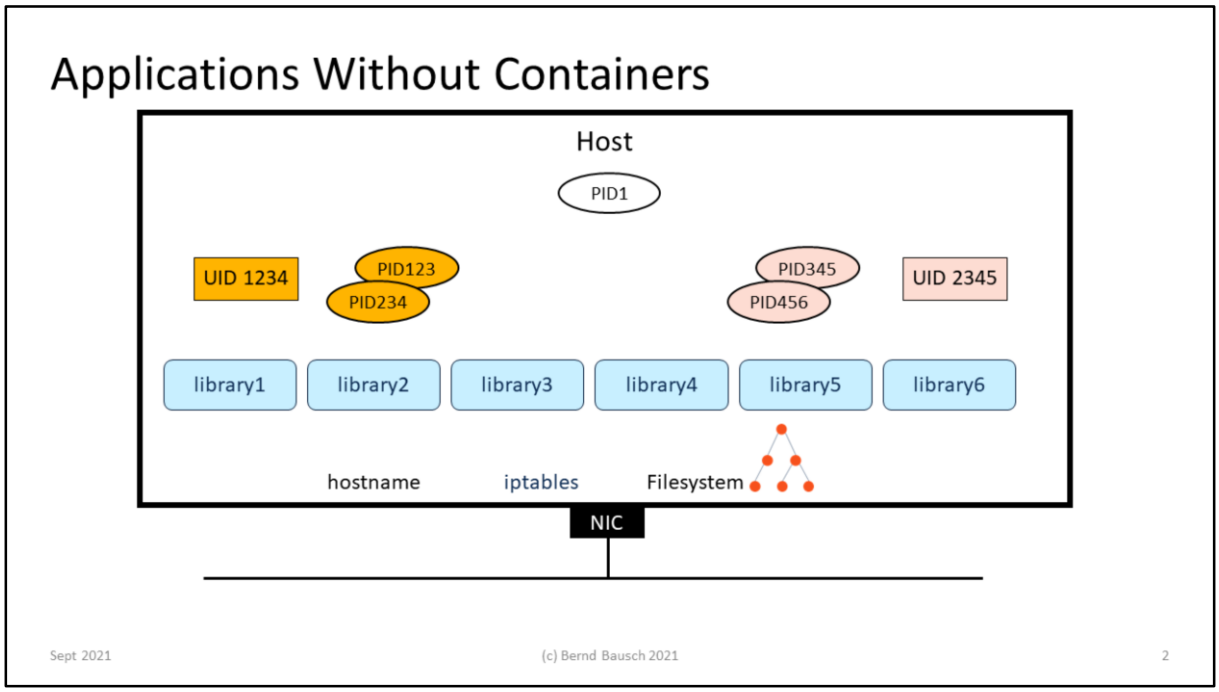
Module 1 Containers

An overview



Traditionally, applications run as processes on a host (or several hosts). This means that they **share** many of the host's resources: Networking (interfaces, routing tables, netfilter), filesystem, the clock, the hostname etc., but also libraries. The host has a single process table that contains all processes. There is one list of users. Applications may run under a certain user ID, and since it is not possible to hide a process, an application can see and interact with the processes of other applications.

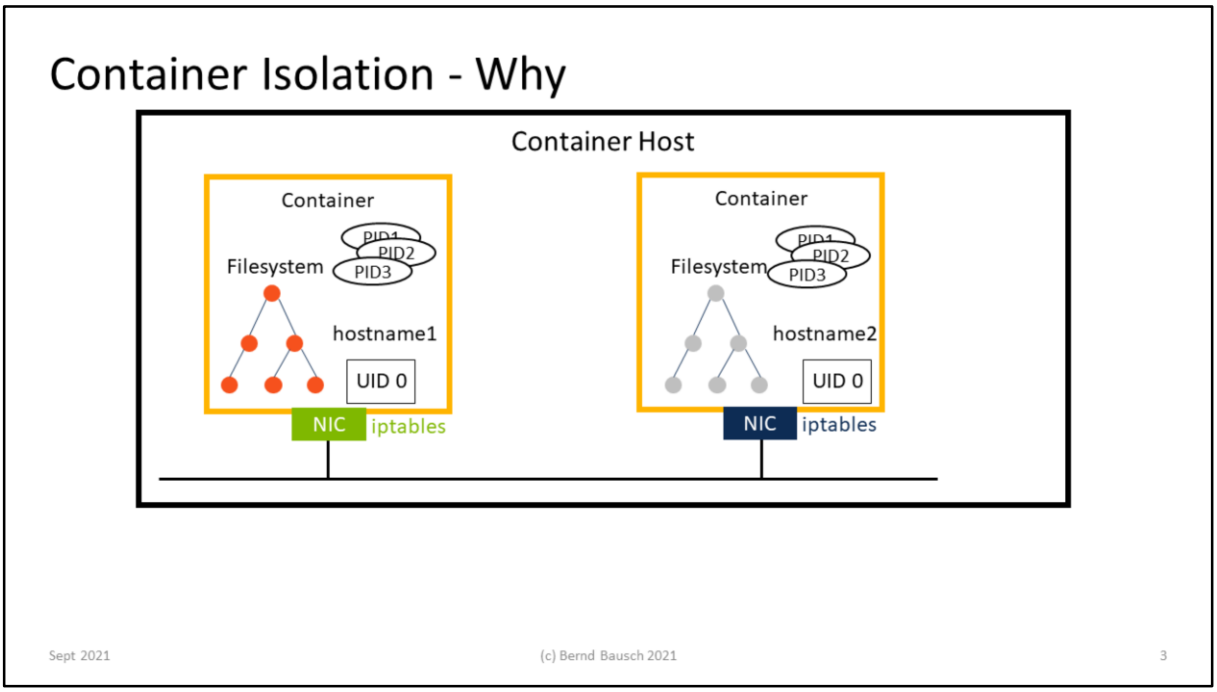
Another aspect is the **resource usage** of an application. It is desirable to limit an application's use of CPU, disk I/O, main memory or network resources so that it can't monopolize the system.

