Lecture 9.3 – Containerization and Docker

Luca Morandini
Data Architect – AURIN Project
University of Melbourne
luca.morandini@unimelb.edu.au

Outline of the Lecture

Part 1: Containerization

- Limits of Virtual Machines
- Virtual Machines vs Containers

Part 2: Introduction to Docker

- Images
- Containers
- Dockerfile
- Networking

Part 3: Hands-on Docker

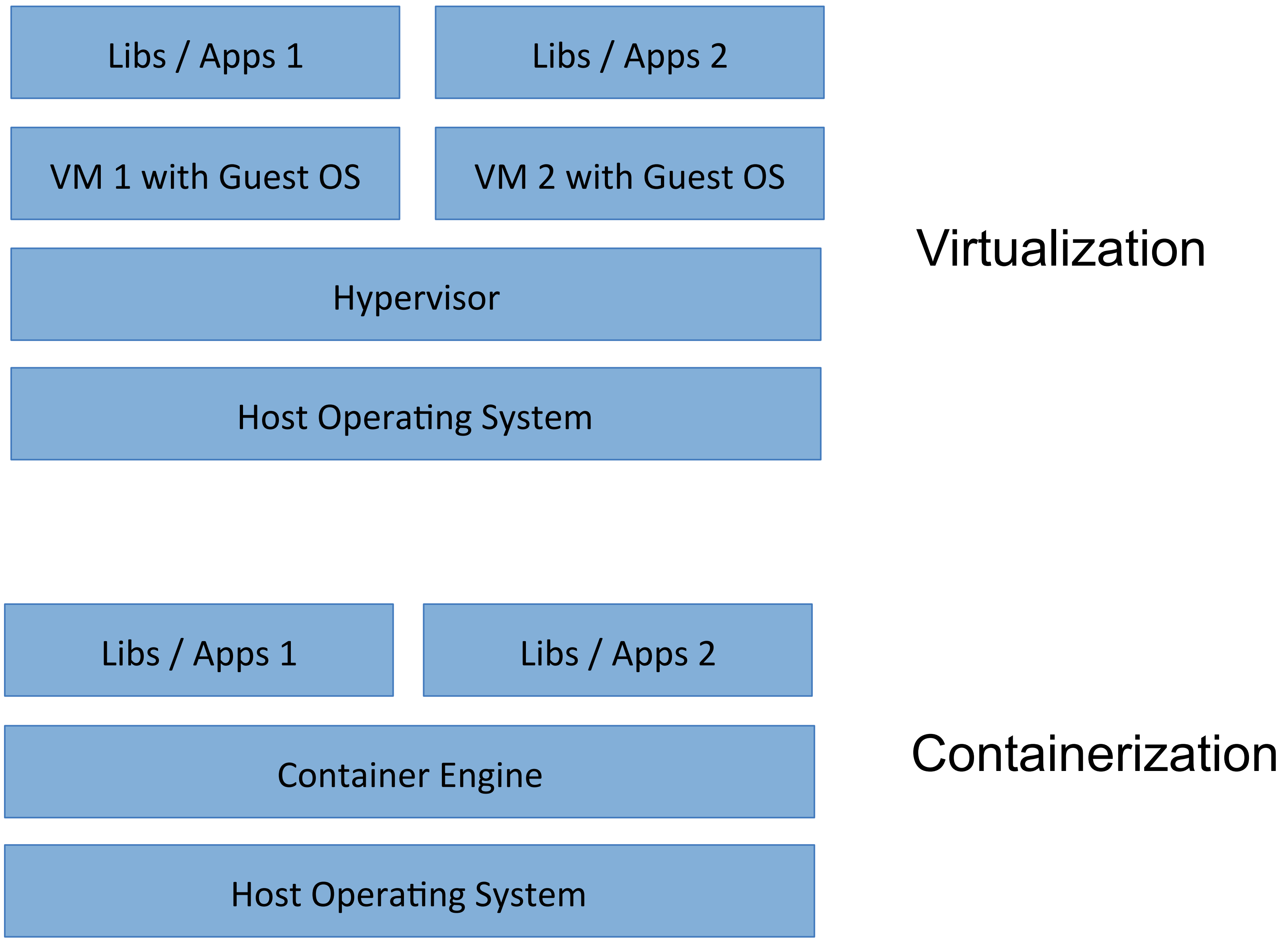
- Building an image
- Running a container

Part 1: Containerization

When Virtualization is not Enough

- The many advantages of virtualization, such as application containment and horizontal scalability, come at a cost: resources. The cost in terms of disk and RAM of replicating an entire operating system can be huge.
- But what if the kernel of the host operating system is used instead of a hypervisor? This would dramatically reduce the resources needed, at the cost of constraining the types of operating systems used as guests.
- Introducing ***containerization***...

