NVM: Containers

D. Carver J. Sopena

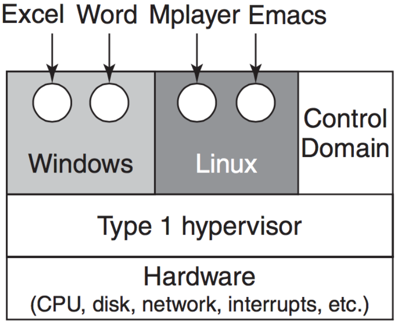
Overview

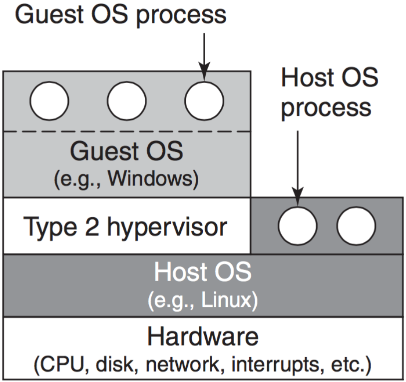
1. Virtualization
   - From Virtual Machines to Containers
2. Linux Containers
   - Linux Cgroups
   - Linux namespaces
3. Memory Cgroup in details
   - Memory Counters
   - Memory Isolation
   - Memory Consolidation
4. Docker
   - Images layers
   - Orchestration

Hypervisor (Type 1 and Type 2)

Hypervisor

The layer of software in charge of running multiple Virtual Machines





We need Virtual Machines

- **Machine Emulation** to run any OS
- **Resource Isolation** to multiplex a big machine into multiple VMs
- **Security Isolation** to avoid propagation of attacks or failures

But Virtual Machines are too heavy

Cost of **virtualization**: The Hypervisor has to make the VMs believe that they are alone and in charge of the hardware.

Cost of the **blackbox**: It is hard for the Hypervisor to reallocate resources because VMs do not cooperate.

Cost of OS **duplication**: Running multiple OS when VMs tend to deliver a single Microservice.

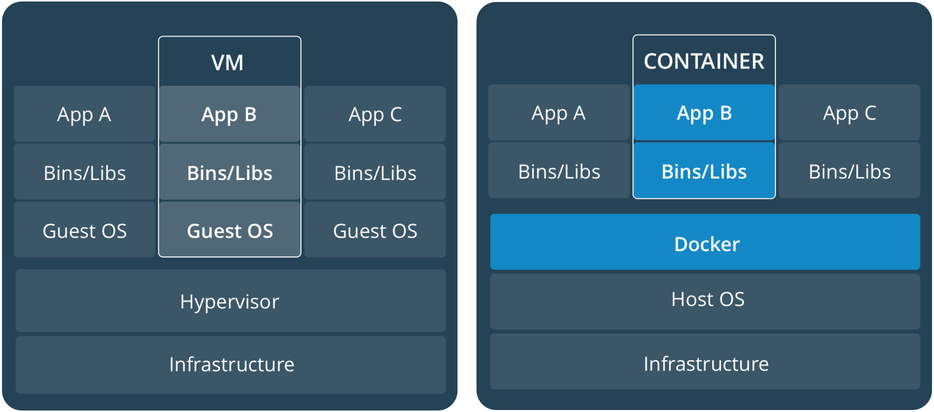
What if???

- **Machine Emulation** to run any OS
- **Resource Isolation** to multiplex a big machine into multiple VMs
- **Security Isolation** to avoid propagation of attacks or failures

What if we only needed Isolation?

- ~~**Machine Emulation** to run any OS~~
- **Resource Isolation** to multiplex a big machine into multiple VMs
- **Security Isolation** to avoid propagation of attacks or failures

Operating-system-level Virtualization



- Single OS that provides Isolation Features
- Container engine (Docker)

Operating-system-level Virtualization

Containers are Lightweight because they remove:

- Cost of virtualization (Containers do not manage the hardware)
- Cost of running multiple OS (there is only one OS)
- Cost of the blackbox (the single OS decides and sees all)

Containers

1982	chroot	UNIX (filesystem isolation only)
2000	FreeBSD jail	First Containers
2005	OpenVZ	Patched Linux Kernel
2008	LXC (LinuX Container)	Mainline Linux Kernel
2013	Docker	runc (OCI, Portability accross OS)

Linux Containers

LXC and Docker rely on the Linux kernel isolation features:

- Linux Cgroups (Resource/Performances Isolation)
- Linux Namespaces (Security Isolation)

