RunC and Kata runtime using Docker: a network perspective comparison

Henrique Zanela Cochak, Guilherme Piêgas Koslovski, Maurício Aronne Pillon, Charles Christian Miers
Graduate Program in Applied Computing – Computer Science Department – Santa Catarina State University
henrique.zc@edu.udesc.br, {guilherme.koslovski, mauricio.pillon, charles.miers}@udesc.br

***Abstract*—The container technology allows packing applications, dependencies, and configurations on a single abstraction, which can be instantiated independently of hosting platform or hardware. Among the existing containers platforms, the present work focus on Docker, specifically the module function runtime *runC*, public available under the Open Container Initiative (OCI), is compared to *kata* open-source runtime. In short, our work presents an in-depth comparison between both runtimes highlighting the communication support. Specifically, our comparison comprehend different container network drivers, using as criteria network bandwidth and latency. The results show an already better expected runC performance, but also reveal a kata baseline, and help to understand how to choose the communication driver. However, kata still offers a better level of security and isolation.**

***Index Terms*—containers, docker, kata, runC, networking**

I. Introduction

The improvement of new technologies in virtualization has shown a significant increase in cloud computing. Clouds become the delivery model for computing services, e.g., networking, software, analytic, servers, and storage all over the Internet, offering faster innovation with flexible resources implying economies of scale [1].

The virtualization technology allows to create multiple virtualized environments using a single physical hardware. This is achieved by the hypervisor, which coordinates the sharing of the physical resources of the computer, serving as an interface between the virtual machines (VMs) and the adjacent physical hardware, certifying that each VM has access to the resources it needs to execute properly [2].

Containerization emerges as an evolution of this classic virtualization of machines. Containers are an executable unit of software, in which application code, libraries, dependencies and configuration files are packaged together in order run it anywhere [3].

Related to containerization management platforms, there is Docker, which is an open Platform-as-a-Service (PaaS)/ Infrastructure-as-a-Service (IaaS) for developing, shipping, and running containerized applications [4]. Inside its modular platform, there is a lightweight universal container runtime denominated runC. The runtime purpose is to run containers using the command line, all according to OCI standard. A new approach to it is the runtime kata, which focus primarily on security issues based on a lightweight VM which holds a container inside, bringing up the benefits of containers and the the security of VMs.

978-1-6654-4035-6/21/$31.00 ©2021 IEEE

We aim to analyze the runC and Kata runtimes regarding the security vs. performance tradeoff for network communication. We define test scenarios and use the iPerf tool to generate traffic. Data was collected and analyzed regarding in how the extra security layer of the Kata impacts its performance against runC.

The article is organized as follows: a brief description about how Docker provides the basic network communication for each container and how the runC runtime functions is presented on Section II. How Kata Containers works and provides the network communication in Section IV. An explanation about the Linux interfaces used in both runtimes in Section V. Details about the experiments and results are shown in Section VI. Section VII addresses the related work.

II. Docker container networking

Docker container technology was first released in 2013 as an open source PaaS, levering concepts from Linux like cgroups, namespaces and unionFS. In 2015 a governance council calledOCI was created and it is responsible to develop standards to the containers infrastructure, developing the Container Network Model (CNM).

The CNM document shows steps to supply traffic and communication through the network to the containers while being abstract enough to support a variety of network drivers [5]. It is build on three components:

- sandbox: A sandbox is an isolated network stack, containing its configurations, managing routing tables, Domain Name System (DNS) settings, ports, and the containers interfaces. A sandbox may contain several endpoints from multiple networks.
- endpoint: An endpoint works as virtual network interface, and it has the attribution of establish a connection from a sandbox to a network. An implementation of an endpoint could be a Virtual Ethernet (vEth) pair and an endpoint. It can belong only to one network and one sandbox at same time.
- network: A network is a software implementation of 802.1d bridge (switch), and as such they group up and isolate endpoints.

The libnetwork library is the CNM implemented as a module inside Docker [5] so it dispatches all the three components from the CNM document, calling built-in network drivers

which come with Docker Engine and the assistance of its own pluggable interface. Docker can arrange the network communication between numerous containers automatically by plugging in the built-in drivers (Figure 1).

