Big Data - Labo

Tuur Vanhoutte

8 februari 2021

Inhoudsopgave

1 Intro

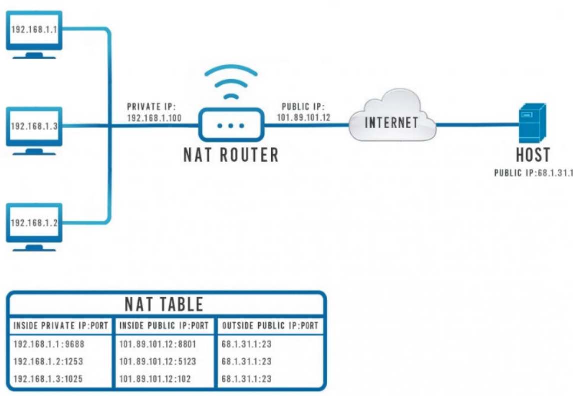
Topics:

- Linux basics + containers
- Elastic search (text search, document store)
- Linux Batch Processing & Dask
- InfluxDB (timeseries)
- Cloud services (Kafka, Kinesis, Lambda, ML services, . . .)

2 NAT-ing

2.1 NAT

= Network Address Translation



Figuur 1: NAT diagram

2.1.1 The problem

- We only have one (public/private) IP-address
  - Howest: 172.23.82.60
- Connecting to a server over a network:
  - Using a protocol (HTTP) which uses TCP
  - Our server has an IP address: 172.23.82.60
  - Our server is listening at port 5000
  - ⇒ `http://172.23.82.60:5000`
- Problem: We want to have multiple IP addresses
  - Student 1 wants to reach http://192.168.20.21:5000
  - Student 2 wants to reach http://192.168.20.22:5000
  - Student x wants to reach http://192.168.20.xx:5000

2.1.2 The solution

Translation is needed!

- 172.23.82.60:5000 should point to 192.168.20.21:5000
- 172.23.82.60:5001 should point to 192.168.20.22:5000
- 172.23.82.60:5xxx should point to 192.168.20.xx:5000

We can use any port, on both sides:

- 172.23.82.60:8000 can point to 192.168.20.21:5000
- 172.23.82.60:8000 can point to 192.168.20.21:3000

2.2 SSH Tunnel

= SSH Port Forwarding

Resource	Internal IP	Username	Password	External port	Internal port
Vyos Router	192.168.50.1	vyos	P@ssw0rd	7000	22
Storage	192.168.50.2	student	P@ssword	n.v.t.	22
SSH	192.168.50.3	student	P@ssword	7040	22
RDP	192.168.50.4	Administrator	P@ssword	7020	3389
vCenter vSphere	192.168.50.10	administrator@vsphere.local	P@ssword	7060	443
vCenter appliance	192.168.50.10	root	P@ssword	n.v.t.	5480
ESXi-00	192.168.50.11	root	P@ssword	n.v.t.	22
ESXi-01	192.168.50.12	root	P@ssword	n.v.t.	22

Figuur 2: Example

3 Container technology

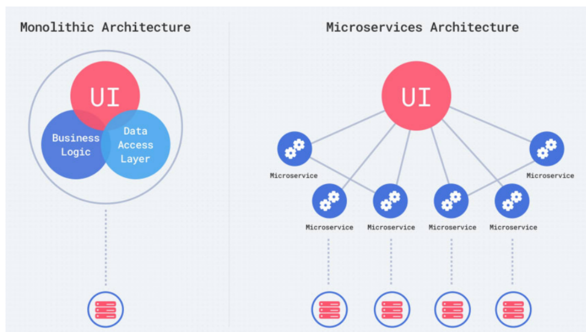
3.1 Docker

- Docker = ecosystem for creating and running containers
- Docker wants to make it possible to install and run software on any system
- Other reasons: Microservices/DevOps/Resource usage
- Docker != Container
  - Docker CLI
  - Docker Engine
  - Docker Image
  - Docker Container
  - Docker Hub
  - Docker Compose
  - Docker Swarm
  - ...

