Holger Gantikow

Linux Containers

Technology and Runtimes

ISC Workshop, June 2017

Atos

LinkedIn

Holger Gantikow

Senior Systems Engineer at science + computing ag

Stuttgart und Umgebung, Deutschland | IT und Services

133 Kontakte

Aktuell	science + computing ag, science + computing ag, a bull group company
Früher	science + computing ag, Karlsruhe Institute of Technology (KIT) / University of Karlsruhe (TH)
Ausbildung	Hochschule Furtwangen University

Zusammenfassung

Diploma Thesis "Virtualisierung im Kontext von Hocherfügbarkeit" / "Virtualization in the context of High Availability , IT-Know-How, Experience with Linux, especially Debian&Red Hat, Windows, Mac OS X, Solaris, *BSD, HP-UX, AIX, Computer Networking, Network Administration, Hardware, Asterisk, VoIP, Server Administration, Cluster Computing, High Availability, Virtualization, Python Programming, Red Hat Certified System Administrator in Red Hat OpenStack

Current fields of interest:
Virtualization (Xen, ESX, ESXi, KVM), Cluster Computing (HPC, HA), OpenSolaris, ZFS, MacOS X, SunRay ThinClients, virtualized HPC clusters, Monitoring with Check_MK, Admin tools for Android and iOS, Docker / Container in general, Linux 3D VDI (HP RGS, NiceDCV, VMware Horizon, Citrix HDX 3D Pro)

Specialties: Virtualization: Docker, KVM, Xen, VMware products, Citrix XenServer, HPC, SGE, author for Linux Magazin (DE and EN), talks on HPC, virtualization, admin tools for Android and iOS, Remote Visualization

Senior Systems Engineer
science + computing ag
April 2009 – Heute (8 Jahre 3 Monate)

System Engineer
Übersetzung anzeigen

science + computing ag, a bull group company
2009 – Heute (8 Jahre)

Graduand
science + computing ag
Oktober 2008 – März 2009 (6 Monate)

Diploma Thesis: "Virtualisierung im Kontext von Hochverfügbarkeit" - "Virtualization in the context of High Availability"

Intern
Übersetzung anzeigen

Karlsruhe Institute of Technology (KIT) / University of Karlsruhe (TH)
August 2008 – September 2008 (2 Monate)

Research on optimization of computing workflow using Sun Grid Engine (SGE) for MCNPX calculations.

Hochschule Furtwangen University
Dipl. Inform. (FH), Coding, HPC, Clustering, Unix stuff :-)
2003 – 2009

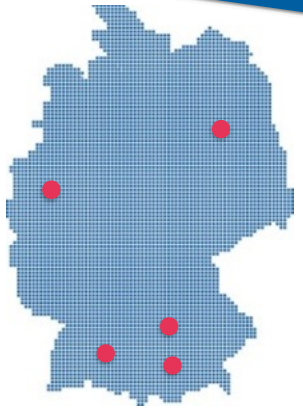
Find me on Linkedin & Xing & Twitter

Rebranded to AtoS!

science + computing ag

Our focus:
IT-services and software for technical computation environments

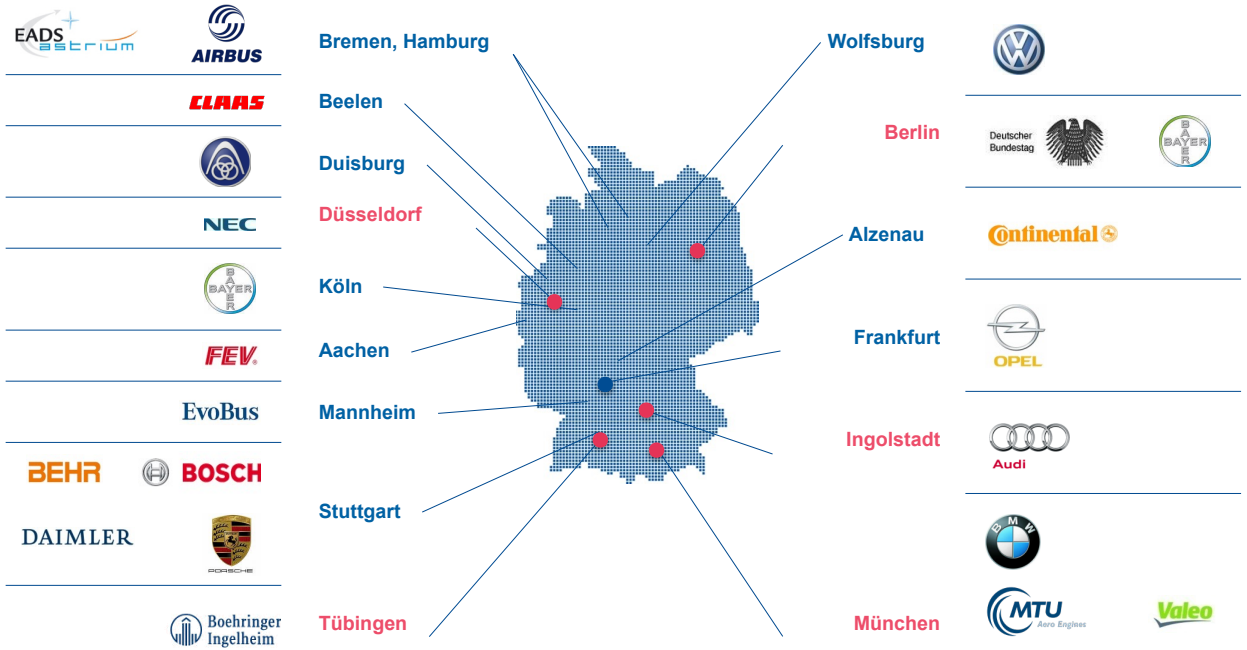
Founded	1989
Offices	Tübingen Munich Berlin Düsseldorf Ingolstadt
Employees	287
Owner	Atos SE (100%) before: Bull
an. turnover (2013)	30,7 Mio. Euro