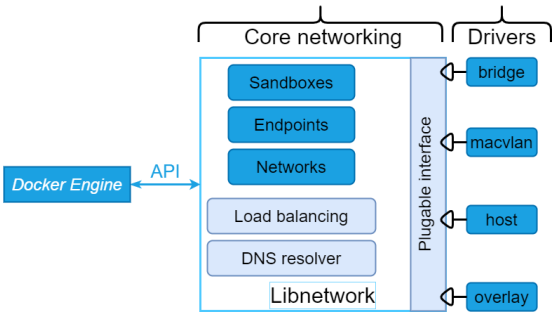

Fig. 1. Libnetwork implementation by Docker

This library still provides some other important network services such as load balancer, a function to help deliver traffic management services more efficiently, mostly used in the Docker swarm mode. It also has a service discovery, meaning containers can discover one another when it are on the same network by name, embedding an inside DNS server.

III. RUNC RUNTIME

Another specification made by OCI is related to runtimes. It exists with the sole purpose of containing the life cycle of a container, including configuration files and the execution environment, whereupon is specified to ensure that containerized applications don't fail between common actions also defined inside the life cycle [6].

The default module option for Docker related about container life cycle is the runC runtime. RunC works as a command line client for running applications packaged according to the OCI format, and is a compliant implementation of the OCI specification [7]. Since Docker is a container technology, it takes advantage of the Linux operating system (OS) virtualization functions of namespaces / cgroups to create the basic container isolation as well as to control the amount of resources each container have access to. Thus, Docker has an binary file called dockerd or Docker Engine whom provides access to create the containers by requesting access to specific Linux kernel features.

Dockerd being a modular application includes other functionalities separated in other binary files which are needed to initialize containers and to make the proper network connection for them. The binary component files are the runtime, containerd, and shim. The containerd is an industry-standard high level container runtime available as a daemon for Linux and Windows, which can manage the complete container life cycle of its host system [8]. Regarding the shim, it is located between the dockerd and the low level runC runtime to facilitate bidirectional communication for the container and also allowing for containers to be daemonless.

In order to a container be created, accessed, and properly configured inside the host network, the dockerd has a command line providing some specifications for the container. The containerd is called by the dockerd process to start the construction, isolation and configuration of the container. Figure 2 exemplifies all the communication between processes, libraries, and kernel to provide a container to the system.

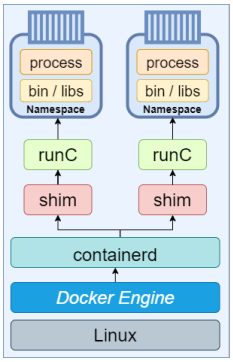

Fig. 2. Docker architecture based on runC

Containerd generates sub calls for modules from the runtime and the shim. The runtime uses the libnetwork to access kernel features, configuring the container, setting up network stack with endpoints, connecting the container endpoints to a network, IPtables rules, and DNS properties if necessary. Finally, the runtime provides a link to the containerd with the shim, now holding all the input and outputs of the container.

IV. KATA CONTAINERS

Due to Docker yielding open source access to [9] with it own container format and runtime to the OCI, it has allowed several new studies to start using this format and technology. Among one of these studies was the Kata Containers in 2017, an open source container runtime proposing to increase container security by allocating the container inside a lightweight VMs. The default kernel provided in Kata Containers was highly optimized for kernel boot time and minimal memory footprint, providing only those services required by a container workload. By using the traditional container technology to create and isolate containers and adding a highly optimized kernel as a guest kernel [10] with a complete hardware virtualization interface, it is possible to have an extra layer of encapsulation (Figure 3). Thus, increasing the security level of the host system.

