# Container's Anatomy

## Namespaces, cgroups, and some filesystem magic

# Who am I?

- Jérôme Petazzoni (@jpetazzo)

- French software engineer living in California

- I have built and scaled the dotCloud PaaS
  (almost 5 years ago, time flies!)

- I talk (too much) about containers
  (first time was maybe in February 2013, at SCALE in L.A.)

- Proud member of the House of Bash
  (when annoyed, we replace things with tiny shell scripts)

# Outline

- What is a container?
  (high level and low level overviews)

- The building blocks
  (namespaces, cgroups, copy-on-write storage)

- Container runtimes
  (Docker, LXC, rkt, runC, systemd-nspawn...)

- Other containers
  (OpenVZ, Jails, Zones...)

- Hand-crafted, artisan-pick, house-blend containers
  (with demos!)

# What is a container?

# High level approach: it's a lightweight VM

- I can get a shell on it
  (through SSH or otherwise)

- It "feels" like a VM:

  - own process space
  - own network interface
  - can run stuff as root
  - can install packages
  - can run services
  - can mess up routing, `iptables` ...

# Low level approach:
# it's chroot on steroids

- It's not quite like a VM:

  - uses the host kernel
  - can't boot a different OS
  - can't have its own modules
  - doesn't need `init` as PID 1
  - doesn't need syslogd, cron...

- It's just a bunch of processes visible on the host machine
  (contrast with VMs which are opaque)

# 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,
  in documentation

# 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,
  in documentation

- ??? Are containers even in the kernel ???

# Ancient archeology

Five years ago...

- When using the LXC tools (`lxc-start`, `lxc-stop` ...)
  you would often get weird messages mentioning
  `/cgroup/container_name/...`

- The `cgroup` pseudo filesystem has counters for CPU, RAM,
  I/O, and device access (`/dev/*`) but nothing about network

- I didn't find about "namespaces" by myself
  (maybe noticing `/proc/$PID/ns` would have helped;
  more on that later)

# Modern archeology

Nowadays:

- Cgroups are often in `/sys/fs/cgroup`

- You quickly stumble on `nsenter`
  (which tips you off about namespaces)