Sharing a host among applications becomes a problem when applications don't behave like good citizens, due to malice or programming errors. An application might kill other processes or use up all of the host's RAM.

For a developer, the most annoying problem may be **updates**. Whenever a library that the application is using is updated, there is a risk that the application breaks. It has to be tested again and possibly adapted.

To address these problems, we could run each application in a separate virtual machine. This would isolate them, and VM resources can be limited. Each application would get the library versions that it requires.

This is wasteful, however: Each virtual machine has its own kernel and (emulated) hardware as well as the entire fleet of management software. Each VM has a full root filesystem that occupies a handful of gigabytes. This severely limits the number of virtual machines that can run on a computer.



Containers are a lighter-weight method for protecting applications from each other. Processes in a container run directly on the host, not in a virtual machine, and use the host's kernel.

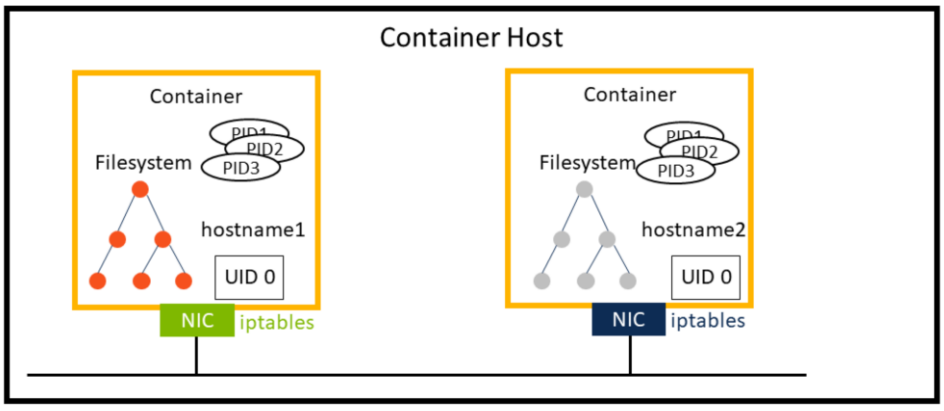
However, processes in a container can only see the resources that were made available in the container. Each container has its own process table, its own filesystem, its own network interfaces, routing table and netfilter rules, and its own hostname. It may have its own user IDs and other resources.

The processes in the left container, with process IDs 1, 2 and 3, are unable to see or otherwise access the processes in the right container, which also have process IDs 1, 2, and 3. Processes that run in containers can be seen from outside, although their process IDs on the host differ from those in the containers.

The same is true for the filesystems. Containerized processes can only see the filesystem in their container. Depending on the container filesystem implementation, processes outside of containers may see all container filesystems, though the file paths on the host differ from the file paths in the container.

Network resource, the hostname and other container resources are not visible to outside processes.

Container isolation by Namespaces



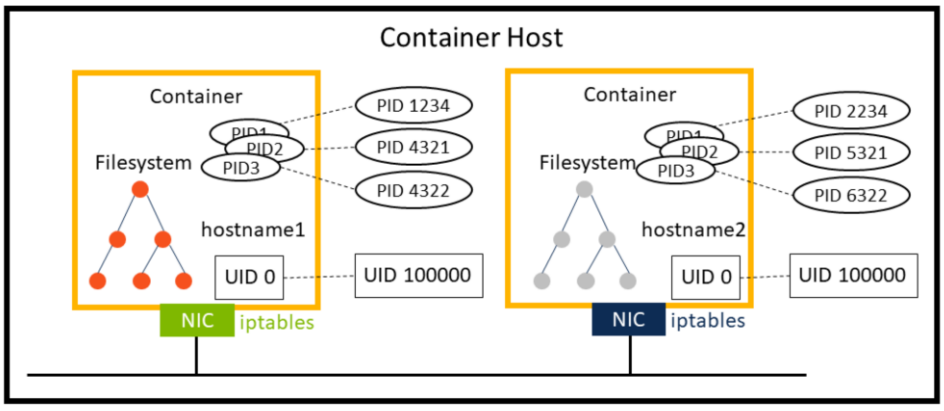
Namespaces: (see `/proc/PID/ns`)

Mount	UID	CGroup
PID	UTS	Time
Network	IPC	

Sept 2021 (c) Bernd Bausch 2021 4

The Linux kernel uses namespaces to isolate containers from each other and from the host.

Container isolation by Namespaces



Namespaces: (see `/proc/PID/ns`)

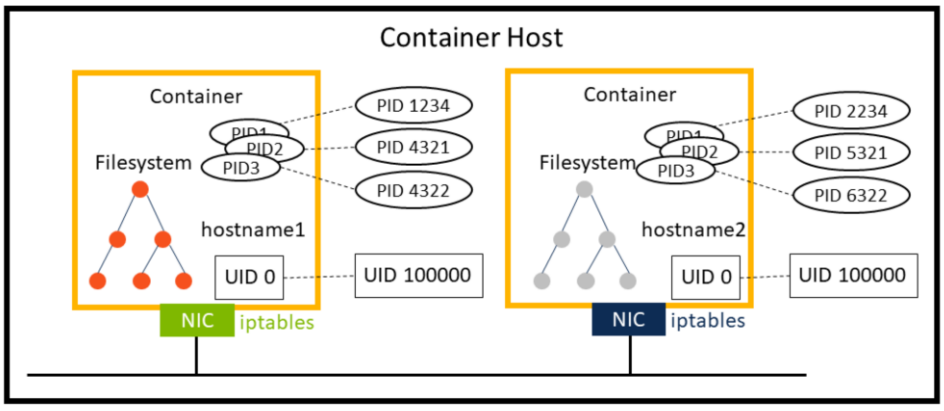
Mount	UID	CGroup
PID	UTS	Time
Network	IPC	

Sept 2021 (c) Bernd Bausch 2021 5

Here we see how the PID and UID namespaces work. A process running in a container is visible from the container host, but has a different process ID.

