Cloud based networks
Containers

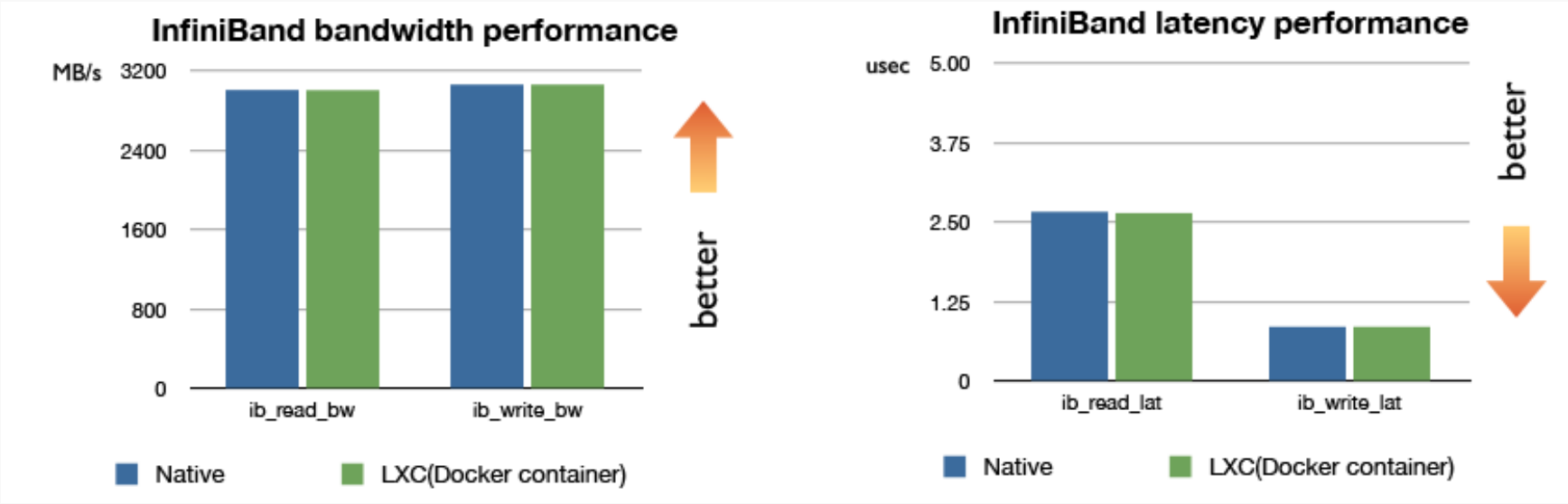
Simon Csaba

BME-TMIT

CONTAINERS

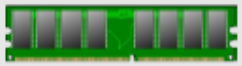
Virtualization and performance?

- Motivation:
  - Virtualization = smthg(runs_smthg)
    - Some kernel tasks are executed two times
  - Increase the performance: decrease the overhead of „smthg”



Container metaphore: logistics

- Logistic (management) problem
  - Many transport platforms, many product types
  - How many package variants are required?

	?	?	?	?	?	?
	?	?	?	?	?	?
	?	?	?	?	?	?
	?	?	?	?	?	?
	?	?	?	?	?	?
	?	?	?	?	?	?
						

Container metaphore: intermodal container

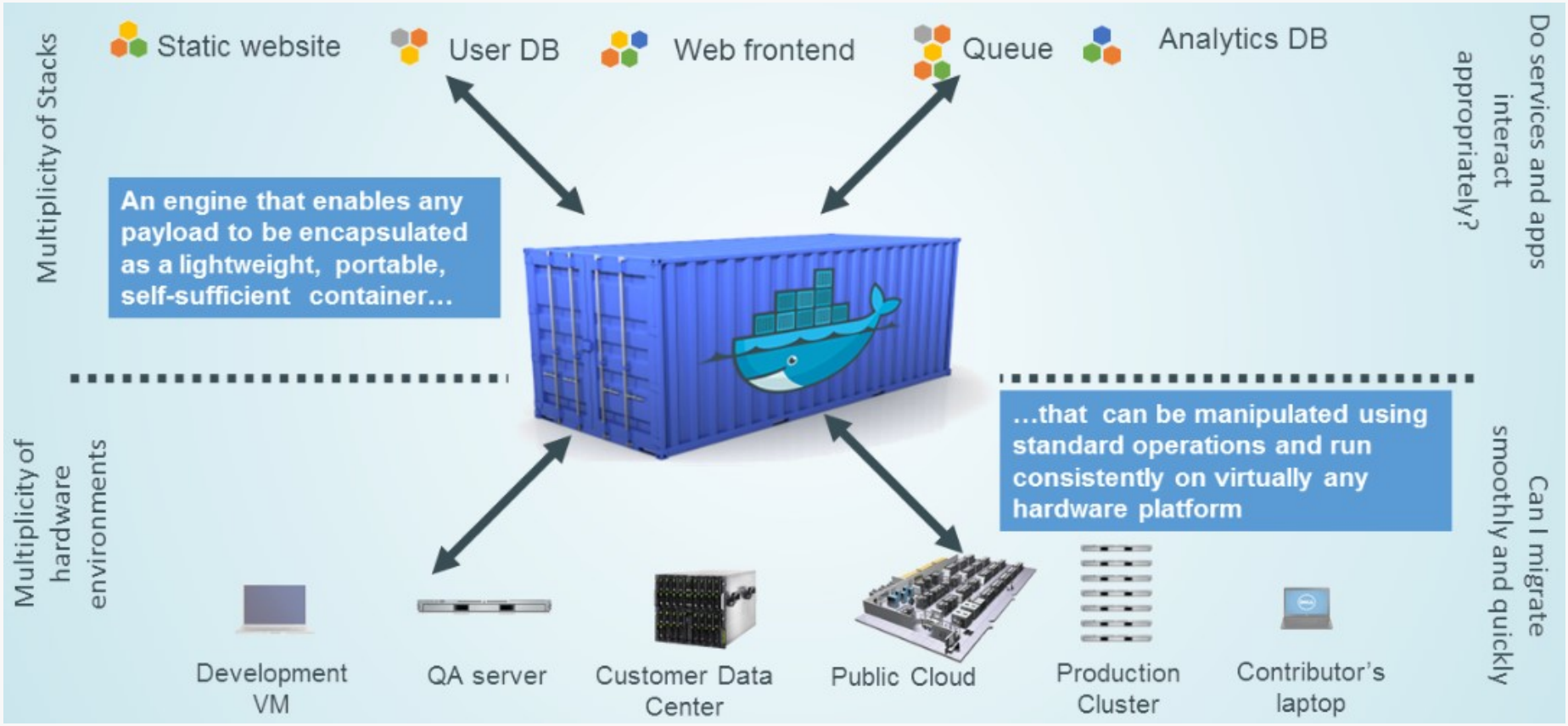
- Logistic (management) problem
  - Many transport platforms, many product types
  - How many package variants are required?
  - Use only one: the container, as a transportation standard

„Transporting” the code in virtualized systems

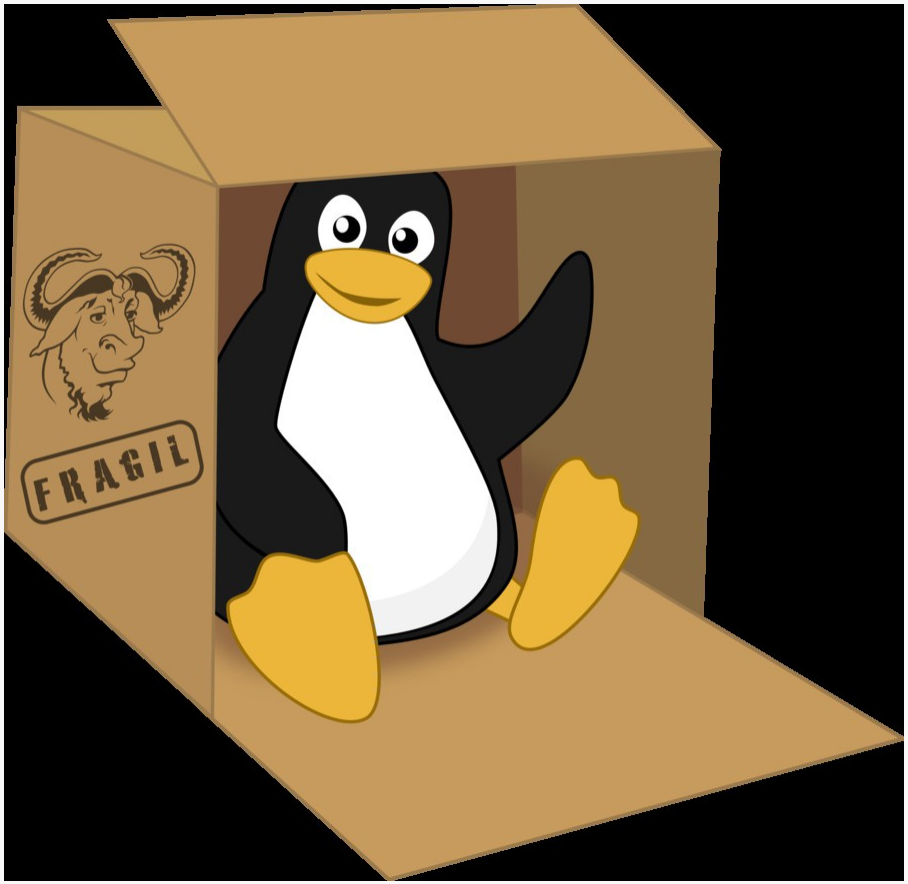
- Trasnportation platform => execution environment
- Product type => computation task

	?	?	?	?	?	?
	?	?	?	?	?	?
	?	?	?	?	?	?
	?	?	?	?	?	?
	?	?	?	?	?	?
	?	?	?	?	?	?

Application containers



Linux containers solve everything (hm..)



Paranthesis: case study
(Why should we replace clouds with Docker?)

Why Docker?

My World Needed To Change

- 5+ individual teams building "micro services" in Java and Scala
- Frictionless deployment of "micro-services" using Chef & AWS
- 25+ separate "micro-services" deployed in the previous 18 months
- Each service is typically deployed to a single AWS virtual machine
- Each service is deployed 6x - dev, test, staging (2x) and production (2x)
- 25+ "micro-services" became nearly 150 AWS virtual machines

Why Docker? COST!

The AWS bill is too damn high!

- Decline in the global price of oil causing churn in our business
- 6 AWS virtual machines per service isn't sustainable with our budget
- AWS monthly bill started to gain visibility from sr. management and ***the board***

Why Docker? WASTE!

We weren't using the compute and memory resources purchased from AMZN!

- Nearly all "micro-services" were at 1% CPU utilization
- Nearly all "micro-services' were only using 40% of memory (JVM)
- 150+ virtual machines essentially sitting idle

Why Docker? LOCK IN!

How would we leave AMZN if we wanted to?

