SCC.311: Software Containment

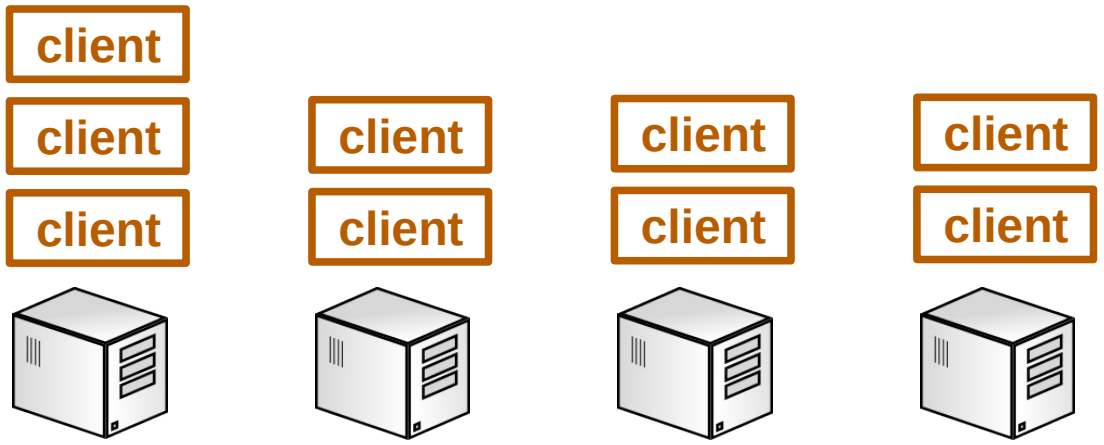
Prof. Barry Porter

Scaling a service

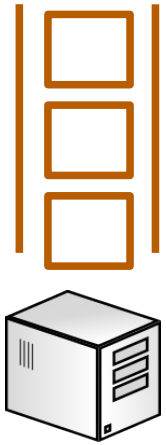
client client client client client client client client client

Scale **up** (vertical scaling)

Scale **out** (horizontal scaling)



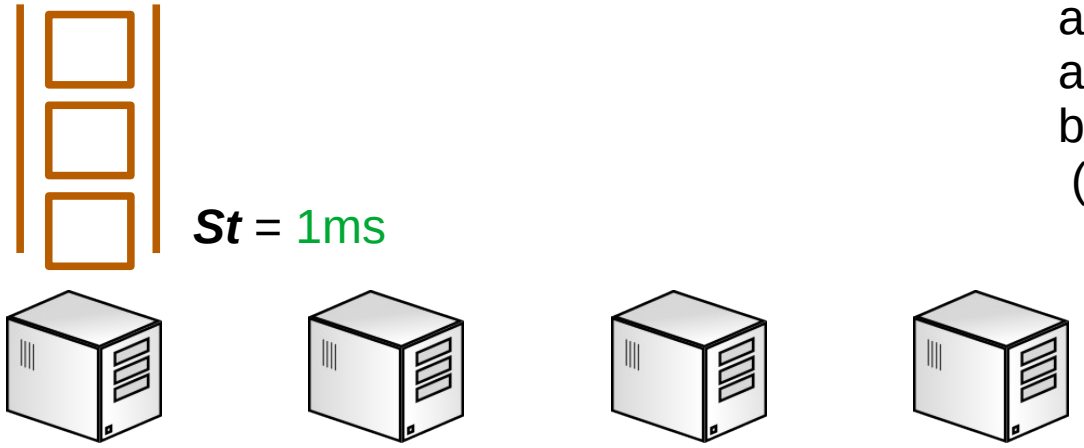
When to scale out?



assume a
single queue
on each machine

assume an
average service
time of ***St***

When to scale out...in a datacentre?



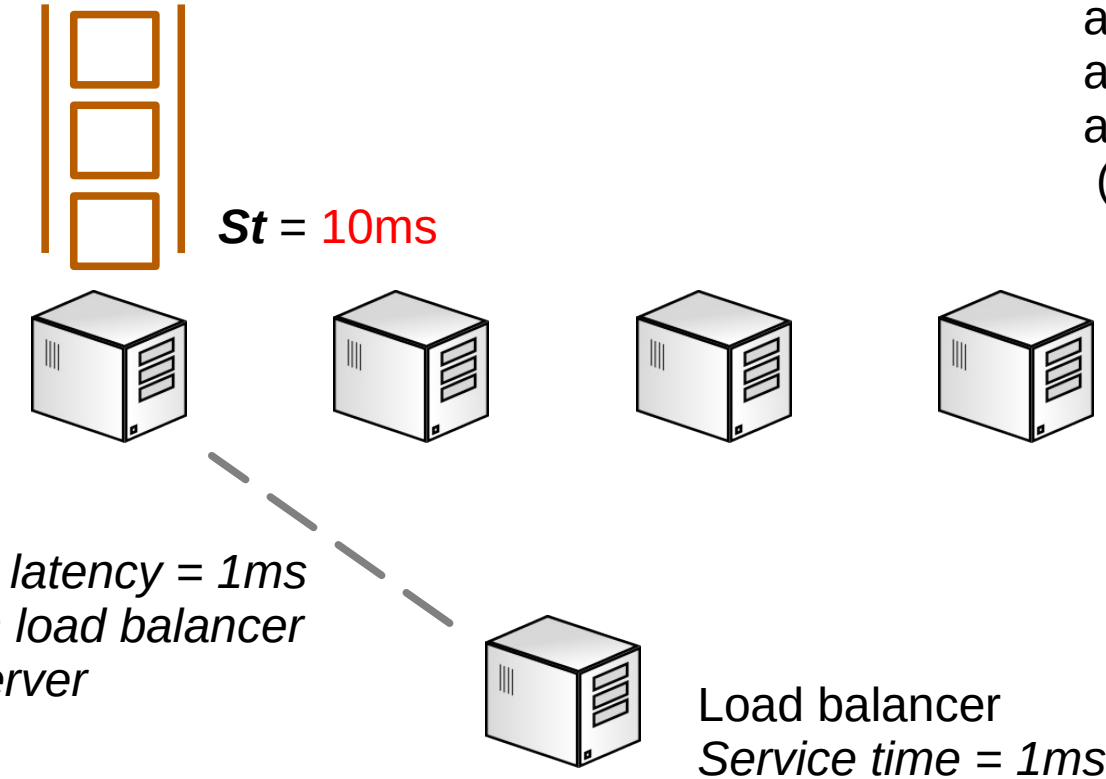
In this scenario, our service response times are *the same* as the load balancer's service time; adding a load balancer here makes this *worse* because we're also adding latency (the server will have finished a client request before the next one could have arrived)

