Docker-Container

Module-5

Open Source Software Development

Sources:

- http://people.irisa.fr/Anthony.Baire/docker-tutorial.pdf
- https://docker-curriculum.com/
- https://docs.docker.com/get-started/overview/
- https://docs.docker.com/get-started/
- https://www.cse.wustl.edu/~jain/cse570-18/ftp/m_21cdk.pdf
- https://www.cse.iitb.ac.in/~puru//courses/spring19/cs695/
- https://www.simplilearn.com/tutorials/docker-tutorial/

Introduction

Advantages of Virtualization

- Minimize hardware costs (CapEx) - Multiple virtual servers on one physical hardware.
- Easily move VMs to other data centers
  - Provide disaster recovery. Hardware maintenance.
  - Follow the sun (active users) or follow the moon (cheap power)
- Consolidate idle workloads. Usage is bursty and asynchronous. Increase device utilization.
- Conserve power- Free up unused physical resources
- Easier automation (Lower OpEx) - Simplified provisioning/administration of hardware and software
- Scalability and Flexibility: Multiple operating systems

Problems of Virtualization

- Each VM requires an operating system (OS)
- Each OS requires a license ⇒ CapEx
- Each OS has its own compute and storage overhead
- Needs maintenance, updates ⇒ OpEx
- VM Tax = added CapEx + OpEx

Drawbacks in OS with respect to processes

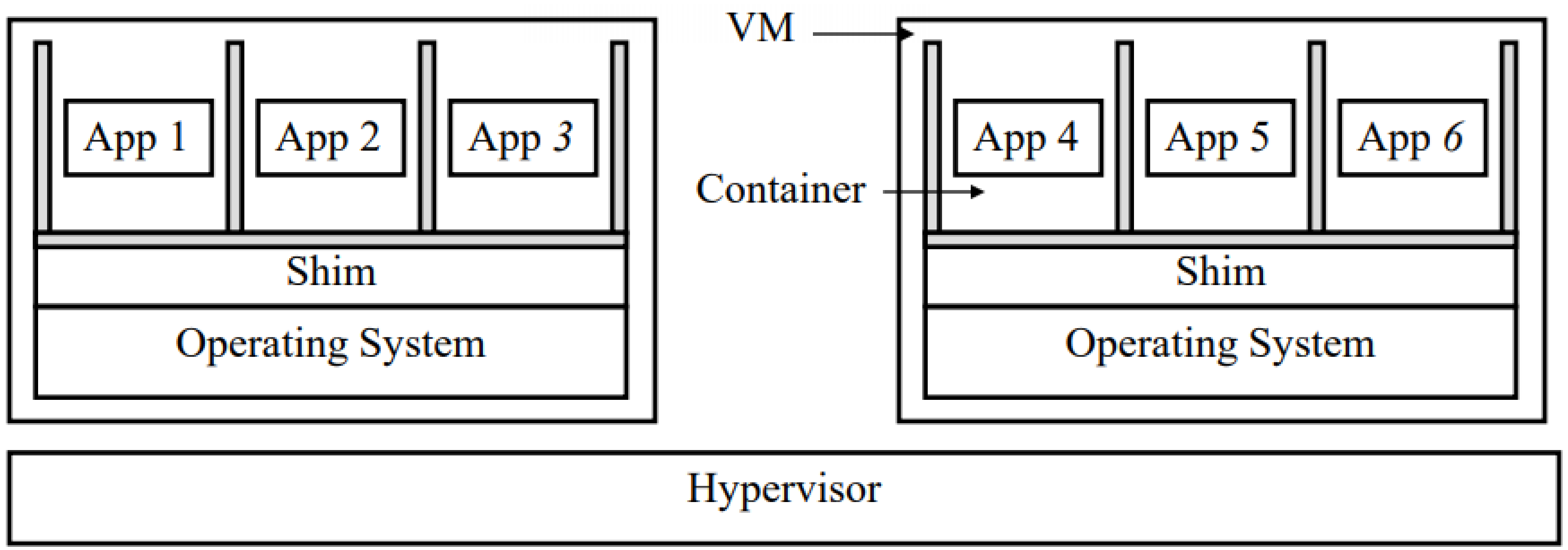
1. **Resource Control :** Sometimes system isn’t able to allocate the requested resources by a process at a given instant. Hence it defers the request and allocates these requested resources when these resources are released by other processes which were using it and allocates them at a later instant to the requesting process. This causes execution of a process to be delayed and this delay may not be always acceptable.

1. **Resource Isolation :** Consider two different processes managed by two different users.These processes would most likely be able to view the resources utilized by one another and might also be able to manipulate them. These processes executed by different users ought to hide process specifics from one another. Hence the inherit design of an operating system provides a very weak form of isolation.

