2.1.1 Comparing containers to virtual machines

Instead of using virtual machines to isolate the environments of individual microservices (or software processes in general), most development and operations teams now prefer to use containers. They allow you to run multiple services on the same host computer, while keeping them isolated from each other. Like VMs, but with much less overhead.

Unlike VMs, which each run a separate operating system with several system processes, a process running in a container runs within the existing host operating system. Because there is only one operating system, no duplicate system processes exist. Although all the application processes run in the same operating system, their environments are isolated, though not as well as when you run them in separate VMs. To the process in the container, this isolation makes it look like no other processes exist on the computer. You'll learn how this is possible in the next few sections, but first let's dive deeper into the differences between containers and virtual machines.

COMPARING THE OVERHEAD OF CONTAINERS AND VIRTUAL MACHINES

Compared to VMs, containers are much lighter, because they don't require a separate resource pool or any additional OS-level processes. While each VM usually runs its own set of system processes, which requires additional computing resources in addition to those consumed by the user application's own process, a container is nothing more than an isolated process running in the existing host OS that consumes only the resources the app consumes. They have virtually no overhead.

Figure 2.1 shows two bare metal computers, one running two virtual machines, and the other running containers instead. The latter has space for additional containers, as it runs only one operating system, while the first runs three – one host and two guest OSes.

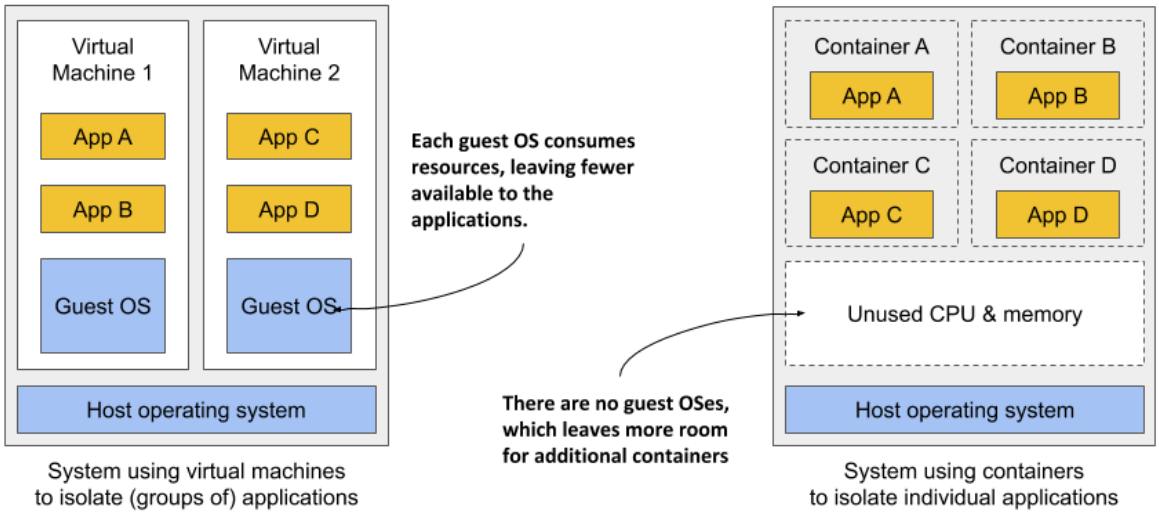

Figure 2.1 Using VMs to isolate groups of applications vs. isolating individual apps with containers

Because of the resource overhead of VMs, you often group multiple applications into each VM. You can't afford to dedicate a whole VM to each app. But containers introduce no overhead, which means you can afford to create a separate container for each application. In fact, you should never run multiple applications in the same container, as this makes managing the processes in the container much more difficult. Moreover, all existing software dealing with containers, including Kubernetes itself, is designed under the premise that there's only one application in a container.

COMPARING THE START-UP TIME OF CONTAINERS AND VIRTUAL MACHINES

In addition to the lower runtime overhead, containers also start the application faster, because only the application process itself needs to be started. No additional system processes need to be started first, as is the case when booting up a new virtual machine.

COMPARING THE ISOLATION OF CONTAINERS AND VIRTUAL MACHINES

You'll agree that containers are clearly better when it comes to the use of resources, but there's also a disadvantage. When you run applications in virtual machines, each VM runs its own operating system and kernel. Underneath those VMs is the hypervisor (and possibly an additional operating system), which splits the physical hardware resources into smaller sets of virtual resources that the operating system in each VM can use. As figure 2.2 shows, applications running in these VMs make system calls (*sys-calls*) to the guest OS kernel in the VM, and the machine instructions that the kernel then executes on the virtual CPUs are then forwarded to the host's physical CPU via the hypervisor.

