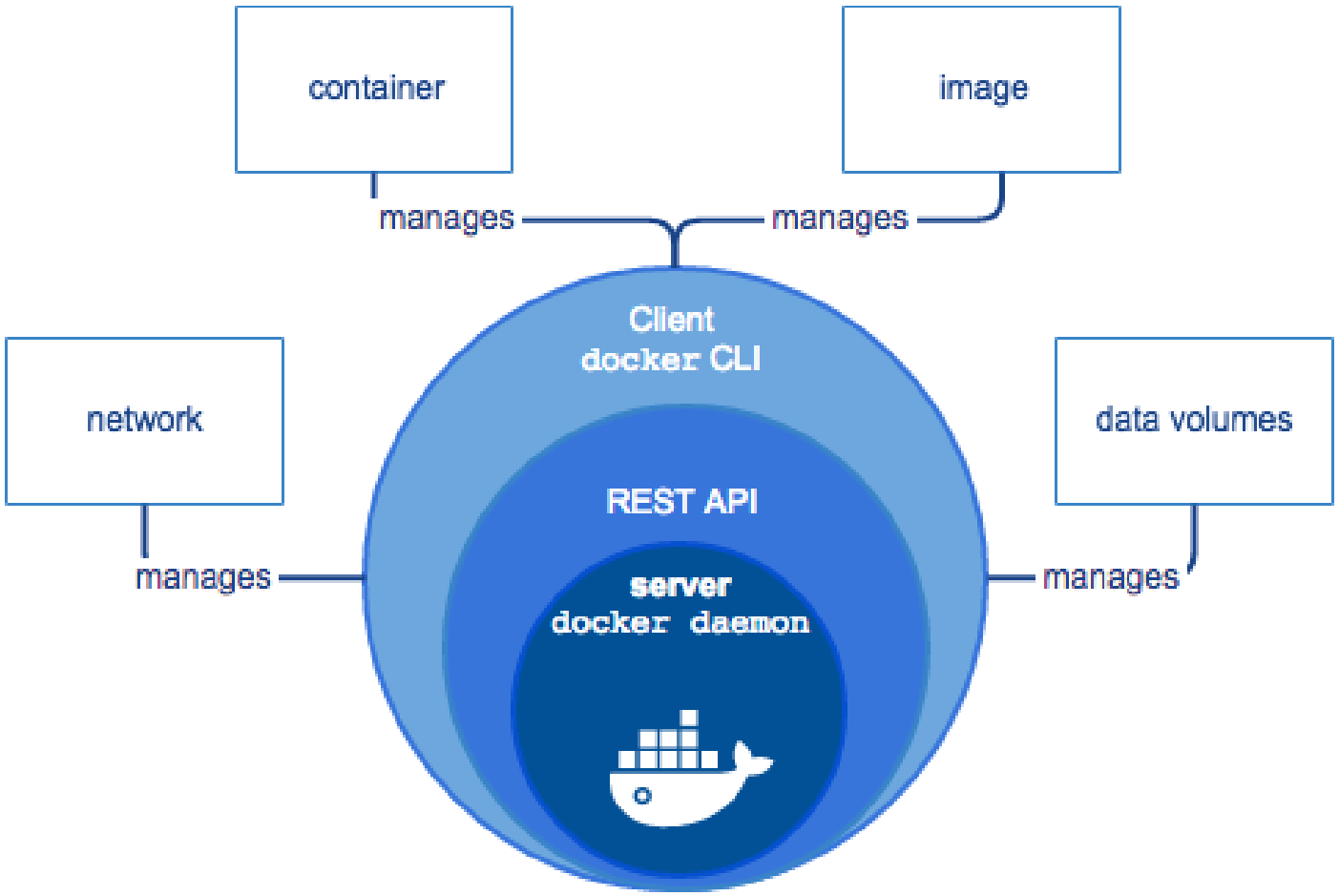
Linux Administration: K8S

Introduction to virtualization

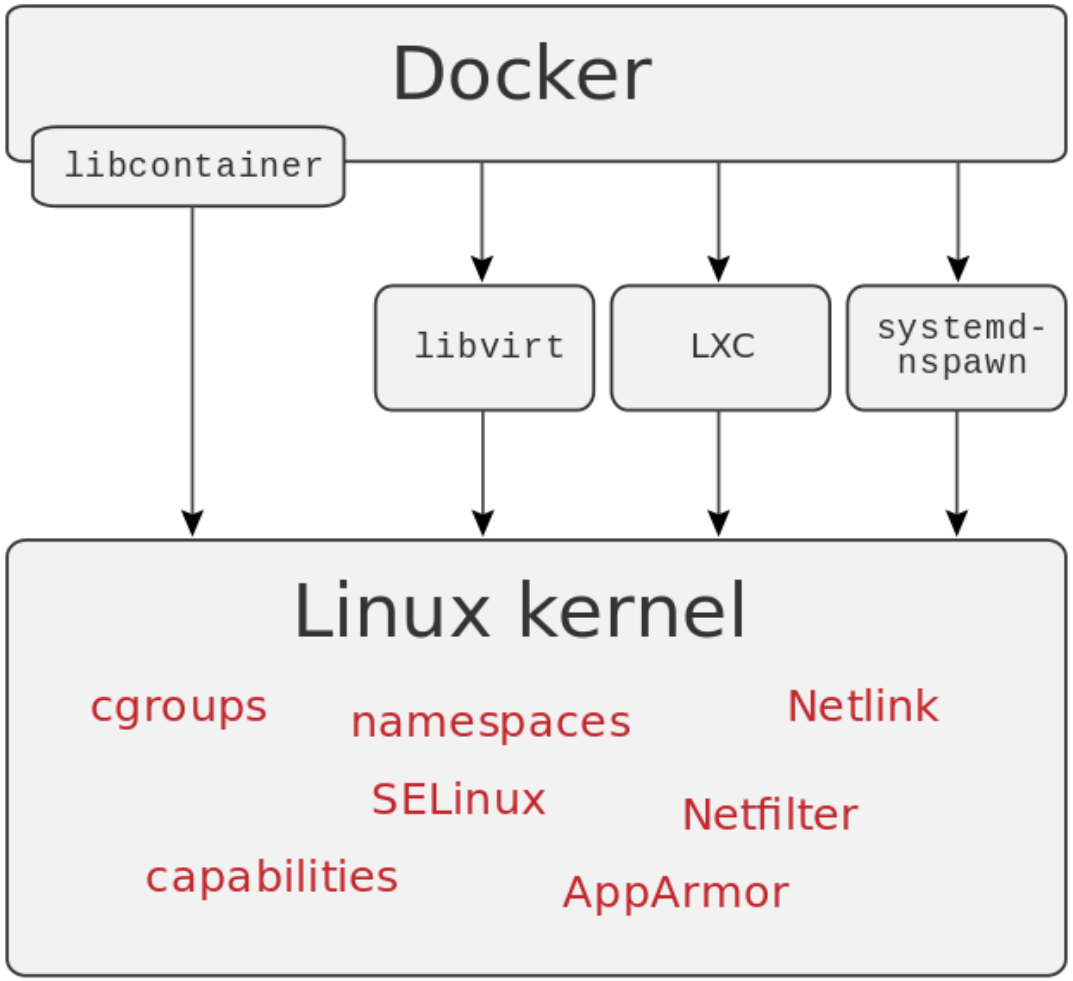
Introduction to Docker

Get Docker

Docker Engine



Docker Engine



Get Docker : Software platform

- Operation Systems:
  - Linux (RedHat / CentOS / Fedora, Debian / Ubuntu, SuSe)
  - Microsoft Windows (desktop and server)
  - MacOS
- Cloud :
  - Docker for Azure
  - Docker for AWS