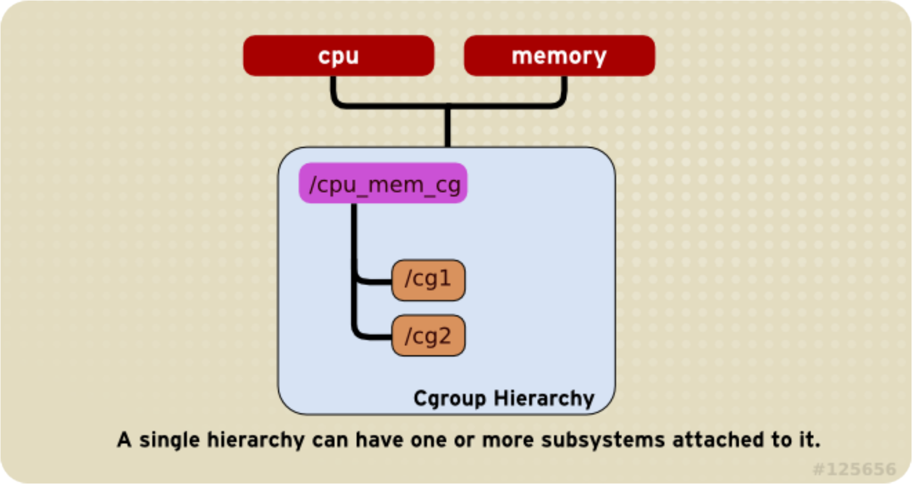
Linux Cgroups

Control Groups

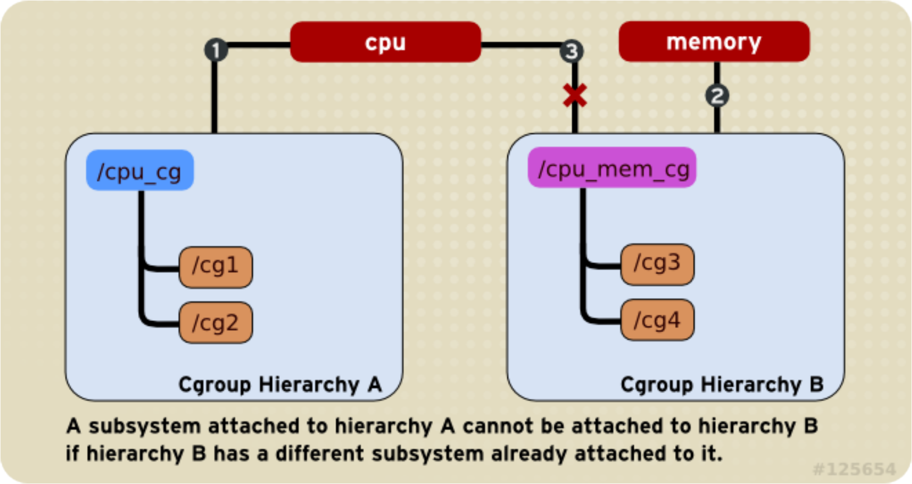
Cgroups is a Linux kernel feature that **limits**, **accounts** for, and **isolates** the resource usage of a collection of processes.

- cpu
- cpuacct
- cpuset
- memory
- blkio
- freezer
- devices
- pids
- hugetlb
- net_prio
- net_cls
- perf_event

