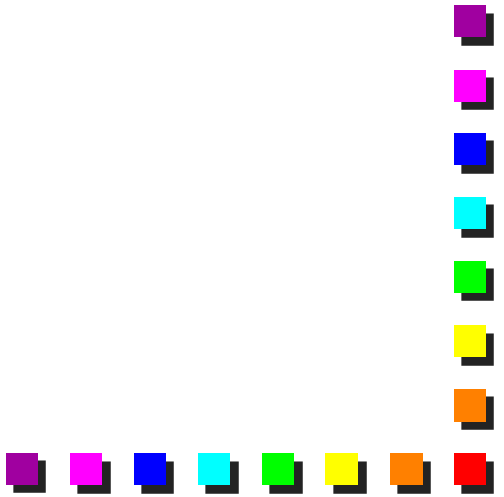
Lightweight virtualization: cgroups, namespaces, LXC, Docker

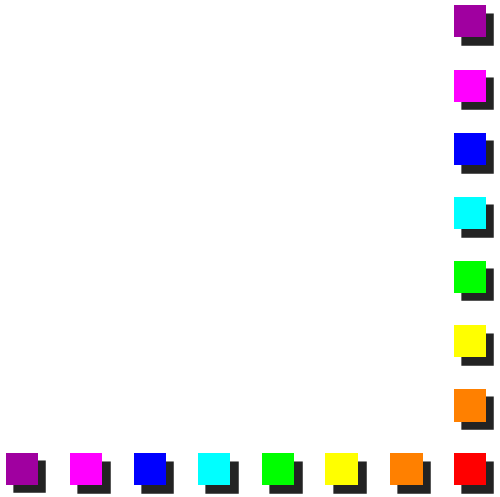
Fulvio Risso

Politecnico di Torino



Part I

INTRODUCTION



Full virtualization with VMs: pro and cons

- Compatible with existing applications / operating systems
- Each application (i.e., VM) can have its own execution environment
  - Kernel version, libraries
- Excellent isolation
  - CPU, memory
- Overhead for the necessity to execute the guest OS
  - Memory (hundred of MB)
  - CPU
- Necessity to configure and keep up-to-date each instance of the guest OS
- OS booting time (tens of seconds or more) may not be acceptable

The idea behind lightweight virtualization

- Create a system that can guarantee the nice properties of computer virtualization (scalability, elasticity, isolation) but that consumes less resources
- Lightweight virtualization is appropriate when:
  - No need for a classical virtual machine
  - The overhead of a classical VM is not acceptable
  - We would like to have an isolated environment that is quick to deploy, migrate and dispose with possibly little or no overhead at all
  - We would like to scale both vertically (thousands of "lightweight VMs" on the same machine) and horizontally (deploy the "lightweight VMs" on many different machines available in a data center)

Lightweight virtualization

- Use Operating System-level virtualization or Application-level virtualization instead of full hardware virtualization
  - Both are software virtualization technologies
- In case of OS-level virtualization, the “hypervisor” is the Linux kernel itself
  - No longer required a dedicated hypervisor
- Virtual environments,, replace classical VMs
  - Virtual environments are also know as virtual private servers, jails, containers
  - Virtual environments feature a given extent of resource management and isolation, usually less than what is achievable with VMs
  - Apps are executed inside these virtual isolated environments

Requirements for lightweight virtualization

- Fine-grained control of resources of the physical machine
  - Possibility to partition and control resources (e.g., CPU cores, RAM) among the different environments
- Security and isolation guarantees
  - Each virtual environment should be assigned to a different app/user, and avoid that a misbehavior in a virtual environment affects the others
- Possibility to manage the entire datacenter as an unique entity, such as with cloud toolkits
  - Even better, capability to integrate lightweight virtualization with a cloud toolkit in order to have the flexibility to deploy VM, containers, etc., upon request