The containers don't share the same kernel, and are even more isolated being inside a set of a virtualized hardware and adding the container inside the Kernel-based Virtual Machine (KVM) VM with its own small Linux kernel provided by the kata runtime. When the VM is created, the container process is spawned inside it by the kata agent and it works similarly as a daemon for the container. The agent runs a gRPC Remote Procedure Calls (gRPC) server with a virtual socket interface using QEMU to expose as a socket on the host network. Thus, any container using the kata runtime have all commands within the container, and input/output streams, passing through the virtual socker exported by the QEMU [10].

Kata runtime needs extra steps to provide network communication when compared to Docker. Since the container is hold

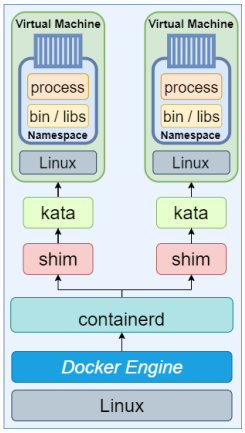

Fig. 3. Kata container architecture

inside a VM, TAP interfaces are needed for the VM network connectivity which the single vEth pair cannot be handled properly by the hypervisor [10]. In order to overcome this container-VM incompatibility, Kata containers connects vEth interfaces with TAP interfaces using traffic control (Figure 4), redirecting the packets based on traffic control rules.

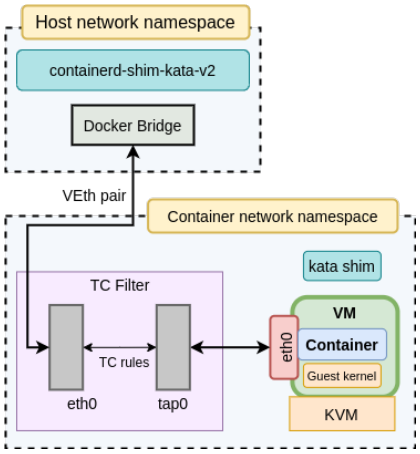

Fig. 4. Kata container network

Those extra configurations, and some packet redirecting and mirroring, allow Kata containers to provide a solid network communication for the containers.

V. Network Drivers

Docker use Linux network interfaces by requesting them to the kernel using the libnetwork library. It has chosen five drivers to be part of its pluggable subsystem by default which supply all the core networking functionalities:

- bridge: A virtual bridge can be seen as a switch, forwarding the packets between interfaces, allowing containers to the same bridge network to communicate while providing a level of isolation. The communication using bridge use Linux IPtables to set the Network Address Translation (NAT) rules to map properly the container IP and port between the host IP address and port.
- none: A type of network driver mainly focused on security and isolation. It does not configure any IP address for the container, thus the container is not visible in the network but the communication between containers using this type of driver still happen employing the loopback interface and Interprocess Communication (IPC) [11].
- host: The container network stack is not isolated from the host network namespace. It shares the host network stack and so it removes the container security layer containers are build in, making it the least secure build-in driver available with the best network performance when associated with throughput [12].
  Some features includes better performance since in a network perspective, it provides navigation through the host connection and doesn't require any forwarding and NAT, only a port from the host network. Since it uses the host network stack, each container using the host driver will not be configured any IP or Media Access Control (MAC), but using the same from the host machine. The communication for the containers inside the same host network uses IPC [11].
- macvlan: A network driver able to configure multiple Layer 2 MAC addresses and IP to each container Virtual Network Interface (VNI), creating the ability to be seen as a physical device in the network [12]. This driver requires the host interface to be linked with the macvlan driver in promiscuous mode, meaning the Network Interface Controller (NIC) will accept every Ethernet packet sent on the network, allowing the host to read packets intended for other network devices.
  There are two modes available: bridge mode and trunk bridge mode. The first one allows all the traffic to be through a physical device on the host while the latter uses sub-interfaces, aligning with the 802.1q Virtual LAN (VLAN), defined by tagging VLAN IDs and carrying up to 4096 sub-interfaces connected to the host. The second one traffic goes through an 802.1q sub-interface created by Docker.
- overlay: This driver creates a distributed network among Docker daemons, creating a Swarm which allows the communication from different hosts inside the data link layer. It works by creating a bridge, available only for the containers inside the Swarm, a default ingress network interface and a Virtual Extensible LAN (VXLAN) tunnel. A VXLAN Tunnel Endpoint (VTEP) is connected to each side of the tunnel and the bridge. When packages or frames are in transition inside the inbound network, whenever they are mapped to the MAC address of the VTEP, the inbound network changes the header by adding an identifier called the VXLAN network ID and encapsulates the data to be sent over the underlay infrastructure. After the transportation, the data is de-encapsulated and sent accordingly to its destiny.

