerized performance is represented by bars while the bare metal performance is indicated with a horizontal line.

1) One host, large packets: A gigabit link could theoretically transmit up to 83333 large packets per second. Fig. 4 shows the measured average pps. Both bare metal experiments reached 98.6 % of this maximum. Both Docker and LXC were also able to saturate the link with one processor with host and macvlan networking. With bridge networking LXC also saturated the link, even with 20 processors. In the bridged arrangements Docker could not saturate the link even with only one processor. With 20 processors it saturated only one third. Rkt's performance never came close to line rate, reaching only 89 % of the theoretical maximum for the single processor host networking arrangement. Rkt's highest pps rate was considerably worse than LXC's lowest. Overall, performance for host and macvlan networking was very similar.

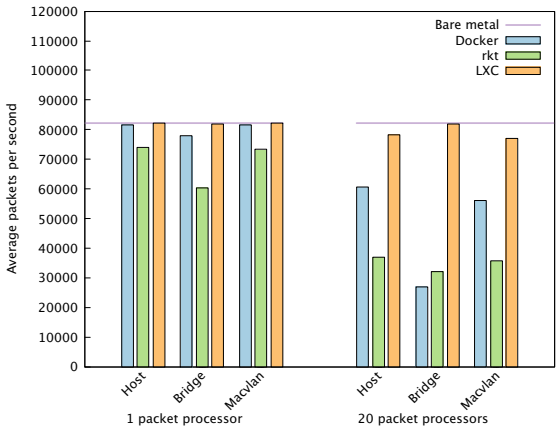

Fig. 4. Packets per second for single host, large packet size

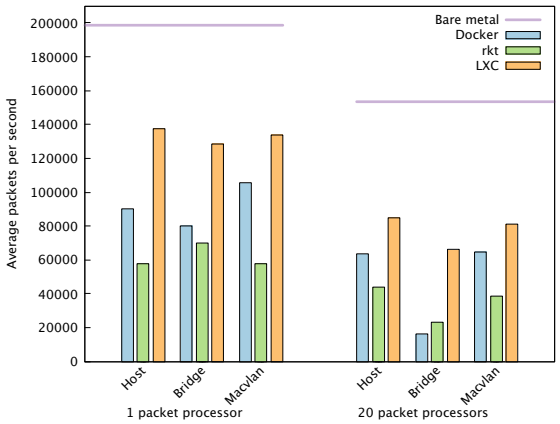

Fig. 5. Packets per second for single host, small packet size

2) One host, small packets: Next we consider the experiments with small, 74 B Ethernet packets in Fig. 5. As we noticed that no arrangement could handle a gigabit stream, we reduced the sending rate to 25 % of line rate. The best performance was attained by the single processor bare metal arrangement, reaching 200 000 pps. Note that, while packets were over 20 times smaller compared to the previous experiment, the pps rate was only 2.4 times as high, indicating these arrangements were bottlenecked by CPU power. Increasing the processor count to 20 reduced the pps by another 23 %, showing non-negligible processor overhead. The containerized results experience additional performance loss. LXC was again clearly the most performant of the three. Single processor pps was 31.6 % to 35.2 % lower than the bare metal case. With 20 processors this rose to 44.6 % to 58.5 %. The additional overhead of the bridge network became more noticeable here: the NAT rules were applied 20 times for both incoming and outgoing packets instead of only once. Docker again performed considerably worse compared to LXC. Notably, Docker host networking was 14.5 % slower than macvlan with 1 processor. Rkt was at worst 45.1 % slower than Docker and 58.1 % slower than LXC. Overall, the host

