«Evolution of the Container Technology»

Patrick P. Bucher

Summary

Containers are a mechanism to isolate and deploy applications. Unlike virtual machines, which run multiple processes within the same environment, containers usually consist of only a single application that cannot interact with the surrounding operating system and its processes.

Whereas virtual machines are full-fledged operating systems, containers only create the illusion of providing an operating system. This is achieved by the means of isolation:

- Process Namespaces: The processes within a container are not aware of other processes running on the surrounding operating system.
- Network Namespaces: Containers have their own virtual network adapter, network configuration and routing tables.
- Mount Namespaces: A container has its own file system and cannot see or modify the files of the surrounding operating system.
- User Namespaces: Containers have their own users with user IDs that do not conflict with the users of the surrounding operating system.

This level of isolation makes containerized applications more portable and more secure. However, in order to be of any practical use, containers must also be able to interact with their surrounding operating system and other container applications:

- The standard output of a container process can be forwarded to the surrounding shell, which is useful for logging.
- Containers can expose certain ports, and the container application can handle connections on those (e.g. web servers).
- Folders of the surrounding operating system's file system can be mounted into the container, which makes it possible to create persistent container applications (e.g. data bases).

Concepts like Containers and namespaces are no recent developments. The Unix system call `chroot`, which provides an isolated file system, exists since the late seventies. Later on, BSD introduced the `jail` command, which also provided an isolated process namespace. Solaris Zones are available since 2004, and offer capabilities similar to those of Docker. However, it was the container engine Docker, with its initial release in late 2013, that pulled off the container revolution.

Docker not only enables developers to create reproducible runtime environments, it was also the starting points for projects like Kubernetes and OpenShift, which combine Docker's container engine with an orchestration framework for containerized applications.

Source

Miell, I. and Hobson Sayers A.: Docker in Practice (2nd Edition). Manning 2019

Questions

0. Is it a good idea to run sub-processes from the container's main process?
   - Technically, sub-processes can be executed.
   - If the use-case suits the application and solves a problem, it is perfectly valid.
   - However, a container application should not become a monolith.
1. Are there any applications that are a bad fit for containers?
   - Huge legacy monolithic applications.
   - Big data bases, like Oracle, for example, are sometimes problematic.
2. How can artifacts from a non-public source repository be brought into a container without revealing sensible information in clear text?
   - The easiest way is to hold the `Dockerfile` and the source artifacts in the same repository.
3. Are there any downsides to containers?
   - Working with containers is not always easy. Especially production-ready environments like Kubernetes need a lot of prior training.
4. Are *Docker Compose* and *Docker Swarm* commonly used in production?
   - Kubernetes is more common than Docker Swarm. It offers more flexibility, is an entire platform (with features like build pipelines) and not just a container orchestration.
5. Are there any other useful container engines besides Docker?
   - Solaris Zones (since 2004) come pretty close to what Docker offers nowadays.
   - `rkt` is a container engine that operates at around the same level as Docker.