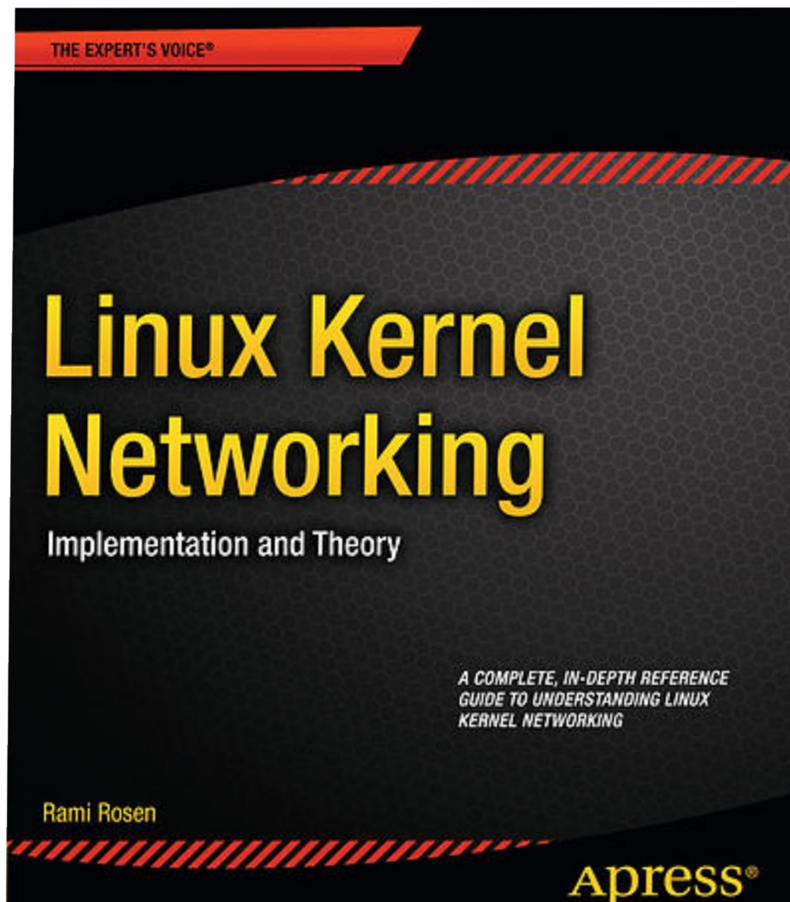
Namespaces and Cgroups – the basis of Linux Containers

Rami Rosen

http://ramirose.wix.com/ramirosen

- About me: kernel developer, mostly around networking and device drivers, author of “Linux Kernel Networking”, Apress, 648 pages, 2014.



Namespaces and **cgroups** are the basis of lightweight process virtualization.

As such, they form the basis of Linux containers.

They can also be used for setting easily a testing/debugging environment or a resource separation environment and for resource accounting/logging.

Namespaces and cgroups are orthogonal.

We will talk mainly about the kernel implementation with some userspace usage examples.

What is lightweight process virtualization ?

A process that gives the user an illusion that he runs a linux operating system. You can run many such processes on a machine, and all such processes in fact share a **single Linux kernel** which runs on the machine.

This is opposed to hypervisor-based solutions, like Xen or KVM, where you run **another instance of the kernel.**

The idea is not revolutionary - we have Solaris Zones and BSD jails already several years ago.

A Linux container is in fact a process.

Containers versus Hypervisor-based VMs

It seems that Hypervisor-based VMs like KVM are here to stay (at least for the next several years). There is an ecosystem of cloud infrastructure around solutions like KVMs.

Advantages of Hypervisor-based VMs (like KVM) :

You can create VMs of other operating systems (windows, BSDs).

Security (Though there were cases of security vulnerabilities which were found and required patches to handle them, like VENOM).

Containers – advantages:

Lightweight: occupies less resources (like memory) significantly then hypervisor.

Density – you can install many more containers on a given host than KVM-based VMs.

elasticity - start time and shutdown time is much shorter, almost instantaneous. Creation of a container has the overhead of creating a Linux process, which can be of the order of milliseconds, while creating a vm based on XEN/KVM can take seconds.

The lightness of the containers in fact provides their density and their elasticity.

There is a single Linux kernel infrastructure for containers (namespaces and cgroups) while for Xen and KVM we have two different implementations without any common code.

Namespaces

Development took over a decade: Namespaces implementation started in about **2002**; the last one, true for today, (user namespaces) was completed in February **2013,** in kernel 3.18.

There are currently 6 namespaces in Linux:

- **mnt** (mount points, filesystems)
- **pid** (processes)
- **net** (network stack)
- **ipc** (System V IPC)
- **uts** (hostname)
- **user** (UIDs)

In the past there were talks on adding more namespaces – device namespaces (LPC 2013), and other (OLS 2006, Eric W. Biederman).

Namespaces - contd

A process can be created in Linux by the ***fork()***, ***clone()*** **or** ***vclone()*** system calls.