Virtualization vs Containerization



VM and Containers (adapted from [1])

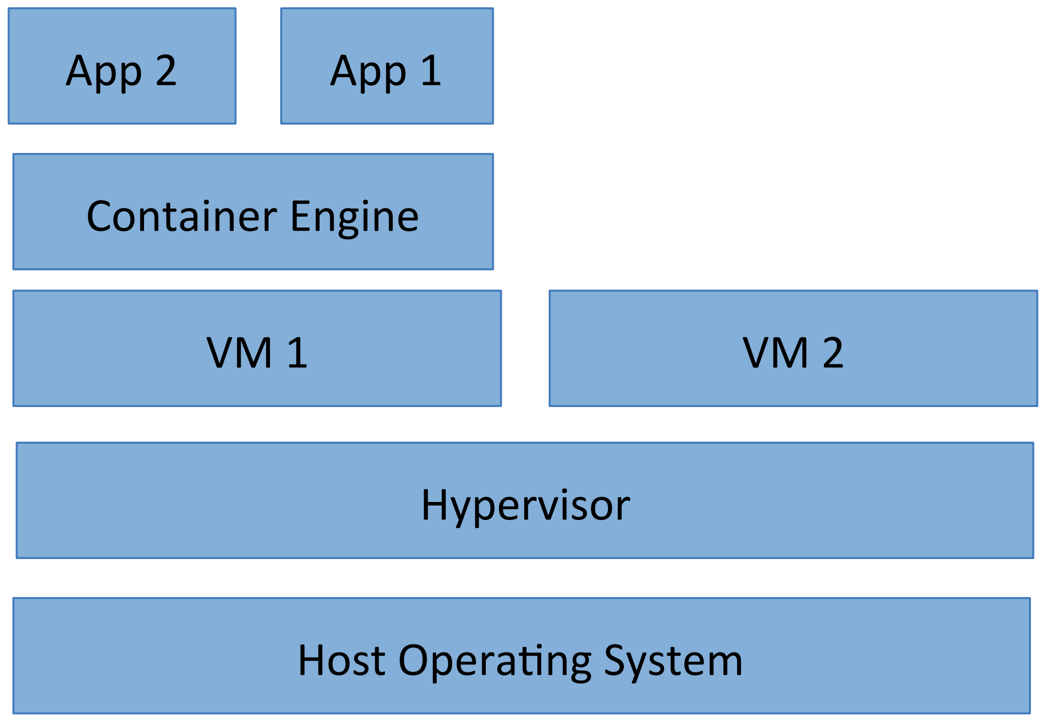
Parameter	Virtual Machines	Containers
Guest OS	Run on virtual HW, have their own OS kernels	Share same OS kernel
Communication	Through Ethernet devices	IPC mechanisms (pipes, sockets)
Security	Depends on the Hypervisor	Requires close scrutiny
Performance	Small overhead incurs when instructions are translated from guest to host OS	Near native performance
Isolation	File systems and libraries are not shared between guest and host OS	File systems can be shared, and libraries are
Startup time	Slow (minutes)	Fast (a few seconds)
Storage	Large size	Small size (most is re-use)

Are Containers Better than VMs?

- It depends on the task at hand…
- If you have to run many applications on the same machine, it is simpler to keep them separated in different containers.
- If you just need to run a single application on a machine, a VM may be simpler to manage
- If you are wondering why *application containment* is so important, think of how upgrading a single library can cause incompatibilities to appear in apparently unrelated applications.
- This grows worse with the age of applications.
- Containers (and VMs) offer a way to isolate applications and related libraries/configurations from each other.

In the Real World, VMs and Containers Co-exist

When deploying applications on the cloud, the base computation unit is a Virtual Machine, hence usually Docker containers are deployed on top of VMs.



Part 2: Introduction to Docker

Docker

- Docker is by far the most successful containerization technology to date
- It uses the Linux kernel (notably cgroups), and LXC technology
- Since Docker is a native Linux technology, it can be used on Mac and Windows machines only by installing a full Linux Virtual Machine
- Microsoft has announced native support for Docker in the next release of Windows Server
- Docker containers run slightly slower than native processes
- Once installed and started, the Docker Engine adds a few commands that are used to create images and containers, manage containers and pull/push images from Docker repositories

A Bit of Docker Nomenclature

- **Container**: a process that behaves like an independent machine, with its own operating system, file system, network interfaces and applications

- **Image**: a blueprint for a container, of which a container is just an instance of, e.g. if I have a CouchDB image on Ubuntu, I could spin four different containers of it on the same computer, everyone offering an independent instance of CouchDB

- **Dockerfile**: the recipe to create an image. Every instruction in the recipe is a layer that is stored independently, so that only changed layers need to be re-run or transferred

- **Docker registry**: a repository of Docker images, of which the most important is the Docker Hub (https://hub.docker.com)

Running a Container

- After installing the Docker Engine, you are ready to go
- Let's suppose you want to spin up a container with a CouchDB cluster of three nodes on it:
- Install a Docker image with the dev version of CouchDB:

```
docker pull klaemo/couchdb:2.0-dev
```

- Start a Docker container with 3 CouchDB instances that listen on port 5984 of the host:

```
docker run -it -p 5984:5984 klaemo/couchdb:2.0-dev --with-admin-party-please --with-
   haproxy -n 3
```