Since kata runtime follows the standardization of CNM, all the functionalities used in standard runC of Docker will work

also on kata runtime. However, kata runtime is not possible to use of the host network driver (details on Subsection VI-C).

VI. Experiments, Results & Analysis

The purpose of the experiments is to verify the impact of the extra security layer of the kata runtime and compare it to runC (Standard Docker runtime) using different network drivers (Section V). The metrics measured in the experiments are latency and network throughput. However, when it comes to virtualized communication, it is known that traffic does not always go through a NIC. This happens because the source and destination are on the same host, so the communication management is done by the hypervisor, which implies CPU consumption. Thus, these two metrics were also evaluated by putting the host processor under stress. Two sets of tests were performed:

1) Measurement of latency and maximum bandwidth without CPU overhead; and
2) Measurement of latency and maximum bandwidth with induced CPU overhead.

A. Measurement & CPU tools

In order to generate the workload and perform the measurements, classical and widely tested tools were used for this purpose:

- iPerf [13]: is used both to generate traffic and measure the maximum possible network bandwidth. iPerf was configured to do TCP communication;
- sockperf [14]: is used to measure the latency and the sensibility of the network to latency; and
- stress-ng [15]: a tool to overload the CPU in order to measure the network traffic under interference of hardware resources

B. Testbed and experiments setup

The testbed environment was configured using a private IaaS cloud available on LabP2D/UDESC. This cloud is based on the cloud operating system OpenStack version Ussuri (Figure 5) using multinode Charm setup.

The testbed (Figure 5) has 2 pre-allocated VMs using the flavor: 4Gb RAM, 2 vCPU, and 20Gb HDD storage. The cloud node in which the VMs were allocated has Intel Xeon E3-12xx 2000MHz, 192Gb RAM, 3 x 1 Gb NICs, and 2Tb HDD storage (CEPH). The VMs have installed GNU / Linux Ubuntu version 18.04.5 LTS Server, and Docker Community version 20.10.2. All experiments were parameterized / automated using shell script, and submitted remotely using SSH via VPN.

Among the containers, client-server pairs were created and communication interfaces were configured by type. The experiments are carried out by exchanging data through these configured interfaces. Each set of tests comprises the following experiments:

- (bridge, bridge);
- (host, bridge);
- (macvlan, macvlan); and
- (overlay, overlay).

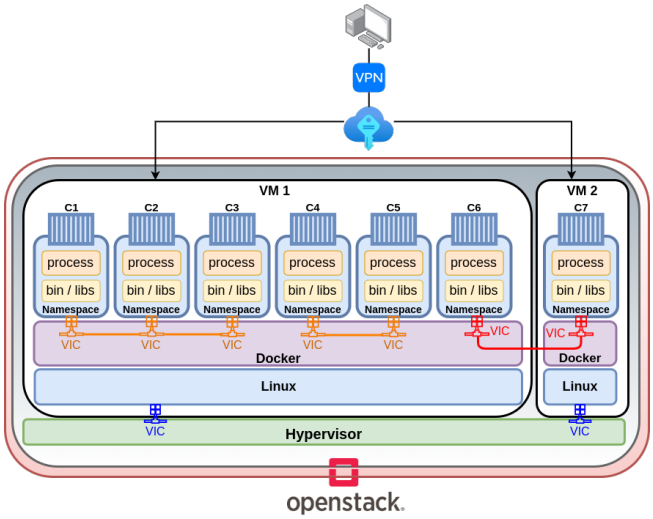

Fig. 5. Testbed scenario example

A total of 15 tests for each server-client driver pair were realized in order to provide statistically significant data. At each test the VMs were destroyed, recreated, and reconfigured to avoid caching. Four group of experiments performed:

- runC runtime without stress-ng;
- runC with stress-ng;
- kata runtime without stress-ng; and
- kata runtime with stress-ng.