When the UID namespace is not used, a container process with UID 0 also has root privileges outside the container, which is an obvious security risk. Such a container is called **privileged**. If a container is configured with the UID namespace, user ID 0 inside the container is a non-root user on the host. This is an **unprivileged container**.

Container Resource Limitations



Namespaces: (see `/proc/PID/ns`)

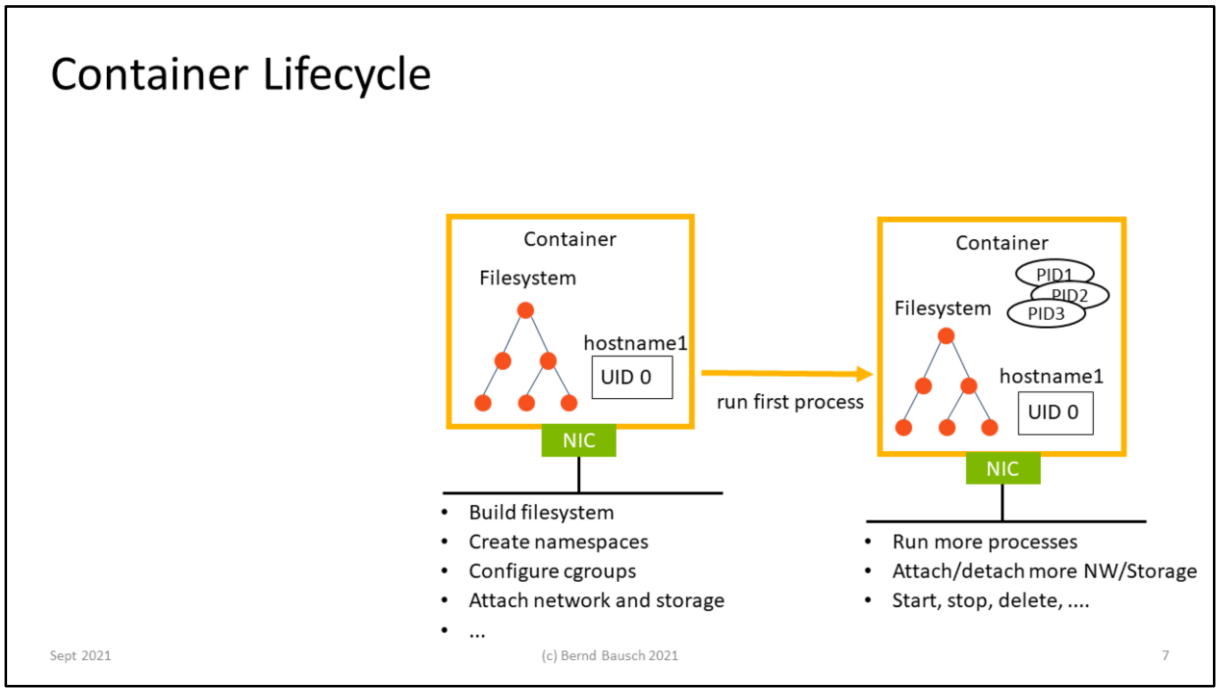
Mount	UID	CGroup	
PID	UTS	Time	
Network	IPC		

Also: Resource control (CPU, RAM, ...) via cgroups

Sept 2021 (c) Bernd Bausch 2021 6

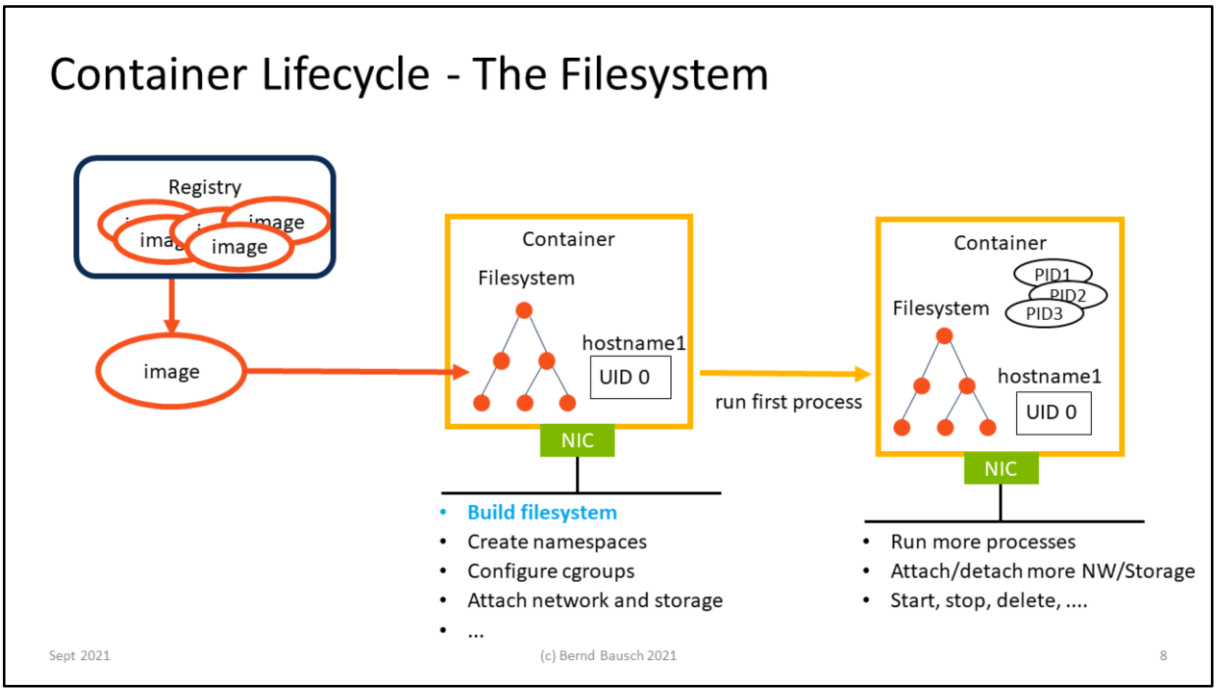
By isolation, we usually mean preventing access to other containers and to the host. However, container processes can freely use system resources like the CPUs or memory.

