doitnow

Whitepaper

Containers for HPC: State of the Art 2022

1. Introduction

This document presents an overview and evaluation of container technologies/solutions, with a focus on their suitability for HPC systems and other scenarios where the demanding needs of performance and security may hinder the adoption of new technologies and trends. The analysis takes into account the following factors:

- The performance of containerized HPC jobs.
- Vulnerabilities from privilege escalations by unauthorized users.
- Integration with CI/CD workflows.

2. Container technologies for HPC

2.1 Docker v 20.10.17

Docker was the first technology to put together all kernel mechanisms in order to provide a consistent container framework. Docker is a technology not very well suited for HPC purposes, but it is included here for interest purposes as it is the reference for all other container technologies.

2.1.1 Architecture

Figure 1 illustrates the docker architecture. The docker engine, responsible for pulling images and spawning containers on a host, follows the paradigm of a **root-owned daemon**. The dockerd daemon can:

- Pull/push images from/to a registry and store them locally.
- Build images from local FS or from a recipe file.
- Spawn containers from an image.

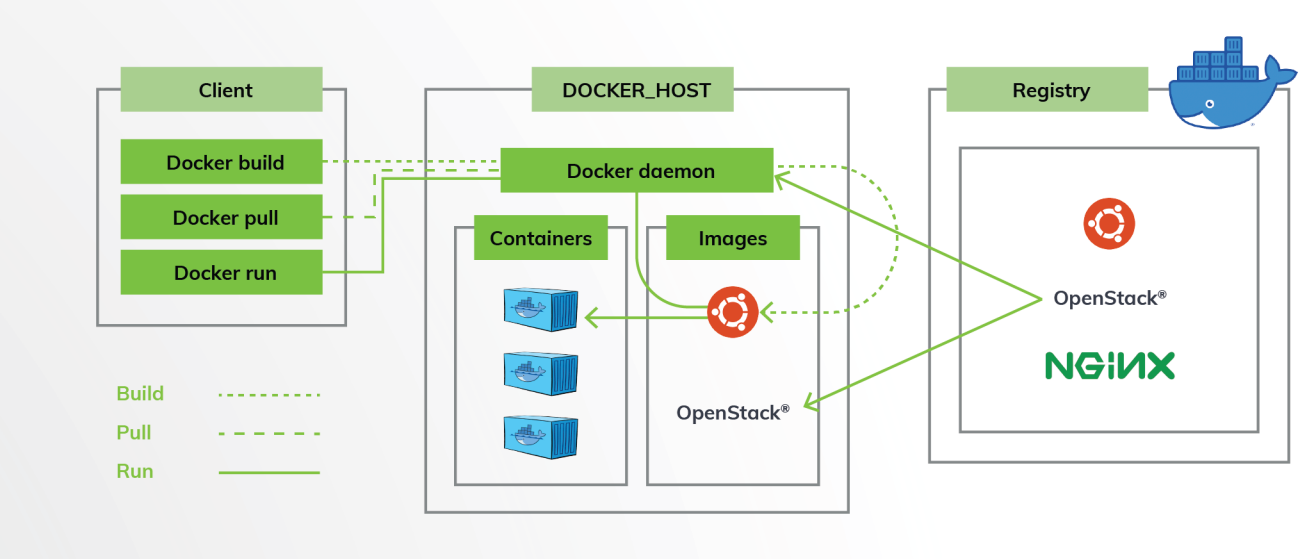

Figure 1 - Docker architecture

Nevertheless, only the root user or a user belonging to the docker group can interact with the daemon. Being owned by root, dockerd can leverage all privileged namespaces in order to provide very advanced features like:

- Create virtual networks devices, or even virtual networks interconnecting privately all the containers in a host.
- PID abstraction, so containerized processes have their own PIDs separated from the host.
- Resource limitation through cgroups. Docker can control the amount of resources available to a container.
- Create and build images. As mentioned before, this feature usually requires privileges and the use of mount namespace, bind and overlay mounts.

