

# Understanding the Design and Applications of Container based System Virtualization

**M.Tech Seminar Report**

Submitted in partial fulfillment of the requirements  
for the degree of

**Master of Technology**

by

**Prashanth**

Roll No: 153050095

under the guidance of

**Prof. Purushottam Kulkarni**



Department of Computer Science and Engineering  
Indian Institute of Technology, Bombay  
Mumbai

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Drawbacks of existing system from a cloud provider's view . . . . .	2
1.1.1	Resource Control . . . . .	2
1.1.2	Resource Isolation . . . . .	2
1.2	Requirements of a New System . . . . .	3
1.3	Virtual Machines . . . . .	3
1.4	Containers . . . . .	4
<b>2</b>	<b>Initial Designs</b>	<b>5</b>
2.1	Resource Container . . . . .	5
2.2	Open-VZ . . . . .	6
2.3	Linux V-Server . . . . .	6
2.3.1	System Overview . . . . .	6
2.3.2	Resource Isolation . . . . .	7
2.3.3	Security Isolation . . . . .	7
2.3.4	File System Unification . . . . .	7
<b>3</b>	<b>Virtualized Containers</b>	<b>8</b>
3.1	Control Groups . . . . .	8
3.1.1	Changes to Kernel . . . . .	9
3.1.2	Hierarchies . . . . .	9
3.1.3	User Space APIs . . . . .	11
3.1.4	Different Subsystems Available . . . . .	11
3.2	Namespaces . . . . .	14
3.2.1	Changes to Kernel . . . . .	14
3.2.2	Six existing namespaces . . . . .	15
3.3	Disk images . . . . .	16
3.4	Present day virtualized containers . . . . .	17
3.5	System Containers . . . . .	17
3.5.1	Linux Containers (lxc) . . . . .	18
3.6	Application Containers . . . . .	18
3.6.1	Docker . . . . .	18

3.7	Summary	21
<b>4</b>	<b>Comparing Containers against Virtual Machines</b>	<b>22</b>
4.1	Metric based comparisons	23
4.1.1	Memory Bandwidth and Size	23
4.1.2	CPU Utilization	25
4.1.3	Network Throughput and Latency	26
4.1.4	Disk I/O Throughput	27
4.1.5	Throughput and Latency for MySQL server & Redis key-value store	29
4.1.6	Startup Latency	29
4.2	Summary	30
<b>5</b>	<b>Optimizations for Startup Latency and Disk I/O</b>	<b>31</b>
5.1	Problem of Startup Latency	31
5.1.1	Proposed Optimizations	32
5.2	Problem of Disk I/O Throughput	32
5.2.1	Proposed Optimizations	33
<b>6</b>	<b>Applications of Containers</b>	<b>34</b>
6.1	Google Borg	34
6.1.1	System Outline	34
6.1.2	Features	35
6.1.3	Architecture	35
6.2	Google Kubernetes	36
6.2.1	Architecture	36
6.3	Facebook's Tupperware	36
6.3.1	System Components	37
6.3.2	Running an Application using Tupperware	37
<b>7</b>	<b>Conclusions</b>	<b>38</b>
7.1	Future Scope	39
<b>A</b>	<b>Appendix: Demonstrations</b>	<b>42</b>
A.1	Control Groups	42
A.1.1	Checking all the mounted cgroup subsystems	42
A.1.2	Mount/Unmount a New subsystem hierarchy	43
A.1.3	Creating a cgroup and adding process to it	43
A.1.4	Setting constraints	45
A.2	Namespaces	46
A.2.1	Creating a new network namespace	46
A.2.2	Creating virtual tunnel between default and new NS	47
A.3	Linux Containers	47

A.3.1	Installing and running a simple container . . . . .	48
A.3.2	Some helpful commands in lxc . . . . .	49
A.3.3	Python API to use lxc . . . . .	50
A.3.4	Disk Images in lxc . . . . .	50
A.4	Docker . . . . .	51
A.4.1	Installing required dependencies for Docker . . . . .	52
A.4.2	Installing Docker . . . . .	52
A.4.3	Running a Docker Container . . . . .	53
A.4.4	Disk Image layers in Docker . . . . .	53
<b>B</b>	<b>Appendix: Benchmarks</b>	<b>56</b>
B.1	Memory . . . . .	56
B.2	CPU . . . . .	56
B.3	Network . . . . .	57
B.4	Disk I/O . . . . .	57
B.5	Application . . . . .	57

# List of Tables

4.1	Performance metrics based on which different papers made comparisons . . . . .	23
4.2	Relative ordering in terms of Memory . . . . .	24
4.3	Relative ordering in terms of CPU . . . . .	26
4.4	Relative ordering in terms of Network . . . . .	27
4.5	Relative ordering in terms of Disk I/O . . . . .	28
4.6	Relative ordering in terms of Application Performance Metrics . . . . .	29
4.7	Relative ordering in terms of Startup Latency . . . . .	29
B.1	Benchmarks used on memory based experiments . . . . .	56
B.2	Benchmarks used on CPU based experiments . . . . .	56
B.3	Benchmarks used on network based experiments . . . . .	57
B.4	Benchmarks used on disk I/O based experiments . . . . .	57
B.5	Application Benchmark used . . . . .	57

# List of Figures

3.1	Control groups illustration using 3 controllers . . . . .	10
3.2	Example for PID namespace in Linux . . . . .	15
3.3	Docker Architecture, source:[1] . . . . .	19
3.4	Disk layers in Docker . . . . .	20
4.1	Average Memory Footprint (proportional set size) of VMs with KSM enabled versus system (LXC) versus application containers (LXC-Bash). Lower the better, source:[2]	24
4.2	CPU utilization in HPC environments between KVM and LXC, source:[3] . . . . .	25
4.3	source:[4] (a) Network transmission efficiency between native, Docker and KVM(b) Network latency comparison between native, Docker and KVM . . . . .	27
4.4	Filebench I/O performance as number of guests increases on a 4-core machine. Higher is better , source:[2] . . . . .	28
5.1	Typical container architecture used by organizations to handle disk Images . . . . .	31
5.2	Mutlilanes Architecture to handle a single Host with multiple containers with isolated I/O stacks for each container . . . . .	33
6.1	Architecture of Google Borg, source: [5] . . . . .	35
A.1	Checking current subsystems mounted . . . . .	42
A.2	Mounting a new hierarchy . . . . .	44
A.3	Mounting a new hierarchy . . . . .	45
A.4	Creating new namespace . . . . .	46
A.5	Creating new namespace . . . . .	48
A.6	Running a lxc container . . . . .	49
A.7	Exploring lxc images . . . . .	51
A.8	Running Docker container . . . . .	53
A.9	Docker pulling images . . . . .	54
A.10	Difference between layers of a same image . . . . .	54

# 1. Introduction

In the early 60's computers were an expensive and inaccessible commodity and hence, computer resources worked similar to how telephony lines are used today. The customer had to send their input as processing request to a centralized computer which would process the data transmitted and return the output as a response. All this communication typically occurred over a simple network. The customer would then be charged for the resources accessed in the shared system based on the time for which it was used for, just as how we pay for telephony today that is by paying for time which we utilized the telephony channel. This was how the concept of time-sharing came into play. Fast forward 50 years over all the evolution of computers, and this is one of the models we extensively use today and it has been given a fancy name called **“Cloud Computing”**.

Cloud computing providers have computing resources that they wish to offer as a service to customers. They charge customers based on the usage of their resources. However unlike the 60's mere timesharing of the resources present at a cloud provider doesn't work out to be an effective solution today.

To support the argument above, consider a case as a hypothetical situation where a cloud computing provider has one system with 16GB of RAM, 4 CPUs, 10TB of storage, and a 100 mbps network connection. Now consider the case where the provider wishes to provide this system as a shared resource to two users who wish to run their own web services but need only half of each of the resources i.e 8GB of RAM, 2 CPUs, 5TB of storage, and 50 mbps network connection. The provider could allow both customers to use the resources simultaneously, but this would cause the problem of isolation. To avoid the problem of isolation, he could use the traditional technique of a timesharing, he could switch the access privileges between the customers after a regular interval of time. This described technique is a valid technique. But is this the most effective and efficient solution ?

There are two main drawbacks to the above proposed solution. Firstly, since both the customers are running a web service each, they would require a close to 100% uptime. But, with the above proposed solution the customers would only receive close to 50% uptime each, even after ignoring the time delay caused by context switches. The uptime problem could also be solved to some extent by using a request queue which queues requests to customers when they are not in context, but this would require additional or a part of the existing resources to be deployed to enforce this. The more important drawback of the above proposed solution is that the resources deployed are being underutilized. At every point only 50% of the entire available resources are being utilized. 50% could also be acceptable to some extent, but consider the case when you have hundreds or thousands of customers. Your utilization would degrade drastically and also a lot of the compute resources would be wasted in switching time.

## 1.1 Drawbacks of existing system from a cloud provider's view

The drawbacks of implementing timesharing concept as a solution to cloud provider was discussed in the previous section. However if we dig deeply, we realise that the root cause exists in the design of OS more than anything. This section expands on the same.

Operating systems (OS) play a key role in providing required services to a cloud provider, who in turn make use of these service to enforce policies on their customers. Operating systems were originally designed to manage and control the different resources present on a system. Operating system running were initially designed with a typical use case in mind, although it has tried to generalize the design as much as possible overtime. The current OS provides an very effective design in most cases, but there are couple major drawbacks of the above existing design with respect to support to cloud providers and they are - resource control and isolation. These drawbacks have been discussed below.

To illustrate the existing drawbacks, consider the initial solution proposed which involved providing customers of a cloud provider concurrent access to a single system.

### 1.1.1 Resource Control

Consider situations where the system isn't able to allocate the requested resources by a customer process due to unavailability of resources at a given instant. Hence it defers the request and allocates the requested resources when the resources are released by other processes. The resources would be allocated to the requesting process at a later instant when they are freed. This causes execution of a process to be delayed and this delay may not be always acceptable. There may even be situations where a malicious process initialized by another customer executing on a system blocks all or majority of the resources on a system and there by not allowing other processes to execute.

To demonstrate the support of the above argument, consider a customer running an apache demon attached to a PHP backend system to handle incoming HTTP requests on the shared system. Now consider a SMTP mail server running on the same shared system which provides a mailing service by another customer. Now consider the situation where the apache server receives a flash crowd due to which most of the system resources are being utilized to handle the enormous traffic which ultimately crashes the server system and there by crashing both the apache and mail server. In this case, the mail server crashed without it being the cause of the instability in the system. The above scenario illustrates the problem of resource control in a cloud server where there no mechanism by which cloud providers can place restriction on a process or group of processes.

### 1.1.2 Resource Isolation

Now that we have established the problem of resource control, let's have look at the problem of resource isolation. Consider the case where two different processes managed by two different customers. Due to the inherit simplistic design of the operating system, these processes would most likely be able to view the resources utilized by one another and might also be able to manipulate them (depending on the resource isolation provided by the underlying OS). Thinking about it logically, these processes executed by different customers and ought to hide process specifics from one another. Hence the inherit design of an operating system provides a very week form of isolation.



## 1.2 Requirements of a New System

To overcome the two problems discussed above, there was a need to come up with new mechanisms by which cloud providers have some control and isolation over processes executing along with their resources present in a system. This would ease the task of enforcing policies by the cloud providers on the customers executing processes on a shared system.

Here after, we would be referring to a set of processes executed by an independent customer of a cloud provider as a process group. Although the term process group could exist without having the involvement of a cloud provider. A cloud provider must be able specify the following policies on a process group to the OS, and the OS in return must be able to enforce them.

1. Account different resources used by process groups.
2. Specify resource constrains.
3. Constrains maybe be resource (CPU/ IO/ Memory etc.) specific.
4. Processes running in the same process group must be kept in isolation from other processes on the same system and vice-versa.
5. Resources utilized by process groups must be kept hidden from other processes on the system.
6. Each process group must be able to have its own group of users who have access to the processes.
7. Users executing on a process group in a shared machine must only be able to access and view processes and resources that have been assigned to them.
8. Processes-ID, User-ID, resource-ID in a process group must be reusable in a different process groups present on the same physical machine.
9. Should provide minimum overhead on enforcing the above mechanisms

## 1.3 Virtual Machines

A Virtual Machine (VM)[\[6\]](#) is an isolated virtual environment created by OS that is running upon another piece of specialized software called hypervisor. Hypervisor typically emulates virtual devices with the help of other software mechanisms to virtual machines to treat them as real hardware. Hypervisor itself maybe either running directly on the physical hardware or on a host OS and they are called Bare-metal(Type-1) and Hosted(Type-2) hypervisors respectively. The end user has the same experience on a virtual machine environments as they would have on OS attached real physical hardware. They provide highly secure and independent systems. Virtual machines can even be configured to limit the amount of resources available to them by the hypervisor.

Virtual machines have been extensively used by cloud providers to host customers for over a decade now. They fulfill most of the desired properties of the required system proposed in the previous section. Hence to solve our initial problem, we could run a hypervisor and deploy two instances of virtual machines with the exact same specification as wanted by the two customers. Using this method, our CPU utilization and server uptimes comes close to 100% if not for exact. One major drawback of Virtual Machines is the heavy performance overheads they incur which has been discussed in [Chapter:4](#).

## 1.4 Containers

Container in simple terms can be defined as,

**“Container is a process or set of processes grouped together along with its dependent resources into a single logical OS entity. It enables multiple isolated user-space instances on a host machine.”**

Containers are built as an extension to the existing operating system and not as an independent system. They were the other later solution proposed to the problem discussed earlier. Processes of both the customers can be grouped together using one instance of container for each. Resource constraints can be specified as needed by the customer onto their respective containers. This solution also achieves 100% CPU utilization and uptime for each of the customers.

The biggest advantage of using containers over virtual machines is that they provide much lesser performance overheads which has been discussed in [Chapter:4](#). Although containers are a generic mechanism of grouping processes, they are often used in a virtualization context and have a few additional features to it. An in-depth discussion on the details of virtualized containers would be done in upcoming chapters. Right now we just trying to setup a basic outline of a container to set up the discussion on the initial works proposed.

