Containerization: an OS level overview

Giovanni Lupi

Politecnico di Milano
Milan, Italy
giovanniedoardo.lupi@mail.polimi.it

1 Introduction

Containerization is the packaging into a single executable of software code with all the operating system libraries, dependencies and the related configuration files required to run it. Using this method, it's possible to create a single portable lightweight executable, called a container, that is capable of running consistently on any infrastructure without the need to refactor it for each environment.

The concept of containerization and process isolation is a few decades old. However, the introduction of the open-source Docker Engine in 2013, with its easy-to-use command-line tools and overall approach, greatly accelerated the interest and employment of this technology among developers.

These recent developments offered a new alternative to virtualization compared to the more traditional Virtual Machines (VMs). Containerization is in fact a particular form of virtualization where applications run in isolated user spaces (the containers) while using the same shared operating system.

Unlike the majority of the resources currently available in the literature, that mostly focus on getting the reader started with the use of specific tools like Docker, this document aims to provide an overview of the overall containerization process at the operating system level.

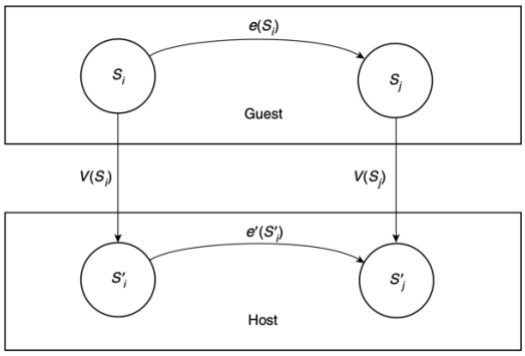
The analysis will first describe the basic concepts of virtualization and the theory behind the use of VMs. It will then move to the use of containers and their main differences with respect to VMs. The next part of the document, perhaps the most interesting one, explains how containers can be created in Linux through the use of basic kernel mechanisms. This breakdown is what really allows one to understand the working principles underlying the Docker tool, which will be addressed in the last part of the document.

2 Virtualization basics

Virtualization is the process of creating a software-based, virtual representation of a set of resources at the same abstraction level. Such resources most commonly include hardware platforms, operating systems, storage devices, and computer network resources. The goal of virtualization is usually one of the following: higher levels of performance, scalability, reliability, availability, agility, or to create a unified security and management domain.

More formally, virtualization is the construction of an isomorphism that maps a virtual *guest* system to a real *host*. This isomorphism maps the guest state to the host state, and for a sequence of operations *e* that

modifies the state in the guest, there is a corresponding sequence of operations *e'* in the host that performs an equivalent modification to the host's state.



A distinctive characteristic of virtualization is that it can make a single large resource, or entire computing environment, appear to be many smaller ones, or vice versa. Let's consider the example of a hard disk. In certain scenarios, it might be desirable to partition a single large hard disk into a number of smaller virtual disks. The virtualizing software is capable of mapping the virtual disks content to the real disk through the isomorphic function *V*. A write to a virtual disk, represented as operation *e* in the picture, is mirrored by a disk write in the real disk of the host system, represented as operation *e'*.

The concept of virtualization can be applied not only to subsystems such as disks, but also to an entire machine. Perhaps the most notable use case of virtualization is to divide the hardware elements of a computing platform into multiple virtual computers, called virtual machines (VMs.) Each VM runs its own operating system and behaves like an independent computer, even though it is running on just a portion of the actual underlying hardware.

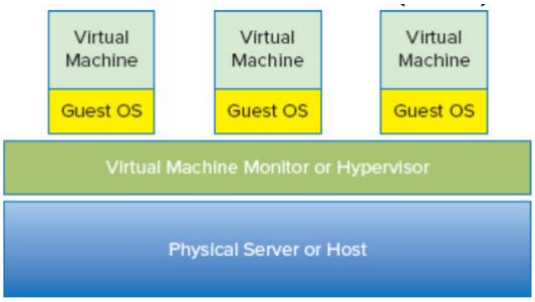
The idea of slicing physical resource by means of VMs, a practice also known as hardware (or platform) virtualization, will be the focus of the next section.

3 Virtual Machines

Hardware virtualization, also called platform virtualization, refers to the creation of a virtual machine that acts like a real computer with an operating system. A virtual machine is the emulated equivalent of a computer system that runs on top of another system. Of course, multiple VMs can run on top of a same shared system.

It is worth noting that, while some VMs are built to match the architecture of an existing machine, others don't have a correspondence with a real one. For example, it's possible to develop a VM specifically tailored to a high-level programming language. Programs written in that language can then be compiled to executables targeted at that VM. A successful application of this principle is the Java Virtual Machine (JVM.)

A VM can virtualize all hardware resources, including processors, memory, storage and network connectivity. The software that provides the environment in which the VMs operate is called Virtual Machine Monitor (VMM), or more commonly hypervisor. For each VM under its management, the VMM assigns some physical and network resources, sets up other virtual devices and starts a guest kernel with access to these resources.



A VMM fundamentally needs to display three properties:

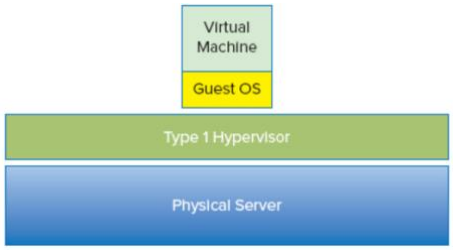
- Fidelity : the environment created for the VM must be identical to the original physical machine.
- Isolation (or Safety) : the VMM must have complete control of the system resources.
- Performance : there should be little difference in performance between the VM and its physical equivalent.

In reality, the third one is only a soft requirement and the VMMs that can satisfy it are often called efficient VMMs.

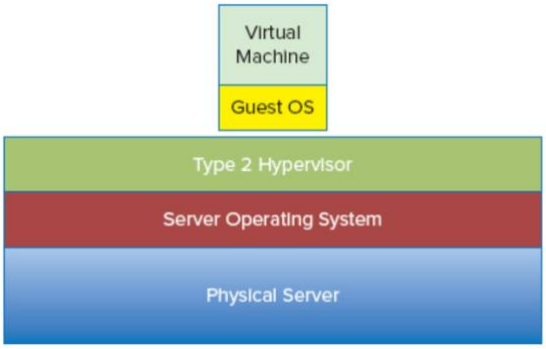
3.1 Hypervisors

In recent years, the term virtual machine manager has fallen out of use, and has been replaced with the term hypervisor. The hypervisor is a software layer with a fairly simple structure, that resides below the virtual machines, named *guests*, and above the physical hardware, named *host*. There are however two classes of hypervisors, differing in the way they are deployed: *Type 1* and *Type 2*.