Network latency = 1ms between load balancer and a server

Load balancer
Service time = 1ms

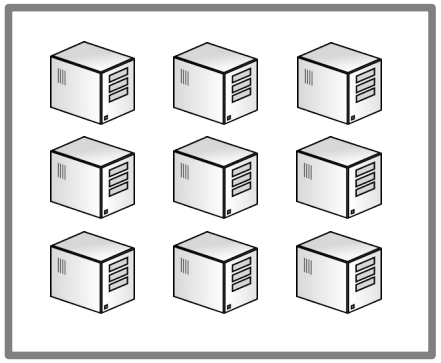
Client requests arriving

When to scale out...in a datacentre?



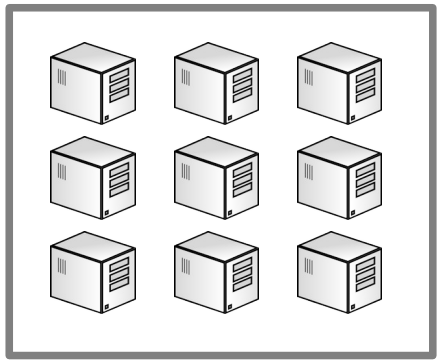
In this scenario, our service response times are *much slower* than the load balancer's time; adding a load balancer here makes this *better* and can distribute load to other machines (each server will enqueue ~10 requests before servicing the first one)

When to scale out...geographically?



datacentre: San Francisco

Geographic distribution creates significantly different latencies between different users; here we might scale out to place services closer to users



Datacentre: Berlin

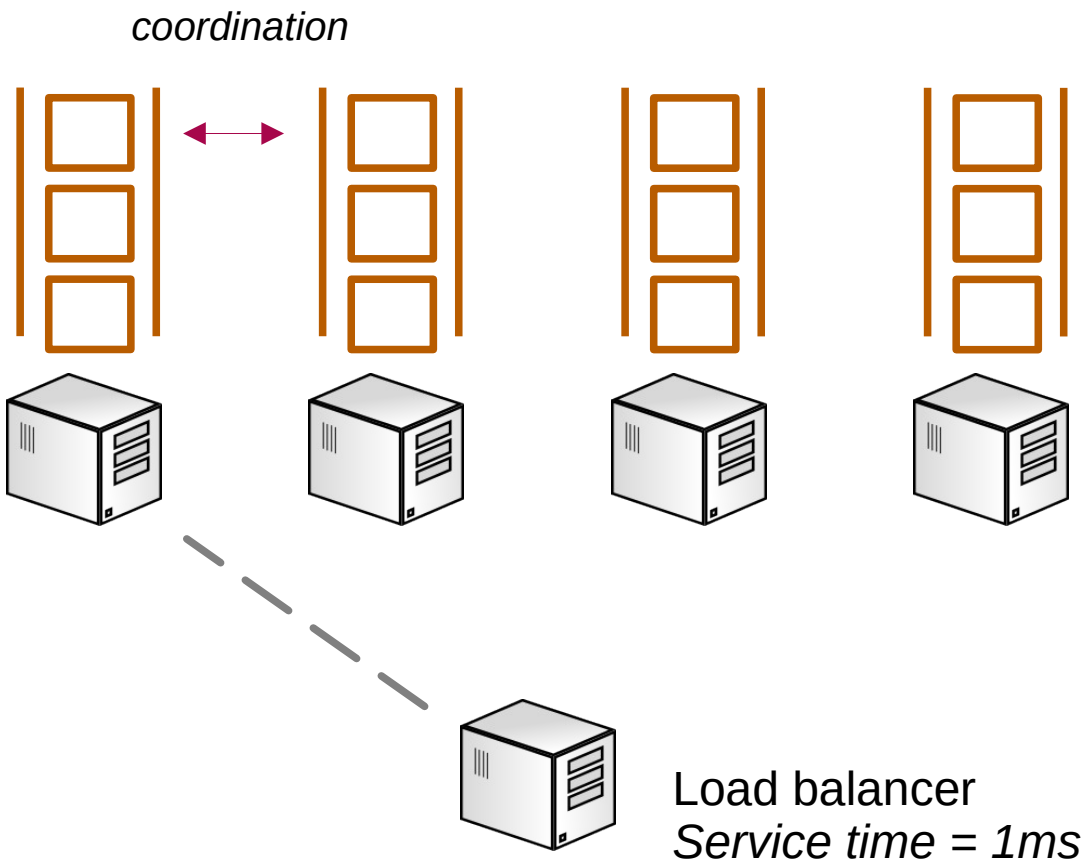
Client: London

Client: NYC

Client: Athens

What about stateful services?