Introduction to Docker

Docker concepts

Docker namespaces

- Docker Engine uses namespaces such as the following on Linux:
  - The pid namespace: Process isolation (PID: Process ID).
  - The net namespace: Managing network interfaces (NET: Networking).
  - The ipc namespace: Managing access to IPC resources (IPC: InterProcess Communication).
  - The mnt namespace: Managing filesystem mount points (MNT: Mount).
  - The uts namespace: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

Docker concepts

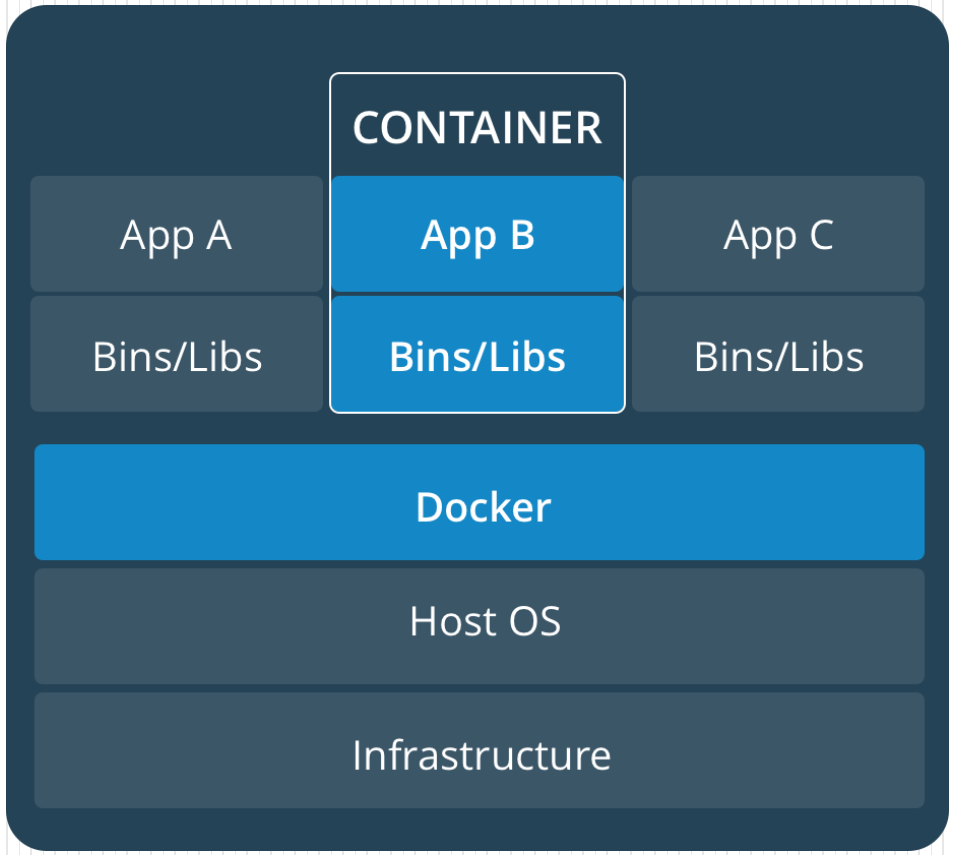
- Docker is a platform for developers and sysadmins to **develop, deploy, and run** applications with containers.
- Containerization is increasingly popular because containers are:
  - Flexible: Even the most complex applications can be containerized.
  - Lightweight: Containers leverage and share the host kernel.
  - Interchangeable: You can deploy updates and upgrades on-the-fly.
  - Portable: You can build locally, deploy to the cloud, and run anywhere.
  - Scalable: You can increase and automatically distribute container replicas.
  - Stackable: You can stack services vertically and on-the-fly.

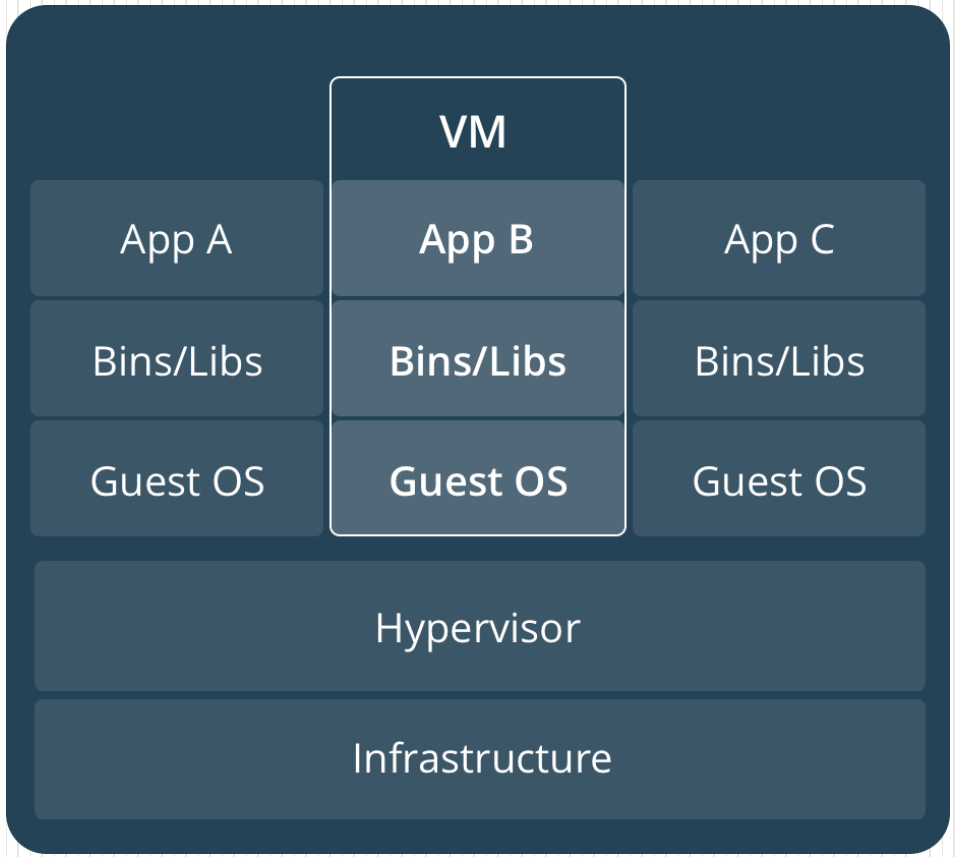
Images and containers

- A container is launched by running an image. An image is an executable package that includes everything needed to run an application--the code, a runtime, libraries, environment variables, and configuration files.