Solution : Containers

- Run many apps in the same virtual machine
  - These apps share the OS and its overhead
  - But these apps can’t interfere with each other
  - Can’t access each other’s resources without explicit permission
  - Like apartments in a complex
- ⇒ Containers

Containers


VM
App 1
App 2
App 3
App 4
App 5
App 6
Container
Shim
Shim
Operating System
Operating System
Hypervisor

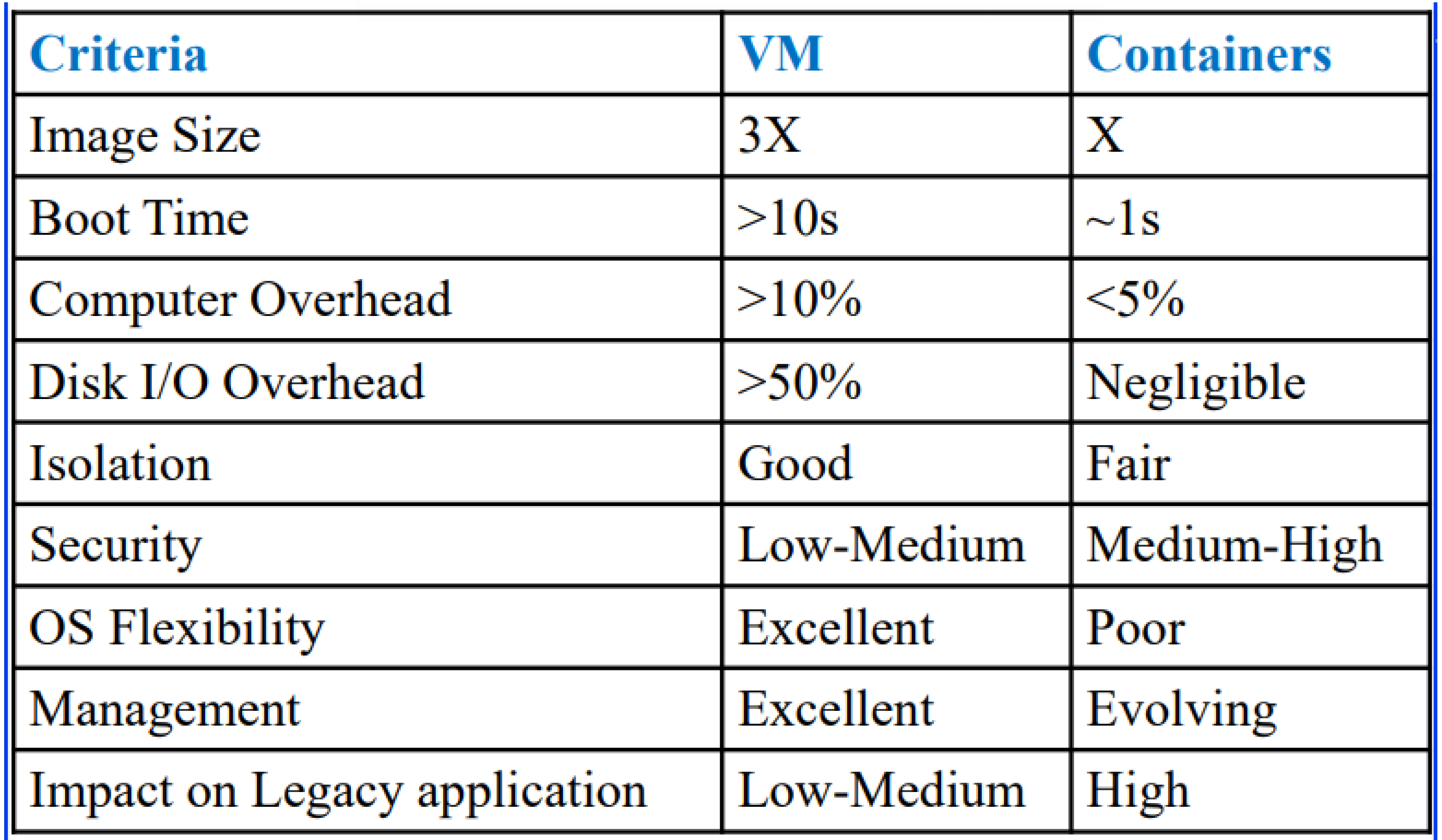
Containers

- Multiple containers run on one operating system on a virtual/physical machine
- All containers share the operating system ⇒ CapEx and OpEx
- Containers are isolated ⇒ cannot interfere with each other
  - Own file system/data, own networking ⇒ Portable

Containers

- Containers have all the good properties of VMs
    - Come complete with all files and data that you need to run
    - Multiple copies can be run on the same machine or different machine ⇒ Scalable
    - Same image can run on a personal machine, in a data center or in a cloud
    - Operating system resources can be restricted or unrestricted as designed at container build time
    - Isolation: For example, “Show Process” (ps on Linux) command in a container will show only the processes in the container
    - Can be stopped. Saved and moved to another machine or for later run

VMs vs Containers



Criteria	VM	Containers
Image Size	3X	X
Boot Time	>10s	~1s
Computer Overhead	>10%	<5%
Disk I/O Overhead	>50%	Negligible
Isolation	Good	Fair
Security	Low-Medium	Medium-High
OS Flexibility	Excellent	Poor
Management	Excellent	Evolving
Impact on Legacy application	Low-Medium	High

What is Docker?

“Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications. Consisting of Docker Engine, a portable, lightweight runtime and packaging tool, and Docker Hub, a cloud service for sharing applications and automating workflows, Docker enables apps to be quickly assembled from components and eliminates the friction between development, QA, and production environments. As a result, IT can ship faster and run the same app, unchanged, on laptops, data center VMs, and any cloud.”