Type 1 hypervisors run directly on the physical hardware without an operating system beneath it. This implementation is also referred to as *bare-metal*. The ability of Type 1 hypervisors to communicate directly with the hardware resources makes them more efficient than Type 2 hypervisors. Because of this reduced overhead, Type 1 hypervisors can generally run a larger number of separate VMs on the same host.



Type 2 hypervisors are applications that run on top of a traditional operating system. Instead of interacting directly with the hardware, these types of hypervisors leverage the capability of the OS to handle the hardware resources. The major benefit of this approach is that it can support a wide range of hardware resources, that is inherited from the OS it uses. However, because of the extra layer between the hypervisor and the hardware, this method is less efficient than Type 1 hypervisors. In fact, every time a VM needs to perform any hardware interaction, the request is handed off to the hypervisor. Type 2 hypervisors need in turn to hand off the request to the underlying OS. The OS then serves the request and passes the information back to the hypervisor, who delivers the information back to the guest. Hence, this mechanism adds two additional steps and a considerable overhead to every transaction



Without a hypervisor, more than one OS from multiple VMs would want simultaneous control of the hardware. Therefore, the hypervisor must first perform a hardware abstraction, fooling the guest into believing it can interact directly with the physical devices of the host. Then, it also needs to balance the workload among the multiple guests.

In some sense, the hypervisor itself can be considered a sort of operating system, that instead of serving normal programs and applications handles the requests of entire (virtual) machines. For example, let's imagine that an application running in a guest VM needs to perform an I/O operation like a disk read. The guest OS makes a read to the disk it sees. Here, the hypervisor traps the read call and translates it to a real physical equivalent that is passed to the storage system. When the response returns, the hypervisor passes the data back to the guest OS, which receives it as if it came directly from the physical device. Of course, as any OS, the hypervisor has a resource scheduling process that ensures the requested resources are always available in a reasonable manner.

3.2 Additional notes on VMs

Virtual machines are a fundamental component of virtualization. With some ambiguity with respect to the containerization technology, it is sometimes said that VMs act as containers for traditional operating systems and applications running on top of a hypervisor on a physical machine.

In practice, a VM is nothing more than a set of files describing the machine to emulate. The configuration files mostly list which hardware resources are available for that VM to use. However, when running, the VM itself doesn't know that these devices are actually virtual. Therefore, there are two different views of a single VM. From the outside, a VM is just a set of files that describe its composition and configuration. From the inside, the view is identical to being inside a physical machine.

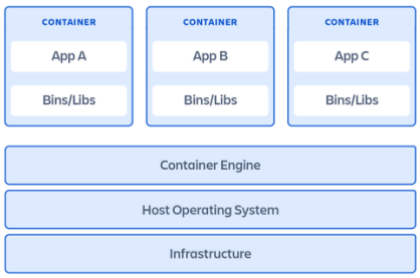
Many VMs can run simultaneously inside a single physical machine and these VMs can also run many different operating systems supporting different applications. While the kernel of a regular OS has to make sure that one application can't access memory that was already assigned to another one, the hypervisor doesn't need to handle such circumstances. In fact, VMs never share memory. As a result, hypervisors are considerably simpler than full kernels and VMs are capable of achieving strong isolation boundaries.

4 Containerization basics

Containerization allows multiple isolated user space instances, named containers, to be run simultaneously on a single host. Each container represents a fully functional and portable computing environment surrounding the application and keeping it independent from other parallelly running environments. Every container simulates a different software application and runs isolated processes by bundling related configuration files, libraries and dependencies.

Collectively, multiple containers running on the same host share a common OS kernel. In fact, containerization is effectively a form of operating-system-level virtualization. This form of virtualization is an OS-level feature in which the kernel allows the existence of multiple isolated user space instances and assigns to them a subset of the overall OS resources. Each of these instances looks like a real machine from

the point of view of the programs running inside them, while in reality each container can only see the contents and devices assigned to it.



Therefore, the key point to understand is that each container running on the same host shares access to the host's kernel. The container itself instead doesn't contain a full kernel. However, containers possess just enough operating system level functionalities to make them operable, which generally include filesystem objects or OS-related files. For example, each container might include the Linux executables necessary to run basic shell commands within its filesystem. Since containers are always optimized to be as lightweight as possible, the only commands and packages available are the strictly necessary ones. For example, a shell running inside a container might be capable to run the `ls` command, but not the `ping` command.

4.1 Main advantages of containers

As the Docker documentation reports, by bundling together an application and all its dependencies and isolating its execution from the rest of the machine it's running on, containers allow developers to "build once, run anywhere". In other terms, the use of containerized application greatly improves portability and simplifies the deployment process.

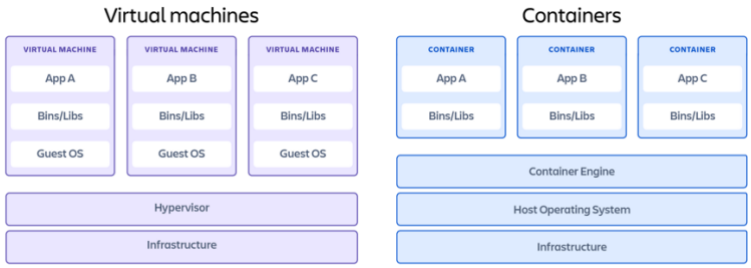
A consequence of the isolated execution is that it's possible to run multiple different containers in parallel without them interfering with each other. Since each container already encompasses all the dependencies required for its application to run, this paradigm effectively allows to isolate the dependencies themselves. This eliminates potential dependency nightmares, where two applications require two different version of the same package.

Being a lightweight and low-overhead construct, it's often possible to run several containers on the same machine without overwhelming it. This drives higher server efficiencies and, in turn, reduces server and licensing costs.

Moreover, since containers provide a form of sandboxing, containerized applications are generally considered more secure than their counterparts not running in a container.

4.2 Containers vs VMs

VMs and containers both need a host to run on. The fundamental difference between them is that, at high level, hypervisors perform hardware virtualization by carving up physical hardware resources into virtual versions (the VMs). On the other hand, containerization is a form of OS virtualization, that doesn’t rely on a hypervisor and carves instead OS resources into virtual versions (the containers). In other terms, virtual machines virtualize an entire machine down to the hardware layers, containers only virtualize software layers above the operating system level. Therefore, instead of running a full-blown OS like a VM, containers share the OS/kernel with the host they’re running on.