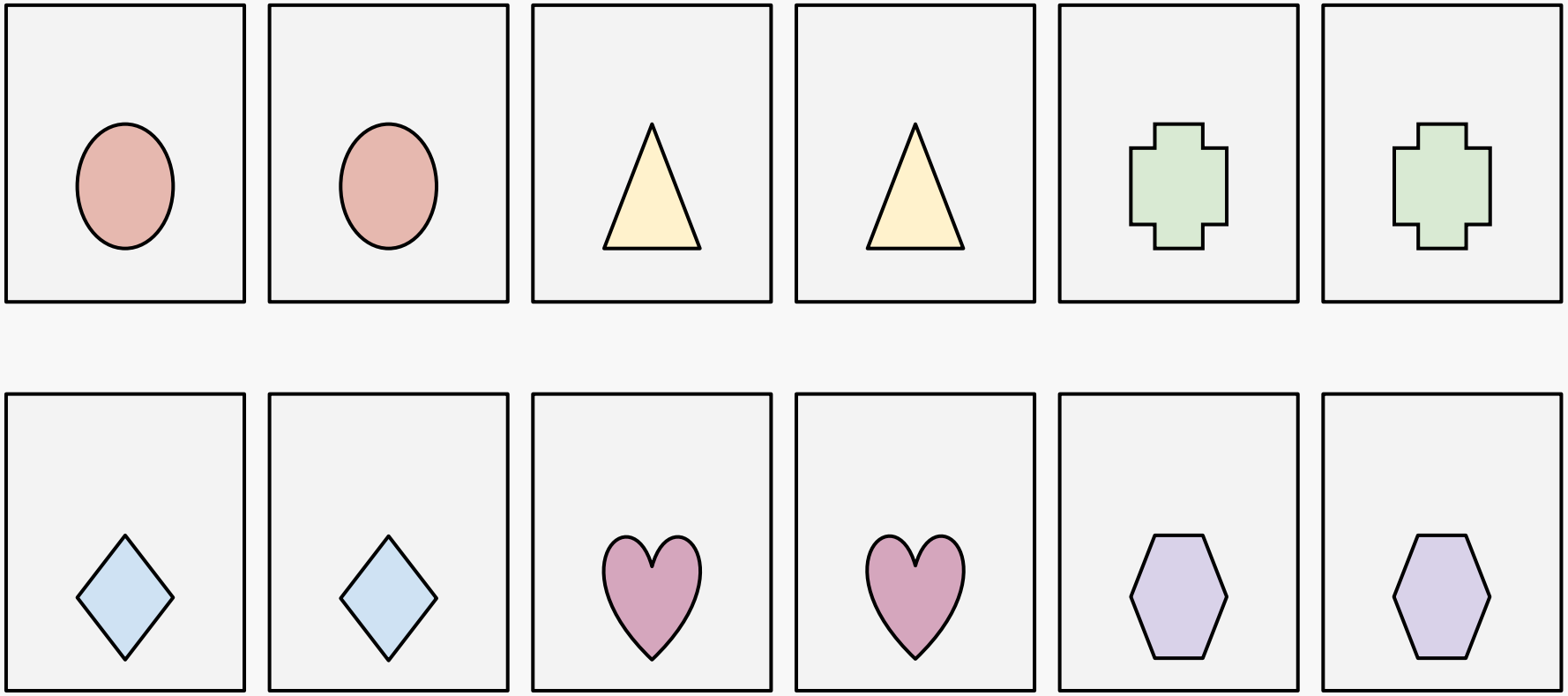
- Could we use Drillinginfo IT's Openstack platform?
- What about alternate IaaS providers like Rackspace or Azure?
- What about Container as a Service (CaaS) providers like Joyent, Tutum or Profitbricks?
- What about using Amazon's Container Service?

My World Needs To Change - Problem Statement

"How can we ***deploy fewer*** virtual machines while ***increasing the density and utilization*** of services per machine ***without locking*** us into a specific IaaS provider?"

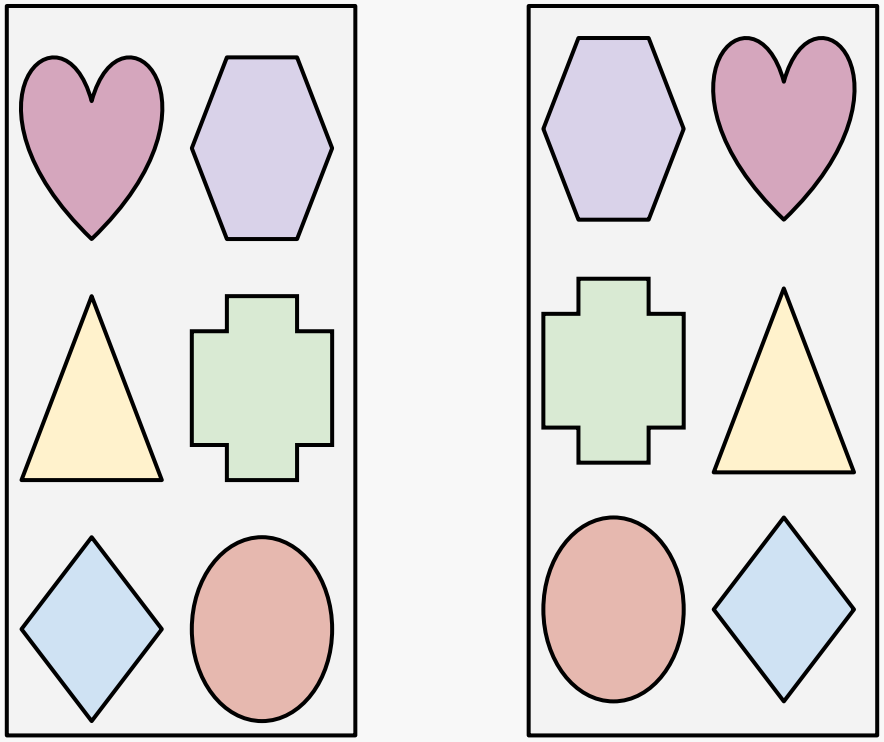
Why Docker Is Important - Before Containers

Very inefficient use of memory and CPU resources



Why Docker Is Important - After Containers

Isolated services in fewer VMs...

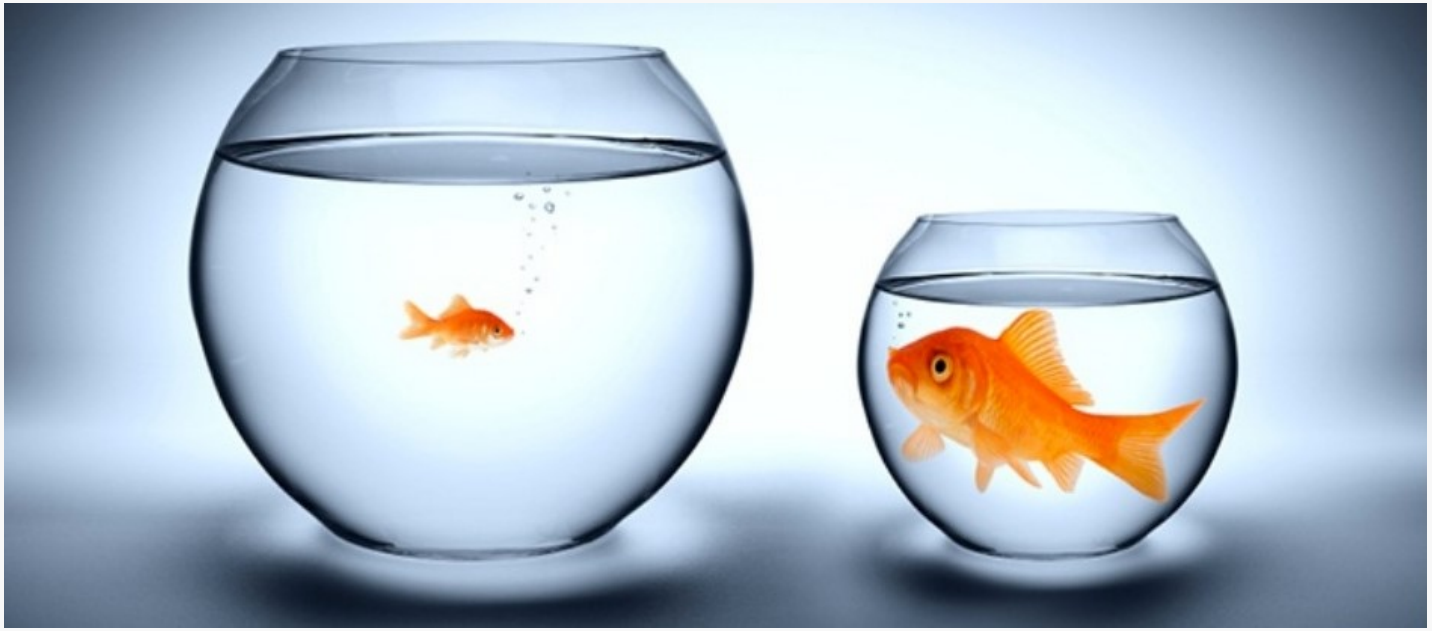


… and use VMs more efficiently.

Why Is Docker Important?

Docker container technology provides our "micro-services" platform:

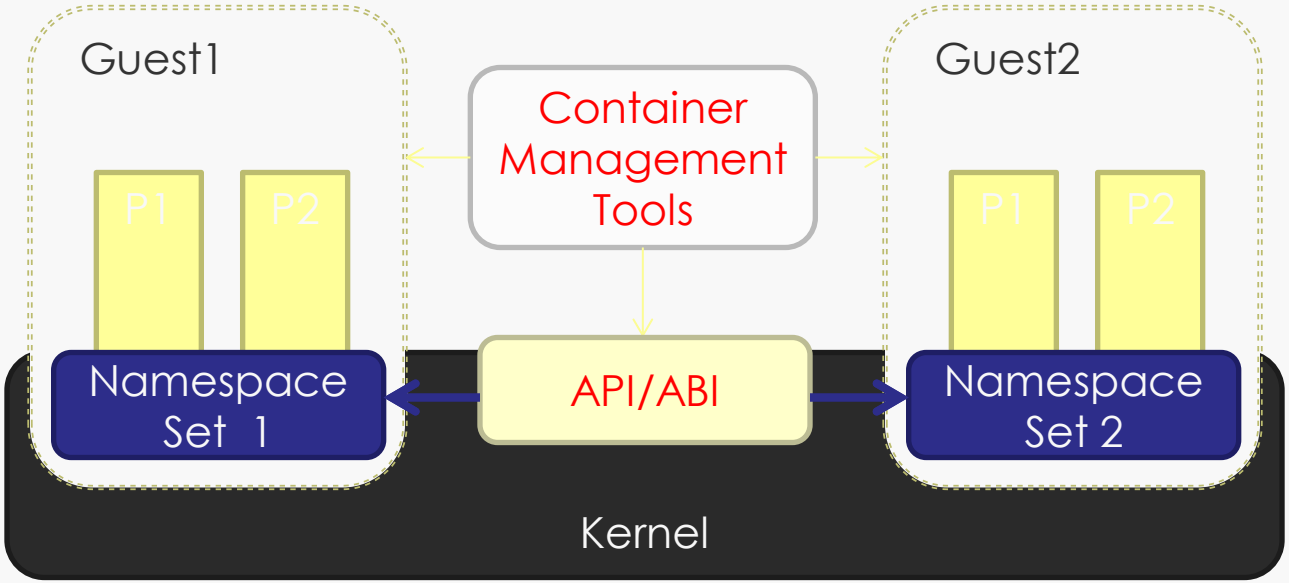
- Increased ***density*** of ***isolated*** "micro-services" per virtual machine (9:1!)
- Containerized "micro-services" are ***portable*** across machines and providers
- Containerized "micro-services" are much ***faster*** than virtual machines