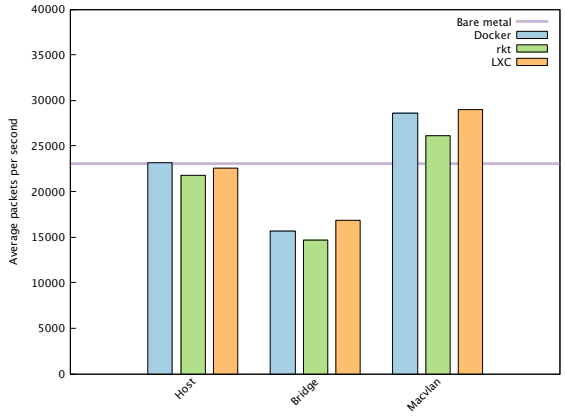

Fig. 6. Packets per second for two hosts, small packet size

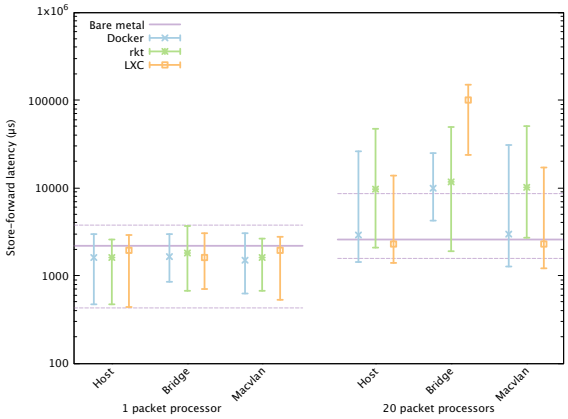

Fig. 7. Store-forward latency for single host, large packet size

network and macvlan performed comparably, with the bridge network showing considerable additional overhead, especially with many processors.

3) Two hosts, large packets: With two hosts and large packets the gigabit links were saturated in all cases. The theoretical maximum pps was 11 times as low as with the single host test, as some links see every packet 11 times in one direction. Every single tested arrangement was only $5.5\,\%$ to $6.3\,\%$ slower than the theoretical maximum, implying all were bottlenecked by the links' capacities.

4) Two hosts, small packets: With packets over 20 times as small, in Fig. 6, the pps rate increased by only a factor 3 for the bare metal arrangement, indicating the CPU was again the bottleneck. Differences between container implementations were not as significant as in previous tests. Host network performance was very close to bare metal performance. Rkt again performed slightly worse than the other two. With the bridge network performance took a hit of up to $36.6\,\%$. Surprisingly, macvlan managed to outperform the bare metal arrangement by up to $25.9\,\%$. We suspect this is due to arriving packets for the different processors being stored in separate queues instead of one large shared queue. Note that, in this configuration, each host in essence behaves like one large multithreaded processor. The incoming stream of packets is divided equally across all cores. This indicates that, as long as one packet is only handled by one to a few containers and the containers have access to enough CPU power, the networking overhead due to containerization is minimal, and may in fact even be negative.

B. Store-forward latency

Next we analyze the store-forward latencies. All latencies are plotted using logarithmic scales. Containerized latencies are represented by bars ranging from the minimum to the maximum pps, with the average indicated. Solid and dashed lines indicate the bare metal averages and extrema, respectively.

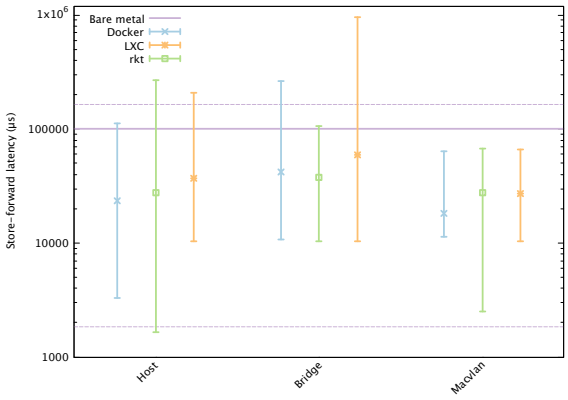

Fig. 8. Store-forward latency for two hosts, small packet size

1) One host, large packets: We again first look at the single-host experiments with large packet sizes, in Fig. 7. The bare metal, single processor arrangement reached an average latency of $2.2\,\mathrm{ms}$, while all containerized arrangements experienced lower average latencies, between $1.5\,\mathrm{ms}$ and $2.0\,\mathrm{ms}$. Moving to 20 processors only slightly increased the bare metal latency to $2.6\,\mathrm{ms}$. This indicates that most of the latency was caused by sending the data over the links. Docker and LXC showed a similar increase with host and macvlan networks. As with the pps, rkt again performed considerably worse, reaching average latencies of $9.6\,\mathrm{ms}$ and $10.0\,\mathrm{ms}$. LXC performed extremely poorly with bridge networking, reaching an average latency of over $100\,\mathrm{ms}$. Note that LXC is the only container implementation for which we had to configure `iptables` rules manually, as the other two handled this automatically.