In order to support namespaces, 6 flags (CLONE_NEW*) were added:

(include/linux/sched.h)

These flags (or a combination of them) can be used in ***clone()*** or ***unshare()*** system calls to create a namespace.

Namespaces clone flags

Clone flag	Kernel Version	Required capability
CLONE_NEWNS	2.4.19	CAP_SYS_ADMIN
CLONE_NEWUTS	2.6.19	CAP_SYS_ADMIN
CLONE_NEWIPC	2.6.19	CAP_SYS_ADMIN
CLONE_NEWPID	2.6.24	CAP_SYS_ADMIN
CLONE_NEWNET	2.6.29	CAP_SYS_ADMIN
CLONE_NEWUSER	3.8	No capability is required

Namespaces system calls

Namespaces API consists of these 3 system calls:

● ***clone()*** **-** creates a **new process** and a **new namespace**; the newly created process is attached to the new namespace.

– The process creation and process termination methods, ***fork()*** and ***exit()***, were patched to handle the new namespace CLONE_NEW* flags.

● ***unshare()*** **– gets only a single parameter, flags.** Does **not** create a new process; creates **a new namespace** and **attaches** the **calling process** to it.

– ***unshare()*** was added in 2005.

see “new system call, unshare” : http://lwn.net/Articles/135266

● ***setns()*** **-** a new system call, for joining the calling process to an existing namespace; prototype: *int setns(int fd, int nstype);*

Each namespace is assigned a unique ***inode*** number when it is created.

- **ls -al /proc/<pid>/ns**

lrwxrwxrwx 1 root root 0 Apr 24 17:29 ipc -> ipc:[4026531839]

lrwxrwxrwx 1 root root 0 Apr 24 17:29 mnt -> mnt:[4026531840]

lrwxrwxrwx 1 root root 0 Apr 24 17:29 net -> net:[4026531956]

lrwxrwxrwx 1 root root 0 Apr 24 17:29 pid -> pid:[4026531836]

lrwxrwxrwx 1 root root 0 Apr 24 17:29 user -> user:[4026531837]

lrwxrwxrwx 1 root root 0 Apr 24 17:29 uts -> uts:[4026531838]

A namespace is terminated when all its processes are terminated and when its inode is not held (the inode can be held, for example, by bind mount).

Userspace support for namesapces

Apart from kernel, there were also some user space additions:

- **IPROUTE** package:
  - Some additions like *ip netns add/ip netns del* and more commands (starting with *ip netns* ...)
  - We will see some examples later.
- **util-linux** package:
  - *unshare* util with support for all the 6 namespaces.
  - *nsenter* – a wrapper around *setns()*.
- See: *man 1 unshare* and *man 1 nsenter*.

UTS namespace

UTS namespace provides a way to get information about the system with commands like ***uname*** or ***hostname***.

UTS namespace was the most simple one to implement.

There is a member in the process descriptor called nsproxy.

A member named **uts_ns** (uts_namespace object) was added to it.

The uts_ns object includes an object (new_utsname struct) with 6 members:

sysname

nodename

release

version

machine

domainname

Former implementation of *gethostname()*:

The **former** implementation of ***gethostname()***:

```
asmlinkage long sys_gethostname(char __user *name, int len)

{

..

if (copy_to_user(name, system_utsname.nodename, i))

... errno = -EFAULT;

}
```

(system_utsname is a global)

kernel/sys.c, Kernel v2.6.11.5

New implementation of *gethostname()*:

A Method called ***utsname()*** was added:

```
static inline struct new_utsname *utsname(void)
{
return &current->nsproxy->uts_ns->name;
}
```

The **new** implementation of ***gethostname()***:

```
SYSCALL_DEFINE2(gethostname, char __user *, name, int, len)
{
struct new_utsname *u;
...
u = utsname();
if (copy_to_user(name, u->nodename, i))
errno = -EFAULT;
.
}
```

Similar approach was taken in ***uname()*** and ***sethostname()*** syscalls.

For **IPC namespaces**, the same principle as in UTS namespace was held, nothing special, just more code.

Added a member named **ipc_ns** (ipc_namespace object) to the nsproxy object.

Network Namespaces

A network namespace is logically another copy of the network stack,

with its own routing tables, firewall rules, and network devices.

● The network namespace is represented by a huge **struct net.** (defined in *include/net/net_namespace.h*)

struct net includes all network stack ingredients, like:

– Loopback device.

– SNMP stats. (netns_mib)

– All network tables:routing, neighboring, etc.

– All sockets

– */procfs* and */sysfs* entries.

At a given moment -

• **A network device belongs to exactly one network namespace.**

• **A socket belongs to exactly one network namespace.**

The default initial network namespace, **init_net (instance of struct net)**, includes the loopback device and all physical devices, the networking tables, etc.

● Each **newly** created network namespace includes only the loopback device.

