Virtual machines vs. containers

Johan Luttu

April 2019

1 Introduction

Both virtual machines and containers can be used to virtualize isolated environments where software can be deployed. They're similar in many ways, which can explain that there's a common misconception that containers serve the same purpose as virtual machines but in a new, better way [1][2]. However, this isn't necessarily true. Virtual machines and containers have some major differences which make them more or less suitable for different problems. This essay will discuss how they differ and when to use one or/and the other.

2 Background

In order to get a deeper understanding of the subject, it's required to have knowledge about operating systems. This is due to the fact that concepts which have to do with how operating systems work will turn out to be the key differences between virtual machines and containers. In order for this essay to be self-contained, concepts related to operating systems will be explained. If you're already deeply familiar with topics such as "what is a kernel?" you might want to skip this section.

2.1 Operating system kernel

The kernel is the core part of an operating system. It controls everything in the system. It decides which programs get to be executed on the CPU by scheduling processes, and it's responsible for allocating memory to the processes. It also communicates with I/O devices such as a hard drive or a network adapter, etc. [3].

2.2 Linux cgroups (control groups)

Cgroups is a feature of the Linux kernel that can be used to control groups of processes, hence the name control groups This feature allows you to group a set of processes that run as a related unit, and that group of running processes can be controlled with respect to how much resources of the host they're allowed to consume. Resources in terms of memory [5][6], CPU utilization [7], how much I/O they do (I/O both over network and to the hard drive and other devices) [8]. Accounting is also provided as a part of the feature, which consists of measuring the usage of resources by a particular group. Cgroups can also be used to freeze groups of processes and save snapshots of applications' states (so they can be restored after an eventual failure) [9]. Another thing about cgroups that is important to know in the context of this essay is that cgroups may be nested, which means that a cgroup can be a parent of another cgroup.

2.3 Linux kernel namespaces

A namespace is another kernel feature that can restrict the view of the system. Instead of showing you every aspect of a running system, it shows you a more narrow perception of the system. This way you get the illusion that you're running on a system that has fewer interfaces. For instance, if you're starting up a Linux box and logging in for the first time, you're in the root. You're going to see all running processes, the full view of all the file systems. Every network interface, every bridge, every tunnel interface, everything is going to be visible to you. However, if you enter a namespace, that view can be restricted [10].

2.4 Hypervisors

A hypervisor, also called a virtual machine monitor can be software, firmware or even hardware that creates and runs virtual machines. Where it runs depends on what type of hypervisor it is. There are two different types. One type is "Type-1", bare-metal hypervisors, also called native hypervisors. They run directly on the hardware of the host machine and manage guest operating systems. The other type is called "Type-2" or hosted hypervisors, which don't run on hardware but on software, on top of operating systems. [11]

3 Virtual machines vs. containers

This section is going to be straight forward. First, containers and virtual machines are going to be explained. Then, the actual difference between them is going to be discussed.

3.1 Containers

The word container is to some extent used to cover a multitude of things, which can be confusing to some. At its most basic, the notion and the concept of a container is that it's basically a sandbox for a process where resources and views of the system are restricted. To be more detailed, what all things that could be called containers have in common is that they're depending on Linux cgroups, Linux namespaces, a container image and a related life cycle to function. By using namespaces, a container can get an isolated workspace where access to processes, network interfaces, IPC resources, the filesystem and the kernel is restricted. The use of cgroups allows a container to be allocated hardware resources such as CPU utilization and memory. Containers share the kernel of the host operating system.

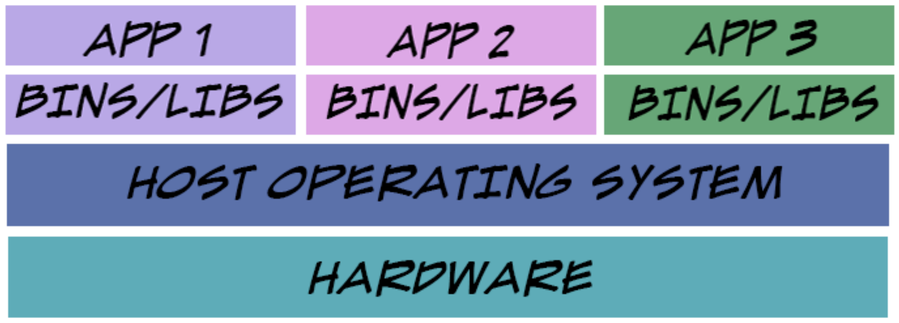

Figure 1: An overview of containers

As security measures, containers make use of other kernel features. All features won't be touched in this essay. The docker engine is for instance using AppArmor and Seccomp which restrict programs' capabilities by for instance filtering system calls issued by programs and performing permission checks [4].