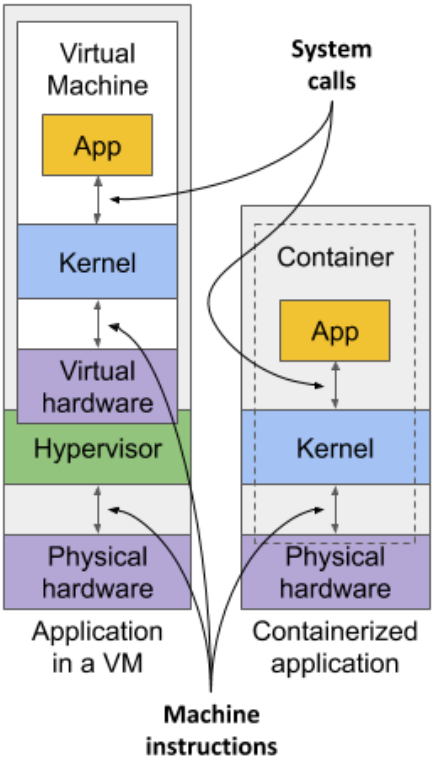

Figure 2.2 How apps use the hardware when running in a VM vs. in a container

> **NOTE** Two types of hypervisors exist. Type 1 hypervisors don't require running a host OS, while type 2 hypervisors do.

Containers, on the other hand, all make system calls on the single kernel running in the host OS. This single kernel is the only one that executes instructions on the host's CPU. The CPU doesn't need to handle any kind of virtualization the way it does with VMs.

Examine figure 2.3 to see the difference between running three applications on bare metal, running them in two separate virtual machines, or running them in three containers.

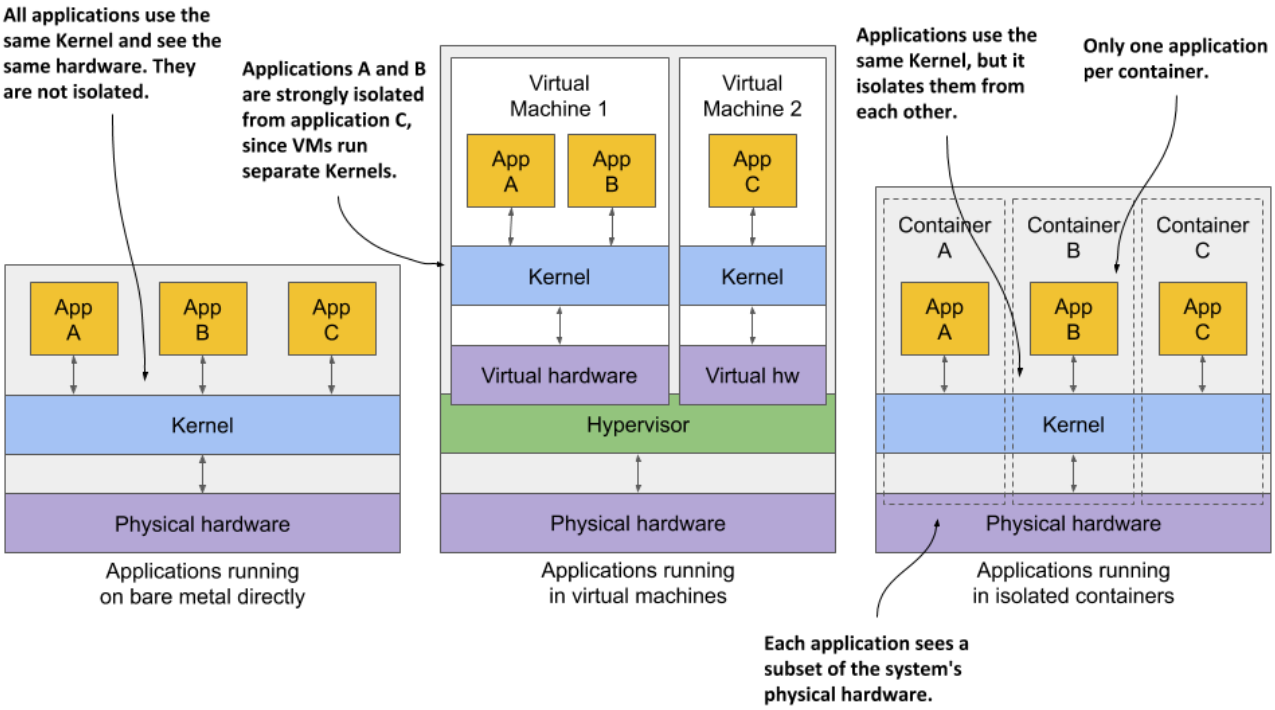

Figure 2.3 The difference between running applications on bare metal, in virtual machines, and in containers

In the first case, all three applications use the same kernel and aren't isolated at all. In the second case, applications A and B run in the same VM and thus share the kernel, while application C is completely isolated from the other two, since it uses its own kernel. It only shares the hardware with the first two.