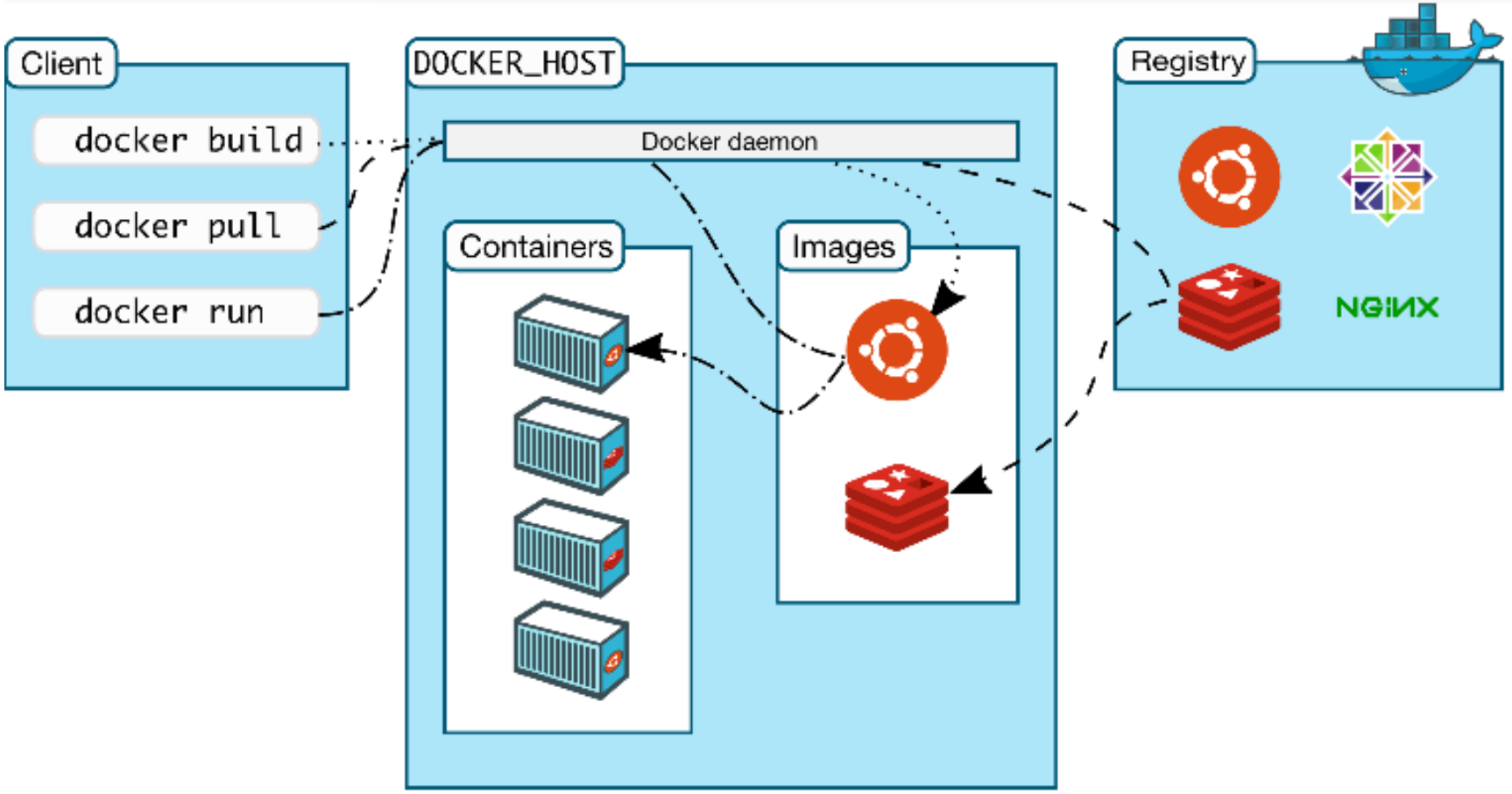
- A container is a runtime instance of an image--what the image becomes in memory when executed (that is, an image with state, or a user process). You can see a list of your running containers with the command, docker ps, just as you would in Linux.

Containers and virtual machines





Docker concepts



Introduction to Docker

Namespaces

Rami Rosen

https://www.andrew.cmu.edu/course/14-712-s20/applications/ln/Namespaces_Cgroups_Conatiners.pdf

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.

Building docker images

Dockerfile

Writing Dockerfiles

- A Docker image consists of read-only layers each of which represents a Dockerfile instruction. The layers are stacked and each one is a delta of the changes from the previous layer. Consider this Dockerfile:
- Each instruction creates one layer:

  FROM creates a layer from the ubuntu:18.04 Docker image.

  COPY adds files from your Docker client's current directory.

  RUN builds your application with make.

  CMD specifies what command to run within the container.

Pipe Dockerfile through stdin

- Docker has the ability to build images by piping Dockerfile through stdin with a local or remote build context. Piping a Dockerfile through stdin can be useful to perform one-off builds without writing a Dockerfile to disk, or in situations where the Dockerfile is generated, and should not persist afterwards.
- The following commands are equivalent:

```
echo -e 'FROM busybox\nRUN echo "hello world"' | docker build -

docker build -<<EOF
FROM busybox
RUN echo "hello world"
EOF
```

Exclude with .dockerignore

- To exclude files not relevant to the build (without restructuring your source repository) use a .dockerignore file. This file supports exclusion patterns similar to .gitignore files. For information on creating one, see the .dockerignore file.

```
# comment
*/temp*
*/*/temp*
temp?
*.md
!README.md
```

Understand how ARG and FROM interact

- FROM instructions support variables that are declared by any ARG instructions that occur before the first FROM.

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app

FROM extras:${CODE_VERSION}
CMD  /code/run-extras
```

Understand how ARG and FROM interact

- An ARG declared before a FROM is outside of a build stage, so it can't be used in any instruction after a FROM. To use the default value of an ARG declared before the first FROM use an ARG instruction without a value inside of a build stage:

```
ARG VERSION=latest
FROM busybox:$VERSION
ARG VERSION
RUN echo $VERSION > image_version
```

EXPOSE Usage

- The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

```
EXPOSE 80/tcp
```

- By default, EXPOSE assumes TCP. You can also specify UDP:

```
EXPOSE 80/udp
```

Exec form ENTRYPOINT

- The following Dockerfile shows using the ENTRYPOINT to run Apache in the foreground (i.e., as PID 1):

```
FROM debian:stable
RUN apt-get update && apt-get install -y --force-yes apache2
EXPOSE 80 443
VOLUME ["/var/www", "/var/log/apache2", "/etc/apache2"]
ENTRYPOINT ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

ENV Usage

- The ENV instruction sets the environment variable <key> to the value <value>. This value will be in the environment for all subsequent instructions in the build stage and can be replaced inline in many as well.
- Allows for multiple variables to be set at one time

```
ENV myName="John Doe" myDog=Rex\ The\ Dog  myCat=fluffy
```

- Set single variable to a value

```
ENV myName John Doe
ENV myDog Rex The Dog
ENV myCat fluffy
```

ADD Usage

- The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

```
ADD hom* /mydir/
ADD hom?.txt /mydir/
ADD --chown=55:mygroup files* /somedir/
ADD --chown=bin files* /somedir/
```

ADD vs COPY

- ADD lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination.