3.1.1 Container image

While cgroups and namespaces make up a container representation at runtime, a container image is basically a binary representation of a container. A docker image can be created by using a dockerfile. Each line of instructions in a dockerfile will create a new layer for the image. [14] There's a notion of a parent-child relationship implemented for container images, which allows images to easily be extended or be built on other container images. This is also beneficial when you want the image uploaded to a version control system, then you can just push or pull the latest layers. The container images also allow smooth portability, services can easily be deployed if an image contains all the required dependencies.

3.2 Virtual machines

Virtual machines are emulated isolated systems. Each virtual machine gets their own kernel, and they believe that they're running directly on the hardware. A hypervisor, of type1 or type2, is used to map the virtual machines' operations to the underlying hardware or operating system. The fact that the hypervisor interferes can lead to poor performance on weaker machines, and it especially starts to shine through if you try to run virtual machines within virtual machines - if you try to nest them.

Virtual machines must host its own operating system in its hypervisor environment, referred to as the guest operating system. This includes everything that comes with the operating system: libraries, binaries and applications, etc. This leads to a lot of overhead as it consumes a lot of system resources. Especially when virtual machines are running on the same host machine as they all have their own guest OS and pre-allocate resources. This makes them heavier than containers in the sense that they pre-allocate memory and that they emulate a whole system instead of just being an isolated process. This is why containers start faster.

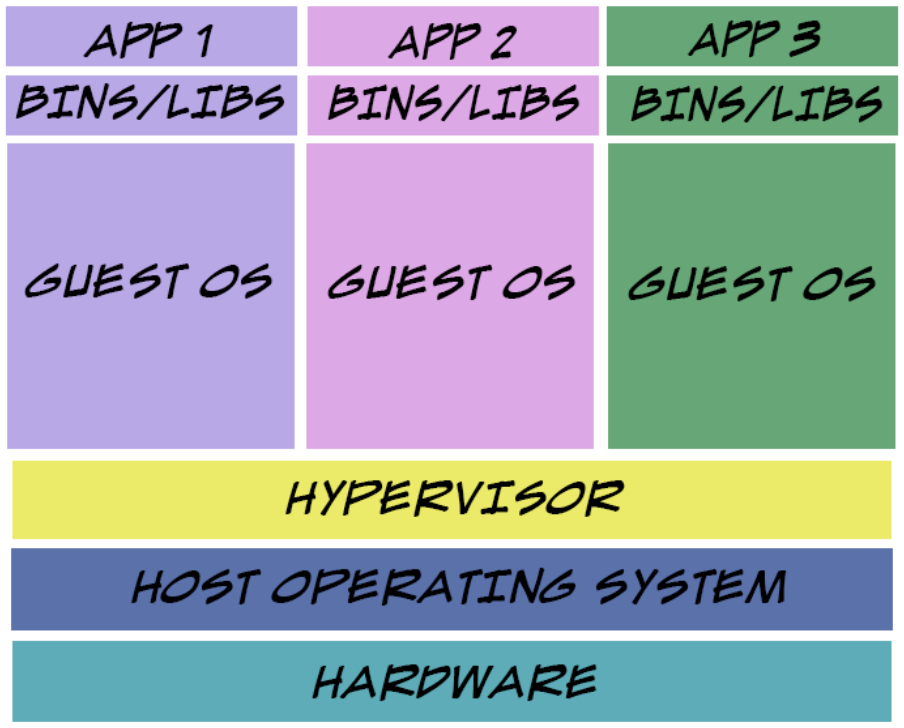

Figure 2: An overview of a system with virtual machines.

3.3 The difference

3.3.1 Performance

3.3.1.1 Runtime performance

If you run an application within a container, and once the process has started, its performance characteristics is exactly the same as it would be on a bare metal machine. This has to do with the things that make a container different from a virtual machine from a runtime perspective, related to the maintenance of the namespaces and the cgroups. That affects the process startup and the process teardown, but it does not affect how that process can interact with the kernel while it's running. For instance, if one makes a syscall from within a container, the performance is not slower because you're running in a container. This is due to the fact that once a process is running, the namespace does not change the execution path between that running process and the equipment it's running on. [10] This is why I've chosen to omit the container engine layer above the host operating system layer in Figure 1. Moreover, the execution path is different in the case of running in a virtual machine, where you got your own kernel and either got a Type 1 or Type 2 hypervisor that is interfering. This is one reason why some people say that containers are faster than virtual machines. There's nothing interfering between the running application and the hardware itself.

3.3.1.2 Resources

One thing that makes containers differ from virtual machines is that they don't pre-allocate resources. For instance, it's possible to specify how much memory a container needs, but it won't pre-allocate it. This can lead to more efficient use of resources where more work is happening on the same host.