End of case study

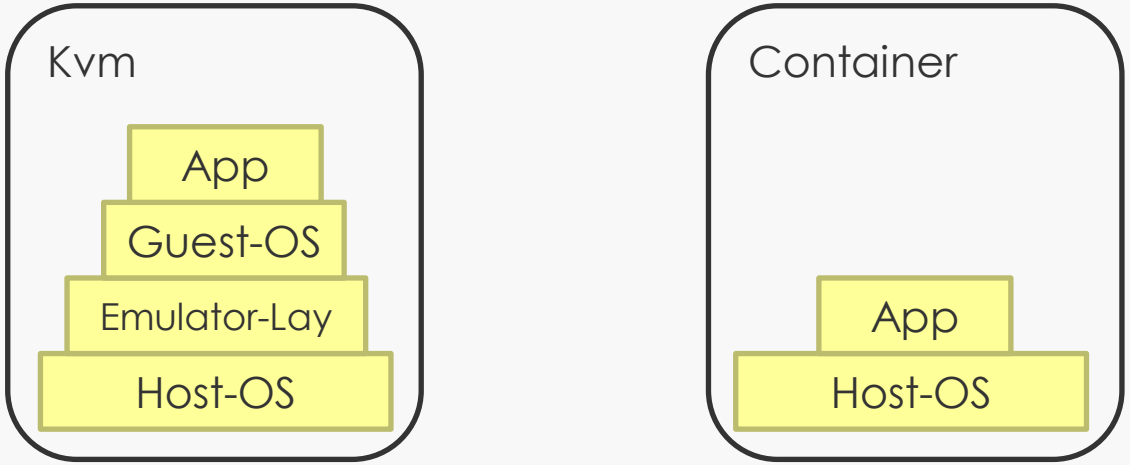
Introduction: Linux containers

- Container = Operation System Level virtualization method for Linux



Introduction: motivation

- Why do we need it?
  - Better performance



  - multi-tenant environment

LINUX NAMESPACES

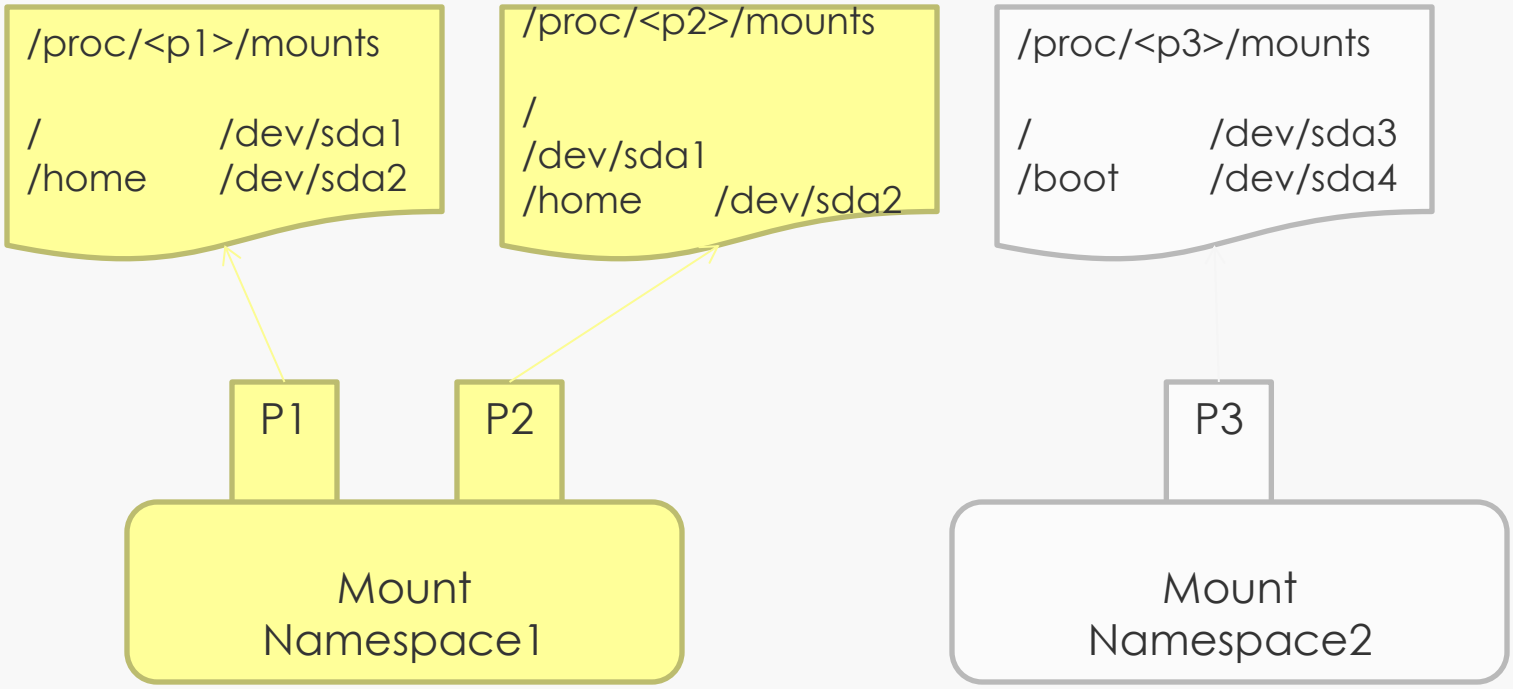
Namespaces

- Isolating system resources

- 6 namespaces in Linux Kernel
  - Mount
  - UTS
  - IPC
  - Net
  - Pid
  - User

