Containerized Systems: Difference towards network IO

Raphael Auzinger, Florian Wiedner*, Jonas Andre*
*Chair of Network Architectures and Services
School of Computation, Information and Technology, Technical University of Munich, Germany
Email: raphael.auzinger@tum.de, {wiedner, andre}@net.in.tum.de

***Abstract*—Containers create an isolated runtime for applications alike and have become an integral part of modern cloud computing. To ensure good isolation and security different runtime architectures emerged. On one side Linux containers like runC and on the other micro VM architectures like Kata Containers, gVisor or Firecracker. These security benefits sometimes come at the cost of additional overhead and resource usage. This work combines and compares various other papers and research which address different architectures and investigate how they influence performance.**

***Index Terms*—containeization, runC, LXD, Kata Container, gVisor, network performance**

1. Introduction

Virtualization and containerization became an important tool in the modern age of developing software and cloud computing. It enables developers to quickly and easily setup systems on their development environment without the necessity to start a virtual machine. Containers can create an environment with all the build tools included for reproducible builds. They are used to run CI/CD tests and verify the project. Further, virtualization enabled cloud service providers to offer not only bare metal products but also parts of a dedicated machine with a variety of configuration options. Starting another virtual private server when the load on a system rises or running edge functions to provide a service, relays on this technology. In a cloud environment speed and efficiency is crucial to reduce latency and energy usage. This might favor one technology over another. Further, containers should be isolated from each other to ensure a secure environment and availability. These two goals are partly in conflict, as we will explore in the following sections.

The paper starts with an introduction to the different standards and how they influence the runtime architectures. In Section 4 the architectures are discussed. How classic containerization with cgroups and namespaces evolved and how optimized micro VMs for enhanced security, performance and seperation from other containers. In Section 5 the networking performance of the different architectures will be compared and analyzed. Section 6 summarizes the paper and will give an outlook on further research and details which can be included in future papers.

2. Related work

This paper is influenced by the research of Wang et al. [1] who analyzed the performance and isolation of runC, gVisor and Kata Containers runtime, Cochak et al. [2], who compared runC and Kata Containers using Docker and on the work of Kovács [3] in which he compared Linux Containers, runC with Docker and KVM virtualization.

3. Containerization Standards

In order to use different runtimes with container images, two standards were established. The Open Container Interface Runtime Specification for the underlying runtime which takes care of execution of a container and the Open Container Interface Image Specification for container images. The OCI Runtime Specification defines the behavior and the configuration interface of low-level container runtimes [4]. Applications like Docker, Kubernetes and Podman [5] [6] [7] can interface with OCI-RS and control the containers. Runtimes which support OCI-RS like runC, LXD, Kata Containers, gVisor and Firecracker can run images which adhere to the OCI Image Spec. It defines the structure of a container image and how the runtime should execute it [8].

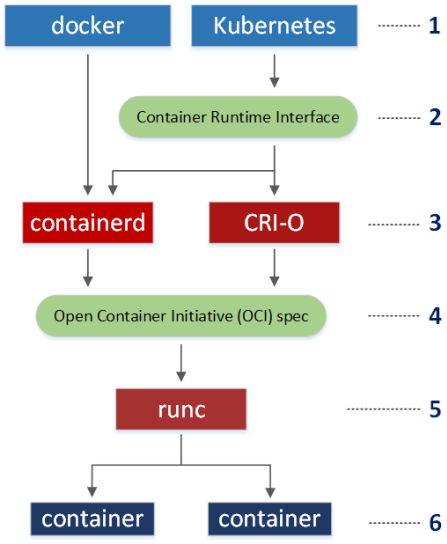

Figure 1: The relationship between Docker, CRI-O, containerd and runc [9]

At level 1. an application like kubernetes communicates with an underlying runtime at level 2. , which

 doi: 10.2313/NET-2024-04-1_01

complies to the Container Runtime Interface Specification. This runtime can now control an OCI image 5. and run it as a new container at level 6. [10].