[source: https://www.docker.com/whatisdocker/]

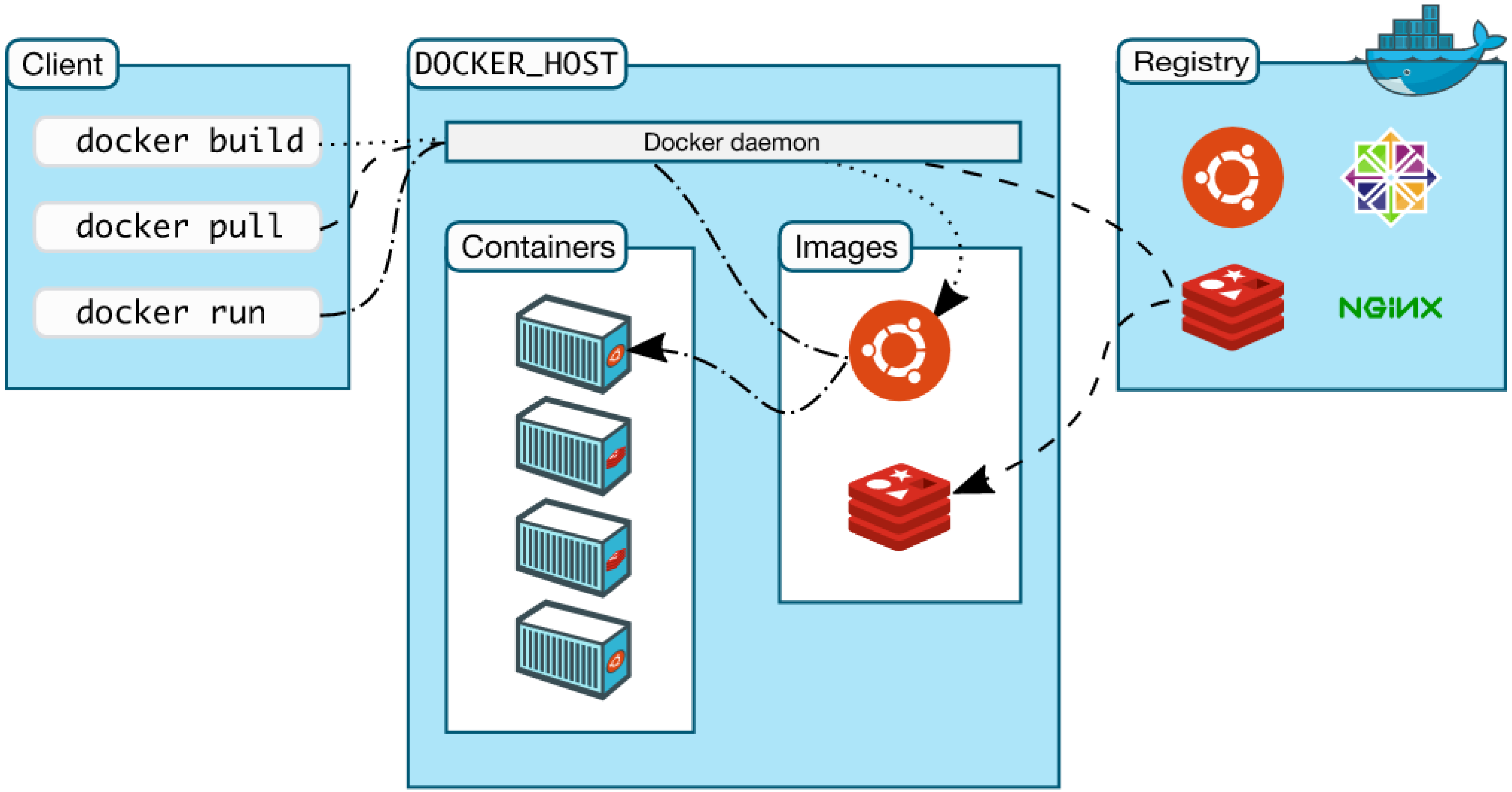
What is Docker ?

- Docker is an open platform for developing, shipping, and running applications.
- Docker enables you to separate your applications from your infrastructure.
- Docker provides the ability to package and run an application in a loosely isolated environment called a container.
- The isolation and security allow you to run many containers simultaneously on a given host.
- Docker provides tooling and a platform to manage the lifecycle of your containers:
  - Develop your application and its supporting components using containers.
  - The container becomes the unit for distributing and testing your application.
  - When you’re ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two.

Docker Architecture

- Docker uses a client-server architecture.
- The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers.
- The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon.
- The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.
- Docker uses a technology called namespaces to provide the isolated workspace called the *container*. When you run a container, Docker creates a set of *namespaces* for that container.

