Container-Based Virtualization

Luca Abeni
luca.abeni@santannapisa.it

March 16, 2020

OS-Level Virtualization

- The OS kernel (or the whole OS) is virtualized
  - Focus on kernel virtualization $\rightarrow$ container-based virtualization
  - Guests can provide the user-space part of the OS (system libraries + binaries, boot scripts, ...) or just an application...
  - ...But continue to use the host OS kernel!
- One single OS kernel (the host kernel) in the system
  - The kernel virtualizes all (or part) of its services
- In this case, a Virtual Machine is based on an efficient, isolated duplicate of an OS kernel!
  - How to provide isolation?

What is a Container, Anyway?

- We consider container-based virtualization, but...
- ...What is a container?
- Guess? Once again, multiple possible definitions...
- Common properties of a container:
  - It contains a group of processes...
    - Organized as a tree, with a root process
  - ...All running on the same host...
  - And provides isolation between this group of processes and the rest of the host!
- Isolation (whatever it means) is the key point, here!
- Again, how to provide this isolation?

Historical Filesystem Isolation: `chroot`

- `chroot()` system call: changes the root directory (`/`) of a process
  - Yes, there are per-process root directories!
- Absolute pathnames start from the root directory and by definition the parent of the root directory does not exist (and `/..` == `/`)
- So, in theory after `chroot(`*path*`)` it is not possible to create pathnames referring files outside of *path*
  - Form of filesystem isolation?
- In the past, used by daemons to limit filesystem access

`chroot` Isolation: Not So Strong...

- The `chroot()` system call just changes the root directory
  - It does not prevent accessing the rest of the filesystem; it just prevents creating pathnames pointing to it...
  - Moreover, it does not prevent mounting the filesystem again...
  - ...It does not affect network connections or devices...
  - ...And it does not isolate processes!
- Very weak form of isolation: easy to break it!
  - Can you show some kind of lack of isolation?
  - Can you escape a chroot?

Real Isolation: Namespaces

- Namespace abstraction: introduced to fix the chroot issues
  - Allow to create isolation for specific functionalities/resources by controlling what a group of processes can see...
- Namespaces allow different groups of processes to have different views of the system
- Main namespaces: mnt, pid, net, ipc, uts, user, ...
  - mnt namespace: filesystems mounted inside the namespace are not visible outside
  - pid namespace: pids are mapped to different values inside the namespaces

Namespaces — Again

- net namespace: network interfaces (and routing tables, etc...) inside the namespace are not visible outside (and vice-versa)
- ipc namespace: isolation on system V IPCs
- uts namespace: allows to have different hostnames inside and outside the namespace
- user namespace: provide virtualization of user IDs (a user who is not root outside the namespace can be root inside, etc...)
- In general, namespaces have to be implemented for every resource that affects isolation
- A first level of isolation is given by namespaces
  - This is for resources visibility; what about resource consumption?

Filesystem Isolation, Revisited

- Why there is no “filesystem namespace”?
  - Should we use `chroot`, again?
- The mount namespace can provide a solution!
  - If the container rootfs is on a different device, it is possible to unmount the rest of the filesystem!
- Of course, we need to play some games to move the container rootfs to “`/`”
  - `pivot_root()`
  - `mount()` with `MS_MOVE`
- Possible to use `tmpfs` or a loop device

Control Groups

- Ok, so we have “visibility isolation” with namespaces...
- Now, let’s assume a bad task inside the “VM” starts forking processes as crazy
  - This will starve the host tasks (or, at least, it will interfere with their execution)!
  - So, we do not have full isolation yet...
- Solution: control groups
  - Allow to control the resource usage of a group of processes
- Control groups for memory, CPUs (cpusets), scheduling, block devices, other devices, PIDs, ...

