Containers

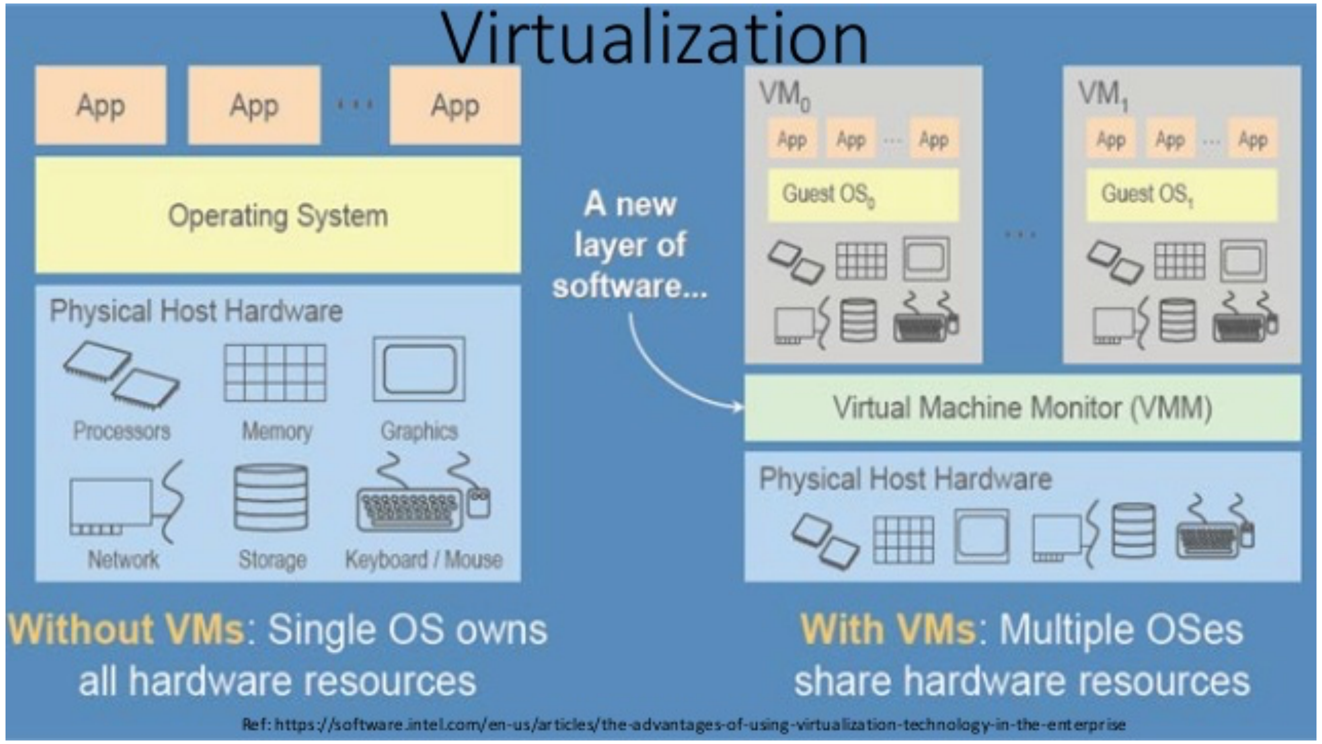
Davide Salomoni (davide@infn.it)

Corso INFN “Istanziazione e utilizzo di batch system on-demand su infrastrutture Cloud. L’esempio pilota dell’esperimento AMS”

Perugia 25-28/11/2019

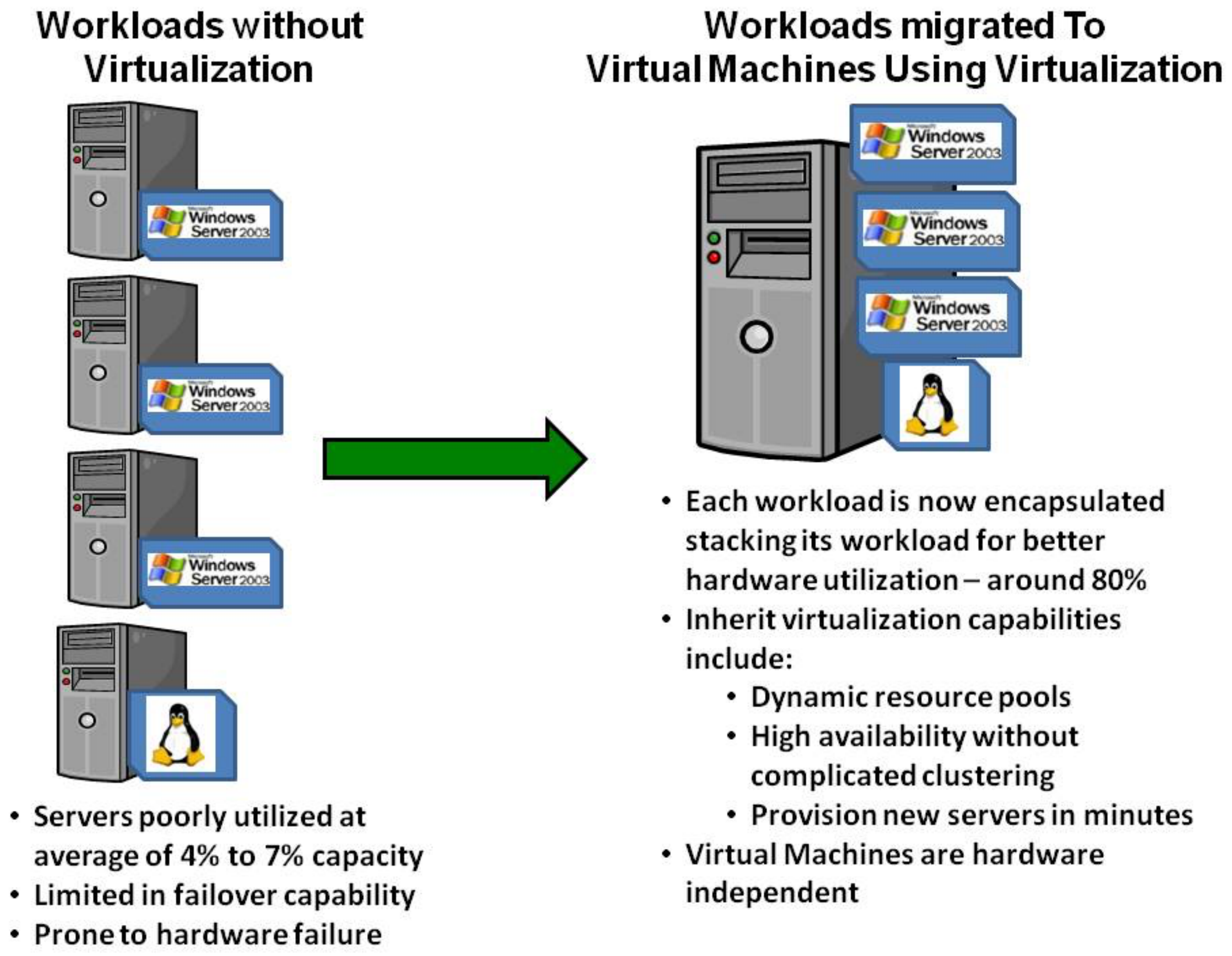
Virtualization

- We have already somewhat talked about "Virtual Machines" (VMs). Informally, a VM is a "virtual copy of a real machine".
- But what is "Virtualization" in general?
  - It is **the creation of a virtual version of *something***: an Operating System, a storage device, a network resource: pretty much almost anything can be made virtual.
  - This is done through an abstraction, that hides and simplifies the details underneath.


Virtualization
App
App
App
Operating System
Physical Host Hardware
Processors
Memory
Graphics
Network
Storage
Keyboard / Mouse
A new layer of software...
VM0
VM1
App
Guest OS0
Guest OS1
Virtual Machine Monitor (VMM)
Physical Host Hardware
Without VMs: Single OS owns all hardware resources
With VMs: Multiple OSes share hardware resources
Ref: https://software.intel.com/en-us/articles/the-advantages-of-using-virtualization-technology-in-the-enterprise

Going virtual

Source: http://bit.ly/2IVk6e5



Workloads without Virtualization

- Servers poorly utilized at average of 4% to 7% capacity
- Limited in failover capability
- Prone to hardware failure

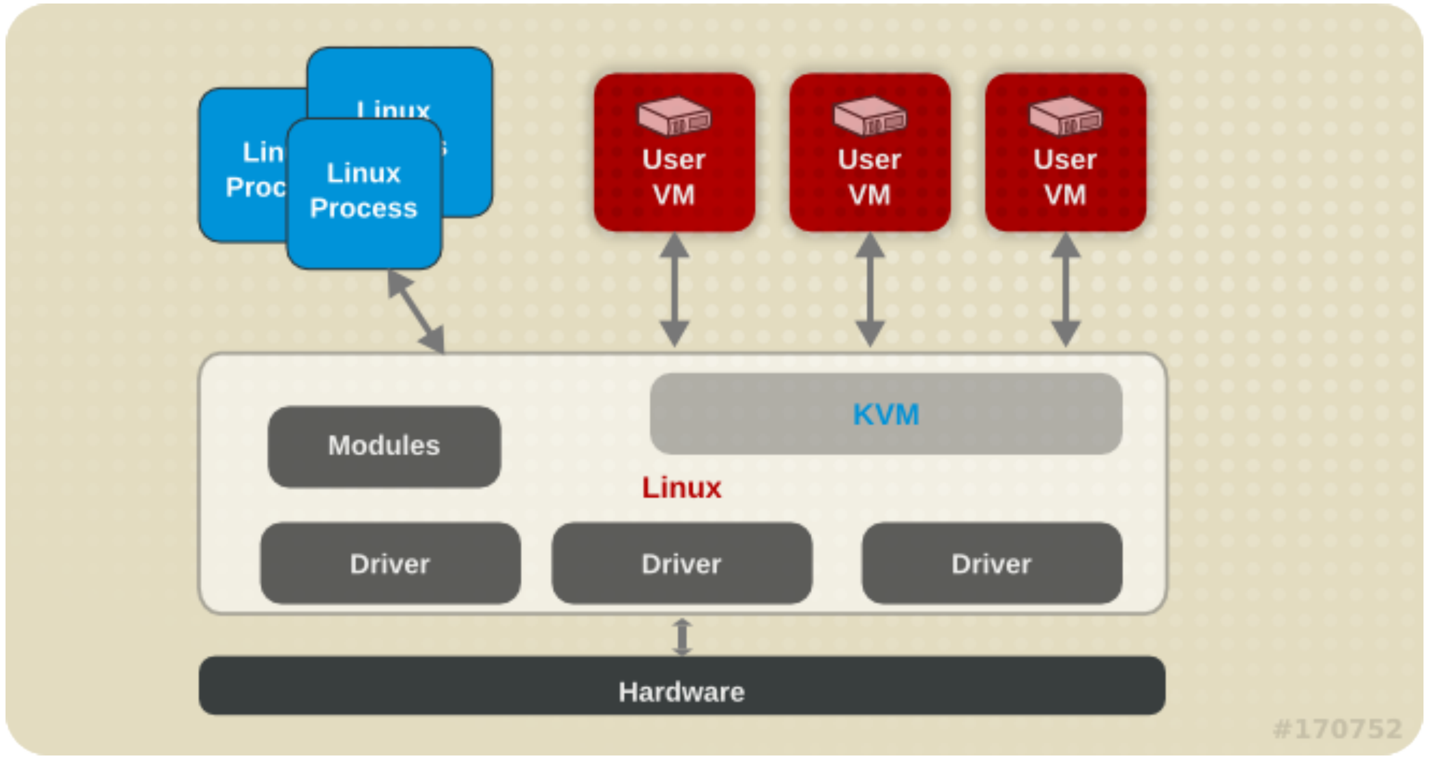
Workloads migrated To Virtual Machines Using Virtualization

- Each workload is now encapsulated stacking its workload for better hardware utilization – around 80%
- Inherit virtualization capabilities include:
  - Dynamic resource pools
  - High availability without complicated clustering
  - Provision new servers in minutes
- Virtual Machines are hardware independent

Virtualization with Linux KVM

- KVM is a kernel module for virtualization in Linux – (there are other ways to handle virtualization in Linux that we won’t discuss).

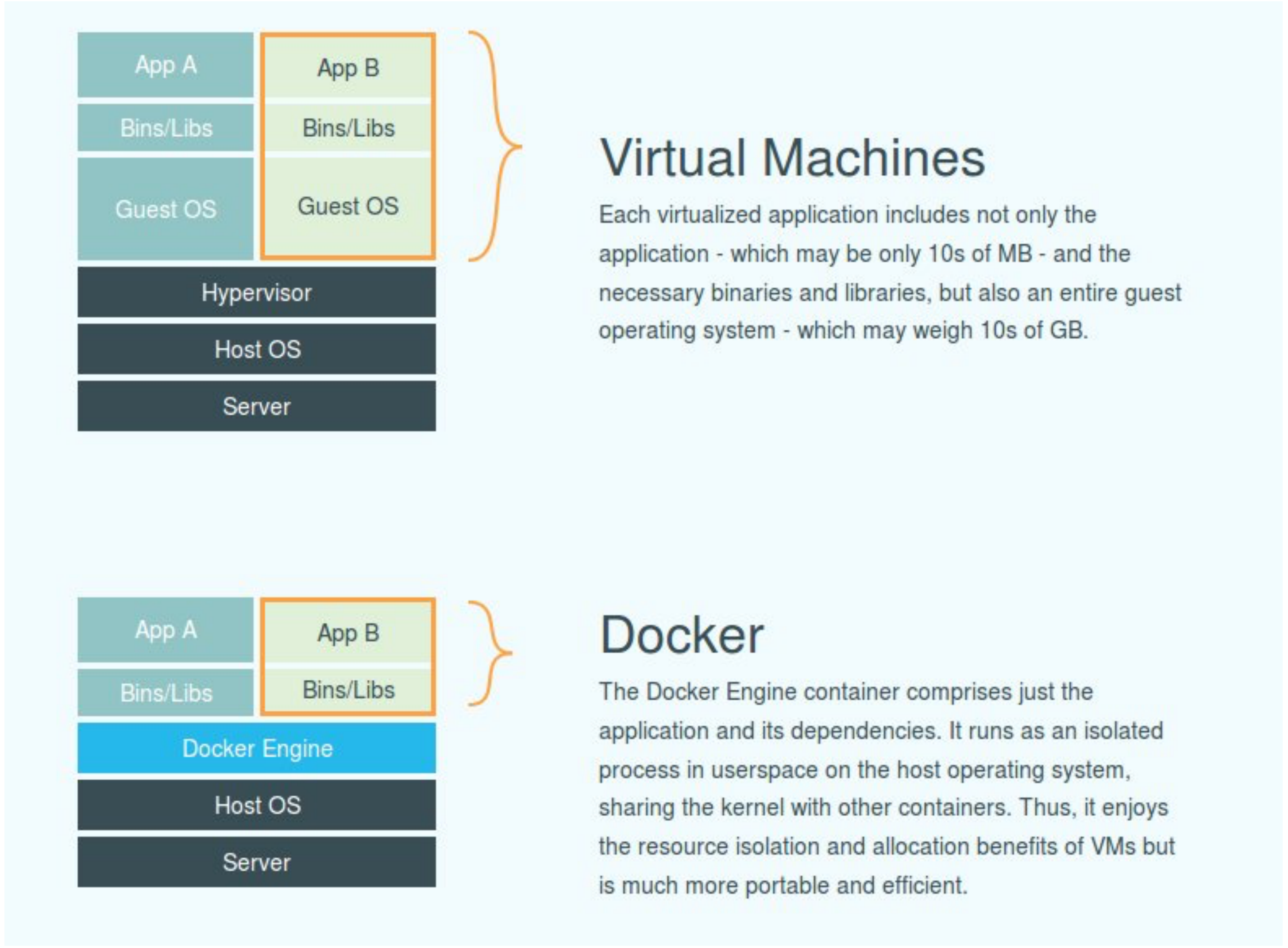
Source: https://red.ht/2IUxJdr



Going beyond Virtual Machines

- Virtual Machines (VMs) carry quite some overhead with them → introducing **Docker Containers**