The third case shows the same three applications running in containers. Although they all use the same kernel, they are isolated from each other and completely unaware of the others' existence. The isolation is provided by the kernel itself. Each application sees only a part of the physical hardware and sees itself as the only process running in the OS, although they all run in the same OS.

UNDERSTANDING THE SECURITY-IMPLICATIONS OF CONTAINER ISOLATION

The main advantage of using virtual machines over containers is the complete isolation they provide, since each VM has its own Linux kernel, while containers all use the same kernel. This can clearly pose a security risk. If there's a bug in the kernel, an application in one container might use it to read the memory of applications in other containers. If the apps run in different VMs and therefore share only the hardware, the probability of such attacks is much lower. Of course, complete isolation is only achieved by running applications on separate physical machines.

Additionally, containers share memory space, whereas each VM uses its own chunk of memory. Therefore, if you don't limit the amount of memory that a container can use, this

could cause other containers to run out of memory or cause their data to be swapped out to disk.

NOTE This can't happen in Kubernetes, because it requires that swap is disabled on all the nodes.

UNDERSTANDING WHAT ENABLES CONTAINERS AND WHAT ENABLES VIRTUAL MACHINES

While virtual machines are enabled through virtualization support in the CPU and by virtualization software on the host, containers are enabled by the Linux kernel itself. You'll learn about container technologies later when you can try them out for yourself. You'll need to have Docker installed for that, so let's learn how it fits into the container story.

2.1.2 Introducing the Docker container platform

While container technologies have existed for a long time, they only became widely known with the rise of Docker. Docker was the first container system that made them easily portable across different computers. It simplified the process of packaging up the application and all its libraries and other dependencies - even the entire OS file system - into a simple, portable package that can be used to deploy the application on any computer running Docker.

INTRODUCING CONTAINERS, IMAGES AND REGISTRIES

Docker is a platform for packaging, distributing and running applications. As mentioned earlier, it allows you to package your application along with its entire environment. This can be just a few dynamically linked libraries required by the app, or all the files that are usually shipped with an operating system. Docker allows you to distribute this package via a public repository to any other Docker-enabled computer.

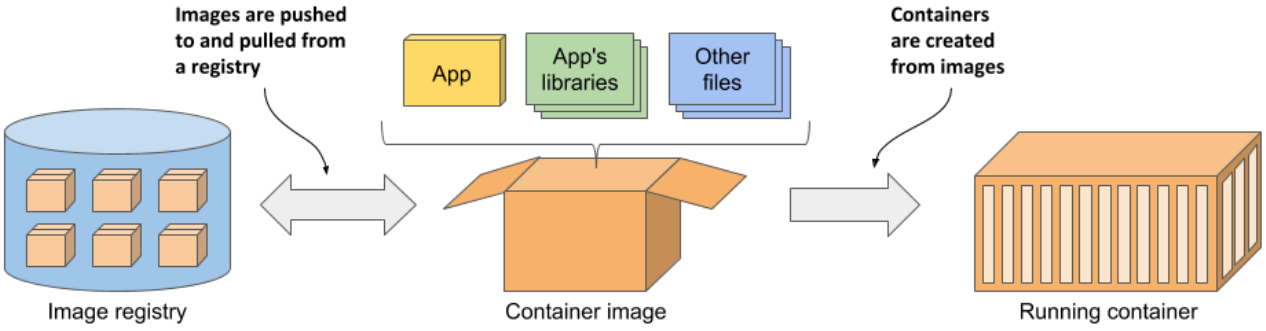

Figure 2.4 The three main Docker concepts are images, registries and containers

Figure 2.4 shows three main Docker concepts that appear in the process I've just described. Here's what each of them is:

- *Images*—A container image is something you package your application and its environment into. Like a zip file or a tarball. It contains the whole filesystem that the application will use and additional metadata, such as the path to the executable file to