Example

Create two namespaces, called "myns1" and "myns2":

● ***ip netns add myns1***

● ***ip netns add myns2***

This triggers:

● Creation of */var/run/netns/myns1,/var/run/netns/myns2* empty **folders.**

● Invoking the ***unshare()*** system call with *CLONE_NEWNET.*

– ***unshare()*** *does not trigger cloning of a process; it does create*

a new namespace (a network namespace, because of the

CLONE_NEWNET flag).

you delete a namespace by:

- ***ip netns del myns1***

– This unmounts and removes /var/run/netns/myns1

You can list the network namespaces (which were added via "*ip netns add*") by:

- ***ip netns list***

You can monitor addition/removal of network namespaces by:

- ***ip netns monitor***

This prints one line for each addition/removal event it sees.

You can **move** a network interface (eth0) to myns1 network namespace by:

- ***ip link set eth0 netns myns1***

You can start a bash shell in a new namespace by:

- ***ip netns exec myns1 bash***

Recent additions – add “all” parameter to exec to allow exec on each netns; for example:

ip -all netns exec ip link

Show link info on all net namespaces.

A nice feature:

Applications which usually look for configuration files under /etc (like /etc/hosts or /etc/resolv.conf), will first look under /etc/netns/NAME/, and only if nothing is available there, will look under /etc.

PID namespaces

Added a member named **pid_ns** (pid_namespace object) to the *nsproxy*.

- Processes in different PID namespaces can have the same process ID.
- When creating the first process in a new namespace, its PID is 1.
- Behavior like the “init” process:

– When a process dies, all its orphaned children will now have the process with PID 1 as their parent (**child reaping**).

– Sending **SIGKILL** signal does not kill process 1, regardless of in which namespace the command was issued (initial namespace or other pid namespace).

- pid namespaces can be nested, up to 32 nesting levels. (MAX_PID_NS_LEVEL).

See: multi_pidns.c, Michael Kerrisk, from http://lwn.net/Articles/532745/.

The CRIU project

PID use case

The CRIU project - Checkpoint-Restore In Userspace

The Checkpoint-Restore feature is stopping a process and saving its state to the filesystem and later on starting it on the same machine or on a different machine. This feature is required in HPC mostly for load balancing and maintenance.

Previous attempts from OpenVZ folks to implement the same in the kernel in 2005 were rejected by the community as they were too intrusive. (A patch series of 17,000 lines, touching the most sensitive linux kernel subsystems).

When restarting a process in a different machine, you can have a collision in PID numbers of that process and the threads within it with other processes in the new machine.

Creating the process which has its own PID namespace avoids this collision.

Mount namespaces

Added a member named **mnt_ns**

(mnt_namespace object) to the *nsproxy*.

- In the new mount namespace, all previous mounts will be visible; and from now on, mounts/unmounts in that mount namespace are invisible to the rest of the system.

- mounts/unmounts in the global namespace are visible in that namespace.

More info about the low level details can be found in “Shared subtrees”by Jonathan Corbet, http://lwn.net/Articles/159077

User Namespaces

Added a member named *user_ns* (user_namespace object) to the credentials object (struct cred). Notice that this is different than the other 5 namespace pointers, which were added to the nsproxy object.

Each process will have a distinct set of UIDs, GIDs and capabilities.

User namespace enables a non root user to create a process in which it will be root (this is the basis for *unprivileged containers*)

Soon after this feature was mainlined, a security vulnerability was found in it and fixed; see:

"Anatomy of a user namespaces vulnerability"

By Michael Kerrisk, March 2013; an article about CVE 2013-1858, exploitable security Vulnerability:

http://lwn.net/Articles/543273/

cgroups

The cgroups (control groups) subsystem is a **Resource Management and Resource Accounting/Tracking** solution, providing a generic process-grouping framework.

- It handles resources such as memory, cpu, network, and more.
- This work was started by engineers at Google (primarily Paul Menage and Rohit Seth) in **2006** under the name "process containers”; shortly after, in **2007**, it was renamed to “Control Groups”.
- Merged into kernel 2.6.24 (2008).
- Based on an OpenVZ solution called “bean counters”.
- Maintainers: **Li Zefan** (Huawei) and **Tejun Heo** (Red Hat).
- The memory controller (memcg) is maintained separately (4 maintainers)
- The memory controller is the most complex.

cgroups implementation

No new system call was needed in order to support cgroups.

– **A new file system (VFS), "cgroup“** (also referred sometimes as cgroupfs).

The implementation of the cgroups subsystem required a few, simple hooks into the rest of the kernel, **none in performance-critical paths:**

– In boot phase (***init/main.c*)** to perform various initializations.

– In process creation and destruction methods, ***fork()*** and ***exit().***

– Process descriptor additions (*struct task_struct*)

– Add *procfs* entries:

● For each process: */proc/pid/cgroup.*

● System-wide: */proc/cgroups*

cgroups VFS

Cgroups uses a Virtual File System (VFS)

– All entries created in it are **not persistent** and are deleted after

reboot.

- All cgroups actions are performed via filesystem actions (create/remove/rename directories, reading/writing to files in it, mounting/mount options/unmounting).

- For example:

– cgroup *inode_operations* for cgroup mkdir/rmdir.

– cgroup *file_system_type* for cgroup mount/unmount.

– cgroup *file_operations* for reading/writing to control files.

