
CERN

Containers and their predecessors

Pablo Llopis - CERN IT

Why Containers?

Why ~~Containers~~ Virtualization?

Why Virtualization? Isolation.

- Resource isolation (compute, memory, I/O)
- Environment isolation (OS, libraries, packages)

Consequences and benefits of Isolation

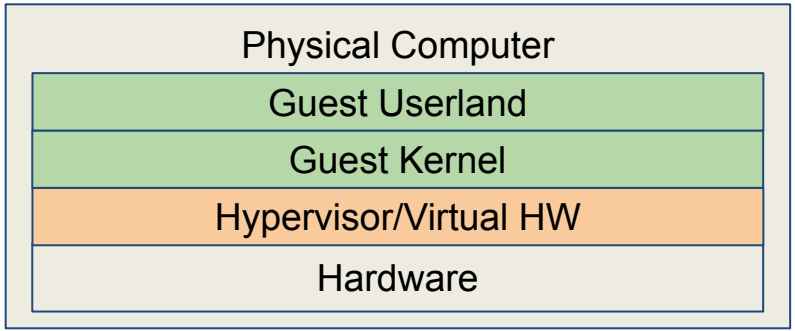
- Higher **resource utilization**: Several unrelated services can share the same physical underlying host.
- **Flexibility**: Services are *decoupled* from the underlying physical layout.
- **Scale**: Easier and faster to instantiate new services, as they are now 100% software and less hardware-dependant.
- **Economy**: Higher utilization results in more usage per €.

A Brief History of **Virtualization**

(With a special focus on x86)

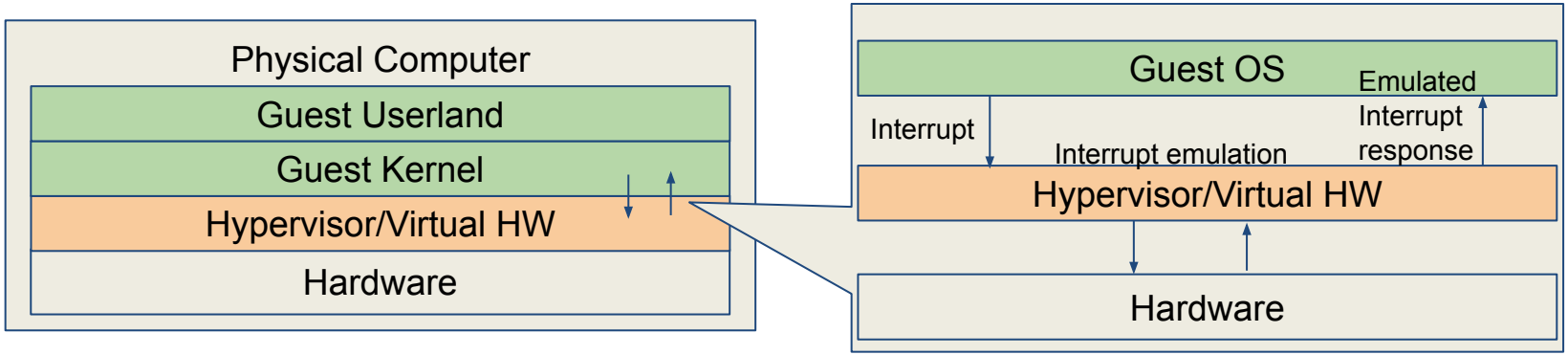
- Virtual machines date back to the 60s, mostly pioneered by IBM. Today, virtualization technologies have grown to become a very relevant and active field of computing.

- Virtual machines traditionally emulate a full computer system.



A Brief History of **Virtualization** (2)

- Full Virtualization: Unmodified Guest OS. Earliest VMs were doing lots of emulation!
  - Hypervisor has to emulate hardware devices in software.
  - Traditionally a costly mode of operation due to emulation overhead. (Or binary translation)