Source: http://bit.ly/2IVk6e5



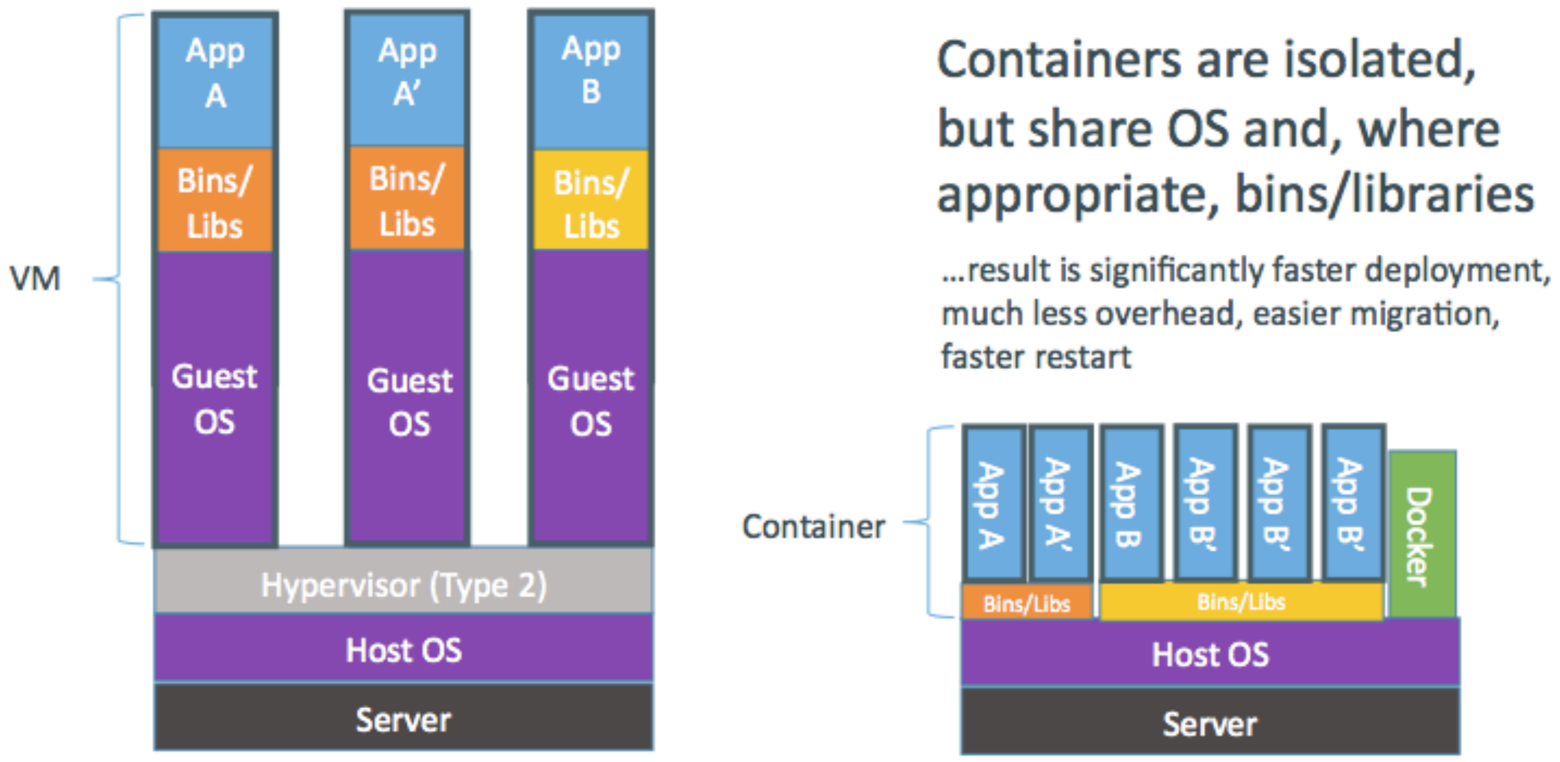
Virtual Machines

Each virtualized application includes not only the application - which may be only 10s of MB - and the necessary binaries and libraries, but also an entire guest operating system - which may weigh 10s of GB.

Docker

The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.

Containers are «lightweight VMs»



Containers are isolated, but share OS and, where appropriate, bins/libraries

...result is significantly faster deployment, much less overhead, easier migration, faster restart

Source: http://goo.gl/4jh8cX

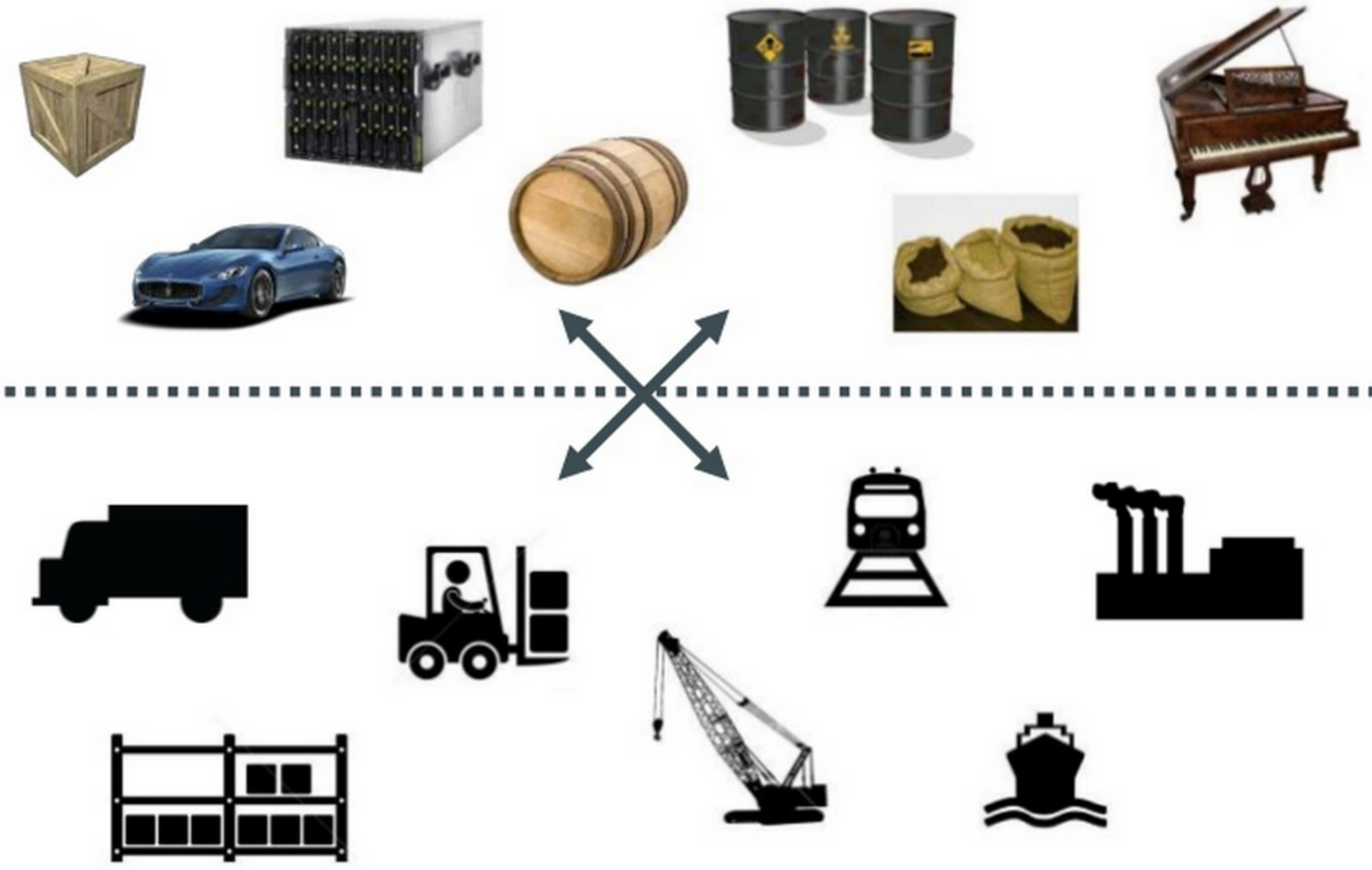
“Lightweight”, in practice

- **Containers require less resources**: they start faster and run faster than VMs, and you can fit many more containers in a given hardware than VMs.
- **Very important**: they provide enormous simplifications to software development and deployment processes, because they allow to easily **encapsulate applications** in a controlled and extensible way.

Cargo Transport Pre-1960

Multiplicity of Goods

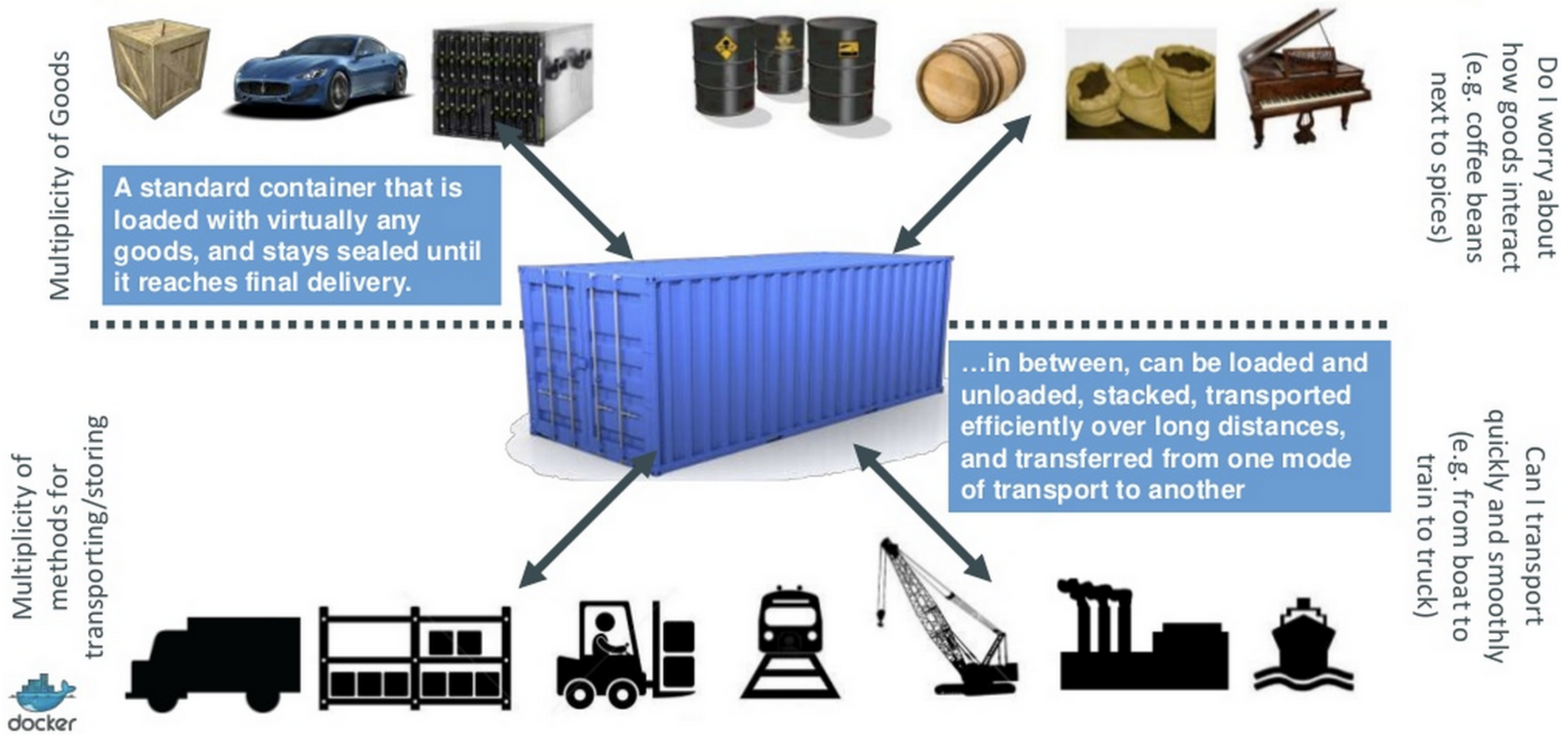
Do I worry about how goods interact (e.g. coffee beans next to spices)



Multipilicity of methods for transporting/storing

Can I transport quickly and smoothly (e.g. from boat to train to truck)

Solution: Intermodal Shipping Container



Multiplicity of Goods

A standard container that is loaded with virtually any goods, and stays sealed until it reaches final delivery.