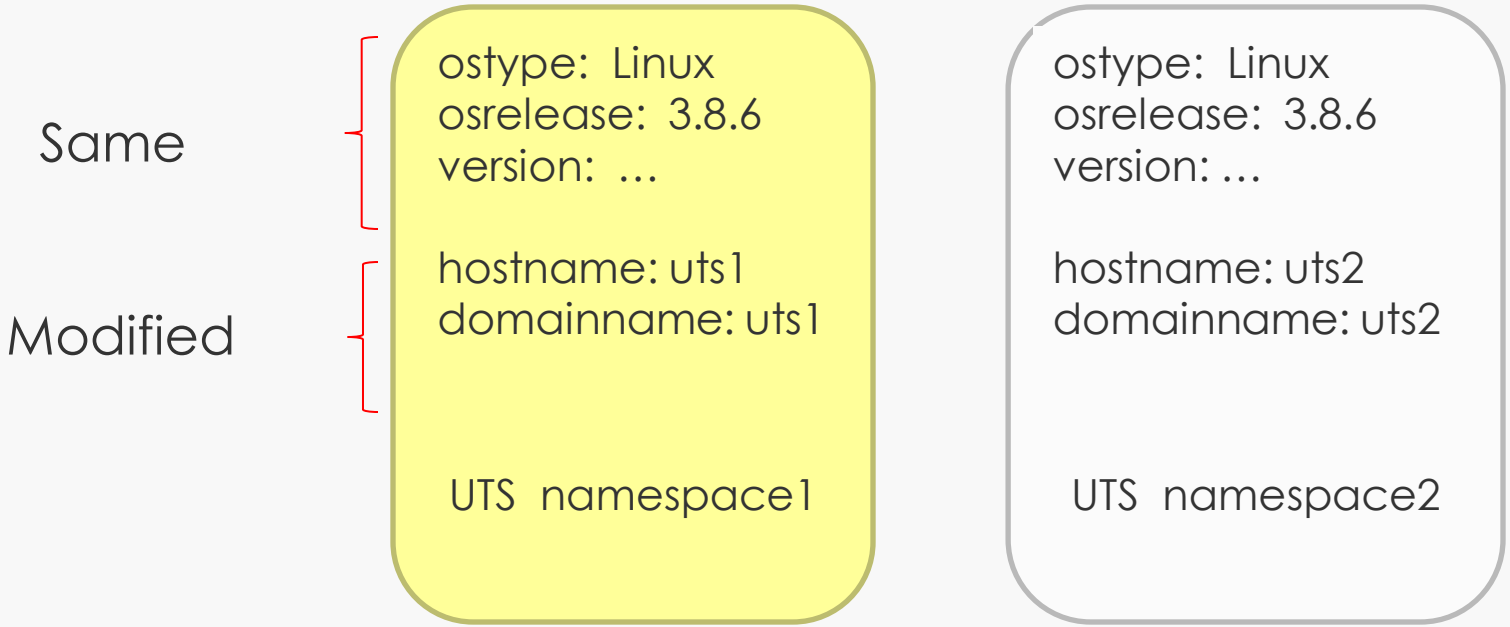
Mount Namespace

- Own file system



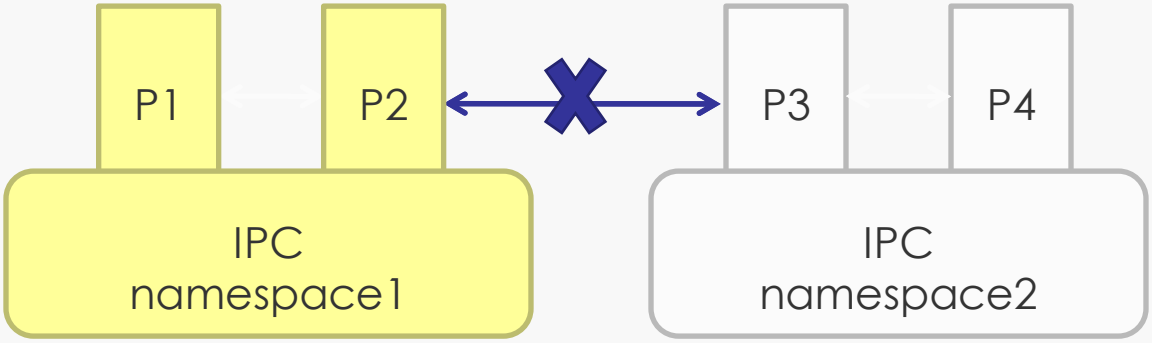
UTS Namespace

- UTS = UNIX Timesharing System
- Own uts-info



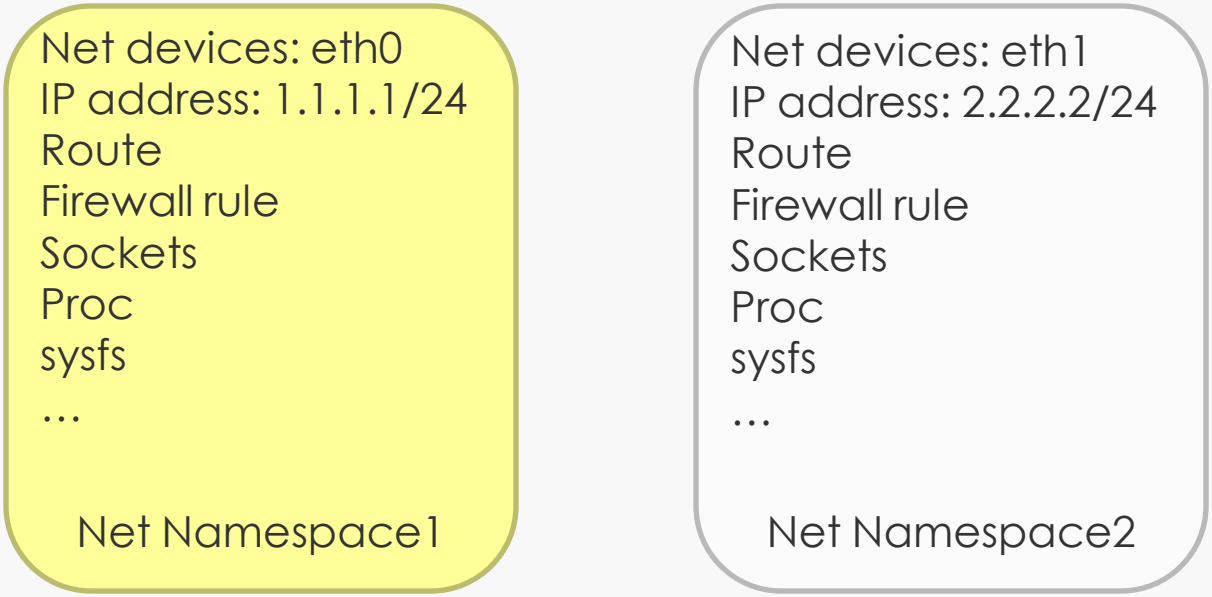
IPC Namespace

- IPC: InterProcess Communication
  - shared memory
  - Semaphore
  - message queue