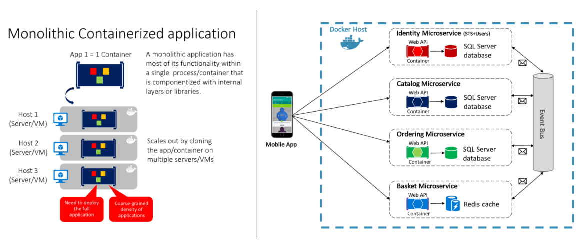
3.2 Microservices

- = A software development technique
- Structure an application as a collection of loosely coupled services
- Lightweight
- Microservices-based architectures enable continuous delivery and deployment
- `https://en.wikipedia.org/wiki/Microservices`

3.2.1 Monolithic vs Microservices



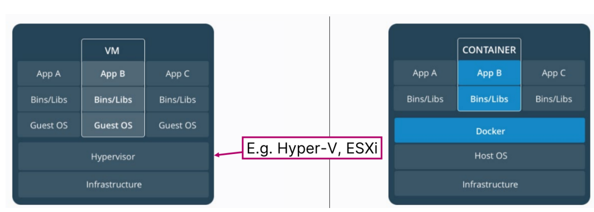
Figuur 3: Monolithic architecture vs Microservices architecture



Figuur 4: Monolithic Containerized application

Microservices does **not** necessarily mean containerization!

3.3 Virtualization vs Containerization



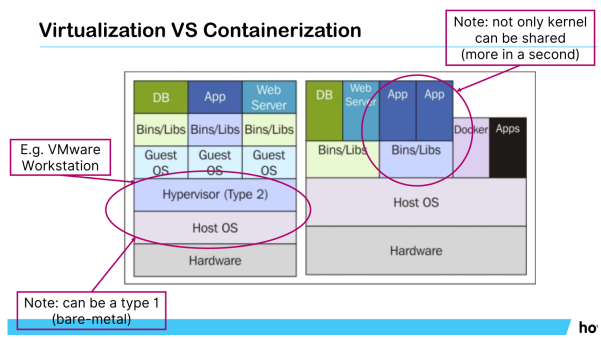
Figuur 5: Virtualization vs Containerization

3.3.1 Virtualization

- = An abstraction of physical hardware turning one server into many servers
- Multiple VMs can run on the same machine
- Each VM includes a full copy of an Operating System (OS), one or more apps
- Takes a lot of space
- Can be slow to boot

3.3.2 Containerization

- = An abstraction at the app layer that packages code and dependencies together
- Multiple containers can run on the same machine, they share the OS kernel with each other, each running as isolated processes in user space.
- Takes up less space than VMs
- Boot up almost instantly

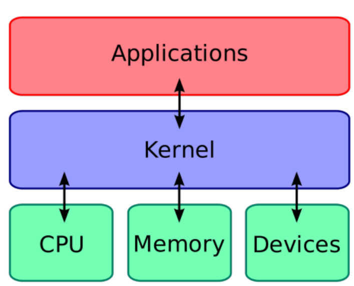


Figuur 6: Schematic

3.4 Shared kernel

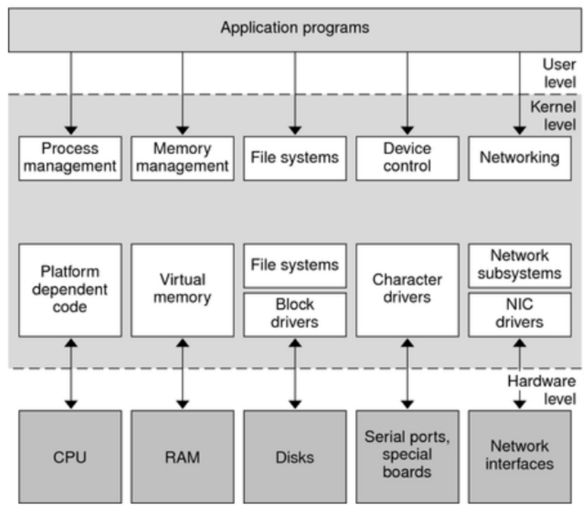
3.4.1 What is a kernel?