4. Container Architectures

New architectures were developed around these standards to enhance the security of the host system running containers and to strengthen the isolation between them. Figure 2 shows where different architectures reside, regarding their level of virtualization. On the left no virtualization on a bare metal host system and on the right virtual machines with fully fledged operating systems. Subsection 4.1 4.2 and 4.3 will get into the details and degree of virtualization.

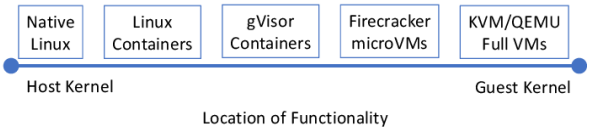

Figure 2: Container isolation spectrum [11]

4.1. Linux Containers

Linux Container, runC and Linux Container Daemon utilize Linux native virtualization methods like cgroups and namespaces to provide a runtime for containers [12] [13] [14]. This approach is very lightweight since the host kernel can be shared with the running container as shown in Figure 3. However, a shared kernel introduces some

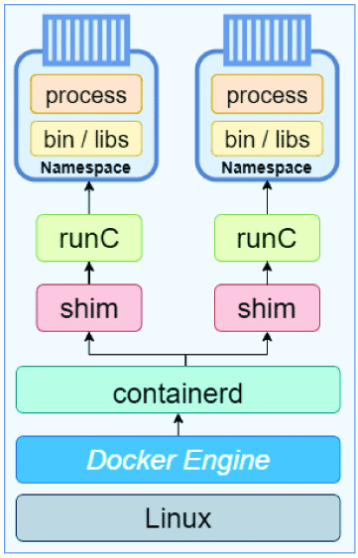

Figure 3: runC virtualization diagram [2]

security concerns if an application escapes the container.

4.2. Sandbox With Multi-purpose Kernel

gVisor tries to solve these security issues and introduces a Sentry and Gofer between the host kernel and a container as shown in Figure 4. System calls inside the container are intercepted and processed by gVisor, which in tern will forward them to the host or abort the call [11].

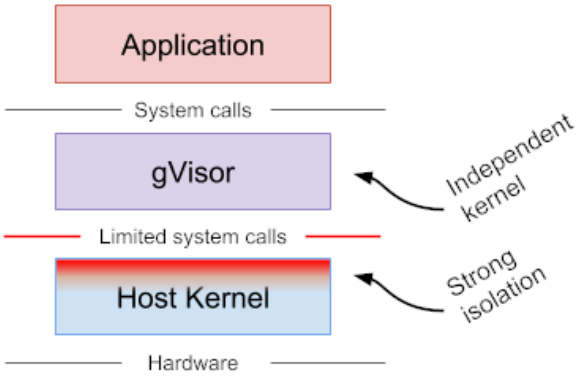

Figure 4: A schematic diagram of gVisor [15]

gVisor handles most of the network tasks inside netstack, but still performs some checks on the host [1]. Netstack is gVisors networking implementation written in Go. All packet processing is done inside gVisors Sentry, which in turn reduces the host I/O syscalls and keeps it isolated from the host networking stack. This virtual networking needs additional processing power. Further the memory management of Go adds additional stress to the underlying system. [15].

4.3. Micro VM Architecture

Kata Containers uses light weight virtual machines to isolate containers, as can be seen in Figure 5. Kernel functionality is moved to a guest operating system or the host QEMU process [11]. These operating systems and kernels are optimized and stripped down to basic functionality to increase start up times and reduce the attack surface [16]. By default, the Kata Containers network uses traffic