Mounting cgroups

In order to use the cgroups filesystem (browse it/attach tasks to cgroups, etc) it must be mounted, as any other filesystem. The cgroup filesystem can be mounted on any path on the filesystem. **Systemd** uses */sys/fs/cgroup*.

When mounting, we can specify with mount options (-o) which cgroup controllers we want to use.

There are 11 cgroup subsystems (controllers); **two** can be built as modules.

Example: mounting net_prio

mount -t cgroup -onet_prio none /sys/fs/cgroup/net_prio

Fedora 23 cgroup controllers

list of cgroup controllers - obtained by ***ls -a /sys/fs/cgroups,***

dr-xr-xr-x 2 root root 0 Feb 6 14:40 blkio

lrwxrwxrwx 1 root root 11 Feb 6 14:40 cpu -> cpu,cpuacct

lrwxrwxrwx 1 root root 11 Feb 6 14:40 cpuacct -> cpu,cpuacct

dr-xr-xr-x 2 root root 0 Feb 6 14:40 cpu,cpuacct

dr-xr-xr-x 2 root root 0 Feb 6 14:40 cpuset

dr-xr-xr-x 4 root root 0 Feb 6 14:40 devices

dr-xr-xr-x 2 root root 0 Feb 6 14:40 freezer

dr-xr-xr-x 2 root root 0 Feb 6 14:40 hugetlb

dr-xr-xr-x 2 root root 0 Feb 6 14:40 memory

lrwxrwxrwx 1 root root 16 Feb 6 14:40 net_cls -> net_cls,net_prio

dr-xr-xr-x 2 root root 0 Feb 6 14:40 net_cls,net_prio

lrwxrwxrwx 1 root root 16 Feb 6 14:40 net_prio -> net_cls,net_prio

dr-xr-xr-x 2 root root 0 Feb 6 14:40 perf_event

dr-xr-xr-x 4 root root 0 Feb 6 14:40 systemd

Memory controller control files

cgroup.clone_children
cgroup.event_control
cgroup.procs
cgroup.sane_behavior
memory.failcnt
memory.force_empty
memory.kmem.failcnt
memory.kmem.limit_in_bytes
memory.kmem.max_usage_in_bytes
memory.kmem.slabinfo
memory.kmem.tcp.failcnt
memory.kmem.tcp.limit_in_bytes
memory.kmem.tcp.max_usage_in_bytes
memory.kmem.tcp.usage_in_bytes
memory.kmem.usage_in_bytes
memory.limit_in_bytes
memory.memsw.failcnt
memory.memsw.limit_in_bytes
memory.memsw.max_usage_in_bytes
memory.memsw.usage_in_bytes
memory.move_charge_at_immigrate
memory.numa_stat
memory.oom_control
memory.pressure_level
memory.soft_limit_in_bytes
memory.stat
memory.swappiness
memory.usage_in_bytes
memory.use_hierarchy
notify_on_release
release_agent
tasks

Example 1: memcg (memory control groups)

mkdir /sys/fs/cgroup/memory/group0

*The tasks entry that is created under **group0** is empty (*processes are called **tasks** in cgroup terminology).

echo $$ > /sys/fs/cgroup/memory/group0/tasks

*The $$ pid (current bash shell process) is moved from the memory controller in which it resides into **group0** memory controller.*

memcg (memory control groups) - contd

echo 10M > /sys/fs/cgroup/memory/group0/memory.limit_in_bytes

The implementation (and usage) of memory baloonning in Xen/KVM is much more complex, not to mention KSM (Kernel Same Page Merging).

You can disable the out of memory killer with memcg:

echo 1 > /sys/fs/cgroup/memory/*group0*/memory.oom_control

This disables the oom killer.

cat /sys/fs/cgroup/memory/group0/memory.oom_control

oom_kill_disable 1

under_oom 0

Now run some memory hogging process in this cgroup, which is known to be killed with oom killer in the default namespace.

- This process will **not** be killed.
- After some time, the value of ***under_oom*** will change to 1
- After enabling the OOM killer again:

echo 0 > /sys/fs/cgroup/memory/group0/memory.oom_control

You will get soon the OOM “Killed” message.

Use case: keep critical processes from being destroyed by OOM. For example, disable ***sshd*** from being killed by OOM – this will allow you to be sure to be able to ssh into a machine which runs low on memory.

Example 2: release_agent in memcg

The release agent mechanism is invoked when the last process of a cgroup terminates.

● The cgroup sysfs ***notify_on_release*** entry should be set so that

release_agent will be invoked.

● Prepare a short script, for example, */work/dev/t/date.sh*:

#!/bin/sh

date >> /root/log.txt

Assign the ***release_agent*** of the memory controller to be *date.sh*:

echo /work/dev/t/date.sh > /sys/fs/cgroup/memory/release_agent

Run a simple process, which simply sleeps forever; let's say it's PID is *pidSleepingProcess*.

echo 1 > /sys/fs/cgroup/memory/notify_on_release

release_agent example (contd)

mkdir /sys/fs/cgroup/memory/0/

echo pidSleepingProcess > /sys/fs/cgroup/memory/0/tasks

kill -9 pidSleepingProcess

This activates the release_agent; so we will see that the current time and date was written to /root/log.txt.