Since containers don't need a full kernel to run, their use can greatly reduce licensing costs. For the same reason, their start-up times can also be several orders of magnitude shorter than VMs. In fact, the only thing that has an impact on a container's start-up time is the time required to start the application it's running.

Another thing to consider is that every OS running on a separate VM has its own considerable overhead. For example, every OS consumes some CPU, RAM and storage resources. The use of a hypervisor can introduce some ulterior performance issues. Containers rely instead on the single OS running on the host. This way the overhead is considerably reduced, so that it's possible to run on a single shared host a significantly greater number of containers than VMs.

Containers also offer a quicker and more convenient way to port applications in cloud scenarios. It would of course also be possible to build an entire machine image for the purpose of running the target application. However, this procedure would be significantly slower and thus is seldom used by developers.

Lastly, from a security standpoint, each OS run by a VM represents a possible attack surface. Hence, using a single OS allows containers to reduce the attack surface.

So overall, the container model appears to be leaner and more efficient than the VM one. It allows one to pack more applications onto less resources, start them faster and pay less in licensing and admin costs. However, containers also have some disadvantages with respect to VMs, which can be particularly relevant in certain scenarios.

The inability of containers to run a guest operating system different from the host one represents by all means a lack of flexibility.

As already mentioned, a container is fundamentally a set of isolated processes with a restricted view. The simple fact that all containers share a kernel necessarily means that the level of isolation they can achieve is weaker compared to VMs. This lower lever of isolation also translates to weaker security boundaries compared to VMs, so that more work is generally required to secure containers.

So, when choosing whether to use VMs or containers, developers must take into account an important series of trade-offs between all the aforementioned factors.

5 Containers in Linux

Modern containers started in the Linux world and are the product of an immense amount of work from a wide variety of people over a long period of time. At the most basic level, a container is a set of isolated processes with a restricted view, that is constructed using a combination of features from the Linux kernel. The essential Linux kernel mechanisms that are used to limit processes' access to host resources, and therefore build containers, are:

- *namespaces* : limit what the processes inside the container can see.
- *chroot* : change the root limit for the set of files and directories that the container can see.
- *cgroup* : control the resources the container can access.

The next sections will individually describe each of these features.

5.1 Namespaces

Kernel namespaces are at the very heart of containers. They slice up an operating system so that it looks and feels like multiple isolated operating systems. In particular, a namespace is a low-level mechanism for

isolating the machine resources that a process is aware of. The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

Currently, the Linux kernel supports 8 different types of namespaces, each capable of isolating a different system resource:

- *Unix Timesharing System (UTS)* : isolates the hostname and the domain name
- *PID* : isolates the process ID number space
- *Mount* : isolates the list of mount points
- *Network* : isolates network interfaces
- *Inter-Process Communications (IPC)* : isolates inter-process communication mechanisms
- *User* : isolates user and group IDs
- *Cgroup* : isolates the cgroup root directory
- *Time* : isolates system clocks

A process is always in exactly one namespace of each type. When the Linux system is started, it has a single namespace of each type. However, it's possible to create additional namespaces and assign processes into them. An organized collection of these namespaces is what we call a container.

All the namespaces used by a machine can be seen using the `lsns` command. Since this command reads information directly from the `/proc` filesystem, it may return incomplete information for non-root users. Thus, `lsns` should always be run with root privilege to get the complete picture.

The most common way to create a new namespace is by using one of the following system calls:

- *clone* : creates a new process. If the flags argument of the call specifies one or more of the `CLONE_NEW*` flags, then new namespaces of the appropriate type are created for each flag, and the child process is made a member of those namespaces.
- *setns* : allows the calling process to join an existing namespace. The namespace to join is specified via a file descriptor.
- *unshare* : moves the calling process to a new namespace. If the flags argument of the call specifies one or more of the `CLONE_NEW*` flags, then new namespaces of the appropriate type are created for each flag, and the calling process is made a member of those namespaces.

The files in the `/proc/sys/user` directory expose per-user limits on the number of namespaces of various types that can be created. For example, the *max_cgroup_namespaces* file defines the maximum number of cgroup namespaces that may be created in the user namespace. For the initial user namespace, the default value in each of these files is half the limit on the number of threads that may be created, which can be found in `/proc/sys/kernel/threads-max`. In all descendant user namespaces, the default value in each file is `MAXINT`. The values in these files are modifiable by privileged processes. Upon encountering the established limits, *clone* and *unshare* fail with the error `ENOSPC`.

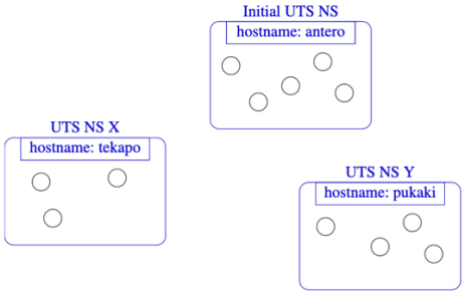
A namespace is automatically torn down when the last process in the namespace terminates or leaves it. However, there are a few other factors that may prolong the lifetime of a namespace even when it no longer has any member processes. For example, the namespace might be hierarchical and have a child namespace.

The next paragraphs will analyze in turn each type of Linux namespace.

5.1.1 UTS

This is perhaps the simplest type of namespace. UTS namespaces provide isolation of two system identifiers: the hostname and the NIS domain name. Changes made to these identifiers are visible to all other processes in the same UTS namespace, but are not visible to processes in other UTS namespaces. So, by putting a process in its own UTS namespace, it's possible to change the hostname for that process independently of the hostname of the machine or virtual machine on which it's running.

A running system may have multiple UTS namespaces instances. Each namespace instance has its own hostname and domain name, so that all processes within a single instance share the same hostname and domain name.



This is a key component of the way containers work. Namespaces give them a set of resources (in this case the hostname) that are independent of the host machine, and of other containers.

5.1.2 PID

PID namespaces isolate the process ID number space and restrict the set of process IDs that are visible. This means that processes belonging to different PID namespaces can have the same PID.

The PID numbers in a new PID namespace always start at 1, similarly to a standalone system. Newly created process will acquire PIDs that are unique within the namespace. The first process created in a new namespace has the PID 1, and is the *init* process for the namespace. If the *init* process of a PID namespace terminates, the kernel terminates all of the processes in the namespace via a `SIGKILL` signal. This behavior reflects the fact that the *init* process is essential for the correct operation of a PID namespace.