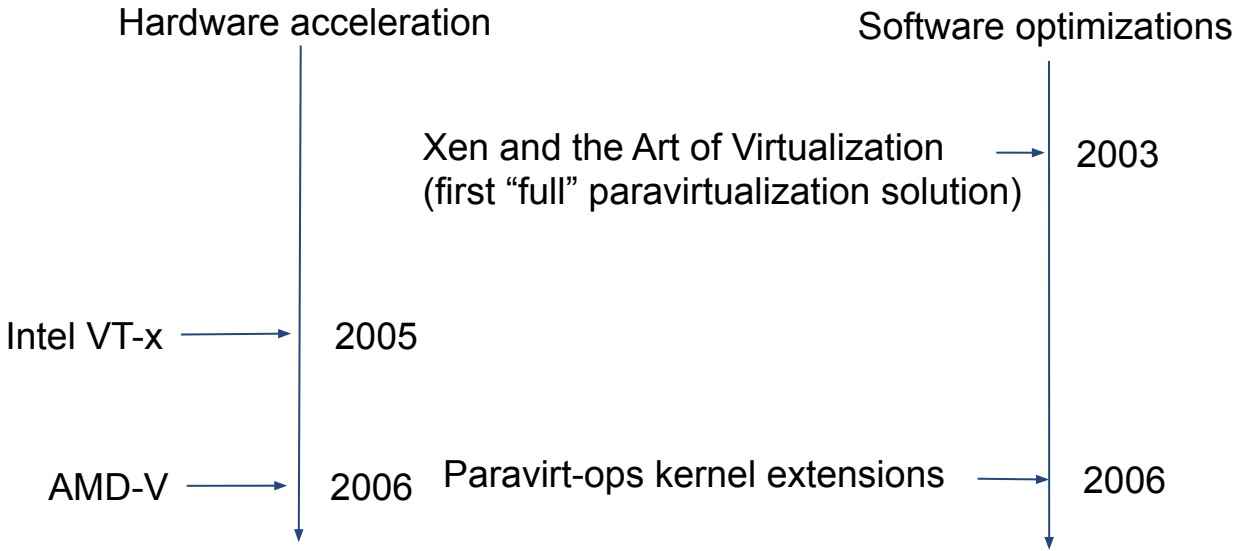
A Brief History of **Virtualization** (3)

- As virtualization made a big comeback in the 2000s, researchers and companies invested heavily in virtualization optimizations (mostly **for x86**).
  - Hardware optimizations
  - Software optimizations
  - Popek and Goldberg virtualization requirements[1]: x86 was NOT designed to be virtualization friendly!
- Optimizations evolved in an uncoordinated way

[1] Popek, G. J.; Goldberg, R. P. (July 1974). "Formal requirements for virtualizable third generation architectures". *Communications of the ACM*. **17** (7): 412–421. doi:10.1145/361011.361073.

A Brief History of **Virtualization** (4)

- Software optimizations were designed without the coming hardware accelerations in mind and vice-versa, sometimes conflicting with each other.



A Brief History of **Virtualization** (5)

- On one hand, **Hardware accelerated virtualization** extension to the x86 ISA make virtualization efficient, mostly by eliminating much of the software-induced overhead.

- On the other hand, **Paravirtualization** is a virtualization technique that relies on the Guest OS being aware that it is being virtualized. Performing certain privileged operations is made efficient by communicating with the Host through a software interface.
  - Almost completely removes the need for the host hypervisor to perform costly hardware emulation.
- VMs can either boot in paravirtualized mode or in fully virtualized mode.

A Brief History of **Virtualization** (6)

- The choice between paravirtualized or fully virtualized is actually not binary, it is a spectrum[2]. More info[3].

Poor Performance
Scope for Improvement
Optimal Performance

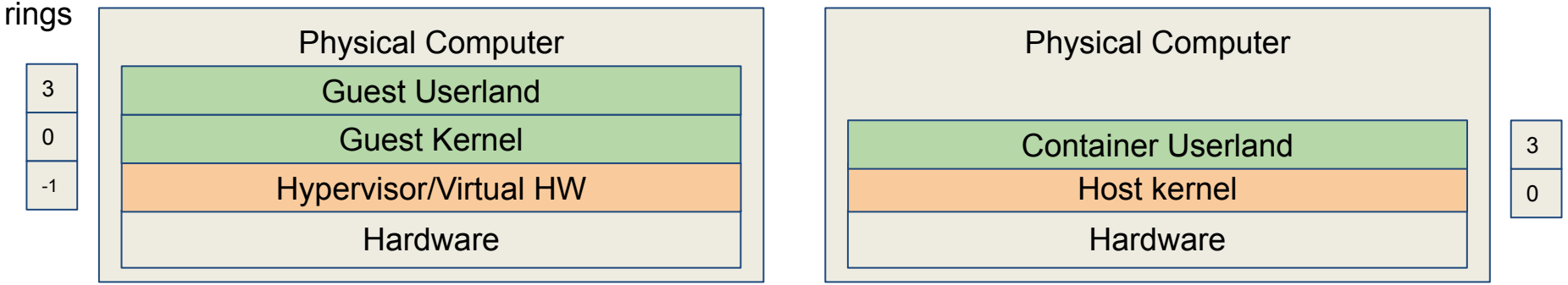
P = Paravirtualized
VS = Software Virtualized (QEMU)
VH = Hardware Virtualized