Using the workflow (Figure 6) and for each client-server pairs (bridge, bridge), (host, bridge), (macvlan, macvlan), (overlay, overlay), the containers are installed with the tools iPerf and sockperf, to capture the bandwidth and latency respectively when communicating with each other.

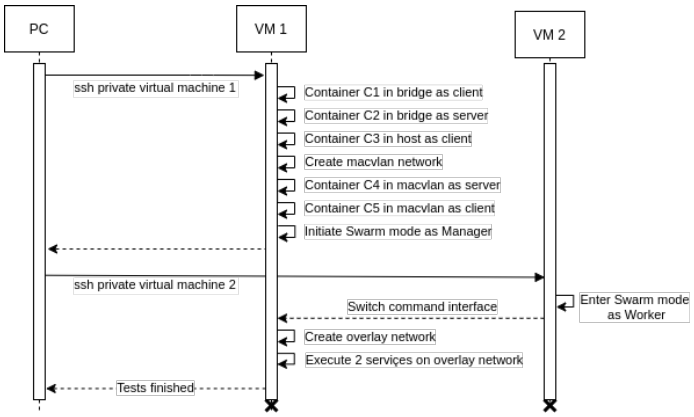

Fig. 6. Sequence diagram illustrating the workflow of (macvlan, macvlan) experiments

In experiments with induction load on the CPU, stress-ng has been installed on the server pair container.

C. Results & Analysis

Kata Containers has a downside of its own runtime kata due to the inability to use the host network. When using the host network driver, the container is linked to the host network stack, enabling the container to get the best throughput performance since it does not need any type of forwarding. It is simply not possible to directly access and establish this link to the host network configuration from within the VM without re-modifying the container network setup and possibly breaking the host networking setup [16]. Thus, in the results of our experiments the host network driver results for kata runtime are set as 0 (latency and bandwidth). The none network driver is also not tested because this drivers does not have any network communication and it is more used to IPC.

Figure 7 reveals an expressive difference between results of the two runtimes, in which runtime runC has the lowest latency for all the drivers tested.

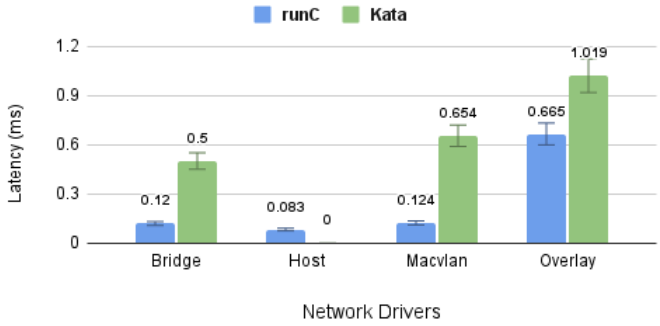

Fig. 7. Latency of each network without stress-ng

This result (Figure 7) is a consequence of how Kata containers implements the networking while providing extra security encapsulation for the container. Runc uses the host kernel to manipulate and separate the processes inside their own namespaces, hence creating a container. This way the containers have almost no extra steps to proceed the information between processes and sockets to the containers isolated process and networks, all inside the same system and level of abstraction. Kata runtime has to implement a lot more extra steps to create the actual communication inside the container while maintaining the whole abstraction level by creating a new VM and building containers inside it. Thus, for the communication to flow from the host to the container, the communication begin by IPC through the kata-shim-v2 whereas exist a virtual socket aligned in the host. When the information hits the shim, which holds the input and output for containers, it reaches a VM. Thus, more IPC is required for the data to go through from the shim to the kata-agent by using gRPC finally carrying out the data to the container. In order to access the container network namespace, the data has to be mirrored through from a vEth pair connected through the Docker Bridge to the traffic control interface, and again mirrored from the TAP to the eth0 container interface. Each step is added to the latency which shows that the extra steps required for the communication has quite a price for the runtime to maintain the security.