A process's PID namespace membership is determined when the process is created and cannot be changed thereafter.

It is worth noting that, by using a new process ID namespace alone, tools such as `ps` won't work correctly. In fact, the `ps` command looks in the `/proc` folder of the host machine for information about running processes, regardless of the process ID namespace it's running in. In order to have `ps` return only the information about the processes inside the new namespace, there needs to be a separate copy of the `/proc` directory, where the kernel can write information about the namespaced processes. Given that `/proc` is a directory directly under root, this means changing the namespace's root directory. This is achieved by using the Linux `chroot` command. Once the container owns its own `/proc` directory, independent of the one of the host, the folder must be mounted as a pseudo-filesystem of type *proc* in order for it to be populated with process information. This is done using the `mount` command.

5.1.3 Mount

Mount namespaces provide isolation of the list of mounts seen by the processes in each namespace instance. Thus, the processes in each of the mount namespace instances will see distinct single-directory hierarchies. Giving a container its own mount namespace allows it to achieve separation between the host filesystem mounts and its own ones.

It's worth noting that the kernel uses by default the `/proc/[pid]/mounts` directory to communicate information about mount points for each process. If a process is created within its own mount namespace, but it is using the host's `/proc` directory, then its `/proc/[pid]/mounts` file will include all the preexisting host mounts. To get a fully isolated set of mounts for the containerized process, the creation of a new mount namespace must be combined with a new root filesystem and a new *proc* mount.

5.1.4 Network

Network namespaces provide isolation of the system resources associated with networking, thus allowing a container to have its own view of network interfaces and routing tables.

A physical network device can live in exactly one network namespace. When a network namespace is freed (i.e., when the last process in the namespace terminates), its physical network devices are moved back to the initial network namespace.

5.1.5 IPC

In Linux it's possible to communicate between different processes by giving them access to a shared range of memory, or by using a shared message queue. The two processes need to be members of the same inter-process communications (IPC) namespace for them to have access to the same set of identifiers for these mechanisms.

However, generally speaking, it's often considered inappropriate for containers to be able to access one another's shared memory. Thus, it's common practice to give them their own IPC namespaces.

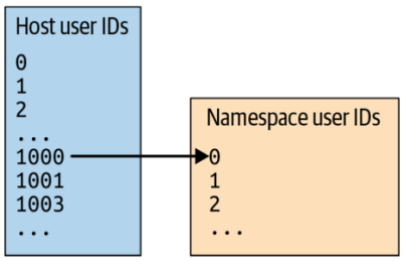
When an IPC namespace is destroyed (i.e., when the last process that is a member of the namespace terminates), all IPC objects in the namespace are automatically destroyed.

5.1.6 User

User namespaces isolate security-related identifiers and attributes, in particular, user IDs and group IDs, the root directory, keys, and capabilities.

Similarly to process IDs, the users and groups still exist on the host, but they can have different IDs compared to the ones inside a user namespace. One of the main benefits is that a process can have a normal unprivileged user ID outside a user namespace while at the same time having a user ID of 0 inside the namespace. This means that the process can have full privileges for operations inside the user namespace, but is simultaneously unprivileged for operations outside of it. This represents a huge advantage from a security perspective, as any attacker capable of escaping from a container to the host will only have a non-root, unprivileged identity. A container that is created and managed by an unprivileged user on the host machine is called a rootless container.

When a user namespace is created, it starts out without a mapping of user IDs (group IDs) to the parent user namespace. The `/proc/[pid]/uid_map` and `/proc/[pid]/gid_map` files expose the mappings for user and group IDs inside the user namespace for the process pid. Hence, when a new user namespace is created, a mapping must be put in place between IDs inside and outside the namespace by modifying the suitable files. These files can only be written once, but can later be read to view the mappings again.



Capabilities are a Linux per-thread attribute, which can be independently enabled and disabled to allow each thread to perform certain privileged operations without providing them all root permissions. Whenever a process is assigned to a new user namespace, or a previously existing one, that process automatically gains a full set of capabilities in that namespace. On the other hand, that process has no capabilities in the parent or any other user namespace, even if the new namespace is created or joined by the root user. Having a capability inside a user namespace permits a process to perform operations (that require privilege) only

on resources governed by that namespace. So, when creating a process using several new namespaces, the user namespace will be created first, so that the process is granted the full capability set that permits it to create other namespaces.

5.1.7 Cgroup

Cgroup namespaces act in some sense as `chroot` for the cgroup filesystem. Adding a process to a new cgroup namespace prevents it from seeing the cgroup configuration higher up in the hierarchy, effectively changing its root cgroup hierarchy folder. Thus, each cgroup namespace has its own cgroup root directory.

When a process creates a new cgroup namespace, either using clone or unshare, its current cgroup directory becomes the cgroup root directory of the new namespace.

The reader should pay attention not to confuse cgroup namespaces with the actual cgroup Linux feature. Cgroup namespaces only control which cgroups are visible within a certain namespace. The namespace cannot assign resource restrictions. Limits to the use of resources must be set using the cgroup mechanism itself, which is another fundamental kernel feature used to build containers.

5.1.8 Time

This is the most recent type of namespaces introduced into the Linux kernel. Time namespaces virtualize the values of two system clocks:

- `CLOCK_MONOTONIC` : a non-settable clock that represents monotonic time since, as described by POSIX, "some unspecified point in the past"
- `CLOCK_BOOTTIME` : a non-settable clock that is identical to `CLOCK_MONOTONIC`, except that it also includes any time that the system is suspended

Thus, the processes in a time namespace share per-namespace values for these clocks. This affects various APIs that measure against these clocks.

5.2 chroot

From within a container, it's not possible to see the host's entire filesystem; instead, only a subset is visible, because the root directory gets changed when the container is created. The root directory can be modified in Linux using the `chroot` command, which changes the root directory of the calling process to the new one specified. This effectively moves the root directory for the current process to point to some other location within the filesystem. Once the `chroot` command is executed, the calling process loses access to anything that was higher in the file hierarchy than its current root directory, since there is no way to go any higher than the root within the filesystem.

An important consequence of using this command is that, if an executable isn't present within the filesystem rooted at the directory specified by `chroot`, the containerized process won't be able to find it and run it. For example, let's create a new directory with the following command:

```
mkdir new_root
```

We can then change the root directory using the `chroot` shell command. This command not only changes the root directory to the one indicated, but also executes a specified command inside of it. If no command is specified, then `/bin/sh` is automatically executed. So, let's try to run:

```
sudo chroot new_root
```

This command however fails. In fact, once the running process is inside the new root directory, there is no `bin` folder inside of it, which results in the impossibility to run a `/bin/sh` shell. This is the exact reason why containers, while not containing a full kernel, still encapsulate a minimal set of OS-related files and filesystem objects.