CAE + Automotive

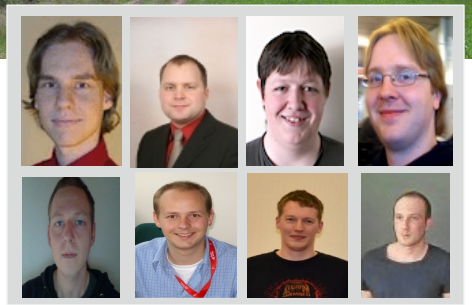
Customers

science + computing
an atos company



Institut für Cloud Computing und IT-Sicherheit (IfCCITS)

previous: ***Cloud Research Lab***



HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

SUCCEED WITH PLYMOUTH UNIVERSITY

facts:

- founded 10/2015
- Head: Prof. Dr. Ch. Reich
- 5 PhDs, 4 Masters, 6 Bachelors
- http://www.wolke.hs-furtwangen.de

research projects:

- Industrie 4.0 (security, data analysis)
- EU: A4Cloud („accountable Cloud“)
- PET Platform as a Service for Ambient Assisted Living Applications

research topics:

- Distributed Systems
- IT Security
- Cloud Computing
- Industry 4.0; IoT

Agenda

1. **Introduction**
2. **Container Runtimes**
3. **Summary**

Introduction

What are Containers good for?

isolating dependencies
+ conflicting requirements
+ dealing with legacy
+ ship code

Workflow*workflow*
+ Reproducibility
„Frozen Environment“
+Flexibility @HPC/Datacenter

Source: www.critic.co.nz/files/article-3423.jpg

Performance

close to bare-metal

An Updated Performance Comparison of Virtual Machines and Linux Containers

Wes Felter, Alexandre Ferreira, Ram Rajamony, Juan Rubio
IBM Research, Austin, TX
{wmf, apferrei, rajamony, rubioj}@us.ibm.com

***Abstract*—Cloud computing makes extensive use of virtual machines (VMs) because they permit workloads to be isolated from one another and for the resource usage to be somewhat controlled. However, the extra levels of abstraction involved in virtualization reduce workload performance, which is passed on to customers as worse price/performance. Newer advances in container-based virtualization simplifies the deployment of applications while continuing to permit control of the resources allocated to different applications.**

In this paper, we explore the performance of traditional virtual machine deployments, and contrast them with the use of Linux containers. We use a suite of workloads that stress CPU, memory, storage, and networking resources. We use KVM as a representative hypervisor and Docker as a container manager. Our results show that containers result in equal or better performance than VMs in almost all cases. Both VMs and containers require tuning to support I/O-intensive applications. We also discuss the implications of our performance results for future cloud architectures.

I. Introduction

Virtual machines are used extensively in cloud computing. In particular, the state-of-the-art in Infrastructure as a Service (IaaS) is largely synonymous with virtual machines. Cloud platforms like Amazon EC2 make VMs available to customers and also run services like databases inside VMs. Many Platform as a Servive (PaaS) and Software as a Service (SaaS) providers are built on IaaS with all their workloads running inside VMs. Since virtually all cloud workloads are currently running in VMs, VM performance is a crucial component of overall cloud performance. Once a hypervisor has added overhead, no higher layer can remove it. Such overheads then become a pervasive tax on cloud workload performance. There have been many studies showing how VM execution compares to native execution [30, 33] and such studies have been a motivating factor in generally improving the quality of VM technology [25, 31].

Container-based virtualization presents an interesting alternative to virtual machines in the cloud [46]. Virtual Private Server providers, which may be viewed as a precursor to cloud computing, have used containers for over a decade but many of them switched to VMs to provide more consistent performance. Although the concepts underlying containers such as namespaces are well understood [34], container technology languished until the desire for rapid deployment led PaaS providers to adopt and standardize it, leading to a renaissance in the use of containers to provide isolation and resource control. Linux is the preferred operating system for the cloud due to its zero price, large ecosystem, good hardware support, good performance, and reliability. The kernel namespaces feature needed to implement containers in Linux has only become mature in the last few years since it was first discussed [17]. Within the last two years, Docker [45] has emerged standard runtime, image format, and build system for containers.