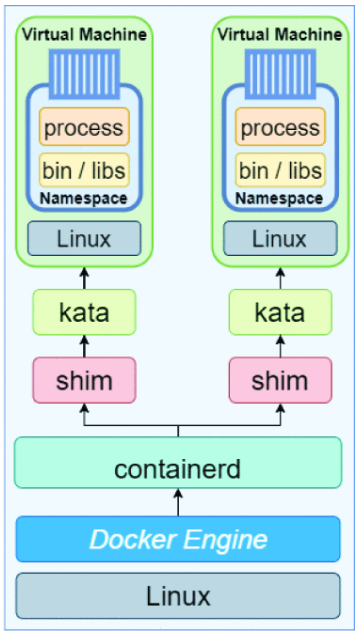

Figure 5: Kata Containers virtualization diagram [2]

control to transparently connect the veth interface with the VM network interface [1].

5. Network Performance

Wang et al. [1] paper looked at the performance between runC, Kata Containers and gVisor. His results are discussed in Subsection 5.1 and are the focus of this paper. Kovács [3] analyzed the networking behavior of Docker compared to LXC and KVM in Subsection 5.2.

5.1. runC - Kata Containers - gVisor

The test was carried out on a machine with an Intel i5-7500 with support for nested virtualization, 8 GB RAM and a 1 TB hard drive. Ubuntu 18.04 LTS (5.4.0) Linux distribution was used as the operating system of choice. On the system were

- Docker version 20.10.1
- KVM version 2.0.0
- gVisor version 20201030.0
- Kata Containers version 1.12.0-rc0

installed [1]. They did not add any restrictions to the containers for the networking tests. Netperf [17] was used to measure network latency and throughput. gVisor was configured to use the user space netstack instead of the host stack [1].

In Figure 6 the TCP_STREAM performance is visualized. runC and Kata Containers are close together and only differ in less than 1%. gVisor has the lowest throughput of the three [1].

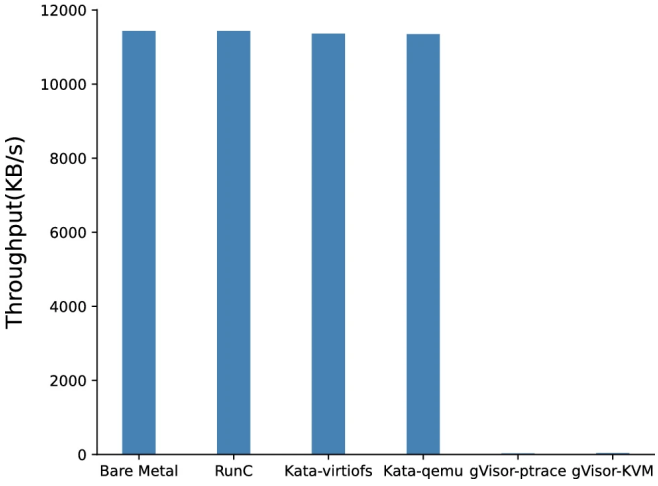

Figure 6: TCP_STREAM network performance [1]

Figure 7 visualizes the TCP_RR, TCP_CRR and UDP_RR performance. TCP_RR are request response tests. The time it takes for a response is measured. While a TCP_CRR test also takes the connection time into account [17]. Bare metal and runC bandwidth performance is almost the same. The loss is less than 1%. Further Kata Containers also looses less than 1% in regard to runC [1] TCP_CRR and UDP_RR show a higher loss of 18.52% and 17.23% respectively. gVisor is again the slowest in terms of network performance. This might be due to its user space networking stack [1].

