State of Containers

Convergence of Big Data, AI and HPC
Technology ReCap

Comparison of Hypervisor and Container Virtualization
Stacked View

Traditional Virtualization

- VM1
  - appA
  - Userland
  - Kernel
  - Services
  - Host Kernel
  - Hardware

- VM2
  - appB
  - Userland
  - Kernel
  - Services
  - Host Kernel
  - Hardware

- Operational Abstraction
  - end-user does not care...

Container Virtualization

- Cnt1
  - Services
  - appA
  - Userland
  - Hardware

- Cnt2
  - Services
  - appB
  - Userland
  - Hardware

VM-shortcuts (PVM, pci-passthrough)

Operational Abstraction
end-user does not care...
Interface View

From Application to Kernel

Traditional Virtualization

Container Virtualization
Technology ReCap

Container Technology 101
Container Namespaces

A starting container gets his own namespaces to keep track of resources. But can share namespaces with other containers or even the host.
CGroups

While namespaces isolate,
Control Groups constraint resources.
Storage Driver

Handling OverlayFS

The storage driver controls how images and containers are stored and managed on your Docker host.

```
FROM ubuntu:16.04
ENV DEBIAN_FRONTEND noninteractive
RUN apt-get update
RUN apt-get install -y libcuda1-384 \\
    && mv /usr/lib/nvidia-384/ /usr/lib/nvidia/
VOLUME ["/usr/lib/nvidia/"]
CMD ["/usr/lib/nvidia/bin/nvidia-smi", "-L"]
```

$q$ docker run -ti --read-only qnib/uplain-cuda 
root@0a53fde7d806:/# echo ${DEBIAN_FRONTEND} | tee /root/test.txt
A /root/test.txt
$
Seccomp (SELinux, AppArmor)

- While namespaces isolate,
- Control Groups constraint resources,
- seccomp filters syscalls (sane default).
- (SELinux/AppArmor allow for system-wide filters)

### 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.

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>acct</td>
<td>Accounting syscall which could let containers disable their own resource limits or process accounting. Also gated by CAP_SYS_Pacct.</td>
</tr>
<tr>
<td>add_key</td>
<td>Prevent containers from using the kernel keyring, which is not namespace.</td>
</tr>
<tr>
<td>adjtime</td>
<td>Similar to clock_settime and settimeofday, time/date is not namespace. Also gated by CAP_SYS_TIME.</td>
</tr>
<tr>
<td>bpf</td>
<td>Deny loading potentially persistent bpf programs into kernel, already gated by CAP_SYS_ADMIN.</td>
</tr>
<tr>
<td>clock_adjtime</td>
<td>Time/date is not namespace. Also gated by CAP_SYS_TIME.</td>
</tr>
<tr>
<td>clock_settime</td>
<td>Time/date is not namespace. Also gated by CAP_SYS_TIME.</td>
</tr>
<tr>
<td>clone</td>
<td>Deny cloning new namespaces. Also gated by CAP_SYS_ADMIN for CLONE_* flags, except CLONE_USERS.</td>
</tr>
<tr>
<td>create_module</td>
<td>Deny manipulation and functions on kernel modules. Obsolete. Also gated by CAP_SYS_MODULE.</td>
</tr>
<tr>
<td>delete_module</td>
<td>Deny manipulation and functions on kernel modules. Also gated by CAP_SYS_MODULE.</td>
</tr>
<tr>
<td>finit_module</td>
<td>Deny manipulation and functions on kernel modules. Also gated by CAP_SYS_MODULE.</td>
</tr>
<tr>
<td>get_kernel_sym</td>
<td>Deny retrieval of exported kernel and module symbols. Obsolete.</td>
</tr>
</tbody>
</table>

https://docs.docker.com/engine/security/seccomp/
Architecture ReCap

Open-Source Container Technology 101
Runtime

runc + containerd

- **runc**
  CLI tool for spawning and running containers according to the OCI specification