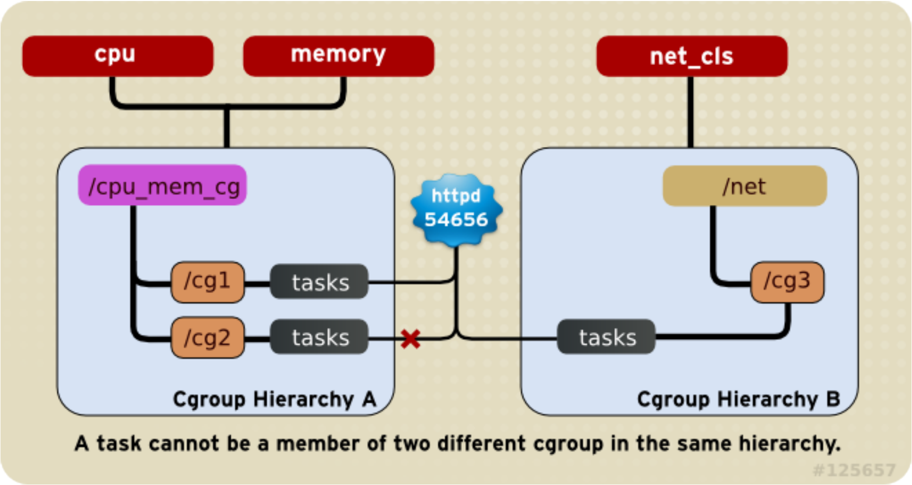
Cgroup-v1 hierarchy



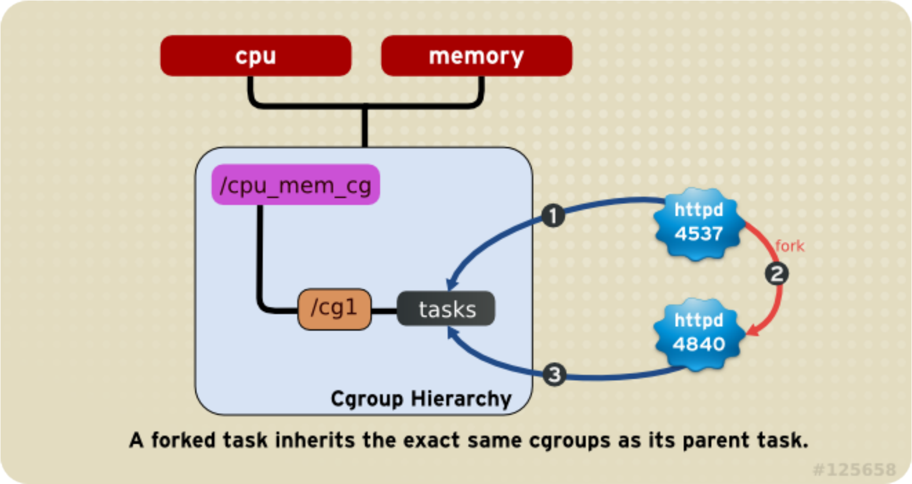
Cgroup-v1 hierarchy



Cgroup-v1 hierarchy



Cgroup-v1 hierarchy



Cgroup-v1 hierarchy

```bash
#!/usr/bin/env bash

for R in cpu memory blkio ...
do
        CGPATH="/sys/fs/cgroup/${R}/${CG}"

        # Create Cgroup
        mkdir "${CGPATH}"

        # Add self to Cgroup
        echo $$ > "${CGPATH}/tasks"

        # Configure isolation of ${R}
        # Examples in next slides....
done

exec ${proc_to_isolate}
```

CPU Cgroup: Shares

```bash
#!/usr/bin/env bash

# cpu:
# Cgroups can be guaranteed a minimum number
# of "CPU shares" when a system is busy.
#
# This does not limit a cgroup's CPU usage
# if the CPUs are not busy.

JOBS="/sys/fs/cgroup/cpu/jobs/"

echo 1024 > "${JOBS}/interactive/cpu.shares"
echo    1 > "${JOBS}/background/cpu.shares"
```

CPU Cgroup: CFS Quota

```bash
#!/usr/bin/env bash

CG="/sys/fs/cgroup/cpu/throttled_tasks"

echo 1000000 > "${CG}/cpu.cfs_period_us"
echo    1000 > "${CG}/cpu.cfs_quota_us"

# Example: Throttling infinite tasks
echo $$ > "${CG}/tasks"
while true; do metric_collection; done
```

Linux Namespaces

Linux Namespaces

Namespaces is a Linux kernel feature that create a private local view of system resources. Resources outside the Namespace are invisible. Resources inside children's Namespace are visible but do not have the same name.

Namespace	Isolates
Cgroup	Cgroup root directory
IPC	System V IPC, POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points
PID	Process IDs
User	User and group IDs
UTS	Hostname and NIS domain name

Linux Security

Linux Security features can be applied to container.

- *AppArmor* define for each applications, what can accessed and with what privileges.
- *seccomp* restrains system calls to exit(), sigreturn(), read() and write() only.
- *Linux Capabilities* avoids all root privileges when only a subset is required.
- *ulimit* limits maximum number of opened files, processes...

Memory Cgroup in details

How does the memory cgroup

1. accounts,
2. limits,
3. and isolates:

process memory, kernel memory, and swap?

Event Counter

Events are monotonically increasing

```
enum mem_cgroup_events_index {
        MEM_CGROUP_EVENTS_PGPGIN,
        MEM_CGROUP_EVENTS_PGPGOUT,
        MEM_CGROUP_EVENTS_PGFAULT,
        MEM_CGROUP_EVENTS_PGMAJFAULT,
        MEM_CGROUP_EVENTS_NSTATS,
};

static const char * const mem_cgroup_events_names[] = {
        "pgpgin",
        "pgpgout",
        "pgfault",
        "pgmajfault",
};
```

Stat Counter

Stats can increase or decrease but are usually non-negative

```
enum mem_cgroup_stat_index {
        MEM_CGROUP_STAT_CACHE,
        MEM_CGROUP_STAT_RSS,
        MEM_CGROUP_STAT_RSS_HUGE,
        MEM_CGROUP_STAT_FILE_MAPPED,
        MEM_CGROUP_STAT_DIRTY,
        MEM_CGROUP_STAT_WRITEBACK,
        MEM_CGROUP_STAT_SWAP,
        MEM_CGROUP_STAT_NSTATS,
};

static const char * const mem_cgroup_stat_names[] = {
        "cache",
        "rss",
        "rss_huge",
        "mapped_file",
        "dirty",
        "writeback",
        "swap",
};
```