- There is significantly more documentation everywhere

# The building blocks

# Control groups

# Control groups

- Resource *metering* and *limiting*

  - memory

  - CPU

  - block I/O

  - network*

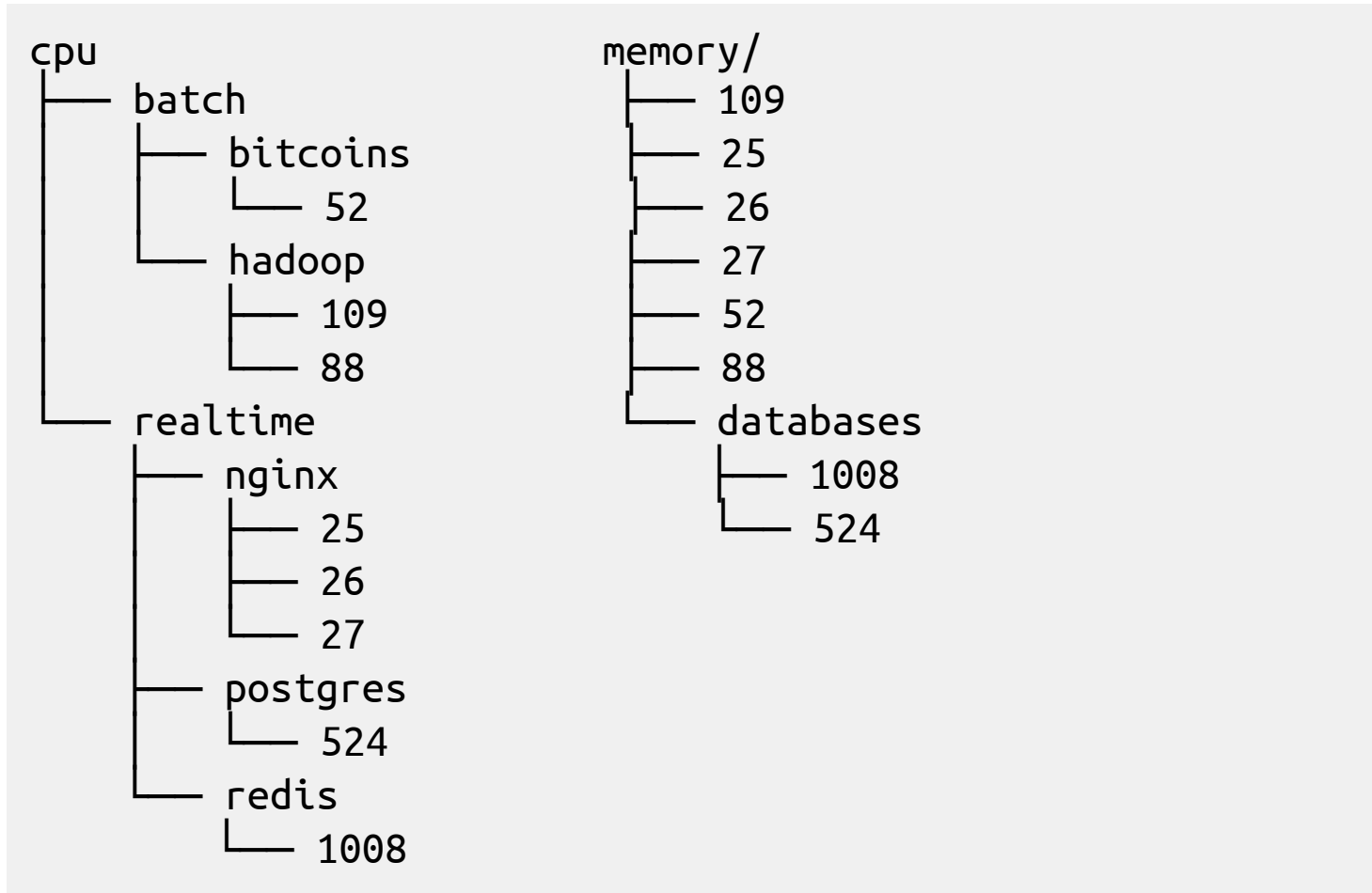- Device node (`/dev/*`) access control


*With cooperation from iptables/tc; more on that later

# Generalities

- Each subsystem (memory, CPU...) has a *hierarchy* (tree)

- Hierarchies are independent
  (the trees for e.g. memory and CPU can be different)

- Each process belongs to exactly 1 node in each hierarchy
  (think of each hierarchy as a different dimension or axis)

- Each hierarchy starts with 1 node (the root)

- All processes initially belong to the root of each hierarchy*

- Each node = group of processes
  (sharing the same resources)

*It's a bit more subtle; more on that later

# Example

```
cpu
├── batch
│   ├── bitcoins
│   │   └── 52
│   └── hadoop
│       ├── 109
│       └── 88
└── realtime
    ├── nginx
    │   ├── 25
    │   ├── 26
    │   └── 27
    ├── postgres
    │   └── 524
    └── redis
        └── 1008
```

```
memory/
├── 109
├── 25
├── 26
├── 27
├── 52
├── 88
└── databases
    ├── 1008
    └── 524
```

# Memory cgroup: accounting

- Keeps track of pages used by each group:

  - file (read/write/mmap from block devices)
  - anonymous (stack, heap, anonymous mmap)
  - active (recently accessed)
  - inactive (candidate for eviction)

- Each page is "charged" to a group

- Pages can be shared across multiple groups
  (e.g. multiple processes reading from the same files)

- When pages are shared, the groups "split the bill"

# Memory cgroup: limits

- Each group can have (optional) hard and soft limits

- Soft limits are not enforced
  (they influence reclaim under memory pressure)

- Hard limits will trigger a per-group OOM killer

- The OOM killer can be customized (oom-notifier);
  when the hard limit is exceeded:

  - freeze all processes in the group
  - notify user space (instead of going rampage)
  - we can kill processes, raise limits, migrate containers ...
  - when we're in the clear again, unfreeze the group

- Limits can be set for physical, kernel, total memory

# Memory cgroup: tricky details

- Each time the kernel gives a page to a process,
  or takes it away, it updates the counters

- This adds some overhead

- Unfortunately, this cannot be enabled/disabled per process
  (it has to be done at boot time)

- Cost sharing means thata process leaving a group
  (e.g. because it terminates) can theoretically cause
  an out of memory condition

# Cpu cgroup

- Keeps track of user/system CPU time

- Keeps track of usage per CPU

- Allows to set weights

- Can't set CPU limits

    - OK, let's say you give N%
    - then the CPU throttles to a lower clock speed
    - now what?
    - same if you give a time slot
    - instructions? their exec speed varies wildly
    - ¯\_ツ_/¯

# Cpuset cgroup

- Pin groups to specific CPU(s)

- Reserve CPUs for specific apps

- Avoid processes bouncing between CPUs

- Also relevant for NUMA systems

- Provides extra dials and knobs
  (per zone memory pressure, process migration costs...)

