
CERN

Linux containers and Docker

Elvin Sindrilaru
IT Data and Storage Services Group - CERN

GridKa School 2015

Outline

- Understanding Linux containers
  - Linux namespaces
  - Linux cgroups
- Docker containers
- Containers orchestration
- Security considerations
- Benefits

What is a container?

"Cmglee Container City 2" by Cmglee - Own work. Licensed under CC BY-SA 3.0 via Wikimedia Commons

Linux containers

- Based on two technologies
  - Linux namespaces
  - Linux control groups (cgroups)

Linux namespaces (1)

- The purpose of a namespace is to **wrap** a particular **global system resource** in an **abstraction** that makes it appear to the process within the namespace that they have their **own isolated instance of the global resource**.

Linux namespaces (2)

- Currently there are 6 namespaces implemented in the Linux Kernel:
  - **Mount namespaces** – isolate the set of file-system mount points seen by a group of processes (Linux 2.6.19)
  - **UTS namespaces** – isolate two system identifiers – *nodename* and *domainname*. (UNIX Time-sharing System)
  - **IPC namespaces** – isolate inter-process communication resources e.g. POSIX message queues

Linux namespaces (3)

- **Network namespaces** – provides isolation of system resources associated with networking. Each network namespace has its own network devices, IP addresses, port numbers etc.

- **PID namespaces** – isolate process ID number space. Processes in different PID namespaces can have the same PID number.

- **User namespace** – isolates the process user and group ID number spaces. A process's UID and GID can be different inside and outside a user namespace i.e a process can have full root privileges inside a user namespace, but is unprivileged for operations outside the namespace. (Linux 3.8)

Linux cgroups (1)

- **Cgroups** allow allocating **resources** to **user-defined groups of processes** running on the system
- **Cgroup subsystems** (resources controllers) = kernel modules aware of cgroups which allocate varying levels of system resources to cgroups
- Everything is exposed through a **virtual filesystem**:
- /cgroups, /sys/fs/cgroup … - mountpoint may vary

- Currently up to 10 subsystems:
  - **blkio** – set limits on input/output access to/from block devices such as physical drives
  - **cpuset** – assign individual CPUs and memory nodes to tasks in a cgroup
  - **memory** – set limits on memory used by tasks in a cgroup
  - etc …

Linux cgroups (2)

- ***libcgroup*** package provides command line utilities for manipulating cgroups.

- **lssubsys** – list available subsystems
- **lscgroup** – list defined cgroups
- **cgget** – get parameters of cgroup
- **cset** – set parameters of cgroup
- **cgexec** – start a process in a particular cgroup
- **cgclassify** – move running task to one or more cgroups

Linux containers a.k.a LXC

- **Containers**
  - tool for **lightweight virtualization**
  - provides a group of processes the **illusion** that they are the only processes on the system

- **Advantages** in comparison to traditional VM:
  - Fast to deploy - seconds
  - Small memory footprint - MBs
  - Complete isolation without a hypervisor

Namespaces + Cgroups => Linux containers

Linux containers – CLI

- **lxc** package contains tools to manipulate containers

- **lxc-create**
  - Setup a container (rootfs and configuration)
- **lxc-start**
  - Boot a container
- **lxc-console**
  - Attach a console if started in background
- **lxc-attach**
  - Start a process inside a container
- **lxc-stop**
  - Shutdown a container
- **lxc-destroy**
  - Destroy the container created with lxc-create

Docker containers

- "A **container** is a basic tool, consisting of any device creating a partially or fully enclosed space that can be used to contain, store and transport objects or materials."
  http://en.wikipedia.org/wiki/Container

