Containers

Florent Gluck - Florent.Gluck@hesge.ch

April 8, 2022

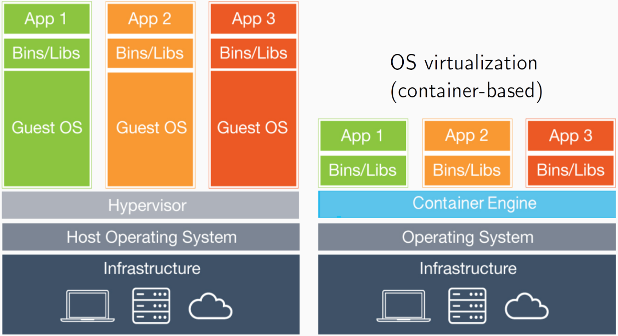
Operating system virtualization

- Previously, we studied platform virtualization
- Here, we're presenting **operating system virtualization**
  - kernel can manage multiple **isolated user-space instances** called **containers**
  - OS virtualization is often called **containerization**
- Processes inside a container **only** see the container's contents and devices assigned to it
  $\rightarrow$ isolation (also called *sandboxing*)
- Kernel provides **resource-management** features to limit impact of a container's activities on other containers
- Examples: BSD jails, Solaris Zones, LXC, Docker, Linux OpenVZ, etc.

OS virtualization vs platform virtualization

Platform virtualization (hypervisor-based)

OS virtualization (container-based)



- Containers share the underlying OS, have different libs, utilities, root filesystem, view of process tree, networking, etc.
- VMs have different guest OS
- Containers have less overhead than VMs at the cost of less isolation

What is a container?

- Multiple definitions depending on the type of containers framework
- **A container is a set of processes that are isolated from the host system and other containers**
  - with most frameworks, the files necessary to run the containers are provided as an image
- Multiple containers can run within the same host machine
- Containers sometimes called “lightweight VM” $\rightarrow$ however, they are NOT a VM!

History of containers

- Container technology has existed for a long time in various forms
- Significant popularity gain since native support in Linux kernel*

Year	**Technology**	**Operating System**
1982	chroot[1]	Unix-like OSes
2000	Jails	FreeBSD
2000	Virtuozzo containers	Linux, Windows
2001	Linux VServer	Linux, Windows
2004	Solaris Zones	Sun Solaris, Open Solaris
2005	OpenVZ	Linux (open source version of Virtuozzo)
2008*	LXC	Linux
2013	Docker	Linux, FreeBSD, Windows
2015	Singularity	Linux
2018	Podman	Linux

[1] changes the root directory for the current running process and its children

Containers look like virtual machines

From a distance: a container looks like a VM:

- I can SSH into my container
- I can have root access in it
- I can install packages in it
- I have my own network interface
- I can tweak routing table, iptables rules
- I can mount filesystems
- etc.

Containers: how?

Containers build upon key Linux kernel features:

- Capabilities (security)
- Namespaces (isolation)
- Control groups (limits)
- Seccomp (security)

Problem with traditional UNIX privilege model

- Traditional UNIX privilege model divides users into two groups:
  - normal unprivileged users
  - superuser (root, effective UID 0)
- Problem: granularity, root/non-root, is too coarse
  - no limit on possible attacks if root program is compromised!
- Solution?
  - capabilities

Capabilities

- **Capabilities divide superuser's privileges into small pieces**
  - 38 capabilities as of Linux 5.4
  - root user = process with full set of capabilities
- Typical goal: replace SUID programs with programs that have capabilities
- Processes and files can each have capabilities
  - process capabilities: defines what privileged operations a process can do
  - file capabilities: what capabilities a process gets when executing the file
    - stored in extended attributes (`security.capability`)

Namespaces

- **Namespaces are used to provide many types of isolation**
- Namespaces affect processes
- Linux supports multiple namespace types:
  - UTS namespace $\rightarrow$ isolates hostnames
  - mount namespace $\rightarrow$ isolates filesystems
  - IPC namespace $\rightarrow$ isolates inter-process communications
  - Network namespace $\rightarrow$ isolates networking resources
  - PID namespace $\rightarrow$ isolates process ID
  - User namespace $\rightarrow$ isolates user and group IDs
  - Cgroup namespace

Control groups (cgroups)