This paper looks at two different ways of achievi source control today, viz., containers and virtual ma and compares the performance of a set of workloads i environments to that of natively executing the worklo hardware. In addition to a set of benchmarks that different aspects such as compute, memory bandwidth, ory latency, network bandwidth, and I/O bandwidth, w explore the performance of two real applications, viz., and MySQL on the different environments.

Our goal is to isolate and understand the overhead duced by virtual machines (specifically KVM) and cont (specifically Docker) relative to non-virtualized Linu expect other hypervisors such as Xen, VMware ESX Microsoft Hyper-V to provide similar performance to given that they use the same hardware acceleration fe Likewise, other container tools should have equal perfor to Docker when they use the same mechanisms. We evaluate the case of containers running inside VMs or running inside containers because we consider such virtualization to be redundant (at least from a perfor perspective). The fact that Linux can host both VM containers creates the opportunity for an apples-to-apple parison between the two technologies with fewer confou variables than many previous comparisons.

We make the following contributions:

- We provide an up-to-date comparison of native tainer, and virtual machine environments using hardware and software across a cross-section of esting benchmarks and workloads that are relev the cloud.
- We identify the primary performance impact of virtualization options for HPC and server work
- We elaborate on a number of non-obvious pr issues that affect virtualization performance.
- We show that containers are viable even at the of an entire server with minimal performance i

The rest of the paper is organized as follows. Section scribes Docker and KVM, providing necessary backgro understanding the remainder of the paper. Section III des and evaluates different workloads on the three environ We review related work in Section IV, and finally, Sect concludes the paper.

Summary

"In general, Docker equals or exceeds KVM performance in every case we tested. [...]

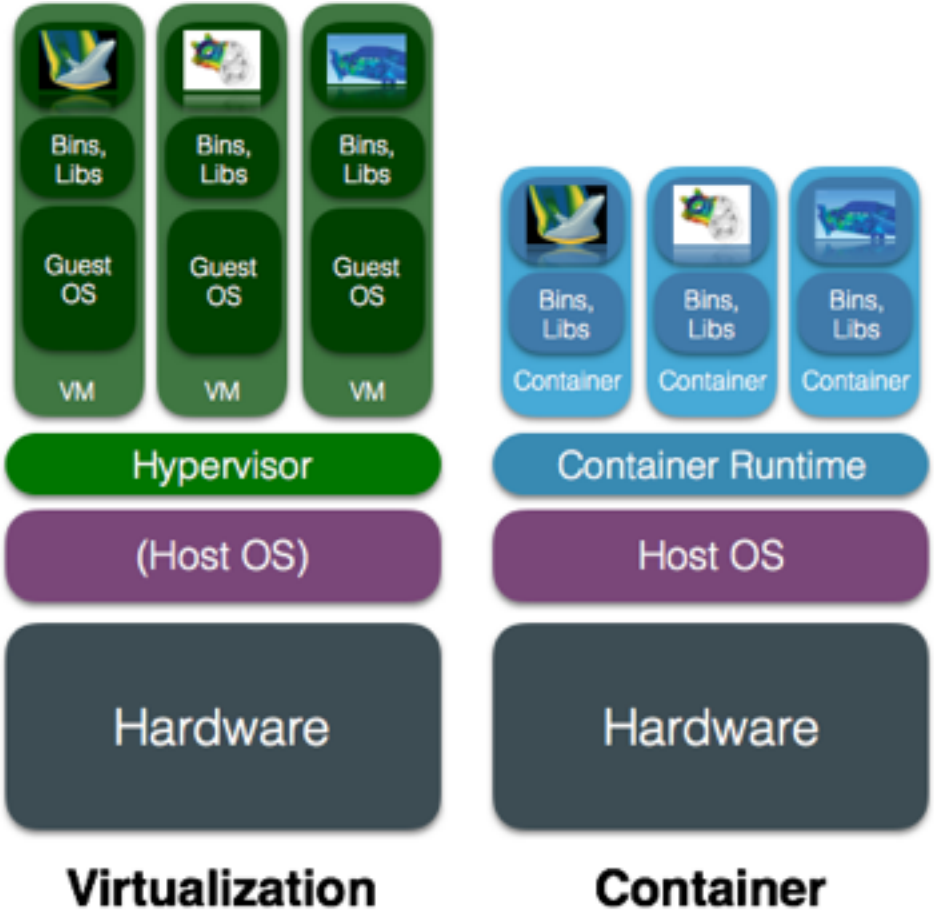
Even using the fastest available forms of paravirtualization, KVM still adds some overhead to every I/O operation [...].

Thus, KVM is less suitable for workloads that are latency-sensitive or have high I/O rates.

Container vs. bare-metal:
Although **containers themselves have almost no overhead**, **Docker is not without performance gotchas**. **Docker volumes** have noticeably better performance than files stored in AUFS. **Docker's NAT also introduces overhead** for work- loads with high packet rates.
These features represent a **tradeoff between ease of management and performance** and should be considered on a case-by-case basis.