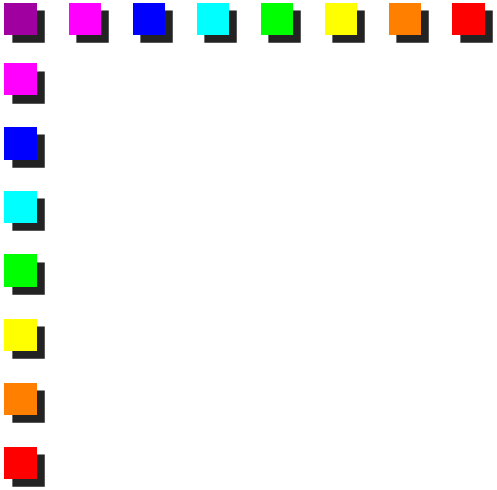
Features for lightweight virtualization

- CPU quotas
- Disk quotas
- I/O rate limit
- Memory limit
- Network isolation
- Checkpointing and live migration
- File system isolation
- Root privilege isolation
- OpenStack support

Possible technologies

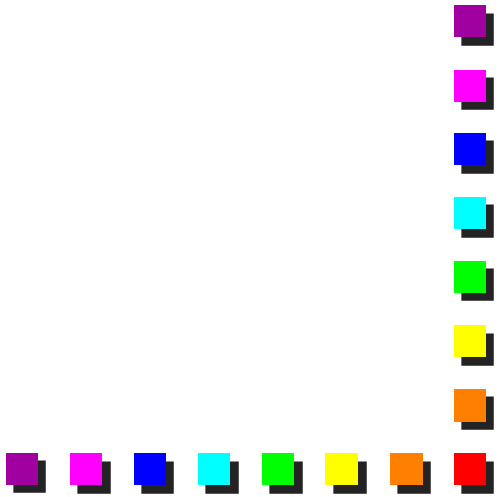
- In theory, several different choices, but, in the end, a few answers
  - Linux containers (LXC) and LXC-based software
  - Other Operating Systems look irrelevant here

- Technologies actually used
  - Linux cgroups and namespaces
  - Linux Containers (LXC)
  - Docker



Part II

LINUX CGROUPS AND NAMESPACES



Linux cgroups

- Linux kernel feature to limit, account, isolate or deny resource usage to processes or to groups of processes
  - Can be seen as a “nice” on steroids
  - Represents the elementary brick that enables a fine-grained control to single processes and resources, providing a way to implement OS-level virtualization
- Consists of two parts: kernel feature and user-space tools that handle the kernel control groups mechanism
- Complex to use, poor documentation and examples available

Cgroup features

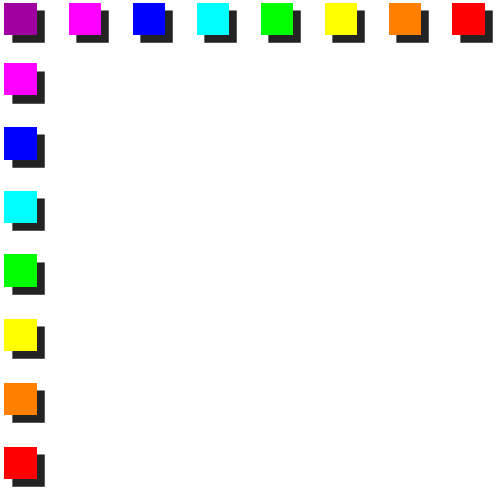
- Resource limiting
  - Memory limiting
- Prioritization
  - CPU quotas
  - I/O throughput
  - Network bandwidth
- Accounting
  - Billing
- Control
  - Freezing
  - Checkpointing
  - Restoring

Linux namespaces

- Another feature of the Linux kernel, technically not part of cgroups, but highly related
- Process groups are separated such that they cannot “see” resources of a given class in the other groups
- Enables to create distinct virtual environment e.g., with respect to networking, file system, and more
  - E.g., two namespaces have completely independent networking stacks, e.g., virtual interfaces, IP addresses associated to it, etc.
  - E.g., two namespaces have visibility on completely independent file systems, such as different the /etc folder, etc.