3.3.2 Security

A common threat when it comes to containers or virtual machines is escape exploits. How secure are containers and virtual machines from neighboring containers or virtual machines on the same host? When one is talking about defending against hostile workloads that are sharing the same host, when those things are isolated in one virtual machine, that is one risk. However, when those things are separated only by containers, that is a much more complicated risk profile. In the hypervisor world, the number of things that compose the attack surface between two virtual machines on the same host is a small set of virtualization mappings. It's a relatively small attack surface. For instance, it is possible to have two virtual machines on the same host and have processes from one escape, through the hypervisor, to the memory space of the neighboring virtual machine. However, this is relatively straightforward to defend against, since the attack surface is narrow. There are only a few ways this can happen. By comparison, the Linux syscall interface is not as narrow. There are over 300 different syscalls for the Linux kernel version 5.0 [12]. That is a much more difficult attack surface to navigate when it comes to securing containers. If the only thing that's between two neighboring containers on the same host is something that got over 300 different syscalls, you need a fundamentally different strategy for how to defend against escapes.

The barrier between containers running on the same host is not as significant as the barrier between neighboring virtual machines on the same host. However, there are ways to make containers almost as secure as virtual machines. There exist techniques to limit the attack surface between neighboring containers. One example is to set a policy that enforces that the containers are by default not allowed to do anything with the kernel except the things that are explicitly allowed [13].

3.4 When to use what?

When one gets to choose from which of these technologies to use, it shouldn't be an exclusive choice of "I'm only going to run this thing on bare metal and containers" or "I'm only going to run this thing in a virtual machine". You can still put containers into virtual machines. It's a perfectly valid use case. For instance, if you care about that additional security isolation and decide to use a virtual machine, you're still going to get benefits from having the portability of a container image, having the microservices compatibilities, etc. The only time you need to feel like you're compromising is if you got an application that's exceedingly performance sensitive and needs to be exceedingly secure. That's when you need to use those advanced techniques that were mentioned in the security section of this essay.

In many cases, it's better to use a container over a virtual machine, but a concrete use case for when you would want to use a virtual machine over a container could be if you're running a web hotel. Then it would probably be a good idea to give each customer an own virtual machine for security reasons, given the stronger isolation. It also guarantees the customer that the promised resources that have been paid for are available.

A use case for when you would want to use containers over virtual machines could be when you want to deploy multiple instances of a single application or service. By using containers, this can be done more efficiently than with virtual machines since the containers don't consume as many resources, which allows full utilization of the hardware of the machine that you're deploying your apps or services on.

4 Conclusion

Briefly speaking, containers are in runtime isolated processes with restricted resources (they do however not pre-allocate memory resources) that share the kernel of the host operating system. They're portable as you can pack an environment with the required dependencies and binaries for an application in a container image that can be layered. Virtual machines are emulated systems with own kernels and stronger isolation between neighboring virtual environments.

When you execute processes in containers it's like running them on bare metal.

In virtual machines, a hypervisor interferes by mapping the operations to the underlying operating system or hardware. The startup and teardown speed is also much faster for containers.

References

[1] https://stackoverflow.com/questions/16647069/should-i-use-vagrant-or-docker-for-creating-an-isolated-environment/21314566#21314566 - accessed 2019-04-29

[2] https://stackoverflow.com/questions/16047306/how-is-docker-different-from-a-virtual-machine accessed 2019-04-30

[3] "Kernel". Linfo. Bellevue Linux Users Group. Retrieved 15 September 2016.

[4] https://docs.docker.com/engine/security/ - accessed 2019-04-30

[5] Jonathan Corbet (31 July 2007). "Controlling memory use in containers". LWN.

[6] Balbir Singh, Vaidynathan Srinivasan (July 2007). "Containers: Challenges with the memory resource controller and its performance". Ottawa Linux Symposium.

[7] Jonathan Corbet (23 October 2007). "Kernel space: Fair user scheduling for Linux". Network World. Retrieved 22 August 2012.

[8] Kamkamezawa Hiroyu (19 November 2008). Cgroup and Memory Resource Controller. Japan Linux Symposium.

[9] Dave Hansen. Resource Management. Linux Foundation.

[10] http://man7.org/linux/man-pages/man7/namespaces.7.html - accessed 2019-04-30

[11] Meier, Shannon (2008). "IBM Systems Virtualization: Servers, Storage, and Software" pp. 2, 15, 20. Retrieved December 22, 2015.

[12] https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl - accessed 2019-04-30

[13] Linux kernel capabilities. Docker. - accessed 2019-04-30

[14] https://docs.docker.com/engine/reference/commandline/images/. Docker. - accessed 2019-04-30