Do I worry about how goods interact (e.g. coffee beans next to spices)

…in between, can be loaded and unloaded, stacked, transported efficiently over long distances, and transferred from one mode of transport to another

Multiplicity of methods for transporting/storing

Can I transport quickly and smoothly (e.g. from boat to train to truck)

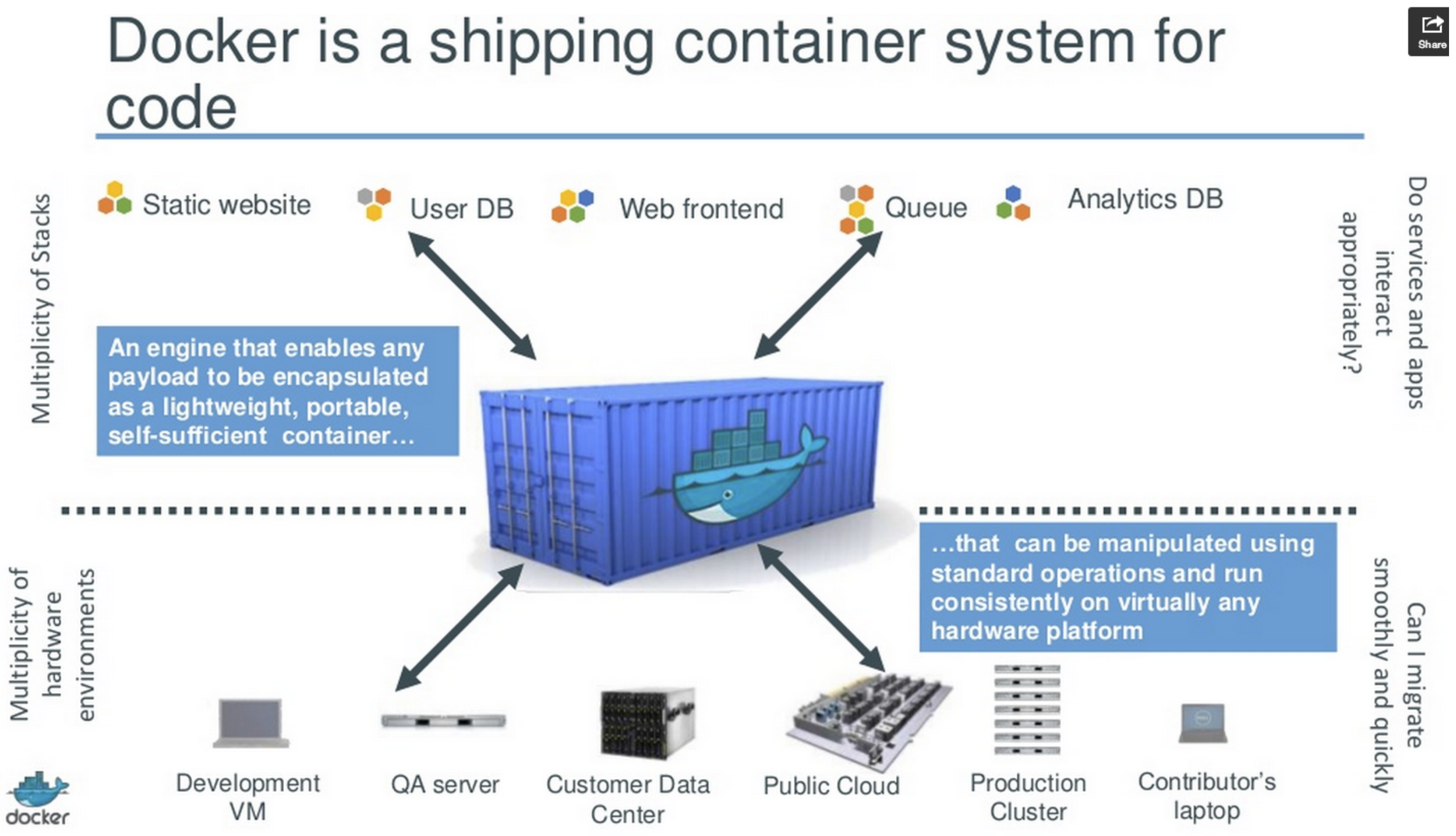
Intermodal Shipping Container Ecosystem

- 90% of all cargo now shipped in a standard container
- Order of magnitude reduction in cost and time to load and unload ships
- Massive reduction in losses due to theft or damage
- Huge reduction in freight cost as percent of final goods (from >25% to <3%)

→ massive globalizations

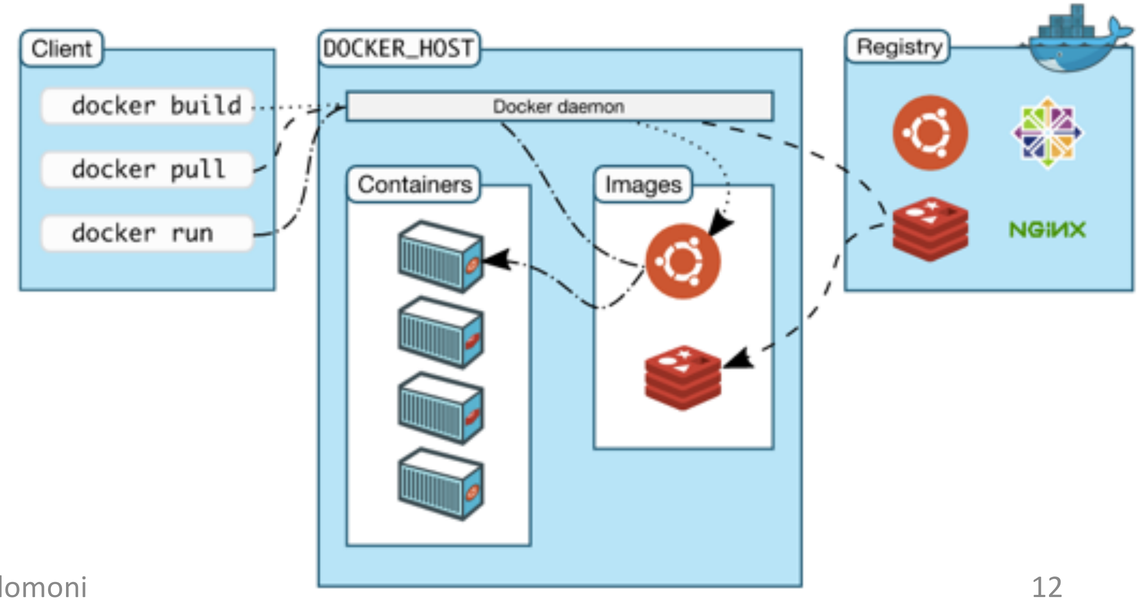
- 5000 ships deliver 200M containers per year

Docker Containers


Docker is a shipping container system for code
Static website
User DB
Web frontend
Queue
Analytics DB
Multiplicity of Stacks
Do services and apps interact appropriately?
An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…
…that can be manipulated using standard operations and run consistently on virtually any hardware platform
Multiplicity of hardware environments
Can I migrate smoothly and quickly
Development VM
QA server
Customer Data Center
Public Cloud
Production Cluster
Contributor's laptop
docker
Share

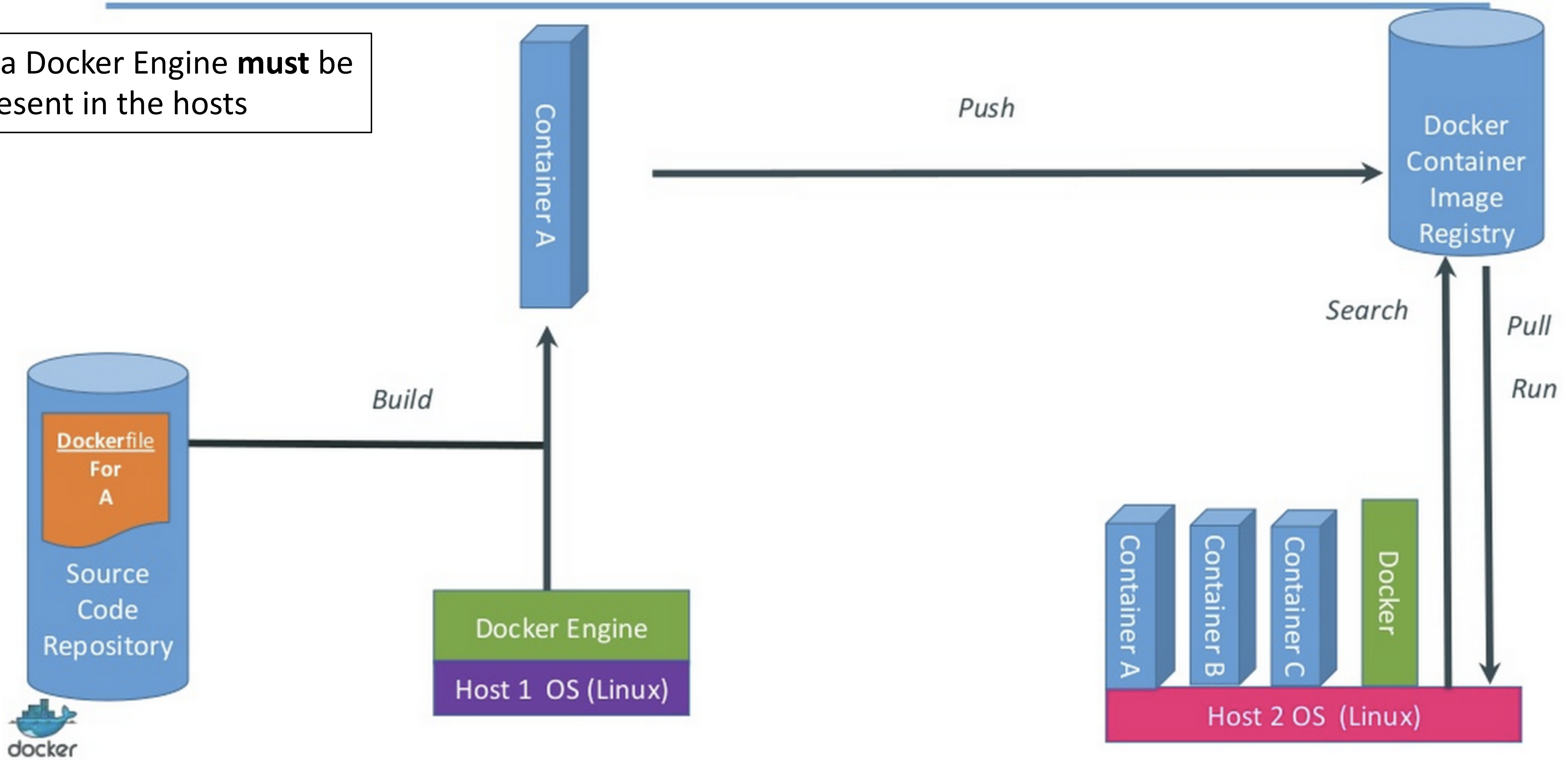
Docker features

- Docker is an open source engine for the **easy creation of lightweight, portable, self-sufficient containers from any application**.
- The **same container image** that a developer builds and tests on a laptop can run at scale, in production, on VMs, private, public clouds and more.
- Main features:
    - versioning (git-like)
    - component re-use
    - sharing (e.g. through public repositories)



What are the basics of the Docker system?

Note that a Docker Engine **must** be present in the hosts



The test infrastructure for this course

- Each of you has been allocated a VM (thanks to Mirko Mariotti), with CentOS 7, a public IP address, 4GB RAM, a 20GB disk and 2 Virtual CPU.
- You should have already received your credentials to access your VM – if not, let us know now.
- **You should now log on to your VM**. We will use it for the hands-on on containers and in other lectures during this course.

Hands-on: the Docker *daemon*

- Recall that a **computer daemon** is a program that runs as a **background process** (and not, for example, a program that is run by an interactive user).
- In order to use Docker on a machine, that machine must have the Docker engine (or the Docker daemon) installed and running. **Do not assume** that, by default, this daemon is already installed.
  - Is it installed on your VM? How can you check?
- If it is not installed, install it yourself:

```
ubuntu@VM1:~$ sudo yum install docker-ce docker-ce-cli containerd.io
```

- Check that Docker is properly installed now, with this command:

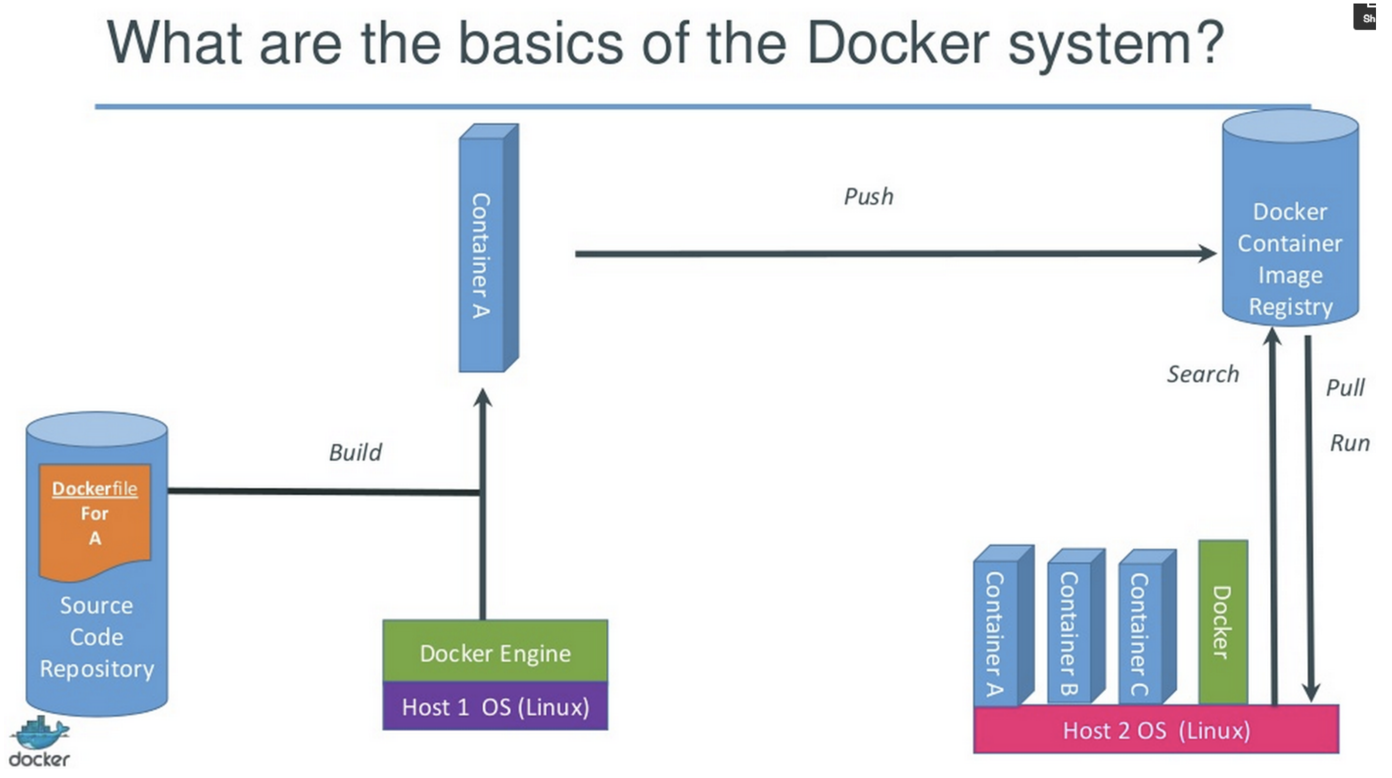
```
ubuntu@VM1:~$ docker --version
```

  It should return something like this:

```
Docker version 18.09.5, build e8ff056
```

The first docker commands

- By default, the “container image registry” on the left is the service running at https://hub.docker.com (called “**Docker Hub**”). It stores more than 100,000 container images.
- To pull a container image from Docker Hub, use the command `docker pull`.
- To run (execute) a container, use the command `docker run`.