Linux namespaces: available types

Namespace	Costant	Isolates
IPC	CLONE_NEWIPC	System V IPC, POSIX message queues
Network	CLONE_NEWNET	Network devices, stacks, ports, etc.
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain name



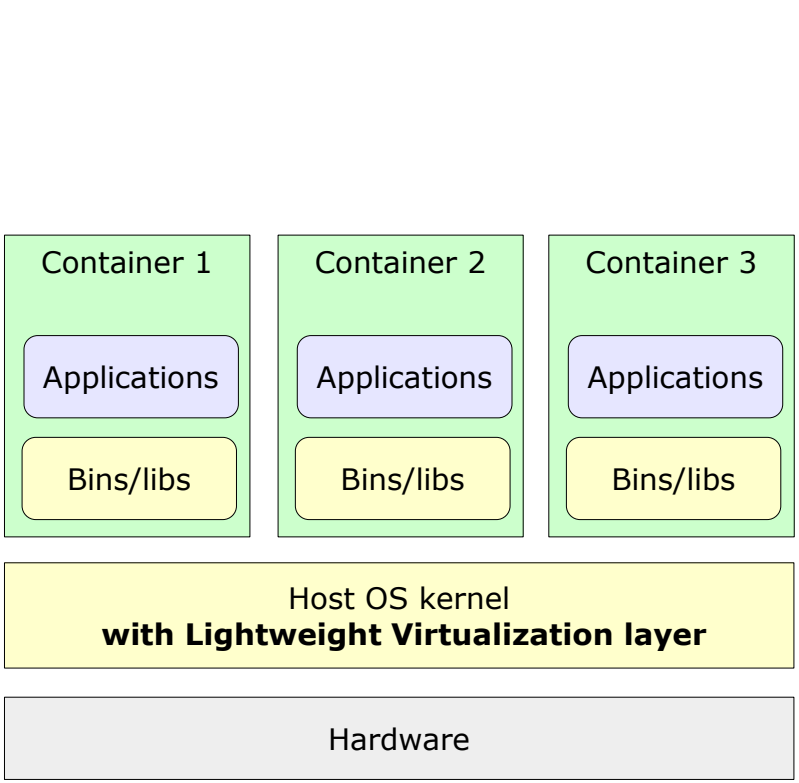
Part III

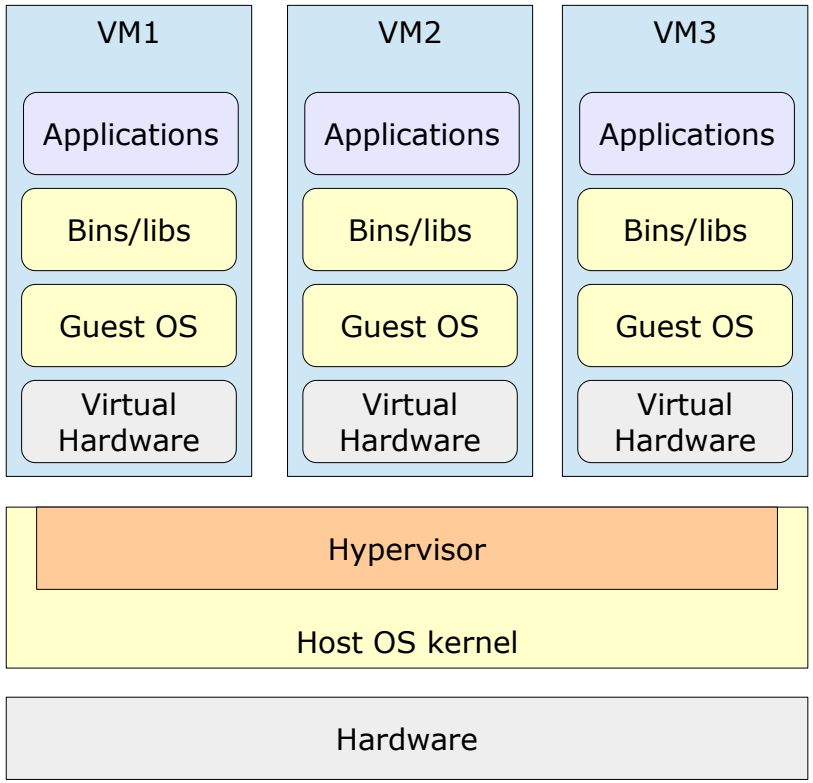
LINUX CONTAINERS

Overview

- Containers provide **lightweight virtualization** that allows to isolate processes and resources without the complexities of full virtualization
- Linux container is an **OS–level** virtualization method for running multiple isolated Linux systems (containers) on a single control host
  - The Linux kernel is shared across all containers

Containers vs. Hypervisors





Containers: what they do...

- They group processes together inside isolated containers (to which different resources can be assigned)
- They share the same operating system as the host
- Inside the box they look like a VM
- Outside the box they look like normal processes

Containers: …and what they don't

- They don't emulate hardware
- They don't run different kernels or OSs (different linux distributions are fine as long as they share the same kernel as the host)
- Security is not an out-of-the-box feature and it must not be taken for granted (still an appropriate level of security can be achieved)

- Q: what is an appropriate level of security for us?

Containers vs VMs

- Containers are:
  - Faster than real VM (to boot, freeze, dispose and to orchestrate)
  - Lighter than VM: less CPU, less memory, no virtualization overhead (e.g., instruction emulation)
    - Denser than VM: due to the inferior resource consumption
    - Container virtualization technology can practically achieve the same performances of native execution
- VMs
  - Provide better isolation (e.g., protect also from kernel exploits)
  - Better security, due to the limited points of attacks (the hypervisor is usually very tiny)
  - Enable the usage of different OSs

LXC

- LXC = **cgroups + namespaces** (+ some other stuff)
  - No kernel patches required, vanilla kernel is ok
- It's a collection of kernel features that can be used to isolate processes in different ways and a userspace tool to use all of these features to create full- fledged containers