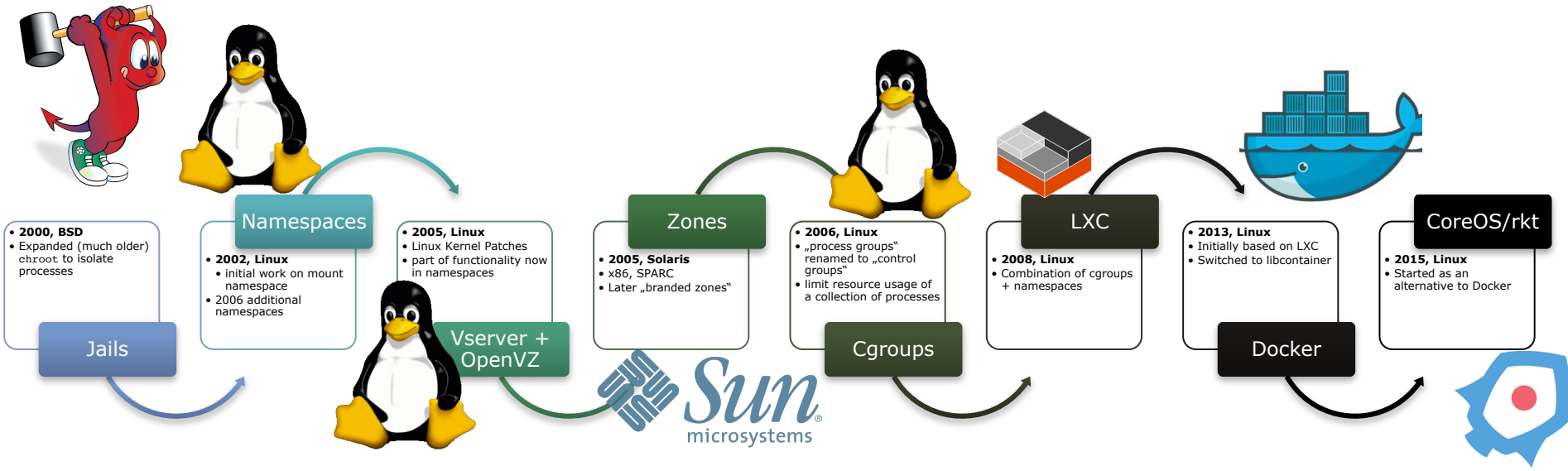
Source: Felter et al. (2014). An updated performance comparison of virtual machines and linux containers.

What is different with Containers?


Bins, Libs
Guest OS
VM
Bins, Libs
Guest OS
VM
Bins, Libs
Guest OS
VM
Hypervisor
(Host OS)
Hardware
Virtualization
Bins, Libs
Container
Bins, Libs
Container
Bins, Libs
Container
Container Runtime
Host OS
Hardware
Container

Source: http://cdn.meme.am/instances/53646903.jpg

Evolution of *OS-level virtualization*



Hypervisor-based virtualization

1999 VMware Workstation 1.0
2001 ESX 1.0 & GSX 1.0
2003 Xen 1st public release
2006 KVM (2.6.10)

Built on existing technology

already included in the Linux Kernel

Proof

How are they implemented? Let's look in the kernel source!

- Go to LXR
- Look for "LXC" → zero result
- Look for "container" → 1000+ results
- Almost all of them are about data structures
  or other unrelated concepts like “ACPI containers”
- There are some references to “our” containers
  but only in the documentation

DockerCon EU 2015

Source: https://www.youtube.com/watch?v=sK5i-N34im8 &&
https://de.slideshare.net/jpetazzo/cgroups-namespaces-and-beyond-what-are-containers-made-from-dockercon-europe-2015