- **Cgroups allow to allocate resources among groups of processes**
  - CPU time, memory, network bandwidth, I/O bandwidth
  - provide fine-grained control over allocating, prioritizing, denying, managing, and monitoring system resources
  - organized hierarchically (like processes) and child cgroups inherit some of their parents' attributes
- Cgroups can be:
  - monitored
  - denied access to resources
  - reconfigured dynamically (i.e. at run-time)
- Hardware resources can be divided up among processes and users to increase overall efficiency

Seccomp

- **Seccomp is used to restrict system calls that a process makes**
- Linux kernel provides ~400 system calls!
- Each syscall is a vector for attack against the kernel
- Most programs use only a small subset of available syscalls
  - remaining syscalls should never occur
  - if they do → potential attack!
- Seccomp allows to reduce the attack surface of the kernel
  - a key component for building application sandboxes

Containers: why?

- Lightweight, fast, disposable... virtual environments
  - boot in milliseconds
  - just a few MB of intrinsic disk/memory usage
  - bare metal performance is possible
- Can be used as “light” virtual machines, but with less isolation
- Can be used to build, ship, deploy, and run applications

Benefits of containerization

- **Isolation (security)**
  - Provide a complete isolated OS environment
  - Allow packaging and isolation of applications with their entire runtime environment
- **Portability**
  - Container packaged with all its dependencies
- **Productivity**
  - Performance: lightweight environment
  - Consolidation: maximize resource utilization
  - Continuous integration: development, test, deployment

Containers use cases

- Application packaging
- Datacenter use
  - System virtualization → lightweight "VM"
  - Limit applications resources' usage: memory (e.g. DB), CPU (e.g. numerical simulations)
- Hosting business
  - Give a user root access without full (root) access to the "real" system.
- Compartimentalization of services
  - Application/service isolation → security
  - Modularity → scalability and flexibility

Containers philosophy: microservices

- Every component should be isolated to the finest details and containerized at that level
- Containers can be grouped together to provide a complete application
- Example: Wordpress deployment:
  - 1 apache or nginx container
  - 1 mariadb container
  - 1 php-fpm container
- Benefits of microservices: **modularity** and **scalability**!
  - Ability to scale on demand: create more php-fpm containers when needed (need to change the webserver config to tell it to use load-balancing)

Containers vs virtual machines

- Containers = lightweight compared to traditional VMs $\rightarrow$ more containers can be run per host than VMs
- Unlike containers, VMs require emulation layers (software or hardware) $\rightarrow$ consume more resources and add overhead
- Containers share resources with the underlying host machine, with user space and process isolations
- Starting a container is much faster[2] than starting a VM

[2] when running an "equivalent" system

Which is better: VMs or containers?

- Containers and VMs serve **different needs**
- Containers solve deployment issues and permit elastic scaling more easily than VMs
- Containers are more lightweight and easier to deploy
- VMs are fully isolated from their host $\rightarrow$ better security
- VMs can provide a full desktop environment
- VMs can run different OSes than the host and even emulate different architectures

Limitations of containers

Containers use same kernel as host → imposes strong **limitations**:

- **Limited** to running applications compiled for the host's **kernel architecture**
  - Limitation from an hardware (CPU) point of view: can't run an `armhf` container on top of an `amd64` system
  - Can't run a Windows container on a Linux system
  - Limited to the host's kernel (and its features)
- **Reliability**: higher impact of a crash, especially in kernel area

Container frameworks

- LXC provides a "lightweight VM" environment
  - provides standard OS shell interface
- LXD provides image management on top of LXC
- Docker containers are optimized to run a single application
  - configuration file specifies the base root filesystem, with dependencies needed to run a specific application
  - runs application in a containerized environment
  - easy way to package an application and all its dependencies
    - purpose $\rightarrow$ run anywhere
- Podman: runs OCI[3] containers, daemonless, rootless alternative to Docker

[3] Open Container Initiative

Container orchestration frameworks

- Docker compose: framework to manage multiple containers on a single host
- Docker Swarm, Kubernetes: frameworks to manage multiple containers on multiple hosts
- Kubernetes: popular container orchestration framework
  - runs over multiple physical machines
  - auto-scaling when load increases, restart services when they crash, etc.
  - not tied to Docker anymore, switched to containerd from the OCI

Resources

- Practical LXC and LXD “Linux Containers for Virtualization and Orchestration”, Senthil Kumaran S., Apress 2017
- “Is it safe to run applications in Linux Containers?” Jérôme Petazzoni, 2014
- Namespaces in operation
  https://lwn.net/Articles/531114/
- Control groups Linux kernel documentation:
  https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html