Docker Architecture



Docker Architecture

- **The Docker daemon :** The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
- **The Docker client :** The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

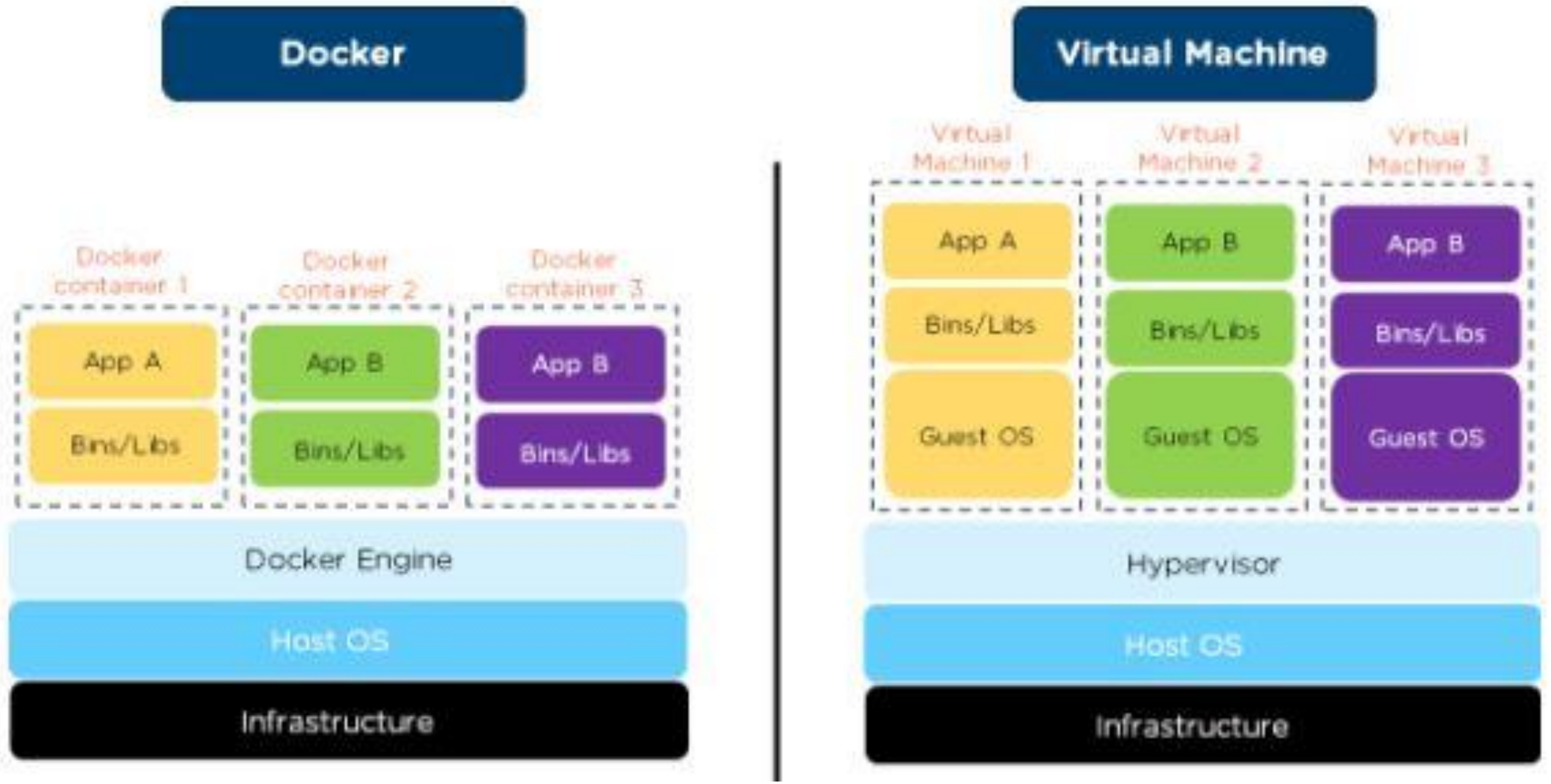
Docker Architecture

- **Docker registries :** A Docker *registry* stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry. When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker Architecture

- **Docker objects:** When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.
    - **Images:** An *image* is a read-only template with instructions for creating a Docker container. Often, an image is *based on* another image, with some additional customization.
    - **Containers:** A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.

Docker vs VM


Docker
Docker container 1
Docker container 2
Docker container 3
App A
Bins/Libs
App B
Bins/Libs
App B
Bins/Libs
Docker Engine
Host OS
Infrastructure
Virtual Machine
Virtual Machine 1
Virtual Machine 2
Virtual Machine 3
App A
Bins/Libs
Guest OS
App B
Bins/Libs
Guest OS
App B
Bins/Libs
Guest OS
Hypervisor
Host OS
Infrastructure

Implementation and Usage

Installation

Link: https://docs.docker.com/get-docker/

For Mac OS: https://docs.docker.com/docker-for-mac/install/

For Windows: https://docs.docker.com/docker-for-windows/install/

For Linux: https://docs.docker.com/engine/install/

Docker Repository: https://hub.docker.com

Automated Script for Linux (Ubuntu)

```
>curl -fsSL https://get.docker.com -o get-docker.sh

>sudo sh get-docker.sh
```

Configure Docker to start on boot

1. **To automatically start Docker and Container on boot for other distros, use the commands below:**

```
$ sudo systemctl enable docker.service

$ sudo systemctl enable containerd.service
```