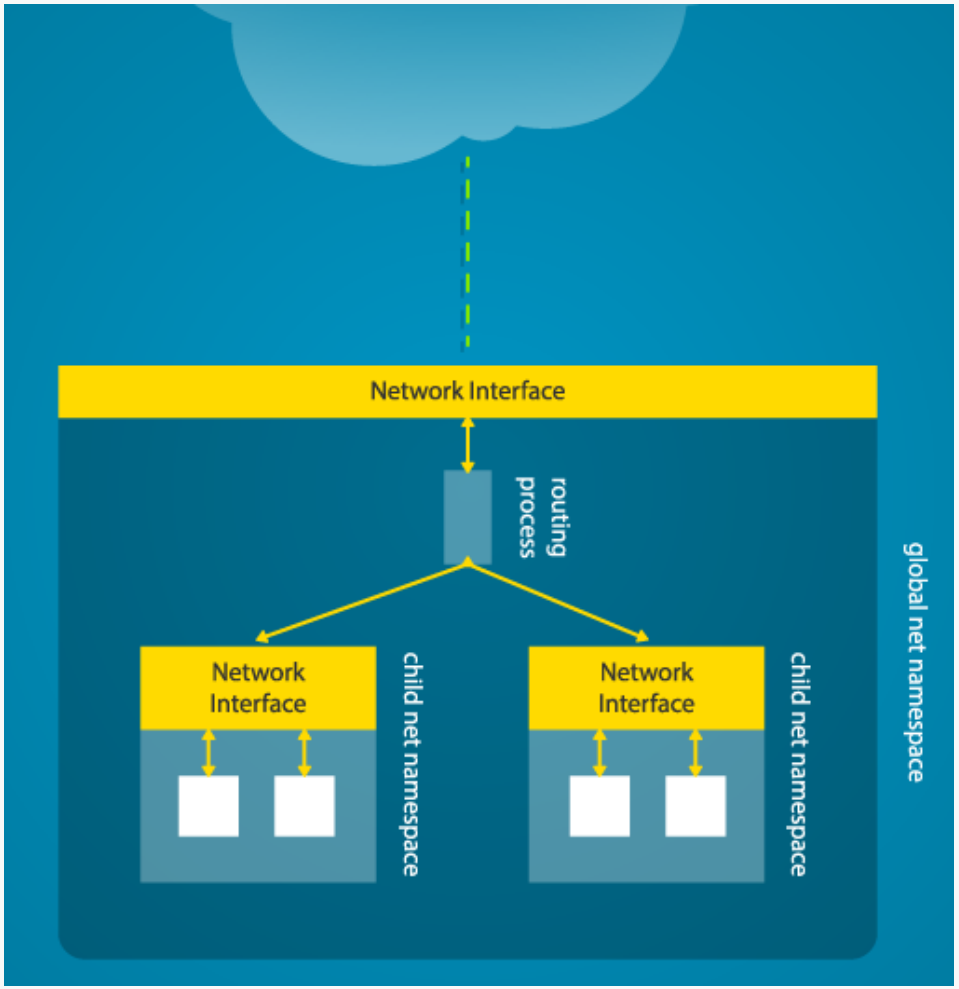
Net Namespace 1/2

- Net namespace: networking resources



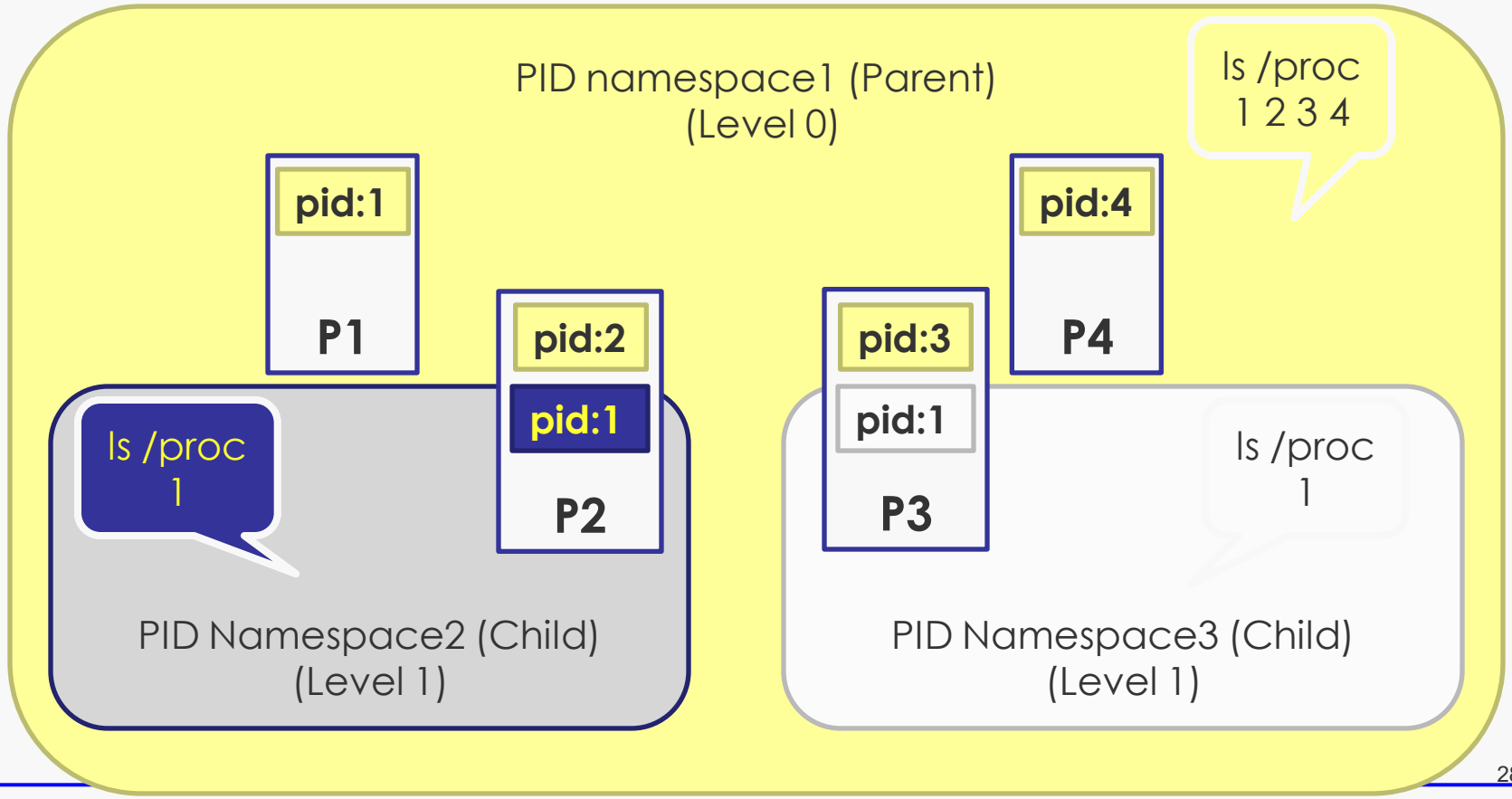
Net Namespace 2/2

- Separated by the Kernel
- In order to connect two namespaces
  - routing



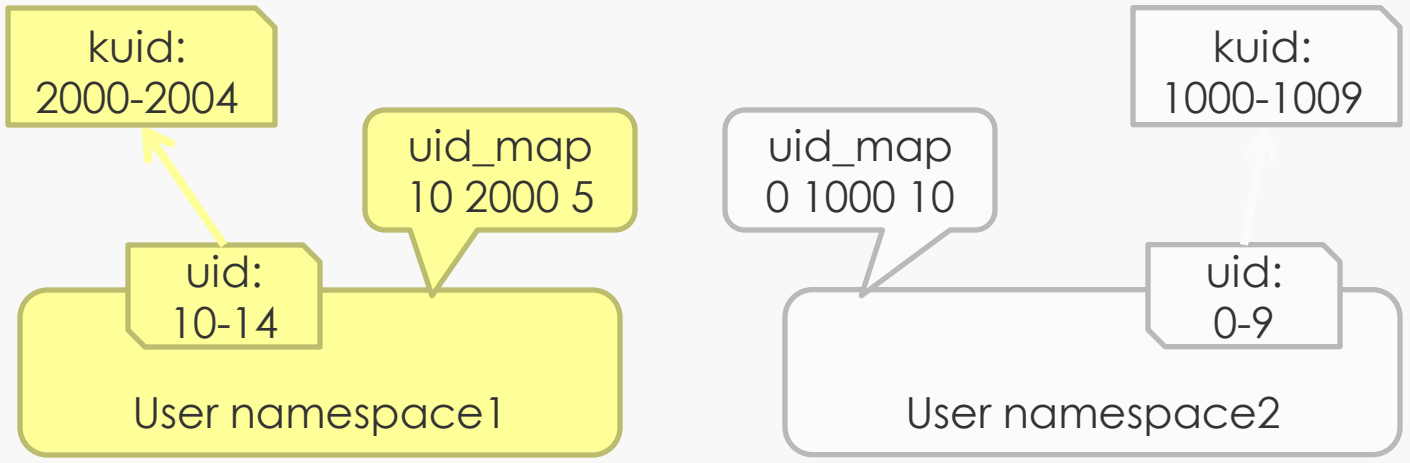
PID Namespace

- PID: Process ID
- Hierarchical system



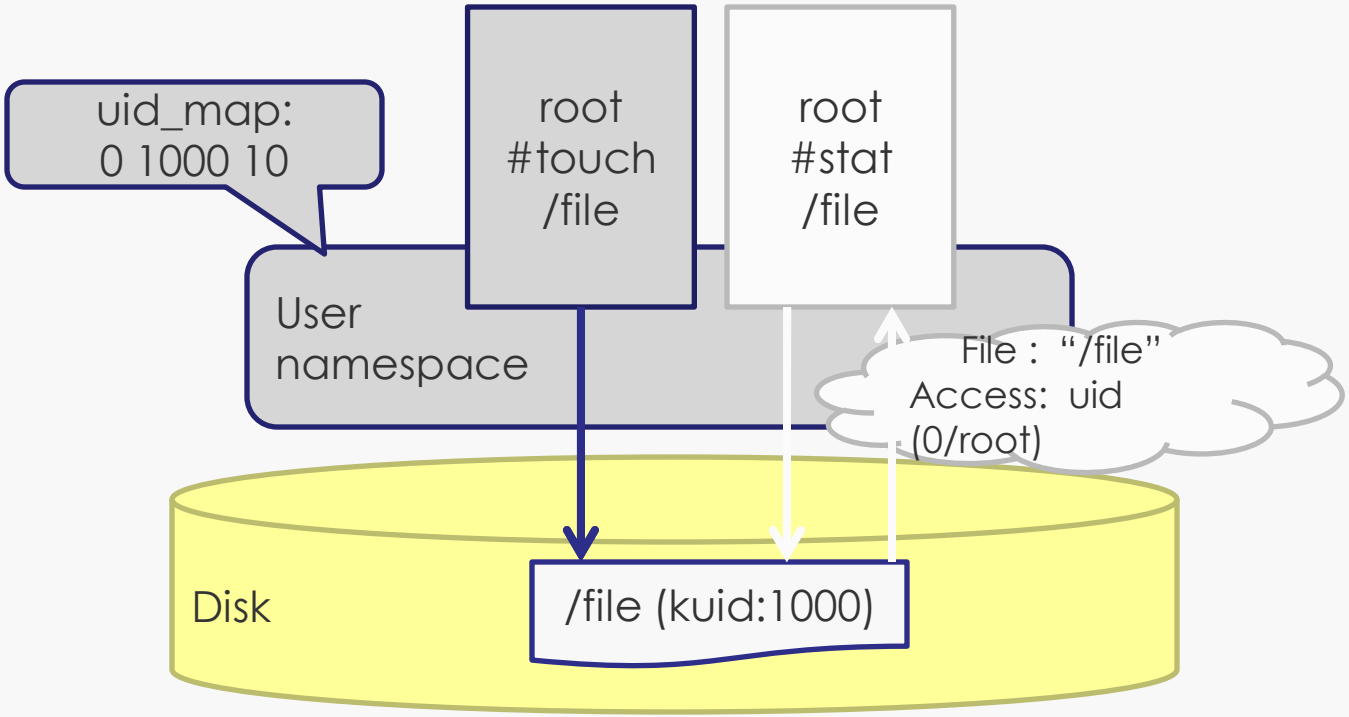
User Namespace

- User attributes linked to secure access
  - kuid/kgid: Original uid/gid, Global
  - uid/gid: user id from „user” namespace mapped to kuid/kgid
- Only the parent user NS can setup the mapping



User Namespace

- Create, stat file



CGROUPS

Linux cgroups

- Limiting the resource usage
  - Storage (mem)
  - Compute (cpu)
  - Communication (blkio)
  - Devices (dev)

LXC

System API/ABI

- Proc
  - /proc/<pid>/ns/
- System Call
  - clone
  - unshare
  - setns

Proc

- /proc/<pid>/ns/ipc: ipc namespace
- /proc/<pid>/ns/mnt: mount namespace
- /proc/<pid>/ns/net: net namespace
- /proc/<pid>/ns/pid: pid namespace
- /proc/<pid>/ns/uts: uts namespace
- /proc/<pid>/ns/user: user namespace

- If the proc file of two processes are the same, then they belong to the same namespace

System calls

- clone

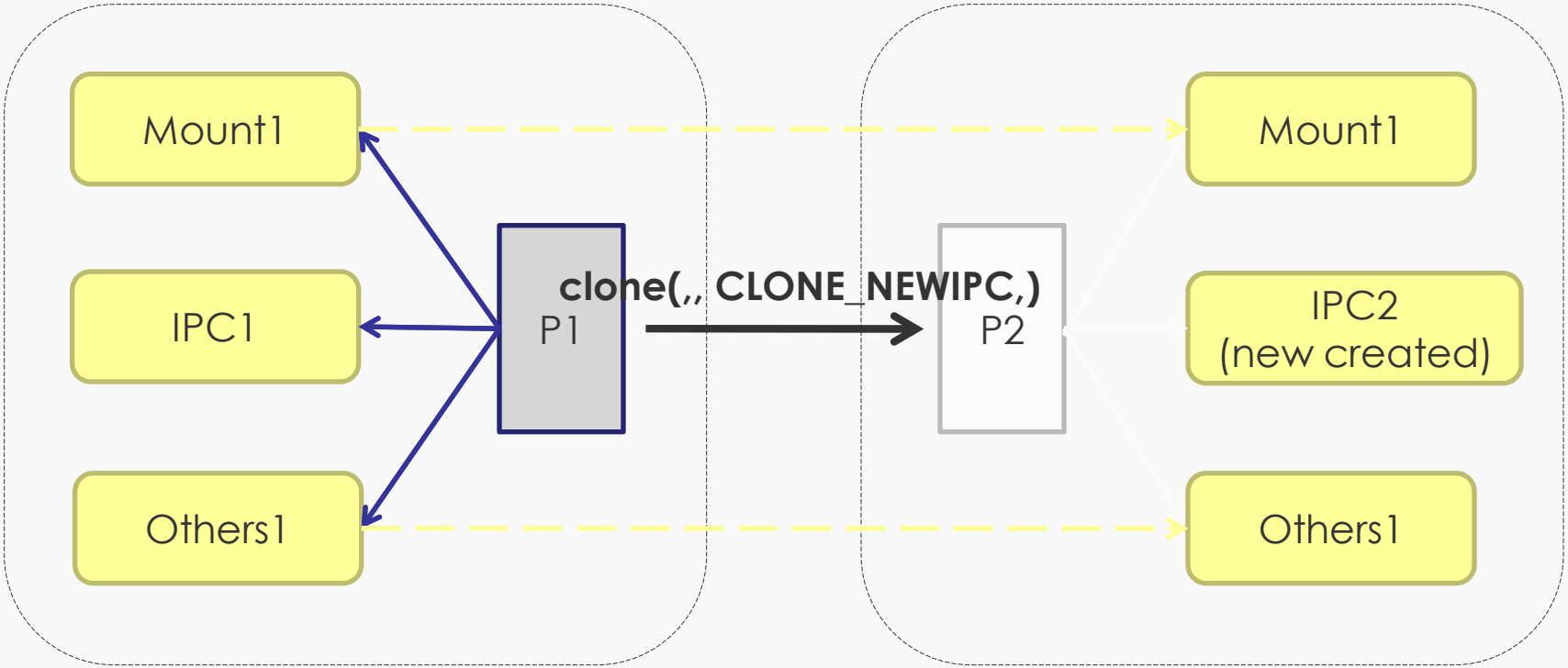
```
int clone(int (*fn)(void *), void *child_stack,
        int flags, void *arg, ...);
```

6 flag:

CLONE_NEWIPC,CLONE_NEWNET,
CLONE_NEWNS,CLONE_NEWPID,
CLONE_NEWUTS,CLONE_NEWUSER

System calls

- clone
  new process (process2) and IPC in namespace2



System calls

- unshare

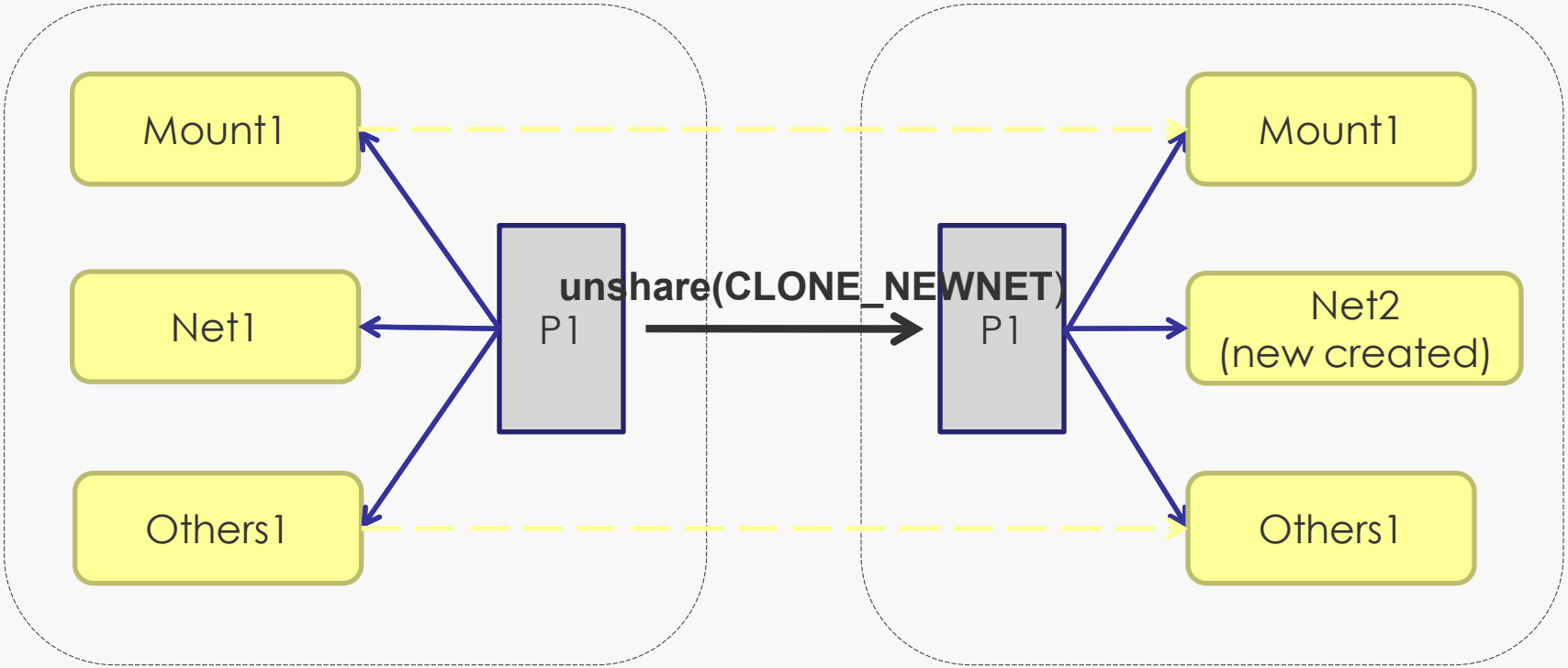
  int unshare(int flags);