- Elementary features are still usable on their own, without LXC

LXC features

- Kernel namespaces (ipc, uts, mount, pid, network and user)
- Chroots (using pivot_root)
- Control groups (cgroups)
- Kernel capabilities
- Apparmor and SELinux profiles
- Seccomp policies

Why LXC?

- Because it is easier than each single elementary component
- For instance
  - Cgroups provides only **resource management**
  - If we want also **isolation**, we need to add also **namespaces**
  - If we want also **security**, we need to add **Apparmor** and/or **SELinux**
  - But what about also live migration, etc…
- LXC does not provide everything, but it provides several features in a (relatively) friendly way
  - LXC allows to configure each container with the list of features it needs, specified in its configuration file
  - We can assign different resources to different containers
  - We can have different levels of isolation between different containers (and the host)

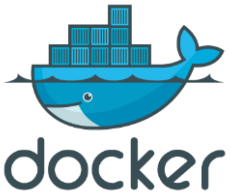
LXC limitations

- Checkpointing and migration
  - Missing features in vanilla linux kernel, we need to user other tools (CRIU), which are not yet 100% working
- Resource isolation
  - We must configure containers not to “see” resources
  - Not always guaranteed; e.g., the resource quota of a container could be affected by the behavior of others

Part IV

DOCKER

Docker overview



- Provides app-oriented OS-level virtualization
  - "Docker is optimized for the deployment of *applications*, as opposed to machines. This is reflected in its API, user interface, design philosophy and documentation. By contrast, the LXC helper scripts focus on containers as lightweight machines, basically servers that boot faster and need less RAM"
  - Usually one application per container
- Easily creates lightweight, **portable**, **self-contained** containers from any application
- Can run "anywhere" (Linux server and architecture must match) with just one build, and is isolated from the host
- Based on LXC, although last versions exploit *libcontainer* (Docker made container toolkit)
- It supports resource management and isolation, plus versioning, component re-use, automatic build and sharing