1. **To disable this behavior, use `disable` instead.**

```
$ sudo systemctl disable docker.service

$ sudo systemctl disable containerd.service
```

1. **Docker Version**

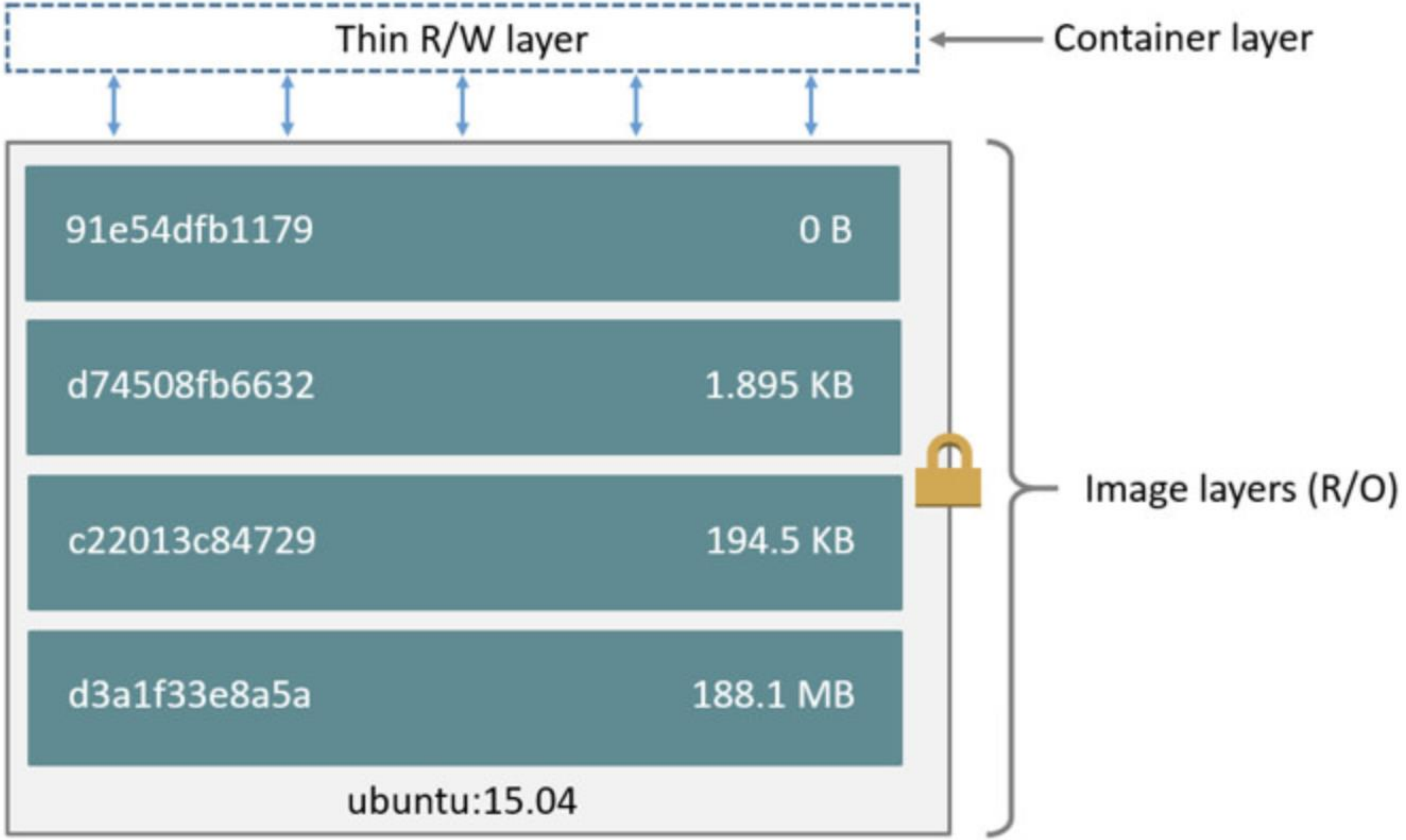
```
$ docker version [OPTIONS]
```

Docker Daemon

- The Docker daemon is a service that runs on your host operating system.
- It currently only runs on Linux because it depends on a number of Linux kernel features, but there are a few ways to run Docker on MacOS and Windows too.
- Start the daemon manually : **dockerd**
- By default this directory is: /var/lib/docker

Docker Images

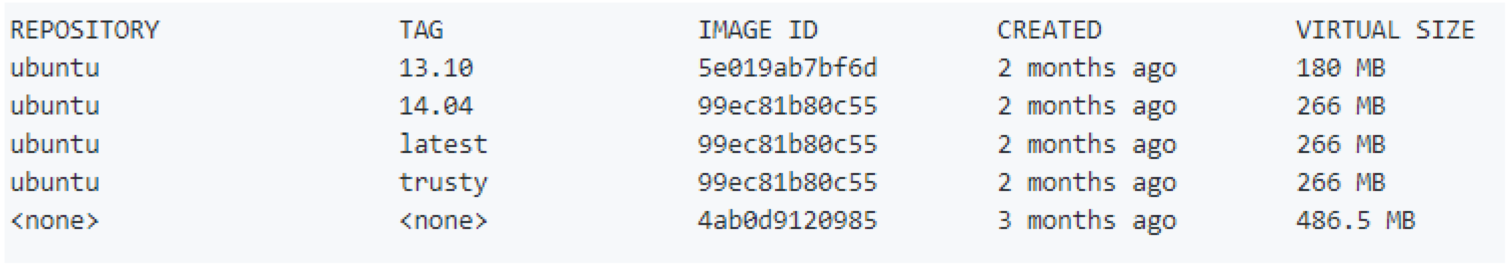
- Docker image is the one that is shipped with code and can be run on any platform where docker Engine is installed.
- Lets take an example: Think a developer writes code and then he packages all the code, dependencies , installables etc in one file called Dockerfile and create an image out of it .