The release_agent can be set also via a mount option; systemd, for example, use this mechanism. For example in Fedora 23, mount shows:

*cgroup on /sys/fs/cgroup/systemd type cgroup (rw,nosuid,nodev,noexec,relatime,xattr,**release_agent=/usr/lib/systemd/systemd-cgroups-agent**,name=systemd)*

The release_agnet mechanism is quite heavy; see: "The past, present, and future of control groups", https://lwn.net/Articles/574317/

Example 3: devices control group

Also referred to as : *devcg* (devices control group)

● devices cgroup provides enforcing restrictions on reading, writing and creating (mknod) operations on device files.

● 3 control files: **devices.allow**, **devices.deny**, **devices.list**.

– **devices.allow** can be considered as devices whitelist

– **devices.deny** can be considered as devices blacklist.

– **devices.list** available devices.

● Each entry in these files consist of 4 fields:

– **type**: can be a (all), c (char device), or b (block device).

● All means all types of devices, and all major and minor numbers.

– **Major number.**

– **Minor number.**

– **Access:** composition of 'r' (read), 'w' (write) and 'm' (mknod).

devices control group – example (contd)

/dev/null major number is 1 and minor number is 3 (see *Documentation/devices.txt*)

mkdir /sys/fs/cgroup/devices/group0

By default, for a new group, you have full permissions:

cat /sys/fs/cgroup/devices/group0/devices.list

*a *:* rwm*

echo 'c 1:3 rmw' > /sys/fs/cgroup/devices/group0/devices.deny

This denies rmw access from ***/dev/null*** device.

echo $$ > /sys/fs/cgroup/devices/group0/tasks #Runs the current shell in group0

echo "test" > /dev/null

bash: /dev/null: Operation not permitted

devices control group – example (contd)

echo a > /sys/fs/cgroup/devices/group0/devices.allow

This adds the 'a *:* rwm' entry to the whitelist.

echo "test" > /dev/null

Now there is no error.

Cgroups Userspace tools

There is a cgroups management package called *libcgroup-tools*.

Running the cgconfig (control group config) service (a systemd service) :

systemctl start cgconfig.service / systemctl stop cgconfig.service

Create a group:

cgcreate -g memory:group1

Creates: /sys/fs/cgroup/memory/group1/

Delete a group:

cgdelete -g memory:group1

Adds a process to a group:

cgclassify -g memory:group1 <pidNum>

Adds a process whose pid is *pidNum* to **group1** of the memory controller.

Userspace tools - contd

cgexec -g memory:group0 sleepingProcess

Runs sleepingProcess in group0 (the same as if you wrote the pid of that process into group/tasks of the memory controller).

lssubsys – shows the list of mounted cgroup controllers.

There is a configuration file, **/etc/cgconfig.conf**. You can define groups which will be created when the service is started; thus, you can provide persistent configuration across reboots.

```
group group1 {
   memory {
     memory.limit_in_bytes = 3.5G;
   }
}
```

cgmanager

Problem: there can be many userspace daemons which set cgroups sysfs entries (like *systemd*, *libvirt*, *lxc*, *docker*, and others)

How can we guarantee that one will not override entries written by the other?

Solution – **cgmanager**: A **cgroup manager** daemon

- Currently under development (no rpm for Fedora/RHEL, for example).
- A userspace daemon based on DBUS messages.
- Developer: Serge Hallyn (Canonical)

- One of the LXC maintainers.

https://linuxcontainers.org/cgmanager/introduction/

CVE found in cgmanager on 6 of January, 2015:

http://www.securityfocus.com/bid/71896

groups and docker

cpuset – example:

Configuring docker containers is in fact mostly setting cgroups and namespaces entries; for example, limiting cores in a docker container to be 0, 2 can be done by:

docker run -i --cpuset=0,2 -t fedora /bin/bash

This is in fact writing into the corresponding cgroups entry, so after running this command, we will have:

cat /sys/fs/cgroup/cpuset/system.slice/docker-64bit_ID.scope/cpuset.cpus

0,2

In Docker and in LXC, you can configure cgroup/namespaces via config files.

Background - Linux Containers projects

LXC and Docker are based on cgroups and namespaces.

LXC originated in a French company which was bought by IBM in 2005; the code was rewritten from scratch and released as an opensource project. The two maintainers are from Canonical.

Systemd has ***systemd-nspawnd*** containers for testing and debugging.

Inspired by RedHat, there was some work done by Dan Welsh for SELinux enhancements for systemd for containers.

OpenVZ was released as an open source in 2005.

It’s advantage over LXC/Docker is that it is in production for many years.

Containers work started on 1999 with Kernel 2.2 (for the Virtuozzo project of SWSoft).

Later Virtuozzo was renamed to Parallels Cloud Server, or PCS for short.

Announced that they will open the git repository of RHEL7-based Virtuozzo kernel early this year (2015).