  New namespace from „user space”, stepping into a new NS

System calls

- unshare

Creating net namespace2



System calls

- setns

  int setns(int fd, int nstype);

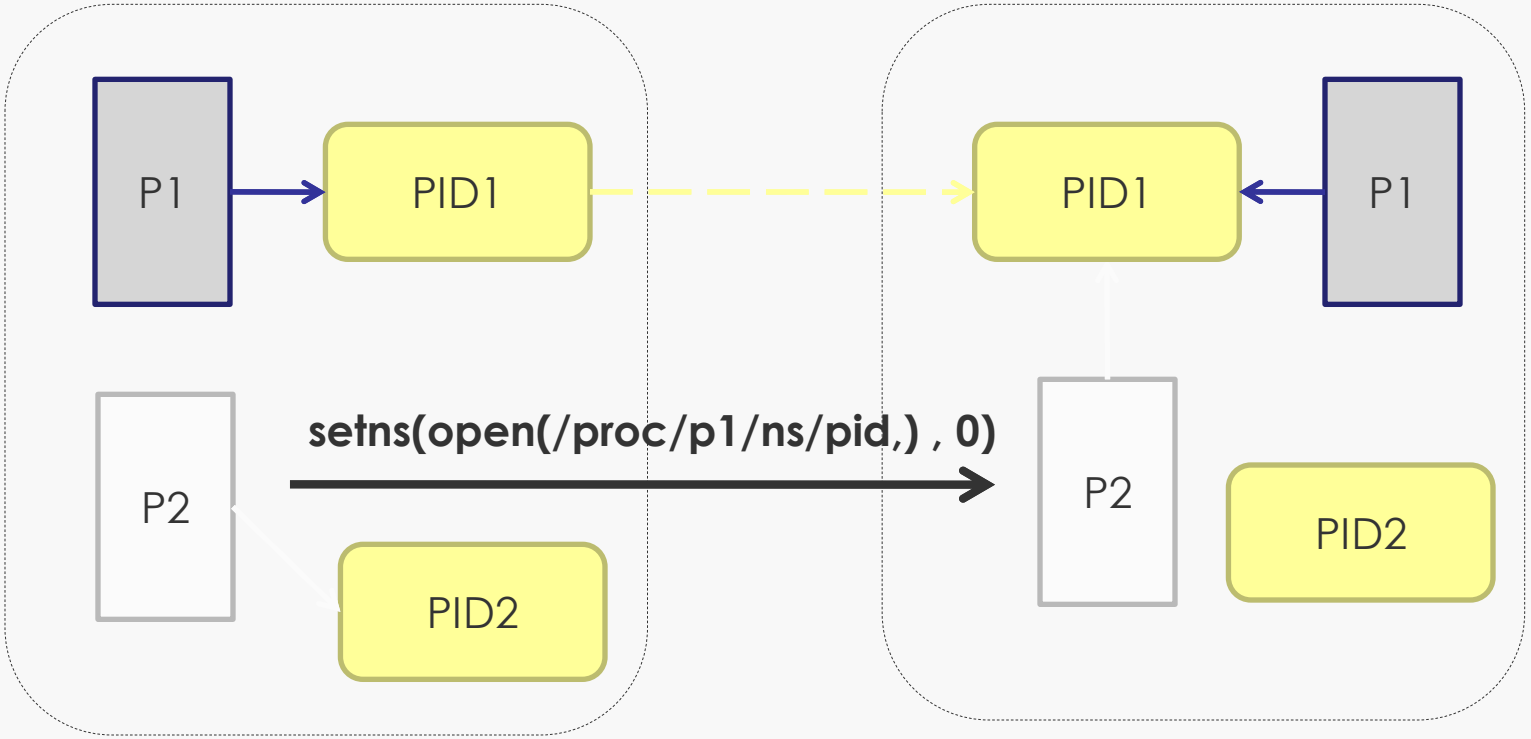
  Defines the NS the new process will belong to

  @fd: file descriptor of namespace(/proc/<pid>/ns/*)
  @nstype: type of namespace.

System calls

- setns

  Chaging PID namespace of P2



Libvirt LXC

- Libvirt LXC: userspace container management tool
  - Implemented as libvirt driver
  - Container management
  - Creating NS
  - Handling private file system within a container
  - Creating the devices of a container
  - Resources controlled through cgroup

Comparison

- Lightweigth virtualization, only one OS
  - „host share the same kernel with guest”

	Container	**KVM**
performance	Great	Normal
OS support	Linux Only	No Limit
Security	Normal	Great
Completeness	Low	Great

Open issues

- /proc/meminfo, cpuinfo…
  - Kernel space (cgroup)
  - User space (low efficiency)
- New namespace proposals under discussion
  - Audit (user namespace?)
  - Syslog (is it required?)

Open issues

- Bandwidth
  - TC Qdisc
    - On host (how to map a container to NICs)
    - On container (user can modify it)
  - Netfilter
    - How to handle ingress bandwidth?
- Disk quota
  - Uid/Gid Quota (many users)
  - Project Quota (xfs OK)

DOCKER

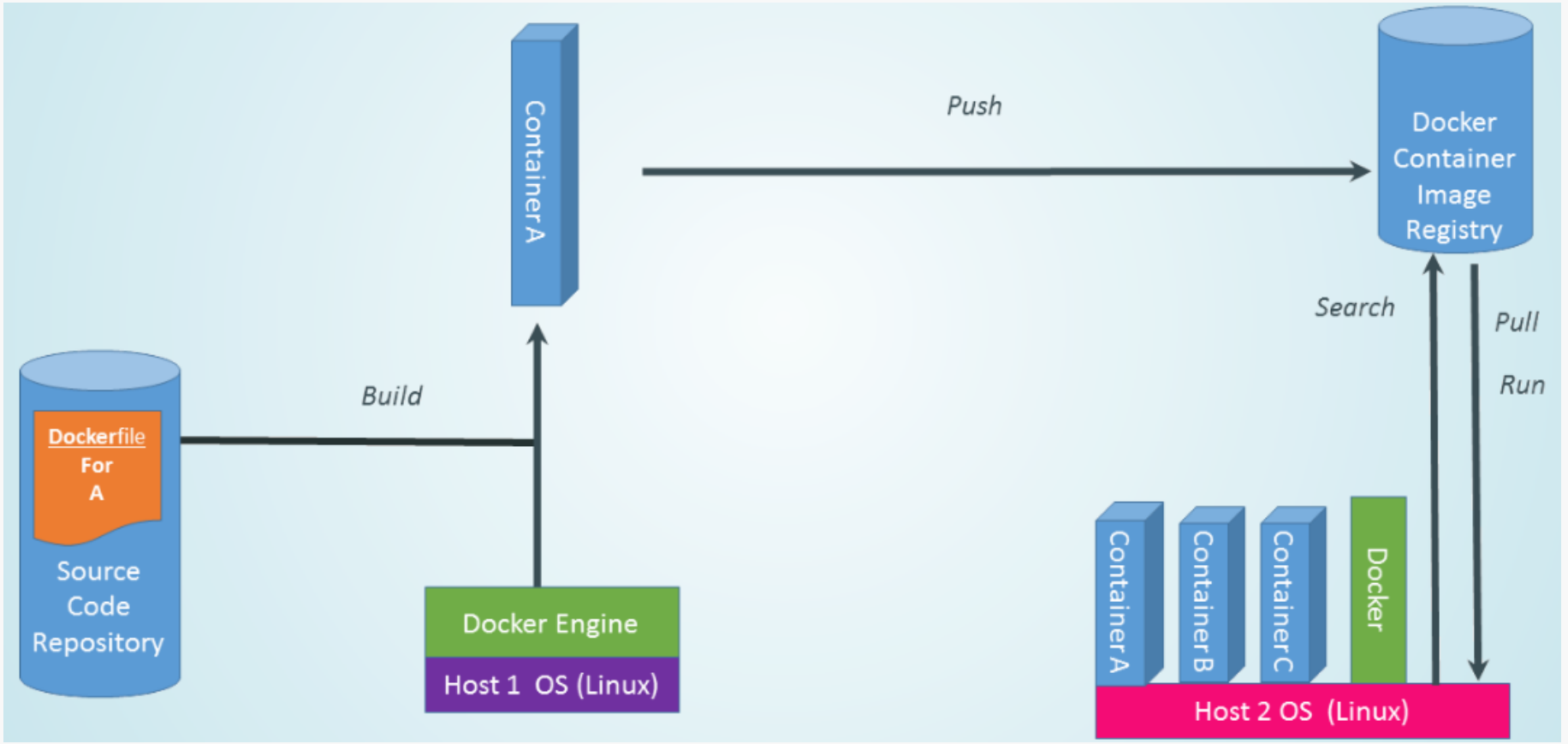
What is Docker?

- Docker = Linux container engine
- Open Source project
  - First release (early beta): 3/2013 by dotCloud
  - Later renamed to Docker Inc
- Python code, later refactored in Go
- https://www.docker.io/
- git repository: https://github.com/dotcloud/docker.git

Docker terminology

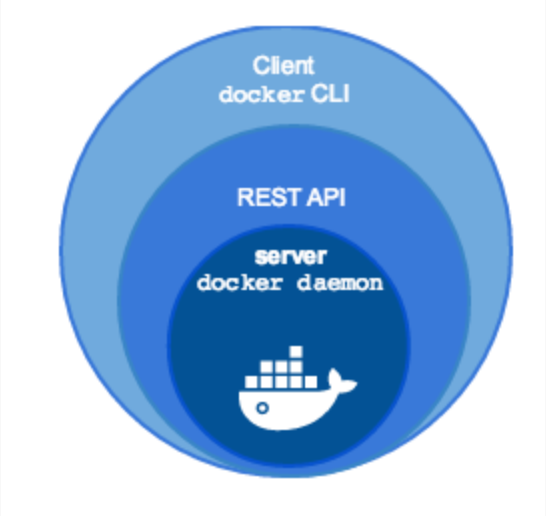
- Docker image = one file group corresponding to a VM, which contains any extension (lib, db, config, etc.) required to run the planned app
- Container = run-time Docker image instance
- Registry = image repository
  - By default is local (on-host)
  - Docker Inc. Supports a global public on-line repository (similar to github)

What is Docker?