## 2. Initial Designs

Often people have this misunderstanding that container is a recent technology and has only been around for while. This confusion is because containers are often treated as light weight alternatives to virtual machines, and the use of containers in the context of virtualization has been popularized only since about 2013. But all this assumption is a myth. The idea of containers dates back to the year 1999, where Resource Containers [7] were proposed to provide fine grained control over resources in an operating system which had nothing to do with virtualization.

Having said all this, it has to be noted that containers are often dealt with from a virtualization viewpoint and throughout our discussion we would also be discussing from a virtualization viewpoint. In the year 99' Alexander Tormasov proposed a new direction towards virtualized containers. This lead to the building containers which mentioned virtualized containers as

“A set of processes with namespace isolation, file system to share code/ram and isolation in resources ”

He lead his team to building the same with adding bits and features to Linux kernel[8] version 2.2. In the year 2002, they released their initial version of virtualized containers known as SWsoft (now known as Virtuozzo, owned by Parallels). It was initially launched for Linux and later in 2004 released the same for windows. In the year 2005, SWsoft created the Open-VZ[9] project to release the core of Virtuozzo under GNU GPL.

While all this was going on, there was another parallel project started by Jacques Glinas called the Linux V-Server[10] in the year 2003 which also aimed to develop virtualized containers. Similarly over the years several new vendors and projects were started at building more and more implementations to virtualized containers. To get an idea about the initial design attempts in containers let's have a look at Resource containers, Open-VZ and Linux V-server.

### 2.1 Resource Container

Resource containers[7] proposed a fine-grained resource management model. This was one of the initial designs proposed in 1999. Resource container is an abstract OS entity that logically contains all the system resources being used by an application (A group of processes) to achieve a particular independent event. All user and kernel level processing for an application was charged to its appropriate resource container. Containers provided attributes like scheduling parameters, resource limits and network QoS. Scheduling model included priorities, CPU shares or CPU usage limits. An process in the system was scheduled according to the scheduling attributes of its container. They formed a hierarchy of containers, where child containers were restricted by limits on parents. Sockets were also attached on a per container

basis and made use of a new `sockaddr` namespace to isolate sockets used by different applications from interfering with other applications. It is to be remembered that resource containers only dealt with resource control over a group of processes and didn't mention anything about virtualization.

## 2.2 Open-VZ

Open-VZ [9] has evolved a lot over time, but this section discusses about its initial architecture. Open-VZ creates isolated virtual environments called Virtual Environments (VE). A VE has its own set of processes (including `init`), users (root included), network interfaces, routing tables, firewalls etc. It makes use of a `vzctl` utility which implements a high level interface to manage the VEs. It provides templates to create new VE. A template consists of a set of packages which is chrooted when onto the VE when deployed.

Each VE has its own set of resources like: files, users, process-trees, network, devices and IPC objects. Its management system consists of three components namely: two-level disk quota, fair CPU scheduler and user bean-counters. The two level disk quota specifies disk quota per VE and per user or user-group. The CPU scheduler first decides which VE to execute and then decides which process to execute from the selected VE. Bean-counters are used to count and limit resource usage. Lastly, Open-VZ has also performs live-migration using stop and copy where the entire VE is frozen and copied to a file and file is moved to a new location and unfrozen there to its resume execution.

## 2.3 Linux V-Server

Operating Systems designed for personal computers provide a weak form of isolation but, reasonable support for sharing between applications. Hypervisors on the other-hand provide full isolation between VMs (used to run applications) but, provide no support for sharing. This degree of isolation provided by VM affects the efficiency of the system. V-server [10] focused to come up with a light weight virtualization technique which could be used in scenarios where it is acceptable to trade isolation for efficiency which would try to fit a solution in somewhere between a native OS and hypervisor.

VServer proposed a container-based operating system (COS) which derived ideology from resource and security containers. Resources were allocated to each VM on creation but could be dynamically altered. V-Server provided Resource isolations in CPU, I/O and Storage limits. Security isolation is achieved using techniques like Process Filtering, Network Separation, Chroot Barrier, Capability Mask etc. Common files between VMs were shared using copy-on-write (COW) techniques.

### 2.3.1 System Overview

The system was given a shared OS image. There was a privileged host VM used to managed all the guest VMs. Applications running on guest were unaware of them being virtualized namespaces and access controls were used to secure the VM. Access controls were used to check VM permissions while accessing resources in runtime.

### 2.3.2 Resource Isolation

CPU Scheduling was achieved using a token bucket filter (TBF) on top of Linux CFS scheduler. Each VM was provided with a token bucket that accumulates tokens at a specific rate. On each clock tick utilized by the VM, the bucket was charged with a token. Once the bucket empties, the VM was preempted and was kept under wait until minimum tokens were accumulated by the VM bucket. A range of isolation policies could have been implemented using this technique. The rate of tokens accumulated depended on reservation or share for the corresponding VM.

Network isolation was achieved using a Hierarchical Token Bucket (HTB) queue. Each VM was given a reserved rate and shared rate. Every VM could consume up to reserved rate and there after can share the bandwidth up to shared rate. Disk I/O was managed using standard Linux Completely Fair Queuing (CFQ) I/O Scheduler.

V-Server could specify limits to the memory and storage limits for a particular VM. The maximum number of disk blocks/inodes to allocate. It could also specify the Max RSS, anonymous memory pages, number of pinned pages etc. on the memory.

### 2.3.3 Security Isolation

PIDs were global across all VMs. Hence process filters were used to hide processes which don't belong to that particular VM. It also provided a fake init process with PID 1 to all VMs. To provide global view of all VMs, a special spectator VM was used. V-server didn't virtualize the networking subsystem instead shares the networking subsystem among all the VMs. V-Server used chroot barrier on the parent directory of each VM to prevent unauthorized access by other VMs.

### 2.3.4 File System Unification

The files which were common to more than one VM are hard linked onto a shared file system. These files were copy-on-write which implied these files created a private copy for each VM when they were modified. The process of finding common files and hard-linking them was called unification.

## 3. Virtualized Containers

After the initial designs that were proposed which were described in the previous section, in 2006 Eric W. Biederman proposed "Multiple Instances of the Global Linux namespaces" to support process group isolation in Linux symposium. Initially 10 different namespaces were proposed, of which the first one already existed as mount namespace and was introduced in 2002. Over the years the namespaces proposed have been implemented one by one and added into the Linux kernel. Right now 6 of the 10 namespaces proposed, are present in the Linux kernel. More are expected to be implemented in the future kernel versions.

In 2007, Google presented a new generic method to solve one of the initial problem of resource control with the cgroups [11] project. This allowed resources to be managed based on process groups in terms of accounting and control. In 2008, this was merged with the mainline kernel. More about cgroups and namespaces are mentioned in the section:3.1 and section:3.2 respectively.

As mentioned earlier, between the years 2007-11 many companies started providing their own patches to support their container management systems and had their own implementation. The virtualization world started realizing the true potential of containers but also noticed a flaw in the design. As in the case of hypervisors which exists today in the mainline kernel to run virtual machines, multiple vendors provide multiple patches to the kernel source to support them. They have their own implementations which are redundant code and each vendor's implementation may not be the most efficient one in all dimensions. The same problem started repeating with containers where each vendor gave their own implementation and a patch.

In 2011, all the container vendors/projects came together and agreed upon a common unification to avoid the redundancy in kernel patches which existed in hypervisors. They agreed to take support from cgroups and namespaces and provide their APIs on top of them to manage their containers. In 2013, first Linux kernel supporting this was released. Today most container technologies make use of these two features and build upon them to provide their own version of a container manager.

Container technologies support different existing operating systems like Linux, Windows, OS-X etc. But for the purposes of our discussion we try to focus on the Linux kernel as it is the most widely used OS for deploying containers. Almost every system built is typically an amalgamation of several smaller components. Similarly, there are three crucial elements which constitute a Linux container, which are - cgroups, namespaces and disk images. The current chapter introduces to them, before moving on to actual design and contents of the present day containers.

### 3.1 Control Groups

The solution to process control and accounting was proposed by Google in 2007 which was originally called Generic Process Containers [11] and was later renamed to Control Groups (cgroups), to avoid

confusion with the term Containers. A cgroup/subsystem refers to a resource controller for a certain type of CPU resource. Eg- Memory cgroup, Network cgroup etc. It derives ideas and extends the process-tracking design used for cpusets system present in the Linux kernel. There are 12 different cgroups/subsystems, one for each resource type classified.

For the purpose of our discussion we will stick to subsystem as the terminology referring to individual resource control and cgroup to refer a cgroup node in hierarchy. The Linux kernel by default enables most subsystems hence even if you think you are using a system without using subsystems, technically all your processes are attached to the default (root) cgroup of each of the subsystem. Default (root) cgroups are described in the coming sections. The overheads introduced by cgroups are negligible.

### 3.1.1 Changes to Kernel

The changes to the kernel which are required to understand the system described here. These are data structures introduced to incorporate cgroups into the kernel. Data structures are shown in green characters in Fig:3.1.

- `container_subsys` is single resource controller and it provides callbacks in the event of a process group creation/modification/destruction.
- `container_subsys_state` represents the a node in a resource controller hierarchy on which constraints can be specified.
- `css_group` holds one `container_subsys_state` pointer for each registered subsystem. A `css_group` pointer is added to the `task_struct`.

Set of tasks with same process group membership are binded to the same `css_group`. The major changes to the kernel code to incorporate cgroups have been listed,

- Hook at the start of `fork()` to add reference count to `css_group`
- Hook at the end of `fork()` to invoke subsystem callbacks
- Hook in `exit()` to release reference count and call exit callbacks

### 3.1.2 Hierarchies

It is possible to have 12 different hierarchies, one for each subsystem, or a single hierarchy with all 12 subsystem attached, or any other combination in between. Every hierarchy has a parent cgroup(node). Each cgroup in a hierarchy derives a portion of resources from its parent node. And resources used by the child is also charged for the parent. Each process in the system has to be attached to a node in each hierarchy, this node may even be the default node. It starts from the root (default) cgroup in each hierarchy. The root node typically has no constraints specified and is bounded by the actual resources available in the system. Although a node could be potentially attached to multiple subsystems at once, our discussion deals with a node being attached to a single system at once, as multiple subsystems attached to a single hierarchy is uncommon in real world use currently.

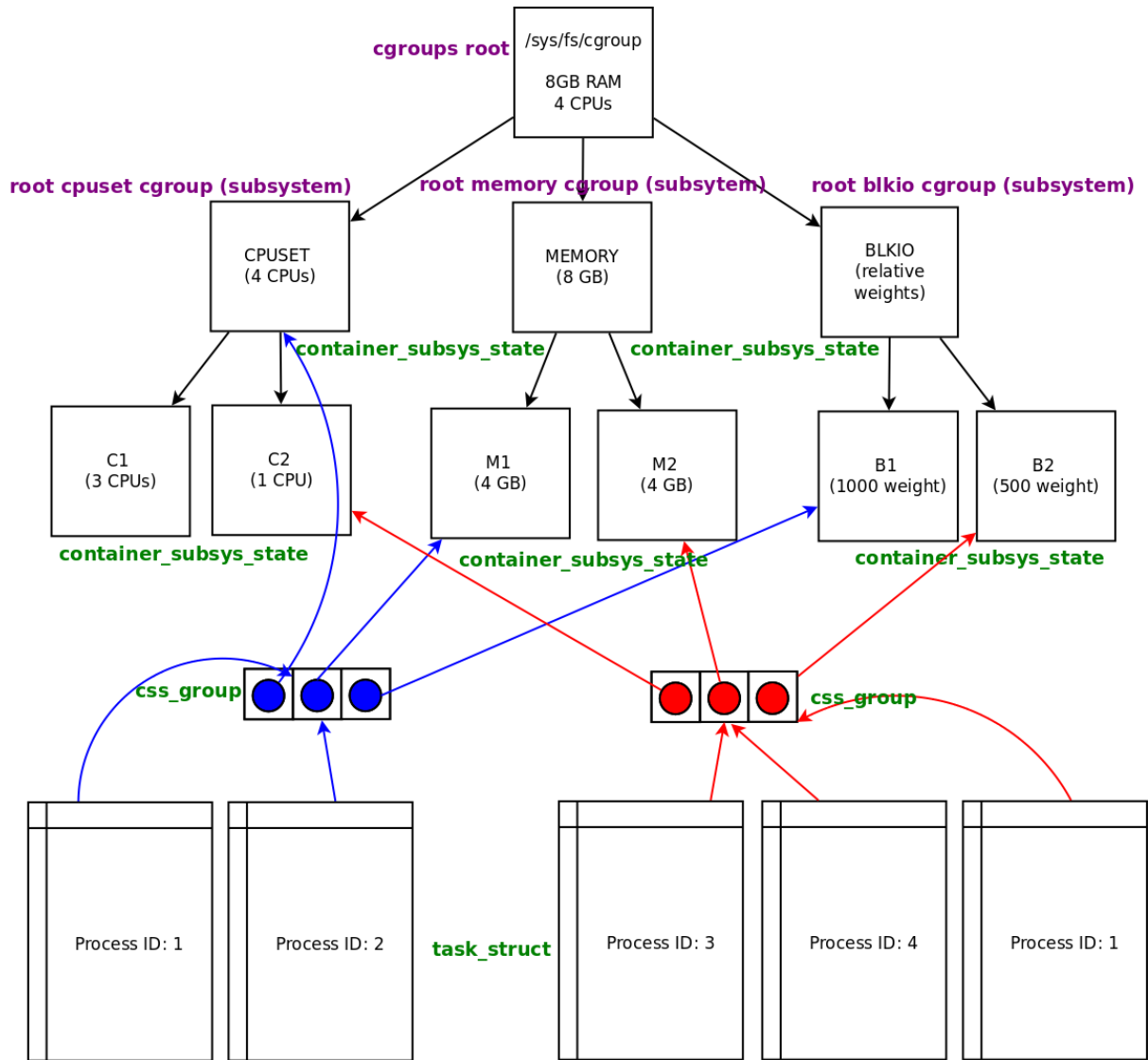


Figure 3.1: Control groups illustration using 3 controllers

Fig:3.1 illustrates a minimalistic outline of a cgroups hierarchy with 3 subsystems mounted in a system onto their own hierarchies. The three subsystems mounted are - memory, cpuset and blkio and are mounted at `/sys/fs/cgroups/`. Memory root cgroup of 8GB is divided into two cgroups M1 and M2 of 4GB each. cpuset root cgroup of 4CPUs is divided into two cgroups C1 and C2 of 3CPUs and 1CPU respectively. blkio root cgroup of is divided into two cgroups B1 and B2 of 1000 and 500 as relative weights respectively. Every process which attaches itself to the same set of subsystems are referred by a single `css_group` which in turn points to the cgroup node the process is attached to. In the Fig, processes 1,2 attach itself to the blue `css_group` and 3,4,5 to the red one. The `css_group` in turn has pointers to `container_subsys_state` that is one for each cgroup. Notice how the blue `css_group` points to the root cpuset cgroup there by assigning it all the CPUs in the system which is also a valid and default value to attach processes.