- Check the CouchDB cluster:

```
curl -X GET "http://localhost:5984/_membership"
 {"all_nodes”:
["node1@127.0.0.1","node2@127.0.0.1","node3@127.0.0.1"],"cluster_nodes":
   ["node1@127.0.0.1","node2@127.0.0.1","node3@127.0.0.1"]
 }
```

Running a Shell within a Container

Containers can be started and stopped, and behave like VMs, as they have their own IP addresses, and can be logged into with the exec command

```
docker exec -ti 24f62d2704c1 /bin/bash
```

```
root@24f62d2704c1:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

Containers can be listed and filtered with the ps command:

```
docker ps –all
```

How to Create an image

- While a Docker image taken from a registry can do the job, usually they are created to fit the particular needs of applications
- To build an image, you just need a recipe (*Dockerfile*), and a way to identify the image (name and version)

```
docker build –-file /path/to/Dockerfile –tag “imagename:1.2” ./path/to
```

- (the last argument points to the *context*, a directory containing all the files needed during the build, usually the same directory holding the Dockerfile)
- An image can then be pushed to a Docker registry to be shared

What is in a Dockerfile

A Dockerfile starts with the declaration of the Docker image it is derived from, then continues with a sequence of environment variable declarations, commands (that are executed on the image being built), e.g. instructions to copy files from elsewhere at build time, and ends with a startup command

```
FROM debian:jessie
ENV COUCHDB_VERSION master
….
RUN apt-get update -y -qq && apt-get install -y curl
...
COPY couchdb.ini /usr/local/couchdb
...
ENTRYPOINT /usr/src/couchdb/dev/run
```

ENTRYPOINT

- The ENTRYPOINT command is a command that gets executed when the container derived from the image starts

- If the ENTRYPOINT terminates, then the container starts, executes the command and then exits. Therefore, usually an open-ended process is started, such as a shell, or `tail -f /dev/null`

- `ENTRYPOINT service start apache2 && /bin/bash`

- Parameters can be added to the entrypoint when creating the container. In addition, the entrypoint instruction can be replaced

```
docker create --entrypoint="/bin/bash" -i
  <image id>
```

Host Volumes

- Host file systems can be mounted on containers, allowing for greater flexibility, For instance, configuration files that undergo relatively frequent changes can be mounted as host volumes, avoiding the need to rebuild the image

- To mount a volume (or a series of volumes), you just need to specify the –volume option on the Docker create command

```
docker start
  --volume="/host/dir/:/container/dir"
  <image id>
```

- The same volume can be shared across containers running on the same host

Networking

- Since containers are running within a host, the networking is more complicated than with an infrastructure that uses only VMs. Docker has different networking options:

- Network mode “host”: every container uses the host network stack as the host; which means all containers share the same IP address, hence ports cannot be shared across containers (the first container to reserve a port, blocks it from other containers)

- Network mode “bridge”: the containers act as nodes of a sub-network within the host and are not visible outside

- With the “bridge” option, containers can re-use the same port, as they have different IP addresses, and expose a port of their own as it belongs to the hosts, allowing the containers to be somewhat visible from the outside

Docker Remote API

- In addition to the command-line interface, commands can be set to remote Docker services using the remote API

- The Remote API is ReSTful and uses JSON over HTTP

- For example, to get the list of all containers running on a remote node

```
curl -X GET
 “http://example.com:2375/containers/json”
```

Part 3: Hands-on Docker

Jupyter and R

- Jupyter is a project that aims at delivering a new user experience to scientific software users, allowing them to mix software outputs (charts, tables) with text and live code fragments

- The aim of this hands-on is to install a Docker image with Jupyter and R on a NeCTAR VM

Provisioning of a VM and Docker Install

- Provision an Ubuntu 16.04 VM from NeCTAR
- Add a security group to open port 80
- Open an SSH session to the VM
- Install Docker

```
sudo su -
apt-get update
apt-get install docker.io -y -f
echo 'DOCKER_OPTS="-H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock"' >> /
   etc/default/docker
usermod -aG docker ubuntu
service docker start
```

- Log out of the VM, then log in and try:

```
docker --version
```

Pulling a Docker Image

- Let's have a look at the Dockerfile first:

https://
github.com/jupyter/docker-stacks/blob/master/r-notebook/Dockerfile

- Pulling the Docker image

```
docker pull jupyter/r-notebook
docker images
```

Creating and Running a Container

- We would like to create the container to listen on port 80 (rather than 8888)

```
docker create -p 80:8888 -e PASSWORD="jupyter"  jupyter/r-
  notebook  start-notebook.sh

docker ps –all
docker start <cont. id>
docker logs <cont. id>
```

- Accessing and using Jupyter-R:

```
http://<vm address>>
```

References

[1] Virtualization vs Containerization to Support PaaS, Rajdeep Dua; A Reddy Raja; Dharmesh Kakadia, Cloud Engineering (IC2E), 2014 IEEE International Conference on Cloud Engineering

[2] Docker https://www.docker.com/

[3] Project Jupyter http://jupyter.org/

[4] R on Jupyter Docker Image https://github.com/jupyter/docker-stacks/tree/master/r-notebook