Thanks to another kernel feature named **Control Groups**, CPU, RAM, disk IO and network traffic limits can be defined for processes. This feature is used by container software to ensure that a container is unable to monopolize system resources.



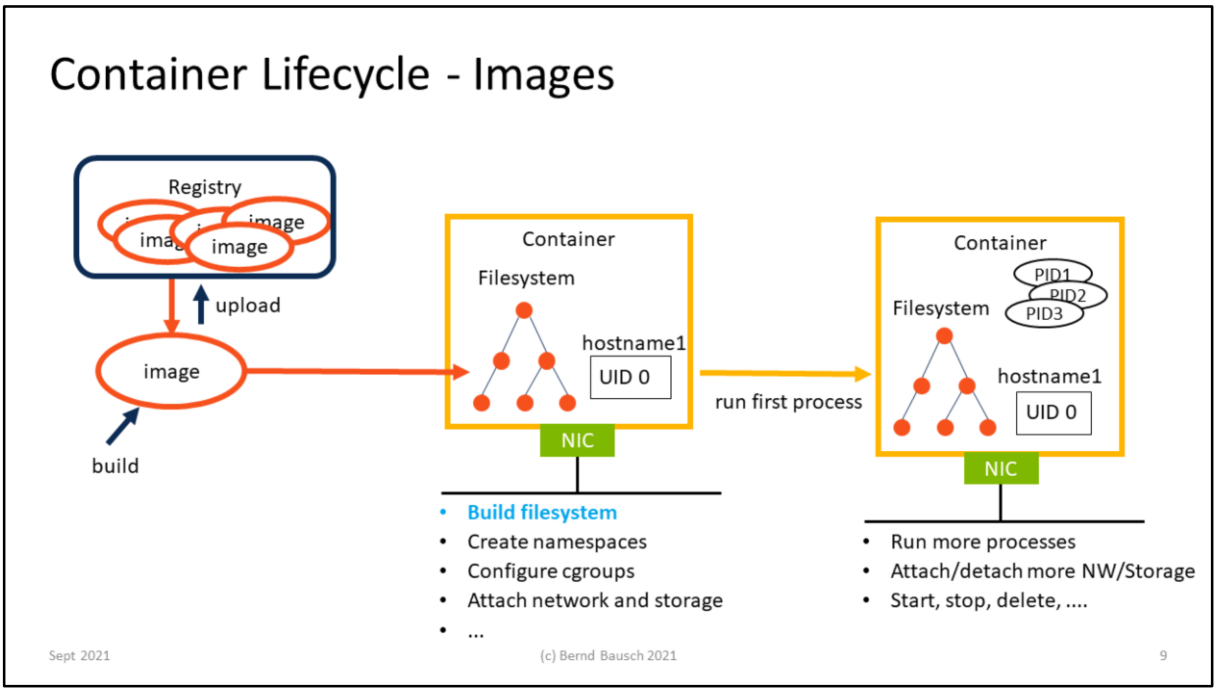
How is a container set up? Before we can run containerized processes, namespaces and CGroups must be configured, and the container's filesystem must be populated. The container must get its network interfaces, which have to be connected to the outside network. It also may get so-called volumes, storage devices that can be attached to several containers.

After that, we are ready to run processes in the container. Normally, a single process is launched, for example a shell or the systemd process, which then runs other processes, such as a web server or the processes needed for a full Linux distribution. We say that we "start the container", although in reality we start processes inside the container. "Stopping the container" means killing or suspending those processes, and deleting the container means removing its filesystem, namespaces, CGroups etc.