Defined per CPU

```
struct mem_cgroup_stat_cpu {
        long count[MEMCG_NR_STAT];
        unsigned long events[MEMCG_NR_EVENTS];
};

struct mem_cgroup {
        struct mem_cgroup_stat_cpu __percpu *stat;
};
```

Per CPU usage

```
/* local update */
__this_cpu_add(memcg->stat->count[STAT_IDX], value);

/* local read */
value = __this_cpu_read(memcg->stat->event[EVENT_IDX]);

/* aggregate values */
unsigned mem_cgroup_read_events(struct mem_cgroup *memcg,
                                enum mem_cgroup_events_index idx)
{
  unsigned  val = 0;
  int cpu;

  for_each_possible_cpu(cpu)
    val += per_cpu(memcg->stat->events[idx], cpu);
  return val;
}
```

Page Counter: Hierarchical memory counter

```
struct page_counter {
        atomic_long_t count;
        unsigned long limit;
        struct page_counter *parent;
};

struct mem_cgroup {
        struct page_counter memory;
        struct page_counter swap;
};

void page_counter_charge(struct page_counter *counter,
                         unsigned long nr_pages);

void page_counter_uncharge(struct page_counter *counter,
                           unsigned long nr_pages);

bool page_counter_try_charge(struct page_counter *counter,
                             unsigned long nr_pages,
                             struct page_counter **fail);
```

Memory Isolation

Quantitative Isolation

With a good quantitative isolation, cgroups should not have more pages than their limit.

Qualitative Isolation

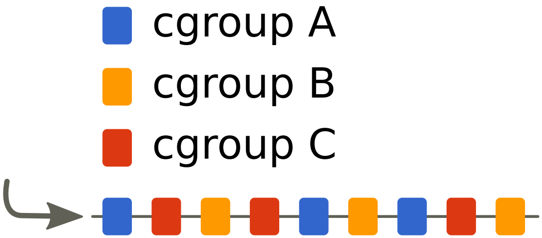
With a good qualitative isolation, cgroups should access their data in memory as if their were alone on the machine.

Quantitative Memory Isolation

```
int try_charge(struct mem_cgroup *cg, int n) {
        struct page_counter ctr;
        struct mem_cgroup *over_limit;

retry:
        if (page_counter_try_charge(&cg->memory, n, &ctr))
                return 0;

        over_limit = mem_cgroup_from_counter(ctr, memory);

        if (try_to_free_mem_cgroup_pages(over_limit, n))
                goto retry;

        mem_cgroup_oom(over_limit);

        return -ENOMEM;
}
```

Cgroups never exceed their limit because whenever their demand a new page, `try_charge` is called.

Qualitative Memory Isolation

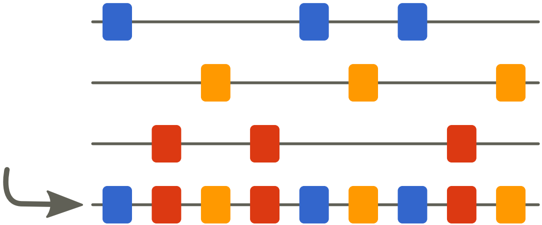


Linux keeps track of the utility of pages with a set of lists (`lru_list`).

When memory is running out, the least recently used pages are reclaimed.

With a single list, the pages of other crgoups have to be filtered out.

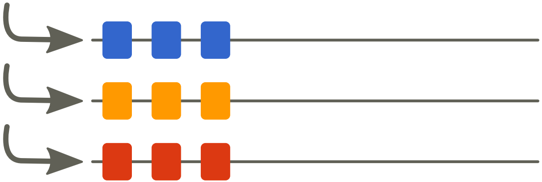
Qualitative Memory Isolation



With additional local lists, the pages of one crgoups can be reclaimed.

But there are still lock interferences between cgroups.

Qualitative Memory Isolation

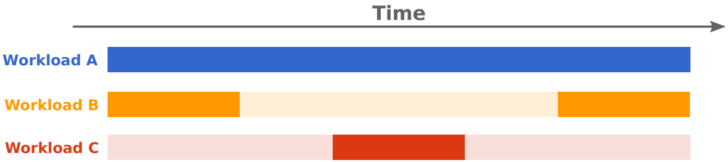


With local lists only, the isolation is very good.

But global recency is lost.