### 3.1.3 User Space APIs

Control groups are managed using pseudo file system. A root cgroups file system directory contains a list of directories which refer to different hierarchies present in the system. A hierarchy refers to a subsystems/ or a group of subsystems attached to a single hierarchy. `mount()` is used to attach a new hierarchy to a system, and it should indicate the subsystems to be bound to an hierarchy. Each hierarchy will have its own hierarchy of directories (nodes).

Every cgroup(node) in hierarchy contains files which specify control elements of the cgroup in the hierarchy. Only unused subsystems maybe be mounted. Initially all processes are in the root cgroup of all hierarchies mounted on the system. Creating a new directory `mkdir()` in cgroup directory creates the new child. Can be nested in any desired manner. Special control file `cgroup.procs` is used to track all processes in the cgroup. `rmdir()` can be used to remove cgroup provided no processes are attached to it.

### 3.1.4 Different Subsystems Available

This section gives a brief intuition to the different subsystems available. Although an in-depth discussion of each on the same is beyond the scope of our discussion as each subsystem itself contains content enough to be discussed over a large chapter.

#### 1. Memory

Memory subsystem use a common data structure and support library for tracking usage and imposing limits using the "resource counter". Resource controller is an existing Linux implementation for tracking resource usage. Memory cgroup subsystem allocates three **res\_counters**. The three of them are described below.

**i. Accounting:** Accounting memory for each process group. Keeps track of pages used by each group. Pages can be classified into four types.

- Anonymous: Stack, heap etc.
- Active: Recently used pages
- Inactive: Pages read for eviction
- File: Reads/Writes/mmap from block devices

**ii. Limits:** Limits can be set on each cgroups. Limits are of two types - soft and hard. Soft limit is the limit up to which the system guarantees availability. Hard limit is the limit up to which the system tries to accommodate, but cannot guaranty this if system is under memory pressure. Limits can be set in terms of byte for,

- Physical memory
- Kernel memory
- Total memory (Physical + Swap)

iii. **OOM:** Out Of Memory killers are used to kill processes or trigger any other such event on reaching hard limit by a process group.

## 2. HugeTLB

HugeTLB restricts the huge pages usable by a process group and also enforces a controller limit during page fault. SIGBUS typically occurs when you try to access a file beyond your mapped region. Since HugeTLB doesn't support page reclaim, when a process group tries to overdo its limit, it send a SIGBUS to the process. HugeTLB subsystem internally it also makes use of one **res\_counters**, similar to memory subsystem.

## 3. CPU

CPU subsystem manages how the scheduler shares CPU time among different processes groups. Access to resources can be scheduled using two schedulers - Completely Fair Scheduler (CFS) & Real-Time scheduler (RT). The stats related to a execution time and throttled time for a process group can be viewed by running `cat cpu.stat`. It allows us to specify the following parameters respect to a cgroup,

- **Period:** Specifies a period of time in microseconds, for how regularly a cgroups access to CPU resources should be reallocated. Maximum is 100000 microseconds.
- **Quota:** Total amount of time in microseconds, for which all processes in cgroup can run in a given time period.
- **Shares:** Shares represent relative shares with respect to other cgroups in the system. Default value of CPU shares is 1024.

Quota and period parameters operate on a CPU basis. To allow a process to fully utilize four CPUs, for example, set `cpu.cfs_quota_us` to 400000 and `cpu.cfs_period_us` to 100000. Say you wish to enforce a 2:1 priority to two cgroup nodes running on a system. You leave the former cgroup at 1024 shares and set the latter one at 512 shares. This would give you the desired ratio.

## 4. CPUset

The CPUset subsystem pins individual logical CPUs and memory nodes to cgroups. Its support existed in the kernel even before introduction of cgroups. These are mandatory parameters, which indicates their values to be set before moving a process into them.

Say you have 4 logical CPUs and 4 memory nodes. You can specify mappings as `cpuset.cpus = 0-2` (Indicates CPU nos 0,1,2 pinned) `cpuset.mems = 0,3` (Indicates Memory nodes 0,3 pinned). Apart from the above mentioned parameters, there are several other parameter that can be configured. But the above two mentioned are of most interest.

## 5. Cpuacct

CPUacct as the name suggested, all it does is accounting; it doesn't exert control at all. It measures the total CPU time used by all processes in the group per logical CPU/Total. Breaks into "user" and "system" time of the total CPU time used by a process group. It also reports a per CPU time consumed by each process group.

## 6. Block I/O

Block IO Keeps track of I/Os for each group using - Per block device, Reads vs Writes, Sync vs Async. It can sets throttle limits for each group using - Per block device, Reads or Write, Block Operations or Bytes. It can also set relative weights for each group.

## 7. Network class

The `net_cls` subsystem tags network packets with a class id that allows the Linux traffic controller (tc) to identify packets originating from a particular cgroup. `net_cls.classid` contains a single value that indicates a traffic control handle. The format for these handles is: 0xAAAABBBB, where AAAA is the major number in hexadecimal and BBBB is the minor number in hexadecimal. The traffic controller can be configured to assign different priorities to packets from different cgroups using simple command line commands.

## 8. Network Priority

The Network Priority (`net_prio`) subsystem provides a way to dynamically set the priority of network traffic per each network interface for processes within various cgroups. A network's priority is a number assigned to network traffic and used internally by the system and network devices. Network priority is used to differentiate packets that are sent, queued, or dropped.

Both Network class and Network Priority subsystems delegate their work to respective network interfaces, which leads to additional work at a lower layer. The traffic controller can be configured to assign different priorities to packets from different cgroups using simple command line commands for both the subsystems.

## 9. Device

Lets you control which devices can be accessed by a cgroup. By default a cgroup cannot access any device. Each entry has four fields: type, major, minor, and access. Access permissions are specified as Read(r), Write(w) and Modify(m). Modify allows to create device files that don't exist. It maintains an list of all devices with the above access permissions specified.

## 10. Perf Event

Collects various performance data for a process group. All cgroups in that hierarchy can be used to group processes and threads which can then be monitored with the perf tool, as opposed to monitoring

each process or thread separately or per-CPU. Count hardware events such as instructions executed, cache-misses, suffered, or branches mis-predicted.

## 11. Freezer

Allows to freeze (suspend) / throw (resume) a process group. Processes in the cgroup are unaware of the freeze time. It is more frequently used for scenarios which require batch scheduling, migration etc.

## 12. Debug

Debug makes a number of internal details of individual groups, or of the cgroup system as a whole, visible via virtual files within the cgroup file system. Some data structures and the settings of some internal flags.

## 3.2 Namespaces

Now that we have understood how resource control and accounting support is provided by the Linux kernel to a group of processes. Lets move on to isolation. Linux Namespaces [12] provides process groups their own isolated system view. Namespaces exist for different types of resources as in the case of cgroups. Each process is in one namespace of each type. There were 10 namespaces proposed initially out of which 6 have been added to the current Linux kernel. These namespaces may or not have their own hierarchy similar to that of cgroups.

### 3.2.1 Changes to Kernel

Two new structs added and referred from the task\_struct of every process.

1. `nsproxy` which in-turn refer to 5 namespace structs
2. `cred` user namespace pointer

Three Syscalls added/modified for supporting namespaces,

1. `unshare()` - Creates new namespace and attaches a process
2. `setns()` - Attaches a process to an existing namespace
3. `clone()` - is modified which in turn modifies `fork()` and `exit()`

6 New flags were added to `clone()` system call, one for each namespace. When any of the flag is set, the process gets created in new namespace of the corresponding flag. An inode is created for created for each namespace created. To view details of namespace of a process,

```
sudo ls -al /proc/<pid>/ns
```

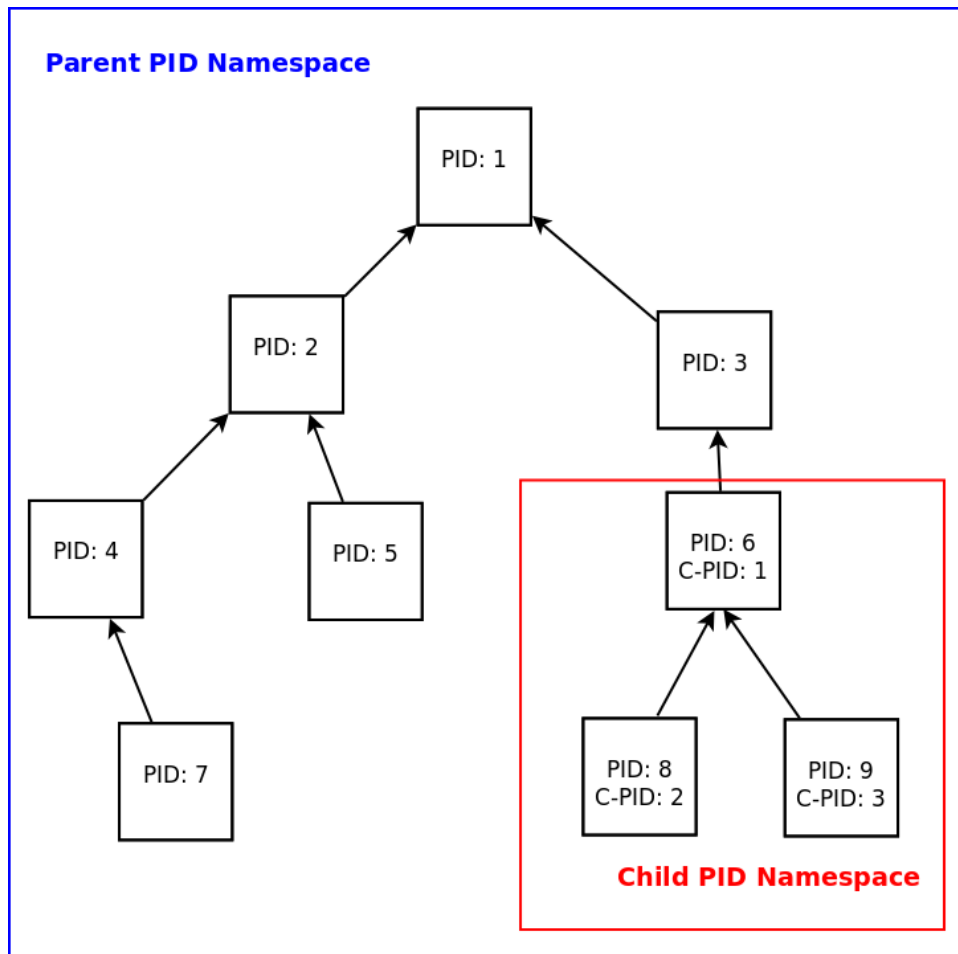


Figure 3.2: Example for PID namespace in Linux

### 3.2.2 Six existing namespaces

The following section describes the 6 namespaces existing currently. It is to be remembered that new namespaces are likely be added in the near future to support higher levels of isolations.

#### 1. PID Namespace

PID namespace exists to isolate process views existing in a process group. Processes within a PID namespace only see processes in the same namespace. Each namespace has its own isolated numbering. PID numbers may overlap across different PID namespace. Each level of process group gives a different PID to a process. If PID-1 of a process group is killed the namespace is removed along with all its processes.

As seen from the Fig:3.2, PID namespaces follow a hierarchy similar to that of cgroups. The Fig, demonstrates how processes with PIDs 1,2,3 in the child namespace internally maps to 6,8,9 in the parent namespace.

## 2. Network Namespace

Each network namespace gets its own networking resources like - network interfaces (including loopback), routing table, IP table routes, sockets etc. You can move a network resource across network namespace. Typically, a virtual network interface is used to network with host. Network namespace is the only namespace that can exist by itself without any processes attached to it. All other namespaces need to attach a process on creation.

## 3. Mount Namespace

Mount Namespaces was the first namespace to be introduced even before the proposal for namespaces were created. It can isolate the set of file system mount points seen by a process group. It creates environments that are similar to chroot jails, but are more secure. Can be set up in a master-slave relationship. They can have private/shared mounts. Mount namespace as a peculiar property compared to other namespaces and that is, on passing a new mount namespace flag, a new mount namespace is created by copying the existing mount points. Any new mount created in the new namespace is only local to the current namespace and is not visible in the parent namespace.

## 4. UTS Namespace

Lets process groups have its own host-name and NIS domain name. Isolates two system identifiers node-name and domain-name. The names traditionally are set using the `sethostname()` and `setdomainname()` system calls. This can be useful to initialize and configure scripts that perform actions based on the two above mentioned parameters.

## 5. IPC Namespace

Allows a process groups to have its own - IPC Semaphores, IPC message queues and IPC shared memory. This is a very useful feature to make processes present in a namespace to only be able to communicate with other processes in the same namespace.