Search, pull, run and push

- **Try these commands on your VM**:
  - **Search** for a container image at Docker Hub:
    - `docker search ubuntu` (or e.g. `docker search rhel` – what would this do?)
  - Fetch (**pull**) a Docker image (in this case, an Ubuntu container):
    - `docker pull ubuntu`
  - Execute (**run**) a docker container:
    - Run the "echo" command inside a container and then exit:

      -
        ```
        docker run ubuntu echo "hello from the container"
        hello from the container
        ```
    - Run a container in interactive mode:
      - `docker run -i –t ubuntu /bin/bash`
  - Ship (**push**) a Docker image to a Docker repository (by default, Docker Hub) – <u>skip these commands for the time being</u>, we'll say more about this later:
    - `docker login`
    - `docker push USER/my-image`

Note that you are inside a Ubuntu container, while the "host system" (your VM) runs CentOS

How efficient is docker?

```
ubuntu@VM1:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu              latest              7698f282e524        7 days ago          64.2MB
```

→ the *latest* Ubuntu image takes less than 70MB of disk space *as a container*. If you had downloaded a full Ubuntu (server) distribution, it would be more in the range of 900MB.

```
ubuntu@VM1:~$ time docker run ubuntu echo "hello from the container"
hello from the container

real 0m1.362s
user 0m0.036s
sys 0m0.050s
```

→ The total time it takes on this system (not really a powerful one) to start a container, execute a command inside it and exit from the container is a bit more than a second. How long would it take if we used a full VM instead?

How to extend a docker container (1)

- Suppose you need to execute a command or app inside a container, but it is not installed in the image you pulled from Docker Hub. For example, you would like to use the `ping` command but by default it's not available:

  -
    ```
    ubuntu@VM1:~$ docker run ubuntu ping www.google.com
    docker: Error response from daemon: OCI runtime create failed:
    container_linux.go:345: starting container process caused "exec: \"ping\":
    executable file not found in $PATH": unknown.
    ```
- We can install it ourselves; it is in the package `iputils-ping`:

  -
    ```
    ubuntu@VM1:~$ docker run ubuntu /bin/bash -c "apt update; apt -y install
    iputils-ping"
    ```
- But it still doesn't work!?

  -
    ```
    ubuntu@VM1:~$ docker run ubuntu ping www.google.com
    docker: Error response from daemon: OCI runtime create failed:
    container_linux.go:345: starting container process caused "exec: \"ping\":
    executable file not found in $PATH": unknown.
    ```
- Who can explain this? The ping command was successfully installed!

How to extend a docker container (2)

- Whenever you issue a `docker run <image>` command, a **new container** is started, based on the original container image.
  - Check it yourself with the `docker ps -a` command.
- If you modify a container and then want to reuse it (which is often the case!), **you need to save the container, creating a new image**.
- So, install what you need to install (e.g. the `iputils-ping` package, using the same command as before) , and then issue a commit command like

```
docker commit xxxx ubuntu_with_ping
```

- This locally **commits** a container, creating an image with the name `ubuntu_with_ping` (or any other name you like). Take `xxxx` from the container ID shown by the `docker ps -a` output.
- **Do it now**.

How to extend a docker container (3)

- Verify that the `ping` command inside our new image is now working:

  -
    ```
    ubuntu@VM1:~$ docker run ubuntu_with_ping ping -c 3 www.google.com
    PING www.google.com (216.58.216.100) 56(84) bytes of data.
    64 bytes from ord30s22-in-f100.1e100.net (216.58.216.100): icmp_seq=1 ttl=43 time=18.5 ms
    64 bytes from ord30s22-in-f100.1e100.net (216.58.216.100): icmp_seq=2 ttl=43 time=18.5 ms
    64 bytes from ord30s22-in-f100.1e100.net (216.58.216.100): icmp_seq=3 ttl=43 time=18.5 ms

    --- www.google.com ping statistics ---
    3 packets transmitted, 3 received, 0% packet loss, time 2003ms
    rtt min/avg/max/mdev = 18.501/18.539/18.586/0.035 ms
    ```
- **To recap**: we have an original image (called "ubuntu"), downloaded from Docker Hub, and a new image (called "ubuntu_with_ping"), created by us extending the "ubuntu" image (i.e. installing some packages). Let's check:

  -
    ```
    ubuntu@VM1:~$ docker images
    REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
    ubuntu_with_ping    latest              3e7a8818665f        11 minutes ago      97.2MB
    ubuntu              latest              7698f282e524        7 days ago          69.9MB
    ```

Cleaning up container space

- When you don't need some containers anymore, it's wise to check and **clean up disk space**. This is done with the `docker system` commands.
- Check disk space used by containers with `docker system df`:

  -
    ```
    ubuntu@VM1:~$ docker system df
    TYPE                TOTAL               ACTIVE              SIZE                RECLAIMABLE
    Images              2                   2                   97.22MB             69.86MB (71%)
    Containers          4                   0                   27.36MB             27.36MB (100%)
    Local Volumes       0                   0                   0B                  0B
    Build Cache         0                   0                   0B                  0B
    ```
- Reclaim disk space with `docker system prune`, then check again:

  -
    ```
    ubuntu@VM1:~$ docker system df
    TYPE                TOTAL               ACTIVE              SIZE                RECLAIMABLE
    Images              2                   0                   97.22MB             97.22MB (100%)
    Containers          0                   0                   0B                  0B
    Local Volumes       0                   0                   0B                  0B
    Build Cache         0                   0                   0B                  0B
    ```

Removing unused images

- Besides containers, you can also **remove images you don’t need anymore** with `docker rmi <image>`:

```
ubuntu@VM1:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
ubuntu_with_ping    latest              3e7a8818665f        29 minutes ago      97.2MB
ubuntu              latest              7698f282e524        7 days ago          69.9MB

ubuntu@VM1:~$ docker rmi ubuntu_with_ping
Untagged: ubuntu_with_ping:latest
Deleted: sha256:3e7a8818665fc7eb1be20e8d633431ad8c0bdfba05d6d11d40edd32a915708bb
Deleted: sha256:a4c24b3590e4e95c30d4d0e82d3f769cde94436a5dd473b4e7ec7bd4682ce1b7

ubuntu@VM1:~$ docker rmi ubuntu
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:f08638ec7ddc90065187e7eabdfac3c96e5ff0f6b2f1762cf31a4f49b53000a5
Deleted: sha256:7698f282e5242af2b9d2291458d4e425c75b25b0008c1e058d66b717b4c06fa9
Deleted: sha256:027b23fdf3957673017df55aa29d754121aee8a7ed5cc2898856f898e9220d2c
Deleted: sha256:0dfbdc7dee936a74958b05bc62776d5310abb129cfde4302b7bcdf0392561496
Deleted: sha256:02571d034293cb241c078d7ecbf7a84b83a5df2508f11a91de26ec38eb6122f1

ubuntu@VM1:~$ docker system df
TYPE                TOTAL               ACTIVE              SIZE                RECLAIMABLE
Images              0                   0                   0B                  0B
Containers          0                   0                   0B                  0B
Local Volumes       0                   0                   0B                  0B
Build Cache         0                   0                   0B                  0B
```

Pushing images to Docker Hub (1)

- We have already seen the command `docker push <image>`. This writes an image **to Docker Hub**.
- In order to issue that command, you first need an account on Docker Hub: go to https://hub.docker.com and sign up (or sign in, if you already have an account there) – it's free.
- **Do it now.**
- Click on Create Repository, make it public (careful: everybody will be be able to see the images you upload there!) and give it a name, for example *test* (only lowercase is allowed), a description, and click on "Create". This will create your public repository, called e.g. "test".

Pushing images to Docker Hub (2)

- To push an image (for example the `ubuntu_with_ping` image we created earlier) to your new repository, we **must** give a **tag** to the image *and* specify **our Docker Hub username and repository** as part of the image name.
  - The full image name should be **<username>/<repository>:<tag>**.
  - In my case, the first part should be "dsalomoni/test". As tag, you can put any string; let's set it to "ubuntu_with_ping_1.0".
  - In order to assign this tag to our existing image, find out its "image id" with the `docker images` command:

Images before the new tag

```
ubuntu@VM1:~$ docker images
REPOSITORY          TAG                   IMAGE ID            CREATED             SIZE
ubuntu_with_ping    latest                7c45b9ad4de6        45 minutes ago      97.2MB
ubuntu              latest                7698f282e524        7 days ago          69.9MB
```

```
ubuntu@VM1:~$ docker tag 7c45b9ad4de6 dsalomoni/test:ubuntu_with_ping_1.0
```

```
ubuntu@VM1:~$ docker images
REPOSITORY          TAG                     IMAGE ID            CREATED             SIZE
ubuntu_with_ping    latest                  7c45b9ad4de6        About an hour ago   97.2MB
dsalomoni/test      ubuntu_with_ping_1.0    7c45b9ad4de6        About an hour ago   97.2MB
ubuntu              latest                  7698f282e524        7 days ago          69.9MB
```

Images after the new tag

Pushing images to Docker Hub (3)

- Now **log in to Docker Hub** with your username and password:

  -
    ```
    ubuntu@VM1:~$ docker login
    Login with your Docker ID to push and pull images from Docker
    Hub. If you don't have a Docker ID, head over to
    https://hub.docker.com to create one.
    Username: dsalomoni
    Password:
    WARNING! Your password will be stored unencrypted in
    /home/ubuntu/.docker/config.json.
    Configure a credential helper to remove this warning. See
    https://docs.docker.com/engine/reference/commandline/login/#cre
    dentials-store

    Login Succeeded
    ```

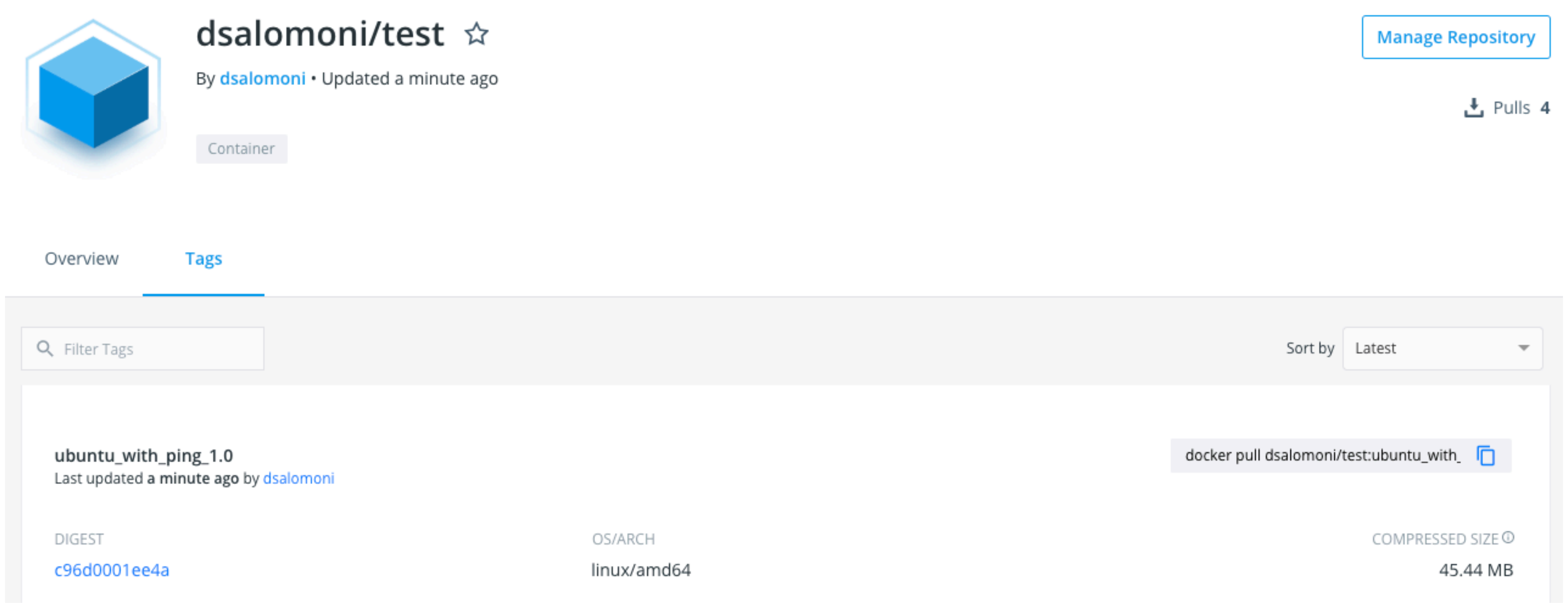
We'll disregard this warning here. For more info, see the URL in the message.