2.1.2 Relevant features

- No-new-privileges parameter.
- Docker on rootless mode. SDocker offers the possibility of running the daemon as a non-root user. Rootless mode executes the Docker daemon and containers inside a user namespace and uses some specific setuid binaries to map users from the created user namespace to the root user namespace. As a counterpart, the set of features available through this mode is limited. Still, the rootless solution is more secure than root-owned daemon from a vulnerability exposure point of view.

2.1.3 Limitations

The following limitations must be highlighted when thinking about the suitability of Docker on HPC environments.

- Any docker user can easily escalate to root.
- Docker is difficult to integrate with some workload managers.

2.2 Singularity v3.10.2

Singularity started in 2015 as an HPC alternative to Docker. Sylabs was founded in 2018 to support Singularity, but in May 2021 they forked the project into SingularityCE. In November 2021 the Singularity open-source project joined the Linux Foundation and was renamed Apptainer. While both projects were similar at the start, they have now diverged.

2.2.1 Architecture

Figure 2 illustrates the architecture of Singularity. Singularity is based on **root owned setuid binaries**. Singularity deploys a set of binaries that can be called by any user, but they are executed with root privileges. As such, they have the right to execute privileged operations.

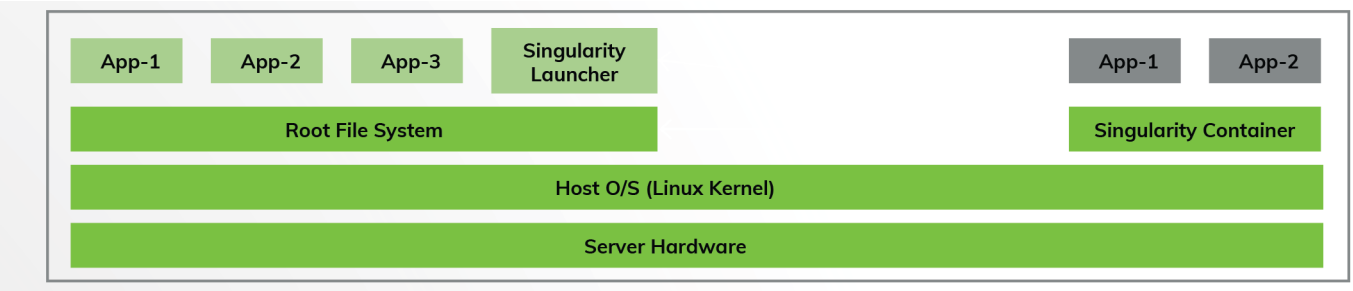

Figure 2 - Singularity architecture.

Singularity is strongly focused on HPC. As such, it brings a specific set of features mostly relevant on HPC environments:

- Strong focus on reproducibility: cryptographic signatures, an immutable container image format, and in-memory decryption.
- Can be used with most workload managers out of the box.
- Facilitates the integration of GPUs, high speed networks, and parallel filesystem.

2.2.2 Relevant features

- Accessible to unprivileged users by design. Singularity security model is based on the idea of untrusted users to run untrusted containers. The processes in the container will run as the calling user.
- Full rootless mode. Singularity supports running containers without using the setuid binaries. Instead, user namespaces can be used for those environments which need to reduce the surface of attack to the minimum.
- Fakeroot feature. Singularity offers this feature to emulate the root user when launching containers or building images.

2.2.3 Limitations

- Limited interoperability with Docker and OCI.
- Divergence of forks. As SingularityCE by Sylabs and Apptainer by the Linux Foundation diverge, the set of features, formats, standards, and performance may diverge, which complicates the situation to the end user.

2.3 Apptainer v1.1.0

Singularity was renamed Apptainer when it joined the Linux Foundation project. At the time of the writing, Apptainer just released the version 1.1.0. The release notes of this version clearly indicate how the main effort of Apptainer is being put towards getting rid of the setuid binaries architecture and perform fully on unprivileged mode by relying on user namespaces, without hindering the performance.