St = 10ms



When services have state, and updates must be coordinated in some way, this can easily remove any benefit of scale-out because all servers work in “lock-step” anyway

Overview for today

- The use of *containment technologies* to encapsulate software is now a near-ubiquitous part of distributed systems engineering
  - It is fundamental to the way in which cloud computing works, allowing us to automatically make *copies* of a whole system to scale out
  - It's also generally a useful way to solve dependency management problems by packaging everything your system needs in one box

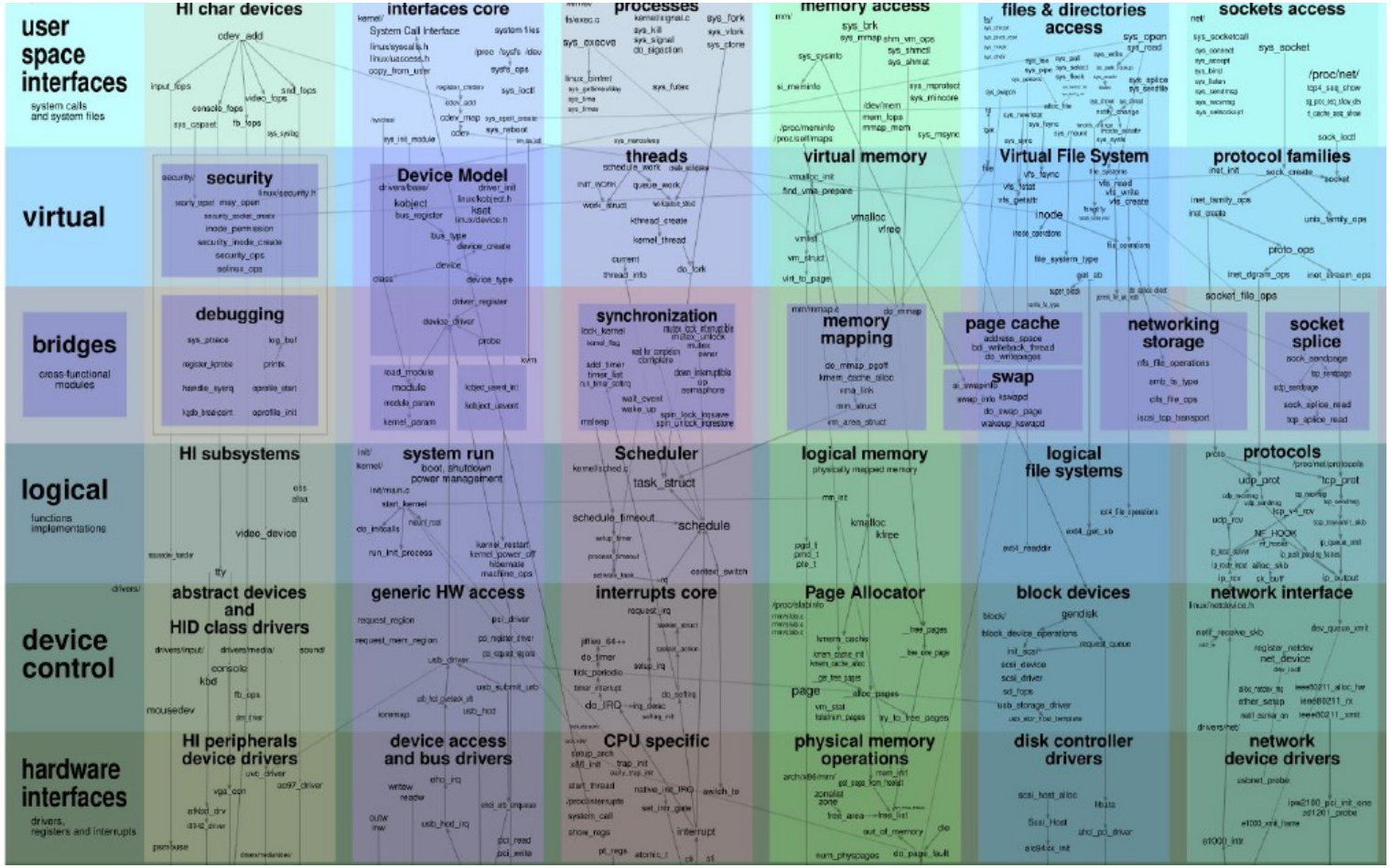
- Today we look at the general concept, and two specific examples using *virtualisation* and *containers*: **VMware** and **Docker**

What is containment?

- Encapsulating all of a software system, and its dependencies, in a single virtual unit which can be moved and copied all together

Why software containment?

- Dependency management



Why software containment?

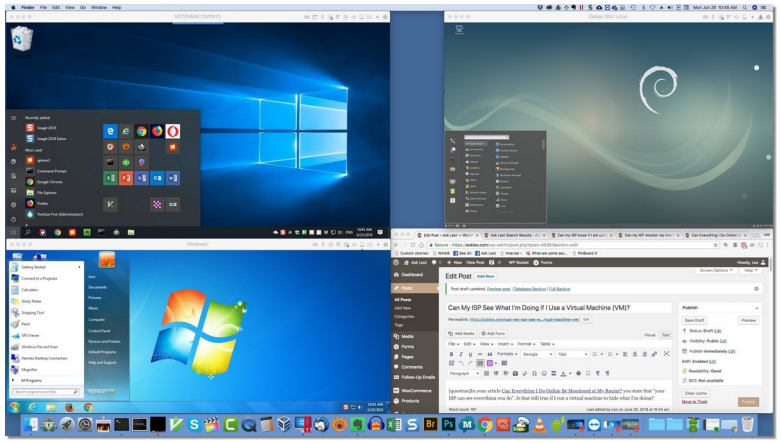
- Cross-platform development

Why software containment?

- Cloud scaling and co-location

Two approaches to this...