- Finally, we can **push our image to Docker Hub**:

  -
    ```
    ubuntu@VM1:~$ docker push dsalomoni/test:ubuntu_with_ping_1.0
    ```

Verifying our Docker Hub repository

- Go to Docker Hub (https://hub.docker.com/), log in with your username, click on the "test" repository, and then on "Public View" and "Tags". You should see something like this:



dsalomoni/test

By dsalomoni • Updated a minute ago

Container

Pulls 4

ubuntu_with_ping_1.0
Last updated a minute ago by dsalomoni

```
docker pull dsalomoni/test:ubuntu_with_
```

DIGEST	OS/ARCH	COMPRESSED SIZE
c96d0001ee4a	linux/amd64	45.44 MB

Handling multiple commands

- If you have **several commands to apply to a container** (for example, you want to install many packages), you could run the container in interactive mode as shown earlier (use the "-i" switch), and then issue the various commands at the prompt once you are in the container.
  - For example, when you are running a container interactively, you could issue a sequence of commands such as

```
# apt update
# apt install -y wget unzip
# wget <some_file>
# unzip <some_other file>
…
```

- Once you exit from the container, remember to **commit** the container, or your modifications to the container will be lost (like in our "ping" example earlier).

Dockerfiles

- Rather than modifying a container “by hand”, connecting interactively and then installing packages as previously shown, it is often much more convenient to **put all the required commands in a text file (called by default <u>Dockerfile</u>)**, and then **build** an image executing these commands.
- As an example, through the following `Dockerfile` we create a new image *starting from an Ubuntu image*. We then install a web server (through the `apache2` package) and tell the new image to serve a simple html page (`index.html`), which we copy from our system:

```
$ cat Dockerfile
FROM ubuntu
RUN apt update
RUN apt install -y apache2
COPY index.html /var/www/html/
EXPOSE 80
CMD ["apachectl", "-D", "FOREGROUND"]
```

This Dockerfile:
- Starts from the Ubuntu container
- Updates all installed packages
- Installs the apache2 web server
- Copies an index.html file from our system
- Exposes port 80 (the standard web port)
- Starts the apache2 web server through the "apachectl" command

The `index.html` file

- This is a sample `index.html` file that will just show a greeting message:

```
ubuntu@VM1:~$ cat index.html
<!DOCTYPE html>
<html>
<h1>Hello from a web server running inside a container!</h1>
This is an exercise for the INFN course.
</html>
```

- **Create a directory** called `containers/simple` with the command `mkdir -p $HOME/containers/simple` and change to that directory (`cd $HOME/containers/simple`).
- **Copy** both the previous `Dockerfile` and `index.html` to that directory.

Build images via Dockerfiles

- Once we have a Dockerfile, we can create ("build") an image and name it for example "web_server" with the command

  ```
  docker build -t web_server .
  ```

  - **Note**: the **.** at the end the line above is important! (it tells docker we are building taking the Dockerfile and other files from the current directory.)
- We can now run our new image in the background (flag `-d`) with

  ```
  docker run -d -p 8080:80 web_server
  ```

- The `-p 8080:80` part redirects port 80 *on the container* (the port we exposed in the Dockerfile) to port 8080 *on the host system* (that is, your VM).
- Check that everything works opening in a browser this page: http://<VM_ip_address>:8080/
- **Try it now!**

Check that our web server is running

- Check with:

```
ubuntu@VM1:~$ docker ps
CONTAINER
ID          IMAGE                   COMMAND                   CREATED             STATUS              PORT
S                   NAMES
f9dc164be001        web_server          "apachectl -D FOREGR…"   12 minutes ago      Up 12
minutes       0.0.0.0:8080->80/tcp   laughing_pare
```

- Stop the container with:

```
ubuntu@VM1:~$ docker stop f9dc164be001
```

- You can now type `docker run -d -p 8080:80 web_server` any time you want to instantiate a new web server.
- What happens if you type `docker run -d -p 8081:80 web_server` ?

Containers are *ephemeral*

- **An important point to remember** is that any data that is created within a running container is only available within the container, and only when the container is running.
- Let's prove this. Run a container using the Ubuntu image in interactive mode:

```
docker run -i -t ubuntu /bin/bash
```

- Once in the container, create a file and verify it is there:

```
root@2000824922fb:/# touch my_new_file  # this creates an empty file in the container file system
root@2000824922fb:/# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  my_new_file  opt  proc  root  run  sbin  srv  sys
  tmp  usr  var
root@2000824922fb:/#
```

- Now exit from the container. Run it again with the same command as above (`docker run -i -t ubuntu /bin/bash`).
- Is the file still there? (it should not!)
  - It is not there, because every time you write `docker run` you start a new Ubuntu container.

Connect a container to a host file system

- So, what if we want to retain data within a container?
- We can map a directory that is available *on the host* (the system where we run the docker command, e.g. your VM), to a directory that is available *on the container*. This is done with the docker flag `-v`, like this:

```
docker run -v /host/directory:/container/directory <other docker arguments>
```

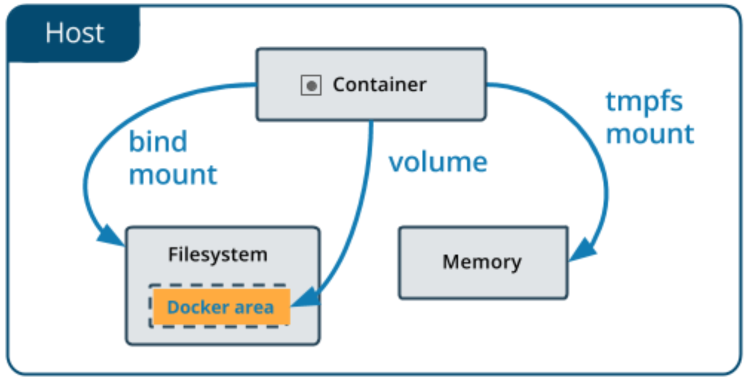
- For example:
  - Create a "scratch" directory with `mkdir -p $HOME/containers/scratch` and change to that directory (`cd $HOME/containers/scratch`).
  - Create a dummy 10MB file there with `head -c 10M < /dev/urandom > dummy_file`
  - Map the scratch directory to the directory `/cointainer_data` on the container:

```
docker run -v $HOME/containers/scratch/:/container_data -i -t ubuntu /bin/bash
```

- Now, when you are *within the container*, if you write `ls /container_data` you should see the dummy file. **Do it now**.
- Verify also that you can write to that directory from the container, and that you can find the written data when you are on the VM. *What about permissions?*

Connect a container to a Docker volume (1)

- In the previous slide, we mapped a directory that was available *on the host* to a directory on the container. This is called a **bind mount**.
- But what if we want to copy or move our docker container to a different host, with a different directory structure? Or perhaps with a different operating system? Remember that Docker promises to be system-independent.
- We can (and should generally prefer to) use **Docker volumes**.
- Docker volumes are persistent but are not tied to the specific filesystem of the host and are completely managed by Docker itself.



We'll see later what a `tmpfs` mount is.

Connect a container to a Docker volume (2)

- You can create a new Docker volume with the command

```
docker volume create some-volume
```

  - Try these self-explanatory commands:

```
docker volume ls
docker volume inspect some-volume
docker volume rm some-volume
```

- You can also start a container with a volume which does not exist yet with the `-v` flag. It will be automatically created:

```
docker run -i -t --name myname -v some-volume2:/app ubuntu /bin/bash
```

  - Notice that we also introduced here **the flag `--name`** to give an explicit name (here: `myname`) to a container.
  - In this case, check the container with the command `docker inspect myname` and look for the `Mounts` section. **Try it now**: what do you see?

Removing docker volumes

- As we said, Docker volumes are directly managed by Docker, in some Docker-specific area (see the `docker inspect` command we used earlier to know more). They occupy some space in the local file system.
- When you do not need a docker volume anymore, it is wise to reclaim its space:

```
docker volume rm <volume_name>
```

  - Can you remove a volume which is being used by a container? Try.
- More in general, you can remove all unused docker volumes with

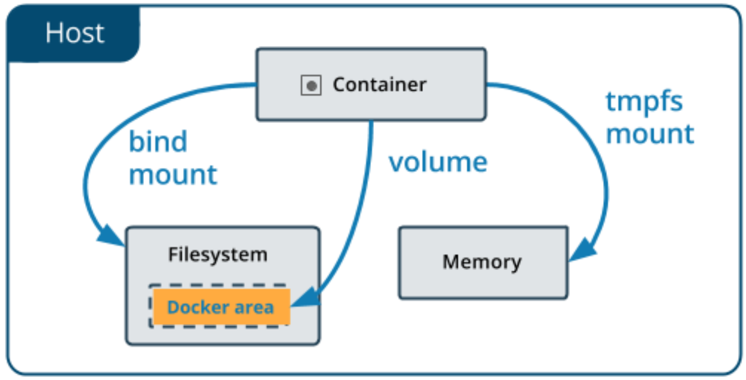
```
docker volume prune
```

  - Note that the `docker system prune` command we showed previously **does not remove volumes**!

`tmpfs` mounts

- If you are running Docker on Linux (so far, this is the case for us), there is a third option to mount a volume on a container: the so-called **`tmpfs` mount** option.
- When you create a container with a `tmpfs` mount, the container can create files outside the container's writable layer, directly into the host system memory (RAM).
- This is a **temporary volume**, i.e. it will be automatically removed once the container exits. It is useful for example if you have sensitive data that you do not want to store neither in the container nor in a dedicated volume (be it filesystem-based or docker-based).
- An example of mounting the `/app` directory of a container under a `tmpfs` mount (whatever you write in that directory will only be stored in RAM):

```
docker run -it --name mytmp --tmpfs /app ubuntu /bin/bash
```



Detour: using the `tar` command

- In Linux, `tar` (for "tape archive": this tells you how old this command is) is one of the most useful commands to package several files or directories into a single file, often called `tarball`. It can be combined with the `gzip` tool to also compress the archived file (with this option, it is like the Windows `zip` and `unzip` tools).
- Typical extensions:
  - `.tar` → uncompressed archive file using tar
  - `.zip` → compressed archive file using zip
  - `.gz` → file (it can be an archive or not) compressed using gzip
  - `.tar.gz` or `.tgz` → a compressed archive file using tar
- Examples of some useful `tar` commands (see e.g. https://www.howtoforge.com/tutorial/linux-tar-command/ for more information):
  - Create an archive file called `my_devstuff.tar` with the directory `/home/davide/devstuff/` and its content:
    ```
    tar -cvf my_devstuff.tar /home/davide/devstuff/ # my_devstuff.tar will be created in the current directory
    tar -xvf my_devstuff.tar   # extract my_devstuff.tar in the current directory
    tar -xvf my_devstuff.tar -C /home/davide/newdir   # extract my_devstuff in another directory
    ```
  - The same archive as above, but compressed:
    ```
    tar -cvzf my_devstuff.tar.gz /home/davide/devstuff/  # note the z flag to enable compression
    tar -xvf my_devstuff.tar.gz # note that the uncompress command is the same as above
    ```
  - List the content of an archive file, compressed or not:
    ```
    tar -tf <tar_filename>
    ```

Copy an <u>image</u> somewhere else

- So far, we have pushed our images to Docker Hub, in a <u>public repository</u>. But what if we wanted to copy our images to another system, *without going through Docker Hub*?
- Docker allows to <u>export an image to a tar file</u> specifying its name (you could also compress it, if you wanted to save space):

```
# docker save –o my_exported_image.tar my_local_image
# docker save my_local_image | gzip > my_exported_image.tar.gz
```

- You can then copy the tar file (`my_exported_image.tar`) to another system via e.g. `scp`, and then import it to a docker image on that system:

```
# docker load –i my_exported_image.tar
```

Copy a Docker volume somewhere else

- Recall that Docker volumes are independent of the local file system structure and are managed directly by the Docker engine.
- In order to transfer a docker *volume* to another host, you must first **back it up to a tar file** using the `--volumes-from` flag. This flag must be applied to an *existing container* (even if not running) which mounted the volume you want to back up, with a command similar to the following one:

```
docker run --rm --volumes-from EXISTING_CONTAINER -v /tmp:/backup ubuntu tar cvf
/backup/backup.tar /app
```

  - This command backs up a volume that was mounted by the EXISTING_CONTAINER under the directory `/app` into the file `backup.tar` in the `/tmp` directory of the local system.
- At this point, you can simply transfer the tar file to another machine and restore it to another running container.
- For example, once you have the tar in the `/tmp` directory of another machine, you can do:

`docker run -it -v /app --name myname2 ubuntu /bin/bash` (this runs `myname2` interactively)

(in another shell) `docker run --rm --volumes-from myname2 -v /tmp:/backup ubuntu bash -c "cd /app && tar xvf /backup/backup.tar --strip 1"`

Hands-on: copy your image to your laptop and run it

- You should now copy an image you created on your VM in the previous assignments to your own laptop.
- You should then load and run it locally. There are a couple of cases here:
  1. **If you have Docker already installed on your laptop**, you can load and run the image immediately.
  2. **If you do not have Docker installed on your laptop**, install it.
     - **Windows**: https://docs.docker.com/docker-for-windows/install/
     - **Linux**: if you have Ubuntu (similar this other Debian-derived distributions), see https://docs.docker.com/install/linux/docker-ce/ubuntu/. If you have RedHat, see https://docs.docker.com/install/linux/docker-ce/centos/
     - **MacOS**: https://docs.docker.com/docker-for-mac/

Doing local development

- Now that you have Docker installed on your laptop, try out the commands you issued on your VM (which is running on some Cloud infrastructure) locally.
- In general, it is very handy to do local developments with docker directly on your laptop. In many cases, this applies also to relatively complex environments, as we will see. Once we are happy with the results in a local environment, we can move to the Cloud.

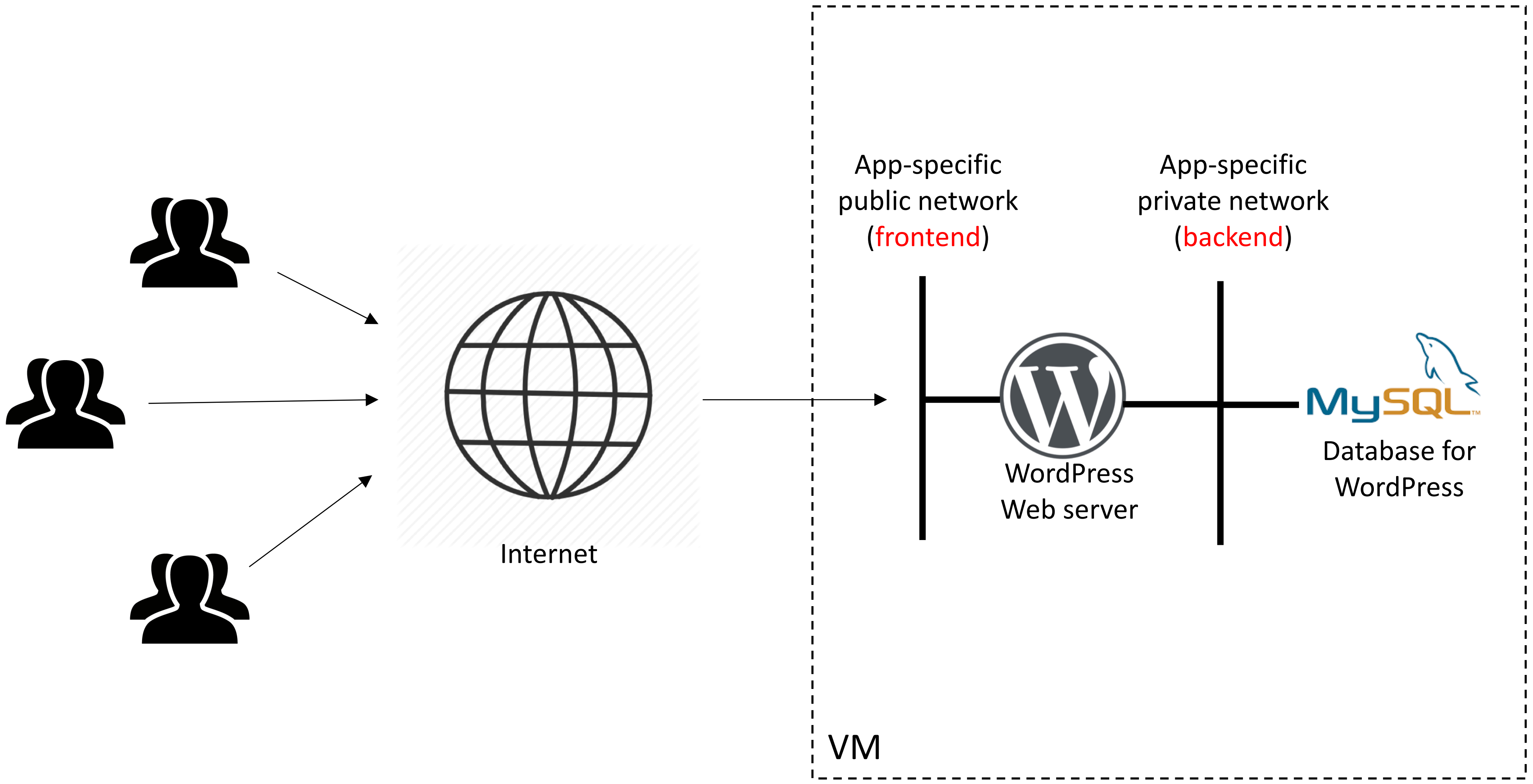
Application stacks: Docker Compose

- We have seen how easy it is to create and run a Docker container, pulling images from Docker Hub. We then learned how to extend an image, either manually adding packages to it and then committing the changes or writing a Dockerfile to automatize the whole process.
- We now also know how to export an image to a tar file, for example because we want to share it without using Docker Hub, or to save it for backup purposes.
- We will now move on to consider how to create “**application stacks**”: that is, how to create multiple containers linked together to provide a multi-container service, all on a single VM.
- This is done via **Docker Compose**, invoked via the `docker-compose` command.

A scenario for Docker Compose

- `docker-compose` works by parsing a text file, written in the YAML language (see https://yaml.org for more info). This file, which is by default called `docker-compose.yml`, defines how our application stack is structured.
- We will now use `docker-compose` to create and launch an application stack made of two connected containers, both running on our VM:
  1. A **MySQL database**. It won't be accessible from the Internet.
  2. A **WordPress instance**. It will be accessible from the Internet. WordPress (https://wordpress.org) is a very popular open source software used to create websites or blogs.

Our app stack architecture



docker-compose.yml