Thin R/W layer
Container layer
91e54dfb1179
0 B
d74508fb6632
1.895 KB
c22013c84729
194.5 KB
d3a1f33e8a5a
188.1 MB
ubuntu:15.04
Image layers (R/O)

Container

Docker Images

Local Images can be seen by: **`docker images`**



```
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              13.10               5e019ab7bf6d        2 months ago        180 MB
ubuntu              14.04               99ec81b80c55        2 months ago        266 MB
ubuntu              latest              99ec81b80c55        2 months ago        266 MB
ubuntu              trusty              99ec81b80c55        2 months ago        266 MB
<none>              <none>              4ab0d9120985        3 months ago        486.5 MB
```

Docker Images

- IMAGE ID is the first 12 characters of the true identifier for an image. You can create many tags of a given image, but their IDs will all be the same (as above).
- VIRTUAL SIZE is virtual because it's adding up the sizes of all the distinct underlying layers. This means that the sum of all the values in that column is probably much larger than the disk space used by all of those images.
- The value in the REPOSITORY column comes from the -t flag of the docker build command, or from docker tag-ing an existing image. You're free to tag images using a nomenclature that makes sense to you, but know that docker will use the tag as the registry location in a docker push or docker pull.

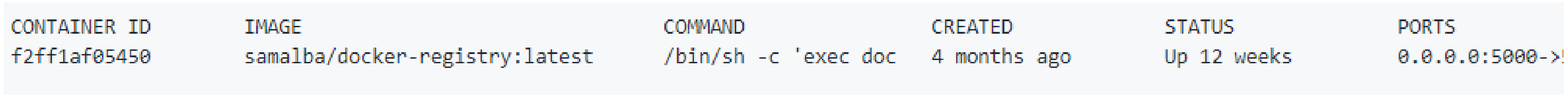
Docker Images

- The full form of a tag is [REGISTRYHOST/][USERNAME/]NAME[:TAG]. For ubuntu above, REGISTRYHOST is inferred to be registry.hub.docker.com. So if you plan on storing your image called my-application in a registry at docker.example.com, you should tag that image docker.example.com/my-application.
- The TAG column is just the [:TAG] part of the full tag. This is unfortunate terminology.
- The latest tag is not magical, it's simply the default tag when you don't specify a tag.
- You can have untagged images only identifiable by their IMAGE IDs. These will get the TAG and REPOSITORY. It's easy to forget about them.

Container

To use a programming metaphor, if an image is a class, then a container is an instance of a class—a runtime object. Containers are hopefully why you're using Docker; they're lightweight and portable encapsulations of an environment in which to run applications.

Local running containers can be seen by: **`docker ps`**



```
CONTAINER ID        IMAGE                             COMMAND              CREATED             STATUS              PORTS
f2ff1af05450        samalba/docker-registry:latest    /bin/sh -c 'exec doc   4 months ago        Up 12 weeks         0.0.0.0:5000->
```

Containers

- Like IMAGE ID, CONTAINER ID is the true identifier for the container. It has the same form, but it identifies a different kind of object.
- docker ps only outputs running containers. You can view all containers (running or stopped) with docker ps -a.
- NAMES can be used to identify a started container via the --name flag.

DTR (Docker Trusted Registry)

- DTR can be installed on any platform where you can store your Docker images securely, behind your firewall.
- DTR has a user interface that allows authorized users in your organization to browse Docker images and review repository events.
- It even allows you to see what Dockerfile lines were used to produce the image and, if security scanning is enabled, to see a list of all of the software installed in your images.

DTR (Docker Trusted Registry)

- **Availability :** DTR is highly available as it has multiple replicas of containers in case anything fails.
- **Efficiency:** DTR has this ability to clean the unreferenced manifests and cache the images as well for faster pulling of images.
- **Built-in access control :** STR has great authentication mechanisms like RBAC , LDAP sync. It uses the same authentication as of UCP.
- **Security scanning :** Image Scanning is built in feature provided out of the box by DTR.
- **Image signing :** DTR has built in Notary, you can use Docker Content Trust to sign and verify images.