A common solution is that of including in the new root folder the filesystem of a light Linux distribution, such as *Alpine Linux*. For example, supposing the *Alpine Linux* filesystem is available in the `alpine` directory, we could successfully run the following command:

```
sudo chroot alpine sh
```

It’s worth noting that trying to run a bash shell through the previous command would instead result in an error. This is because *Alpine* is such a simple Linux distribution that it doesn’t include it in its filesystem. Containers are optimized by design to be as lightweight as possible, so that they only offer a minimal set of commands and packages.

5.3 cgroup

While namespaces are about isolation, control groups (cgroups) are instead about setting limits. Using the cgroup mechanism it's possible to constrain the resources, such as memory, CPU, and network input/output, that a group of processes can use. Since containers run as regular Linux processes, cgroups are fundamental kernel feature that can be used to limit the resources available to each container.

From a security perspective, well-tuned cgroups can ensure that one process can’t affect the behavior of other processes by consuming all the host resources, for example using all the CPU or memory to starve other applications. There is also a control group called pid for limiting the total number of processes allowed within a control group, which can prevent the effectiveness of a fork bomb.

The use of this kernel feature can be rather complex and explaining its use would require an extensive documentation itself. A detailed survey about its functioning can be found in [8].

6 Docker

As already mentioned in the Virtual Machines chapter, the VM model has by several shortcomings. These limitations have in recent times motivated companies to switch to container technologies, resulting in an explosion of their employment.

The concepts around containers have existed for several years, as fundamentally they are implemented using a set of features of the Linux kernel. However, implementing a container using these low-level primitives directly can be quite complex and challenging, so that their use remained outside of the realm of most organizations for many years. Thanks to its easy-to-use interface and set of tools, the introduction of Docker is what effectively made the container technology popular and accessible to the masses.

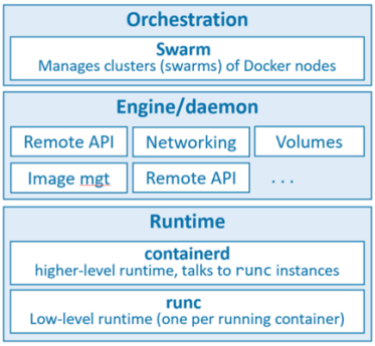
Docker is an open-source containerization engine that automates the build, deployment and management of containerized applications. Docker is indisputably the leading software in the area of containers. Developers have been using it to package their applications, frameworks, and libraries into containers, and then ship them to testers or operations engineers. To these professionals, a container is just a standardized black box, that can be treated equally regardless of the type of application it's running. Thus, Docker containers are a means to package applications and their dependencies in a standardized way. This way, the containerized application has everything it needs to run correctly, so that if a server is capable of running any container, then any other container should be able to run as well.

The next section will give an overview of the architecture that allows Docker to run containers.

6.1 High-level architecture

Most people use the term Docker to refer to the overall technology that runs containers. In reality, from a high-level perspective the Docker architecture can be divided in three main layers:

- The *runtime*
- The *daemon* (also called *engine*)
- The *orchestrator*



The runtime operates at the lowest level and is responsible for starting and stopping containers. Its main tasks include building the appropriate OS-level constructs needed to manage containers, such as namespaces and cgroups. Docker implements a tiered runtime architecture that involves the cooperation of a high-level and a low-level tier. The low-level runtime is called *runc*. Its job is to interface with the underlying OS and start and stop containers. Every running container on a Docker node has a *runc* instance managing it. The higher-level runtime is called *containerd*. This manages the entire lifecycle of a container, including pulling images, creating network interfaces, and managing lower-level *runc* instances. A typical Docker installation comprises a single *containerd* process (`docker-containerd`) controlling the *runc* (`docker-runc`) instances associated with each running container.

The Docker daemon (dockerd) sits above containerd and performs higher-level tasks, such as exposing the Docker remote API, managing images, managing volumes, managing networks, and more. A major job of this daemon is also to provide an easy-to-use standard interface that abstracts the lower levels.

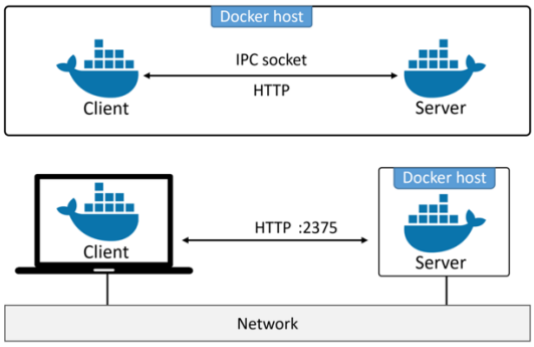
Docker also has native support for managing clusters of nodes running Docker. These clusters are called swarms and the native technology is called Docker Swarm. However, in practice most people prefer using Kubernetes instead of Docker Swarm.

6.2 Client-server model

Docker implements a client-server model. In this paradigm:

- The client component implements the CLI.
- The server component is represented by the daemon. It implements all the functionalities and offers a RESTful interface to the outside world, over which all container operations can be automated.

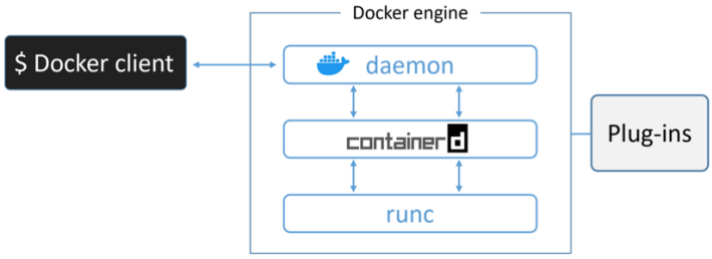
These two components are implemented as separate binaries. In Linux, the client is called `docker` and the daemon is called `dockerd`. A default installation puts them on the same host and configures them to communicate over a local IPC socket at `/var/run/docker.sock`. It's also possible to configure them to communicate over the network. By default, network communications occur over an unsecured HTTP socket on port `2375/tcp`.



Clearly, an insecure configuration like this might be suitable for labs, but is unacceptable for anything else. Therefore, Docker also give the possibility to force the client and daemon to only accept network connections that are secured with TLS.