This strategy translates into the following technical differences with Singularity:

- Apptainer no longer installs a setuid-root portion by default, as full-rootless is the intended mode of operation.

- They have greatly mitigated the performance impact of using FUSE filesystem (required for mounting SquashFS without privileges) by developing a parallel multithreaded version of SquashFS with FUSE.
- Added the ability to use persistent overlay and --writable-tmpfs without using setuid-root.
- Extended the fakeroot feature to be used even if subid and subgid mappings have not been set.
- Encryption is not yet supported. Container signature and verification requires Apptainer to be installed in setuid mode.

2.4 Podman v4.2.0

Podman, developed and maintained by Red Hat, is a lightweight runtime for managing and running docker without the overhead of the docker daemon. Podman started in 2018 and it has lately gained some attention as an alternative to docker to let users run containers without the need of privileges.

2.4.1 Architecture

Figure 3 illustrates the way Podman works. Podman runs without daemon, and neither uses setuid executables. Instead, the tool is intended to be run as a non-root user and uses **sudo** only on those precise operations where root escalation is required.

Podman can provide the following operations to unprivileged users:

- Pull/push images from/to a registry and store them locally.
- Build images from local FS or from a recipe file.
- Spawn containers from an image.

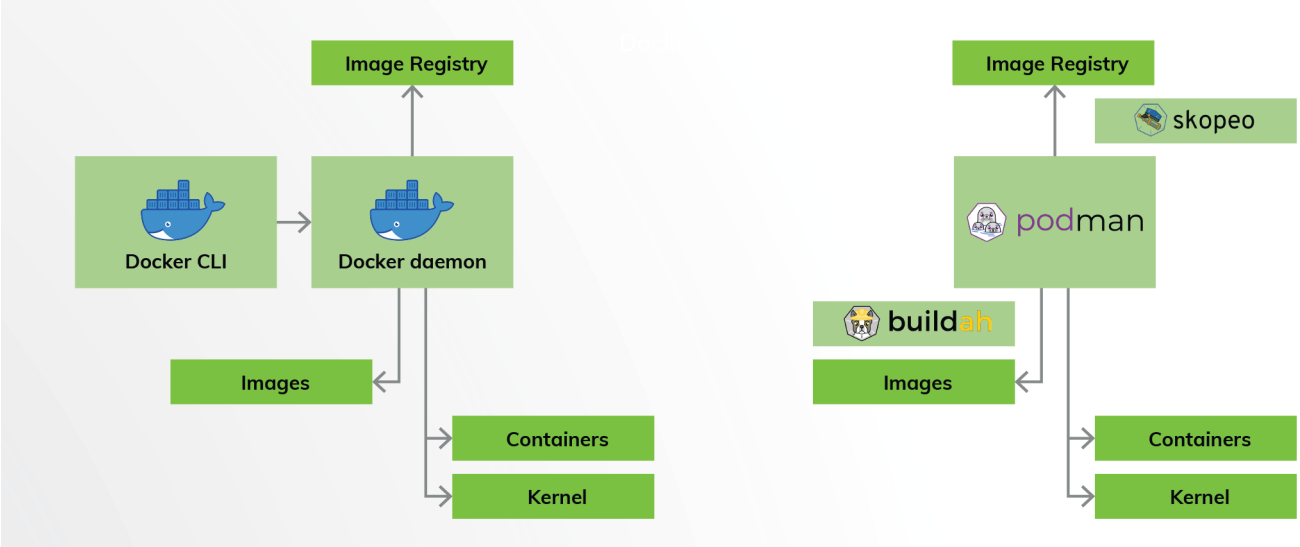

Figure 3 - Podman architecture vs Docker architecture

As we have seen with other technologies, some parts of the workflow in order to build images or even spawn containers require some sort of privilege. Podman neither uses a root daemon, nor uses setuid executables. Instead, it uses some extra libraries and packages in order to work around these issues.