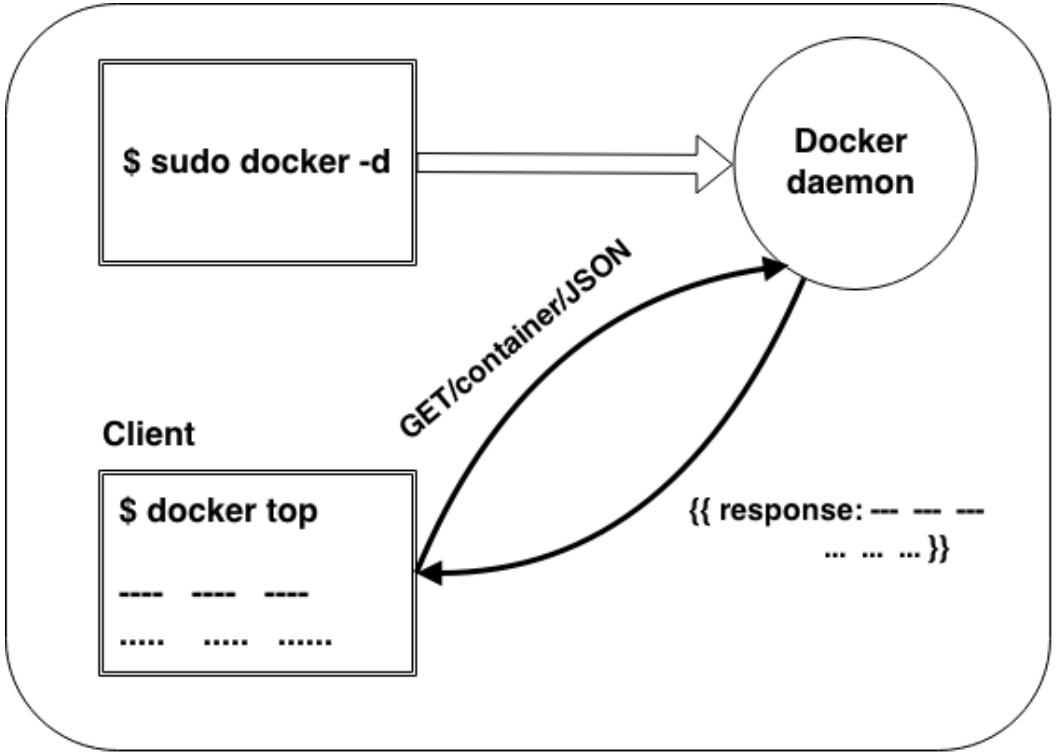
- "Open-source project to easily create **light-weight, portable, self-sufficient containers** from any application"
  https://www.docker.com/whatisdocker/

- **Docker motto**: "Build, Ship and Run Any App, Anywhere"

Docker interaction

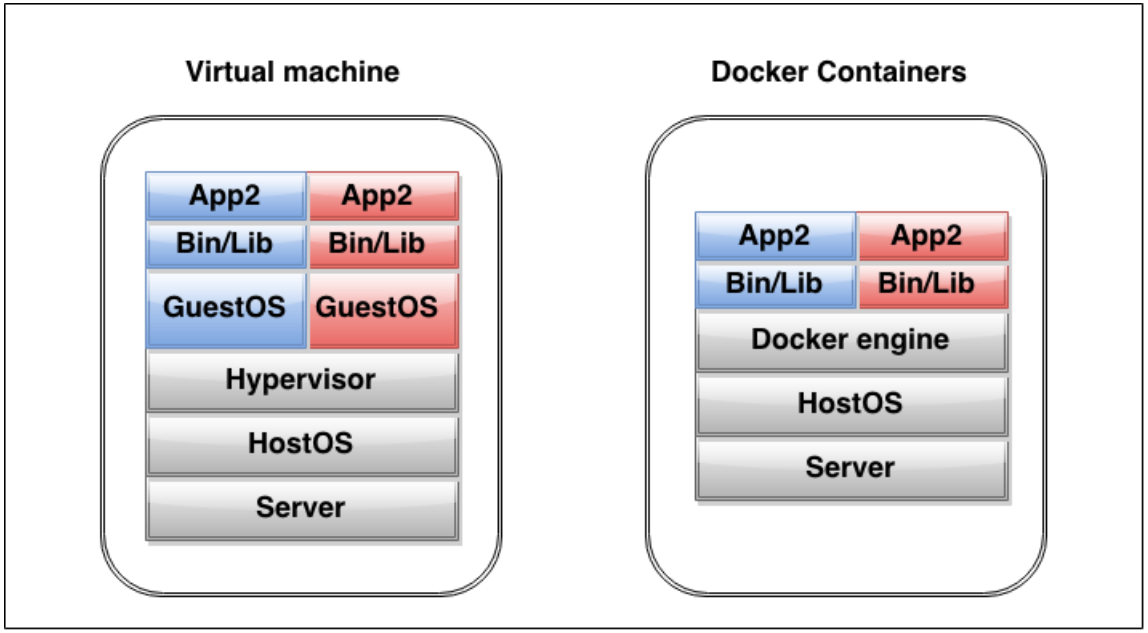
- Client-server model
  - **Docker daemon**
    - Process that manages the containers
    - Creates files systems, assigns IP addresses, routes packages, manages processes
      => needs **root privileges**
    - RESTful API
  - **Docker client**
    - Same binary as the daemon
    - Makes GET and POST request to the daemon