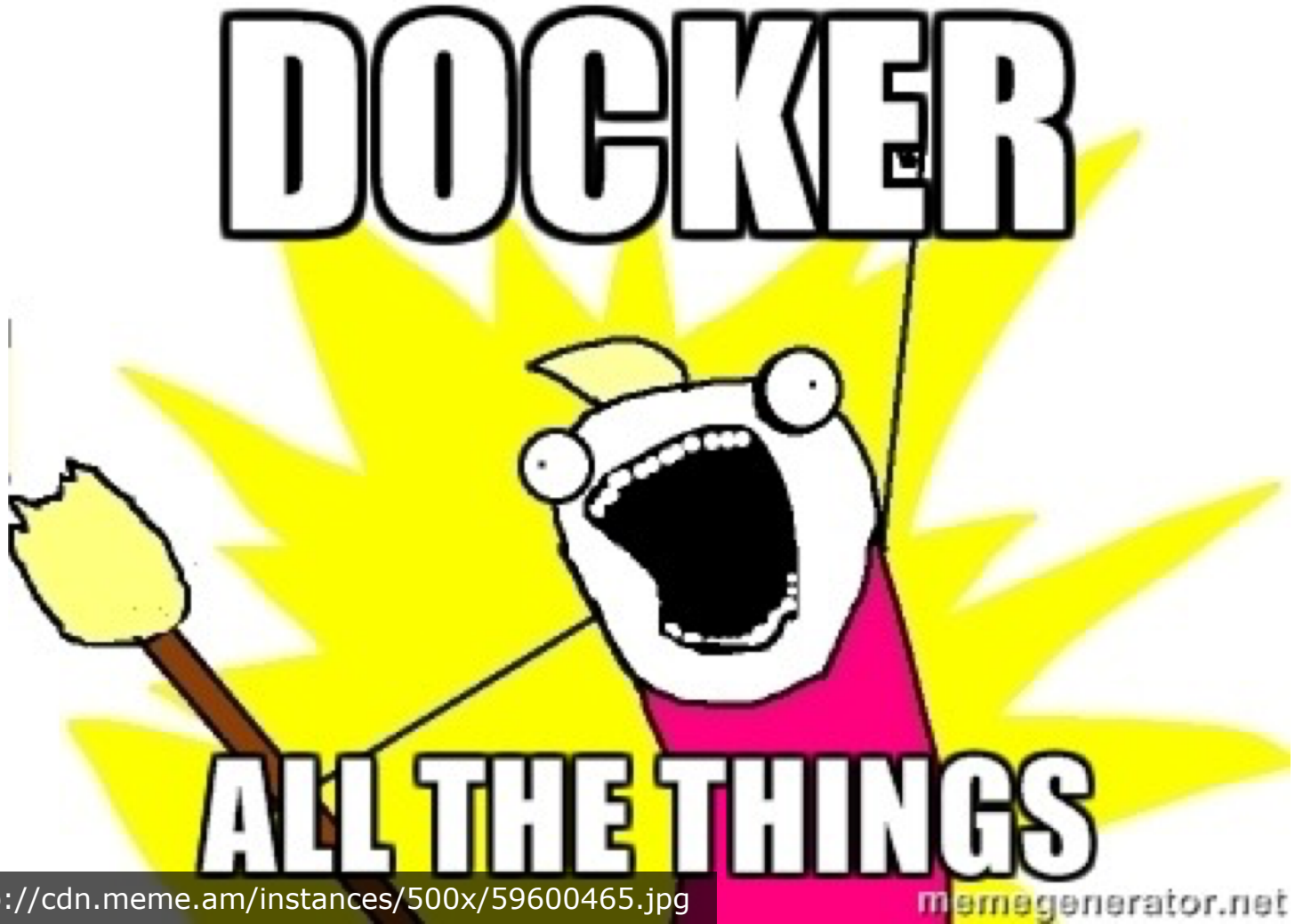
Container = Namespaces + cgroups

- Both Kernel features – „Containers“ use these + some „glue“ around it
  - **Namespaces** : certain sub systems *ns-aware – isolated operation*
  - **Cgroups**: certain resources controlable – limits for resource usage

Namespace	Description	Controller	Description
pid	Process ID	**blkio**	Access to block devices
net	Network Interfaces, Routing Tables, …	**cpu**	CPU time
ipc	Semaphores, Shared Memory, Message Queues	**devices**	Device access
mnt	Root and Filesystem Mounts	**memory**	Memory usage
uts	Hostname, Domainname	**net_cls**	Packet classification
user	UserID and GroupID	**net_prio**	Packet priority


2
Container Runtimes

Docker



Source: http://cdn.meme.am/instances/500x/59600465.jpg

What is *Docker*?

It depends...
on the time
Engine -> Company -> Platform

From engine to platform

Docker Hub

Docker Toolbox

Docker Compose

Docker Swarm

Docker Machine

Docker Universal Control Plane

Docker Trusted Registry

Docker Cloud

Docker Enterprise Edition

Docker „XYZ“ ;)

More on that later :)

The new old Microsoft?

Swarm, Platform lock in (Enterprise Edition), …

Hopefully not

Decoupling the plumbing

runC / containerd

Docker >= 1.11 is based on runC and containerd
Effort to break Docker into smaller reusable parts

runC

- **runC -** low-level container runtime / executor
  - CLI tool for spawning + running containers
  - Implementation of the OCI specification
  - Built on Libcontainer (performs the container isolation primitives for the OS)
  - Can be integrated into other systems – does not require a daemon
  - But not really end-user friendly