2.4.2 Relevant features

- No daemon required. Podman can be deployed as a tool and does not require any daemon running on the hosts.
- Full rootless mode. Podman provides, even with certain constraints and limitations, a full rootless mode that may cover up most of the container uses cases.
- Powerful additional tools. Podman makes use of very powerful extra tools like skopeo (in order to inspect remote images and manipulate the images layer by layer) and buildah (in order to create and build images).

2.4.3 Limitations

- Performance issues. The use of user namespaces requires passing by a number of additional libraries that implement some features in user space that are usually implemented in kernel space (e.g., the network or the filesystem). This may pose a performance penalty, as FUSE is usually less performant than its kernel space counterparts.
- Issues with NFS home directories. Podman does not support container storage over NFS, which may pose a problem for all those installations which have HOME directories on an NFS, as the HOME is the place by default in which user images are stored.

2.5 Sarus v1.5.1

Sarus is a container technology specifically purposed for HPC environments but keeping compatibility with OCI standards. Sarus is developed and maintained by CSCS, the Swiss National Supercomputing Center. First released in 2018, Sarus benefited from the experience gained at CSCS working with Shifter, another technology for HPC containers now abandoned.

2.5.1 Architecture

Figure 4 illustrates the architecture of Sarus. Sarus is based on a root owned setuid binary. This binary can be called by any user, but it is executed with root privileges required to launch the container without user namespaces or the use of workarounds in user space. As it is the case with Singularity's default mode, user namespaces are not used.

Sarus does not implement any image building functionality. This means that Sarus cannot be used to build containers. Instead, Sarus follows an approach in which it delegates the image construction to docker (or any other OCI image builder).

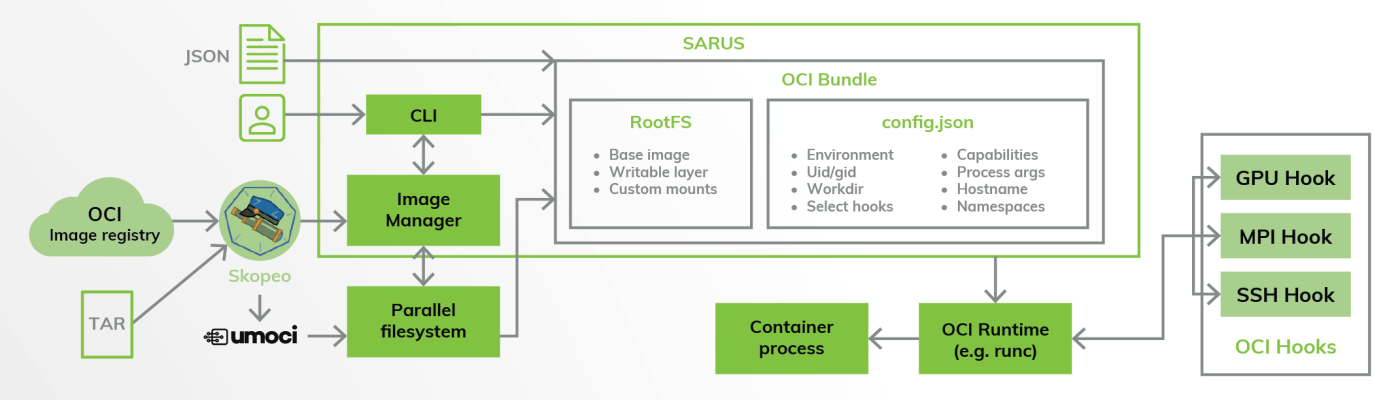

Figure 4 - Sarus architecture

Sarus keeps the functionality to the minimum, managing the images locally, creating an OCI bundle, and finally launching the container in the cluster. In order to implement features like pulling images from remote registries or manipulating them, Sarus deploys as part of its suite additional tools like Skopeo, or umoci, just like Podman does.