Memory Consolidation

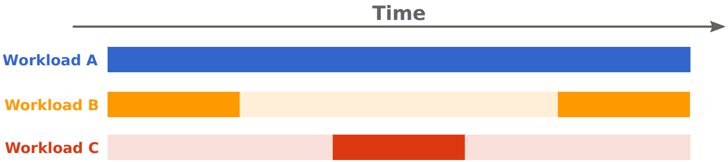
Activity model



Workloads are **active** or **inactive** (need **all** or **none** of their memory):

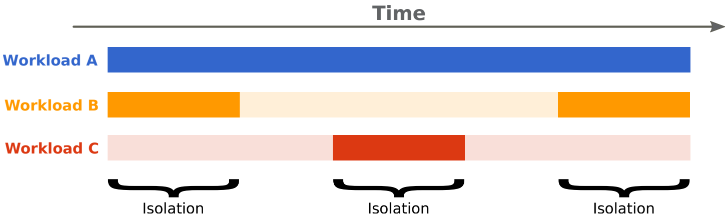
- A is always active
- B and C are **never** active at the same time

Resource multiplexing opportunities



A, B and C can be hosted on same machine but:

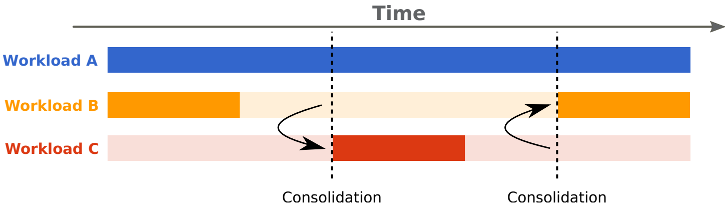
Resource multiplexing opportunities



A, B and C can be hosted on same machine but:

- they **should not interfere** with each other during the steady phases

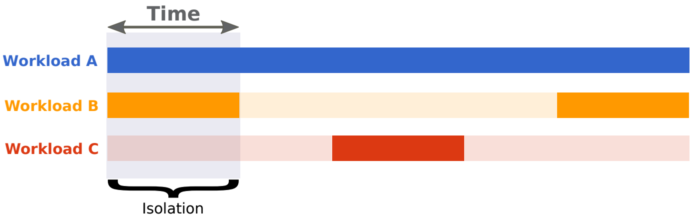
Resource multiplexing opportunities



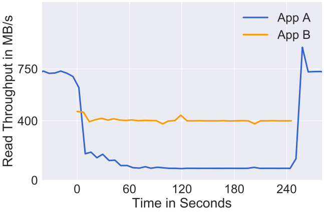
A, B and C can be hosted on same machine but:

- they **should not interfere** with each other during the steady phases
- unused memory **should be transfered** to avoid wastage