Due to the way the network driver was implemented in the kata runtime, it is also expected to have an impact on throughput Figure 8). The Swarm mode which uses the overlay driver has the lowest bandwidth of TCP packets in our results, and it is related to the latency and the overlay driver implementation/architecture to provide the communication of the containers.

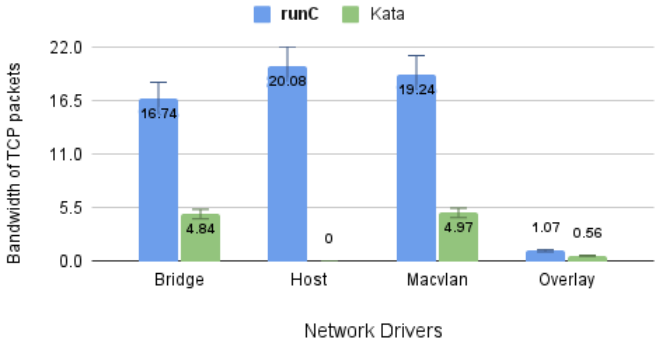

Fig. 8. Bandwidth result by iPerf without stress-ng

The same experiments (Figure 9, Figure 10) with induced CPU overhead revealed that runC suffers less than kata runtime The reason for the drop in bandwidth is related to the CPUs process queue. As the server container is handling multiple CPU intensive requests by the stress-ng tool, the queue is in high demand. As each packet overhead needs to be de-encapsulated and information properly handled, it enters the queue and spend most of its cycle time waiting since the CPU is stalled.

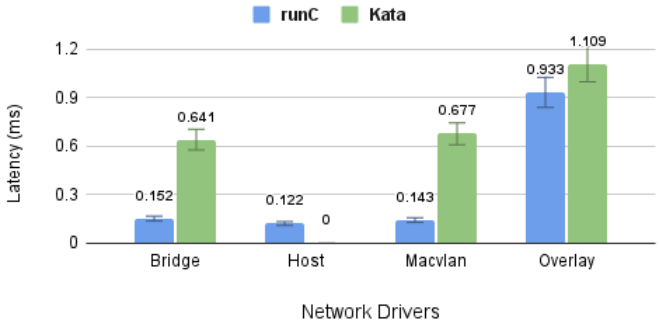

Fig. 9. Latency of each network with stress-ng

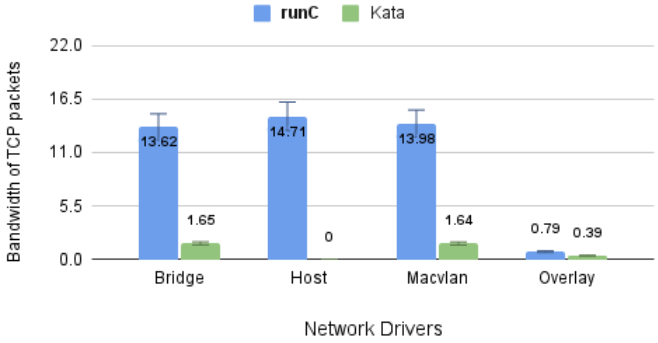

Fig. 10. Bandwidth result by iPerf with stress-ng

As the data travel through the layers, each layer has to encapsulate and de-encapsulate information and besides, if the information transmitted is too large for the size of the network, it has to be fragmented into smaller packages. Both concepts need to be computed inside the CPU but if the CPU queue is full all the time because of a process requiring most of it resources, it will delay the processing of those processes. Overtime the stalled time affects the amount of data being transmitted over the network, influencing the final bandwidth conveyed by the iPerf tool test.

VII. Related work

Although there has been a plenty of articles comparing the performance of containers and virtual machines [17], [18], only two articles were found specifically addressing different container network drivers: [19], [20].

[19] presents a good description of the Docker drivers using runtime runC, but does not address the kata runtime. Several approaches used in this work served as a reference for our article. However, the authors also did not address CPU overhead and its impact on hypervisor-managed network communications.