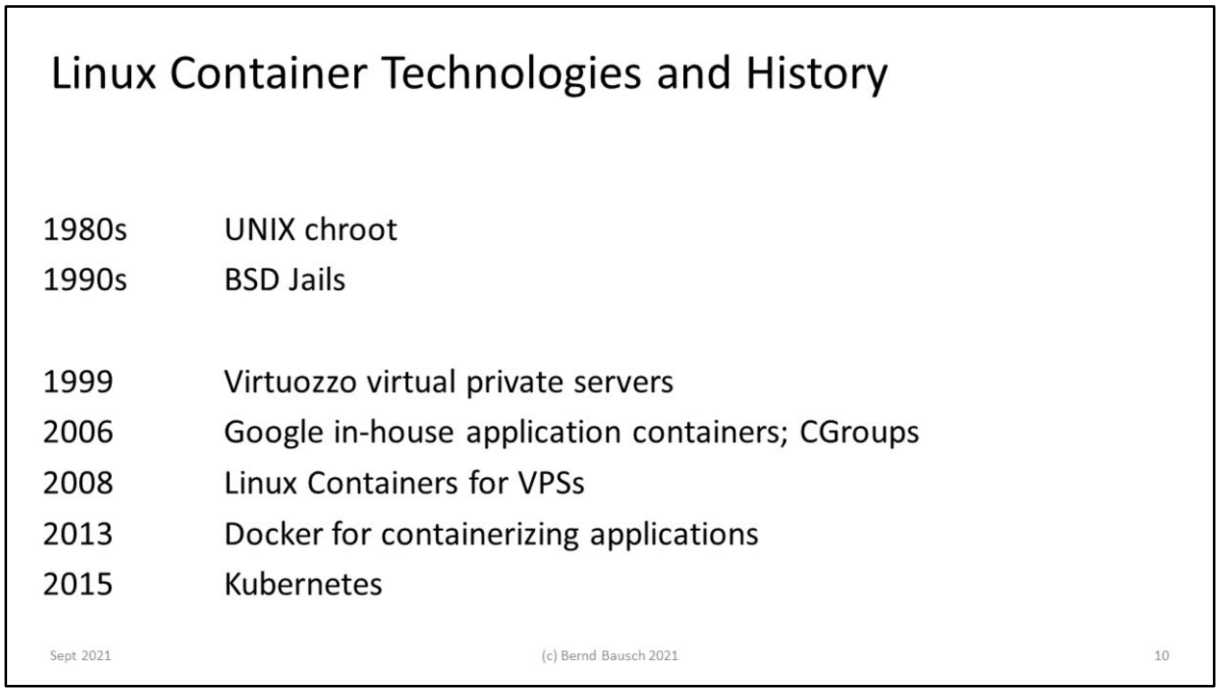


When you perform a Linux installation, the installation tool generates and copies files to the root filesystem. You could use the same approach to fill the container's filesystem. At least one container technology, the original Linux Containers, does exactly that.

Docker, however, takes the tar archive of an existing filesystem and simply unpacks it at the container filesystem's root. In Docker's terminology, such a filesystem archive is named an **image**. This method is considerably faster than the traditional installation method, but it requires a place where all the images are stored. Docker stores images in a so-called **registry**.



One of the reasons for Docker's success is its strong support for building your own images and deploying your applications in the form of images. We should therefore add image **building**, **uploading** and **downloading** to the tasks that are required during the container's lifecycle.



This is a look at the history of Linux container technologies. The first container implementation on Linux came from a company named Virtuozzo, which still exists. Since 1999, Virtuozzo's business has been light-weight virtual private servers using their own container technology, modeled after existing technologies in IBM mainframes and UNIX.

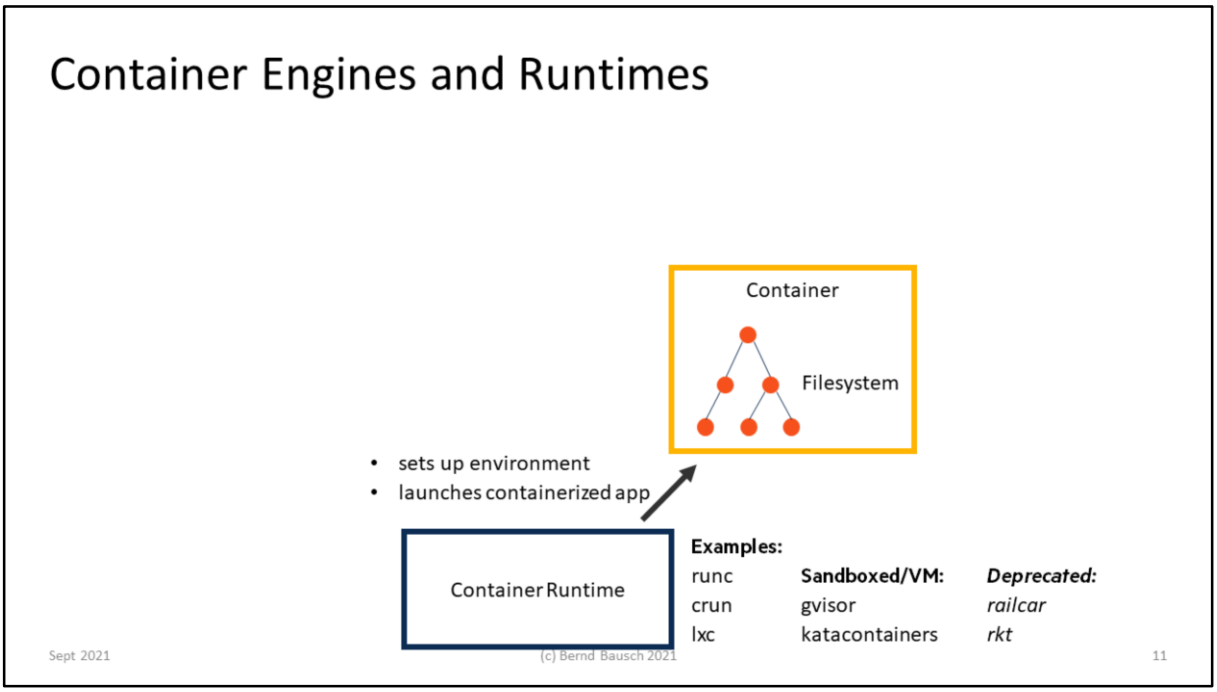
Virtuozzo's technology was partially proprietary; their kernel patches were originally not accepted in the mainstream Linux kernel.

Google has been containerizing its applications since the early 2000s. It added its CGroups technology in the Linux kernel.

The Linux Containers project, mainly sponsored by Canonical, may be the first entirely open-source Linux container implementation. Like Virtuozzo's containers, Linux Containers are aimed at creating virtual private servers.

Docker may be the first widely available *application* container technology. Since its goal is enabling developers to containerize their applications, it has strong support for building images and provides the Docker Hub for making them available to the general public or the development team.