- **Shifter / Charlie Cloud / Singularity**
  are not OCI compliant runtimes…

```json
{
  "process": {
    "terminal": true,
    "consoleSize": {
      "height": 25,
      "width": 80
    },
    "user": {
      "uid": 1,
      "gid": 1,
      "additionalGids": [5, 6]
    }
  }
}
```
Runtime

runc + containerd

- **runc**
  CLI tool for spawning and running containers according to the OCI specification

- **containerd**
  An industry-standard container runtime with an emphasis on simplicity, robustness and portability.
Architecture on Linux

Docker Engine

- REST interface
- libcontainerd
- libnetwork
- storage
- plugins

- containerd + runc

HPC-runtime workarounds

Operating System

- Control Groups (cgroups)
- Namespaces (mnt,pid,ipc,...)
- Layer Capabilities AUFS,overlay,...
- Other OS Functionality
Architecture on Windows

Docker Engine
- REST interface
- libcontainer
- libnetwork
- storage
- plugins

Host Compute Service
- Control Groups
- Namespaces
- Layer Capabilities
- Other OS Functionality

Docker Client
- Docker Compose
- Docker Registry
- Docker Swarm/K8s
HPC Challenges

Host-Agnostic vs. Host-Specific
Shared Systems

Scientist self-service

Service Cluster

Compute Cluster

Engine rank0
Engine rank1
Engine rank2
Engine rankN

Create Data

Display Data

/home/
/scratch/
/proj/

Storage
Shared File-Systems

Compute Jobs pin the user

Compute jobs need to be imperative, as processes work closely together.

➔ UID:GID ownership can be handled within container
Honor Shared Environments

Example

Works as designed - host-agnostic!

➔ Best-practice (as of today) for containers suggest to do not trust the UID within the container at all!

root@n40l:/home# ls -l
total 16
drwx------  2 alice alice  4096 Feb 17 11:12 alice
drwx------  2 bob  bob   4096 Feb 17 11:11 bob
root@n40l:~# su - bob
bob@n40l:~$ docker run -ti --rm
   -v /home/:/home/ --user=$(id -u alice):$(id -g alice)
   ubuntu touch /home/alice/bob_was_here

bob@n40l:~$ logout
root@n40l:~# ls -l /home/alice/
total 0
-rw-r--r-- 1 alice alice  0 Feb 17 11:38 bob_was_here
root@n40l:~#
Honor Shared Environments [proof-of-concept]

Manipulating API payload via proxy

$ docker run -v /var/run/:/var/run/ -ti --rm qnib/doxy:gpu doxy --proxy-socket=/var/run/hpc.sock --pattern-key=hpc \
 --pin-user --user 1003:1003

2018/02/17 10:47:27 [II] Start Version: 0.2.4
2018/02/17 10:47:27 [doxy] Listening on /var/run/hpc.sock
2018/02/17 10:47:27 Serving proxy on '/var/run/hpc.sock'

bob@n40l:~$ docker -H unix:///var/run/hpc.sock create -v /home/:/home/ \ 
 --user=$(id -u alice):$(id -g alice) ubuntu touch /home/alice/bob_tried
2117ed54e6063928db5c3ed7688a3ab96a3c60fcf3b54407fb27336dfe14e9de

Alter User setting - Overwriting User with '1003:1003', was '1002:1002'
Mounts: [/home/:/home/]

charlie@n40l:~$ docker start -a 2117ed54e6063928db5c3ed7688a3ab96a3c60fcf3b54407fb27336dfe14e9de
touch: cannot touch '/home/alice/bob_tried': Permission denied
Host-Agnostic vs. Host-Specific

kernel-bypassing devices

- To use kernel-bypassing devices and drivers have to be present into a container
  - drivers can be part of the container, which bloats the container
  - a local volume on the host, which adds/augments the library into the container
- These devices/drivers might be host-specific (e.g. different GPUs per host require different mappings)
Kernel-bypassing devices [proof-of-concept]

