The State of Linux Containers

Christian Kniep, 2017-02-08
Containers in a Nutshell
Containers do not spin up a distinct kernel

- all containers & the host share the same user-lands
- user-lands are independent
- they are separated by Kernel Namespaces

**Traditional Virtualization**

**Containerization**

**Share nothing**

**Share everything**

**Share something**
Linux Containers

Containers do not spin up a distinct kernel

- all containers & the host share the same
- user-lands are independent

**NEWS ALERT:** That’s how public container clouds operate!
Interface View

Traditional Virtualization

- Application
- Userland
- Kernel
- Hardware
- Virtualization magic (HVM, PV, SR-IOV, PCI-passthrough, hyper-calls, ...)

Containerization

- Application
- Userland
- Kernel
- Hardware
- Lightweight abstraction with negligible overhead

Sys-calls

10^2
Containers are ‘grouped processes’

- isolated by Kernel Namespaces

Namespaces:

<table>
<thead>
<tr>
<th>Container</th>
<th>PID</th>
</tr>
</thead>
<tbody>
<tr>
<td>container1</td>
<td>bash</td>
</tr>
<tr>
<td></td>
<td>ls -l</td>
</tr>
<tr>
<td>container2</td>
<td>apache</td>
</tr>
<tr>
<td>container3</td>
<td>mysql</td>
</tr>
<tr>
<td>container4</td>
<td>slurmd</td>
</tr>
<tr>
<td></td>
<td>orted</td>
</tr>
</tbody>
</table>
1. Linux containers => forks in distinct Namespaces

```c
#define _GNU_SOURCE
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <unistd.h>

static char child_stack[1048576];

static int child_fn() {
    printf("PID: %ld\n", (long)getpid());
    return 0;
}

int main() {
    pid_t child_pid = clone(child_fn, child_stack+1048576, CLONE_NEWPID | SIGCHLD, NULL);
    printf("clone() = %ld\n", (long)child_pid);

    waitpid(child_pid, NULL, 0);
    return 0;
}
```
Containers are ‘grouped processes’

- isolated by Kernel Namespaces
- resource restrictions applicable through CGroups (disk/netI/O)

Namespaces: PID, Network, Mount, IPC, UTS, User, cgroup, RDMA
Looks familiar?

- First Namespace entered the 2.4.19 kernel (2002)
- Namespaces can be shared with host as well
$ docker version
Client:
  Version: 1.13.1-rc1
  API version: 1.25
  Go version: go1.7.4
  Git commit: 2527cfc
  Built: Sat Jan 28 00:43:00 2017
  OS/Arch: darwin/amd64

Server:
  Version: 1.13.1-rc1
  API version: 1.25 (minimum version 1.12)
  Go version: go1.7.4
  Git commit: 2527cfc
  Built: Sat Jan 28 00:43:00 2017
  OS/Arch: linux/amd64
  Experimental: true

macOS
$ docker
Alpine Linux
dockerd
alpine
$ docker run -ti alpine ash
/ #
BE ADVISED: If you are easily scared, distract yourself for 1 min!
1. Packaging
   - package once, run everywhere (due to sys-call abstraction)

2. Distribution
   - reproducible, easily shareable package, creates a DAG w/ sha256 (like git)

3. Simple Usage
   - $ docker inspect -f {{.RepoDigests}} alpine
   - [alpine@sha256:dfbd4a3a8ebca874ebd2474f044a0]

4. Monitoring and auditing for free
   - # Pin a specific parent image
   - FROM alpine@sha256:dfbd4a3a8ebca87

5. Hand-tailored user-lands
   - instead of comparing native and containerized user-land, compare generic and optimized user-land
MPI Benchmark

Open MPI comparison [2 task @2nodes, avg(1B->64B)]

QNIBTerminal Plus InfiniBand - Containerized MPI Workloads

RichReport

Published on Nov 25, 2014
8 nodes (CentOS 7, 2x 4core XEON, 32GB, Mellanox ConnectX-2)
Spinning up production-like environment is great

- MongoDB, PostreSQL, memcached as separate containers
- python2.7, python3.4

Like python’s virtualenv on steroids, iteration speedup through reproducibility
Runtime -> Cluster
1. runC (Docker Inc.)
   - low-level runtime, reference implementation of OCI

2. rkt (CoreOS)
   - Runtime to download, verify and start App Containers

3. Singularity
   - Runtime aimed to integrate well in existing, share-everything workflows

4. runV (hyperhq / runv)
   - Hypervisor-based Runtime for OCI
1. runC
   - containerd -> docker-engine