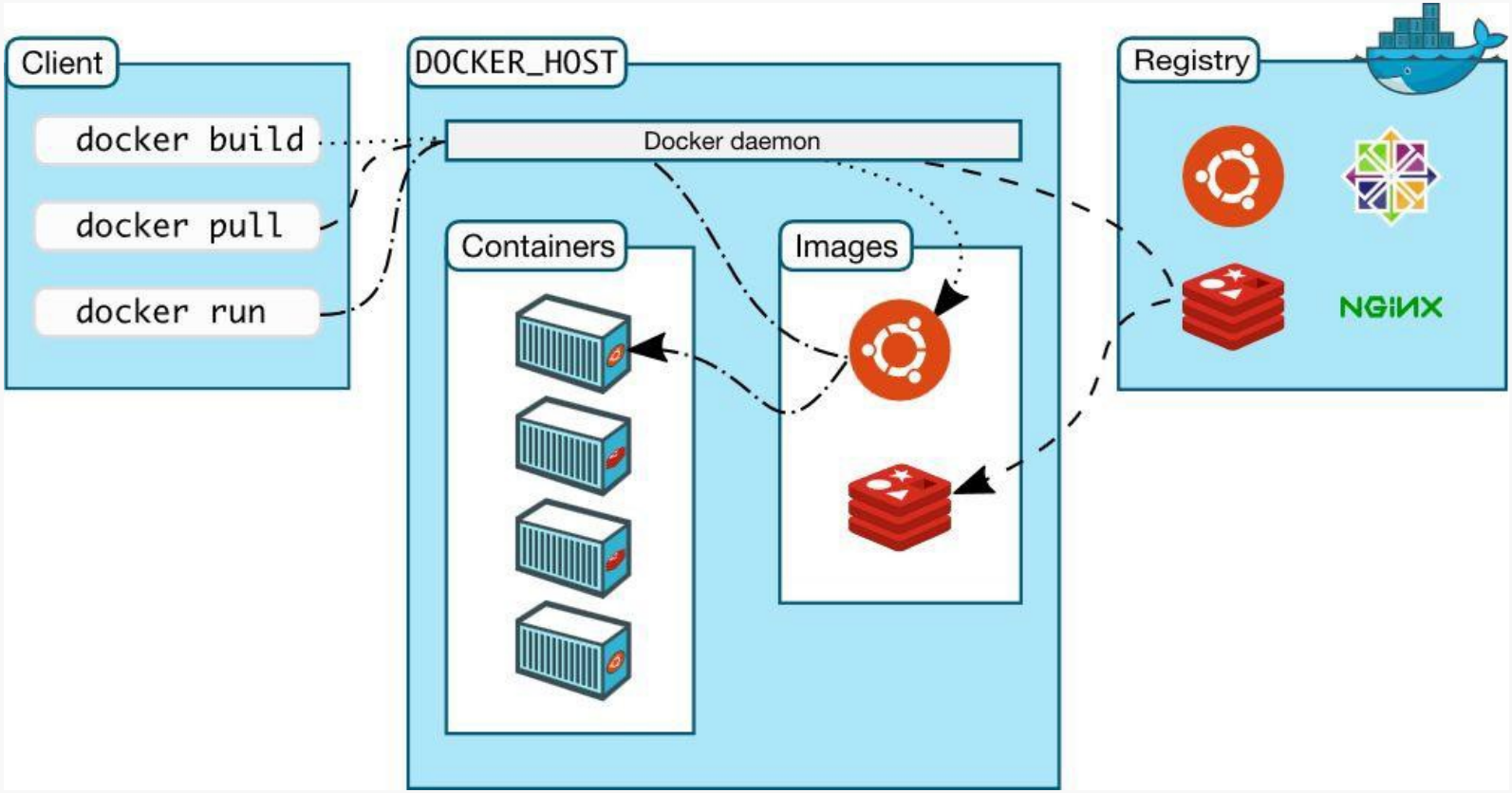


Docker Engine

- start docker service
- It executes every „docker command”
- Keeps track the locally stored docker images

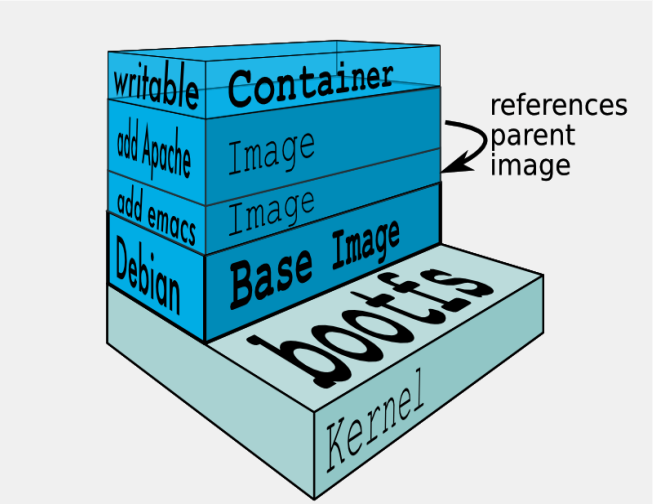


Docker system overview



Docker images

- Composed of layers
- Union file system
  - One single image from layers

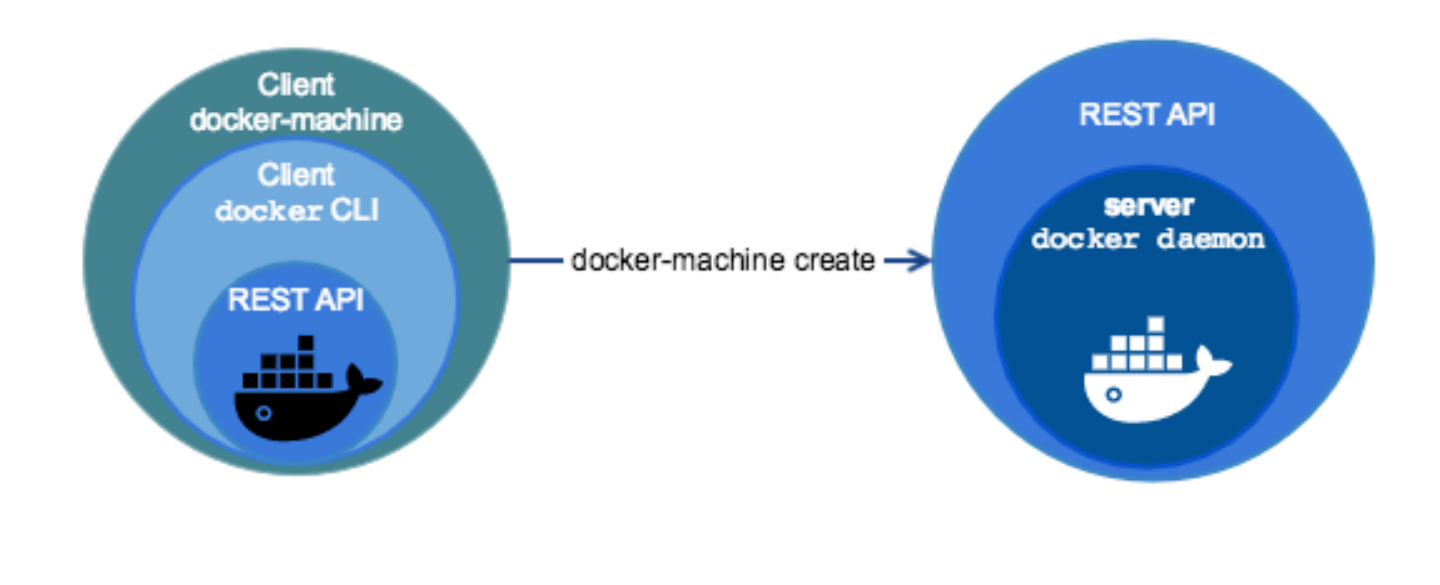


[Source: https://docs.docker.com/terms/layer/]

- Created based on a template
  - Dockerfile
  - Starting point: base image (e.g. ubuntu, fedora, etc.)
  - Own command to add new layers

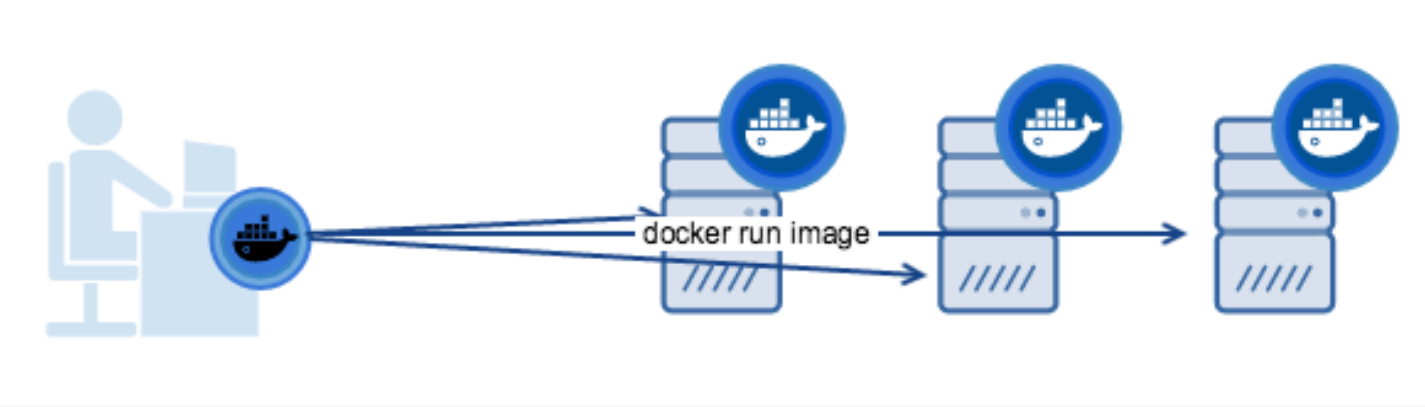
- Visulaization of different layers of an image:
  - https://imagelayers.io/

Docker Machine



» Handling containers in remote hosts

Docker Machine



- Handling containers in remote hosts
  - Own cli (`docker-machine`)