Virtualisation (VMware, VirtualBox, Hyper-V)



Containers (Docker, Kubernetes)

Virtualisation with VMware

Virtualisation with VMware

- VMware was the first example of *full-machine virtualisation*, and still occupies a large market share

- Virtualisation is arguably simpler to manage than containers, but is more expensive in resources because you need to run multiple full operating systems

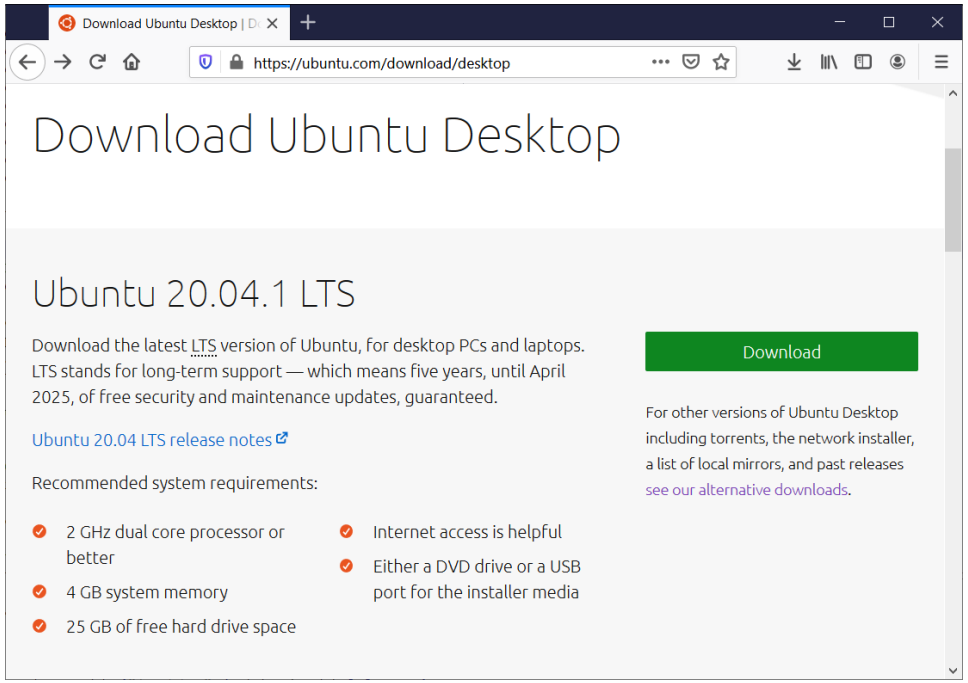
Virtualisation with VMware

- VMware offers “full virtualisation” in which the guest OS (i.e., the OS running in the VM) *is not aware* that it is running on virtual hardware

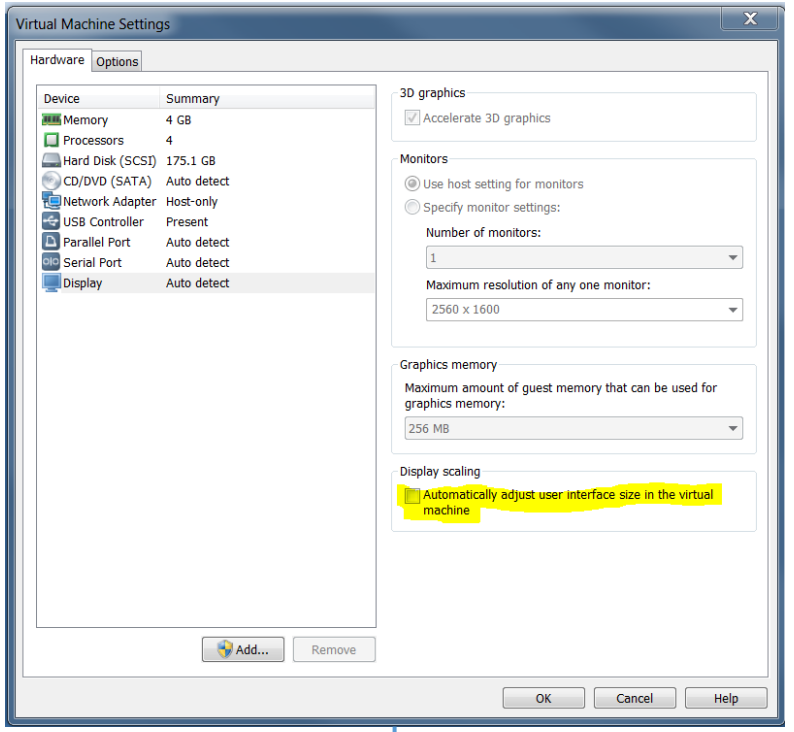
- Remember that an operating system is just a program like any other, so it’s made up of a sequence of CPU instructions (machine code)

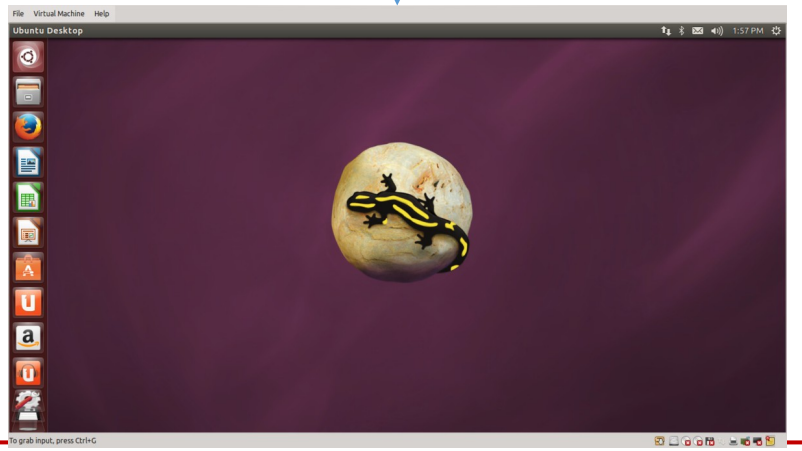
- VMware works by detecting and intercepting particular kinds of CPU instructions while leaving others unmodified to run on the real CPU

Virtualisation with VMware



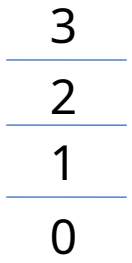
Ubuntu-20.iso





How transparent virtualisation works...