Kubernetes, another technology donated by Google, aims to manage application containers on a large number of hosts. It originally used Docker to run containers. In recent years, other container technologies can be used by Kubernetes.

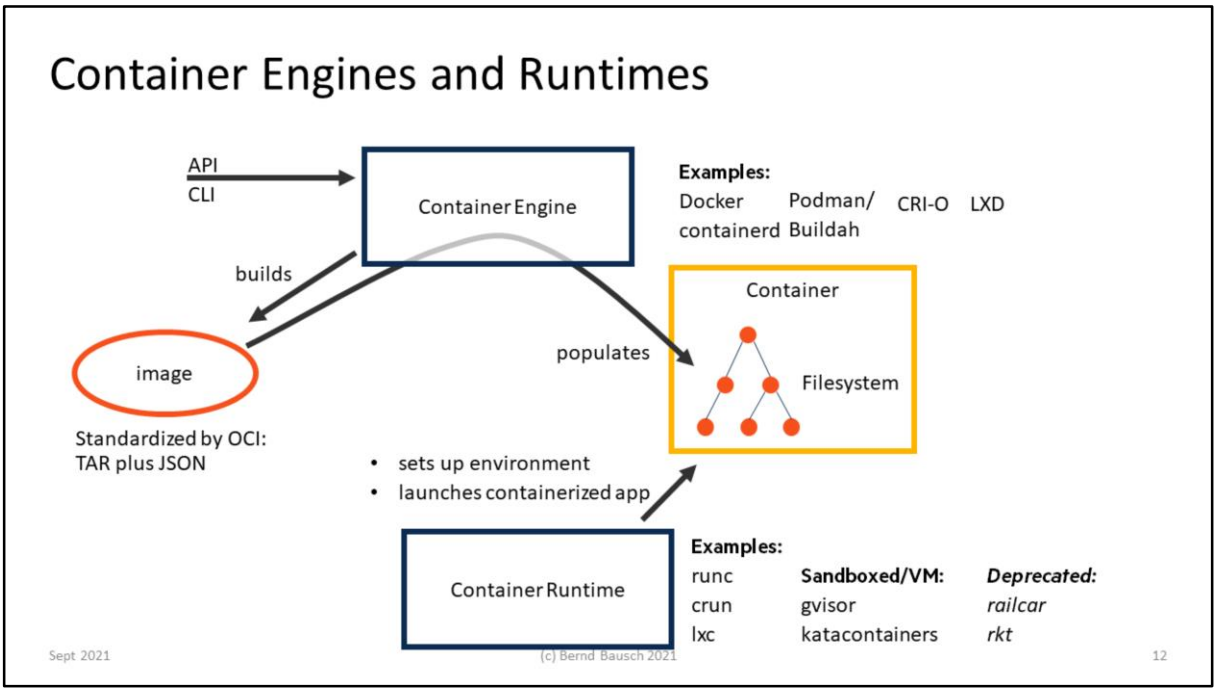


Container lifecycle management includes three important steps:

1. Container setup
2. Running and monitoring containerized processes
3. Image management

Software that deals with tasks 1 and 2, is often named a "**container runtime**". Docker's runtime, which was the result of modularizing the large monolithic Docker daemon, is **runc**.

https://www.docker.com/blog/runc/
https://www.redhat.com/sysadmin/introduction-crun



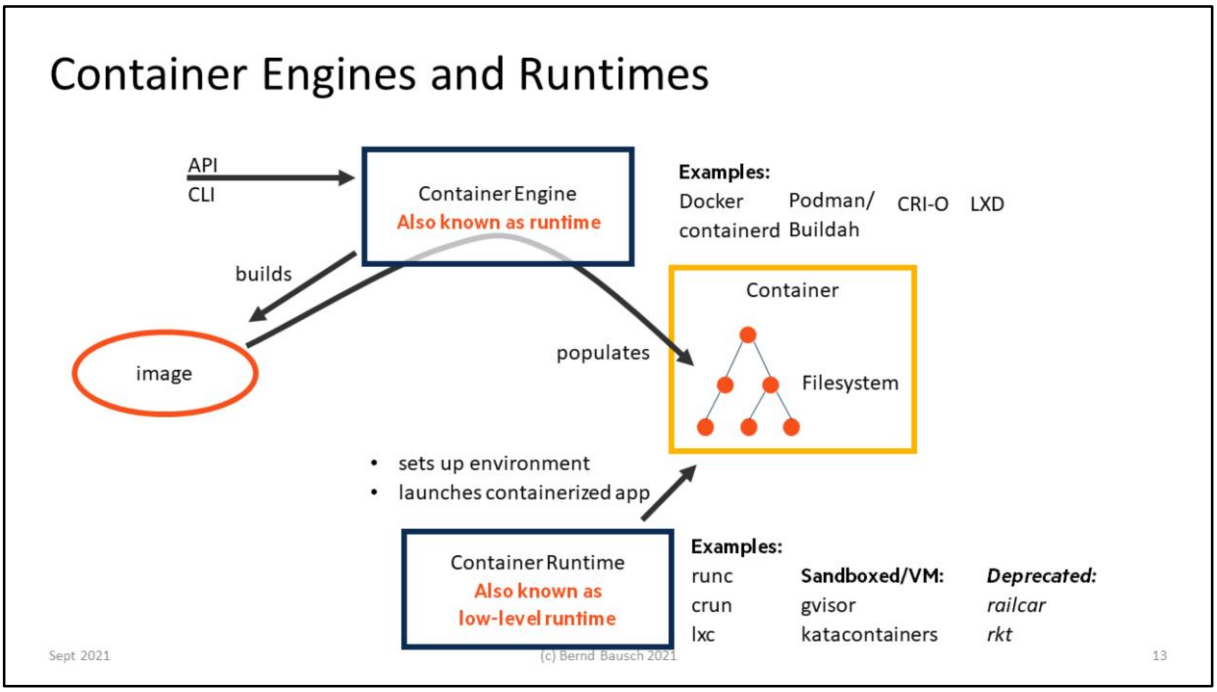
A **container engine** adds features like a RESTful API and the management of images, networks and storage. The Docker Engine is a full-featured container engine. It also allows managing applications that are distributed over a cluster of several container hosts, a feature named **Docker Swarm**.

containerd is the Docker Engine's module that takes care of images and the container lifecycle. For running and managing containers, containerd uses runc. containerd does not manage networking or storage.