## 6. User Namespace

It is the newest namespace which was introduced in kernel version 3.8. Using user namespace you can be root inside a process group, but once you move out a process group, your just a normal unprivileged user. With the introduction of this new namespace an unprivileged process has root privileges in its own namespace and hence have access to functionality that was earlier only limited to root user. User namespace is relatively a new one, and not many container providers have still incorporated this feature.

## 3.3 Disk images

The final component that constitutes a container is a disk image. The disk image gives a ROOTFS, which contains the list of files and directory structure containing the root mount for a container. It

typically has the structure of the root directory found in an Linux machine with lesser files. This disk image is usually smaller than the typical OS-disk image because it doesn't contain the kernel as it shares the kernel with the host machine.

A disk image could also contain only your application and make use of the host ROOTFS hence could even be as small as a few kilo bytes. These disk images could also be designed in such a way that multiple containers could share a read-only disk image using COW (copy-on-write) techniques. The exact Implementation details of disk images in context of containers would be described in chapter: 5.

### 3.4 Present day virtualized containers

Container makes use of the three above mentioned components and combine them to form the building structure. Container is a set of processes grouped together along with its dependent resources into a single logical OS entity. It enables multiple isolated user-space instances on a host machine. This is achieved by sharing the OS kernel among the different containers running on the host machine and hence is also referred to as OS-Virtualization. It is to be remembered that by sharing the host kernel with the container, we are actually exposing the host system call table to the container.

Control groups are used to attain resource control i.e tracking resource usage and imposing quantitative limits. Typically all physical resources can be controlled using cgroups however, controlling network bandwidth allotted to a container is not currently supported by any of the container managements systems. There are few articles over the Internet which mention work around for this, but this has to be done from outside the container and at a kernel level.

Kernel namespaces are used to limit the visibility and reach of processes to only resources provisioned to its corresponding container. Typically, for every container a new mount namespace is initialized and on this new mount namespace a new ROOTFS is mounted on to the root. Of course this is a typical usage, but however there might be cases where mount points are shared with the host system or other containers and so on.

Containers provide close to near native overheads and hence applications working using containers are called cloud-native applications. Having said this containers have their own set of drawbacks which would be discussed later. Containers are not a part of Linux kernel, but only cgroups and namespaces are. Containers are usually designed by a vendor own makes use of the above kernel features to implement the required basic functionality and adds their own proprietary features on top of it and release them as a commercial product/projects and such products/projects are referred to as Container Managers. Containers are classified into two types - system and application containers which have been described below.

### 3.5 System Containers

Containers whose virtual environment provides is similar to a native machine is often referred to as system or full containers. You can install, configure and run different applications, libraries, demons, etc., just as you would on any OS. It is more useful to someone like a cloud PAAS service provider. Such containers have been used by cloud providers for a while now. Examples of system containers are

- Linux containers (lxc)

- Parallels Virtuizmo
- Solaris Zones

### 3.5.1 Linux Containers (lxc)

lxc provides an userspace API to deploy containers using the underlying kernel features. It helps in easy creation and management of system containers. lxc provides wide range of APIs for a user to deploy containers few of which are - Go, ruby, python, Haskell. lxc is a free software and is licensed under GNU. lxc was first launched in the year 2008. Recently lxc has come up with its latest stable release of 2.0.

lxc uses cgroups to limit resource limits. A basic configuration file is generated at container creation time with default keys coming from the default.conf file. The configuration file can specify various cgroup parameters as options. These configurations are then mapped to their corresponding cgroup subsystem. For example setting the memory limit at 2GB would indicate creating a new cgroup in memory subsystem with a limit of 2GB and attaching the processes of the container to this cgroup. Similarly other resources of a container would be mapped to a new node in each subsystem.

Isolation is mapped by specifying `CLONE_NEWIPC`, `CLONE_NEWNET`, `CLONE_NEWPID`, `CLONE_NEWUSER`, `CLONE_NEWUTS`, `CLONE_NEWNS` (`mount`) flags to a `clone()` system call. This uses the previously described namespaces concept to keep the container in a new namespace. lxc is one of the few new container version which support user namespaces currently where an user can be mapped to root inside a container but is just a normal user outside it. Further more the a new ROOTFS is mounted using a base container image. The base container image is fetched from an online repository for the first time and is then after is fetched from the local cache. Each time a new copy of the ROOTFS is made on which a new container makes its changes. Running an existing container makes use of its previously used ROOTFS to execute.

## 3.6 Application Containers

Application containers on the other hand provide a more similar execution environment. They are used as an virtual environment for application development and testing. It allows minimalistic configurations with no demon processes. Application containers have gained popularity over the last few years. It is often used by devops to develop new applications, build, test and finally ship the application. It is often recommended by application container managers to deploy only one application per container and deploy multiple applications by creating one container for each application. Examples of application containers are,

- Docker
- Rocket

### 3.6.1 Docker

Docker is an application container manager. It is an open platform primarily used for developing, shipping and running applications. Application developers can write code locally and share their development stack with their team. When the application is ready, they push the same stack into a test



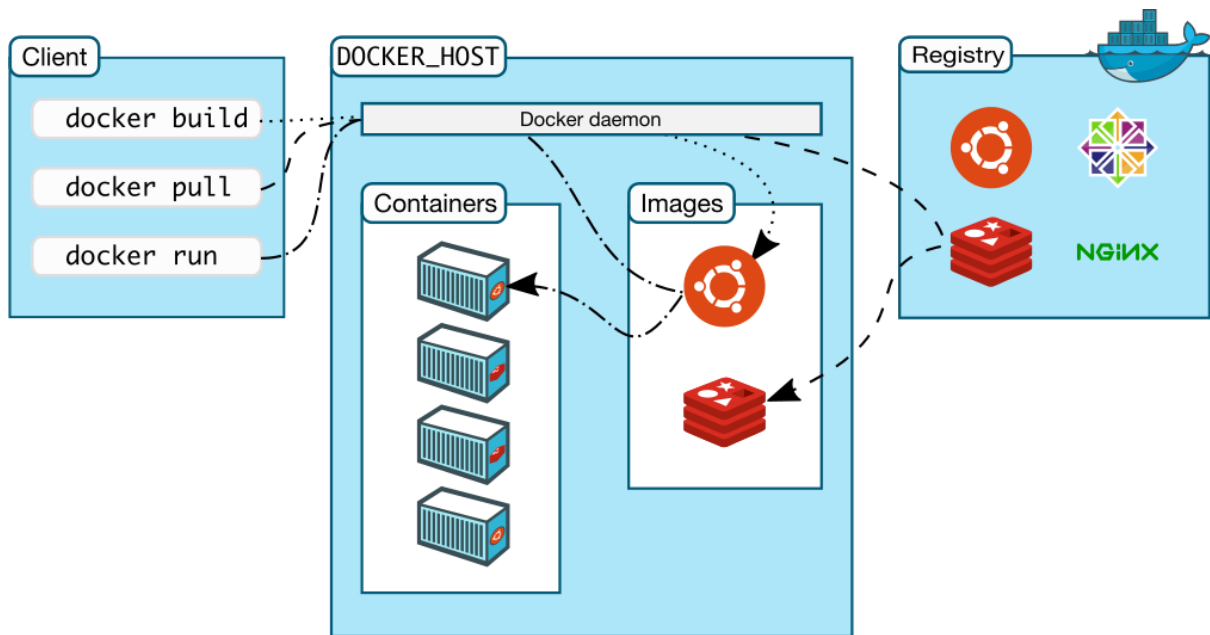


Figure 3.3: Docker Architecture, source:[1]

environment where the application is further tested. From here, you can push the code into deployment. It can run on either a local host machine, physical or virtual machines in a data center, or on a cloud.

Docker similar to lxc uses cgroups to limit resource limits. A basic configuration file is generated at container creation time with default keys coming from the default.conf file. The configuration file can specify various cgroup parameters as options. The mapping is similarly to how resources of a container was mapped to a new node in each subsystem as described for lxc.

Isolation is mapped by specifying `CLONE_NEWIPC`, `CLONE_NEWNET`, `CLONE_NEWPID`, `CLONE_NEWUTS`, `CLONE_NEWNS` (mount) flags to a `clone()` system call. This uses the previously described namespaces concept to keep the container in a new namespace. Unlike lxc docker doesn't yet have the support for user namespace to run a container in a local root privilege. But this support is expected to come out soon in the newer docker versions. Disk images are more optimized by Docker and these optimizations are described in the upcoming sections.

## Docker Architecture

The Docker architecture is illustrated in Fig:3.3 and shows how it forms a client-server architecture between client and host. They can both be on the same machine or even on different machines. Communication between client-host-registry happens using RESTful API. Components of Docker architecture are,

- Client: User interface to Docker using commands
- Images: Read-only template, used to create Docker containers
- Daemon: Building, running, and distributing Docker containers
- Registry(hub): public or private(paid) stores from which you upload or download images
- Container: Created from a Docker image. Can be run, started, stopped, moved, and deleted

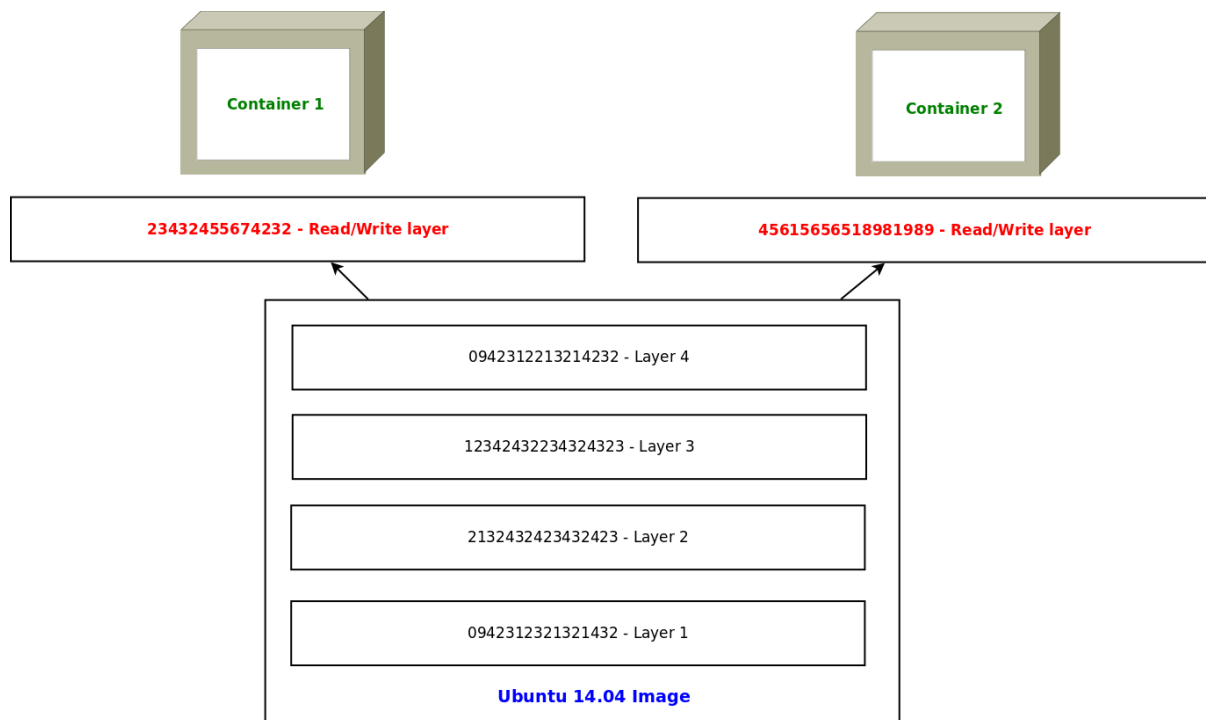


Figure 3.4: Disk layers in Docker

## Docker Images

Two key technologies behind Docker image and container management are stackable image layers and copy-on-write (CoW). They make use of an overlay FS like AUFS to combine layers into a single image. We would refer to it as COW. The main advantage of using COW mechanism is to reduce storage space. This is where Docker leads over other container technologies. COW disk images have shown great potentials in lowering the amount of storage space required but this comes at a cost of minimal performance overhead.

Each Docker image references a list of read-only layers that represent filesystem differences. Images are read-only templates. Each image consists of a series of layers. These layers are content addressable since Docker version 1.10 and hence can be reused by other images which use the same.

There are again two kinds of images - one is a base image and another is an image created from a base image. Consider running a container from an ubuntu image which is directly fetched from the Docker registry, now this image is a base image. Now, given a base image the user can add layers over it like,

- Run a command
- Add a file or directory
- Create an environment variable.
- What process to run when launching a container from this image

And such instructions above are stored in a file called the `DOCKERFILE`. While building an image using a `DOCKERFILE`, it adds a new layer on top of existing base image for every instruction to give a final image with several layers as shown in Fig:3.4 as the layers 2 to 4 within the ubuntu 14.04 image.

## Docker Containers

When you create a container using Docker, the Docker client pulls the requested image (if doesn't exist locally, pulls from registry). Creates new container using this image. The container adds a Read-Write (R/W) layer on the image on top of the read-only image layers on which all the changes to the container are stored, thus changes made to container doesn't affect its image. Hence multiple containers can use the same image and by only adding one additional R/W layer for each container as shown in Fig:3.4. On deleting a container only this R/W layer is deleted.

## Data volumes

Lastly, Docker volumes could be used for multiple containers to share common data by mounting a shared mount point onto multiple containers. This can be used to share data to process among containers.

## 3.7 Summary

This chapter introduced the various components that make up a container. It also discussed how these components interact with each other to give us the real container. It also described the different types of containers and concluded with a discussion on docker containers. To get a hands on experience on cgroups, namespaces and containers refer to the Appendix:[A](#) .

## 4. Comparing Containers against Virtual Machines

Virtual machines have been used extensively by cloud providers today. Their superior isolation and heavily customizable resource configurations make them a prime candidate to be used by cloud providers. They have been used by cloud providers for over a 15 years now. One major drawback of virtual machines is the performance overheads introduced by them. These overheads are caused by the additional layer of hardware emulations and dual control loop (one control in the VM and the other at the hypervisor). The performance overheads incurred by virtual machines affects customers drastically while calculating the price/performance costs as providers incur higher performance costs and they in turn over price customers to make up for this.