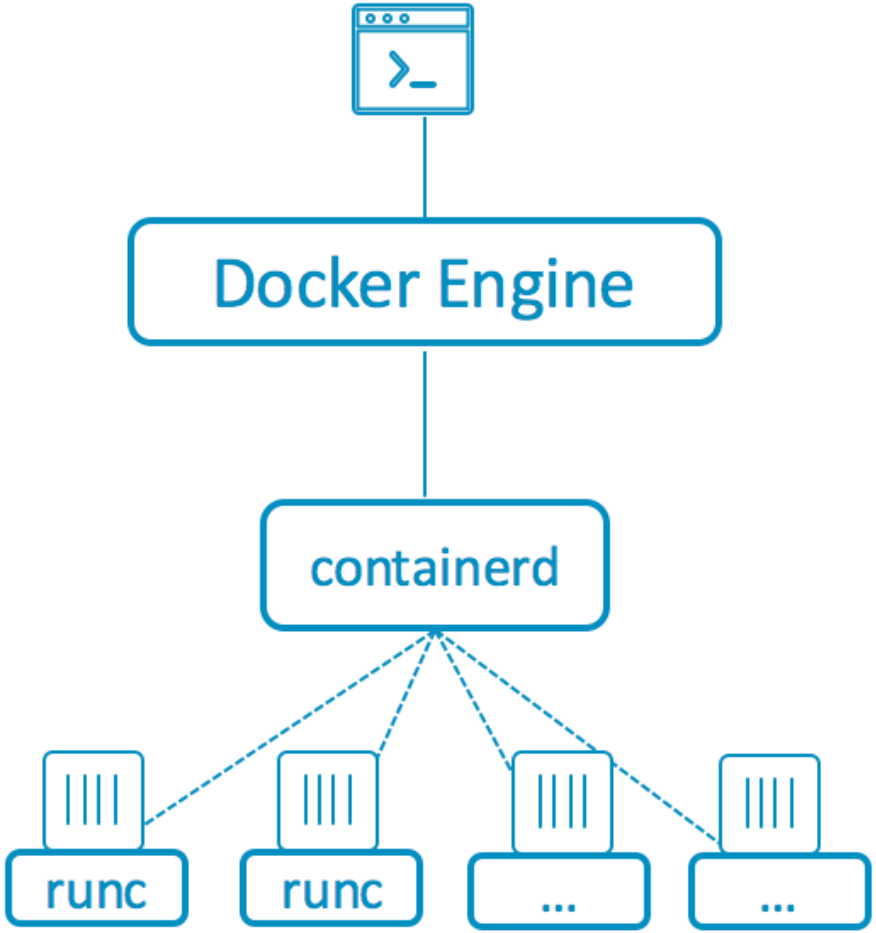
- Given to the **OCI (Open Container Initiative)**
  - Founded 2015 by Docker and others. 40+ members
  - Aims to establish common standards and avoid potential fragmentation
  - Two specifications for interoperability: Runtime + Image (Both supported)

Links: https://github.com/opencontainers/runc && https://opensource.com/life/16/8/runc-little-container-engine-could && https://www.opencontainers.org/

containerd

- **Containerd** - daemon to control runC
  - Sticker says: „small, stable, rock-solid container runtime“
  - Can be updated without terminating containers
  - Can manage the complete container lifecycle of its host system
    - image transfer + storage, container execution + supervision, ...
  - Designed to be embedded into a larger system, not directly for end-users
- Donated to the **CNCF (Cloud Native Computing Foundation)** – as is rkt ;)
  - Linux Foundation project to accelerate adoption of microservices, containers and cloud native apps.

Links: https://github.com/docker/containerd/ && https://www.youtube.com/watch?v=VWuHWfEB6ro && https://www.cncf.io/



Same Docker UI and commands

User interacts with the Docker Engine

Engine communicates with containerd

containerd spins up runc or other OCI compliant runtime to run containers

Source: https://blog.docker.com/2016/04/docker-engine-1-11-runc/

Moby

*„Moby is an open framework created by Docker to **assemble specialized container systems without reinventing the wheel**. It provides a **“lego set” of dozens of standard components** and a framework for assembling them into custom platforms.“*

Audience

Moby is recommended for anyone who wants to assemble a container-based system. This includes:

- Hackers who want to customize or patch their Docker build
- System engineers or integrators building a container system
- Infrastructure providers looking to adapt existing container systems to their environment
- Container enthusiasts who want to experiment with the latest container tech
- Open-source developers looking to test their project in a variety of different systems
- Anyone curious about Docker internals and how it's built

Moby is NOT recommended for:

- Application developers looking for an easy way to run their applications in containers. We recommend Docker C instead.
- Enterprise IT and development teams looking for a ready-to-use, commercially supported container platform. W recommend Docker EE instead.
- Anyone curious about containers and looking for an easy way to learn. We recommend the docker.com website instead.

Security

Seccomp Profiles >=Docker 1.10

GitHub, Inc. github.com/docker/docker/blob

Significant syscalls blocked by the default profile

Docker's default seccomp profile is a whitelist which specifies the calls that are allowed. The table below lists the significant (but not all) syscalls that are effectively blocked because they are not on the whitelist. The table includes the reason each syscall is blocked rather than white-listed.

Syscall	Description
`acct`	Accounting syscall which could let containers disable their own resource limits or process accounting. Also gated by `CAP_SYS_PACCT`.
`add_key`	Prevent containers from using the kernel keyring, which is not namespaced.
`adjtimex`	Similar to `clock_settime` and `settimeofday`, time/date is not namespaced.
`bpf`	Deny loading potentially persistent bpf programs into kernel, already gated by `CAP_SYS_ADMIN`.
`clock_adjtime`	Time/date is not namespaced.
`clock_settime`	Time/date is not namespaced.
`clone`	Deny cloning new namespaces. Also gated by `CAP_SYS_ADMIN` for CLONE_* flags, except `CLONE_USERNS`.
`create_module`	Deny manipulation and functions on kernel modules.
`delete_module`	Deny manipulation and functions on kernel modules. Also gated by `CAP_SYS_MODULE`.
`finit_module`	Deny manipulation and functions on kernel modules. Also gated by `CAP_SYS_MODULE`.
`get_kernel_syms`	Deny retrieval of exported kernel and module symbols.
`get_mempolicy`	Syscall that modifies kernel memory and NUMA settings. Already gated by `CAP_SYS_NICE`.