Docker client-server interaction



- The client can run on the same host or on a different one from the daemon

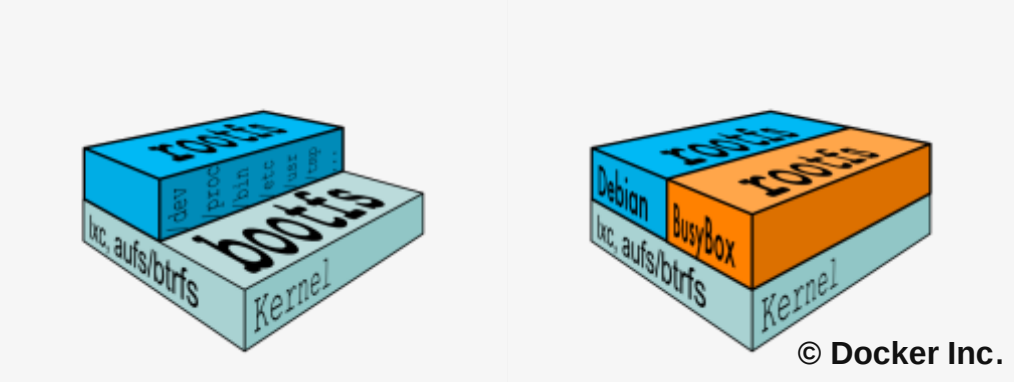
VMs vs. Docker containers



- VMs are **fully virtualized**
- Containers are optimized for **single applications**, but can also run a full system

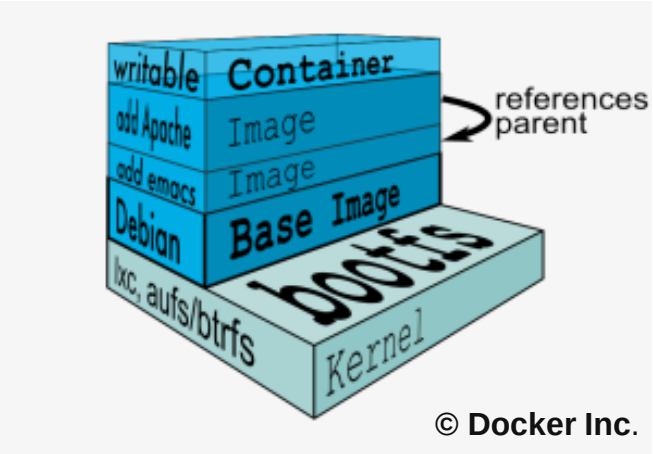
Docker filesystem

- Typical Linux system needs two filesystems:
  - **Boot** file system (bootfs)
  - **Root** file system (rootfs) - /dev, /etc, /bin, /lib …



- Docker can use Another Unionfs (**AUFS**) which is copy-on-write
- AUFS
  - Helps sharing common portions of the fs among containers
  - Layers are **read-only** and the merger of these layers is visible to the processes
  - Any changes go into the rd/wr layer