6.3 Docker engine architecture

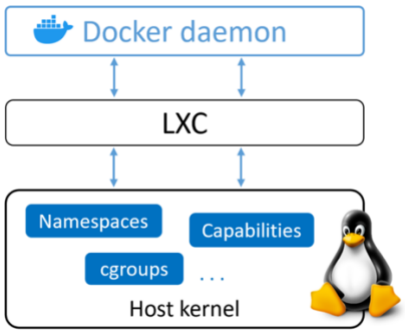
This section will be focused on the analysis of the Docker engine. The Docker engine is the core software that runs and manages containers. It is modular in design and built by connecting many small specialized tools. Currently, the major components that make up the Docker engine are; the Docker *daemon*, *containerd*, *runc*, and various plugins, such as networking and storage. Together, these create and run containers.



However, when Docker was first released, its engine fundamentally had only two major components:

- The *daemon*
- LXC

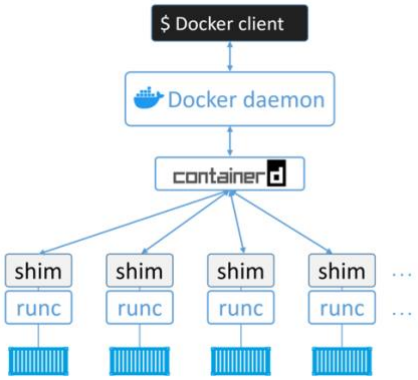
The Docker daemon was a monolithic binary. It contained all of the code for the Docker client, the Docker API, the container runtime, image builds, and more. LXC instead provided the daemon with access to the fundamental building-blocks of containers that exist in the Linux kernel, such as namespaces and cgroups.



The reliance on LXC entailed a few issues from the very beginning. First, LXC is Linux-specific, which is relevant problem for a project that had aspirations of being multi-platform. Second, being reliant on an external tool for something so core to the project was a big risk that could hinder development. This motivated Docker. Inc. to replace LXC with a new tool developed internally, called *libcontainer*. This tool is platform-agnostic and provides Docker with access to the fundamental container building-blocks that exist in the host kernel.

Over time, the use and management of a monolithic daemon became substantially more problematic. Hence, Docker, Inc. began a huge effort to break apart the daemon and modularize it. Following the Unix philosophy, the aim of this work was to break out as much of the functionality as possible from the daemon and re-implement it in smaller specialized tools. This project would then allow these tools to be easily swapped out or re-used by third parties to build new other tools. In the end, all of the container execution

and runtime code was entirely removed from the daemon and refactored into small specialized tools. The following image provides a high-level view of the current Docker engine architecture.



`runc` is a small, lightweight CLI wrapper for *libcontainer*. Fundamentally, `runc` has a single purpose in life: create containers. Being a CLI wrapper, it effectively is a standalone container runtime tool. Hence, downloading and building its binary is theoretically enough to start building `runc` containers. However, the implementation is very minimal and low-level, so it doesn't offer any of the richness provided by the full-blown Docker engine.

`containerd` is the tool that encapsulates all the container execution logic. Originally, `containerd` was intended to be small, lightweight, and designed with the sole purpose of managing container lifecycle operations, such as `start`, `stop`, `pause` or `rm`. However, over time more functionalities were progressively added, including for example image pulls, volumes and networks. One of the main reasons behind this extension was to make `containerd` easier to use in other projects. For example, in projects like Kubernetes, it was beneficial for `containerd` to do additional things like push and pull images. Nevertheless, all the extra functionality is modular and optional, so it's possible to include `containerd` in tools such as Kubernetes, but employing just the pieces needed for its development.

So, in the runtime layer, `containerd` uses `runc` to create new containers. In particular, a single instance of `containerd` forks a new instance of `runc` for every new container to create. However, once each container is created, the parent `runc` process exits. This makes it possible to run hundreds of containers without having to run hundreds of `runc` instances. It is this moment that the `shim` component represented in the previous diagram comes into play. `shim` is a piece of software that resides in between `containerd` and `runc`, facilitating the integration of these counterparts. Once a container's parent `runc` process exits, the associated `containerd-shim` process automatically becomes the container's parent. Some of the main operations that `shim` performs include managing all `STDIN` and `STDOUT` streams and reporting the container's exit status back to a higher-level tool like Docker.

Detaching all the logic and code required to start and manage containers from the daemon, allowed Docker to completely decouple the entire container runtime from the daemon. For this reason, Docker containers are also called *daemonless containers*. An important advantage of *deamonless containers* is that they make it possible to perform maintenance and upgrades to the daemon without impacting running containers. Instead in the old model, where all of container runtime logic was implemented in the daemon, starting and stopping the daemon would kill all running containers on the host, which represented a very relevant problem in production environments.

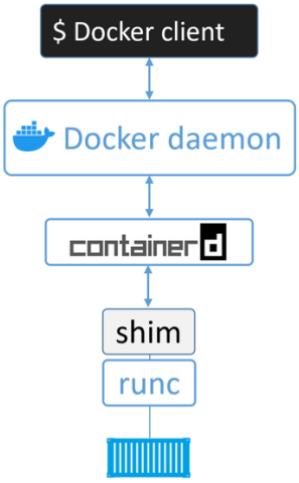
Even though all the execution and runtime code has been progressively stripped out of the daemon and modularized, the daemon still includes some major functionalities, such as image management, image builds, the REST API, authentication, security, core networking and orchestration.

6.4 Starting a new container

To wrap up everything said in the previous section, let's walk through the process of creating a new container. The following Docker CLI command will start a simple new container based on the latest image of *Alpine Linux*.

```
docker container run --name ctr1 -it alpine:latest sh
```

The Docker client converts the CLI command into the appropriate API payload and POSTs them to the API endpoint exposed by the Docker daemon. Once the daemon receives the command to create a new container, it makes a call to `containerd`. `containerd` in turn converts the required Docker image into a suitable format and tells `runc` to use this to create a new container. `runc` interfaces with the OS kernel to pull together all of the constructs necessary to create a container, such as namespaces and cgroups. The container process is then started as a child-process of `runc` and, as soon as it is started, `runc` will exit, allowing `shim` to become its parent process.



6.5 Images

A Docker image is a unit of packaging that contains everything required for an application to run. This includes application code, application dependencies, and OS constructs. Basically, once an application's Docker image has been downloaded, the only other thing required to run it is a computer running Docker.