Figure 8 shows the result of the TCP and HTTP throughput and latency measurements [1]. Kata Containers QEMU TCP and HTTP bandwidth is 1.08% and Kata

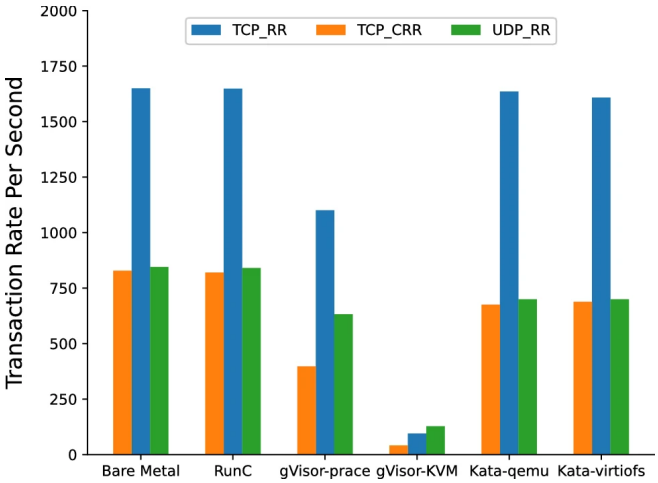

Figure 7: TCP_RR, TCP_CRR and UDP_RR performance [1]

virtiofs 13.12% slower compared to runC. gVisor had the lowest throughput. Under the hood it uses a double packet management mechanism at the Sentry stage (netstack) which leads to the lowest transaction rate and lowest throughput [1].

5.2. Docker - LXD - KVM

The test was carried out on a Huawei CH121 BladeServer with 2x E5-2630 v3 @ 2.40GHz 8Core CPUs and 128 GB DDR4 memory. The server supports nested virtualization. The host operating system was Ubuntu 16.10. The container images ran Ubuntu 16.04.1 LTS. IPerf version 3.1.3 was used to measure network performance. Another HUAWEI BladeServer CH140 with 2x E5-2670 v3 @ 2.30GHz with 12 Cores and 128 GB DDR4 acted as an IPerf server. The two servers were connected via a 10GE HUAWEI CX311 Switch. Docker and LXC were configured with host networking and used the same physical network card as the host operating system [3].

Figure 9 and 10 visualize the results of the measurements. KVM falls behind Linux Container and runC [3]. The throughput is lower and the standard deviation higher due to the higher virtualization effort. It uses a hypervisor and every guest system includes its own guest kernel and virtual networking [18]. This overhead results in a performance penalty. In return, it is more secure since it does not share a kernel with the host OS [18].

6. Conclusion

Each architecture has its drawbacks and benefits. However, each runtime tries to enhance the security of the host system and container isolation. These improvements add layers to the system at the cost of performance.

The research of the cited papers might not reflect the current state regarding networking speed. The overhead might be reduced and speed and security increased, since the projects Kata Containers and gVisor are in active development. runC also got a lot of updates over the past view years, which had an impact on security and networking performance. The results give a rough idea

	Bare metal	RunC	gVisor-ptrace	gVisor-KVM	Kata-QEMU	Kata-virtiofs
TCP Bandwidth (KB/s)	96662.02	96204.8	655.36	686.08	95621.12	86778.88
HTTP Bandwidth(KB/s)	53196.8	49112.58	262.40	268.80	46215.68	43569.15
TCP Latency(us)	593.71	592.29	938.48	900.09	710.96	700.76
HTTP Latency(us)	730.29	776.61	1382.93	1161.59	1015.23	1010.72

Figure 8: TCP and HTTP network performance of container runtimes (Ethr benchmark) [1]

CORRECTED IPERF MEASUREMENT RESULTS

	Docker (MB)	*LXC (MB)*	*Singularity (MB)*	*KVM (MB)*	*Native (MB)*
Average	1122,0	1121,8	1122,1	1116,7	1122,2
Std. Deviation	0,471	0,632	0,316	15,720	0,632

Figure 9: Corrected IPerf Measurement Results [3]

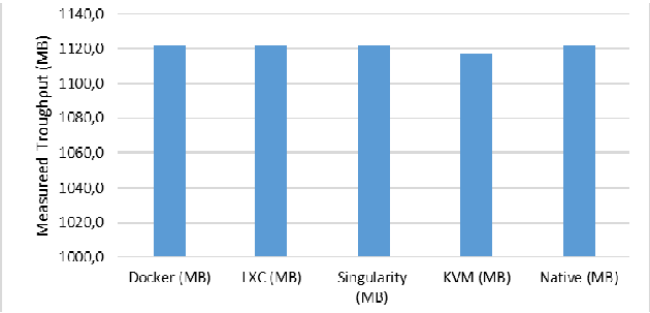

Figure 10: IPerf Corrected Network Measurements [3]

of what can be expected from the different runtimes, but should be taken with a grain of salt.