2. rkt (CoreOS)
   - rkt || containerd -> docker-engine

3. Singularity
   - leverage current workflows
1. **runC**
   - containerd -> docker-engine -> SwarmKit

2. **rkt (CoreOS)**
   - Kubernetes || Nomad || containerd-*->SwarmKit

3. **Singularity**
   - integrate in your current cluster management (Slurm)
Orchestration
1. Mesos (DC/OS)
   - “A distributed systems kernel”
   - Marathon to schedule containers
   - Running SWARM/Kubernetes on-top

2. Kubernetes
   - matured as (one of) the go-to orchestrators of services

3. SWARM
   - on the rise as orchestrator?
1. Shifter (NERSC)
   - Extracts Docker Image into squash-FS and runs chroot

2. Singularity
   - starting point is the user’s privilege level, no need to run as root
Cloud

1. aka. share-nothing cluster

2. AWS (re:invent 2016)
   - new instances c5: 72vCPU, 144GB RAM
     - Elastic Network Adapter (25Gb/s)
   - elasticGPU
   - snow(flake/mobile)
   - AWS batch

Reinvent 2016 Announcements Recap by Ian Massingham - YouTube
https://www.youtube.com/watch?v=ZAwk8pyvz5l
Jan 9, 2017 - Uploaded by AWS User Group UK
Reinvent 2016 Announcements Recap by Ian Massingham ... AWS re:Invent 2016: Analyzing Streaming ...
Separation of Concerns?

Split input iteration / development from operation

- non-distributed stays vanilla, to benefit from ecosystem innovation
- move to share-nothing/-everything cluster it becomes trickier
Spinning up production-like environment is…

- …not that easy
- focus more on engineer/scientist, not the software developer

1. For development it might work
   - close to non-HPC software dev

2. But is that the iteration-focus?
   - rather job settings / input data?

3. Operational environment
   - share everything/nothing
Container Focus

- Docker focused on Dev(Ops)
- Kubernetes on Ops(Dev)
- SWARM targets DevOps
- slurm scales Ops
- singularity makes DevOps status quo compatible

Disclaimer: subjective exaggeration
1. …becomes container ready!
   - fork MPI process in namespaces
   - even better: use container runtime to be compliant

2. Ideally the container engine understands MPI
   - AWS ECS + Snowflake + AWS Batch + AWS Lambda
   - Docker-Engine (and thus SWARM)

#getRichFast
#freeStartUpIdea
Q&A

ISC HIGH PERFORMANCE
SUNDAY, JUNE 18 - THURSDAY, JUNE 22, 2017
FRANKFURT, GERMANY

Container Workshop Proposed

http://qnib.org
Almost explored
Running OpenFOAM on small scale is cumbersome

- manually install OpenFOAM on a workstation
- pin the parent to it’s sha256

✔️ A containerised OpenFOAM installation tackles both

http://qnib.org/immutable-paper

http://qnib.org/immutable
1. Where to base images on?
   - Ubuntu/Fedora: ~200MB
   - Debian: ~100MB
   - Alpine Linux: 5MB (musl-libc)

2. Definitely small!
   - pid-sharing allows for monitoring hooks to reach in
   - docker-engine provides HEALTHCHECK
   - debug-container with different permissions
1. One Application
   - Ideally even only one process
   - At least only one application (nginx needs master/workers)

2. Benefits
   - Context of the processes within is limited, easier to understand
   - Clean metrics and logs provide easy to grasp information for Dev and Ops
   - Clean metrics and logs are easy for AI, auto-mediation
Allow Dev to run CI based

- around working on single App
- locally to test their work
  - spin up auxiliary services
- fast, rather simple check to speed up dev-cycle
If containers are immutable within pipeline

- continuous evaluation of stacks
- testing/deployment should be automated
- developers should have a production replica
- pipeline to enable fast release/deploy cycle
To Be Explored
1. Since the environments are rather dynamic…

- how does the containers discover services?
- external registry as part of the framework?
- discovery service as part of the container stacks?
1. Containers should be controlled via ENV or flags
   - External access/change of a running container is discouraged
   - docker secrets

2. Configuration management
   - Downgraded to bootstrap a host?
1. modularisation of basic components
   - push from outside to stabilise the core
   - make sure Docker Inc. has a foothold in the standardisation (?)
1.11
  - runC - Independent Runtime

1.12
  - containerd
1. consolidate the core
1.11
- runC - Independent Runtime

1.12
- containerd - runtime environment
- SwarmKit - orchestration framework
- ingress load-balancing
1. Manager (leader/follower) model
1. Manager (leader/follower) model
2. ingress load-balancer
1.11 runC - Independent Runtime

1.12 containerd - runtime environment
   - SwarmKit - orchestration framework
   - ingress load-balancing

1.13 host network-mode
   - secrets