Shortcut	Mode	With	Disk and Network	Interrupts & Timers	Emulated Motherboard, Legacy Boot	Privileged Instructions, Page Tables
HVM / Fully Virtualized	HVM		VS	VS	VS	VH
HVM + PV drivers	HVM	PV Drivers	P	VS	VS	VH
PVHVM	HVM	PVHVM Drivers	P	P	VS	VH
PV	PV		P	P	P	P

[2] https://wiki.xen.org/wiki/Virtualization_Spectrum
[3] https://wiki.xen.org/wiki/Xen_Project_Software_Overview#PVH_.28x86.29

So what about Containers?

- More lightweight, Isolation happens at the host kernel level.
- Less overhead, since every VM runs its own kernel, especially memory.
- Containers *can not* take advantage of hardware-level isolation without a hypervisor. Isolation relies on software-defined Linux namespaces.
  - Inherently *less secure* than hardware-assisted VMs.
  - Sitting on top of the host kernel means *bare-metal performance*.

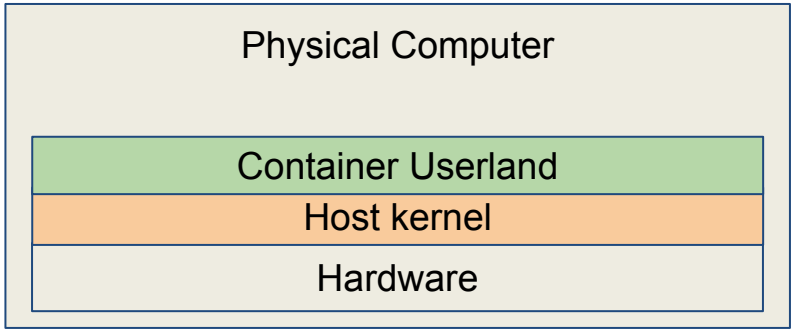


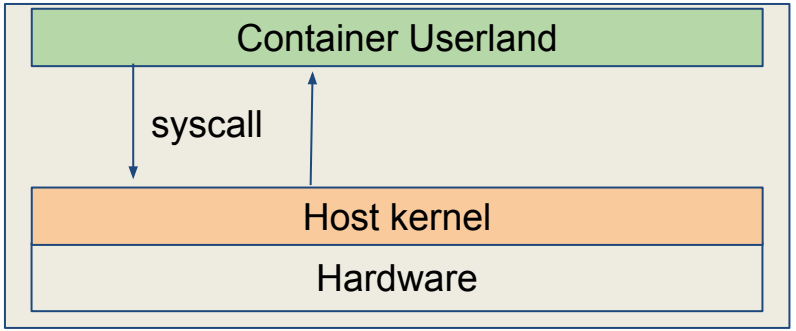
How do Containers achieve Isolation without a Hypervisor?

- Linux namespaces!
- Container processes are spawned and put in namespaces to achieve isolation.
- Linux namespaces (for now, *time* and *syslog* may come soon):
  - Mount
  - UTS (hostname)
  - Inter Process Communication
  - PID
  - Network
  - User
  - Cgroup

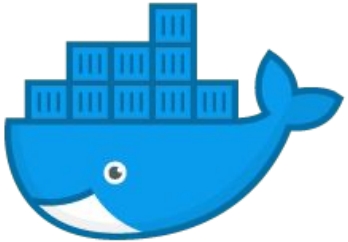
How do Containers achieve Isolation without a Hypervisor?

- Syscalls go directly to the host kernel, processes in containers work in the same way as native ones. No overhead involved! They just run in the context of one namespace or another.