Docker Image Commands

1. Downloading Docker Image:
   - Search: **`docker search ubuntu:18.04`** (specific version)
   - Pull: **`docker pull ubuntu`** (latest version)
2. Listing Images
   - **`docker images`**
   - **`docker images -a`** (show all images)
3. List images by name and tag
   - **`docker images <image-name>:<tag>`**
4. List the full length image IDs
   - **`docker images --no-trunc`**
5. Remove Images:
   - **`docker rmi <imagename>`**

Saving Images and Containers as Tar Files for Sharing

Four basic Docker CLI for saving and loading images and containers:

- The **`docker export`** - Export a container’s filesystem as a tar archive
- The **`docker import`** - Import the contents from a tarball to create a filesystem image
- The **`docker save`** - Save one or more images to a tar archive (streamed to STDOUT by default)
- The **`docker load`** - Load an image from a tar archive or STDIN

Example

1. Displaying Running Container
   - **`docker ps -a`**
2. Export the file
   - **`docker export containerid > output.tar`**
3. For Images:
   - **`docker save -o mynginx1.tar nginx`**
   - **`docker load < mynginx1.tar`**

Container Commands

- Start a new Container from an image
  - `docker run <imagename>`
- List Running Containers
  - `docker ps`
- List All Containers
  - `docker ps -a`
- Stop/Start a container
  - `docker stop <name> or <id>`
  - `docker start <name> or <id>`
- Remove a Container
  - `docker rm <name>or <id>`

Container Commands

- Execute a Command in Container:
  - **`docker exec <container-name> <command>`**
- Run a container in attached mode:
  - **`docker run <container-name>`**
- Run a container in detached mode:
  - **`docker run -d <container-name>`**
- Attaching a detached container
  - **`docker attach <container-id>`**
- Delete all stopped containers
  - **`docker container prune`**

Docker Run Command (in Detail)

- Running a container with specific image tag
  - **`docker run <image>:<tag>`**
  - **`Default mode for running a container is non-interactive.`**
- Running in Interactive mode:
  - **`docker run -i <image>`**
- Running Interactive Mode with Terminal
  - **`docker run -i -t <image>`**
- Port Mapping of Containers (Help to run multiple instances separately)
  - **`docker run -p <docker-port>:<container-port> image`**
  - **`docker run -p 80:19990 mywebserver`**

Container Command

- Volume Mapping (DataBase Mapping with System Storage)
  - **`docker run -v <system-storage location>:<containerdefaultloc> image`**
  - **`docker run -v /opt/data/datadir:/var/lib/mysql mysql`**
- Inspect Container (Additional Detail About a Container)
  - **`docker inspect <container name>`**
- Container Logs:
  - **`docker logs <container-name>`**
- Stats of a Running Container
  - **`docker stats`**

Dockerfile

1. Docker gives you the capability to create your own Docker images, and it can be done with the help of Docker Files.
2. A Dockerfile is a text file which contains a series of commands or instructions.
3. These instructions are executed in the order in which they are written.
4. Execution of these instructions takes place on a base image.
5. On building the Dockerfile, the successive actions form a new image from the base parent image.

Creating a Dockerfile

- Create a file called Docker File and edit it using any default editor.
- Please note that the name of the file has to be "Dockerfile" with "D" as capital.
- sudo gedit Dockerfile
- Build your Docker File using the following instructions.

```
FROM ubuntu

MAINTAINER usr@gmail.com

RUN apt-get update

RUN apt-get install –y mininet

CMD [“echo”,”Image created”]
```

Creating a DockerFile

The following points need to be noted about the above file −

- The first line "#This is a sample Image" is a comment. You can add comments to the Docker File with the help of the # command
- The next line has to start with the **FROM** keyword. It tells docker, from which base image you want to base your image from. In our example, we are creating an image from the ubuntu image.
- The next command is the person who is going to maintain this image. Here you specify the **MAINTAINER** keyword and just mention the email ID.
- The **RUN** command is used to run instructions against the image. In our case, we first update our Ubuntu system and then install the nginx server on our ubuntu image.
- The last command **CMD** is used to display a message to the user.

Additional Dockerfile Instructions

1. VOLUME : define mountable directories
2. WORKDIR : define working directory
3. EXPOSE : listens on the specified network ports at runtime
4. ENTRYPOINT : This command can also be used to execute commands at runtime for the container.
5. ENV: This command is used to set environment variables in the container.

Building the Dockerfile

```
docker build
docker build  -t ImageName:TagName dir
```

Options

- -t − is to mention a tag to the image
- ImageName − This is the name you want to give to your image.
- TagName − This is the tag you want to give to your image.
- Dir − The directory where the Docker File is present.

Example: `sudo docker build –t myimage:0.1`

Sample Dockerfile

```
FROM ubuntu

MAINTAINER usr@gmail.com

WORKDIR /newtemp

CMD pwd
```

Sample Dockerfile

```
FROM node:8.11-slim
ENV workdirectory /usr/node
WORKDIR $workdirectory
WORKDIR app
COPY package.json .
RUN ls -ll &&\
   npm install
# command executable and version
ENTRYPOINT ["node"]
```

Sample Dockerfile

```
FROM node:$NODE_VERSION
ENV workdirectory /usr/node
WORKDIR $workdirectory
WORKDIR app
COPY package.json .
RUN ls -ll &&\
   npm install
RUN useradd abc
USER abc
ADD index.js .
RUN ls -l
EXPOSE 3070
ENTRYPOINT ["node"]
```

Pushing and Pulling to and from Docker Hub

Getting an image to Docker Hub

- Imagine you made your own Docker image and would like to share it with the world you can sign up for an account on https://hub.docker.com/. After verifying your email you are ready to go and upload your first docker image.
- Log in on https://hub.docker.com/
- Click on *Create Repository*.
- Choose a name (e.g. verse_gapminder) and a description for your repository and click *Create*.