Docker vs. LXC

- Looks similar, but Docker has with a simplified interface and more powerful management tools
  - Users are more productive with Docker
- Docker containers are portable across machines, while LXC need to be rebuild on the target machine
  - Different LXC versions on different machines can cause problems

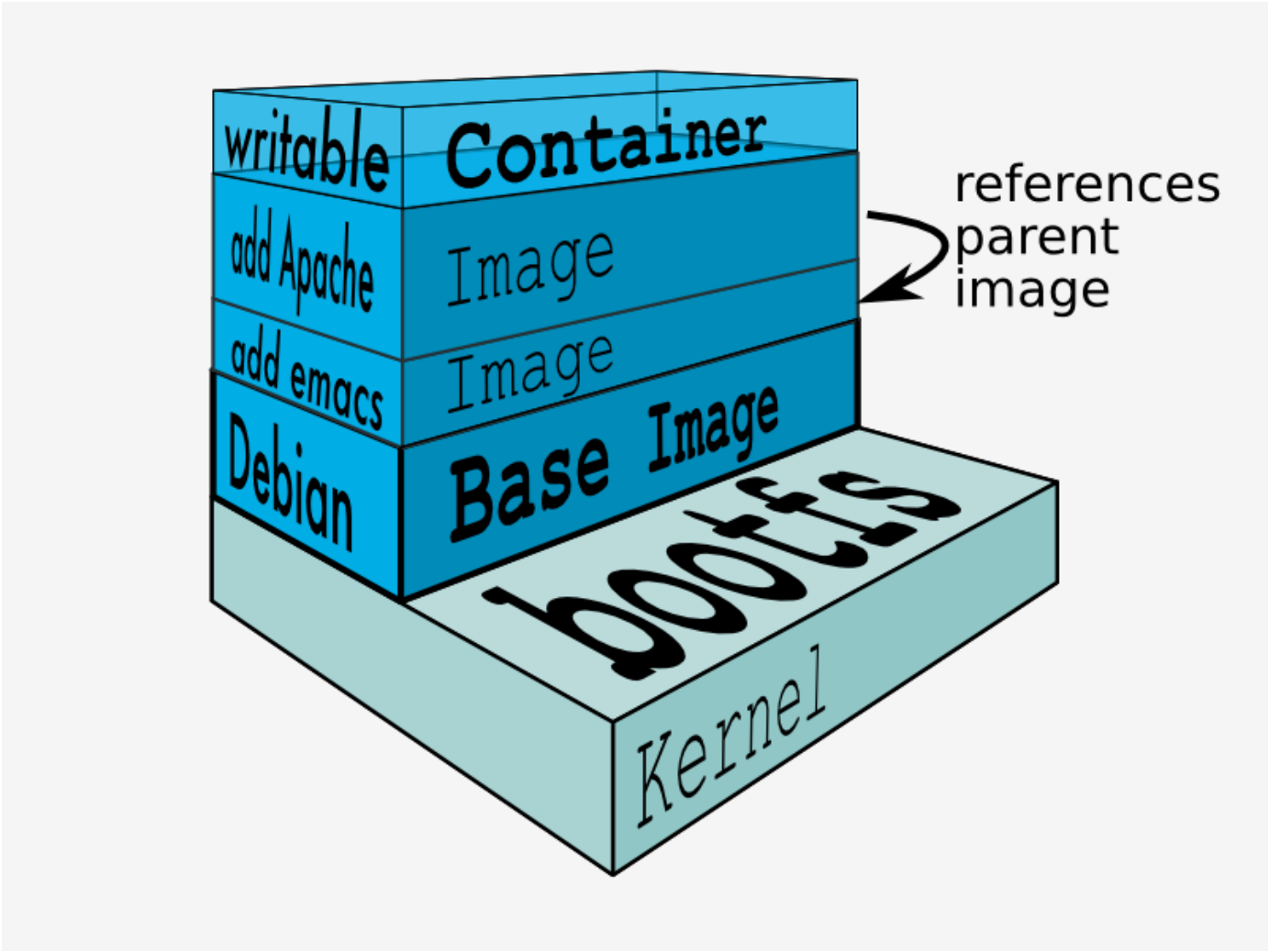
Docker: additional features compared to LXC