Container runtimes



docker

podman

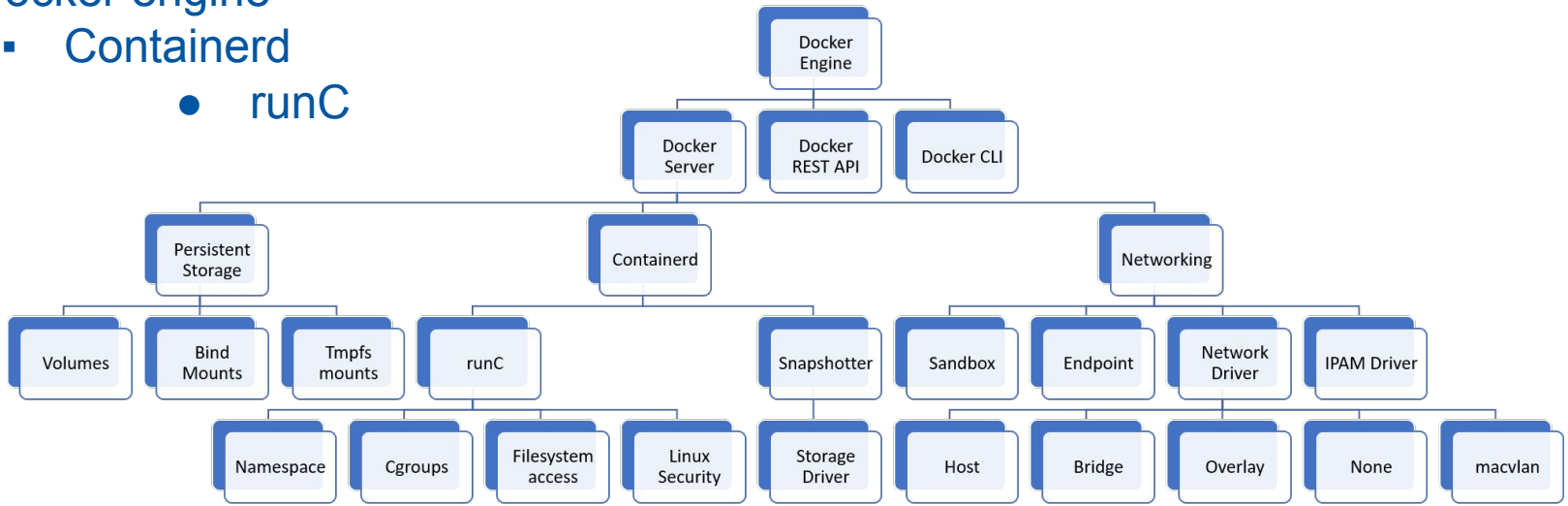
Container runtimes

- Standards: OCI (Open Container Initiative)
- OCI develops specifications that ensure vendor-neutrality.
  - Container Runtime specification (OCI runtime-spec)
  - Container Image specification (OCI image-spec)

- Major vendors contribute and/or implement these standards.