Different container network drivers could also influence the networking performance gap. Research done by Cochak et al. [2] gives some interesting insights. He compared bridged, host, macvlan and overlay modes with Kata Containers and runC. Some combinations increased the bandwidth but also increase the latency.

Host system optimization was also ignored in this paper. A system tuned for networking / memory / storage performance could lead to higher results or shrink the gap between some runtime approaches.

References

[1] X. Wang, J. Du, and H. Liu, "Performance and isolation analysis of runc, gvisor and kata containers runtimes," *Cluster Computing*, vol. 25, no. 2, pp. 1497–1513, Apr 2022. [Online]. Available: https://doi.org/10.1007/s10586-021-03517-8

[2] H. Z. Cochak, G. P. Koslovski, M. A. Pillon, and C. C. Miers, "Runc and kata runtime using docker: a network perspective comparison," in *2021 IEEE Latin-American Conference on Communications (LATINCOM)*, 2021, pp. 1–6.

[3] Á. Kovács, "Comparison of different linux containers," *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*, pp. 47–51, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:2864721

[4] opencontainers.org, "Oci runtime spec v1.1," https://opencontainers.org/posts/blog/2023-07-21-oci-runtime-spec-v1-1/, 2023, online; accessed 25-September-2023].

[5] Docker, "Docker docs," https://docs.docker.com/, 2023, online; accessed 25-September-2023].

[6] T. K. Authors, "Kubernetes components," https://kubernetes.io/docs/concepts/overview/components/#node-components, 2023, online; accessed 26-September-2023].

[7] P. team, "What is podman?" https://docs.podman.io/en/latest/, 2023, online; accessed 26-September-2023].

[8] opencontainers.org, "Open containers," https://opencontainers.org/, 2023, online; accessed 25-September-2023].

[9] T. Donohue, "Die unterschiede zwischen docker, containerd, cri-o und runc," https://www.kreyman.de/index.php/others/linux-kubernetes/232-unterschiede-zwischen-docker-containerd-cri-o-und-runc, 2023, online; accessed 25-September-2023].

[10] ——, "The differences between docker, containerd, cri-o and runc," https://www.tutorialworks.com/difference-docker-containerd-runc-crio-oci/, 2023, online; accessed 25-September-2023].

[11] Anjali, T. Caraza-Harter, and M. M. Swift, "Blending containers and virtual machines: a study of firecracker and gvisor," *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:211828470

[12] linuxcontainers.org, "What's lxc," https://linuxcontainers.org/lxc/introduction/, 2023, online; accessed 25-September-2023].

[13] . C. Ltd., "Run system containers with lxd," https://ubuntu.com/lxd, 2023, online; accessed 25-September-2023].

[14] opencontainers.org, "runc readme.md," https://github.com/opencontainers/runc/blob/main/README.md, 2023, online; accessed 25-September-2023].

[15] Google, "What is gvisor?" https://gvisor.dev/docs/, 2023, online; accessed 24-September-2023].

[16] A. W. Services, "Firecracker how it works," https://firecracker-microvm.github.io/#how_it_works, 2023, online; accessed 26-September-2023].

[17] R. Jones, "netperf," https://github.com/HewlettPackard/netperf, 2023, online; accessed 26-September-2023].

[18] C. Ltd., "Kvm hypervisor: a beginners' guide," https://ubuntu.com/blog/kvm-hyphervisor, 2023, online; accessed 26-September-2023].