[20] does not provide a complete insight, and a better understanding, of how the network occurs between the runtimes and how they differ from each other.

VIII. Considerations & Future work

In virtualization and cloud solutions, the search for performance and security is constant. While traditional Docker containers (runC) enjoy good performance and consume few computational resources, they have a shadow of security aspects. In this sense, the evolution of containers to improve security is natural. Kata Containers aims to achieve both by trading off the performance while boosting the container security, spawning them each inside a complete new virtualized QEMU/KVM VM.

In all tests, Docker runtime, runC, has achieved a lower latency for each network driver than Kata Containers. While latency is correlated to bandwidth, runC achieved a higher bandwidth for all cases and scenarios as well.

When using the stress-ng tool to simulate a server being used at its fullest, runC is slightly affected by it while Kata Containers is more shacked, resulting in worse bandwidth usage. As demonstrated, Kata Containers have a long way to actually achieve the speed of containers while maintaining the extra encapsulation security they created, but the security trade-off is a necessary step to decrease chances of unauthorized access and solve some security flaws the container technology possess, even at a severe cost on the network communication.

Despite the result of the kata runtime in relation to the runC, tests are also needed comparing Kata Containers with traditional virtual machines (e.g., KVM) to measure the performance. This is expected future work. Other future work includes more driver combinations, as in this article we only focus on one model. Research into cloud native architecture to use as a test scenario is planned.

Acknowledgment

The authors would like to thank the support of FAPESC, and LabP2D / UDESC.

References

[1] Microsoft, "What is cloud computing," https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/, December 2020.

[2] IBM, "What is virtualization?" July 2019. [Online]. Available: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide

[3] Docker, "What is container," https://www.docker.com/resources/what-container, 2020.

[4] ——, "Docker overview," https://docs.docker.com/get-started/overview/, December 2020.

[5] N. Poulton, *Docker Deep Dive: Zero to Docker in a single book*. Packt Publishing, 2020.

[6] OCI, "Runtime and lifecycle," https://github.com/opencontainers/runtime-spec/blob/master/runtime.md/, 2020.

[7] Ubuntu, "runc," 2019. [Online]. Available: http://manpages.ubuntu.com/manpages/focal/man8/runc.8.html

[8] ContainerD, "Containerd documentation," https://containerd.io/docs/, 2021.

[9] OCI, "Overview," https://opencontainers.org/about/overview/, 2020.

[10] Kata, "Kata architecture," https://github.com/kata-containers/kata-containers/blob/main/docs/design/architecture.md, October 2020.

[11] L. L. Mentz, W. J. Loch, and G. P. Koslovski, "Comparative experimental analysis of docker container networking drivers," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, 2020, pp. 1–7.

[12] Docker, "Network overview," https://docs.docker.com/network/, 2021.

[13] Iperf, "Iperf," 2019. [Online]. Available: https://iperf.fr/

[14] Mellanox, "Sockperf," April 2021. [Online]. Available: https://github.com/Mellanox/sockperf

[15] Ubuntu, "Stress-ng manual," 2019. [Online]. Available: https://manpages.ubuntu.com/manpages/artful/man1/stress-ng.1.html

[16] Kata, "Kata limitations," https://github.com/kata-containers/kata-containers/blob/main/docs/Limitations.md, May 2021.

[17] R. R. Yadav, E. T. G. Sousa, and G. R. A. Callou, "Performance comparison between virtual machines and docker containers," *IEEE Latin America Transactions*, vol. 16, no. 8, pp. 2282–2288, 2018.

[18] Q. Zhang, L. Liu, C. Pu, Q. Dou, L. Wu, and W. Zhou, "A comparative study of containers and virtual machines in big data environment," in *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 178–185.

[19] L. L. Mentz, W. J. Loch, and G. P. Koslovski, "Comparative experimental analysis of docker container networking drivers," in *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, 2020, pp. 1–7.

[20] R. Kumar and B. Thangaraju, "Performance analysis between runc and kata container runtime," in *2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*. IEEE, 2020, pp. 1–4.