Container runtimes

- Docker engine
  - Containerd
    - runC



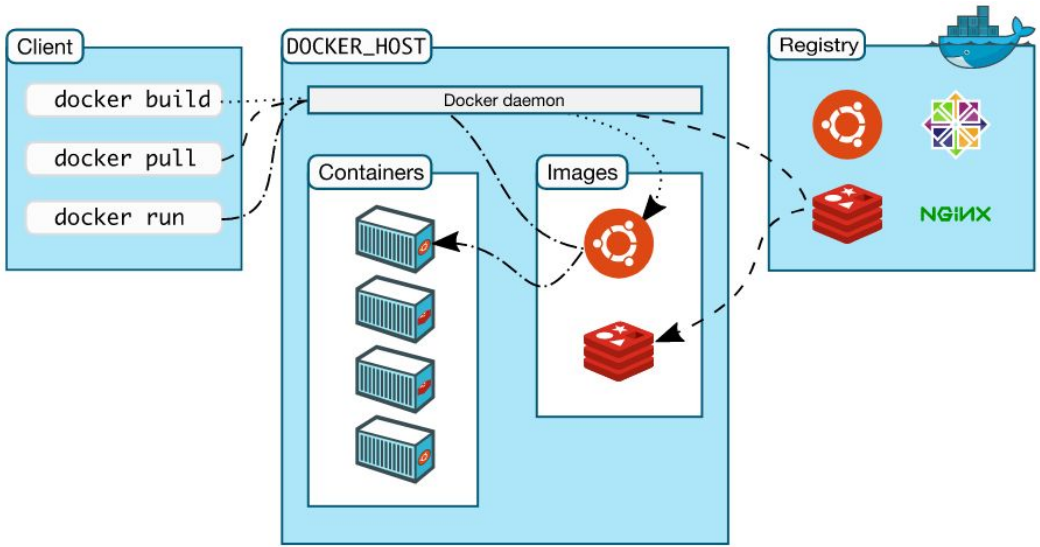
Container use cases

“It works on my computer”

https://www.reddit.com/r/ProgrammerHumo

Docker architecture

- **Privileged** daemon
- Client talks to the daemon to perform operations
- Spawns containers using most of the namespaces. Isolation.



https://docs.docker.com/engine/docker-overview/#docker-architecture

For HPC?

- You will probably want to run the MPI application as the same user who is launching the workload in the cluster
- Container will want to connect to other worker nodes.
- Cluster-specific MPI environment?
- Privileged daemon considered bad for security, especially in multi-tenant environments.

Host users?

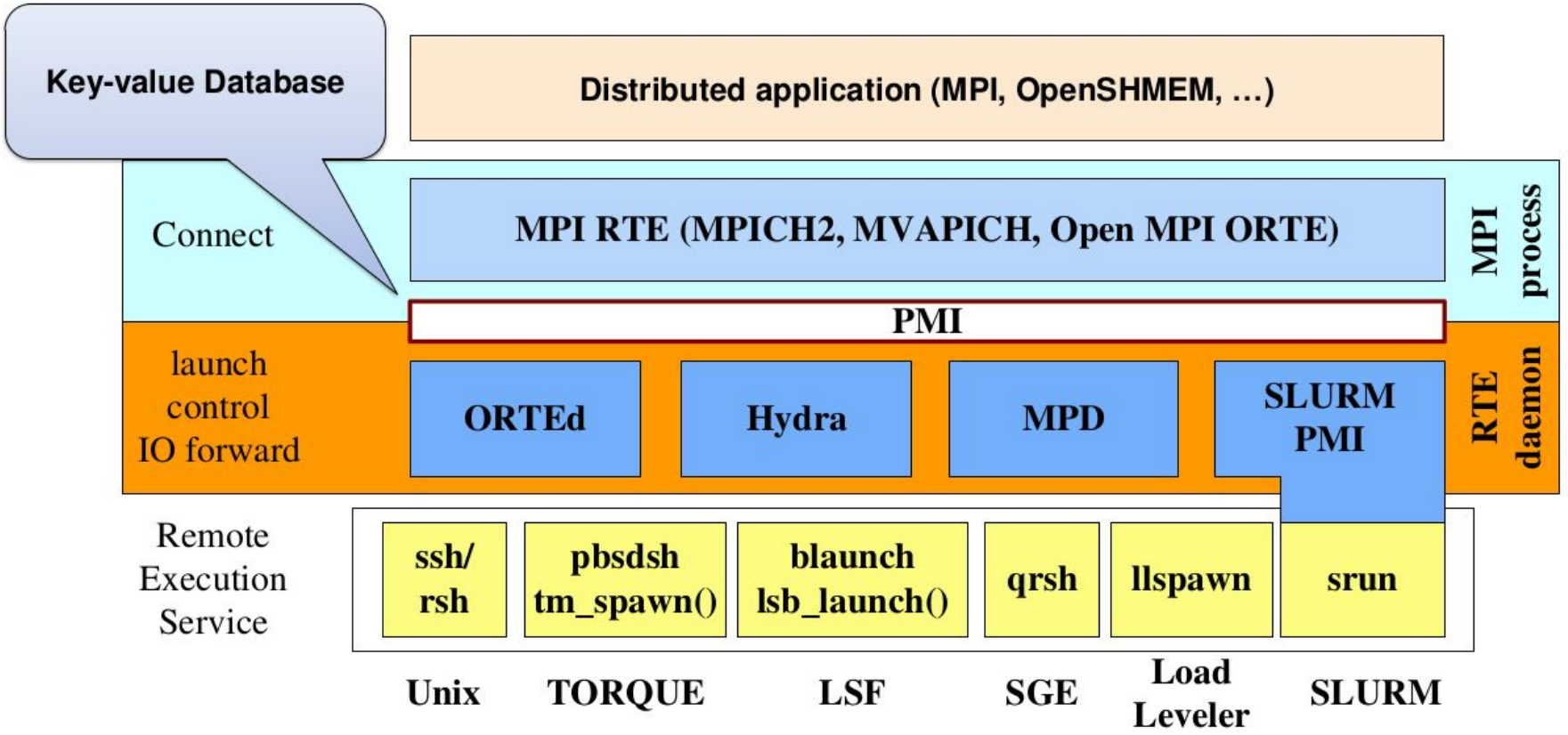
Host network?