- The x86 architecture has 4 instruction levels which have different permissions to access the computer's hardware



- Most applications always run at level 3 – i.e. they only use the instructions that are in the level 3 instruction set

- An OS assumes that it has access to all 4 levels, so its machine code may consist of instruction types from across all of those levels

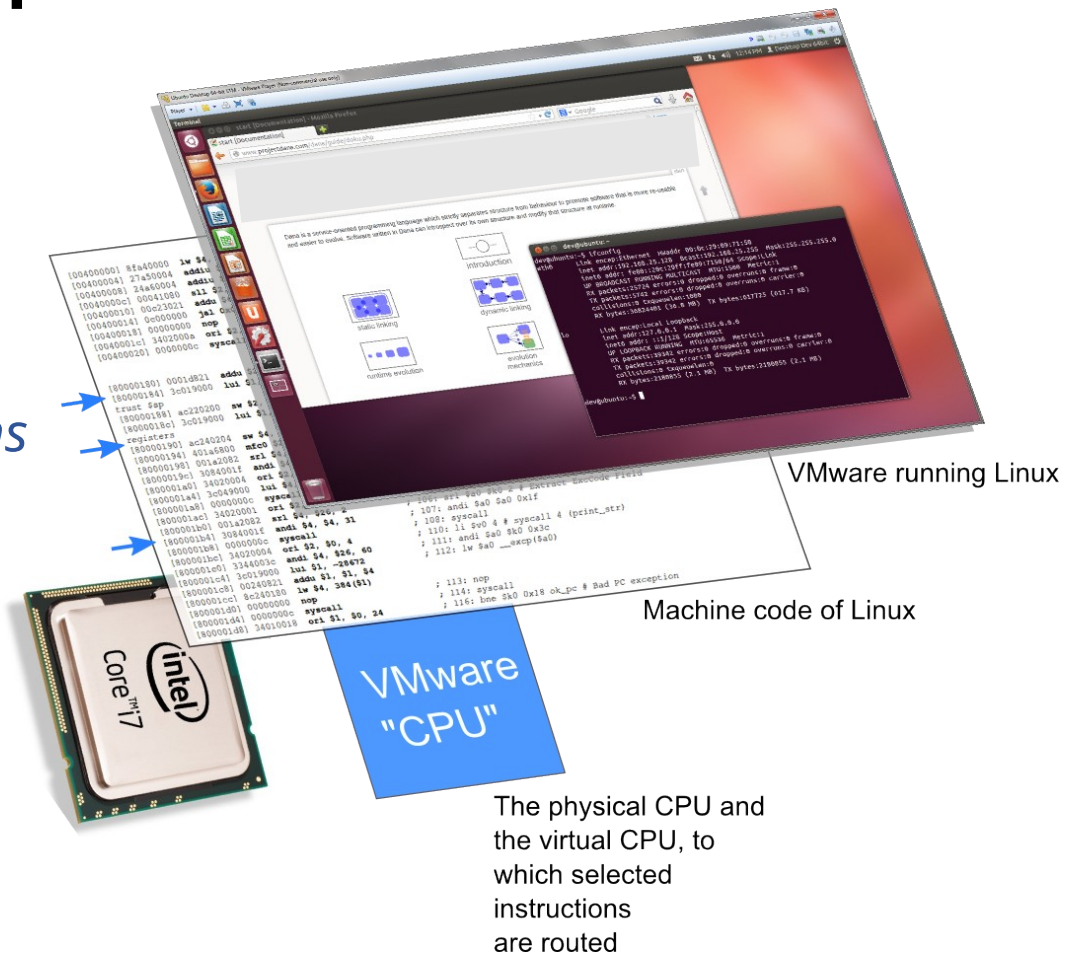
How transparent virtualisation works...

- VMware needs to dynamically trap instructions at levels 0, 1 and 2 and re-write those instructions so that they target virtual representations of the instructions' effects

- Examples of level 0 instructions include certain kinds of unrestricted memory access and hardware interrupt handling
  - Instructions in these levels also cover communication with specific hardware, so by intercepting these instruction we can pretend to the guest OS that the machine has a particular hardware profile

How transparent virtualisation works...

Detect instructions at levels 0,1,2



How transparent virtualisation works...

- The speed of a VM service like VMware generally comes down to how efficiently it can “ignore” as many CPU instructions as possible and let them pass directly to the real CPU

- Modern CPUs like Intel’s i7 processor now include specific instructions which help to further speed up virtualisation

Virtualisation at scale

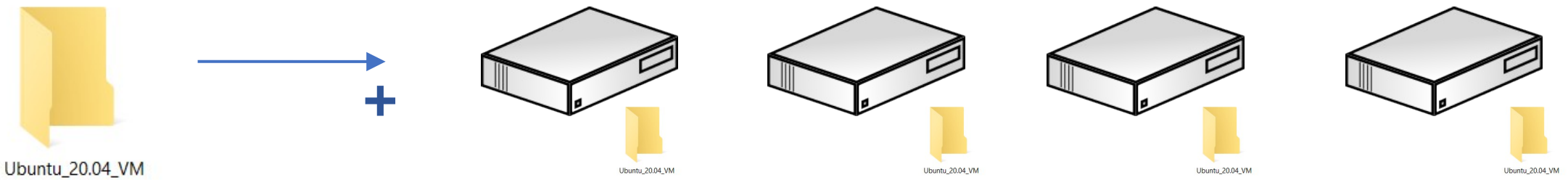
- In practice, a virtual machine is a collection of files on a hard disk (containing both the VM's virtual hard drive files, plus the current state of the VM's RAM when the VM is powered on)

- This means we can *move* the entire virtual machine by copying its host-OS-resident data folder to a different computer