Containers are relatively new and their potential to be used in cloud providers has been brought about only in the past few years. Containers provide similar features to that of a virtual machine except for the fact that they are claimed to be faster in terms of performance overheads. If this claim is true, they have potential to replace the existing virtual machines. To support this claim, it would be interesting to run experiments to investigate this claim. Hence this section consists of a detailed performance comparison between virtual machines and containers.

Performance metrics were **compared along the axis of Memory size and bandwidth, CPU utilization, Network throughput and latency, Disk I/O throughput, Startup latency and few other application specific metrics**. This understanding could be then extended to use case scenarios for using Virtual Machines (VMs) and containers which could potentially be suitable instead existing native machines. In order get an overall picture about the same, several case studies which compare the above two virtualization techniques were selected.

Most case studies took one example from each of the two virtualization techniques and compared its performance to the native execution. This is with the assumption that other hypervisors use similar acceleration features and other container technologies use the same underlying OS support and hence both of the examples taken are likely to represent a generic comparison between the two virtualization technologies. They compare different scenarios by performing the same experiment with native, VM and container environments. The following sections provides a resource wise comparison and a few application specific comparisons.

Each case study selected focuses the comparison from a different perspective. The different performance metrics based on which different studies focused on have been described in Table:4.1. A list of the different benchmarks used in the cases studies have been attached to the Appendix:B

Table 4.1: Performance metrics based on which different papers made comparisons

Paper	Focus of paper	Memory throughput	Memory size	CPU utilization	Disk I/O throughput	Network latency	Network throughput	Startup latency
[4]	Price/ performance to the customers of a cloud service provider	✓	✗	✓	✓	✓	✓	✗
[13]	Overall comparison based on physical resources	✓	✗	✓	✓	✗	✓	✗
[3]	HPC (High performance computing)	✗	✗	✓	✗	✓	✓	✗
[14]	Use as virtual routers	✗	✗	✓	✗	✓	✓	✗
[2]	Density and start-up latency	✗	✓	✓	✓	✗	✗	✓

## 4.1 Metric based comparisons

This section compares the performance metrics of containers to virtual machine along the axis of Memory size and bandwidth, CPU utilization, Network throughput and latency, Disk I/O throughput, Startup latency and few other application specific metrics. Inferences were made and a relative ordering in terms of the various performance metrics taken was established.

### 4.1.1 Memory Bandwidth and Size

Memory itself can be compared from different perspectives. But considering our use-case we focus on containers for deployment in real work use case scenarios we focus on two main areas, namely - **Bandwidth and Size**. It is to be noted that experiments we performed using an hypervisor (KVM with KSM enabled) that de-duplicated anonymous pages between VMs.

Bandwidth refers to the data transfer rate achieved in copying data from main memory on a cache miss. Bandwidth is measured using throughput(B/s). It is made sure creating working set size significantly larger than cache size. Most experiments made sure that working set size was atleast 4x of that of the cache size. Bandwidth itself can be again categorized into bandwidth performance on regular access and random access. Regular access is when data accessed are sequential and random is when its randomly accessed. Regular access generally is more faster due to hardware pre-fetching of upcoming instructions. Memory experiments were studied which involved both these kinds of experiments.

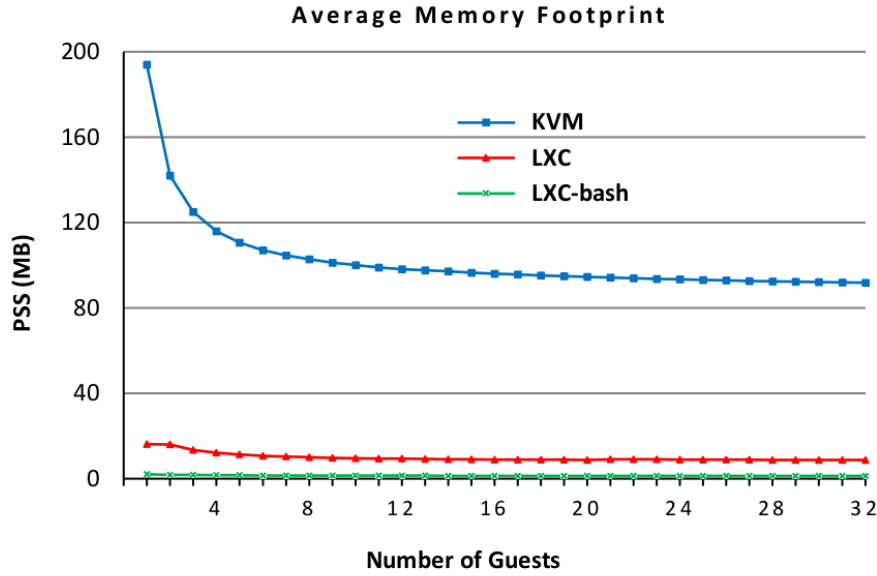


Figure 4.1: Average Memory Footprint (proportional set size) of VMs with KSM enabled versus system (LXC) versus application containers (LXC-Bash). Lower the better, source:[2]

Memory size refers to memory occupied by each of the virtualization technologies. It is also referred to as memory footprint. It is measured in RSS (Resident set size) pages. Size was studied by taking two physical machines with the same configuration and increasing the number of guest VMs (Refers to container instance when using a container manager and a virtual machine while using a hypervisor) and observing the behavior of the physical machine while doing the same.

## Inferences

Table 4.2: Relative ordering in terms of Memory

Test	App Container	System Container	Virtual Machine
B/W on Regular accesses	1	-NA-	1*
B/W on Random accesses	1	-NA-	1
Memory Size	1	2	3

It was observed that memory bandwidth has nearly the same overhead in both the environments on both regular and random accesses. The biggest advantage of containers over VMs is the memory size. The sharing of OS kernel among guests is the reason for this small size. It is to be noted that **application containers have half of the memory size of that of a system container, and system containers have about 11x size smaller memory size when compared to VMs** which is quite evident from Fig:4.1.

## Optimizations

- De-duplication can be extended to file-backed pages, between other virtual machines and Host.

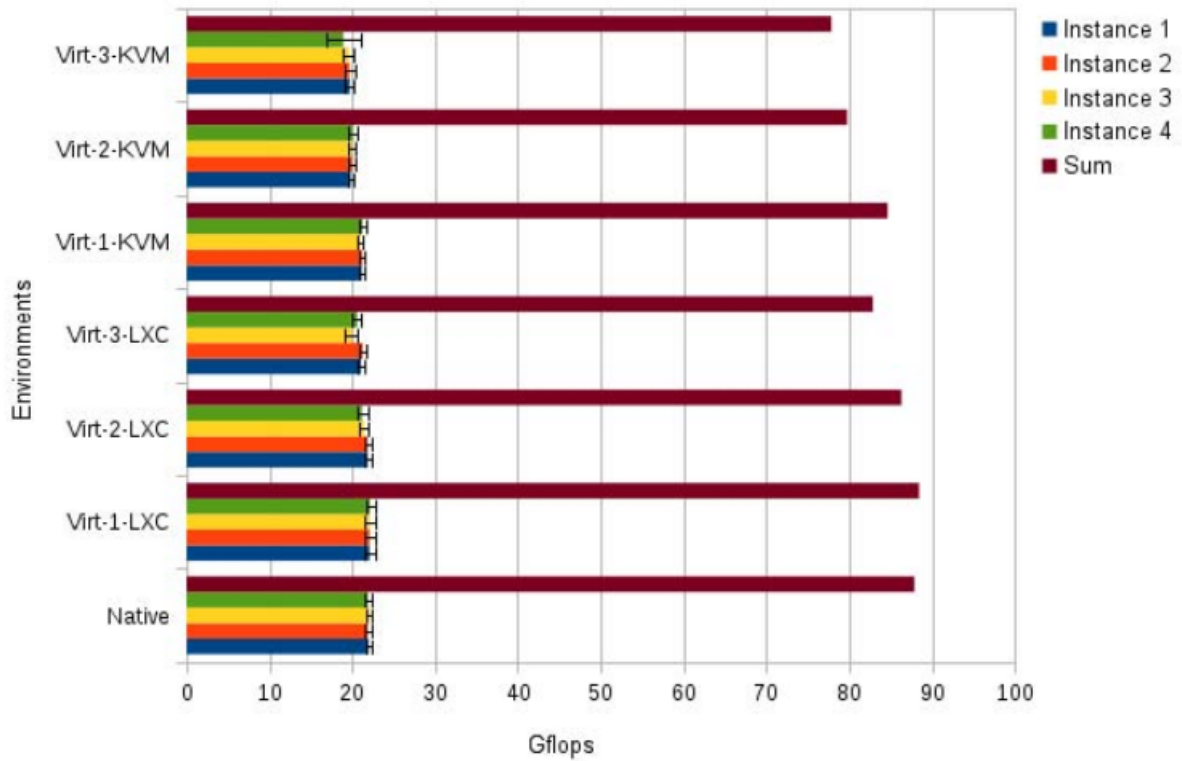


Figure 4.2: CPU utilization in HPC environments between KVM and LXC, source:[3]

- Unwanted devices emulation to be avoided by device emulators and save memory footprint in a virtual machine.

#### 4.1.2 CPU Utilization

The dual control loop is a well known problem in virtual machines where the processes in the virtual machines have to be scheduled by two layers - one at the guest machine and the other at the host. Different use cases are used to put to test of the effect of this compared to the CPU scheduling used in containers.

CPU performance can be testing the CPU on a **single threaded, multi-core and HPC environments**. One of the commonly methods used to test CPU efficiency is by performing data compression on a system and testing its throughput (B/s) achieved while doing the same. From a cloud provider's perspective, CPU performance can also be compared by increasing the guest VM per physical machine and observing the change in CPU utilization.

#### Inferences

CPU utilization in native and containers are nearly the same. **VMs on the other hand introduce a slight overhead (22%)** and hence optimizations like CPU pinning, exposing cache topology could be used to reduce this overhead (17%). Over the evolution of VM technology this gap seems to have lessened and soon might even disappear. Scaling up of guest machines shows negligible overheads in CPU execution in both the cases.

Fig 4.2 illustrates how containers(lxc) perform better than virtual machines (KVM) in HPC environments on CPU utilization. The number of guests increased per host is denoted by Virt:1-3 xxx with 3 being the highest number of guests. As the number of guest VMs increase in both the environments, containers perform better than virtual machines.

Table 4.3: Relative ordering in terms of CPU

Test	App Container	System Container	Virtual Machine
Single threaded	1	1	1
Nop loop	1	1	1
Multi-core processing	1	1	2
Change in number of Guests	1	-NA-	2
Data compression	1	1	2
HPC environment	1	2	3

## Optimizations

The following optimizations in VMs helps in bettering its performance, if not for making it as good as native.

- vCPU pinning in virtual machines
- exposing cache topology reduces CPU overheads in VMs and becomes relative less slower to native

### 4.1.3 Network Throughput and Latency

The two main metrics which are used to compare network performance on any system are **Throughput and Latency**. With this in mind, the comparison between virtual machines and containers can be compared in the same axes. Containers by default handled traffic using NAT. One of the typical ways used to test a system for network performance is by measuring the CPU cycles required to transmit one Byte of data. Benchmarks were used to obtain the network metrics for the two environments. The benchmarks typically involved a client-server architecture where the execution environment of both of them were the different virtualization technologies and metrics were noted.

HPC environment was also put to test by first testing the intra physical system communication and then later testing the inter physical system communication between guest machines.

Lastly, the performance on using a physical system as **virtual router used to route packets** was put to test using both the virtualization technologies and the hardware support provided to virtualization by hardware vendors with techniques like SRIOV. Tests were performed using single and multiple v-routers on a host machine and test its performance to route traffic. The number of traffic connections were varied to track the performance behavior of the virtual router.

## Optimizations

The overhead caused by NAT in containers (Docker) can be eliminated by deploying containers in the host network namespace.



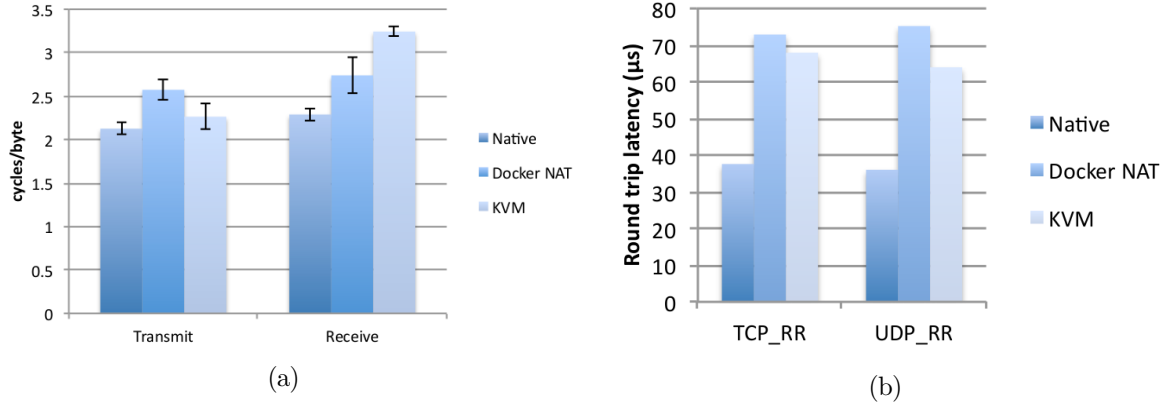


Figure 4.3: source:[4] (a) Network transmission efficiency between native, Docker and KVM(b) Network latency comparison between native, Docker and KVM

## Inferences

Network bandwidth utilization and latency is best in native (containers using host networking stack gave similar results in other case studies) and worst in containers using NAT as shown in Fig: 4.3b and Fig:4.3a respectively. To optimize this, it is advised to use host networking stack for the container which gives near native performance. Network latency while using TCP in both cases performed better than while using UDP. **Containers with NAT give an overhead of 100% and with host provides near native performance and KVM sits between the two by introducing a 80% overhead.**

A system as a V-router using SRIOV is better handled by hypervisors as they execute in user-space and are able to attain better efficiency due to lesser dropped packets when compared to processing at the kernel layer by containers. To conclude, containers with host networking stack provide near native performance.