Informally, a Docker image is like a stopped container. In fact, a container can be stopped to create a new image from it. So, images are considered build-time constructs, whereas containers are run-time constructs. Using an analogy from the development world, images are similar to classes.

The figure above shows a high-level view of the relationship between images and containers. Using the appropriate commands, as `docker container run`, it's possible to start one or more containers from a single image. Once a container has been started from an image, the two constructs become dependent on each other, so that it's impossible to delete the image until the last container using it has been stopped and destroyed. Attempting to delete an image without stopping and destroying all containers using it will inevitably result in an error.

The purpose of a container is to run a single application or service. Since a container only needs the code and dependencies of the service it's running, images tend to be small and stripped of all non-essential parts.

Image also don't contain a full kernel; however, they may contain some OS-related files and filesystem objects.

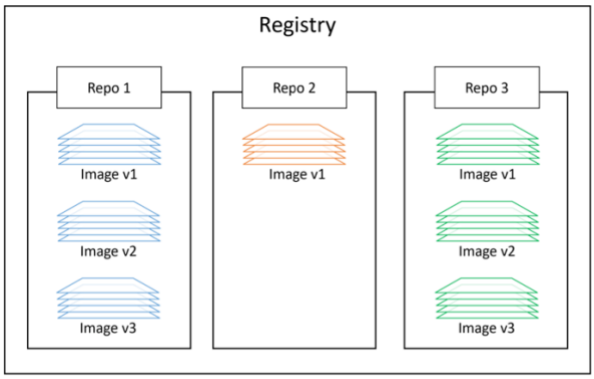
A cleanly installed Docker host has no images in its local repository. The local image repository on a Linux machine is usually located at `/var/lib/docker/<storage-driver>`. The following command lists all images available in the local repository:

```
docker image ls
```

The process of getting images onto a Docker host is called pulling. All the pulled images are automatically added in the Docker host's local repository. The following command can be used to pull the latest image of *Alpine Linux*:

```
docker image pull alpine:latest
```

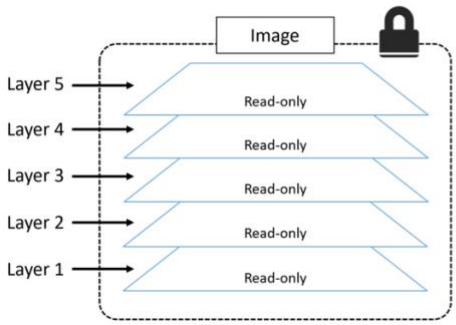
Images are stored in centralized places called image registries, which greatly simplify their sharing and access. The most popular registry is by far Docker Hub. The Docker client is configured to use Docker Hub by default when pushing or pulling images. Of course, other third-party registries exist as well. An image registry, like Docker Hub, contains one or more image repositories. In turn, image repositories contain one or more images. As an example, the following figure shows an image registry with three repositories, and each repository has one or more images.



To address a specific image from a repository, one must provide the repository name and an image tag separated by a colon (:). The format to pull an image from Docker Hub is always `docker image pull <repository>:<tag>`. For example, to pull the image tagged as *latest* from the official *alpine* repository, one could specify `alpine:latest`. It is also possible to pull a specific version of an image, for example by writing `mongo:4.2.6`. While using the pull command, if an image tag is not explicitly specified after the repository name, the image tagged as *latest* will be pulled by default and, if the repository doesn't have an image tagged as *latest*, the command will fail.

6.6 Images and layers

A Docker image is a set of loosely-connected read-only layers, with each layer comprising one or more files. Docker takes care of stacking these layers and representing them as a single unified object. Inside of the image there's a cut-down operating system (OS) and all of the files and dependencies required to run an application.



A possible way to see the layers that comprise an image is to inspect it with the `docker image inspect command`. For example:

```
docker image inspect ubuntu:latest
```

All Docker images start with a base layer, which is most often a cut-down operating system. As changes are made and new content is added, new layers are added on top. When additional layers are added, the image is always the combination of all layers stacked in the order they were added.

Multiple images can, and do, share layers. This leads to a superior efficiency in terms of both space and performance. This means that Docker is capable to recognize when it's being asked to pull an image layer that it already has a local copy of. For example, when downloading an updated version of the same image, generally the two images will share many layers. Thus, only the few layers that are differing will actually be downloaded.

7 Docker kernel interaction

This section focuses on the interaction that happens under the hood between Docker and some of the features from the Linux kernel introduced in section 5.

7.1 chroot

On Linux, the root directory where Docker stores its files is `/var/lib/docker`.

Docker uses storage drivers to store image layers and to store data in the writable layer of a container. The container's writable layer does not persist after the container is deleted, but is suitable for storing ephemeral data, that is data generated at runtime. Different type of drivers can be used and all of them are optimized for space efficiency. The driver used by default is *overlay2*. Thus, all Docker images and containers are stored in the `/var/lib/docker/overlay2` folder. Each directory therein is named using the SHA256 hash of the respective image or container.

To find the location of a specific image, it's possible to use the inspect command: `docker inspect <image>:<tag>`. The output of this command consists of several sections. The *GraphDriver* section contains the *Data* subsection, which in turn can own up to four fields:

- *LowerDir* : these are the image layers assembled in order.
- *UpperDir* : the layer which contains changes made by that container.
- *WorkDir* : an empty directory for internal use.
- *MergedDir* : this is the result of the overlay filesystem. Docker effectively performs `chroot` into this directory when running the container.

Before mentioning containers, we recall that an image is just a build-time construct; in other terms it is similar to a class of an object-oriented programming language. A container can instead be thought as an instance of that class, so multiple containers can be started from the same image. Whenever a container is run from an image, a new folder is created in `/var/lib/docker/overlay2` to accommodate its data. The content of the folder is just a duplicate of the base image data. For example, if multiple containers are started from an *Alpine Linux* image, each container will have its own copy of the *alpine* filesystem. The

location of a container can be found using the inspect command: `docker inspect <container_id>`. The *GraphDriver* section of the output is organized in the same way as images. Therefore, it would be more appropriate to say that each container performs `chroot` into its own *MergedDir* folder. The container will use the constrained filesystem starting at this directory to jail and spawn processes. Clearly, any binaries to be executed must be contained within it. All the mounts that happen inside the container are automatically cleaned up when the container exits.

7.2 Namespaces

By default, when running a container, Docker creates a set of 6 namespaces for it:

- *UTS*
- *PID*
- *Mount*
- *Network*
- *IPC*
- *Cgroup*

Nevertheless, it's also possible to configure the daemon to use the *User* namespace and create rootless containers. One way to achieve that is to start `dockerd` using the `--userns-remap` flag.