WORKDIR Usage

- The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile. If the WORKDIR doesn't exist, it will be created even if it's not used in any subsequent Dockerfile instruction.
- The WORKDIR instruction can be used multiple times in a Dockerfile. If a relative path is provided, it will be relative to the path of the previous WORKDIR instruction.

```
WORKDIR /a
WORKDIR b
WORKDIR c
RUN pwd
```

SHELL Usage

- The SHELL instruction allows the default shell used for the shell form of commands to be overridden. The default shell on Linux is ["/bin/sh", "-c"], and on Windows is ["cmd", "/S", "/C"]. The SHELL instruction must be written in JSON form in a Dockerfile.

```
# Executed as cmd /S /C echo default
        RUN echo default
# Executed as cmd /S /C powershell -command Write-Host default
        RUN powershell -command Write-Host default
# Executed as powershell -command Write-Host hello
        SHELL ["powershell", "-command"]
        RUN Write-Host hello
# Executed as cmd /S /C echo hello
        SHELL ["cmd", "/S", "/C"]
        RUN echo hello
```

Build from scratch

- While scratch appears in Docker's repository on the hub, you can't pull it, run it, or tag any image with the name scratch.
- Instead, you can refer to it in your Dockerfile.
- For example, to create a minimal container using scratch:

- FROM scratch
- ADD hello /
- CMD ["/hello"]

Multiple service images

- If you need to run more than one service within a container, you can accomplish this in a few different ways.
- Put all of your commands in a wrapper script, complete with testing and debugging information. Run the wrapper script as your CMD. This is a very naive example. First, the wrapper script:

- ./my_first_process -D
- status=$?
- if [$status -ne 0]; then
- echo "Failed to start my_first_process: $status"
- exit $status
- fi

Multiple service images

- Next, the Dockerfile:

- FROM ubuntu:latest
- COPY my_first_process my_first_process
- COPY my_second_process my_second_process
- COPY my_wrapper_script.sh my_wrapper_script.sh
- CMD ./my_wrapper_script.sh

Multiple service images

- Use a process manager like supervisord. This is a moderately heavy-weight approach that requires you to package supervisord and its configuration in your image (or base your image on one that includes supervisord), along with the different applications it manages.
- FROM ubuntu:latest
- RUN apt-get update && apt-get install -y supervisor
- RUN mkdir -p /var/log/supervisor
- COPY supervisord.conf /etc/supervisor/conf.d/supervisord.conf
- COPY my_first_process my_first_process
- COPY my_second_process my_second_process
- CMD ["/usr/bin/supervisord"]

Build docker

- Create docker image

```
docker build --tag=friendlyhello .
```

Building docker images

Multi-stage builds

Use multi-stage builds

- Multi-stage builds are a new feature requiring Docker 17.05 or higher on the daemon and client.
- Multistage builds are useful to anyone who has struggled to optimize Dockerfiles while keeping them easy to read and maintain.
- It was actually very common to have one Dockerfile to use for development (which contained everything needed to build your application), and a slimmed-down one to use for production, which only contained your application and exactly what was needed to run it.
- This has been referred to as the “builder pattern”. Maintaining two Dockerfiles is not ideal.

Without multi-stage builds

- #!/bin/sh
- echo Building alexellis2/href-counter:build. Stage 1
- docker build -t alexellis2/href-counter:build . -f Dockerfile.build

- docker container create --name extract alexellis2/href-counter:build
- docker container cp extract:/go/src/github.com/alexellis/href-counter/app ./app
- docker container rm -f extract

- echo Building alexellis2/href-counter:latest. Stage 2

- docker build --no-cache -t alexellis2/href-counter:latest .
- rm ./app

With multi-stage builds

- FROM golang:latest AS builder
- WORKDIR /go/src/github.com/alexellis/href-counter/
- RUN go get -d -v golang.org/x/net/html
- COPY app.go .
- RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .

- FROM alpine:latest
- RUN apk --no-cache add ca-certificates
- WORKDIR /root/
- COPY --from=builder /go/src/github.com/alexellis/href-counter/app .
- CMD ["./app"]

Using standard images

- When using multi-stage builds, you are not limited to copying from stages you created earlier in your Dockerfile.
- You can use the COPY --from instruction to copy from a separate image, either using the local image name, a tag available locally or on a Docker registry, or a tag ID.
- The Docker client pulls the image if necessary and copies the artifact from there. The syntax is:

- COPY --from=nginx:latest /etc/nginx/nginx.conf /nginx.conf

Stop building process

- When you build your image, you don’t necessarily need to build the entire Dockerfile including every stage.
- You can specify a target build stage.
- The following command assumes you are using the previous Dockerfile but stops at the stage named builder:

- docker build --target builder -t alexellis2/href-counter:latest .

With multi-stage builds

- You can pick up where a previous stage left off by referring to it when using the FROM directive

- FROM alpine:latest as builder
- RUN apk --no-cache add build-base

- FROM builder as build1
- COPY source1.cpp source.cpp
- RUN g++ -o /binary source.cpp

- FROM builder as build2
- COPY source2.cpp source.cpp
- RUN g++ -o /binary source.cpp

Run application in production

Docker run examples

PID settings (--pid)

- By default, all containers have the PID namespace enabled.
- PID namespace provides separation of processes. The PID Namespace removes the view of the system processes, and allows process ids to be reused including pid 1.
- You want to use some tools when debugging processes within the container.
- Joining another container's pid namespace can be used for debugging that container.

```
docker run -it --rm --pid=host myhtop
```

IPC settings (--ipc)

- The following values are accepted:

”” Use daemon’s default.

“none” Own private IPC namespace, with /dev/shm not mounted.

“private” Own private IPC namespace.

“shareable” Own private IPC namespace, with a possibility to share it with other containers.

“container: <_name-or-ID_>" Join another (“shareable”) container’s IPC namespace.

“host” Use the host system’s IPC namespace.

Network settings

- With the network set to container a container will share the network stack of another container.
- The other container's name must be provided in the format of --network container:<name|id>.
- Note that --add-host --hostname --dns --dns-search --dns-option and --mac-address are invalid in container netmode, and --publish --publish-all --expose are also invalid in container netmode.

```
docker run --rm -it --network container:redis \
    example/redis-cli -h 127.0.0.1
```

Clean up (--rm)

- By default a container's file system persists even after the container exits. This makes debugging a lot easier (since you can inspect the final state) and you retain all your data by default. But if you are running short-term foreground processes, these container file systems can really pile up.
- If instead you'd like Docker to automatically clean up the container and remove the file system when the container exits, you can add the --rm flag:
- --rm=false: Automatically remove the container when it exits

Overriding Dockerfile image defaults

- Four of the Dockerfile commands cannot be overridden at runtime: FROM, MAINTAINER, RUN, and ADD.
- Everything else has a corresponding override in docker run. We'll go through what the developer might have set in each Dockerfile instruction and how the operator can override that setting.
  - CMD (Default Command or Options)
  - ENTRYPOINT (Default Command to Execute at Runtime)
  - EXPOSE (Incoming Ports)
  - ENV (Environment Variables)
  - HEALTHCHECK
  - VOLUME (Shared Filesystems)
  - USER
  - WORKDIR

Overriding Dockerfile image defaults

- Here is an example of how to run a shell in a container that has been set up to automatically run something else (like /usr/bin/redis-server):

```
docker run -it --entrypoint /bin/bash example/redis
```

- or two examples of how to pass more parameters to that ENTRYPOINT:

```
docker run -it --entrypoint /bin/bash example/redis -c ls -l
```

- You can reset a containers entrypoint by passing an empty string, for example:

```
docker run -it --entrypoint="" mysql bash
```

Overriding Dockerfile image defaults

- Here is an example of how to run a shell in a container that has been set up to automatically run something else (like /usr/bin/redis-server):

```
docker run -it --entrypoint /bin/bash example/redis
```

- or two examples of how to pass more parameters to that ENTRYPOINT:

```
docker run -it --entrypoint /bin/bash example/redis -c ls -l
```

- You can reset a containers entrypoint by passing an empty string, for example:

```
docker run -it --entrypoint="" mysql bash
```

Run application in production

Resource Management

Runtime constraints on resources

- User memory constraints.
- There is no memory limit for the container. The container can use as much memory as needed
- -m, --memory="" Memory limit (format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g. Minimum is 4M.
- --memory-swap="" Total memory limit (memory + swap, format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g.
- --memory-reservation="" Memory soft limit (format: <number>[<unit>]). Number is a positive integer. Unit can be one of b, k, m, or g.