- Log into the Docker Hub from the command line

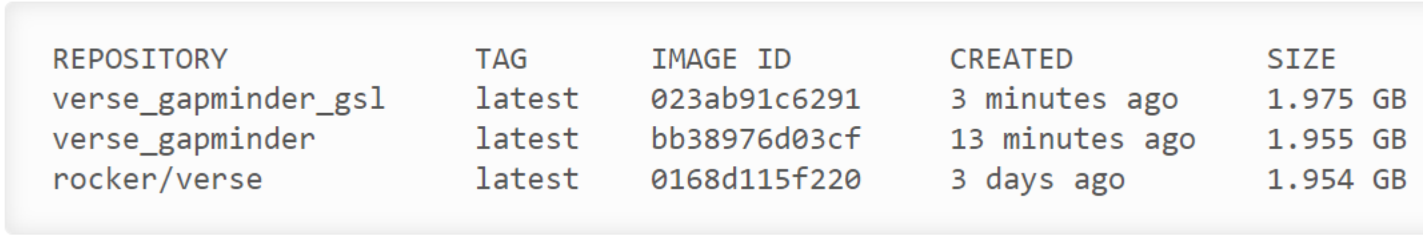
```
docker login --username=yourhubusername --email=youremail@company.com
```

- just with your own user name and email that you used for the account. Enter your password when prompted. If everything worked you will get a message similar to

```
WARNING: login credentials saved in /home/username/.docker/config.json
   Login Succeeded
```

Check the image ID using
`docker images`

- and what you will see will be similar to



```
REPOSITORY            TAG       IMAGE ID        CREATED           SIZE
verse_gapminder_gsl   latest    023ab91c6291    3 minutes ago     1.975 GB
verse_gapminder       latest    bb38976d03cf    13 minutes ago    1.955 GB
rocker/verse          latest    0168d115f220    3 days ago        1.954 GB
```

and tag your image

```
docker tag bb38976d03cf yourhubusername/verse_gapminder:firsttry
```

- Push your image to the repository you created

```
docker push yourhubusername/verse_gapminder
```

Pushing to Docker Hub is great, but it does have some disadvantages:

- Bandwidth - many ISPs have much lower upload bandwidth than download bandwidth.
- Unless you’re paying extra for the private repositories, pushing equals publishing.
- When working on some clusters, each time you launch a job that uses a Docker container it pulls the container from Docker Hub, and if you are running many jobs, this can be really slow.

Solutions to these problems can be to save the Docker container locally as a a tar archive, and then you can easily load that to an image when needed.

- To save a Docker image after you have pulled, committed or built it you use the docker save command.

For example, lets save a local copy of the verse_gapminder docker image we made:

```
docker save verse_gapminder > verse_gapminder.tar
If
```

- we want to load that Docker container from the archived tar file in the future, we can use the docker load command:

```
docker load --input verse_gapminder.tar
```

- Getting an image from Docker Hub
- Docker Hub is the place where open Docker images are stored. When we ran our first image by typing

```
docker run --rm -p 8787:8787 rocker/verse
```

- the software first checked if this image is available on your computer and since it wasn’t it downloaded the image from Docker Hub. So getting an image from Docker Hub works sort of automatically. If you just want to pull the image but not run it, you can also do

```
docker pull rocker/verse
```

Network containers

- Docker includes support for networking containers through the use of **network drivers**.
- By default, Docker provides two network drivers for you, the `bridge` and the `overlay` drivers.
- Every installation of the Docker Engine automatically includes three default networks.
- List networks: **`docker network ls`**
- Inspecting the network: **`docker network inspect <networkname>`**
- Example: **`docker network inspect bridge`**

Network Drivers

- **None:** None is straightforward in that the container receives a network stack, but lacks an external network interface.
- **Bridge:** A Linux bridge provides a host internal network in which containers on the same host may communicate, but the IP addresses assigned to each container are not accessible from outside the host.
- **Host:** a newly created container shares its network namespace with the host, providing higher performance and eliminating the need for NAT.
- **Overlay :** Overlay networks connect multiple Docker daemons together and enable swarm services to communicate with each other. You can also use overlay networks to facilitate communication between a swarm service and a standalone container, or between two standalone containers on different Docker daemons.

What to choose when?

- **User-defined bridge networks** are best when you need multiple containers to communicate on the same Docker host.
- **Host networks** are best when the network stack should not be isolated from the Docker host, but you want other aspects of the container to be isolated.
- **Overlay networks** are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services.
- **Macvlan networks** are best when you are migrating from a VM setup or need your containers to look like physical hosts on your network, each with a unique MAC address.
- **Third-party network plugins** allow you to integrate Docker with specialized network stacks.

More Reading Container Networking: (Not in Syllabus)

- Standalone networking tutorial
- Host networking tutorial
- Overlay networking tutorial
- Macvlan networking tutorial