Table 4.4: Relative ordering in terms of Network

Test	Container (host stack)	Container (NAT)	Virtual Machine
Bandwidth Utilization	1	3	2
Latency	1	3	2
Node as single V-router	1	-NA-	1*
Node as multiple V-router	1	-NA-	2
HPC environment	1	-NA-	2

### 4.1.4 Disk I/O Throughput

Disk I/O is typically performed to check **throughput on two dimensions - sequential and random access**. The different case studies ran such performance benchmarks on both dimensions to obtain throughputs. The test setup made use of a 20GB SSD drive and accessed a large file of 16GB with typical requests of 1MB and the disk block size of 4kB. Disk I/O was performed using open, read, write and mixture of the three. Performance changes were also noted by increasing the guest VM density on a system. One of tests was to set up a Filebench benchmark in one guest and while loop with integer increment in other guests.

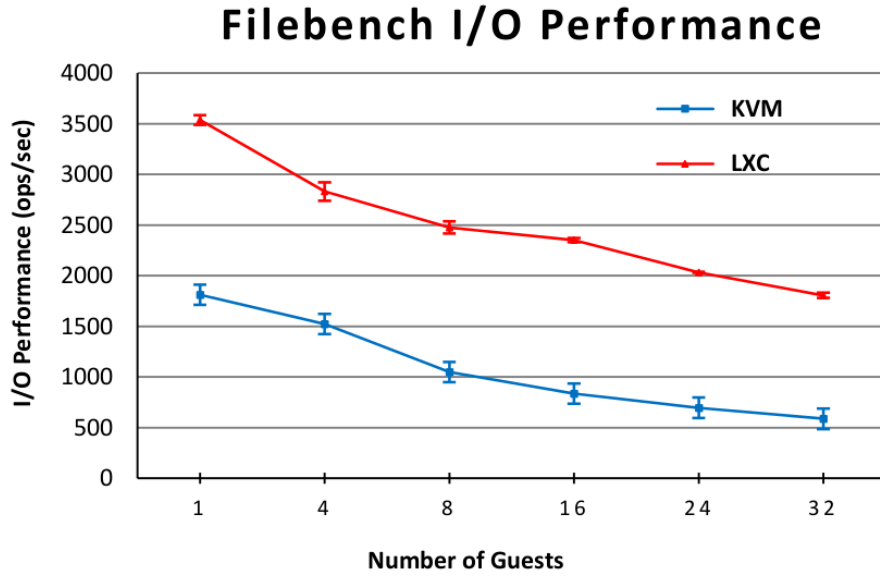


Figure 4.4: Filebench I/O performance as number of guests increases on a 4-core machine. Higher is better , source:[2]

## Inferences

Table 4.5: Relative ordering in terms of Disk I/O

Test	Container	Virtual Machines
Sequential Access	1	1
Random Access	1	2
Scalability	1	2

Disk I/Os have always shown containers performing better than VMs specially in random accesses, and this can be accounted due to the overhead causes by device emulations and requests passing through an additional layer. Sequential accesses on block I/Os produces nearly the same throughput in both, but **random accesses incur significant (2x) overhead in the case of VMs** which is caused due to the above mentioned problem.

Scaling of guests in both cases have a near-constant rate of performance degradation with the primary premise of containers performing better holding strongly as shown in Fig:4.4. It is to be noted that block I/Os have showed anomalies when benchmarked using different workbenches on application and system containers and hence solid implications are difficult to be drawn.

## Optimizations

I/O pass through, where device is directly mapped to VM. Recent studies also suggest I/O drivers to be pushed to application layer. The use of filesystems like AUFS (cow) in container technologies like Docker saves storage space at the cost of noticeable performance overheads of about 20% more than native.

#### 4.1.5 Throughput and Latency for MySQL server & Redis key-value store

The previous four axes on which performance was compared was based on physical resources available to any system. This comparison is more of running macro benchmarks on a system to test its performance in a real work application. The two tests were performed to test the same.

The first test involved testing the performance of MySQL in both the virtualization environments. This was done using a SysBench. The transactional throughput and latency were noted to make inferences. The second test involved performance in running key-values stores which are extensively used by cloud providers to cache client sessions. This is especially used by PAAS (Platform as a service) providers. This experiment involved scaling the number of connected clients to the system until the CPU saturated. An equal mix of read and write requests were simulated.

##### Inferences

Table 4.6: Relative ordering in terms of Application Performance Metrics

Test	Container (host stack)	Virtual Machines
MySQL server	1	2
Key-value stores	1	2

As discussed in the networking section, containers using host networking stack provide better performance and this argument is support in the two experiments above. Containers performed near native with host networking stack where as **virtual machines offered an overhead close to 40% compared to native for MySQL server. Redis server only adds a slight overhead for VMs.**

#### 4.1.6 Startup Latency

Startup latency is a very important metric for a cloud provider as on receiving a burst traffic the provider should be able to launch instances of guest instantaneously. The lesser the startup latency is, the lesser instances of machines the cloud provider will have to be running on stand by. This is because lesser running instances would cost lesser power costs to the provider. This experiment involved noting the startup times of both the technologies by booting a similar disk image application.

##### Inferences

Table 4.7: Relative ordering in terms of Startup Latency

Application Container	System Container	Virtual Machines
1	2	3

Start-up latency is one of the crucial reasons why cloud providers move to containers but studies [2] propose checkpoint/restoring VM optimization which brings down start-up latency by a large factor

and must be kept in mind while deploying cloud services. **VM typically takes about 50-100x (50 for system, 100 for application) the time required to startup a container.**

## Optimizations

Checkpoint/Restoring VM optimizes this to bring it down to a factor of 6-12x the time required to start up a container.

## 4.2 Summary

To summarize the interferences; **Memory size, Startup latency, Disk I/O throughput, and Network I/O throughput (using host network stack)** are performance metrics where containers out weigh virtual machines in terms of performance. This conclusion is purely based on performance and it has to be remember that virtual machines allow you to potentially run any OS on top of the host whereas containers only allow you to run a guest OS of the host kernel as the kernel is shared. Virtual machines also provides better security isolation.

Container technologies are relatively new, and are just now close to stabilization. There are several factors why no research might still give the correct overall picture of the overall comparison, and these factors have been discussed below.

Firstly, lack of well-formed scientific studies on how much performance benefits containers actually offer in exchange for the qualitative differences. Many of the studies taken are dated and major changes have been introduced into the virtualization technologies in the recent past. Many of the researches are narrowly focused as shown in Table:4.1 and hence aren't able to pitch in an overall idea of the comparison. Many researches have even forgotten to look into optimizations before concluding results. Few of the experimental observations described earlier might be counter intuitive due to the above mentioned problems in the existing research in containers.

To partially overcome the above problems, many researches were skimmed and a handful of them were read for detailed analysis. Ideas from different researches which overlapped each other have to be considered to draw stronger implications in the conclusions drawn above, but are still likely to be prone to errors and this has to be taken into consideration.

## 5. Optimizations for Startup Latency and Disk I/O

There are a couple of drawbacks in containers when it comes to start up Latency and Disk I/O throughput. The following sections brings them out and proposes optimizations for them.

### 5.1 Problem of Startup Latency

Disk images are handled differently by different container managers. lxc for instance just makes multiple copies of each ROOTFS that is one for each container where as Docker tries to optimize this by doing a COW at a file layer level as described in the previous sections. Although the below proposed problem of startup latency is generic, for the purpose of this discussion we focus on Docker which is hugely used by organizations to manage their applications.

Since containers virtualize at an OS-level, we expect containers to act just like processes in terms of start up time. But this isn't the case, in a real world scenario because most organizations which make use of containers typically store their disk images in a central repository which could be local to their network or

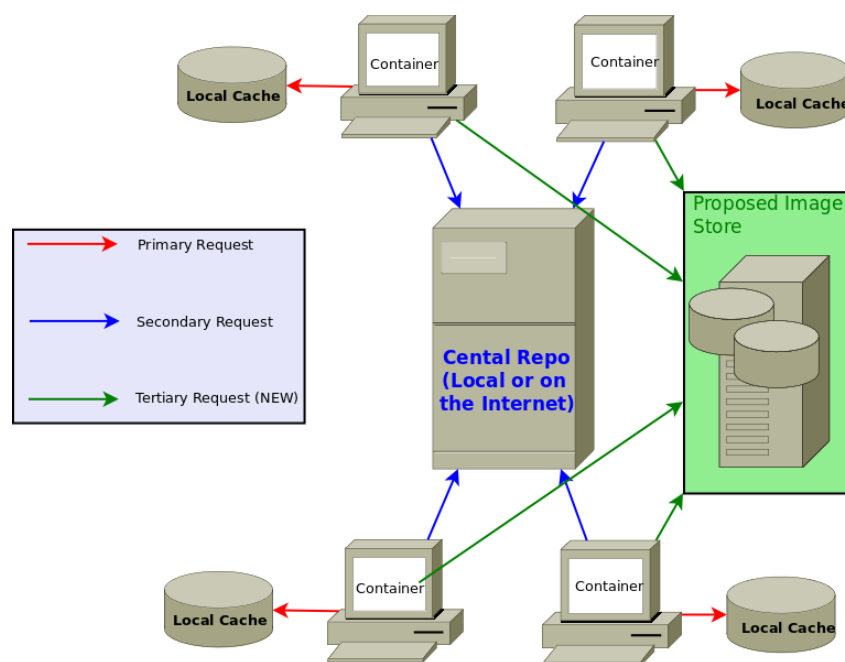


Figure 5.1: Typical container architecture used by organizations to handle disk Images

even over the Internet as shown in Fig:5.1. This is because, organizations typically have a large compute farm and any container can be run on any machine. Now when a container starts up it typically looks for a copy of the disk image in its local cache on failing to find, it looks for the images in its centralized repository. On failure to find the disk image on the local cache causes time delay for fetching the images account for more than 76% of its start up time, bring the start time of a typical application container close to 25 seconds which includes the time to fetch the image and setup the container before running applications. It was also observed that only about 6% of the image data pulled was actually used for starting the image.

Further specific to Docker, Docker compresses images using gzip compression of individual images layers to transfer them over the network. This heavily impacts the startup overhead. It is also noted that Docker does a file level de-duplication of the same image layers of different containers which reuse layers, but previous works have proven that Block level de-duplication achieves better compression rates than file level.

### 5.1.1 Proposed Optimizations

Research done by Harter, Tyler, Salmon and few others [15] proposes a solution to the above mentioned problems. They propose a new storage driver for Docker called Slacker. The essential idea behind slacker is that it stores docker images as flat images at a central image store (VMWare Tintri) using the NFS protocol. The de-duplication now occurs at the central image store at a block level and blocks are fetched on demand instead of dockers previous approach of pulling it all at once. This store is complementary to the existing Docker registry. The store now contains the actual image and the registry only stores the image snapshot ID used to fetch images. This approach of block level de-duplication at server allows the system to fetch blocks on demand and eliminates the need to compress before pushing or pulling which is used in the default docker storage driver.

Slacker is partially backward compatible with non slacker managers. When a new image is to be fetched by a container it first asks for the snapshot ID to the Docker registry and this snapshot ID returned (in gzip format) is used to send requests to the image store as and when data is required. This technique creates a small problem when it comes to caching at the local host as multiple images using the same base image layer no longer share the cache, to overcome this the research suggests kernel modifications using a loop back driver to implement caching in host at a block level complementary to the existing file level de-duplication. Slacker shows median speed ups of about 5-20x in startup of containers, although it brings an runtime overhead of about 15%. The speed up of pushing images are even higher.

## 5.2 Problem of Disk I/O Throughput

With advancement in non volatile storage technologies like PCM (Phase Change Memory) based SSDs, storage devices today offer very low latency and high degree of application parallelism. In context of containers, earlier containers were limited by the bandwidth offered by storage devices but now the number of containers that can be run on a physical machine maybe limited by the bottlenecks in I/O stack due to contention among several containers. Furthermore the I/O stack scales badly in a multi-core environment.

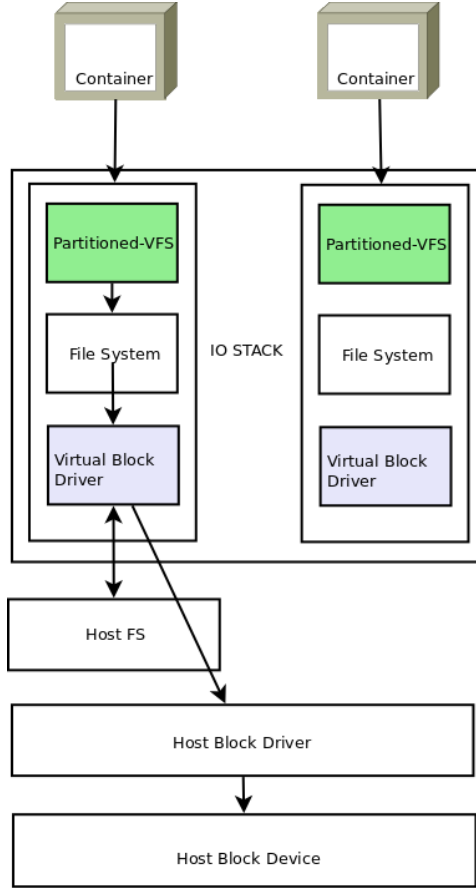


Figure 5.2: Mutlilanes Architecture to handle a single Host with multiple containers with isolated I/O stacks for each container

### 5.2.1 Proposed Optimizations

Works done by Kang, Junbin, Zhang and few others [16] proposes two things - A new virtualized block device and a partitioned VFS per container as shown in Fig:5.2.

