Standard Surface for Applications

- Packaging and deployment
- Composability
- Reuse

Déjà-Vu: Virtual Machines?

Containers seem to tip the balance because they are more efficient and more convenient to use.

Idealised Container: Wraps a Slim Service

For instance: A Python application including dependencies to render https://phonebook.cern.ch
## Container pros and cons:

- 😊 **Smaller virtualization overhead for system calls, I/O, memory translation**
- 😊 **Better at overcommitting with idle services**
- 😊 **Boots faster (with caveats)**
- 😊 **Orchestration tools available**
- 😞 **Weaker isolation**
- 😞 **No “priviliged operations”, e.g. mount**
- 😞 **Linux only**
- 😞 **More moving parts**
Name Spaces

Virtualization of individual kernel resources
Useful utilities: unshare, nsenter, /proc/PID/ns, /proc/PID/mountinfo

- **pid**: Virtual process identifiers:
  - `sudo unshare -fork -pid /bin/bash`
  - `echo $$ → 1`

- **user**: Virtual uid/gid mappings. Enables fake root:
  - `unshare -U -r /bin/bash`

- **net**: detach network adapters

- **mount**: detach directory tree from parent process
  - Mount points can be
    - **private**: complete isolation between process groups
    - **shared**: mounts are propagated upwards and downwards
    - **slave**: mounts are only propagated downwards

- **more**: inter-process communication, host name, ...

Powerful, but: complex to handle manually, hard to diagnose!
Hierarchical resource containers, confines applications

- Steered through the cgroups file system:
  ```
  $ mkdir /sys/fs/cgroup/memory/small
  $ echo $((1024*1024)) > /sys/fs/cgroup/memory/small/memory.limit_in_bytes
  $ echo $$ > /sys/fs/cgroup/memory/small/tasks
  $ cat /sys/fs/cgroup/memory/small/tasks
  13600
  13658
  $ firefox
  Killed
  ```

- Higher level interfaces: cgconfig, cgcreate, cgdelete, ...

- Controllers for memory, cpu pinning, device access, freezing, dots

Useful in its own right, e.g. HTCondor, benchmarks
chroot, layers, and union file systems

- The “image” is usually a tarball with the root file system of the container
- Docker can assemble images from multiple tarballs in “layers”
- The layered approach requires a union file system to create a single root mount point
- Another option: bind mount of writable parts into a read-only root file system (/var, /tmp, /home, ...)

Source: Docker
Container Ecosystem: Engines

**Container Engines**

- **Docker**: most influential one, introduced the push-pull model for containers
  

- **Singularity**: interesting new engine from the HPC world, very lightweight

- **lxc, rkt, systemd-nspawn**
Container Ecosystem: Clusters

**Container Orchestration**

- Mesos and DC/OS: two-level cluster scheduler, good for production services
- Kubernetes: container orchestration, good for running ensembles of containers
- Docker Swarm

**Example: Test cluster with Kubernetes**

Source: Julien Leduc

jblomer@cern.ch
Container Ecosystem: Clusters

**Container Orchestration**

- **Mesos and DC/OS**: two-level cluster scheduler, good for production services
- **Kubernetes**: container orchestration, good for running ensembles of containers
- **Docker Swarm**

**Example: Production Services Cluster on Mesos**

Source: Dario Berzano

Source: Dario Berzano
# Reality: Container Images are Big

## Image Distribution Problem:

<table>
<thead>
<tr>
<th>iPhone App</th>
<th>Docker “App”</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 MB</td>
<td>1 GB</td>
</tr>
<tr>
<td>changes every month</td>
<td>changes twice a week</td>
</tr>
<tr>
<td>phones update staggered</td>
<td>servers update synchronized</td>
</tr>
</tbody>
</table>

**Example: R in Docker**

```
$ docker pull r-base
→ 1GB image
$ docker run -it r-base
$ ... (fitting tutorial)
→ only 30 MB used
```
### CernVM-FS In Containers

#### Bind Mount

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>docker run -v /cvmfs:/cvmfs:shared ...</code> or <code>docker run -v /cvmfs/sft.cern.ch:/cvmfs/sft.cern.ch ...</code></td>
<td>Cache shared by all containers on the same host</td>
</tr>
</tbody>
</table>

#### Docker Volume Driver

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>docker run --volume-driver cvmfs -v cms.cern.ch:/cvmfs/cms.cern.ch ...</code></td>
<td></td>
</tr>
</tbody>
</table>

#### From Inside Container

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>docker run --privileged ...</code></td>
<td>Probably not very much used in practice</td>
</tr>
</tbody>
</table>
Options for CernVM-FS

1. Fuse, mapped from host
   - Shared cache
   - Requires privileges on the host

2. Using Parrot-Cvmfs
   - Pure user-space (ptrace)
   - Can impact performance and stability

Limitations
Can be used to run tasks, does not allow derived containers
Docker Graph Driver Plugin

Work in Progress by Nikola Hardi

graphdriver (plugin)

RW1  RW2  RW3  RW4
L3    thin  L1
L2
L1

Up to 1 GiB

/getcvmfs

L3  L2  L1

Read-Only layer. ~ 300 MiB
Fetched from network as whole.

Read-Only layer.
Created locally per container.

Read-Only layer stored on CVMFS.
Fetched per file, on demand.

thin

RO layer, only metadata. ~ 100 KiB
List of parent layers stored in CVMFS.

getParentLayers(RW1)
getParentLayers(thin)
Containers used for

- Isolation: e.g. replacing glexec, resource containment
- Virtual environments:
  CentOS on Ubuntu, SL4 on CentOS7 (data preservation)
- Unit of scheduling in distributed systems: Kubernetes, Mesos

Docker/Singularity for isolation + CernVM-FS for image distribution:

- Works out of the box with Singularity
- Bind mounts and volume driver for experiment software in Docker
- Full support for Docker’s pull – commit – push lifecycle: CernVM-FS graph driver (expected H2/17)

There are certain dangers with containers

- More moving parts (and moving targets) in your system
- Containers foster an attitude of “capturing the mess”
- Requires automation: containers need to be disposable items (e.g. no carriers for storage, databases)