Virtualisation at scale

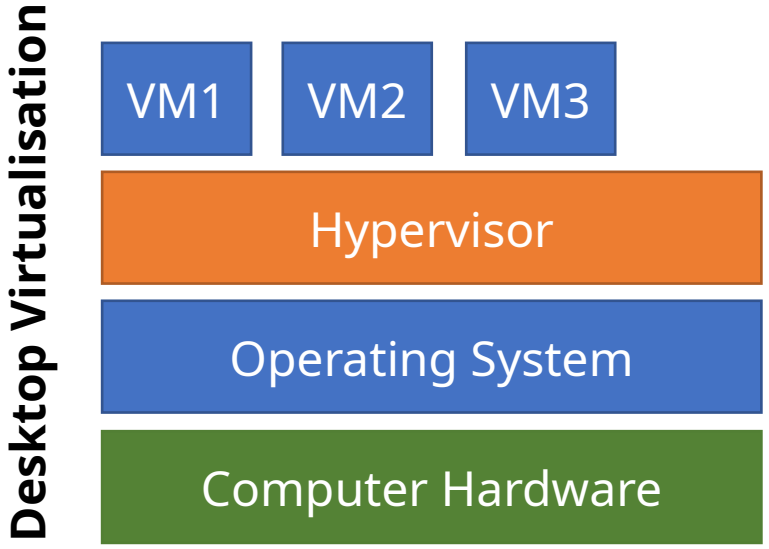
- For a VM image that's powered off, we can also *copy* that VM's folder to lots of computers and turn it on at each one

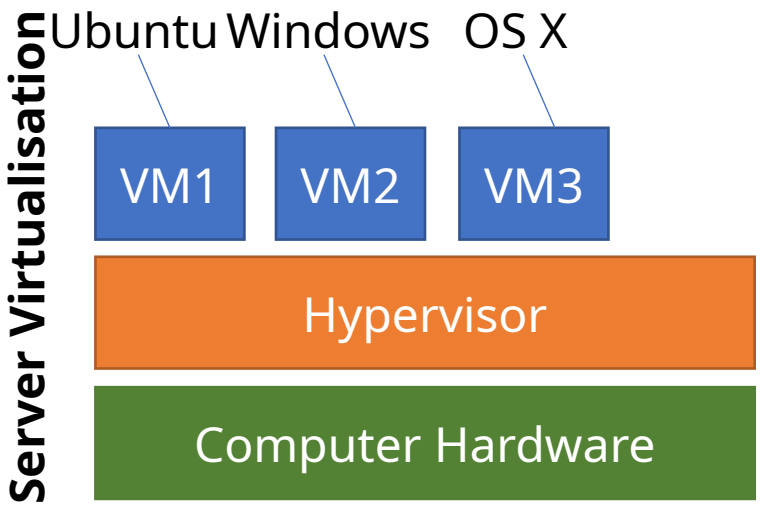
- This is the basis of cloud scaling: we build a VM with our remote service it in (like a Spark node, or web server) and copy it to many servers – each one boots up and gets a unique IP address



Virtualisation at scale

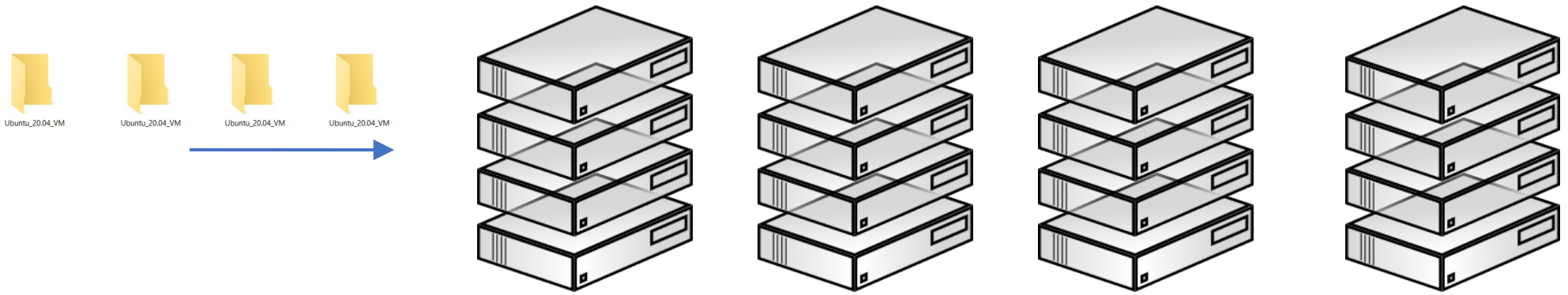
- When we're using lots of VMs in the same datacentre, a special resource management layer called a *hypervisor* sits below them to perform cross-VM resource scheduling & process management





Virtualisation at scale

- In a large data centre like those running Amazon Web Services, many VM hosting requests may arrive every minute
  - With each VM having a different amount of allocated RAM and hard disk capacity, as well as other hardware, the data centre's job is to efficiently decide which VM should be placed on which server – with the aim of using as few servers as possible overall



Virtualisation at scale

- As well as starting new copies of VMs on more servers, we can also *migrate* virtual machines between servers
  - This means we spin up a VM at a new location, briefly "pause" an existing one, and ship all of its RAM pages into the new location
  - This can be useful in latency-sensitive services which need to migrate around the world to be closest to their users ("follow-the-sun" VM clusters)

Containers with Docker

Containers with Docker

- One of the key issues with virtualisation solutions like VMware is that they can be very **expensive** in resources
  - We have an entire copy of an operating system, and all of its services, running many times over on each server

- Containers are an alternative, lightweight software containment approach which aim to achieve the same benefits of VMs without overhead
  - One of the most popular container technologies is **Docker**

Containers with Docker

- Docker uses capabilities of the Linux OS to provide a lightweight sandbox (you can run Docker on Windows, but via a mini Linux VM)

- This requires two main things to operate:
  - A *naming* sandbox to provide a virtualised view of user IDs, filesystems, process trees, and networking services
  - A *resource* sandbox to provide limits on how much RAM and CPU a container will be able to use
  - File storage is managed as a separate service using *volumes* which can be mounted within a Docker image

Containers with Docker

- Resources are encapsulated via the Linux *cgroup* (control group) paradigm

- cgroups can be defined manually on boot, or can be dynamically created in an ad-hoc way