2.5.2 Relevant features

- Support for container customization through hooks. Sarus allows containers to be customized by other programs or scripts leveraging the interface defined by the OCI.

- OCI compliance. Sarus focuses on OCI compliance and bases its image retrieval toolchain on external tools like Skopeo, which are much more optimized and focused on this workflow. As result, Sarus is much more compatible with not only Docker registries, but also with third party registry providers like Gitlab, JFrog, or Harbor.

2.5.3 Limitations

- No build features. Sarus delegates all the build phase of the container's workflow to external tools, and assumes the images have been already built, and that the users are sure to have available either an external repository or a build environment. Whether this is a smart strategy or a limitation in the tool is left to the reader's discretion.

- User synchronization. Sarus keeps, for its internal purposes, its own copy of /etc/password and /etc/group files. It is necessary to keep both copies (the host copy and the sarus copy) synchronized, either using a periodic cron script or any other mechanism. This adds a small layer of complexity to the solution administration.

2.6 Other container technologies

The following list contains some interesting technologies in the field. They were not adequate enough for an in-deep analysis, but it is convenient to know about them within the container landscape:

- Charliecloud: An ultra-light, full-rootless container technology developed at Los Alamos National Laboratory. Good for ultra-secure environments, it uses user namespaces but it requires packing container images in a compressed file format, in order to avoid mounting operations that require privileges.

- Enroot: A small container engine based on transforming traditional containers into SquashFS sandboxes. It uses several pre-startup hooks to set HPC infrastructure like Nvidia GPUs and Mellanox networks. The features implemented are too limited to be counted as a viable alternative to Singularity or Sarus, but it may be the right alternative for some specific projects with strict performance requirements and heavily relying on Nvidia GPUs and Mellanox IB.

- Udocker: A python wrapper that encapsulates multiple technologies aiming to run containers in full rootless mode. It allows to decide the container engine used underneath. Singularity, runc, or PRoot are some of the engines available.

3. Conclusions

During the previous sections we have analyzed functional and non-functional characteristics for the five candidate technologies: Docker, Singularity, Apptainer, Podman, and Sarus. We extract the following conclusions from this analysis:

- Docker and Podman can be considered general purpose container technologies, with features and characteristics not specifically tailored for any computing environment, whereas Singularity and Sarus are strongly focused on HPC environments, and both best deliver on HPC at the expense of having a narrower catalog of features.

- Docker and Singularity/Apptainer are more mature technologies than Podman and Sarus. Both Docker and Singularity started earlier, have a more developed product with companies backing it, and commercial support models. Podman, even if it's a product developed and maintained by RedHat, does not have yet the set of features that the other two have, and Sarus, developed by CSCS, does not offer yet any official support.

- Singularity/Apptainer and Sarus, being both focused on HPC environments, have followed during the past years opposite strategies to develop their product. Singularity/Apptainer has opted for developing their own ecosystem, aiming towards tailoring the product for best performance on HPC. Sarus, on the other hand, has opted for full integration with the Docker and OCI ecosystem.

- Singularity and Apptainer are already starting to diverge and appear as differentiated products. Their main differentiation at this point is their opposite strategies concerning the use of unprivileged user namespaces.

- Lately, the container ecosystem has exploded, and the OCI initiative has gained momentum, so all technologies realize the importance of complying with OCI standards.

- Concerning security aspects, Podman is the simplest and best architecture from the security point of view, as it does not run any root-owned daemon nor setuid executables. Apptainer also is pursuing this goal by setting as default mode the full unprivileged mode. Docker implements many security features, but their architecture and model makes it unsuitable for multi-tenant non-privileged platform as those of HPC. Concerning CVEs, mature technologies like Docker and Singularity have many more vulnerabilities than recent technologies like Podman and Sarus, probably due to being more years under scrutiny and a wider set of features, which translates to more surface of attack.