Manipulating API payload via proxy

```bash
p2.xlarge:/root/# docker run -v /var/run:/var/run/ -ti --rm qnib/doxy:gpu doxy --proxy-socket=/var/run/hpc.sock \ 
   --pattern-key=hpc --gpu --cuda-lib-path=/usr/lib/nvidia-384

2018/02/17 10:47:27 [II] Start Version: 0.2.4
2018/02/17 10:47:27 [doxy] Listening on /var/run/hpc.sock
2018/02/17 10:47:27 Serving proxy on '/var/run/hpc.sock'
```

```bash
bob@p2.xlarge:$ docker -H unix:///var/run/hpc.sock create ubuntu /usr/local/nvidia/bin/nvidia-smi -L
   a3307eefa31233bdcb36e161ffe0ef433d93f80e43cc1dfe9ee32c45c10dc50d
```

Add GPU stuff
New device: /dev/nvidia0:/dev/nvidia0
New device: /dev/nvidiactl:/dev/nvidiactl
Mounts: [/usr/lib/nvidia-384:/usr/local/nvidia/]

```bash
bob@p2.xlarge:$ docker start -a a3307eefa31233bdcb36e161ffe0ef433d93f80e43cc1dfe9ee32c45c10dc50d
GPU 0: Tesla K80 (UUID: GPU-4095713a-1f9b-791d-841d-8b35143127d4)
```
Underpin Orchestration

Combine Data Lake and Distributed Compute
Engine vs. Workload Scheduler

- As of today, service orchestrators start containers unaware of shared systems.
- HPC workload schedulers start userland processes within shared environment.

Docker Engine

- SWARM
- Kubernetes

Shared System

process1

process2

HPC Workload Scheduler

- slurmd

Shared System

job-process1

job-process2

HPC-runtime workarounds
Engine serves Service and Workload Schedulers

By making the Docker-Engine aware of shared environments and kernel-bypassing hardware, it can serve all use-cases.
Outlook

2018 the year of High Performance Containers
Tentative Milestones

Critical MVP milestones

#1 Secure Operations and kernel-bypassing support

#1.1 Secure Operation in shared environment
In particular pinning a user to use his UID in shared file-systems, so that users are not able to access other userdata.

#1.2 Transparent Support of kernel-bypassing Devices
Host-specific GPU and Interconnects (e.g. InfiniBand)

#2 Integration of Workload Scheduler

#2.0 Integration of HPC workload schedulers
As a reference the open-source and widely adopted scheduler SLURM should implement container scheduling. Other can follow.
Student Cluster Competition

Docker is coaching the SCC-Team of the Student Cluster Competition at ISC 2018 in Frankfurt. Goals are:

1. Use of containerized auxiliary services (Monitoring, …)
2. Prototype workflow
Call For Action

How to get involved?

If you are already a customer of ours
➔ push for HPC/Big Data use-cases

If not
➔ become one and push for HPC/Big Data use-cases

To prepare for adoption:
➔ Enhance your operation / container knowledge by using DockerEE in non-HPC services
➔ Create Center-Of-Excellence for Containers
➔ Make yourself familiar with Supply Chain (a lot of gold here)
➔ Educating you Dev/Scientists/Engineers to use container (HPC) best-practices
➔ Ask for all of the above...
Mid-/Long-Term

Containers are a gold-mine

➔ Reproducibility
  ◆ signed images (/content-hashes) ensure integrity

➔ Bundle apps/libs using a Recipe (Dockerfile)
  ◆ Fosters collaboration
  ◆ optimize the user-land to best support the application, no need to consider impact to others.
  ◆ automate optimization with permuted pkg / config

➔ Fingerprint application (/workload)
  ◆ battle-tested workload OR needs to run through Q&A?

➔ diskless Storage Driver?
  ◆ treat shared FS as object store for all image blobs
  ◆ burst-buffer as to instantiate file-system of container

➔ read-only input data as volume pre-loaded on host, used by containers

➔ Using Containers fine-grained observibility comes for free
THANK YOU :)

[Image of Docker logo]