The name of the new project: Virtuozzo Core

lmctfy of google – only based on cgroups, does not use namespaces.

lmctfy stands for: *let me contain that for you.*

Docker

Docker is a popular open source project, available on github, written mostly in Go. **Docker** is developed by a company called dotCloud; in its early days it was based on using LXC, but later developed it own lib instead.

Docker has a good delivery mechanism.

Over 850 contributors.

https://github.com/docker/docker

Based on **cgroups** for resource allocation and on **namespaces** for resource isolation.

Docker is a popular project, with a large ecosystem around it.

There is a registry of containers on the public Docker website.

RedHat released an announcement in September 2013 about technical collaboration with dotCloud.

Docker - contd

Creating and managing docker containers is a simple task. So, for example, in Fedora 23, all you need to do is to run "dnf install docker".

(In Feodra 21 and lower Fedora releases, you should run "yum install docker-io")

To start the service you should run:

systemctl start docker

And then, to create and run a **fedora** container:

sudo docker run -i -t fedora /bin/bash

Or to create and run an **ubuntu** container:

sudo docker run -i -t ubuntu /bin/bash

In order to exit from a container, simply run exit from the container prompt:

docker ps - shows the running containers.

Dockerfiles

Using **Dockerfiles**:

Create the following Dockerfile:

FROM fedora

MAINTAINER JohnDoe

RUN yum install http

Now run the following from the folder where this Dockerfile resides:

docker build .

docker diff

Another nice feature of Docker is a **git diff functionality** of images, by ***docker diff;*** A denotes “added”, “C” denotes change; for example:

docker diff docakerContainerID

docker diff 7bb0e258aefe

...

C /dev

A /dev/kmsg

C /etc

A /etc/mtab

A /go

...

Why do we need yet another containers project like Kubernetes?

Docker Advantages

Provides an easy way to create and deploy containers.

A Lightweight solution comparing to VMs.

Fast startup/shutdown (flexible): order of milliseconds.

Does not depend on libraries on target platform.

Docker Disadvantages

Containers, including Docker containers, are considered less secure than VMs.

Work is done by RedHat to enhance Docker security with SELinux.

There is no standard set of security tests for VMs/containers.

In OpenVZ they did a security audit in 2005 by a Russian Security expert.

Docker containers on a single host must share the same kernel image.

Docker handles containers **individually**, there is no management/provisioning of containers in Docker.

You can link Docker containers (using the **--link** flag), but this provides only exposing some environment variables between containers and entries in /etc/hosts.

The Docker Swarm project (for containers orchestration) is quite new; it is a very basic project comparing to Kubernetes.

Google is using containers for over a decade. The **cgroups** kernel subsystem (resource allocation and tracking) was started by Google in 2006. Later on, in 2013, Google released a container open source project (**lmctfy**) in C++, based on cgroups in a beta stage.

Google claims that is starts each week over 2 billion containers.

which is over 3300 per second

These are not Kubernetes containers, but google proprietary containers.

The **Kubernetes** open source project (started by Google) provides a container management framework. It is based currently only on **Docker containers**, but other types of containers will be supported (like CoreOS rocket containers, which is currently being implemented).

https://github.com/googlecloudplatform/kubernetes

What does the word **kubernetes** stand for?

Greek for shipmaster or helmsman of a ship

Also sometimes being referred to as:

kube or **K8s** (that's 'k' + 8 letters + 's‘)

An open source project, Container Cluster Manager.

Still in a **pre production, beta phase**.

Announced and first released in **June 2014**.

302 contributors to the project on github.

Most developers are from **Google**.

RedHat is the second largest contributor.

Contributions also from Microsoft, HP, IBM, VMWare, CoreOS, and more.

There are already rpms for Fedora and REHL 7.

Quite small rpm – comprises of 60 files, for example, in Fedora.

6 configuration files reside in a central location: **/etc/kubernetes.**

Can be installed on a single laptop/desktop/server, or a group of few servers, or a group of hundreds of servers in a datacenter/cluster.

Other orchestration frameworks for containers exist:

- Docker-Swarm
- Core-OS Fleet
- Apache Mesos Docker Orchestration

The Datacenter as a Computer

The theory of google cluster infrastructure is described in a whitepaper:

The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines (2009), 60 pages, by Luiz André Barroso, and Urs Hölzle.

http://www.cs.berkeley.edu/~rxin/db-papers/WarehouseScaleComputing.pdf

"The software running on these systems, such as Gmail or Web search services, execute at a **scale far beyond a single machine or a single rack:** they run on no smaller a unit than clusters **of hundreds to thousands of individual servers**."

Focus is on the applications running in the datacenters.

Internet services must achieve high availability, typically aiming for at least 99.99% uptime (**about an hour of downtime per year**).

Proprietary Google infrastructure:

Google Omega architecture: http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/41684.pdf

Google Borg:

A paper published last week:

"Large-scale cluster management at Google with Borg"

http://research.google.com/

"Google's Borg system is a **cluster manager** that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up **to tens of thousands of machines**."