The virtualized block device maps each container to a regular file in the underlying host device. It also has a block device driver attached to it which has two main tasks namely - block translation and request handling. The block translator tries to map each incoming request from the container into the host device block with the help of the thread that is used to fetch mappings from virtual blocks to actual blocks using the host file system mappings. All block translations are to be done on dedicated cores to increase core specific cache hit rates. Once this virtual to physical mapping is obtained each request is divided into slices as contiguous requests on container device may not be the same on host device and hence are divided into slices which are contiguous. Proper care is to be taken while returning the sliced data to host in consistent manner. The partitioned VFS creates new copies of kernel data structures like locks which are local to a container file system. This way there is no overhead in contention for locks across containers.

The proposed solution outperformed the modified version of the system when compared to native container system by about 11.32 - 11.75x. Although this optimization is very effective it might not be the best solutions for situations where containers share data.

## 6. Applications of Containers

Containers have been used widely these days in several domains like PAAS (Platform as a Service) providers to provide container environment to customers using the cloud, HPC (High performance computing) clusters to solve complex computational problems, Large corporates to manage the various different processes executing on their servers etc. Even a lot of vendors are trying to do away with Virtual Machines and are trying to adopt the lighter weighted containers. With all this in our mind, we focus this section on discussing how large corporates like Google and Facebook use application containers to manage their servers.

### 6.1 Google Borg

Google had a wide range of applications/processes that had to be run on servers with different constraints and requirements for various purposes. Application were being run at their respective servers with complete isolation from one another there by under-utilizing the resources allocated to their corresponding physical servers and ultimately leading to higher operational expenses. Physical systems at disposal were heterogeneous in their resources, dimensions & properties. At that time, there were several management systems adopted by other organizations which were used to manage different kinds applications to run on a cluster of systems within their organization but on a smaller scale. This lead to the initial motivation for developing a similar system which operates a larger scale for managing their applications.

The solution developed to the above problem by Google was called Borg [5]. Borg organizes machines into cells to which jobs (further divided into tasks) are submitted. The workload consists of variety of services which could be broadly classified into Production (latency-sensitive) or Non-Production jobs (Batch-jobs with no strict deadline). Production jobs are allotted into machines to provide up-to 90%ile resource utilization after which the under utilized resources are assigned to Non-Production jobs (which maybe reclaimed if necessary) thereby utilizing resources efficiently. Tasks are executed on machines typically using Linux Containers or if required by Virtual Machines (VMs). Centralized entity called Borgmaster co-ordinates with De-centralized entities called Borglets in managing jobs/tasks assigned to the cell. Borgmaster is further supported by a Scheduler to schedule tasks onto machines

#### 6.1.1 System Outline

Borg organizes machines into a large ( 10k) cluster called cells which lives inside a single building. Cells receive jobs to be processed from users (app developers) which is assigned attributes like - name, owner, number of tasks, constraints (soft/hard) etc. Every job is further divided into tasks which can have further properties. Tasks which maps to a set of processes running on mostly Linux Containers or rarely



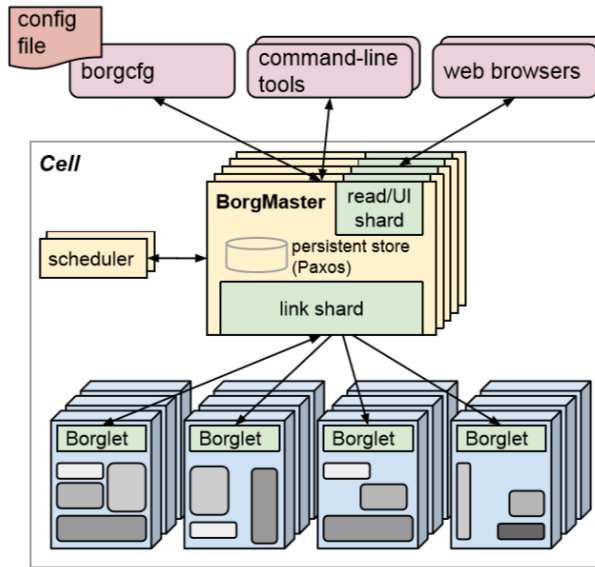


Figure 6.1: Architecture of Google Borg, source: [5]

on Virtual machines(VMs). Alloc is a set of reserved resources on a machine to one or more tasks. Every job has a priority and quota (vector representing resource requirements). Borg Name Services (BNS) is used to identify individual tasks.

### 6.1.2 Features

Borg makes use of a web-based UI service through which users can keep track of their jobs called Sigma. A single cell typically handles up-to 10k systems and handles jobs effectively using mechanisms like queuing, scheduling, restarting etc. HA is implemented using techniques like automatic rescheduling of evicted tasks, rate-limits, task spreading, idempotent operations, data recovery using local disks, multiple borgmaster instances etc.

### 6.1.3 Architecture

There are four components that make up the Borg architecture as shown in Fig:6.1 which have been described below,

- **Borgmaster:** A single logical process (replicated 5 times) per cell that is used to manage and maintain states of jobs submitted by users, Borgmaster checkpoints its states to Paxos stores which can be used to retrieve data when current instance of borgmaster crashes
- **Fauxmaster:** Borgmaster simulator used to debug failures using check-pointed files
- **Scheduler:** As the name suggests provides a scheduling algorithm which prioritizes jobs based on higher priority, two parts in algorithm - feasibility checking (find set of machines based on task constraints) and scoring (goodness of each feasible machine based on tuning parameters)
- **Borglet:** Local agent present on every machine which manages tasks running in process which coordinates with the borgmaster

## 6.2 Google Kubernetes

Google has been developing and using containers for managing applications for more than 10 years using Borg 6.1. Containers are preferred as they offer less overhead compared to virtual machines. Kubernetes is advanced container based management system built upon using positive techniques used in Borg. It is designed to be used by Google to manage applications running in its data center in the near future. It merges two kinds of containers - Docker and Linux Containers (LXC). Docker deals with packaging, control, layered file system of the system and LXC with performance isolations, resource management. Application containers view the world as a pool of machines, failure of a machine doesn't really affect application.

Kubernetes decouples a large application into micro-services to solve them individually. This is similar to Service Oriented Architecture (SOA) where an application developer designs application in which the application exposes micro services to other components via protocols over a network. These micro-services are executed with the help of containers. This abstraction of application into micro-services helps both application developer development and system in management.

### 6.2.1 Architecture

Kubernetes API server exposes services which could be used by other components. The API server is stateless. APIs can be accessed using kubectl command line interface (CLI) or AJAX based web interface. Individual Kubelet (similar to borglet in Borg) processes run on each physical machine which is used to manage pods (explained in coming section) on its system. Furthermore a scheduler is used to schedule pods onto machines based on a scheduling algorithm.

- **Pod** is used as the atomic unit for scheduling. It is typically a collections of one/more container(s). It also contains collections of data volumes. It is to be noted that data volumes are independent of containers. Name spaces are shared inside a pod. Furthermore every object inside a pod has a label.
- **Label** is a key/value pair. A label is present on each object present in a pod. Labels are used to issue queries. They provide powerful mechanism to slice pods that are created by a user based on requirement.
- **Reconciliation** are tight small loops used to convert the current state to desired state at specified kubelet by an user. Tasks like creation, health check, deletion of pods are carried out by reconciliation.
- **Services** typically contain one or more pods. A service gets a known static IP address, DNS name etc. It helps us achieve tasks like configuration and reference. It is always a good idea to use load balancers at every layer while designing applications. The Load balancer present outside the service layer in Kubernetes is responsible to keeping track of the services/pods.

## 6.3 Facebook's Tupperware

The process of deploying applications running single instance is simple. The challenge lies when deploying on a production scale. There are several to be taken care of while doing the same like - provisioning, distribute binaries, geo-distribution, daemonize process, monitoring, failover etc. This leads to more time devoted on making application run, than on application logic. This was to core problem engineers were facing at Facebook.

Facebook came up with a system called Tupperware, which provisions services by taking user requirements and mapping them into actual allocation using containers. Using Tupperware an engineer doesn't have to worry about machines in production. It Handles failover by rescheduling services onto a different machine automatically. It makes efficient use of resources. In 2014, Facebook used to manage about 15k+ services simultaneously making use of Tupperware. A job referred to a service and task to an instance of a service.

### 6.3.1 System Components

The main components of the architecture are as described below,

- **Scheduler:** Multiple scheduler with one of them as primary and the others as standby
- **Agent process:** Single instance present on each physical machine. It takes care of managing tasks present on that machine by co-ordinating with the scheduler
- **Server Database:** Stores the details of resources available across machines. The scheduler makes use of this information while scheduling jobs/tasks

### 6.3.2 Running an Application using Tupperware

Initially, the engineer runs a configuration/spec file that is to be deployed and passes it onto the scheduler using CLI. The scheduler computes the services to machine mapping by looking up at server database. After which it schedules the task onto an agent process present on the scheduler machine. The agent process downloads the required binaries using torrent like sharing. Once the host has the binaries it reads the spec file to know how to start it up and performs various dependency checks, allocations etc. accordingly.

## 7. Conclusions

In conclusion, containers have been here for a while now and are close to reaching stability. The major limitation of a container is the ability to only run OS of host kernel type, and not other OSes which are of a different kernel. There are several articles which speak about its security flaws due to the exposure of host system call interface. However this problem has been addressed with more care over the years but hasn't been completely solved yet.

Linux cgroups and namespaces have provided an amazing platform for a system like containers the develop however, more efforts need to be put in stabilizing its security flaws. COW disk images have showed great potentials in lowering the amount of storage space required but this comes at a cost of minimal performance overhead and to balance between the two is a requirements decision.

Virtual machines has been a great technology but the performance overheads it carries along with is a heart breaker. System containers stretch this arena by providing something in the between isolation and performance provided by native and virtual machines. It provides isolation close to that of virtual machines, at a performance cost of near native. This gives enormous potential for system containers to be used in cloud computing (PAAS), HPC etc. in the near future.

Application container managers like Docker, Rocket are excellent choices for application developers as they provide an isolated environment for development which makes it easy to transfer from one person to another along with its dependencies. Application containers generally provide better performance then system containers, however it must be remembered that both have a different purpose as application containers are typically only used to hold a single application instance, system containers on the other hand provide a complete OS feel. It would largely benefactor large organizations who run large number of applications in a compute farm to adopt application containers into their environment the easy the process of development and developing applications. Studies like Borg, Kubernetes and Tupperware have shown the same.

In all containers are here to stay, and have tremendous potential for the future in virtualization for usage across different areas by providing good amounts of resource control and isolation by with minimal overhead. But only time will tell us if it would actually replace virtual machines or not.

Having said that containers are a relatively new arena, there is tremendous potential for improvement in different areas. A couple of the proposed optimizations were discussed for startup latency and disk I/O throughput in our discussion, but there are many more issues of containers that are still not addressed and have a great scope which have been mentioned in the upcoming section.

## 7.1 Future Scope

A quick pointers to the list of areas where containers have potential scope has been mentioned below. These pointers are generic to both system and application containers form a virtualization viewpoint.

1. **Multi Subsystem cgroup Hierarchies:** Control group hierarchies used by container technologies today are mostly individual hierarchies for each subsystem. Using a single hierarchy to support multiple subsystems hasn't been explored much yet. Doing so could potentially decrease the overheads as traversing a single hierarchy instead of multiple ones might provide better performance.
2. **Existing Bugs Fixes:** There are several bugs in the existing container disk images like for instance proc files for meminfo, cpuinfo etc. display the details for the host system and aren't container aware. Few of the projects like lxc have fixed a few of these bugs but there must be several more of such bugs that have been left unexplored. These bugs could be tracked down to make the container feel more like an actual system. This is particularly interesting for applications who tune their workload based on the system resources available.
3. **Sharing Packages with Host:** Currently there is typically no support for sharing packages between the host and container. One of the future extension could be a design a container manager which would permit the guest OS to use and access packages of the host and override these only when necessary and not otherwise. This would have the additional storage and installation overheads.
4. **Vulnerabilities in Containers:** Most of the vulnerabilities in containers are primarily because the Host's system call table is exposed to all containers but also depends on what the image consists of. The initial design on OS didn't incorporate the support for containers and hence most system calls are container unaware. This causes trouble and to look for all the system call faults from a container's perspective isn't an easy job. Although over years many the vulnerabilities have been fixed, a recent study [17] on Docker images still showed that over 90% of Docker images still suffer for vulnerabilities.
5. **Inadequate Support for Live-Migration:** Projects like CRIU claim to support live migrations for containers but these solutions are either incomplete or inefficient. For instance it takes lesser time to kill the current container and start the same one on a new system than to perform live migration. Hence the support for live migration could be made more effective.
6. **Memory Reclamation:** Current host system's global reclamation policy is container aware but this reclaims memory solely based on the absolute difference in the memory soft and hard limits. The absolute difference might not be the right solution as we aren't able to give preference to containers with higher limits. To fix this the policy could be modified to accommodate reclamation based on relative difference in soft and hard limits. This policy then could even be extended to assigning relative reclamation weights to containers.
7. **Per cgroup Memory Accounting Enabler:** Studies have showed that memory cgroup subsystem introduces significant overheads (about 15% of page fault time) in accounting memory. This memory accounting can only be turned off and on a system level. An interesting work would be to allow enabling/disabling accounting for a per cgroup basis. This way overheads are limited to only when required and not otherwise.