# Blkio cgroup

- Keeps track of I/Os for each group

  - per block device
  - read vs write
  - sync vs async

- Set throttle (limits) for each group

  - per block device
  - read vs write
  - ops vs bytes

- Set relative weights for each group

Full disclosure: this used to be clunky for async operations, and I haven't tested it in ages. ☹

# Net_cls and net_prio cgroup

- Automatically set traffic class or priority,
  for traffic generated by processes in the group

- Only works for egress traffic

- Net_cls will assign traffic to a class
  (that has to be matched with tc/iptables,
  otherwise traffic just flows normally)

- Net_prio will assign traffic to a priority
  (priorities are used by queuing disciplines)

# Devices cgroup

- Controls what the group can do on device nodes

- Permissions include read/write/mknod

- Typical use:

  - allow /dev/{tty,zero,random,null} ...
  - deny everything else

- A few interesting nodes:

  - /dev/net/tun (network interface manipulation)
  - /dev/fuse (filesystems in user space)
  - /dev/kvm (VMs in containers, yay inception!)
  - /dev/dri (GPU)

# Subtleties

- PID 1 is placed at the root of each hierarchy

- When a process is created, it is placed in the same groups as its parent

- Groups are materialized by one (or multiple) pseudo-fs (typically mounted in `/sys/fs/cgroup`)

- Groups are created by `mkdir` in the pseudo-fs

- To move a process:

```
echo $PID > /sys/fs/cgroup/.../tasks
```

- The cgroup wars: systemd vs cgmanager vs ...

# Namespaces

# Namespaces

- Provide processes with their own view of the system

- Cgroups = limits how much you can use;
  namespaces = limits what you can see (and therefore use)

- Multiple namespaces:

  - pid
  - net
  - mnt
  - uts
  - ipc
  - user

- Each process is in one namespace of each type

# Pid namespace

- Processes within a PID namespace only see processes in the same PID namespace

- Each PID namespace has its own numbering (starting at 1)

- When PID 1 goes away, the whole namespace is killed

- Those namespaces can be nested

- A process ends up having multiple PIDs (one per namespace in which its nested)

# Net namespace: in theory

- Processes within a given network namespace get their own private network stack, including:

    - network interfaces (including `lo`)

    - routing table**s**

    - iptables rules

    - sockets (`ss`, `netstat`)

- You can move a network interface from a netns to another

```
ip link set dev eth0 netns PID
```

# Net namespace: in practice

- Typical use-case:

  - use `veth` pairs
    (two virtual interfaces acting as a cross-over cable)

  - `eth0` in container network namespace

  - paired with `vethXXX` in host network namespace

  - all the `vethXXX` are bridged together
    (Docker calls the bridge `docker0`)

- But also: the magic of `--net container`

  - shared `localhost` (and more!)

# Mnt namespace

- Processes can have their own root fs (à la chroot)

- Processes can also have "private" mounts

    - `/tmp` (scoped per user, per service...)

    - Masking of `/proc`, `/sys`

    - NFS auto-mounts (why not?)

- Mounts can be totally private, or shared

- No easy way to pass along a mount
  from a namespace to another ☹

# Uts namespace

- gethostname / sethostname

- 'nuff said!

# Ipc namespace

# Ipc namespace

* Does anybody knows about IPC?

# Ipc namespace

- Does anybody knows about IPC?

- Does anybody *cares* about IPC?

# Ipc namespace

- Does anybody knows about IPC?

- Does anybody *cares* about IPC?

- Allows a process (or group of processes) to have own:

    - IPC semaphores
    - IPC message queues
    - IPC shared memory

    ... without risk of conflict with other instances.

# User namespace

- Allows to map UID/GID; e.g.:

  - UID 0→1999 in container C1 is mapped to
    UID 10000→11999 on host
  - UID 0→1999 in container C2 is mapped to
    UID 12000→13999 on host
  - etc.