7.3 cgroup

Cgroups are used to handle resource allocation for containers. This includes system resources like cpu, memory, and device access. By default, a container has no resource constraints and can use as much of a given resource as the host's kernel scheduler allows. However, Docker provides ways to control how much resources a container can use.

For each container, one cgroup is automatically created in the hierarchy at the moment of its instantiation. On a system mounting cgroup v2 and the *systemd* daemon, the root cgroup folder for a specific container is always located at `/sys/fs/cgroup/system.slice/docker-<longid>.scope/`. The easiest way to find out a container's long ID is by using the inspect command: `docker inspect <short_id>`. For example, if a container shows up as `ae836c95b4c3` when using `docker ps`, its long ID might look something like:

`ae836c95b4c3c9e9179e0e91015512da89fdec91612f63cebae57df9a5444c79`.

It's possible to apply resource limitations to a container at the moment of its creation by using a set of flags that the `docker run` command offers for this purpose. The list of supported options is reported in the following table.

Option	Description
`-m`, `--memory=""`	Memory limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b`, `k`, `m`, or `g`. Minimum is 4M.
`--memory-swap=""`	Total memory limit (memory + swap, format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b`, `k`, `m`, or `g`.
`--memory-reservation=""`	Memory soft limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b`, `k`, `m`, or `g`.
`--kernel-memory=""`	Kernel memory limit (format: `<number>[<unit>]`). Number is a positive integer. Unit can be one of `b`, `k`, `m`, or `g`. Minimum is 4M.
`-c`, `--cpu-shares=0`	CPU shares (relative weight)
`--cpus=0.000`	Number of CPUs. Number is a fractional number. 0.000 means no limit.
`--cpu-period=0`	Limit the CPU CFS (Completely Fair Scheduler) period
`--cpuset-cpus=""`	CPUs in which to allow execution (0-3, 0,1)
`--cpuset-mems=""`	Memory nodes (MEMs) in which to allow execution (0-3, 0,1). Only effective on NUMA systems.

Option	Description
`--cpu-quota=0`	Limit the CPU CFS (Completely Fair Scheduler) quota
`--cpu-rt-period=0`	Limit the CPU real-time period. In microseconds. Requires parent cgroups be set and cannot be higher than parent. Also check rtprio ulimits.
`--cpu-rt-runtime=0`	Limit the CPU real-time runtime. In microseconds. Requires parent cgroups be set and cannot be higher than parent. Also check rtprio ulimits.
`--blkio-weight=0`	Block IO weight (relative weight) accepts a weight value between 10 and 1000.
`--blkio-weight-device=""`	Block IO weight (relative device weight, format: `DEVICE_NAME:WEIGHT`)
`--device-read-bps=""`	Limit read rate from a device (format: `<device-path>:<number>[<unit>]`). Number is a positive integer. Unit can be one of `kb`, `mb`, or `gb`.
`--device-write-bps=""`	Limit write rate to a device (format: `<device-path>:<number>[<unit>]`). Number is a positive integer. Unit can be one of `kb`, `mb`, or `gb`.
`--device-read-iops=""`	Limit read rate (IO per second) from a device (format: `<device-path>:<number>`). Number is a positive integer.
`--device-write-iops=""`	Limit write rate (IO per second) to a device (format: `<device-path>:<number>`). Number is a positive integer.
`--oom-kill-disable=false`	Whether to disable OOM Killer for the container or not.
`--oom-score-adj=0`	Tune container's OOM preferences (-1000 to 1000)
`--memory-swappiness=""`	Tune a container's memory swappiness behavior. Accepts an integer between 0 and 100.
`--shm-size=""`	Size of `/dev/shm`. The format is `<number><unit>`. `number` must be greater than `0`. Unit is optional and can be `b` (bytes), `k` (kilobytes), `m` (megabytes), or `g` (gigabytes). If you omit the unit, the system uses bytes. If you omit the size entirely, the system uses `64m`.

For example, one could allow processes in a container to be executed exclusively on cpu 1 and cpu 3 by running the following command

```
docker run -it --cpuset-cpus="1,3" ubuntu:14.04 /bin/bash
```

Of course, it's also possible to update the resource limits configuration of a running container. This can be performed by using the `update` command in combination with one or more of the flags in the previous table. For example, to limit a running container's cpu-shares to 512, one could use the following command:

```
docker update --cpu-shares 512 <container_id>
```

It's possible to indicate the specific cgroup that a container should run in using the `--cgroup-parent="<path>"` flag. If the path specified is not absolute, the path is considered to be relative to the cgroups path of the *init* process. If the cgroups doesn't already exist, it will instead be created. Overall, this option allows users to create and manage cgroups on their own. The user could then define custom resources for those cgroups and put containers under a common parent group. Alternatively, instead of setting the root cgroup directory per container, it's also possible to configure the default cgroup parent to use for all containers by modifying the docker daemon configuration through the following command: `dockerd --cgroup-parent="<path>"`. In both cases, on machines using *systemd*, `--cgroup-parent` must necessarily be a slice name. For example, `--cgroup-parent=user1.slice` means the cgroup for the container is created in `/sys/fs/cgroup/user1.slice/docker-<id>.scope`.

When a Docker container is killed, its `docker-<longid>.scope` directory is automatically removed from the cgroup hierarchy, while all upper folders remain unchanged.

References

[1] Rice, L. (2020), Container Security: Fundamental Technology Concepts That Protect Containerized Applications

[2] Turnbull, J. (2014), The Docker Book: Containerization is the new virtualization

[3] Michael Kerrisk. Containers unplugged: Linux namespaces - NDC TechTown 2019; 4 September 2019, Kongsberg, Norway (https://man7.org/conf/ndctechtown2019/Linux-namespaces-NDC-TechTown-2019-Kerrisk.pdf)

[4] IBM Cloud Learn Hub, Containerization (https://www.ibm.com/cloud/learn/containerization)

[5] Matthew Portnoy (2016), Virtualization Essentials, 2nd Edition

[6] Jim Smith, Ravi Nair (2005), Virtual Machines: Versatile Platforms for Systems and Processes

[7] Nigel Poulton (2020), Docker Deep Dive: Zero to Docker in a single book

[8] Giovanni Lupi (2022), A brief survey on control groups v2 (https://github.com/giovannilupi/cgroup-v2-training/blob/main/cgroups_survey.pdf)

[9] Docker Documentation (https://docs.docker.com/)