2) One host, small packets: Latency results for the small packet sizes were largely similar to the previous case. Bare metal latency with one processor decreased significantly from $2.2\,\mathrm{ms}$ to $1.2\,\mathrm{ms}$, now outperforming the containers. Due to the smaller packet size, copying the payload was

TABLE II
THROUGHPUT LOSS COMPARED TO BARE METAL

		Docker	Rkt	LXC
One host, large packets	1 proc.	0 % to 5 %	10 % to 26 %	0 %
	20 proc.	26 % to 67 %	55 % to 61 %	0 % to 6 %
One host, small packets	1 proc.	47 % to 60 %	65 % to 71 %	31 % to 35 %
	20 proc.	58 % to 89 %	71 % to 85 %	45 % to 57 %
Two hosts, large packets		0 %	0 %	0 %
Two hosts, small packets		−24 % to 32 %	−13 % to 36 %	−26 % to 27 %

faster. Furthermore the transmission delay (i.e. time between first and last bit entering a link) is significantly lower for small packets. With 20 processors however, latency increased considerably to 5.5 ms. We suspect that this is due to queueing delays at the processors. For Docker and rkt with 1 processor, latency increased by 55 % to 160 % compared to the large packet arrangements. LXC suffered less from the smaller packets: bridge network latency increased by 37 % while host and macvlan network latencies managed to decrease by 8.5 % and 6.1 %. With 20 processors, LXC with bridge network again showed an extremely high average. All other containerized 20 processor arrangements saw an increase in latency of 137 % to 420 %. The relative differences between the container implementations remained similar for each network type.

3) Two hosts, large packets: As with the throughput, all results for two hosts with large packets were very similar. Every average latency was between 16.8 ms and 17.5 ms, with maxima up to only 25.7 ms.

4) Two hosts, small packets: With small packets and two hosts in Fig. 8, the bare metal arrangement again performed poorly compared to containers. The bare metal arrangement reached a latency of over 100 ms. With an average of 59.5 ms and a maximum of nearly 1 s, LXC with bridge networking again performed worst of all containerized arrangements. The other arrangements reached average latencies between 18.2 ms and 42.2 ms. Compared to the large packets, latency increased by 7.8 % to 142 %.

In summary, rkt performs worst in most cases, with LXC being most performant in nearly all cases. Table II summarizes this for the pps. When line rate serves as a bottleneck, container performance is indistinguishable from bare metal performance. Of the three network types, bridge usually performs worst. With only one host machine, host and macvlan networks perform similarly, while macvlan takes a clear lead when ping-ponging packets between two hosts. Overall we consider macvlan as the best choice for most applications. While LXC is usually more performant than Docker, LXC containers and networks take more effort to deploy, configure and maintain.

VII. CONCLUSION

While containers are already sporadically used in NFV-based applications, it is unclear how performant different container providers are from a networking perspective. Especially in large-scale microservice-based applications, such as those operated by telcos, choosing the most performant container and network combination could lead to a significantly cheaper and faster configuration. In this paper we compared the networking performance of Docker, rkt and LXC, the three major Linux container providers. We compared the performance of host, bridge and macvlan networking for all three providers. We measured the performance by running one or more containerized UDP datagram forwarding VNFs on host machines. We discovered that rkt offers the worst performance, with LXC being most performant. With LXC, one single-core packet processing VNF suffices to handle a gigabit stream of 1500 B packets. With small 78 B packets, LXC throughputs were 30 % to 40 % lower than bare metal with host and macvlan networking, and 30 % to 66 % lower with bridge networking. With 10 packet processing VNFs each handling $1/10^{th}$ of incoming 78 B packets, bridge networking loses about 30 % throughput. With host networking the loss is negligible, and macvlan networking actually gains over 20 % in throughput, possibly due to macvlan networking using a separate buffer for every processor. We conclude that macvlan is the best choice of the three when high throughput is required. With one processor handling all incoming data this leads to an increase in latency of 50 % with LXC, 100 % with Docker and 240 % with rkt. With packet processing distributed over multiple applications, all container providers and networking solutions improve latency compared to the bare metal case, by up to a factor 5. Overall, we found that LXC with macvlan is the most performant solution. For a telco aiming for the most cost-efficient and performant configuration, this combination is the most advantageous. While LXC usually requires more effort to deploy and maintain than Docker or rkt, the performance gain of using LXC is significant.