User-Space Tools

- Building and running a container can be difficult...
  - But users do not have to do it “by hand”!!!
- User-space tools for building containers and deploying OSs/applications in them
  - Simplest tool: `lxc` (`http://linuxcontainers.org`)
  - Server-based version of `lxc`: `lxd`
  - Docker: more advanced features
  - Kubernetes
  - ...
- Recent proliferation of tools, all with different interfaces/features

lxc / lxd

- `lxc`: set of tools and libraries that allow to easily use containers, namespaces and friends
  - Focus on installing and running Linux distributions in containers
- Need root privileges, at least partly
- `lxd`: daemon running with root privileges and using the `lxc` library
  - Clients can connect to it through a socket to request operations on containers
  - More secure, because user tools do not need to be privileged (the only privileged component is the daemon)

More Advanced Tools

- Docker, Kubernetes and similar allow to also containerize single applications
  - Container with application binary, libraries, needed files, etc...
  - Useful for distributing consistent execution environments
- More advanced tools respect to `lxc`/`lxd`
- Also provide “container images” distributed with custom image formats
- Lot of different solutions with different features, interfaces, etc...
  - Let’s try to organize them

Modular Design

- Modern advanced tools such as Kubernetes or similar have a modular design
  - The high-level tool can rely on different components, with well-defined interfaces
- The component responsible for managing the containers execution is the *container runtime*
  - Lot of different tools (even with different features) with this name
- Example: Kubernetes invokes a runtime manager implementing the CRI (Container Runtime Interface)...
  - ...Which invokes yet another container runtime!

Container Runtimes

- Container runtime: software component used to create, run, and control/manage containers
  - Two different kinds: low-level container runtimes, and high-level ones
  - Low-level runtimes just creates, run and control the execution of containers
  - Based on kernel virtualization $\rightarrow$ must be provided with an image format
- High-level runtimes use a low-level container runtime implementing features over it
  - For example, image management
  - Allow to containerize single applications

Container Runtimes — Examples

- `runc`: *standard* low-level container runtime (see OCI standard)
- `crun`: C re-implementation of `runc`
- `lxc`: simple low-level container runtime, lxc commands are more or less reference implementations
- `cri-o`: higher level container runtime, uses `runc` as a low level, and interfaces with Kunernetes
- `podman`: higher level container runtime, can use `runc` or other standard container runtimes; same functionalities as Docker
- `containerd`: higher level container runtime, implemented as a daemon, used by Docker

Standardizing the Container Tools

- Open Container Initiative (OCI): `https://www.opencontainers.org/`
  - Tries to define standards for the user-space tools
  - Currently, two standards: runtime specification and image specification
- Runtime specification: standardizes the configuration, execution environment, and lifecycle of a container
  - A “filesystem bundle” described according to this specification can be started in a container by any compliant runtime
- Image specification: standardizes how the content of a container is represented in binary form

OCI's Goals

- Define containers in a "technology neutral" way
- Container: encapsulates a software component and all its dependencies
  - Using a format that is self-describing and portable
  - Any compliant "runtime" must be able to run it without extra dependencies
- This must work regardless of the implementation details
  - Underlying machine, containerization technology, contents of the container, ...

OCI Runtime Specification

- Standardizes important aspects of containers
  - Configuration: specified through a standardized `config.json`, describing all the details of the container
  - Execution environment: standardized so that applications running in containers see a consistent environment between runtimes
  - Standard operations possible during the containers' lifecycles
- If a "runtime" is compliant with these specifications, the implementation details do not matter

More than Containers

- Looking at the OCI definitions, there is not mention to OS-level virtualization anymore...
  - The terms “container” and “containerized application” are evolving...
- “container” is just a synonim for “lightweight virtual machine”, independently from the used technology
  - Kata containers: use kvm-based VMs (qemu/nemu) instead of namespaces and cgrouops
  - Compliant with the OCI runtime specification
- Thanks to OCI, it is possible to *almost* transparently replace the runtime/containerization mechanism without changing userspace tools!