- App portability
- Union file system
- Automatic build
- Focus on running applications
- Integration with OpenStack and several cloud vendors (e.g., Google) support
- Versioning
- Component re-use
- Sharing (Docker server)
- Better docs and ecosystem

Union file system

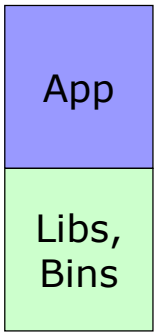
- Layered FS, so there can be read-only parts and read-and-write parts, and those parts can be merged together
- So we could have the common parts of the operating system as read-only, which are shared between all of our containers, and then give to each container its own mount for writing

Union File System in Docker

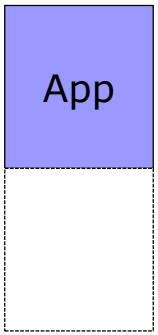


Union File Systems: advantages

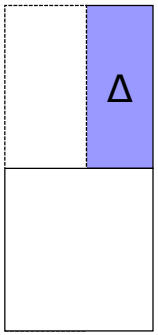
- Only differences are stored
  - E.g., in case an application is modified, we do not need to copy also shared bin/libs
  - E.g., two apps can be created using the same set of base "layers", hence only apps have to be copied
- Reduces
  - Disk footprint on the target host
  - Loading time when launching containers (layers can be cached)



Original App
(with dependencies)



Copy of the App
(shares libs and bins)



Modified App
(only diffs are saved)

Automatic build

- Docker allow developers to automatically assemble a container from its composing elements with full control over application dependencies, build tools, packaging etc.
  - E.g., source code of their app, or package manager (e.g., apt in Linux)
  - The so called “Dockerfile”
- Developers are free to use make, maven, chef, puppet, salt, debian packages, rpms, source tarballs, or any combination of the above, regardless of the configuration of the machines
- Additional advantage: we can create a container that has only the software we need, avoiding the “default” software that is always installed in a vanilla copy of the operating system

Dockerfile example (1)