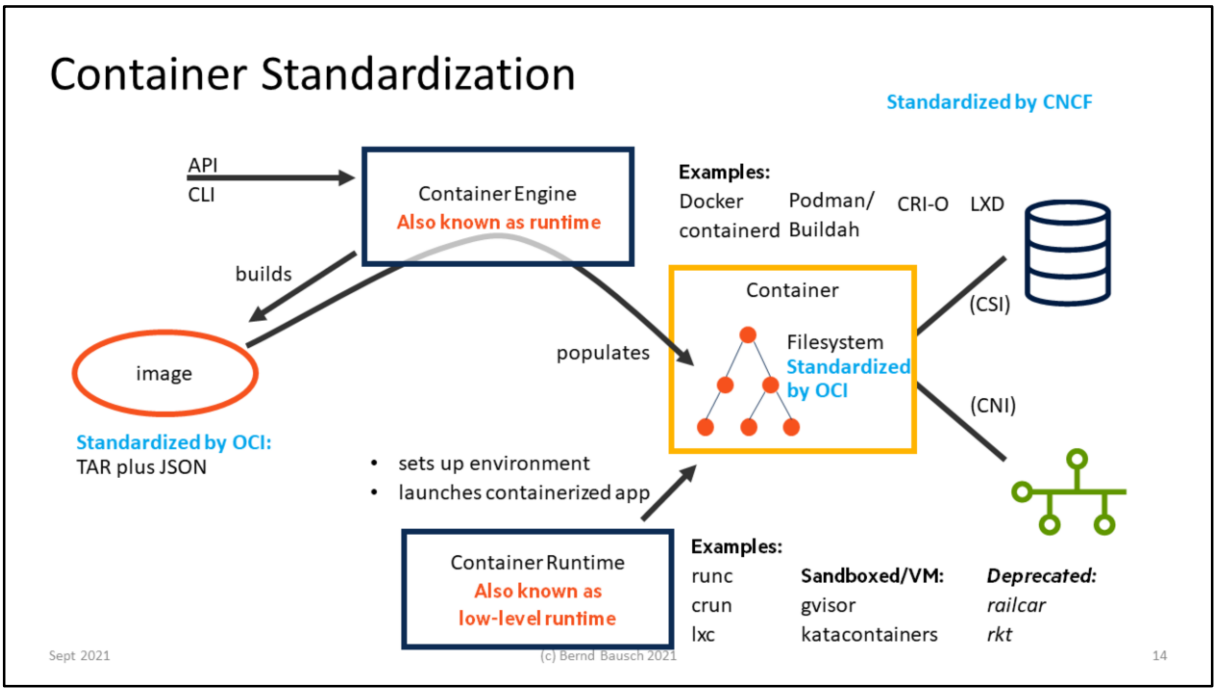
The Docker Engine is implemented as a background process, the docker daemon. Since Kubernetes and Red Hat's OpenShift don't require the docker daemon's full feature set, Red Hat started developing smaller tools for container management, image building and image registry management, named **Podman**, **Buildah** and **Skopeo** respectively. Their underlying technology is runc, and they don't have a daemon.

CRI-O comes from the Kubernetes project and can be compared with containerd. It is an implementation of the standard Container Runtime Interface used by Kubernetes to manage containers. In addition to image and container management, it also takes care of networking via the Container Networking Interface CNI and storage (CSI).

https://www.docker.com/blog/what-is-containerd-runtime/



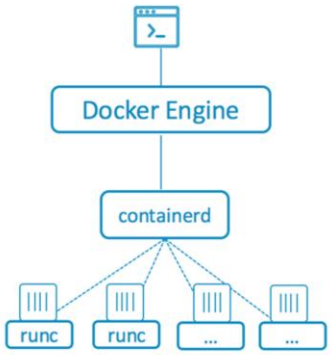
The terminology is not very precise. While the term runtime is used for lower level container management components, higher-level components like containerd are often named runtimes as well.



Two standardization efforts exist.
The Cloud Native Computing Foundation CNCF, which is part of the Linux Foundation, drives the container runtime, networking and storage interfaces (CRI, CNI, CSI), which define how container orchestration engines like Kubernetes communicate with container engines, network implementations and storage solutions. For example, the container runtime component of a Kubernetes cluster can be implemented with Docker, CRI-O or containerd - they are all CRI-compliant.
The Open Containers Initiative, OCI, originally formed by Docker and now also part of the Linux Foundation, standardizes image distribution (tar file including the filesystem and filesystem metadata in JSON format), container filesystem structure and the low-level runtime interface. For example, runc is OCI-compliant, and the images used in Docker and Kubernetes are as well.

https://opencontainers.org/about/overview/
https://caylent.com/understanding-kubernetes-interfaces-cri-cni-csi
https://kubernetes.io/docs/setup/production-environment/container-runtimes/
https://github.com/kubernetes/kubernetes/blob/242a97307b34076d5d8f5bbeb154fa4d97c9ef1d/docs/devel/container-runtime-interface.md
https://github.com/containernetworking/cni
https://github.com/container-storage-interface/spec/blob/master/spec.md