```
version: '3'
services:
  database:
    image: mysql:5.7
    environment:
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
      - MYSQL_PASSWORD=testwp
      - MYSQL_RANDOM_ROOT_PASSWORD=yes
    networks:
      - backend
  wordpress:
    image: wordpress:latest
    depends_on:
      - database
    environment:
      - WORDPRESS_DB_HOST=database:3306
      - WORDPRESS_DB_USER=wordpress
      - WORDPRESS_DB_PASSWORD=testwp
      - WORDPRESS_DB_NAME=wordpress
    ports:
      - 8080:80
    networks:
      - backend
      - frontend

networks:
  backend:
  frontend:
```

This builds the container for the database, with only the "backend" network

Container image for mySQL (from Docker Hub)

Configuration variables for the container software

This builds the container for WordPress, with both the "backend" and "frontend" networks

Container image for WordPress (from Docker Hub)

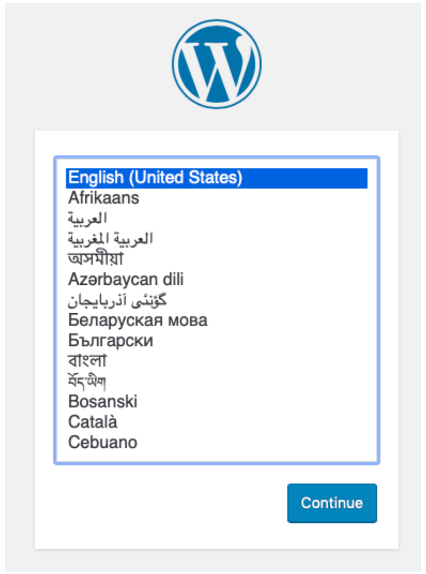
Note that here we refer to the other container

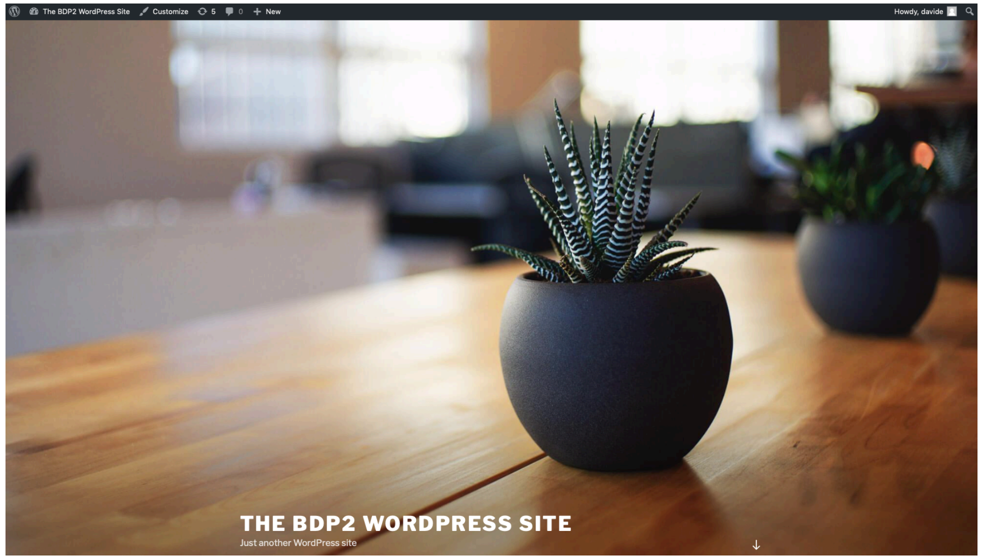
Port 8080 on the host (VM1) is mapped to port 80 on the container

"Obvious" note: although this is just for a demo, do not use the passwords shown in this screen!

Build & run the application stack

- Create a directory with `mkdir -p $HOME/containers/compose`, change to that directory with `cd $HOME/containers/compose` and copy there the `docker-compose.yml` file of the previous slide.
- Install docker-compose on your VM with
```
# sudo yum install docker-compose
```
- Build the application stack:
```
# docker-compose up --build --no-start
```
- Start it:
```
# docker-compose start
```
- If you now open a browser pointing to your VM's public IP address on port 8080 (look at the previous `docker-compose.yml`), you should get the set-up page for WordPress on the right. Go on and set it up.
- Once WordPress is set up, you should see the default WordPress home page, like the one on the right (which of course you could graphically customize).
- Once the app stack is started, the running containers can be seen with the usual `docker ps` command.
- The application stack can be stopped with:
```
# docker-compose stop
```
- **Try this yourself now.**





Specifying volumes in `docker-compose`

- If you wish to use docker volumes, they can also be specified in the `docker-compose` YAML file. For example:

```
version: ’3’
volumes:
    my_volume_1:
    my_volume_2:
services:
    application_1:
            volumes:
                    - my_volume_1:/app1/dir
            […]
    application_2:
            volumes:
                    - my_volume_2:/app2/dir
[…]
```

This automatically creates the Docker volume `my_volume_1`, mapping it to the directory `/app1/dir` on the container

Why use Docker volumes?

- Take the previous Wordpress + MySQL example. If you completely **remove** the application stack (with `docker-compose down`, this is <u>different</u> from just stopping it) and then build and start it, you will get an entirely new Wordpress installation – so you will have to set it up from scratch.
- You could instead persist your MySQL database modifying your `docker-compose.yml` file to use volumes, like this:

```
volumes:
  db_data:
services:
  database:
    image: mysql:5.7
    volumes:
      - db_data:/var/lib/mysql
    [...]
```

Create a Docker volume called `db_data`, mapped to the `/var/lib/mysql` directory on the container (this is where MySQL data is stored)

Some “advanced” tips on Docker Compose (1)

- You can check what’s going on with your application stack after you have started it (with `docker-compose start`) with

  ```
  docker-compose logs -f
  ```

  This will continuously follow the logs generated by your containers. Very useful to verify possible issues.

- Adding `restart: always` for any or all the services in your `docker-compose.yml` file will make them restart automatically when your host system boots, or when the Docker daemon restarts.
  - Note: **it will not** automatically restart your containers if you manually kill them.

Some “advanced” tips on Docker Compose (2)

- Depending on `depends_on` is not always a good idea: this only **starts** containers in a certain dependency order.
- Example from the Wordpress + MySQL stack, taken from the log output:

```
database_1  | 2019-11-23T09:30:22.821543Z 0 [Note] InnoDB: Buffer pool(s) load completed at 191123  9:30:22
wordpress_1  |
wordpress_1  | MySQL Connection Error: (2002) Connection refused
wordpress_1  |
wordpress_1  | WARNING: unable to establish a database connection to 'database:3306’
wordpress_1  |   continuing anyways (which might have unexpected results)
```

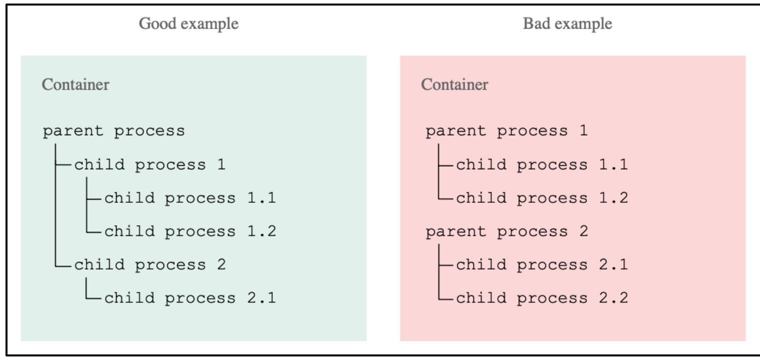
- If you need to wait until a certain container is really “ready” (whatever this means for your application), you must use something more specific than `depends_on`, e.g. https://github.com/vishnubob/wait-for-it or https://github.com/Eficode/wait-for. Be warned that in some cases this is not enough. For more info: https://docs.docker.com/compose/startup-order/

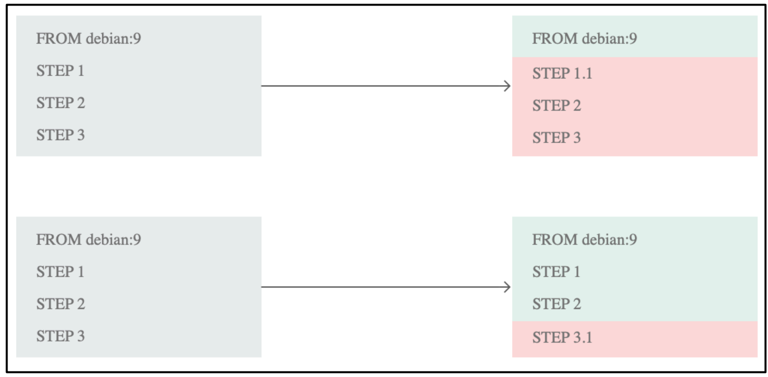
Limitations of Docker Compose

- As we have seen, Docker Compose is very handy to create combinations of containers running on the same machine (in our case, your VM).
- It is best suitable **if you don't need automatic scaling of resources or multi-server environments**.
- For complex set ups, other tools such as Docker Swarm or Kubernetes are more appropriate. We'll cover them in one of the next lectures.

Some best practices for writing containers

1. Put a **single application per container**. For example, do not run an application *and* a database used by the application in the same container.
2. Explicitly **define the entry point in the container** with the CMD command in the Dockerfile.
3. If in a Dockerfile you have **commands that change often, put them at the bottom of the Dockerfile**. This way, you speed up the process of building the image out of the Dockerfile.
4. **Keep it small**: use the smallest base image possible, remove unnecessary tools, install only what is needed.
5. Properly **tag your images**, so that it is clear which version of a software it refers to.
6. **Do you really want / can you use a public image?** Think about possible vulnerabilities, but also about potential license issues.





More (and more detailed) information available at https://bit.ly/2Zr6Hyq

A few words on Docker security (1)

- As we have seen so far, if you want to run Docker containers, you need to have Docker installed on your host system.
- If Docker is not installed, you can install it yourself, **but you must have root access** in order to do that.
- Once you have installed Docker, you can download and execute containers from DockerHub or other sources.
  - **Be careful**, because this is potentially a big security threat: some containers that you download might be compromised (e.g. include viruses or trojan)!
- How can you send passwords, certificates, encryption keys, etc. to tasks / applications in a Docker swarm cluster? **Do not** embed them into the containers, and **do not** store them e.g. in GitHub repositories!
  - Docker has a "Secrets Management" feature, which is a standardized interface for accessing secrets. See https://dockr.ly/2H4M5SU for details.
  - Other resource orchestrators, such as Kubernetes (check next lectures), have similar solutions.

A few words on Docker security (2)

- If the *host* where the Docker daemon is running gets compromised, container isolation is gone. So, it is important to make sure that the **host system is properly secured** (i.e. regularly update it!).
- On other hand, there could be exploits that make it possible for **containers to bypass isolation** (remember that the Docker daemon requires root privileges) and get access in privileged mode to the host system.
- Since you can so easily start up containers on a system, there is the possibility of a **Denial of Service** attack, targeting to consume all resources on the host system.
- **Do not assume that containers should be immutable!** They might contain outdated software, that must be periodically patched and upgraded.
- For more details, see http://bit.ly/2kEpV16.

An important issue with Docker

- There is no doubt that Docker containers are very handy and useful. However, in general, the adoption (i.e. installation) of Docker is quite slow <u>in traditional clusters and in HPC centers</u>.
- In other words, what often happens is that Docker itself is **not installed** on the machines. Therefore, you cannot run containers (unless you have root privileges and can therefore install it autonomously).
  - This is sometimes because the system administrators might think that there could be important security concerns with Docker, or because it is another service to maintain, or because it is too new… and so on.

udocker

- In the INDIGO-DataCloud project, we developed **udocker**: it’s a kind of “userland docker”, i.e. a tool to run contents of Docker images without requiring any support from the kernel.
- There are no special dependencies, aside from python 2.7 and libc.
- In particular, udocker is intended to be run by unprivileged users.
- No special daemon is required. System-wide installation is possible but entirely optional (each user can “install it” individually).
- It is freely available at https://github.com/indigo-dc/udocker

The `udocker` architecture

- It is a single-file python script.
- It fetches public images by default from Docker Hub.
  - It can also import image tarballs exported via `docker save`.
- It creates a container filesystem hierarchy in `$HOME/.udocker`
- It internally uses PRoot (see https://proot-me.github.io) for limited sandboxing.
  - Almost no CPU overhead.
  - Negligible data I/O overhead.
  - Sensible metadata I/O overhead.
- Other execution mechanisms than PRoot are available (see later).

`udocker` advantages

- Provides a docker-like command line interface.
- Supports a subset of docker commands: search, pull, import, export, load, create and run.
- Understands docker container metadata.
- Can be deployed by end users.
- Does not require privileges for installation.
- Does not require privileges for execution.
- Does not require compilation: just transfer the Python script and run.
- Encapsulates several execution methods.
- Includes the required tools already compiled to work across systems.
- Tested also with GPGPU and MPI applications.
- Runs on new and old Linux distributions, including CentOS 6, CentOS 7, Ubuntu 14, Ubuntu 16, Fedora, etc.

`udocker` limitations

- Images cannot be created by `udocker`.
  - That is, you must use Docker on another system to build images!
- Privileged OS operations are not possible.
- Debugging inside containers does not work.
- **`udocker` is not a privilege boundary!** (i.e., it does not enforce special security measures: the `udocker` process runs with the privilege of the current user.)

Install `udocker` on your VM

- Let’s create a dedicated directory and change to it:

```
mkdir -p $HOME/containers/udocker
cd $HOME/containers/udocker
```

- `udocker` is a single file Python script, which we can download from its public repository:

```
curl https://raw.githubusercontent.com/indigo-
dc/udocker/master/udocker.py > udocker
```

- Let’s make the script executable and run it with the “install” option to set it up (this only needs to be done once):

```
chmod u+x ./udocker
./udocker install
```

Check that `udocker` works

- After installation, `./udocker help` should return something like this:

```
Syntax:
  udocker  <command>  [command_options]  <command_args>


Commands:
  search <repo/image:tag>        :Search dockerhub for container images
  pull <repo/image:tag>          :Pull container image from dockerhub
  images                         :List container images
  create <repo/image:tag>        :Create container from a pulled image
  ps                             :List created containers
  rm  <container>                :Delete container
  run <container>                :Execute container
[…]
```

`docker` vs. `udocker`: comparison

docker

- `# docker search ubuntu`
- `# docker pull ubuntu`
- `# docker run --name=my_ub ubuntu echo 'Hello from a container'`
- `# docker rm mycontainer`

udocker

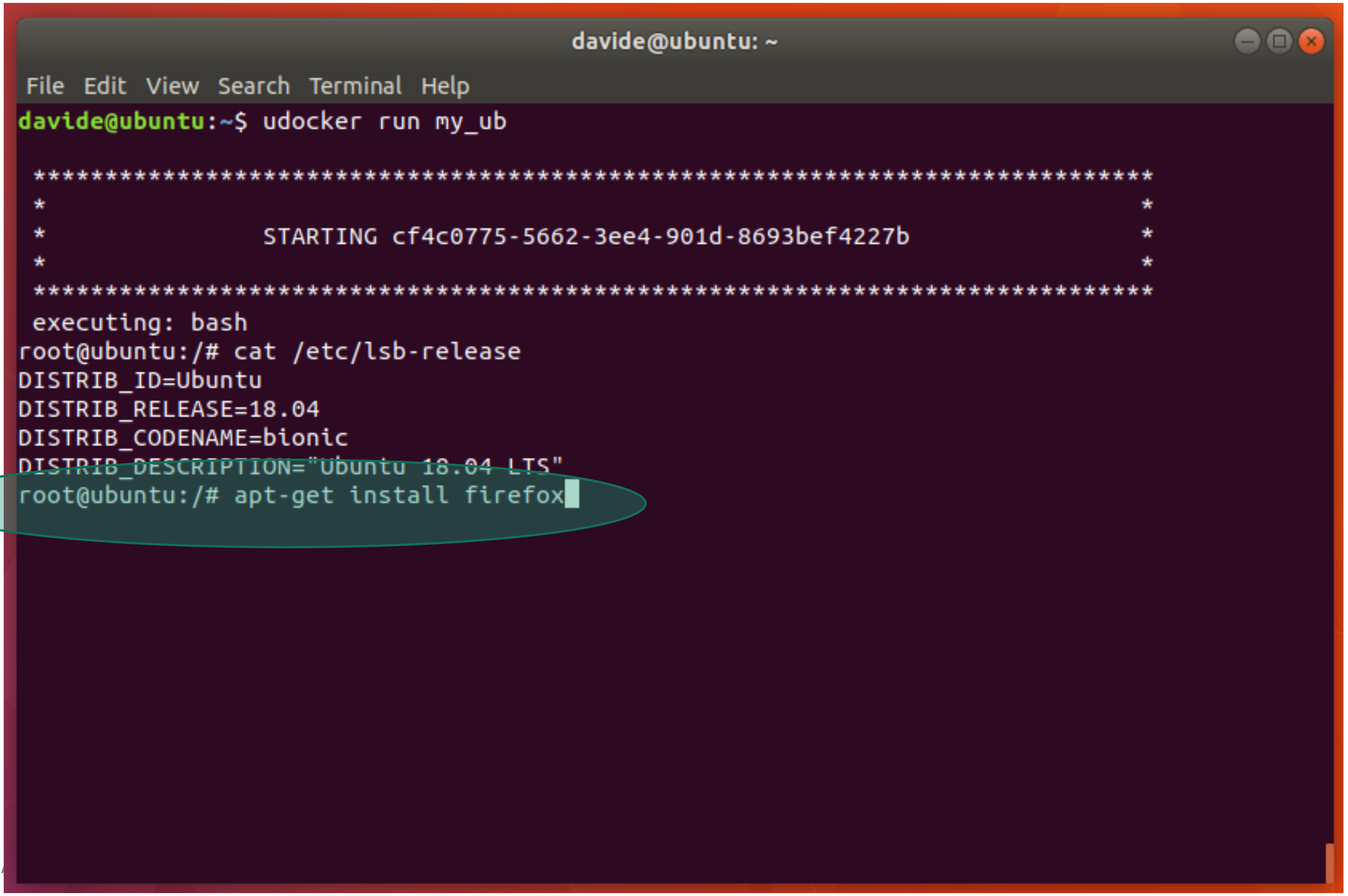
- `$ udocker search ubuntu`
- `$ udocker pull ubuntu`
- `$ udocker create --name=my_ub ubuntu`
- `$ udocker run my_ub echo 'Hello from a container'`
- `$ udocker rm my_ub`

Tip: if you have installed `udocker` in directory `/mydir`, put the following line at the end of the file `.bashrc`:

```
alias udocker= '/mydir/udocker'
```

You can then call `udocker` directly, without specifying any path before it.

udocker and root emulation



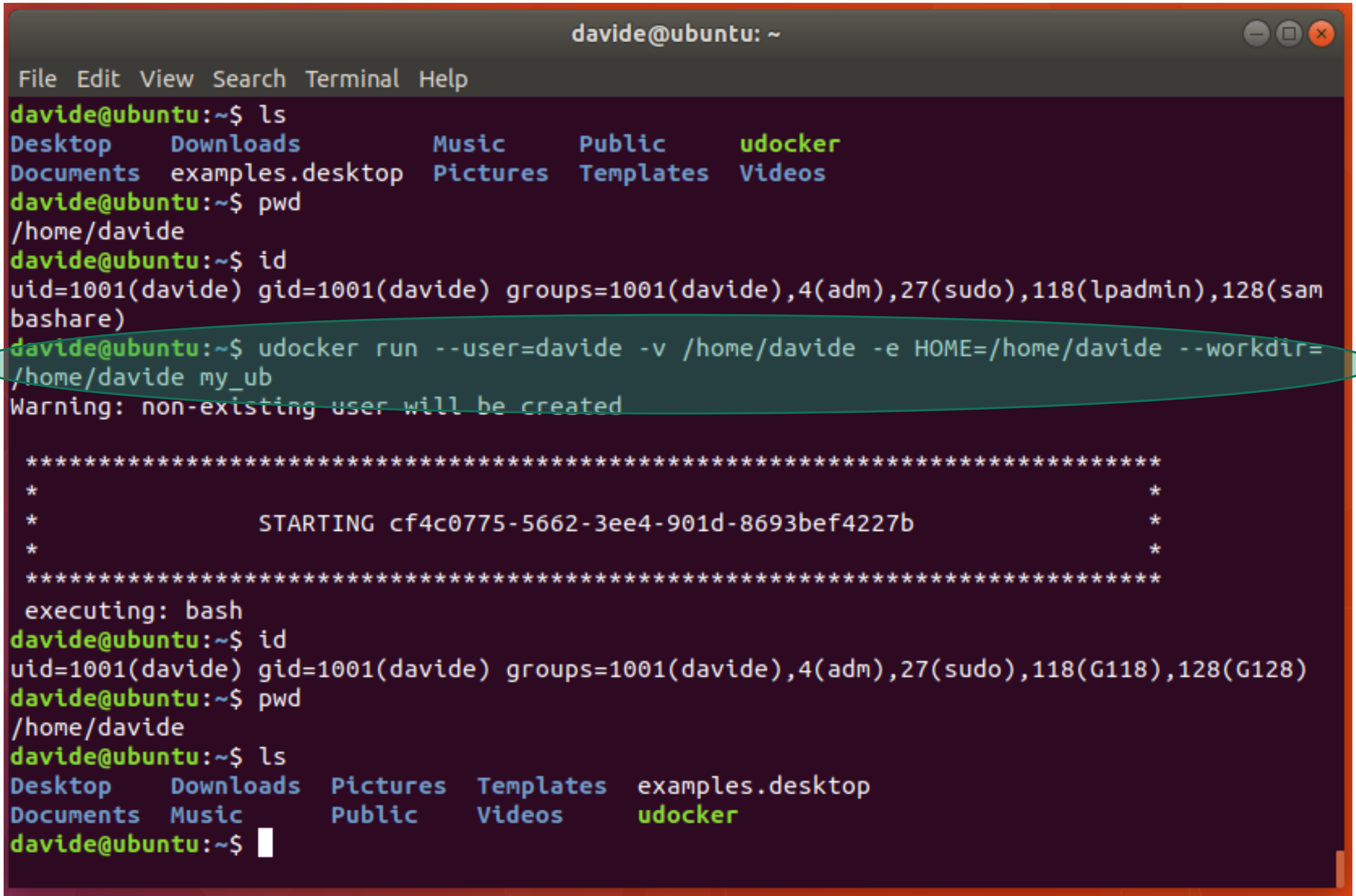
```
davide@ubuntu:~$ udocker run my_ub

 ******************************************************************************
 *                                                                            *
 *               STARTING cf4c0775-5662-3ee4-901d-8693bef4227b                *
 *                                                                            *
 ******************************************************************************
 executing: bash
root@ubuntu:/# cat /etc/lsb-release
DISTRIB_ID=Ubuntu
DISTRIB_RELEASE=18.04
DISTRIB_CODENAME=bionic
DISTRIB_DESCRIPTION="Ubuntu 18.04 LTS"
root@ubuntu:/# apt-get install firefox
```

Root emulation

Run a container as yourself

Run as myself, mounting my own local directory



```
davide@ubuntu:~$ ls
Desktop     Downloads           Music      Public      udocker
Documents   examples.desktop    Pictures   Templates   Videos
davide@ubuntu:~$ pwd
/home/davide
davide@ubuntu:~$ id
uid=1001(davide) gid=1001(davide) groups=1001(davide),4(adm),27(sudo),118(lpadmin),128(sam
bashare)
davide@ubuntu:~$ udocker run --user=davide -v /home/davide -e HOME=/home/davide --workdir=
/home/davide my_ub
Warning: non-existing user will be created