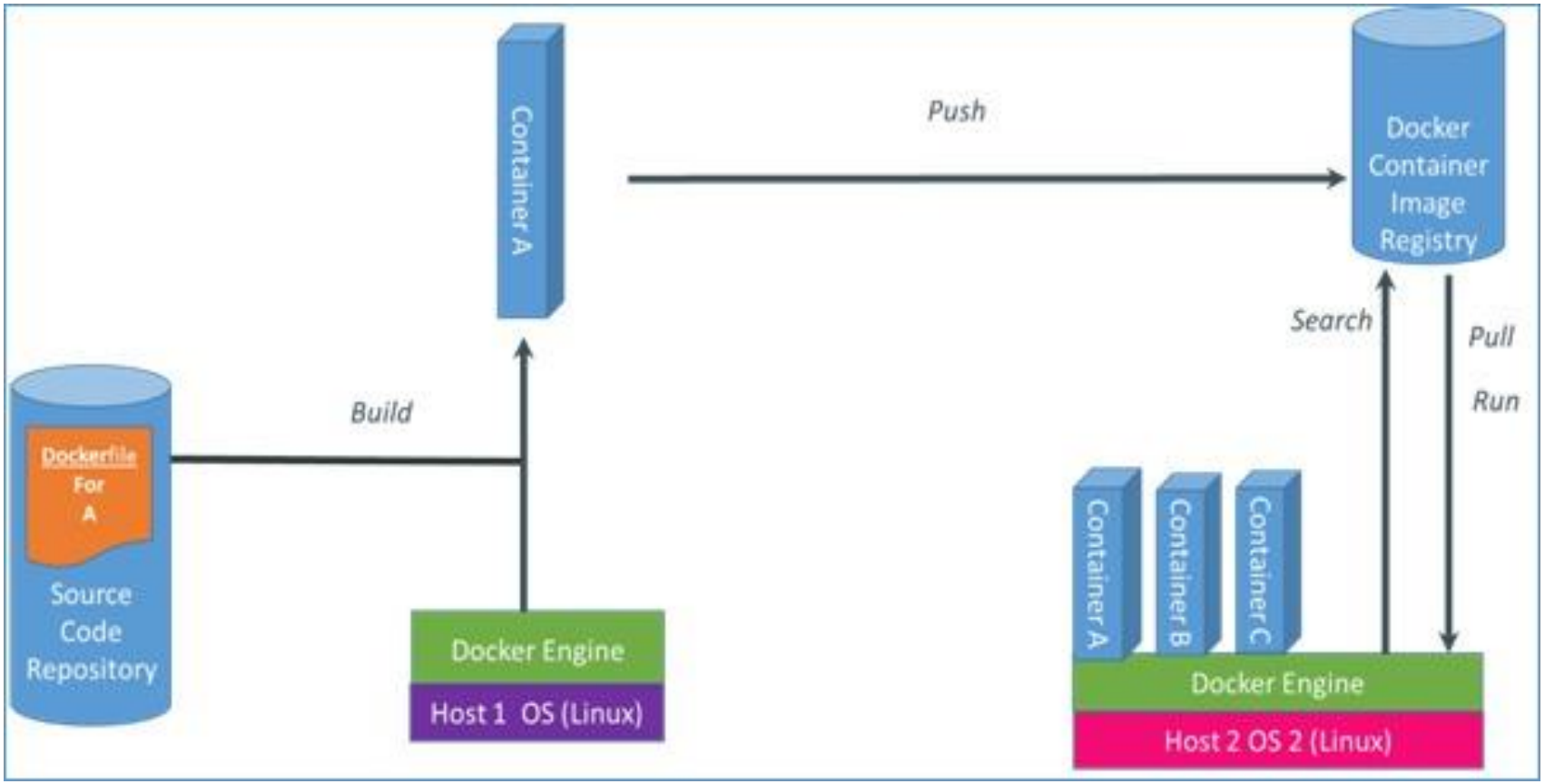
- From DigitalOcean (link)

```
##############################################################
# Dockerfile to build MongoDB container images
# Based on Ubuntu
##############################################################

# Set the base image to Ubuntu
FROM Ubuntu

# File Author / Maintainer
MAINTAINER Example McAuthor

# Update the repository sources list. Not strictly needed, but good
practice
RUN apt-get update

################## BEGIN INSTALLATION ######################
# Install MongoDB Following the Instructions at MongoDB Docs
# Ref: http://docs.mongodb.org/manual/tutorial/install-mongodb-on-ubuntu/

# Add the package verification key
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
```

Dockerfile example (2)

```
# Add MongoDB to the Ubuntu default repository sources list
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-
upstart dist 10gen' | tee /etc/apt/sources.list.d/mongodb.list

# Update the repository sources list once more
RUN apt-get update

# Install MongoDB package (.deb)
RUN apt-get install -y mongodb-10gen

# Create the default data directory
RUN mkdir -p /data/db

###################### INSTALLATION END ######################

# Expose the default port
EXPOSE 27017

# Default port to execute the entrypoint (MongoDB)
CMD ["--port 27017"]

# Set default container command
ENTRYPOINT usr/bin/mongod
```

Docker components and deployment workflow



Resource consumption: some examples

- Memory and CPU
  - A container with a tiny “hello world” program takes about 670 KB of additional RAM compared to the same program running on a vanilla OS
  - The additional CPU consumption of the same program running in a container or on the base host is negligible
- Disk
  - A container for the Ubuntu 14.04 LTS takes up 266MB on disk
  - Installing openjdk-7-jre on adds about 140MB

Conclusions

- Lightweight virtualization is definitely important and it is increasingly used in big enterprises
  - Several cloud companies are already offering commercial services
  - Not always possible to integrate different forms of virtualization (e.g., VMs and Docker) on the same physical infrastructure
- Docker is perhaps the well-known technology, Kubernetes (derived by Google) is another interesting choice