Runtime constraints on resources

- We set memory limit only, this means the processes in the container can use 300M memory and 300M swap memory, by default, the total virtual memory size (--memory-swap) will be set as double of memory, in this case, memory + swap would be 2*300M, so processes can use 300M swap memory as well.

  docker run -it -m 300M ubuntu:14.04 /bin/bash

- We set both memory and swap memory, so the processes in the container can use 300M memory and 700M swap memory.

  docker run -it -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash

Runtime constraints on resources

- Memory reservation is a kind of memory soft limit that allows for greater sharing of memory. Under normal circumstances, containers can use as much of the memory as needed and are constrained only by the hard limits set with the -m/--memory option.
- When memory reservation is set, Docker detects memory contention or low memory and forces containers to restrict their consumption to a reservation limit.

```
docker run -it -m 500M --memory-reservation 200M \
    ubuntu:14.04 /bin/bash
```

CPU share constraint

- CPU period constraint
- If there is 1 CPU, this means the container can get 50% CPU worth of run-time every 50ms.

```
docker run -it --cpu-period=50000 --cpu-quota=25000 \
    ubuntu:14.04 /bin/bash
```

- cpu-period=50000 (50ms)
- cpu-quota=25000 (50% : 25000 of 50000)

CPU share constraint

- Cpuset constraint
- This means processes in container can be executed on cpu 1 and cpu 3.

```
docker run -it --cpuset-cpus="1,3" ubuntu:14.04 /bin/bash
```

- This means processes in container can be executed on cpu 0, cpu 1 and cpu 2.

```
docker run -it --cpuset-cpus="0-2" ubuntu:14.04 /bin/bash
```

Block IO bandwidth (Blkio) constraint

- By default, all containers get the same proportion of block IO bandwidth (blkio). This proportion is 500. To modify this proportion, change the container's blkio weight relative to the weighting of all other running containers using the --blkio-weight flag.
- The --blkio-weight flag can set the weighting to a value between 10 to 1000. For example, the commands below create two containers with different blkio weight:

```
docker run -it --name c1 --blkio-weight 300 ubuntu:14.04 /bin/bash
docker run -it --name c2 --blkio-weight 600 ubuntu:14.04 /bin/bash
```

Block IO bandwidth (Blkio) constraint

- The --blkio-weight-device="DEVICE_NAME:WEIGHT" flag sets a specific device weight.
- The DEVICE_NAME:WEIGHT is a string containing a colon-separated device name and weight.
- For example, to set /dev/sda device weight to 200:

```
docker run -it  --blkio-weight-device "/dev/sda:200"   Ubuntu
```

- If you specify both the --blkio-weight and --blkio-weight-device, Docker uses the --blkio-weight as the default weight and uses --blkio-weight-device to override this default with a new value on a specific device.

Block IO bandwidth (Blkio) constraint

- The --device-read-bps flag limits the read rate (bytes per second) from a device. For example, this command creates a container and limits the read rate to 1mb per second from /dev/sda:

```
docker run -it --device-read-bps /dev/sda:1mb ubuntu
```

- The --device-write-bps flag limits the write rate (bytes per second) to a device. For example, this command creates a container and limits the write rate to 1mb per second for /dev/sda:

```
docker run -it --device-write-bps /dev/sda:1mb ubuntu
```

Block IO bandwidth (Blkio) constraint

- The --device-read-iops flag limits read rate (IO per second) from a device. For example, this command creates a container and limits the read rate to 1000 IO per second from /dev/sda:

```
docker run -ti --device-read-iops /dev/sda:1000 ubuntu
```

- The --device-write-iops flag limits write rate (IO per second) to a device. For example, this command creates a container and limits the write rate to 1000 IO per second to /dev/sda:

```
docker run -ti --device-write-iops /dev/sda:1000 ubuntu
```

Run application in production

Swarm

Swarm Feature highlights

- Cluster management integrated with Docker Engine
- Decentralized design
- Declarative service model
- Scaling
- Desired state reconciliation
- Multi-host networking
- Service discovery
- Load balancing
- Secure by default
- Rolling updates

Swarm mode key concepts

- A swarm consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services).
- A node is an instance of the Docker engine participating in the swarm. You can also think of this as a Docker node.
- A service is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm.
- The swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm.

Create a swarm

- Open a terminal and ssh into the machine where you want to run your manager node.
- This tutorial uses a machine named manager1.

- Run the following command to create a new swarm:

```
docker swarm init --advertise-addr <MANAGER-IP>
```

Create a swarm

- Run the following command to create a new swarm (10.0.0.1 – ip address of the manager):

```
docker swarm init --advertise-addr 10.0.0.1
```

- To add a worker to this swarm, run the following command on the another node:

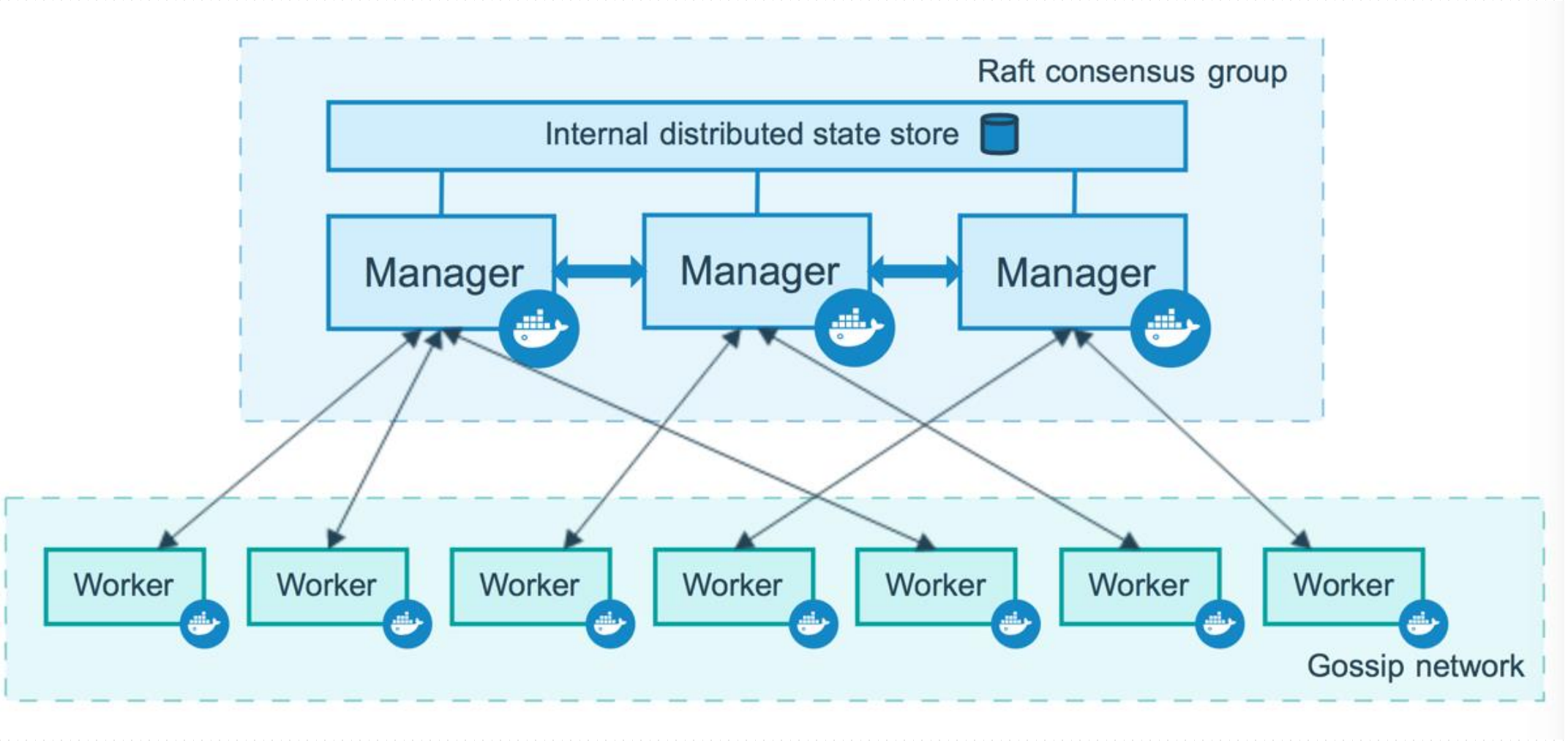
```
docker swarm join
--token SWMTKN-1-49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c  10.0.0.1:2377
```

Create a swarm

- Open a terminal and ssh into the machine where the manager node runs and run the docker node ls command to see the worker nodes:

```
docker node ls
ID                            HOSTNAME        STATUS  AVAILABILITY  MANAGER STATUS
03g1y59jwfg7cf99w4lt0f662     worker2   Ready                 Active
9j68exjopxe7wfl6yuxml7a7j     worker1   Ready                 Active
dxn1zf6l61qsb1josjja83ngz *   manager1  Ready                 Active        Leader
```

Create a fault tolerant swarm



Create a swarm: Change roles

- You can promote a worker node to be a manager by running docker node promote. For example, you may want to promote a worker node when you take a manager node offline for maintenance.

  docker node promote <node name>

- You can also demote a manager node to a worker node.