Docker, containerd and runc



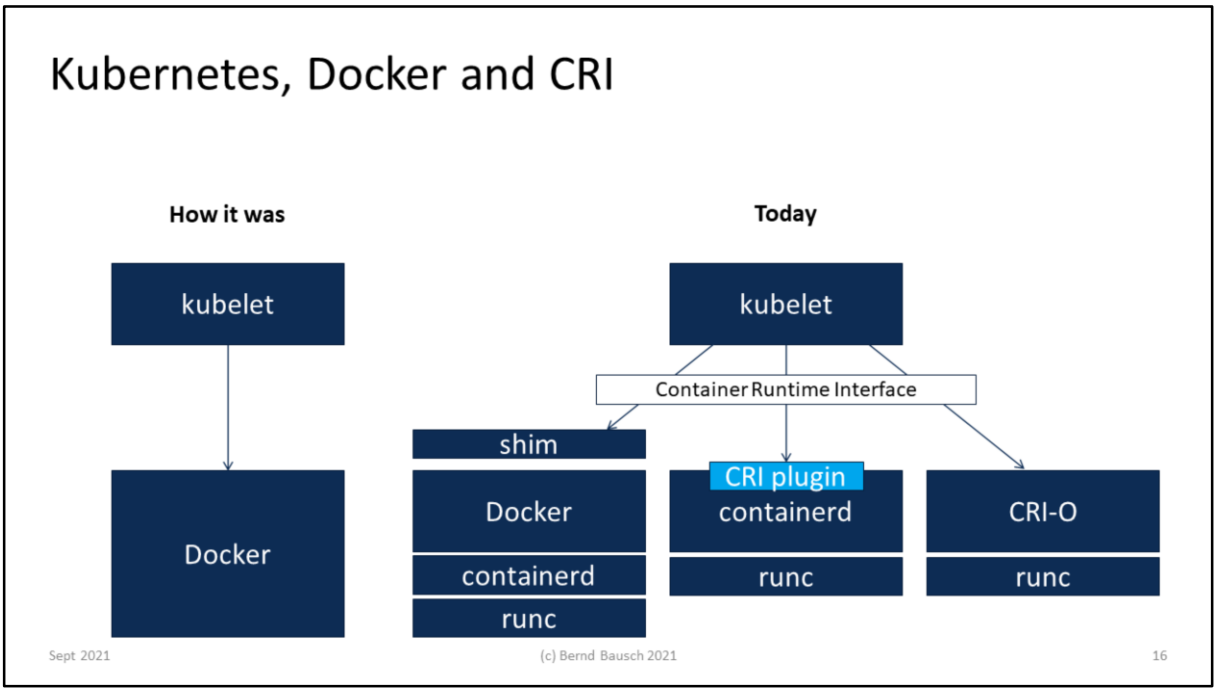
Same Docker UI and commands

User interacts with the Docker Engine

Engine communicates with containerd

containerd spins up runc or other OCI compliant runtime to run containers

From https://www.docker.com/blog/docker-engine-1-11-runc/



The original Kubernetes used Docker for managing images. To make container management pluggable, the Container Runtime Interface was developed, an interface standard that allows a Kubernetes installation to select from several container management solution. In 2021, Kubernetes can use Docker, Docker's containerd or Kubernetes' CRI-O.

CRI-O not only manages the containers, but also network and storage to which containers are attached. It does however not build and manage images. In Red Hat's Kubernetes distribution OpenShift, Podman, Buildah and Skopeo handle these tasks.

CRI-O can use runc or katacontainers as the underlying container runtime.

Standards For Container Technologies

- Founded by Docker and CoreOS
  Now Linux Foundation

- Image specification
- Filesystem runtime specification

- Part of Linux Foundation

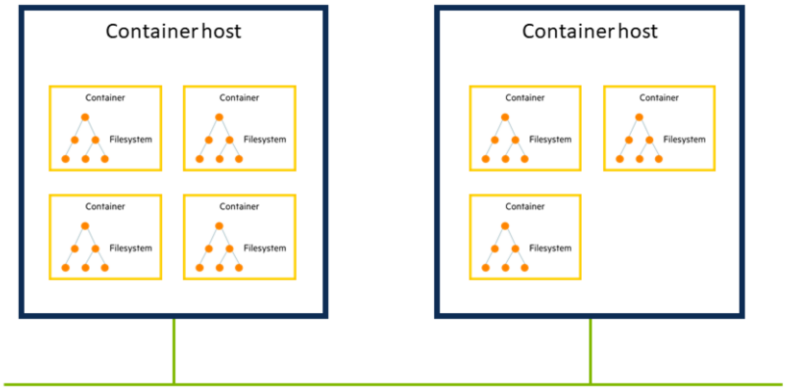
- Kubernetes
- Cloud Storage Interface CSI
- Cloud Networking Interface CNI
- Container runtimes/engines:
  - CRI-O, containerd

Sept 2021 (c) Bernd Bausch 2021 17

For OCI, see https://opencontainers.org and https://github.com/opencontainers
For CNCF, see https://www.cncf.io, https://www.cncf.io/projects and https://github.com/cncf. CNCF oversees a large number of other standards and technologies, including etcd, Helm, and fluentd.

Container Orchestration

Docker Swarm
Kubernetes
Mesos
LXD



Sept 2021 (c) Bernd Bausch 2021 18

Large installations spread containers over several hosts. Such architectures raise a number of challenges, for example:

- how should we determine the host for a new container
- what happens when a container host goes down
- how are containers on separate hosts networked together
- how can storage be shared among containers on different hosts

Container Orchestration Engines manage such large container architectures. Docker Swarm, Kubernetes, Mesos and LXD are examples for COEs. They will be covered in the last courses module. As you learned earlier, the CNCF's CRI, CNI and CSI attempt to standardize COEs' communications with runtime, network and storage technologies.