 ******************************************************************************
 *                                                                            *
 *               STARTING cf4c0775-5662-3ee4-901d-8693bef4227b                *
 *                                                                            *
 ******************************************************************************
 executing: bash
davide@ubuntu:~$ id
uid=1001(davide) gid=1001(davide) groups=1001(davide),4(adm),27(sudo),118(G118),128(G128)
davide@ubuntu:~$ pwd
/home/davide
davide@ubuntu:~$ ls
Desktop     Downloads   Pictures   Templates   examples.desktop
Documents   Music       Public     Videos      udocker
davide@ubuntu:~$ 
```

udocker: Execution methods

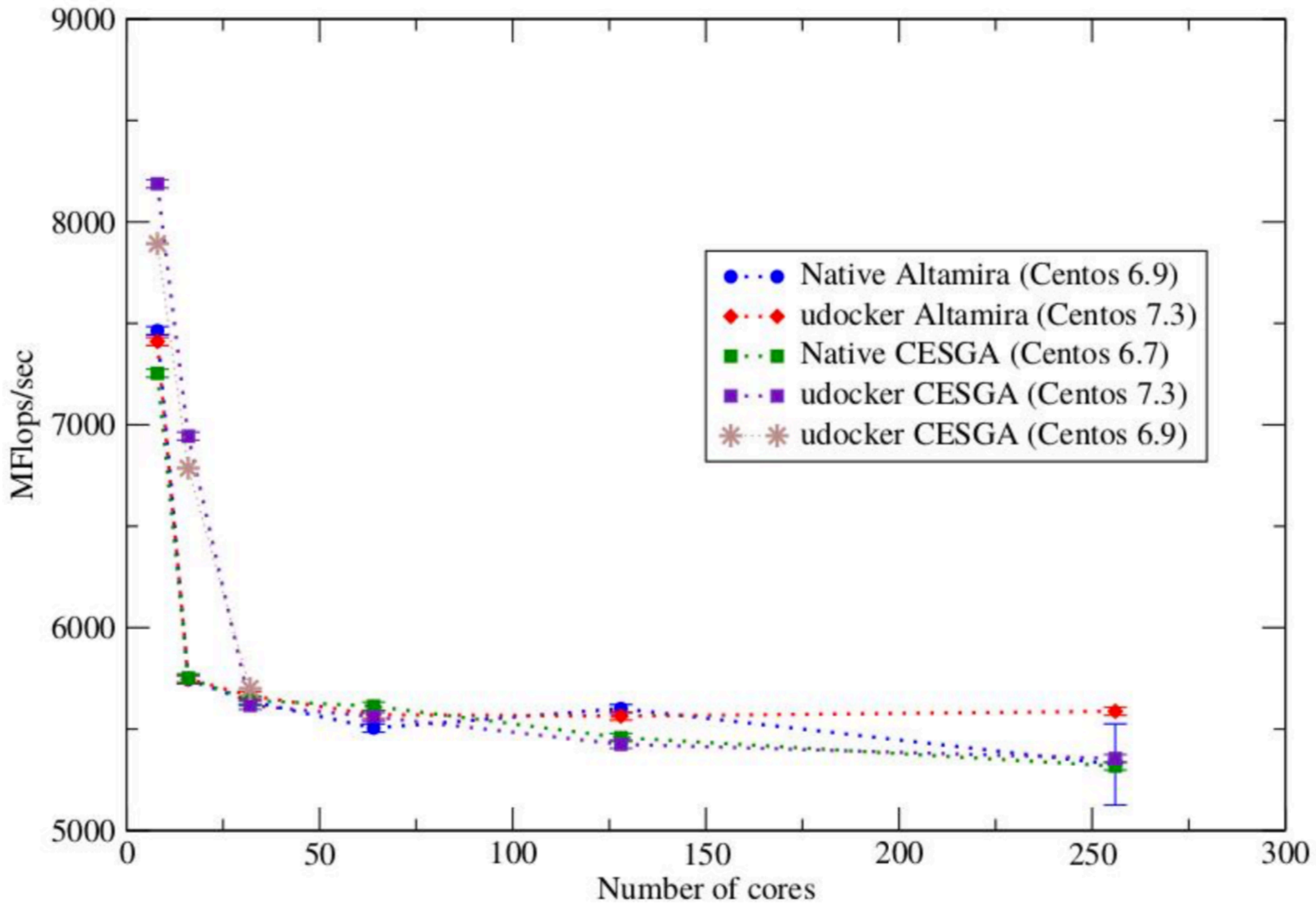
Slide courtesy Jorge Gomes

- udocker supports several techniques to achieve the equivalent to a chroot without using privileges
  - They are selected per container id via execution modes

Mode	Base	Description
P1	PRoot	PTRACE accelerated (with SECCOMP filtering) ← DEFAULT
P2	PRoot	PTRACE non-accelerated (without SECCOMP filtering)
R1	runC	rootless unprivileged using user namespaces
F1	Fakechroot	with loader as argument and LD_LIBRARY_PATH
F2	Fakechroot	with modified loader, loader as argument and LD_LIBRARY_PATH
F3	Fakechroot	modified loader and ELF headers of binaries + libs changed
F4	Fakechroot	modified loader and ELF headers dynamically changed
S1	Singularity	where locally installed using chroot or user namespaces

udocker & Lattice QCD

Slide courtesy Jorge Gomes



OpenQCD is a very advanced code to run lattice simulations

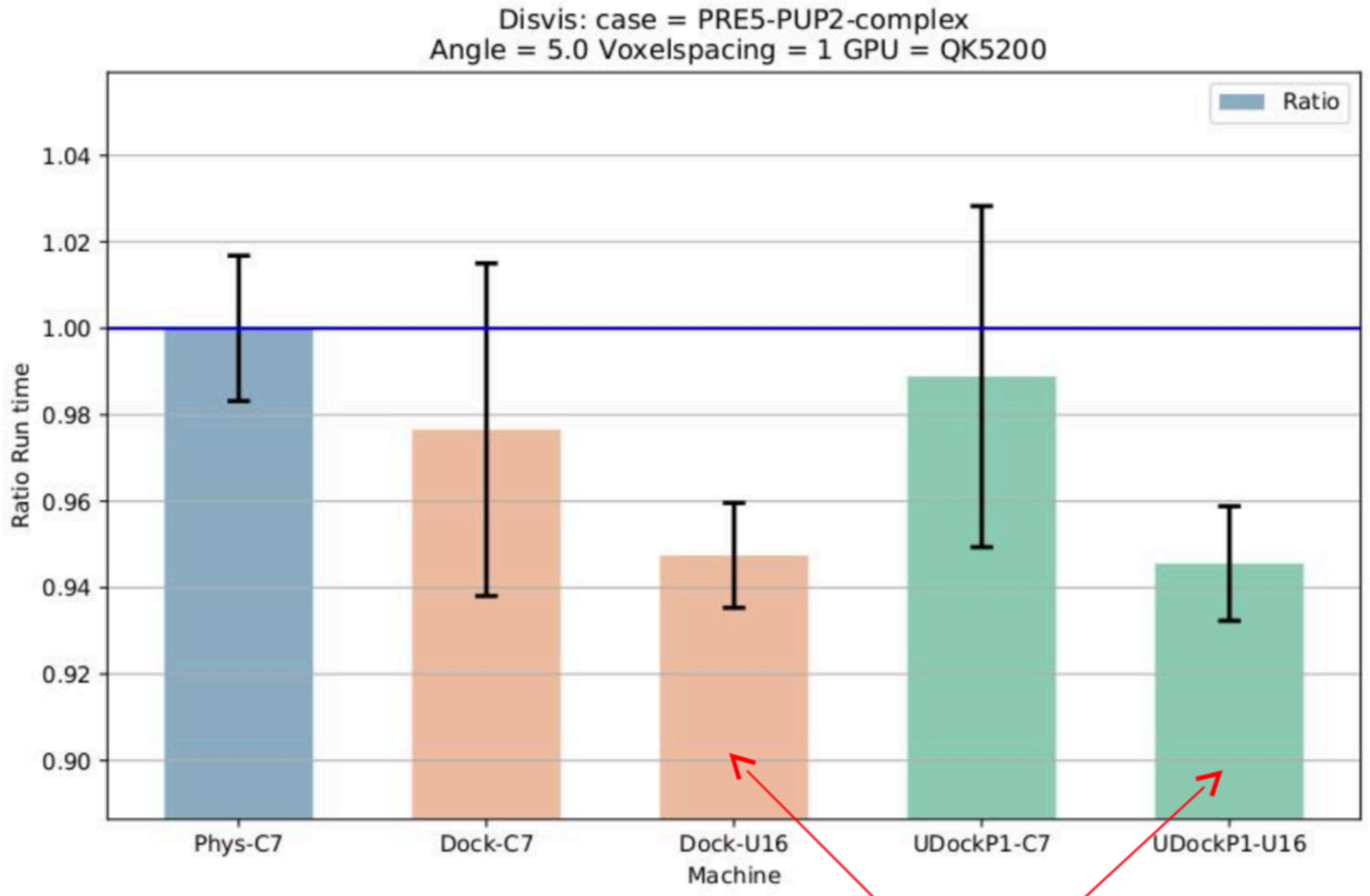
Scaling performance as a function of the cores for the computation of application of the Dirac operator to a spinor field.

Using OpenMPI

udocker in P1 mode

udocker & Biomolecular complexes

Slide courtesy Jorge Gomes



DisVis is being used in production with udocker

Performance with docker and udocker are the same and very similar to the host.

Using OpenCL and NVIDIA GPGPUs

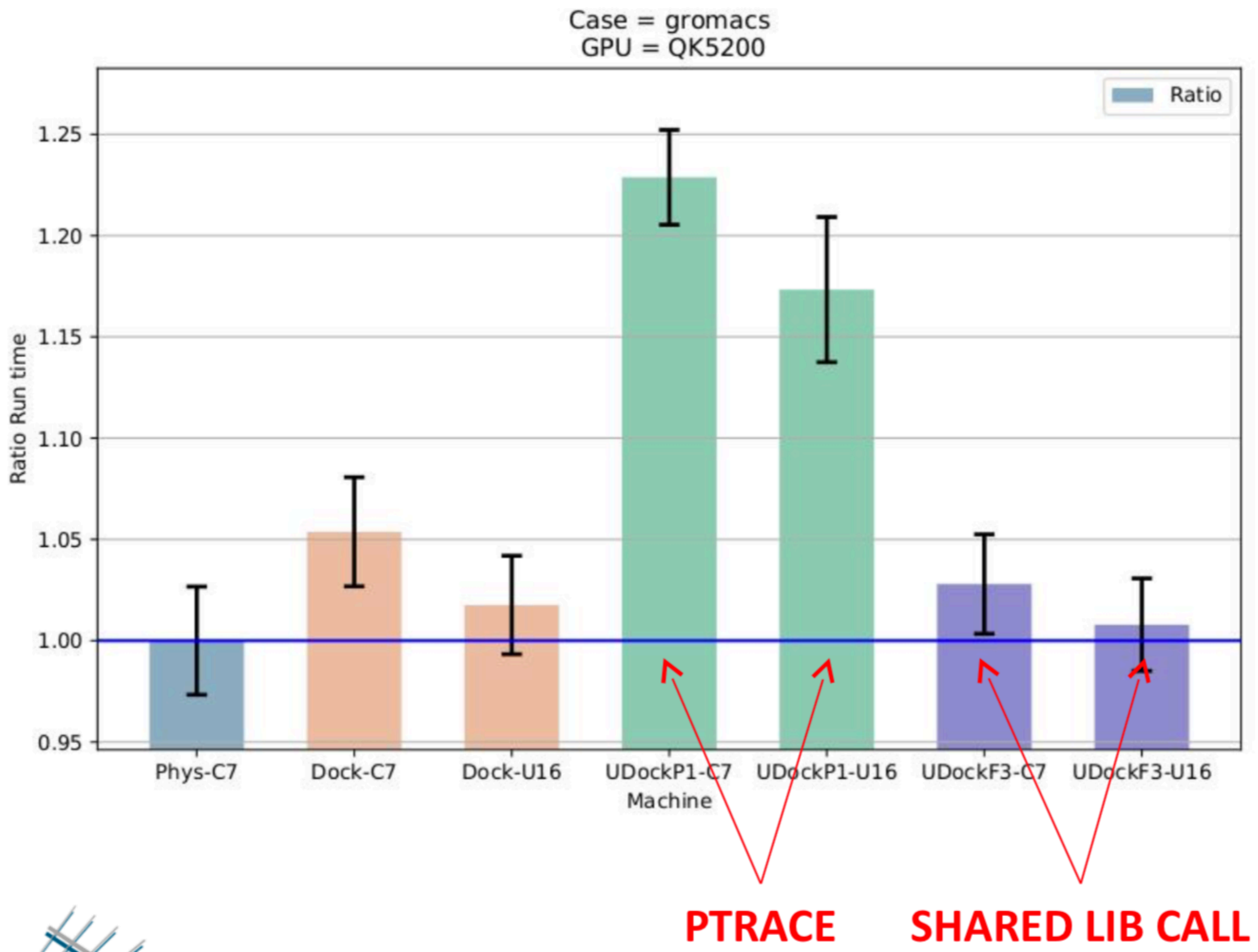
udocker in P1 mode

Better performance with Ubuntu 16 container

Co

udocker & Molecular dynamics

Slide courtesy Jorge Gomes



Gromacs is widely used both in biochemical and non-biochemical systems.

udocker P mode have lower performance udocker F mode same as Docker.

Using OpenCL and OpenMP

udocker in P1 mode
udocker in F3 mode

udocker & Phenomenology

Slide courtesy Jorge Gomes

Performance Degradation

	Compiling	Running
HOST	0%	0%
DOCKER	10%	1.0%
udocker	7%	1.3%
VirtualBox	15%	1.6%
KVM	5%	2.6%

udocker in P1 mode

MasterCode connects several complex codes. Hard to deploy.

Scanning through large parameter spaces. High Throughput Computing

C++, Fortran, many authors, legacy code

Hands-on: test `udocker` (1)

- Write this simple Python program to the file `quotes.py` in the udocker directory on your VM. It extracts the ”Quote of the Day” using the free `quotes.rest` API provided by “They Said So”:

```
import json
import requests

r = requests.get('http://quotes.rest/qod.json')
j = r.json()

author = j['contents']['quotes'][0]['author'].encode("utf-8")
quote  = j['contents']['quotes'][0]['quote'].encode("utf-8")

print '"%s"' % quote
print '(%s)' % author
```

- Verify that it works, running it with `python quotes.py`.

Hands-on: test `udocker` (2)

- Now create a **new docker image** on your VM and copy `quotes.py` to that image through this `Dockerfile` (here we are deriving the new image from the minimal Linux image called `alpine`):

```
FROM alpine
RUN apk add --no-cache python py-pip
RUN pip install requests
COPY quotes.py /quotes.py
ENTRYPOINT ["/usr/bin/python", "quotes.py"]
```

  - You are encouraged to investigate yourself the difference between the command `ENTRYPOINT` used above and the command `CMD` used in previous examples.
- As usual, you should build the new Docker image, let's call it `my_quotes`, with

```
docker build -t my_quotes .
```

- We now have a new Docker image called `my_quotes` that prints the quote of the day. Check that it works with

```
docker run my_quotes
```

- **Note**: the size of this new image is about **59MB** and on my VM it took about 20 seconds to build (with the base Alpine image already downloaded). When I instead used a Dockerfile deriving from Ubuntu (this is left as an exercise to you), I got a 434MB image, and the build time (with the base Ubuntu image already downloaded) was 3 minutes and 13 seconds.

Hands-on: test `udocker` (3)

- Now, push the `my_quotes` image to your Docker Hub repository. **Do this on your own** (look back at how we did it previously if you need).
- We now want to check that it works with `udocker` through these commands:

```
udocker pull dsalomoni/test:my_quotes_v1.0
```
(or whatever is the name of your container on Docker Hub)

```
udocker create –name=my_quotes dsalomoni/test:my_quotes_v1.0
udocker --quiet run my_quotes
```

- **Do it now**.
- We have then verified that through `udocker` we are able to pull and run standard containers. This would have worked even if we did not have root privileges nor Docker installed.

Optional Hands-on: containers in batch jobs

- Since `udocker` basically only requires Python, it is possible to submit jobs to a batch system that runs containers, even if Docker is not installed on the execution nodes of the batch system.
- It should work like this:
  - Write a job script that, when executed on a node of the batch system cluster, fetches `udocker` from its public repository.
  - The job script should then run `udocker` *on some container*, writing the output somewhere.
- **An assignment could therefore be to encapsulate one of your applications** in a container and submit one or more jobs making use of `udocker` <u>using the HTCondor cluster that you will create during this course</u>.

Summary

- We covered basic concepts about **Containers**, comparing them to Virtual Machines.
- We saw how to execute a container, list docker images and extend them to create new containers.
- We then learned how to push containers to repositories on Docker Hub and simplified the building of containers via Dockerfiles.
- We created a container serving web pages and then connected containers to external file systems, to volumes and to `tmpfs` mounts. We also learned how to export and import containers.
- We studied how to combine multiple containers in an application stack with `docker-compose`.
- We then discussed some Docker limitations, in particular regarding security, introduced `udocker` and installed it on our VM.
- We then created a container running a Python program retrieving the Quote of the Day via a REST call, pushed it to our individual public repository on Docker Hub and used `udocker` to run it on our VM.
- Finally, we offered as optional assignment the task to use `udocker` to run batch jobs on an HTCondor cluster.

- The next part of this course will deal with **Orchestration**.