ACKNOWLEDGMENT

We would like to thank Newtec [24] for granting us access to an Ixia XGS2 and other hardware for our setup.

REFERENCES

[1] Datadog, "8 Surprising Facts About Real Docker Adoption," https://www.datadoghq.com/docker-adoption/, 6 2016, [Online; accessed 4-April-2017].

[2] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2013, pp. 233–240.

[3] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *2016 International Conference on Computing, Networking and Communications (ICNC)*, Feb 2016, pp. 1–7.

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

[5] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, "Nfv: state of the art, challenges, and implementation in next generation mobile networks (vepc)," *IEEE Network*, vol. 28, no. 6, pp. 18–26, Nov 2014.

[6] ClusterHQ and DevOps.com, "Container Market Adoption Survey 2016," https://clusterhq.com/assets/pdfs/state-of-container-usage-june-2016.pdf, 2016.

[7] J. Claassen, R. Koning, and P. Grosso, "Linux containers networking: Performance and scalability of kernel modules," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 713–717.

[8] R. Morabito, J. Kjllman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *2015 IEEE International Conference on Cloud Engineering*, March 2015, pp. 386–393.

[9] Z. Kozhirbayev and R. O. Sinnott, "A performance comparison of container-based technologies for the cloud," *Future Generation Computer Systems*, vol. 68, no. Supplement C, pp. 175 – 182, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X16303041

[10] R. Rosen, "Resource management: Linux kernel Namespaces and cgroups," http://www.haifux.org/lectures/299/netLec7.pdf, 2013, haifux 2013.

[11] ——, "Namespaces and cgroups: the basis of Linux Containers," http://www.netdevconf.org/1.1/proceedings/slides/rosen-namespaces-cgroups-lxc.pdf, 2016, netDev 1.1: The Technical Conference on Linux Networking.

[12] C. Ltd., "Linux Containers - LXC - Introduction," https://linuxcontainers.org/lxc/introduction/, 2017, [Online; accessed 16-February-2017].

[13] D. Inc., "What is Docker?" https://www.docker.com/what-docker, 2017, [Online; accessed 16-February-2017].

[14] ——, "Docker Documentation - Docker," https://docs.docker.com/, 2017, [Online; accessed 16-February-2017].

[15] S. Hykes, "Docker 0.9: Introducing Execution Drivers And Libcontainer," https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/, 2014, [Online; accessed 16-February-2017].

[16] A. Polvi, "CoreOS is building a container runtime, rkt," https://coreos.com/blog/rocket.html, 2014, [Online; accessed 16-February-2017].

[17] A. Container, "docker2aci," https://github.com/appc/docker2aci, 2017.

[18] H. Cube, "Bridge vs Macvlan," 2016, [Online; accessed 14-June-2017]. [Online]. Available: http://hicu.be/bridge-vs-macvlan

[19] T. L. Foundation, "DPDK," http://dpdk.org, 2017.

[20] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "A study of network stack latency for game servers," in *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games*. IEEE Press, 2014, p. 15.

[21] R. Kawashima, S. Muramatsu, H. Nakayama, T. Hayashi, and H. Matsuo, "A host-based performance comparison of 40g nfv environments focusing on packet processing architectures and virtual switches," in *Software-Defined Networks (EWSDN), 2016 Fifth European Workshop on*. IEEE, 2016, pp. 19–24.

[22] H. P. Enterprise, "HPE ProLiant BL460c Gen9 Server Blade," https://www.hpe.com/us/en/product-catalog/servers/proliant-servers/pip.hpe-proliant-bl460c-gen9-server-blade.7271227.html, 2014.

[23] Ixia, "XGS2 Chassis Platform," https://www.ixiacom.com/products/xgs2-chassis-platform, 2015.

[24] Newtec, "About Newtec - Newtec," http://www.newtec.eu/about-newtec, 2015.