  docker node demote <node name>

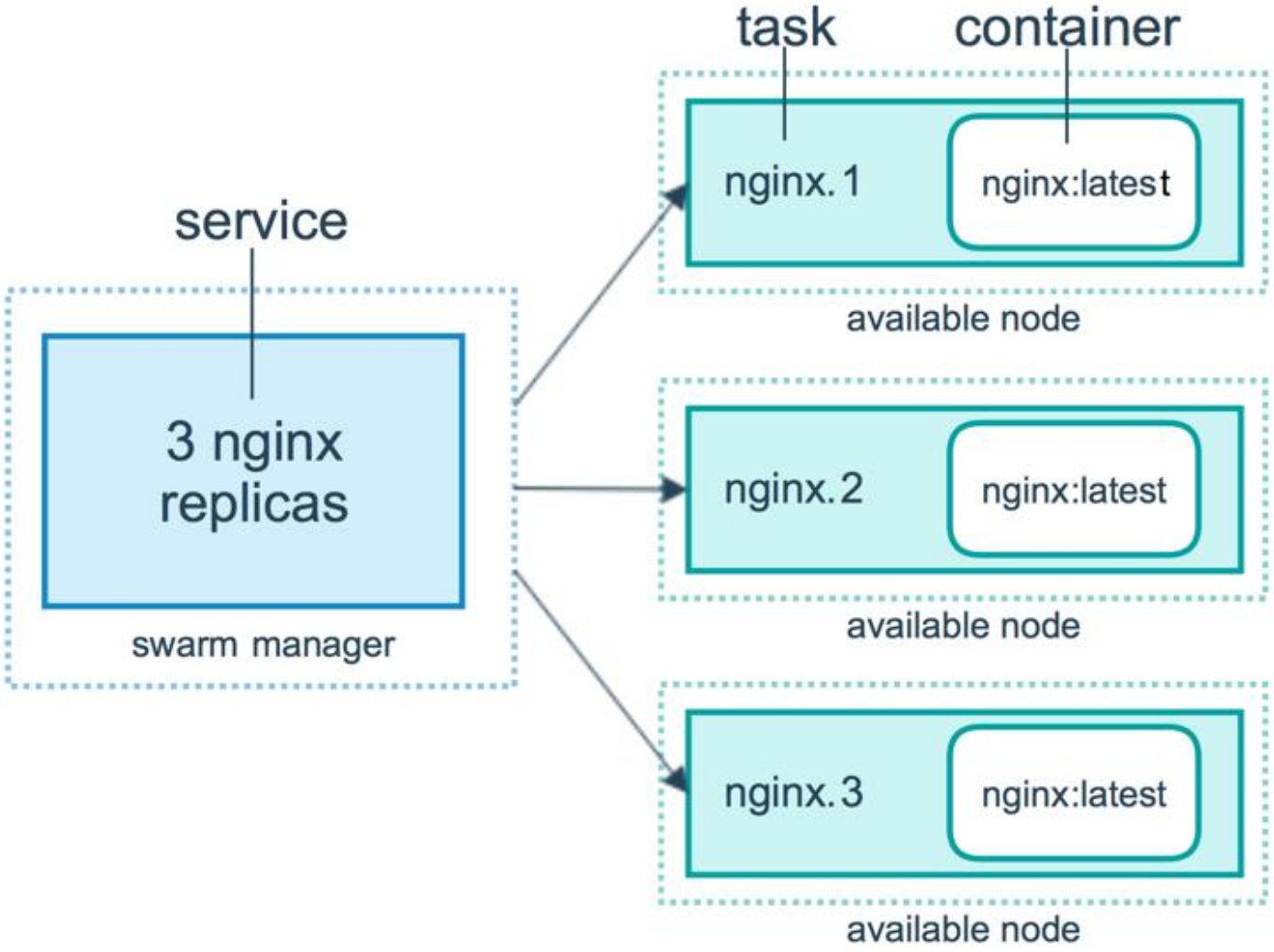
Run application in production

Services

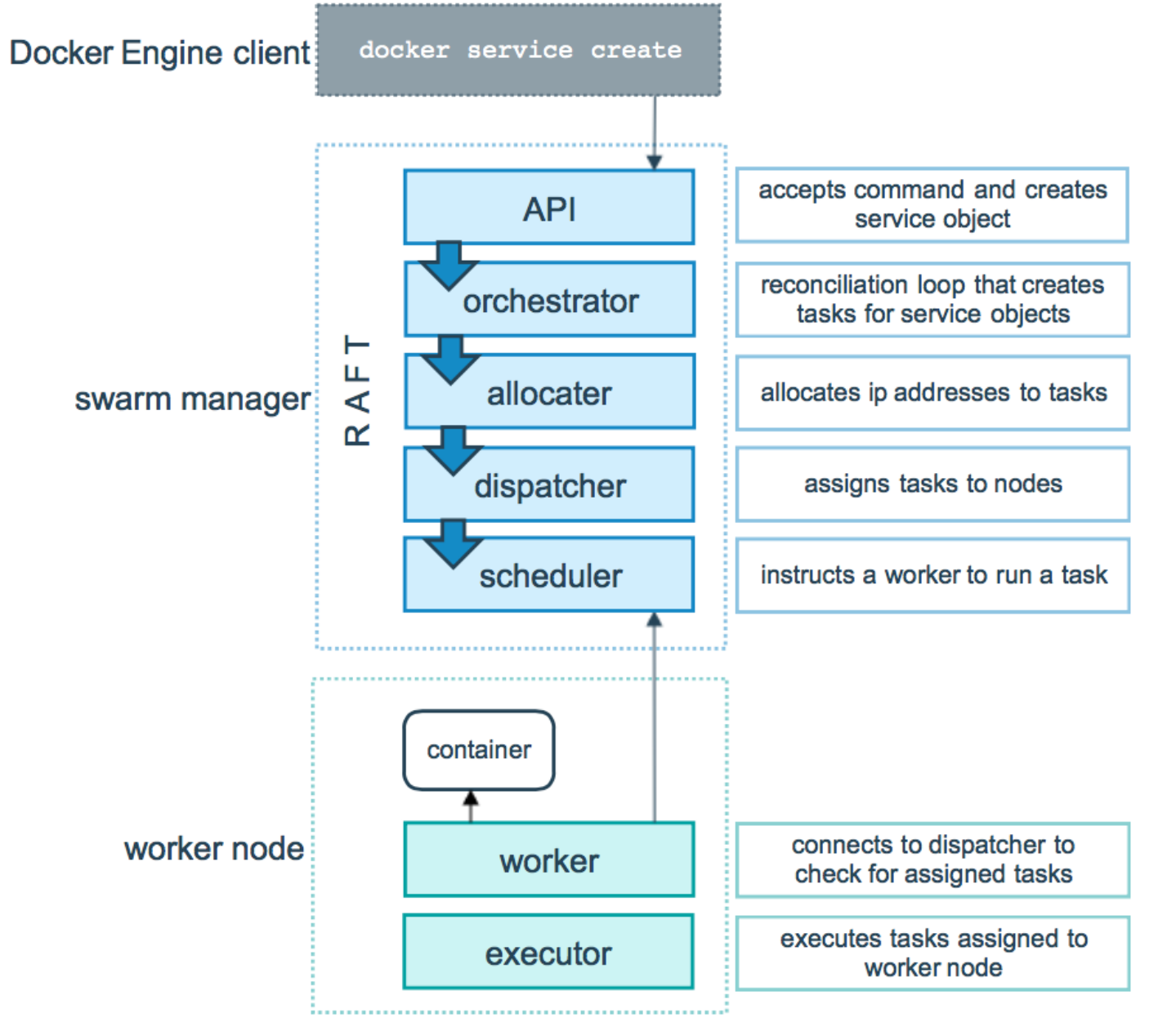
About services

- In a distributed application, different pieces of the app are called “services”.
- Services are really just “containers in production.”
- A service only runs one image, but it codifies the way that image runs—what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on.
- Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

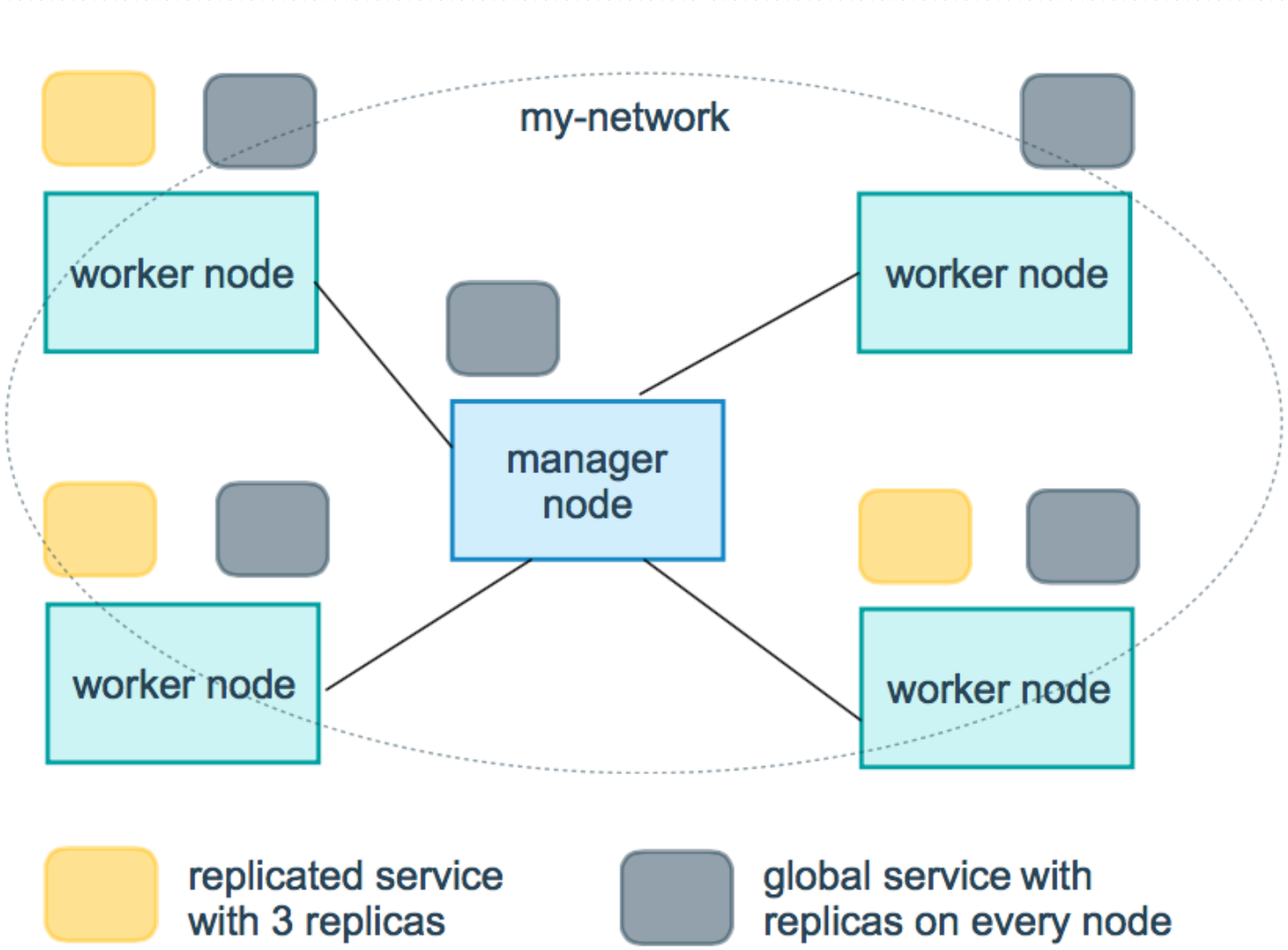
About services



Start service



Replicated and global services



Create a service

- Run the following command:

```
docker service create --replicas 1 \
        --name helloworld alpine ping docker.com
```

- The --name flag names the service helloworld.
- The --replicas flag specifies the desired state of 1 running instance.
- The arguments alpine ping docker.com define the service as an Alpine Linux container that executes the command ping docker.com.