Source: https://docs.docker.com/engine/security/seccomp/#significant-syscalls-blocked-by-the-default-profile
@Rkt: https://coreos.com/rkt/docs/latest/seccomp-guide.html

User Namespaces
>=Docker 1.10

ContainerCon 2015

"Phase 1" Usage Overview

```
# docker daemon --root=2000:2000 ...
drwxr-xr-x root:root  /var/lib/docker
drwx------ 2000:2000  /var/lib/docker/2000.2000
```

Start the daemon with a remapped root setting (in this case uid/gid = 2000/2000)

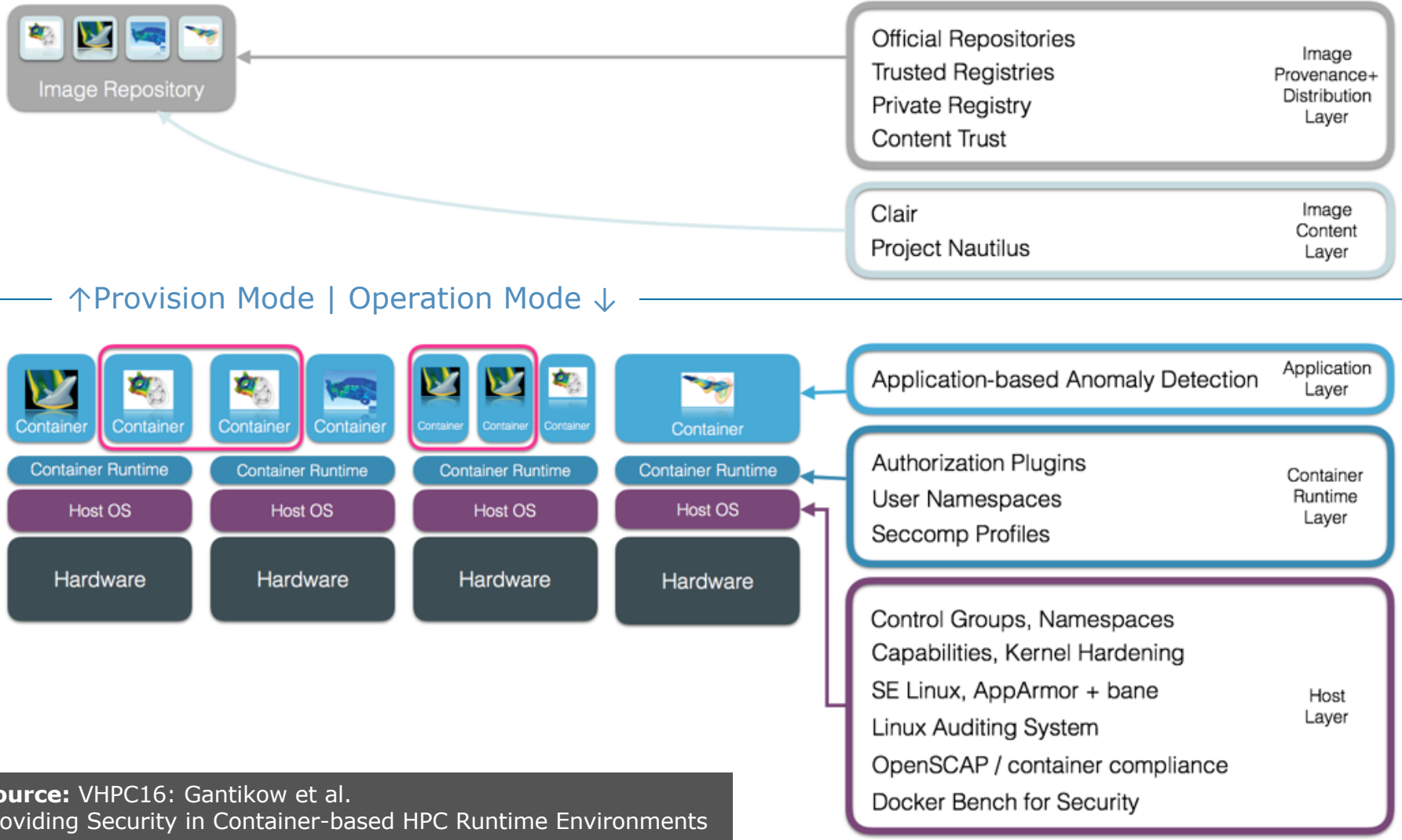
```
$ docker run -ti --name fred --rm busybox /bin/sh
/ # id
uid=0(root) gid=0(root) groups=10(wheel)
```