We need isolation between active applications

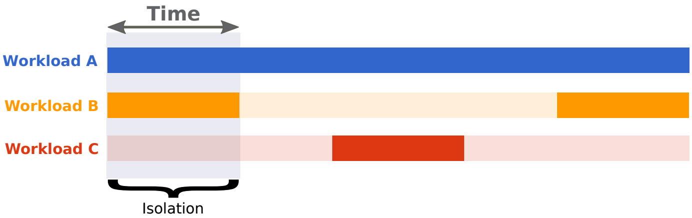


Without Linux Containers

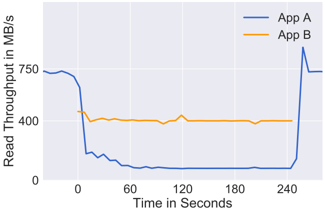


A is disturbed when **B** runs

We need isolation between active applications

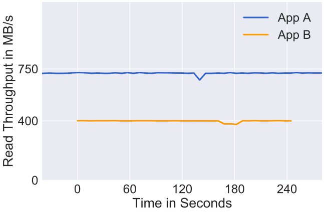


Without Linux Containers



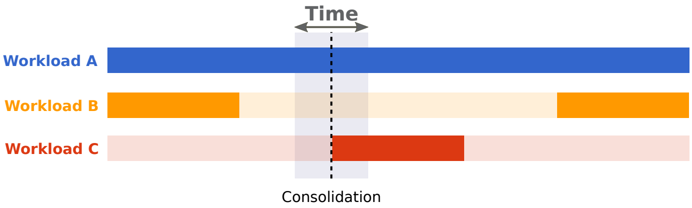
A is disturbed when B runs

With Linux Containers

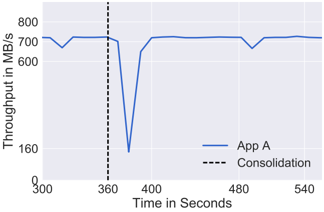


A is **not** disturbed

But performance losses occurs during consolidation

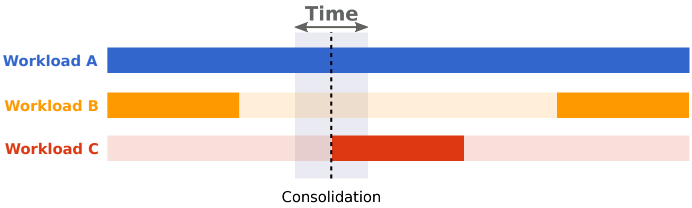


With Linux Containers

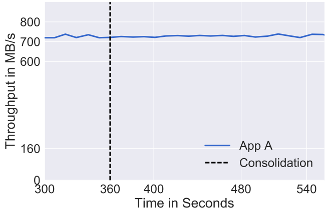


A is disturbed for 40s

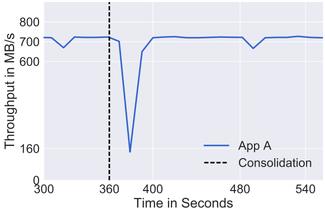
But performance losses occurs during consolidation



Without Linux Containers



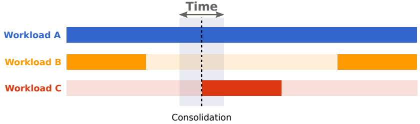
With Linux Containers

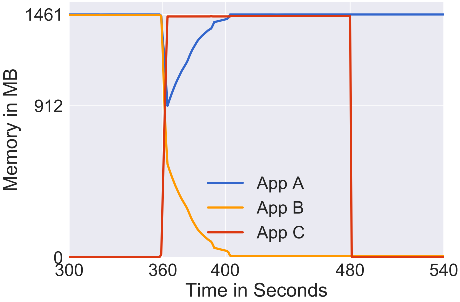


A is **not** disturbed

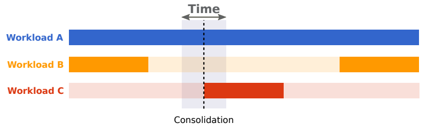
A is disturbed for 40s

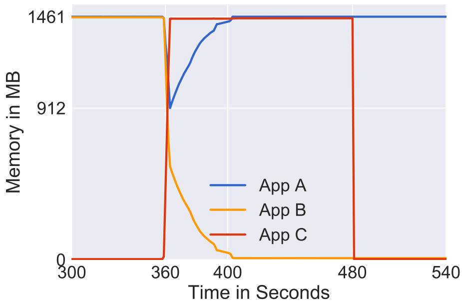
Root Cause Analysis : Memory of A, B and C





Root Cause Analysis : Memory of A, B and C



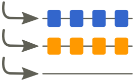


The pages of B are the LRU

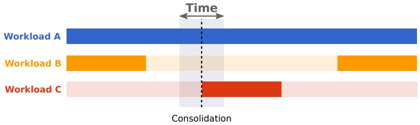
Structures: before C's allocation

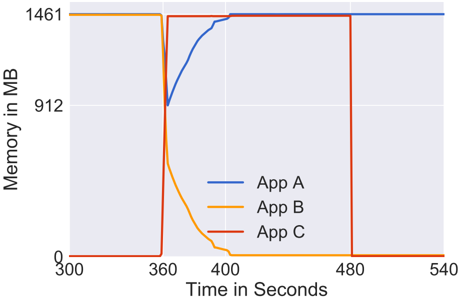
Without Containers:
One LRU

With Containers:
One LRU per Container

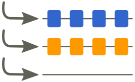


Root Cause Analysis : Memory of A, B and C

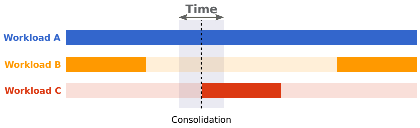




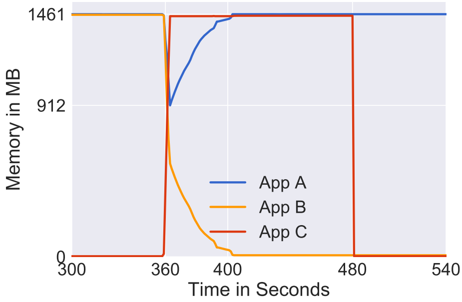
The pages of B are the LRU

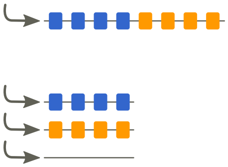
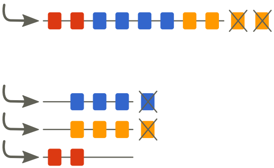
Structures:	before C's allocation	after C's allocation
Without Containers: One LRU		
With Containers: One LRU per Container		