Kubernetes abstractions

Kubernetes provides a set of abstractions to manages containers:

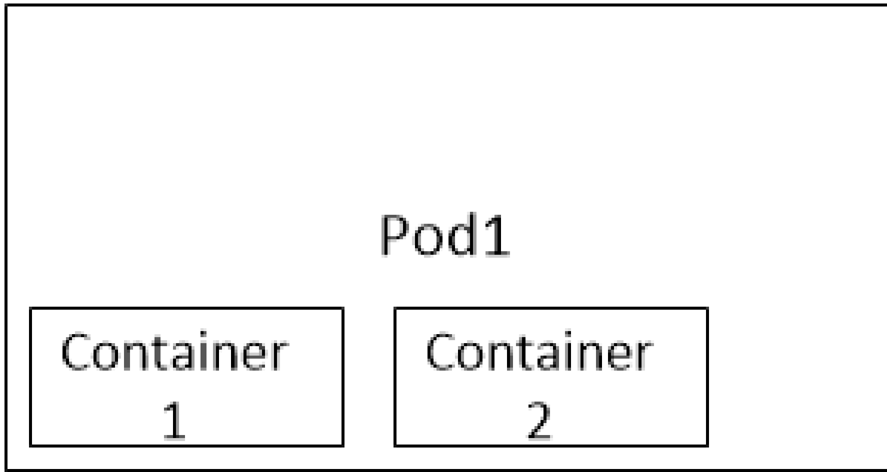
POD – the basic unit of Kubernetes. Consists of several docker containers (though it can be also a single container).

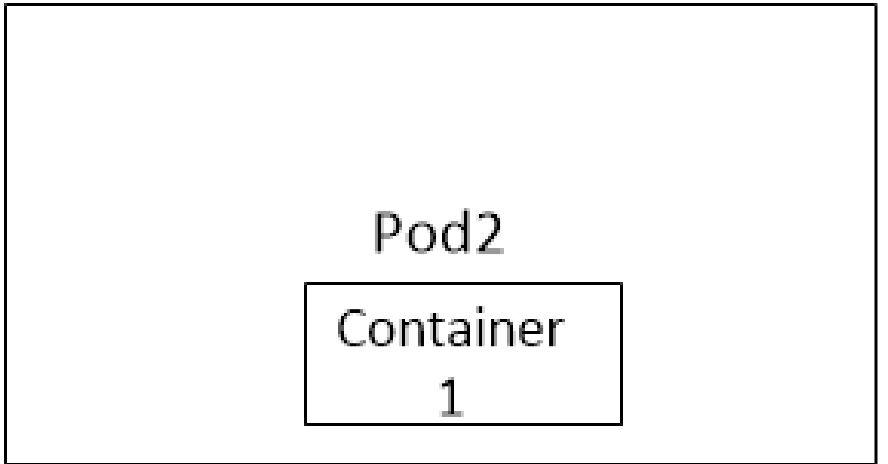
In Kubernetes, all containers run inside pods.

A pod can host a single container, or multiple cooperating containers

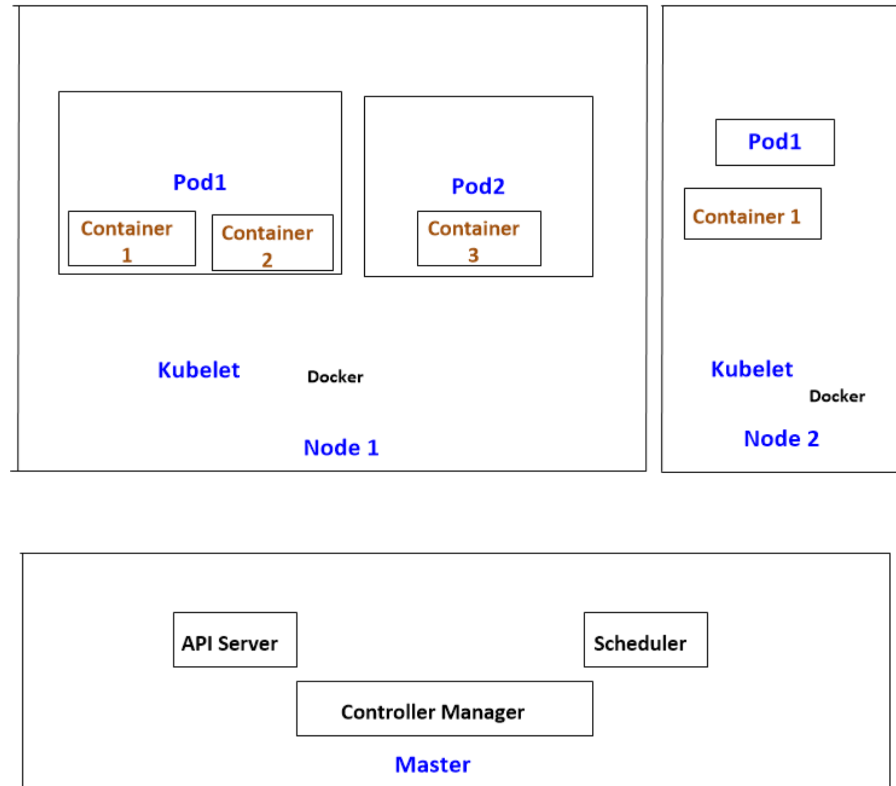
All the containers in a POD are in the same network namespace.