Docker Compose

- Starting multiple services (containers)
- Dockerfile -> application specific details
- docker-compose.yml
- docker-compose up

```
version: '2'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Docker workflow 1/2

- Single dev environment (local machine or container)
- All services run in containers (eg. DB)
  - And run the same way

- Testing in „real” deployment conditions
  - Build in seconds
  - Run immediately

Docker workflow 2/2

- If local build OK, then
  - Upload to registry (public/private)
  - Automatized run
  - In production, enterprise environment, too
  - Simple shift between dev and production

- In case of errors: Rollback
  - Use an earlier working version

a.) Docker images - (run/commit)

- 1) docker run ubuntu bash
- 2) apt-get install this and that
- 3) docker commit <containerid> <imagename>
- 4) docker run <imagename> bash
- 5) git clone git://.../mycode
- 6) pip install -r requirements.txt
- 7) docker commit <containerid> <imagename>
- 8) repeat steps 4-7 as necessary
- 9) docker tag <imagename> <user/image>
- 10) docker push <user/image>

a.) Pro/con

- Pro
  - Well-known technologies and steps
  - roll back/forward – as required
- Con
  - Manual process
  - Iterative steps „add on”, hard to remember
  - Complete re-build prone to errors

b.) Docker files

- RUN apt-get -y update
- RUN apt-get install -y g++
- RUN apt-get install -y erlang-dev erlang-manpages erlang-base-hipe ...
- RUN apt-get install -y libmozjs185-dev libicu-dev libtool ...
- RUN apt-get install -y make wget
- RUN wget http://.../apache-couchdb-1.3.1.tar.gz | tar -C /tmp -zxf-
- RUN cd /tmp/apache-couchdb-* && ./configure && make install
- RUN printf "[httpd]\nport = 8101\nbind_address = 0.0.0.0" >
- /usr/local/etc/couchdb/local.d/docker.ini

EXPOSE 8101

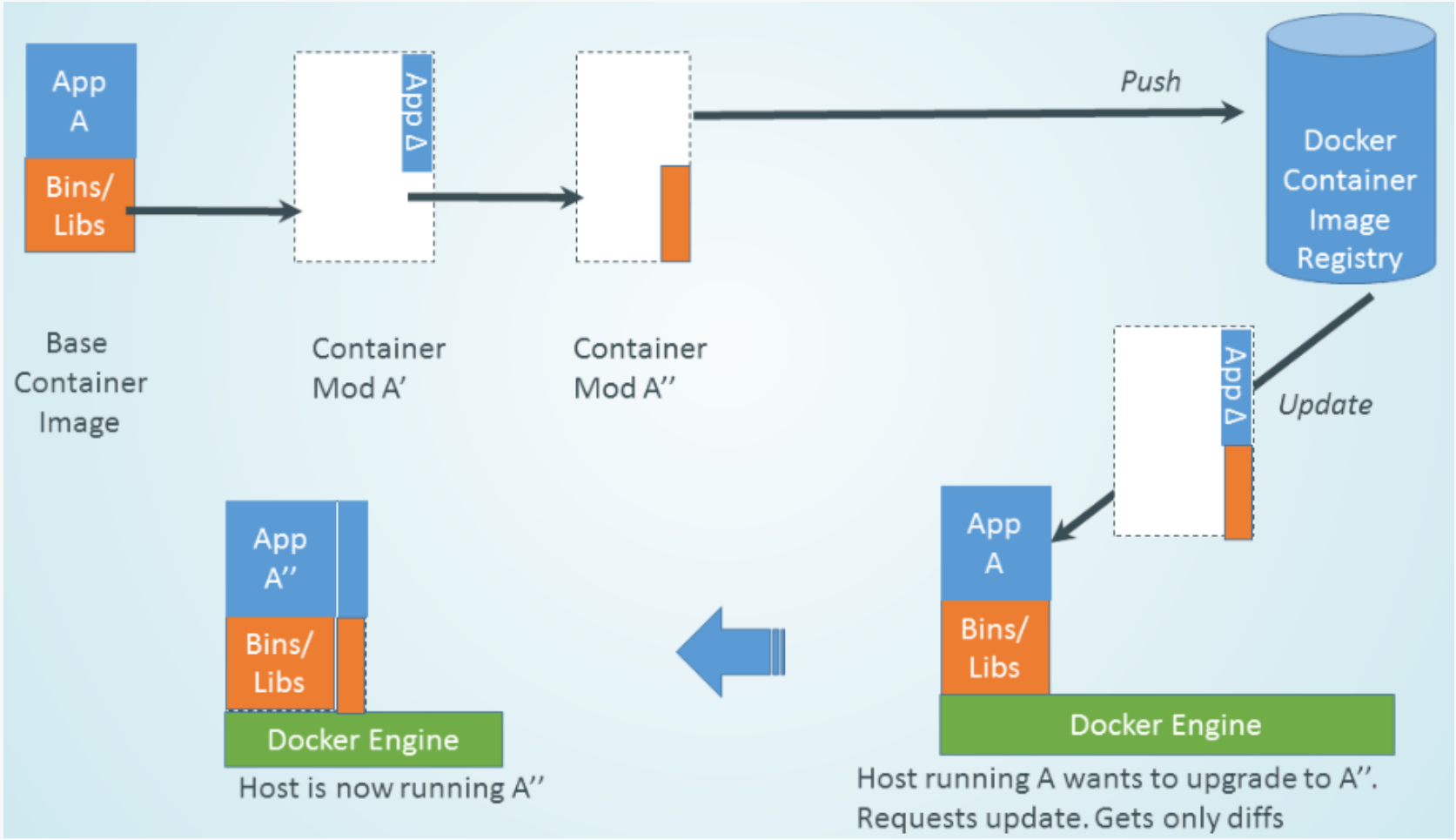
CMD ["/usr/local/bin/couchdb"]

docker build -t author_name/couchdb

b.) Advantages

- Easy to learn
- Easy re-build
  - Caching system
- build process described in a single file

Docker – why is fast?



Docker

- Multi-arch, multi-OS
- Stable control API
- Stable API plugin
- Resiliency
- Signed
- Organized in clusters, scalable

Docker vs. VM

- **Latency**: Applications with a low tolerance for latency are going to do better on physical. This something we see quite a bit in financial services (trading applications are prime example).
- **Capacity**: VMs made their bones by optimizing system load. If your containerized app doesn't consume all the capacity on a physical box, virtualization still offers a benefit here.
- **Mixed Workloads**: Physical servers will run a single instance of an operating system. So, you if you wish to mix Windows and Linux containers on the same host, you'll need to use virtualization
- **Disaster Recovery**: Again, like capacity optimizations, one of the great benefits of VMs are advanced capabilities around site recovery and high availability. While these capabilities may exist with physical hosts, the are a wider array of options with virtualization.
- **Existing Investments and Automation Frameworks** : A lot of the organizations have already built a comprehensive set of tools around things like infrastructure provisioning. Leveraging this existing investment and expertise makes a lot of sense when introducing new elements.
- **Multitenancy**: Some customers have workloads that can't share kernels. In this case VMs provide an extra layer of isolation compared to running containers on bare metal.
- **Resource Pools / Quotas**: Many virtualization solutions have a broad feature set to control how virtual machines use resources. Docker provides the concept of resource constraints, but for bare metal you're kind of on your own.
- **Automation/APIs**: Very few people in an organization typically have the ability to provision bare metal from an API. If the goal is automation you'll want an API, and that will likely rule out bare metal.
- **Licensing Costs**: Running directly on bare metal can reduce costs as you won't need to purchase hypervisor licenses. And, of course, you may not even need to pay anything for the OS that hosts your containers.

Advantages of Docker

- Easy installation
- Every app, many environments
- Repeatable build
- Great hype, strong community, fast bugfixes
- New virtualization processes
- **Con**
- Docker container type
  - Host OS dependent
- „Orchestration”
- Networking

Docker cons

- Docker container type
  - Host OS dependent
- „Orchestration”
- Networking

- But: continous upgrades, developments
  - E.g., Docker on Windows, Docker Swarm Mode
  - Favored by the hype and strong community support

Security?

- Docker over REST API / HTTP?
  - Authentication!
- Docker daemon runs with root priviliges
  - Containers are then OK (in user space)
  - Can they „reach back”?
- docker-1.3 --cap-add, --cap-drop
  - man capabilities
  - „overview of Linux capabilities”
  - „Starting with kernel 2.2”
  - „per-thread attribute”
- More developments in the make
  - Docker daemon

Sources

- Docker story in a nutshell:

  http://www.infoworld.com/article/3025870/paas/the-sun-sets-on-original-docker-paas.html

- Docker overview:

  http://www.linuxjournal.com/content/docker-lightweight-linux-containers-consistent-development-and-deployment

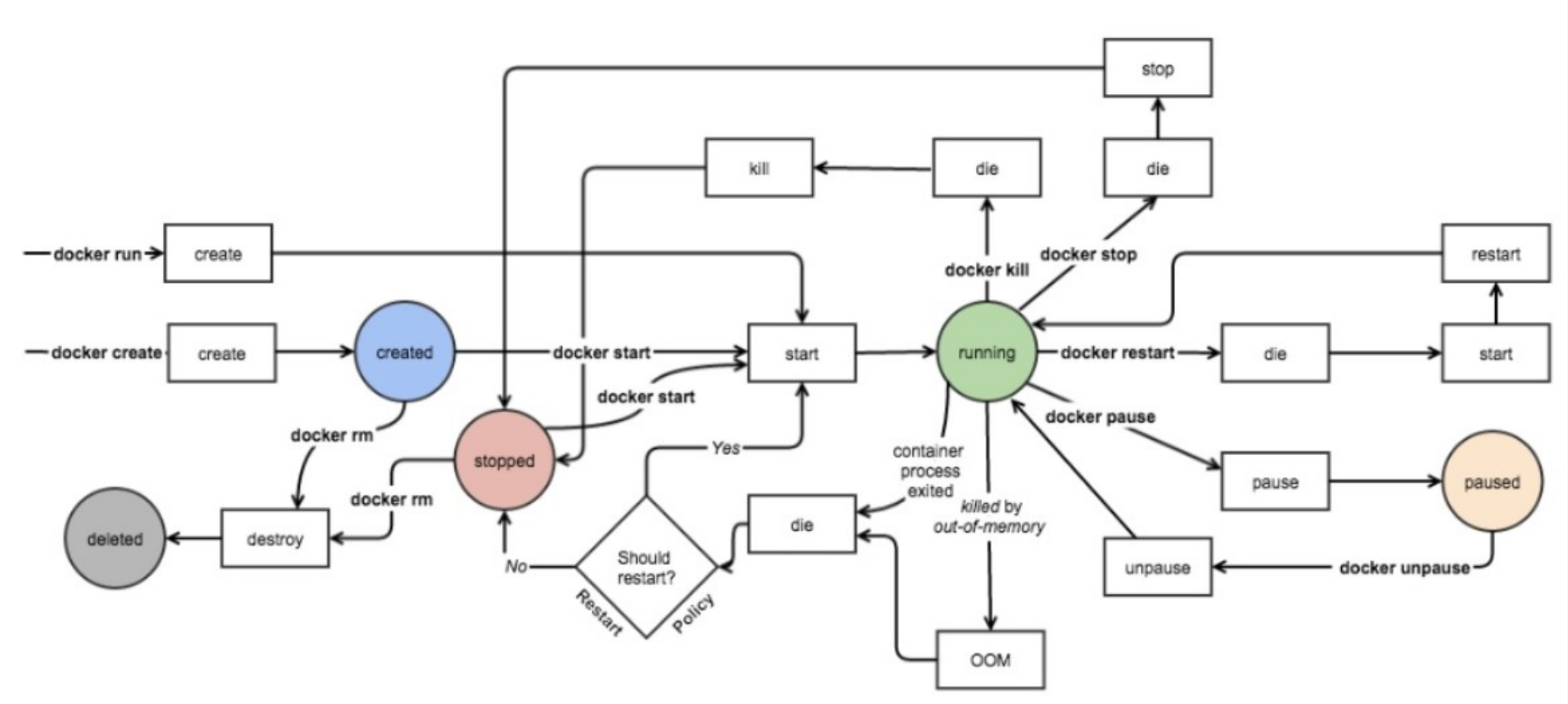
- „The Docker Book”

  http://www.dockerbook.com/#toc

- Docker Meetup @Budapest

  http://www.ustream.tv/recorded/60277876

Docker state machine



» die, kill ≠ destroy