Root Cause Analysis : Memory of A, B and C



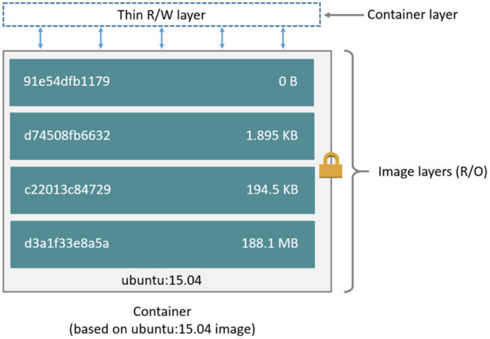
The pages of B are the LRU



Structures:	before C's allocation	after C's allocation
Without Containers: One LRU		
With Containers: One LRU per Container		

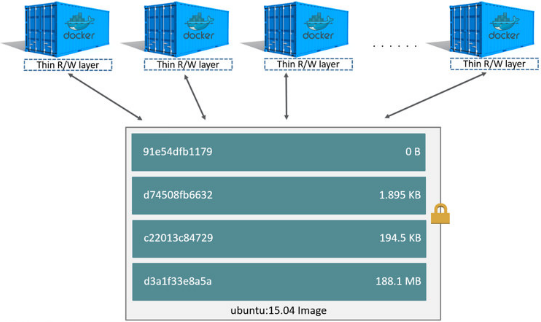
docker

Image layers



- Updates in the C_layer (CopyOnWrite has an overhead on first write)
- C_layer lost when the container is removed
- Dockerfile describe how to build layers
- C_layer can be commited to form a new image

Image layers



Sharing layers avoids duplication:

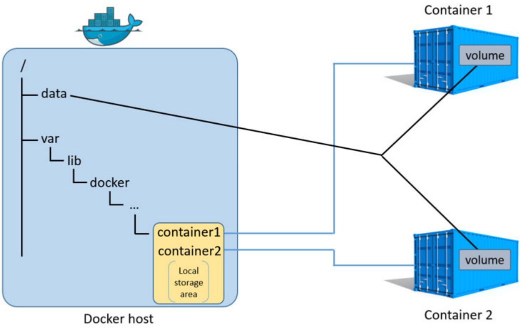
- Saves storage
- Speeds boot of new instances

Image layers

Image layers are:

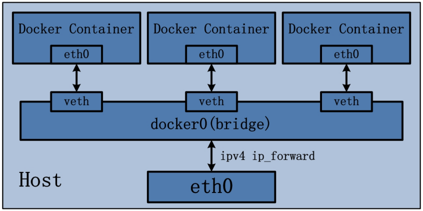
- good at storing the initial state of container
- bad at storing data for I/O heavy applications

Volumes are designed for storing data



- Volumes are independent of the running container
- Volumes can be shared among containers

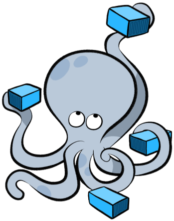
Networking



Docker uses Network Namespaces combined wih:

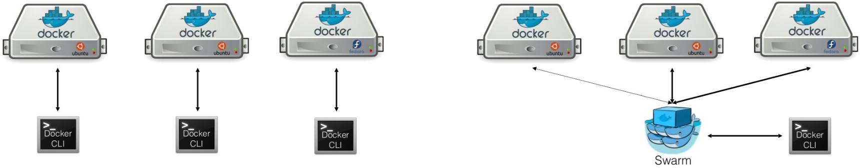
- Virtual Ethernet Bridges (docker0 or br-XXX)
- Virtual Ethernet Devices (vethXXX in host, eth0 in container)
- Packet Forwarding (sysctl net.ipv4.conf.all.forwarding)
- NAT (iptables -L)

docker-compose



```yaml
version: '2.2'
services:
  memtiera:
    hostname: memtiera
    build: images/memtier_benchmark
    links:
      - redisa
      - influxdb
  redisa:
    image: redis:latest
  influxdb:
    image: influxdb:latest
  collector:
    build: images/collector
    command: [ "--influx", "--influxdbhost=influxdb" ]
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
    links:
      - influxdb
```