A pod has a single unique IP address (allocated by Docker).


Pod1
Container
1
Container
2


Pod2
Container
1

Kubernetes cluster



POD api (***kubectl)***

Creating a pod is done by ***kubectl create -f configFile***

The configFile can be a ***json*** or an ***yaml*** file.

This request, as well as other kubectl requests, is translated into http POST requests.

pod1.yaml

apiVersion: v1beta3

kind: Pod

metadata:

  name: www

spec:

  containers:

    - name: nginx

      image: dockerfile/nginx

POD api (***kubectl) - continued***

https://github.com/GoogleCloudPlatform/kubernetes/blob/master/examples/walkthrough/v1beta3/pod1.yaml

Currently you can create multiple containers by a single pod config file only with json config file.

Note that when you create a pod, you do not specify on which node (in which machine) it will be started. This is decided by the scheduler.

Note that you can create copies of the same pod with the ReplicationController – (will be discussed later)

Delete a pod:

kubectl delete configFile – remove a pod.

Delete all pods:

kubectl delete pods –all

POD api (***kubectl) - continued***

List all pods (with status information):

kubectl get pods

Lists all pods who’s name label matches 'nginx':

kubectl get pods -l name=nginx

Nodes (Minion)

Node – The basic compute unit of a cluster.

- previously called Minion.

Each node runs a daemon called kubelet.

Should be configured in */etc/kubernetes/kubelet*

Each node also runs the docker daemon.

Creating a node is done on the master by:

kubectl create –f nodConfigFile

Deleting a node is done on the master by:

kubectl delete –f nodConfigFile

Sys Admin can choose to make the node **unschedulable** using kubectl. Unscheduling the node will not affect any existing pods on the node but it will disable creation of any new pods on the node.

kubectl update nodes 10.1.2.3 --patch='{"apiVersion": "v1beta1", "unschedulable": true}'

Replication Controller

Replication Controller - Manages replication of pods.

A pod factory.

Creates a set of pods

Ensures that a required specified number of pods are running

Consists of:

Count – Kubernetes will keep the number of copies of pods matching the label selector. If too few copies are running the replication controller will start a new pod somewhere in the cluster

Label Selector

RplicationController yaml file

```
apiVersion: v1beta3
kind: ReplicationController
metadata:
  name: nginx-controller
spec:
  replicas: 3
  # selector identifies the set of Pods that this
  # replicaController is responsible for managing
  selector:
    name: nginx
```

```
template:
  metadata:
    labels:
      # Important: these labels need to match the selector above
      # The api server enforces this constraint.
      name: nginx
  spec:
    containers:
      - name: nginx
        image: dockerfile/nginx
          ports:
          - containerPort: 80
```

Master

The **master** runs 4 daemons: (via **systemd** services)

kube-apiserver

- Listens to http requests on port 8080.

kube-controller-manager

- Handles the replication controller and is responsible for adding/deleting pods to reach desired state.

kube-scheduler

- Handles scheduling of pods.

Etcd

- Distributed key-value store

Label

Currently used mainly in 2 places

- Matching pods to replication controllers
- Matching pods to services

Service

Service – for load balancing of containers.

Example - Service Config File:

```
apiVersion: v1beta3
kind: Service
metadata:
  name: nginx-example
spec:
  ports:
    - port: 8000 # the port that this service should serve on
      # the container on each pod to connect to, can be a name
      # (e.g. 'www') or a number (e.g. 80)
```

```
targetPort: 80
      protocol: TCP
 # just like the selector in the replication controller,
  # but this time it identifies the set of pods to load balance
  # traffic to.
  selector:
    name: nginx
```

https://github.com/GoogleCloudPlatform/kubernetes/blob/master/examples/walkthrough/v1beta3/service.yaml

Appendix

Cgroup namespaces

Work is being done currently (2015) to add support for Cgroup namespaces.

This entails adding **CLONE_NEWCGROUP** flag to the already existing 6 CLONE_NEW* flags.

Development is done mainly by Aditya Kali and Serge Hallyn.

Links

http://kubernetes.io/

https://github.com/googlecloudplatform/kubernetes

https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/pods.md

https://github.com/GoogleCloudPlatform/kubernetes/wiki/User-FAQ

https://github.com/GoogleCloudPlatform/kubernetes/wiki/Debugging-FAQ

https://cloud.google.com/compute/docs/containers

https://github.com/GoogleCloudPlatform/kubernetes/blob/master/examples/walkthrough/v1beta3/replication-controller.yaml

Summary

We have looked briefly into the implementation guidelines of Linux namespaces and cgroups in the kernel, as well as some examples of usage from userspace. These two subsystems seem indeed to give a lightweight solution for virtualizing Linux processes and as such they form a good basis for Linux containers projects (Docker, LXC, and more).

Thank you!