- Each cgroup can have its memory capacity and CPU availability (usually expressed in which cores are available) constraints defined
  - any processes started within that cgroup are then forced by the OS to exist within those constraints

Containers with Docker

- Namespaces are managed via the Linux *unshare* API
  - This allows separate control over private namespaces for networking, processes (i.e., process IDs), user accounts, time, and file system mounting

- Docker uses namespace management + cgroups to create an isolated container for a set of applications and libraries, which internally appears to have its own operating system

Docker in practice

- There are lots of prebuilt Docker images which you can use, for example on Docker Hub

- Docker uses a (text-based) *Dockerfile* to describe an image

- New Docker images are often based on extending an existing image (for example with a particular Java environment already installed) and adding more to it

Docker in practice – Dockerfile example

```
FROM ubuntu:jammy
ENV DEBIAN_FRONTEND=noninteractive
RUN apt update && apt install -y libgles2-mesa && rm -rf /var/lib/apt/lists/*
ADD obm /home/obm
RUN chmod +x /home/obm/myscript.sh
WORKDIR /home/obm/
CMD ./myscript.sh
```


Docker in practice – Dockerfile example

The base image (from DockerHub) that you're starting from (e.g. Ubuntu Server)

Don't ask me to confirm package installs...

Install some extra Linux packages that my service needs

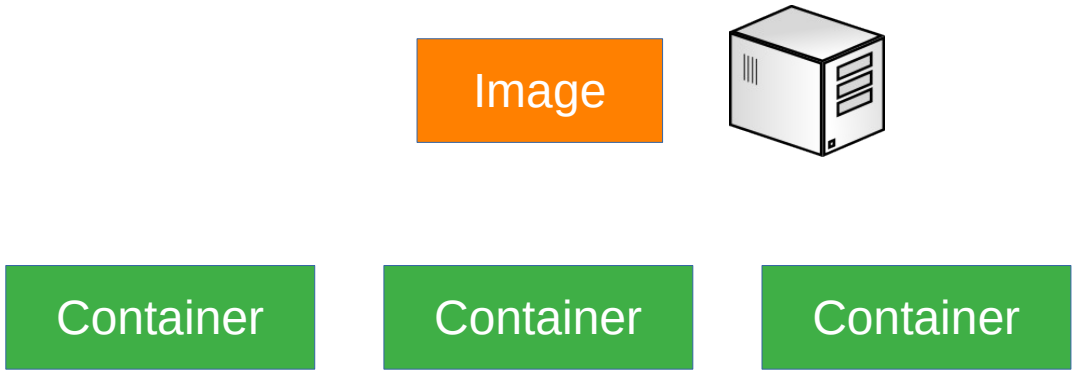
```
FROM ubuntu:jammy
ENV DEBIAN_FRONTEND=noninteractive
RUN apt update && apt install -y libgles2-mesa && rm -rf /var/lib/apt/lists/*
ADD obm /home/obm
RUN chmod +x /home/obm/myscript.sh
WORKDIR /home/obm/
CMD ./myscript.sh
```

Copy a local directory (obm) from my computer to the given path in the Docker image (/home/obm);
- things that are specific / custom-build for my service are going to be here

Run this command when a new container is created from this image. The Docker container will stop and shut down when this command finishes, so this is usually a "blocking" command.

Docker in practice

- Once we have a Docker *image*, which can copy that image to many computers, and start one or more *containers* of that image, each of which will run a copy of our service



Docker in practice

- There are lots of prebuilt Docker images which you can use, for example on Docker Hub

- New Docker images are often based on extending an existing image (for example with a particular Java environment already installed) and adding more to it

- We run an image with e.g.: `docker run -it --rm -d -p 8080:80 nginx`
  - This maps our host port 8080 to the internal container port 80