MPI?

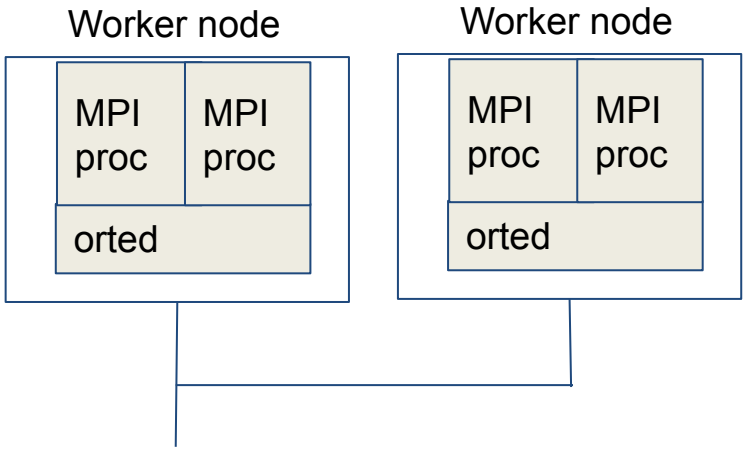
Singularity architecture

- Designed with HPC mind. Completely oriented towards being run in a typical HPC cluster environment.
- No daemon, unprivileged
- By default only isolates the filesystem, by design! Trades-off *isolation* for *simplicity* and *integration*.
  - "Mobility of Compute, Reproducibility, HPC support, and **Security**." Singularity docs.
- User outside the container will be the same user inside the container!
- Connecting to other processes over sockets, IB, or even IPC just works.

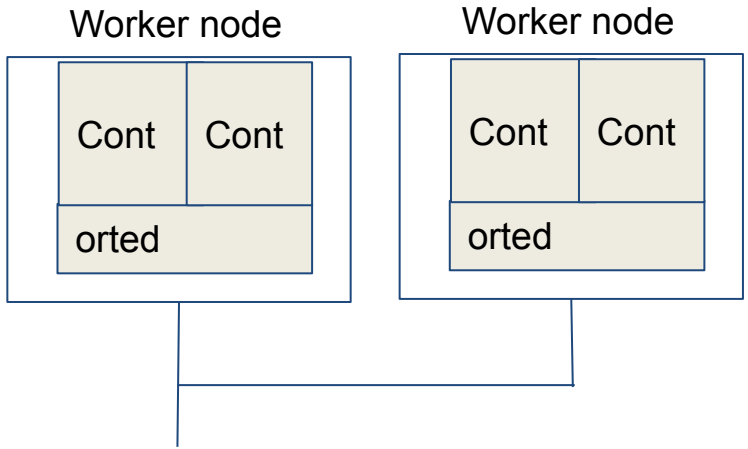
You can almost consider a singularity container as a statically compiled binary… but that wraps all OS dependencies, runs in an isolated filesystem namespace, and is contained in a single portable executable format.



Execution model



mpirun -np 2 ./app arg1 arg2



mpirun -np 2 singularity run ./app.img arg1 arg2

Is the same isolation-usability trade-off possible in other Container runtimes?

- Yes. Sort of.
- It's just more **faff**.
  - Need to explicitly drop namespace creation
  - Not designed to keep the same user context inside the container.

- Watch out! Having a privileged root daemon running "à la Docker" poses the following immediate, non-trivially solvable issues:
  - **Security** in multi-tenant environments. Privilege escalation risk.
  - This model **breaks the process hierarchy** expected by batch systems and MPI itself! Workloads are spawned by the daemon.