Start a container and verify that inside the container the uid/gid map to root (0/0)

```
$ docker inspect -f '{{ .State.Pid }}' fred
8851
$ ps -u 2000
  PID TTY          TIME CMD
 8851 pts/7    00:00:00 sh
```

You can verify that the container process (PID) is actually running as user 2000

Link: https://events.linuxfoundation.org/sites/events/files/slides/User Namespaces - ContainerCon 2015 - 16-9-final_0.pdf



Source: VHPC16: Gantikow et al.
Providing Security in Container-based HPC Runtime Environments

In the future…

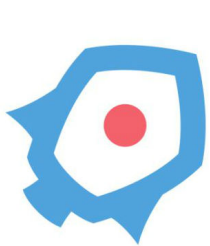
- **Fully unprivileged containers**
  - Container startup by non-root Users without privilege escalation
  - Lot‘s of development going on, early patch for runC, but not finished yet

- **More activation of security features** *„by default“*

- **Phase 2 User Namespaces:**
  - custom namespaces per Container
  - Goal: multi-tenant environment with uid/gid mapping per customer
  - Support at upstream kernel already available

Link: Rootless containers - https://github.com/opencontainers/runc/pull/774 && https://www.cyphar.com/blog/post/rootless-containers-with-runc

Alternatives to Docker



rkt

Rocket / rkt

Docker is „fundamentally flawed“
- CoreOS CEO Alex Polvi

Key facts - rkt

- **Not a Docker fork**
  - Started by the disappointed CoreOS team as Docker moved away from a *simple building block* to a plattform

- Mission: *build a top-notch systemd oriented container runtime for Linux*
  - Not attempting to become a wider containerization platform
  - Reached 1.0 in 02/2016 – production ready? Current: v1.27.0

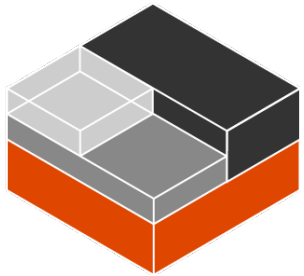
- Features:
  - Sticker says „Secure by default", besides *daemon-less* including
    - Support for executing pods with KVM hypervisor
    - SELinux support, signature validation (as in Docker)
  - Can run Docker images (-> appc, Docker, OCI)

Key facts II - rkt

- Very Linux oriented
  - No Windows / MacOS „version“
    - using Docker easier vor Devs with tools like “Docker for Mac/Windows“
  - Process model is more Linux-like than Docker‘s

- 3rd party support:
  - Images: worse than Docker, but can run Docker images
  - Schedulers (Kubernetes, ...): good

- Also project at the CNCF
  - *Merger* unlikely, would rather lead to a third option
    - (containerd &OCI compatible runtime + runc)



LXC/LXD

"Containers which offer an environment as close to possible as the one you'd get from a VM but without the overhead that comes with running a separate kernel and simulating all the hardware."
– LXC Documentation

Key facts - LXC

- Idea for Linux Containers (LXC) started with Linux Vservers
- Developers from IBM started the LXC project in 2008, currently led by Ubuntu
- Had support for user namespaces ages before Docker ;)
- Often considered *‚more complicated to use*"
- Concept much closer to VMs than Docker
  - *Operating System containerization* vs *Application containerization*
  - Less living the „one application per container" mantra

More on that later? ;)

Key facts - LXD

- LXC „hypervisor“, originally developed by Ubuntu
- Offers integration with OpenStack
- Manages containers through a REST APIs
- Like “*Docker for LXC*", with similar command line flags, support for image repositories and other container management features

systemd-nspawn

Key facts - systemd-nspawn

- Limited – but might be sufficient in some cases
  - "namespace spawn" - it only handles process isolation
  - no resource isolation like memory, CPU, etc.
- Does not download or verify images by itself
- Less enduser-friendly than rkt or Docker
  - More „like using runc with less features“
  - No „manager“ like containerd

Alternatives for HPC

Shifter

More on that later? :)

Singularity

More on that later :)

3 Summary

Decision helper

Runtime	Reason
Docker	You want a platform, if needed with support You want one solution for different use cases
Docker lowlevel	You want to integrate Docker into something „bigger“
Rkt	You want a general purpose alternative to Docker You get confused by Docker
LXC	You want system-, not application-container
systemd-nspawn	You want *Docker lowlevel* with much lesser features
Shifter	Your other computer is a Cray and you want something like containers
Singularity	You do HPC and only HPC

Summary

- *Containers* are based on existing Linux kernel features

- Several viable options exists for containerizing workloads
  - rkt now provides a viable alternative to Docker
    - Linux centric
    - Strong competitor keeps monopolists sharp :)
  - Breaking Docker into smaller reusable parts makes sense
  - LXC for containerizing OS instead of application

- Docker‘s security has strongly improved

- But the war is won. And now moving to a different layer

Thanks

For more information please contact:
Holger Gantikow
T +49 7071 94 57-503
h.gantikow@atos.net
h.gantikow@science-computing.de

Atos