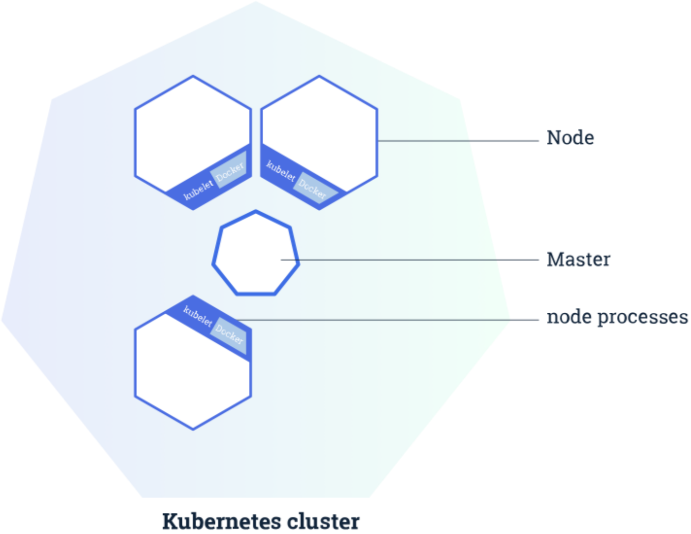
docker swarm



Use docker client on a cluster of machines as if it was one machine.

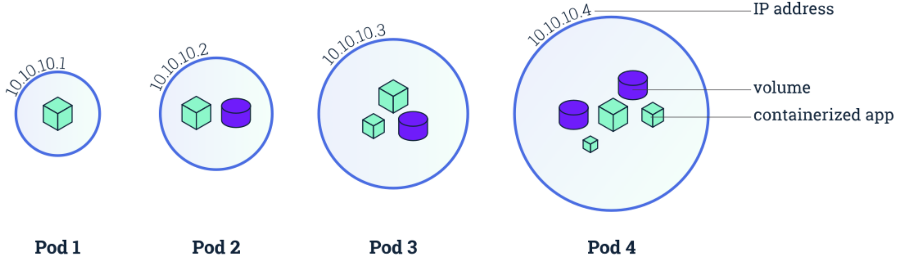
Kubernetes

Cluster

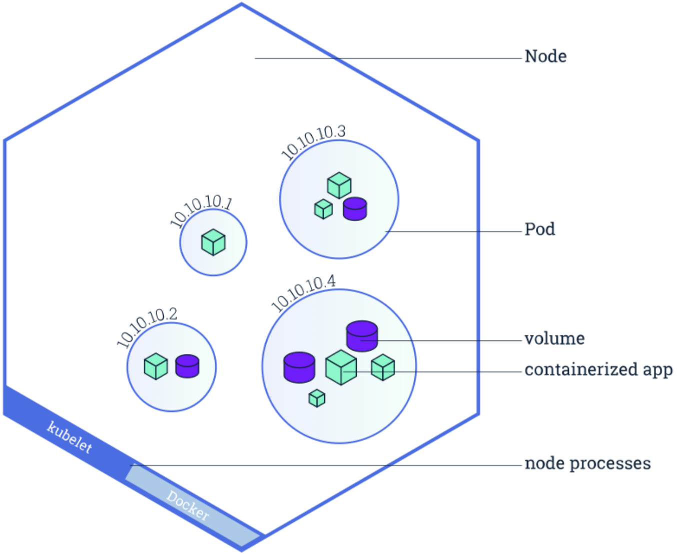


Kubernetes

Pods

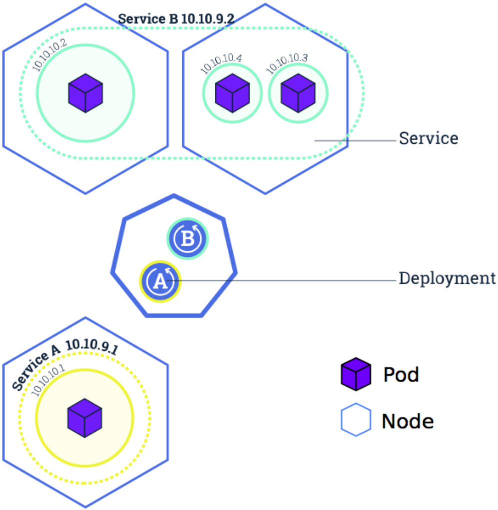


Kubernetes
Nodes



Kubernetes

Services and Microservices



APPENDIX

Book References

- Modern Operating Systems, Tanenbaum, Andrew S. and Bos, Herbert
- Understanding the Linux Kernel, Daniel Bovet, Marco Cesati

Wikipedia References

- Hypervisor
- Full virtualization
- Virtual machine
- Comparison of platform virtualization software
- Operating-system-level virtualization
- Docker (software)
- Cgroups
- Linux namespaces
- AppArmor
- Seccomp

Man/Linux References

- man cgroups
- man namespaces
- man capabilities
- kernel doc cgroup-v1
- kernel doc cgroup-v2
- Redhat resource management guide
- lwn Control groups series by Neil Brown

Other References

- Docker get started
- Docker images
- Docker networking
- docker-compose overview
- docker-compose django
- docker-compose rails
- docker-compose wordpress
- kubernetes
- criu: checkpoint/restore