Create a service

- Run docker service ls to see the list of running services:

```
docker service ls

ID                  NAME        SCALE  IMAGE   COMMAND
9uk4639qpg7n  helloworld  1/1    alpine  ping docker.com
```

Inspect and remove a service

- Run docker service inspect --pretty <SERVICE-ID> to display the details about a service in an easily readable format.

```
docker service inspect --pretty helloworld
```

- Run docker service rm helloworld to remove the helloworld service.

```
docker service rm helloworld
helloworld
```

Scale the service in the swarm

- Run the following command to change the desired state of the service running in the swarm:

```
docker service scale helloworld=5
helloworld scaled to 5

docker service ps helloworld
NAME                                    IMAGE   NODE     DESIRED STATE  CURRENT STATE
helloworld.1.8p1vev3fq5zm0mi8g0as41w35  alpine  worker2  Running        Running 7 minutes
helloworld.2.c7a7tcdq5s0uk3qr88mf8xco6  alpine  worker1  Running        Running 24 seconds
helloworld.3.6crl09vdcalvtfehfh69ogfb1  alpine  worker1  Running        Running 24 seconds
helloworld.4.auky6trawmdlcne8ad8phb0f1  alpine  manager1 Running        Running 24 seconds
helloworld.5.ba19kca06l18zujfwxyc5lkyn  alpine  worker2  Running        Running 24 seconds
```

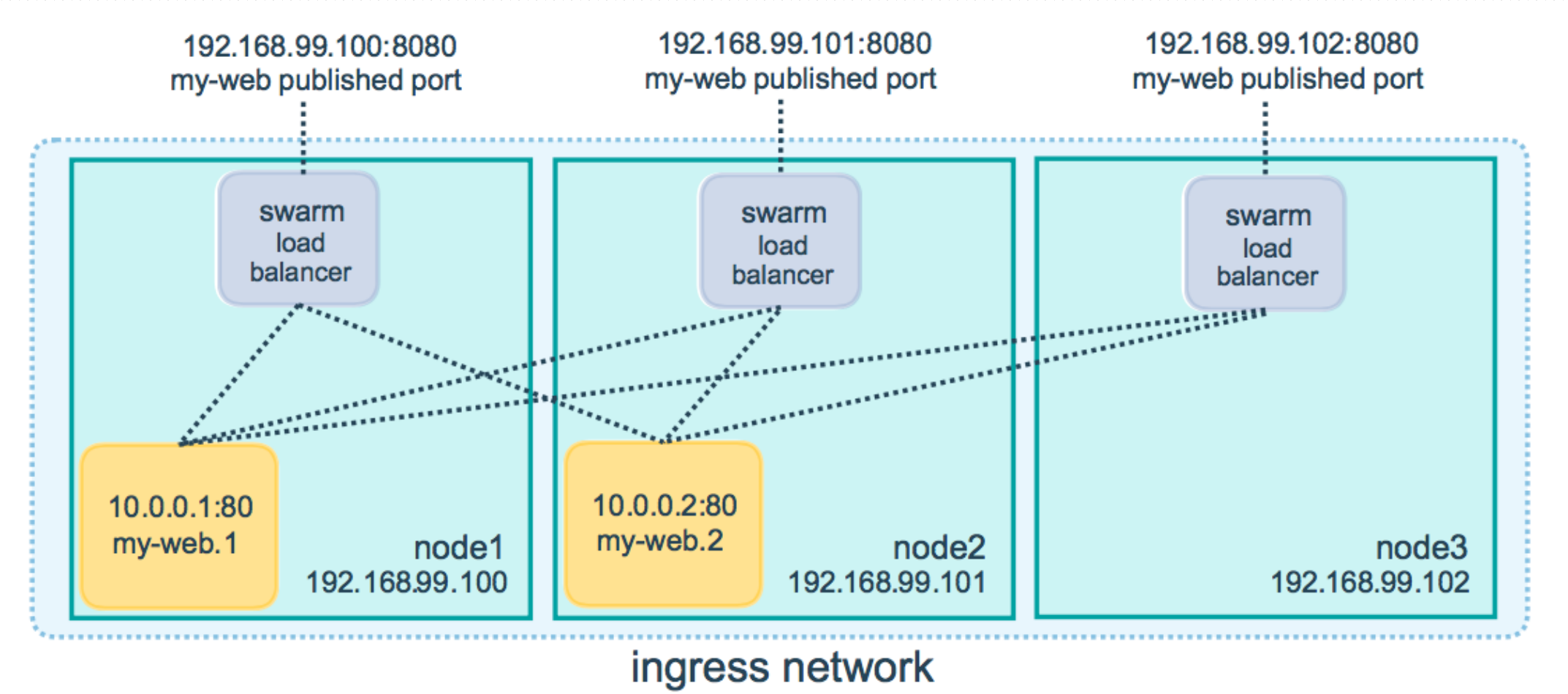
Drain a node on the swarm

- Sometimes, such as planned maintenance times, you need to set a node to DRAIN availability. DRAIN availability prevents a node from receiving new tasks from the swarm manager. It also means the manager stops tasks running on the node and launches replica tasks on a node with ACTIVE availability.

- Change docker node availability availability state:

```
docker node update --availability drain worker1
docker node update --availability active worker1
```

Publish a port for a service



Drain a node on the swarm

- The following command publishes port 80 in the nginx container to port 8080 for any node in the swarm:

```
docker service create --name my-web \
  --publish published=8080,target=80 --replicas 2 nginx
```

- You can publish a port for an existing service using the following command:

```
docker service update \
  --publish-add published=8080,target=80
```

Run application in production

Stacks

Using stacks

- A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together.
- A single stack is capable of defining and coordinating the functionality of an entire application (though very complex applications may want to use multiple stacks).

Define service docker-compose.yml

- version: "3"
- services:
- web:
- image: username/repo:tag
- deploy:
- replicas: 5
- resources:
- limits:
- cpus: "0.1"
- memory: 50M
- restart_policy:
- condition: on-failure
- ports:
- - "4000:80"
- networks:
- - webnet
- networks:
- webnet:

Create or update the stacked service

- Re-run the docker stack deploy command on the manager, and whatever services need updating are updated:

```
docker stack deploy -c docker-compose.yml getstartedlab
Updating service getstartedlab_web (id: angi1bf5e4to03qu9f93trnxm)
Creating service getstartedlab_visualizer (id: l9mnwkeq2jiononb5ihz9u7a4)
```

Docker-compose

- Create a new Dockerfile .

- FROM nginx:latest
- EXPOSE 80

Docker-compose

- Create docker-compose.yml .

```
version: "3.3"
services:
  web:
    build: .
    ports:
      - "8088:80"
```

Docker-compose

- Build and run your app with Compose

- From your project directory, start up your application by running docker-compose up.

- docker-compose up

Questions?

andry.cheredarchuk@gmail.com