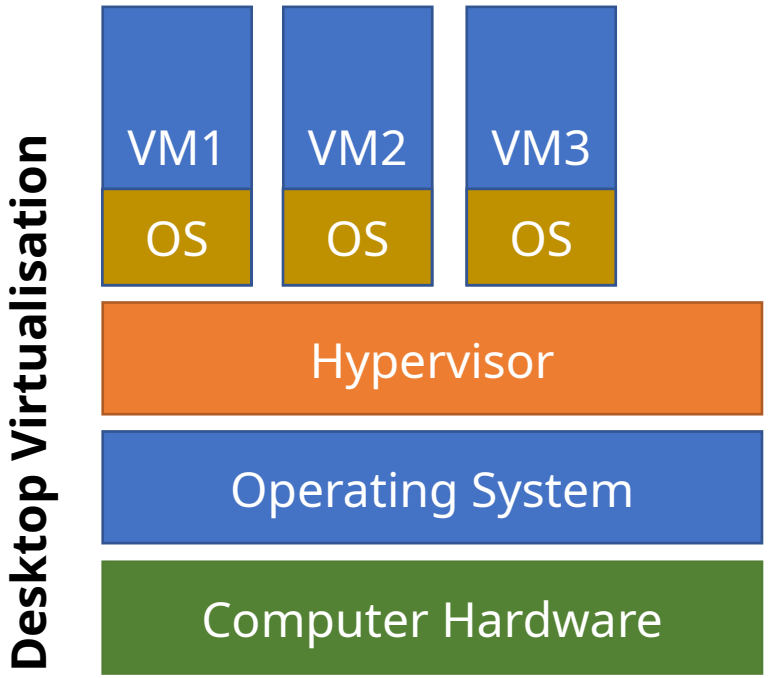
Docker in practice

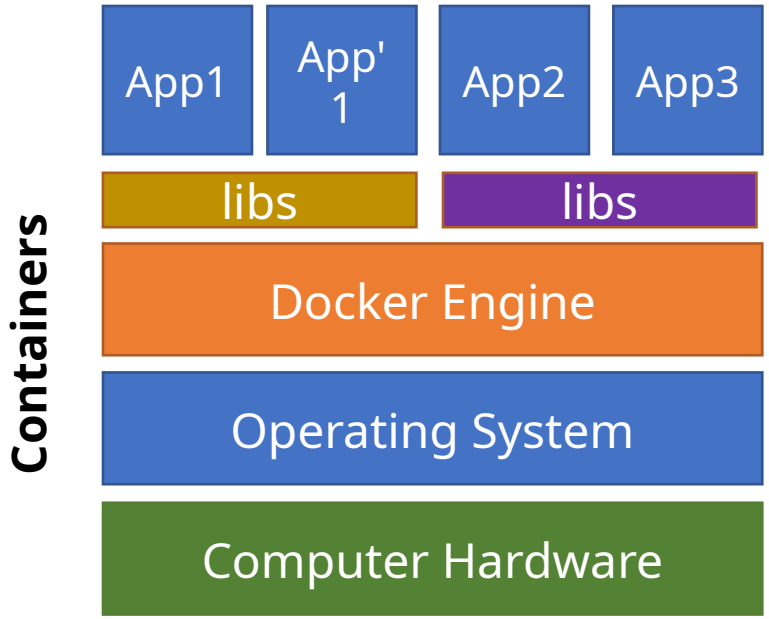
- Just like VMs, we can then copy a docker image many times to many different hosts, and start each one separately

- When the docker container starts up, it launches any specified programs inside the container (like web servers) and maps a host network port onto the listening port in the container

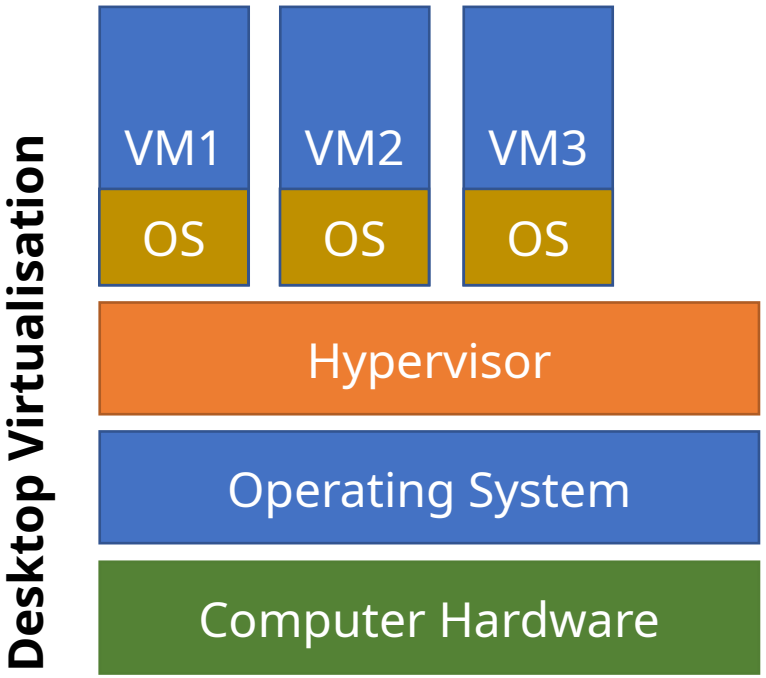
- This again supports the idea of cloud scaling, where stateless services are scaled up by making more copies of them

VMs vs. Containers





VMs vs. Containers



VM Analysis

Good for cross-platform development

Potentially more secure (total isolation)

OS-agnostic for host and guest

Very high resource usage

Multiple operating systems to keep up to date

Compatibility issues between cloud providers (lots of VM architectures – VMware, Xen, Hyper-V, ...)

VMs vs. Containers

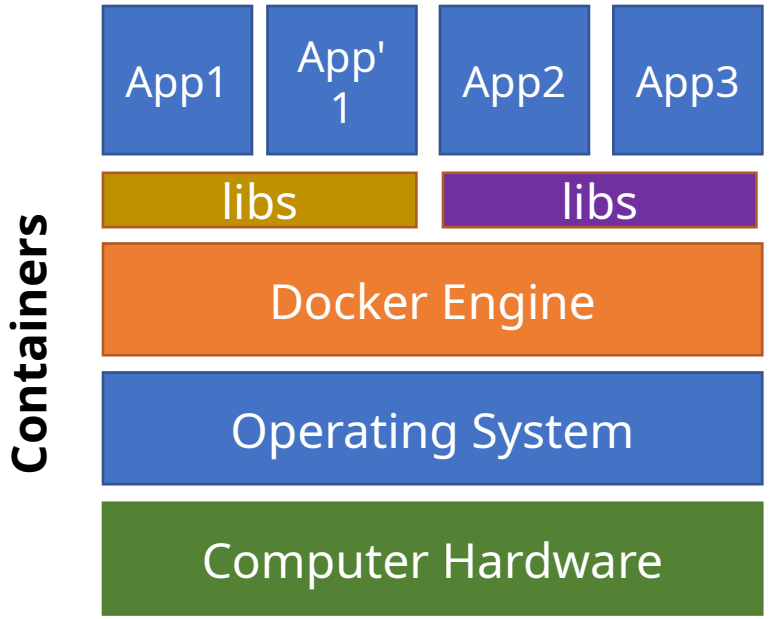
Container Analysis

Very low resource usage (just the libraries)

Compatible on every Linux distribution (works on any cloud provider)

System-level updates are easy to manage (update the host OS once, every container get the updates)

Guest software (inside the container) Linux-based only



Summary

- Software containment can be achieved using virtualisation (e.g. VMware) for containers (e.g. Dockers), with provide fundamentally different approaches

- Containment can help with cross-platform testing, with software distribution for dependency management, and is crucial to enabling cloud scaling

Further reading

- Virtual machine placement challenges:

  "A Survey of Virtual Machine Placement Techniques in a Cloud Data Center", Usmani and Singh, 2016

- Technical docs on Linux container APIs

  https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-i-linux-control-groups-and-process

  https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01