- Piece of software that offers basic functionality to the OS
- System calls: open, read, write, close, wait, exit, ...
- A typical kernel has a few hundred system calls



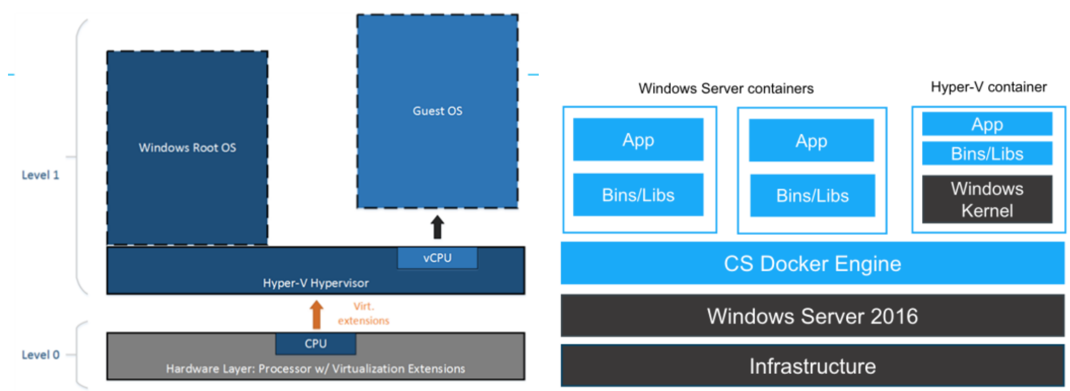
Figuur 7: The kernel is the layer that communicates between hardware and applications

- Docker shares the host OS kernel
  - Host OS: Windows / MacOS / Linux
  - Shared Linux Kernel



Figuur 8: Kernel in detail

- The Ubuntu container requires the Linux kernel
- The Linux kernel runs in a Virtual Machine



Figuur 9

3.4.2 How?

Two important Linux kernel features:

- **Namespaces** are a feature of the Linux kernel that partitions kernel resources
- **cgroups** (control croups) is a Linux kernel feature that limits, accounts for, and isolates resource usage of a collection of processes

Simpler:

- Namespaces = isolating resources per process (or group of processes)
- cgroups = Limitating resource usage per process (or group of processes)

3.4.3 Namespaces

- 7 types:
  - mount, UTS, IPC, network, PID, cgroup, user
- For the process (or group of processes) it looks like there is a completely isolated set of resources

3.4.4 Containers

What is a container?

- One or more running processes (if not running anymore ⇒ container dead)
- Resources are specifically assigned to it
- The real bulding blocks: Linux kernel features
  - Namespaces
  - cgroups

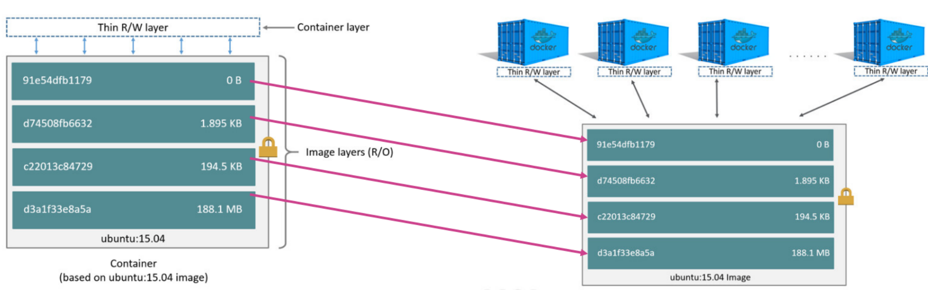
3.5 Images

What is an image?

- Filesystem snapshot
- Startup command
- Layered structure **(!)**

Instance of image = container

3.5.1 Image layer



Figuur 10: Image layers

- RUN, COPY, ADD
  - = new read-only layer

- Top layer = container layer
  - Writeable
- Delete container = delete container layer
  - Image will still exist
  - Peristent volumes

3.6 Docker is lightweight

- Shared kernel
- Container has no OS
- Less disk space ⇒ sharing layers
- Small community images
  - ex: Alpine Linux (small, simple, secure)
- Current Docker version is using runC (previously LXC = Linux Containers)
  - runC = tooling (written in Go) that makes it possible to create and run containers
  - runC = CLI to 'easily' access kernel features such as cgroups and namespacing
  - runC = successor of libcontainer (developed by Docker)
  - Open-sourced ⇒ better community
  - runC implements 'Open Container Initiative Runtime Specification'
  - `https://github.com/opencontainers/runtime-spec`

Docker is 'nothing more' than an ecosystem about creating & running containers

3.7 Using Docker

(see slides 40-55)