- Avoids extra configuration in containers

- UID 0 (`root`) can be squashed to a non-privileged user

- Security improvement

But: devil is in the details

# Namespace manipulation

- Namespaces are created with the `clone()` system call (i.e. with extra flags when creating a new process)

- Namespaces are materialized by pseudo-files in `/proc/<pid>/ns`

- When the last process of a namespace exits, it is destroyed (but can be preserved by bind-mounting the pseudo-file)

- It is possible to "enter" a namespace with `setns()` (exposed by the `nsenter` wrapper in `util-linux`)

# Copy-on-write

# Copy-on-write storage

- Create a new container instantly
  (instead of copying its whole filesystem)

- Storage keeps track of what has changed

- Many options available

  - AUFS, overlay (file level)
  - device mapper thinp (block level)
  - BTRFS, ZFS (FS level)

- Considerably reduces footprint and "boot" times

See also: Deep dive into Docker storage drivers

# Other details

# Orthogonality

- All those things can be used independently

- Use a few cgroups if you just need resource isolation

- Simulate a network of routers with network namespaces

- Put the debugger in a container's namespaces,
  but not its cgroups (to not use its resource quotas)

- Setup a network interface in an isolated environment,
  then move it to another

- etc.

# One word about overhead

- Even when you don't run containers ...
  ... you are in a container

- Your host processes still execute in the root
  namespaces and cgroups

- Remember: there are three kind of lies

# One word about overhead

- Even when you don't run containers ...
  ... you are in a container

- Your host processes still execute in the root namespaces and cgroups

- Remember: there are three kind of lies

- Lies, damn lies, and benchmarks

# Some missing bits

- Capabilities

  - break down "root / non-root" into fine-grained rights

  - allow to keep root, but without the dangerous bits

  - however: `CAP_SYS_ADMIN` remains a big catchall

- SELinux / AppArmor ...

  - containers that actually contain

  - deserve a whole talk on their own

# Container runtimes

# LXC

- Set of userland tools

- A container is a directory in `/var/lib/lxc`

- Small config file + root filesystem

- Early versions had no support for CoW

- Early versions had no support to move images around

- Requires significant amount of elbow grease
  (easy for sysadmins/ops, hard for devs)

# systemd-nspawn

*From its manpage:*

- "For debugging, testing and building"

- "Similar to chroot, but more powerful"

- "Implements the Container Interface"

- Seems to position itself as plumbing

# systemd-nspawn

*From its manpage:*

- "For debugging, testing and building"

- "Similar to chroot, but more powerful"

- "Implements the Container Interface"

- Seems to position itself as plumbing

- Recently added support for ɹǝʞɔop images

# systemd-nspawn

*From its manpage:*

- "For debugging, testing and building"

- "Similar to chroot, but more powerful"

- "Implements the Container Interface"

- Seems to position itself as plumbing

- Recently added support for ɹǝʞɔop images

```
#define INDEX_HOST "index.do" /* the URL we get the data from */ "cker.io"
#define HEADER_TOKEN "X-Do" /* the HTTP header for the auth token */ "cker-Token:"
#define HEADER_REGISTRY "X-Do" /*the HTTP header for the registry */ "cker-Endpoints:"
```

# Docker Engine

- Daemon controlled by REST(ish) API

- First versions shelled out to LXC,
  now uses its own `libcontainer` runtime

- Manages containers, images, builds, and more

- Some people think it does too many things

# rkt, runC

- Back to the basics!

- Focus on container execution
  (no API, no image management, no build, etc.)

- They implement different specifications:

  - rkt implements appc (App Container)

  - runC implements OCP (Open Container Project),
    leverages Docker's libcontainer

# Which one is best?

- They *all* use the same kernel features

- Performance will be exactly the same

- Look at:

  - features

  - design

  - ecosystem

# Other containers

# OpenVZ

- Also Linux

- Older, but battle-tested
  (e.g. Travis CI gives you root in OpenVZ)

- Tons of neat features too

  - ploop (efficient block device for containers)

  - checkpoint/restore, live migration

  - venet (~more efficient veth)

- Still developed

# Jails / Zones

- FreeBSD / Solaris

- Coarser granularity than namespaces and cgroups

- Strong emphasis on security

- Great for hosting providers

- Not so much for developers
  (where's the equivalent of `docker run -ti ubuntu`?)

Note: Solaris branded zones can run Linux binaries

# Build your own

# FOR EDUCATIONAL PURPOSES ONLY

```
root@dockerhost:~# ▯
```

# Thanks!
# Questions?

@jpetazzo
@docker