Docker Images



- **Image**
  - Never changes
  - Stack of read-only fs layers
  - Changes go in the topmost writable layer created when the container starts
  - Changes are discarded by default when the container is destroyed
- Where to get Docker Images from?
  - https://registry.hub.docker.com/
  - Similar to what GitHub is for Git - think “git repository for images”
  - Use your own private registry e.g. pull the docker registry image and run it in a container

Docker Containers

- **Container**
  - Read-write layer
  - Information about Parent Image (RO layers)
  - Unique id + network configuration + resource limits
- Containers have state: **running / exited**
- **Exited container**
  - preserves **file system state**
  - does **NOT** preserve **memory state**

- Containers can be promoted to an Image by using "*docker commit*"

  ➔Takes a **snapshot** of the whole filesystem **(RW+RO**)

Docker paradigm shift

- **Motto:** “write-once-run-anywhere”

- **Developers:**
  - Concentrate on building applications
  - Run them inside containers
  - Avoid the all too common: “But it works fine on my machine …” ☺

- **Sysadmins/operations/DevOps:**
  - Keep containers running in production
  - No more “dependency hell” … almost … at least not traditional ones ☺

⇒ Clean **separation of roles**

⇒ Single underlying tool which (hopefully) **simplifies:**

⇒ code management

⇒ deployment process

Docker workflow automation

- **Dockerfile**
  - Repeatable method to build Docker images – makefile equivalent
  - **DSL(Domain Specific Language)** representing instructions on setting up an image
  - Used together with the context by the *"docker build"* command to create a **new image**

```
# Use the fedora base image
FROM fedora:20
MAINTAINER Elvin Sindrilaru, esindril@cern.ch, CERN 2015

# Add a file from the host to the container
ADD testfile.dat /tmp/testfile

# Install some packages
RUN yum -y --nogpg install screen emacs

# Command executed when container started
CMD /bin/bash
```

Containers orchestration

- **Orchestration** describes the automated arrangement, coordination and management of complex systems, middleware and services.

- **Library dependencies** **“**sort of” become **container dependencies**

Container data management

- **Docker volume**
  - Directory **separated** from the container's root filesystem
  - **Managed by the docker daemon** and can be shared among containers
  - Changes to the volume are not captured by the image
  - Used to mount a directory of the host system inside the container
- **Data-only containers**
  - Expose a volume to other data-accessing containers
  - Prevents volumes from being destroyed if containers stop or crash

Port binding and linking containers

- Bind container ports to host ports using the *"-p"* flag
- Not all containers need to bind internal ports to host ports
  - E.g. only front-end applications need to connect to backend services

- **Linking within the same host**
  - Profit from the unified view that the docker daemon has over all running containers
  - Use the *--link* option: *docker run --link CONTAINER_ID:ALIAS …*
  - Effectively alters the */etc/hosts* file

- **Cross host linking**
  - Requires a distributed, consistent discovery service
  - Needs a **distributed key-value store** to keep info about running containers e.g. etcd
  - Application must be aware of the **discovery service**

Orchestration tools/frameworks

- Docker Machine, Compose and Swarm

- Kubernetes
- https://github.com/GoogleCloudPlatform/kubernetes

- Shipyard
- https://github.com/shipyard/shipyard

- LXC/LXD/CGManage
- https://linuxcontainers.org/

Security considerations

- Docker containers are started with a **reduced capability set** which restricts:
  - Mount/unmount devices
  - Managing raw sockets
  - Some fs operations
- **Fine-grained control of capabilities** using the *docker --cap-add --cap-drop* options
- **End-goal** is to run even the Docker daemon as a non-root user and delegate operations to dedicated subprocesses
- Keep the **host Kernel updated** with the latest security patches

Docker – Benefits

- **Portable deployment across machines** – overcomes machine specific configuration issues
- **Application-centric** – optimized for deployment of applications and not machines
- **Automatic build** – can use any configuration management system (puppet, chef, ansible etc.) to automate the build of containers
- **Versioning** - git like capabilities to track container versions
- **Component reuse** – any container can become a base image
- **Sharing** – use the public registry to distribute container images, just like a git repository

www.cern.ch