# Bibliography

- [1] D. Inc., “Docker official documentation,” 2016.
- [2] K. Agarwal, B. Jain, and D. E. Porter, “Containing the hype,” in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, p. 8, ACM, 2015.
- [3] D. Beserra, E. D. Moreno, P. Takako Endo, J. Barreto, D. Sadok, and S. Fernandes, “Performance analysis of lxc for hpc environments,” in *Complex, Intelligent, and Software Intensive Systems (CISIS), 2015 Ninth International Conference on*, pp. 358–363, IEEE, 2015.
- [4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, pp. 171–172, IEEE, 2015.
- [5] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *Proceedings of the Tenth European Conference on Computer Systems*, p. 18, ACM, 2015.
- [6] J. E. Smith and R. Nair, “The architecture of virtual machines,” *Computer*, vol. 38, no. 5, pp. 32–38, 2005.
- [7] G. Banga, P. Druschel, and J. C. Mogul, “Resource containers: A new facility for resource management in server systems,” in *OSDI*, vol. 99, pp. 45–58, 1999.
- [8] D. P. Bovet and M. Cesati, *Understanding the Linux kernel*. ” O’Reilly Media, Inc.”, 2005.
- [9] K. Kolyshkin, “Virtualization in linux,” *White paper, OpenVZ*, vol. 3, p. 39, 2006.
- [10] S. Soltész, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, “Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors,” in *ACM SIGOPS Operating Systems Review*, vol. 41, pp. 275–287, ACM, 2007.
- [11] P. B. Menage, “Adding generic process containers to the linux kernel,” in *Proceedings of the Linux Symposium*, vol. 2, pp. 45–57, Citeseer, 2007.
- [12] M. Kerrisk, “Lwn namespaces overview,” 2013.
- [13] R. Morabito, J. Kjallman, and M. Komu, “Hypervisors vs. lightweight virtualization: a performance comparison,” in *Cloud Engineering (IC2E), 2015 IEEE International Conference on*, pp. 386–393, IEEE, 2015.
- [14] M. S. Rathore, M. Hidell, and P. Sjödin, “Kvm vs. lxc: comparing performance and isolation of hardware-assisted virtual routers,” *American Journal of Networks and Communications*, vol. 2, no. 4, pp. 88–96, 2013.

- [15] T. Harter, B. Salmon, R. Liu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Slacker: Fast distribution with lazy docker containers,”
- [16] J. Kang, B. Zhang, T. Wo, C. Hu, and J. Huai, “Multilanes: providing virtualized storage for os-level virtualization on many cores,” in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, pp. 317–329, 2014.
- [17] A. Bettini, “Vulnerability exploitation in docker container environments,” 2015.

## A. Appendix: Demonstrations

Let's get started with some hands on. It is recommended to go through all the demos in this section one after the other in sequence to get the real feel of the system, however feel free to try other options which you come across along the way. In this section we focus on short demos pertaining to cgroups, namespaces containers and disk images. Each demo section is typically followed by an exercise section, and its again strongly recommended to try out the same.

All demo's illustrated in the section are run on an ubuntu 14.04 LTS distro running on top of Linux kernel version of 4.1.3. However it is not necessary for you to run the same kernel version. Anything **above kernel version 3.8 is recommended**, although most features(except for user namespaces) are supported by kernel versions 3.2 and up. Remember that cgroups and namespaces are enabled by the default kernel configs these days by the kernels. Enough said, let's get started.

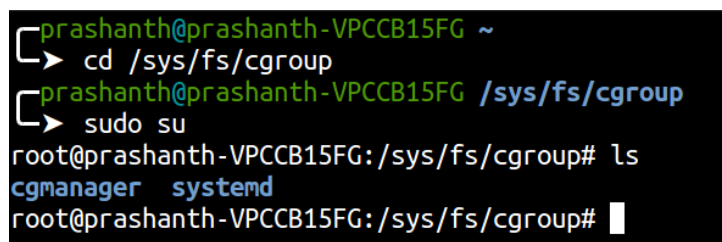
### A.1 Control Groups

This section gives a set of simple demonstration on memory subsystem to get you a feel of control groups. The current section first makes you mount a memory subsystem on a system.

#### A.1.1 Checking all the mounted cgroup subsystems

This demo helps you getting to know all the subsystems currently mounted onto your system. It gives you an overall picture of cgroup hierarchies existing on your system.

1. Navigate to the cgroups pseudo filesystem it contains all the cgroup subsystems which are currently attached to the system,  
`cd /sys/fs/cgroup`



```
prashanth@prashanth-VPCCB15FG ~  
└─> cd /sys/fs/cgroup  
prashanth@prashanth-VPCCB15FG /sys/fs/cgroup  
└─> sudo su  
root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls  
cgmanager systemd  
root@prashanth-VPCCB15FG:/sys/fs/cgroup#
```

Figure A.1: Checking current subsystems mounted



2. Login as root user,  
`sudo su`
3. List all the contents of directory,  
`ls`

You should see a directory similar to as shown in Fig:A.1. These folders list the cgroups mounted in the system currently. Although directories like `systemd` & `cgmanager` aren't subsystems and have to do with further options on cgroups, `systemd` is present by default and `cgmanager` (not relevant to you) is only on my machine. If you see additional directories, you need not worry as it's a good thing, as they represent the cgroup subsystems already mounted on your system.

### A.1.2 Mount/Unmount a New subsystem hierarchy

This section introduces you on how to attach a new hierarchy to your existing system with one or more cgroup subsystems. It allows you to see how to unmount an existing subsystem if you want to. Fig:A.2 gives you a proper illustration of its working.

1. Create a new directory named `memory` (if it doesn't exist) this will be used to manage the memory subsystem,  
`mkdir memory`
2. Mount the memory cgroup (if it hasn't been mounted already). The options specified after `-o` represent the subsystems to be attached. You can specify multiple subsystems with comma separated values.  
`mount -t cgroup -o memory cgroup /sys/fs/cgroup/memory`
3. Change directory to `memory`  
`cd memory`
4. Now we are in the root memory cgroup, remember that a root cgroup is typically not configurable, it merely provides statistics and accounting information
5. List all the files in the current directory and see all the files present each file corresponds to different attributes (maybe configurable) to the child cgroup  
`ls`
6. Try running a `cat` on most of the files and see what they display.
7. To unmount existing hierarchy, do the following  
`cd ..`  
`umount memory`  
`rmdir memory`

### A.1.3 Creating a cgroup and adding process to it

Now that we have mounted a memory cgroup let's move on to creating a cgroup node and adding a process to it. Refer Fig:A.3

```

root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls
cgmanager systemd
root@prashanth-VPCCB15FG:/sys/fs/cgroup# mkdir memory
root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls
cgmanager memory systemd
root@prashanth-VPCCB15FG:/sys/fs/cgroup# mount -t cgroup -o memory cgroup /sys/fs/cgroup/memory/
root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls
cgmanager memory systemd
root@prashanth-VPCCB15FG:/sys/fs/cgroup# cd memory/
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory# ls
cgroup.clone_children          memory.kmem.tcp.failcnt      memory.soft_limit_in_bytes
cgroup.event_control           memory.kmem.tcp.limit_in_bytes memory.stat
cgroup.procs                   memory.kmem.tcp.max_usage_in_bytes memory.swappiness
cgroup.sane_behavior           memory.kmem.tcp.usage_in_bytes memory.usage_in_bytes
lxc                             memory.kmem.usage_in_bytes  memory.use_hierarchy
memory.failcnt                 memory.limit_in_bytes       notify_on_release
memory.force_empty             memory.max_usage_in_bytes   release_agent
memory.kmem.failcnt            memory.move_charge_at_immigrate tasks
memory.kmem.limit_in_bytes     memory.numa_stat            user
memory.kmem.max_usage_in_bytes memory.oom_control
memory.kmem.slabinfo           memory.pressure_level
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory# cd ..
root@prashanth-VPCCB15FG:/sys/fs/cgroup# umount memory
root@prashanth-VPCCB15FG:/sys/fs/cgroup# rmdir memory/
root@prashanth-VPCCB15FG:/sys/fs/cgroup# ls
cgmanager systemd
root@prashanth-VPCCB15FG:/sys/fs/cgroup# █

```

Figure A.2: Mounting a new hierarchy

1. Create a new child cgroup of the root memory cgroup by making a new directory  
`mkdir memcg1`
2. Now navigate into this newly created cgroup, you could browse its contents which appear to be similar to its parent  
`cd memcg1`
3. procs file stores the tasks belonging to the current cgroup. Display the contents of this file, it initially is empty as no task exists  
`cat cgroup.procs`
4. Open a parallel terminal and start a new process and note its process id. (If you are creating a new firefox process, make sure it wasn't running earlier),  
`ctrl + shift + t`  
`firefox &`
5. Now come back to the original terminal and add the created process into the current cgroup (memcg1) as shown in  
`echo <pid-of-process> > cgroups.procs`
6. Now display contents of tasks again and you will find the pids of all your added processes,  
`cat cgroups.procs`
7. Now you can view the memory statistics of the current cgroup by as shown in  
`cat memory.stat`

```

root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory# mkdir memcg1
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory# cd memcg1
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1# cat cgroup.procs
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1# echo 29117 > cgroup.procs
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1# cat cgroup.procs
29117
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1# cat memory.stat
cache 4096
rss 421888
rss_huge 0
mapped_file 0
writeback 0
pgpgin 104
pgpgout 0
pgfault 103
pgmajfault 0
inactive_anon 0
active_anon 421888
inactive_file 4096
active_file 0
unevictable 0
hierarchical_memory_limit 9223372036854771712
total_cache 4096
total_rss 421888
total_rss_huge 0
total_mapped_file 0
total_writeback 0
total_pgpgin 104
total_pgpgout 0
total_pgfault 103
total_pgmajfault 0
total_inactive_anon 0
total_active_anon 421888
total_inactive_file 4096
total_active_file 0
total_unevictable 0
root@prashanth-VPCCB15FG:/sys/fs/cgroup/memory/memcg1#

```

Figure A.3: Mounting a new hierarchy

#### A.1.4 Setting constrains

1. Initially the memory limit is the parent cgroup node value, and this can be set using  
`echo 128M > memory.limit_in_bytes`
2. You could check various resource accounting information like current memory usage, maximum memory used, limit of on memory etc.  
`cat memory.usage_in_bytes`  
`cat memory.max_usage_in_bytes`  
`cat memory.limit_in_bytes`
3. One important parameter to track memory oom is failcnt, it lets us know how many times a cgroup has wanted to exceed its allotted limit  
`cat memory.failcnt`
4. Similarly `memory.kmem.*` and `memory.kmem.tcp.*` stats could be accounted/controlled

```

root@prashanth-VPCCB15FG:/home/prashanth# ip netns add netns1
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ping 127.0.0.1
connect: Network is unreachable
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ip link set dev lo up
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.084 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.091 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.094 ms
^C
--- 127.0.0.1 ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 1998ms
rtt min/avg/max/mdev = 0.084/0.089/0.094/0.011 ms
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ping 192.168.200.1
connect: Network is unreachable
root@prashanth-VPCCB15FG:/home/prashanth# █

```

Figure A.4: Creating new namespace

5. It is to be remembered that by default processes in memory that exceed usage limit maybe killed depending on the load of the system. To avoid this situation, you will need disable it as given below  
`echo 1 > memory.oom_control`

## A.2 Namespaces

Now that we have gotten a feel of cgroups, lets move on to namespace. Namespaces are usually created by passing appropriate clone flags to the `clone()` system call. But network namespace provides us with simple command line interfaces to create new namespaces. We use network namespace for this reason, for the purpose of simplicity in demonstration, however it is advised to try the exercise section where you will create other namespaces using the `clone()` system call. This section gives a simple demonstration of network namespace.

### A.2.1 Creating a new network namespace

This section demonstrates creating a new namespace and pinging its loopback interface.

1. Switch to root user  
`sudo su`
2. Create a new network namespace  
`ip netns add netns1`
3. List all the devices present in the newly created network namespace. We observe that the devices exposed in default network namespace isn't visible here. Only loop-back is present by default  
`ip netns exec netns1 ip link list`

4. Try pinging loop-back address or any other address. The network is unreachable as devices are either unavailable or not up (loopback)  
`ip netns exec netns1 ping 127.0.0.1`
5. To enable loop back,  
`ip netns exec netns1 ip link set dev lo up`
6. Now repeat step 4, and you are able to ping loop-back, but no other address as shown in  
`ip netns exec netns1 ping 192.168.200.1`

## A.2.2 Creating virtual tunnel between default and new NS

Now that we have created a network namespace, let's try to create a tunnel between the newly created namespace and default namespace to route traffic between them.

1. Now we can create two virtual devices which forms a tunnel in the default namespace,  
`ip link add veth0 type veth peer name veth1`
2. Now we move one of the virtual devices to the newly created namespace  
`ip link set veth1 netns netns1`
3. We can look at IP tables and routing tables in the new namespace,  
`ip netns exec netns1 route`  
`ip netns exec netns1 iptables -L`
4. Now assign IPs to both the virtual devices  
`ifconfig veth0 192.168.100.1/24 up`  
`ip netns exec netns1 ifconfig veth1 192.168.100.2/24 up`
5. Now we can look at the devices on both the namespaces and see the difference,  
`ip link list`  
`ip netns exec netns1 ip link list`
6. We can now ping the virtual device on both namespaces, using the other as shown in Fig:[A.5](#)  
`ping 192.168.100.2`  
`ip netns exec netns1 ping 192.168.100.1`

Furthermore, a software bridge could be setup to provide Internet access to processes running on a network namespace.

## A.3 Linux Containers

This section gives a set of demonstration on setting up of lxc and how to use it to manage containers. The lxc version for the purposes of demonstration is 1.1 although, the version of lxc shouldn't be making much of a difference.

```

root@prashanth-VPCCB15FG:/home/prashanth# ip link add veth0 type veth peer name veth1
root@prashanth-VPCCB15FG:/home/prashanth# ip link set veth1 netns netns1
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 route
Kernel IP routing table
Destination      Gateway          Genmask          Flags Metric Ref    Use Iface
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 iptables -L
Chain INPUT (policy ACCEPT)
target          prot opt source                destination

Chain FORWARD (policy ACCEPT)
target          prot opt source                destination

Chain OUTPUT (policy ACCEPT)
target          prot opt source                destination
root@prashanth-VPCCB15FG:/home/prashanth# ifconfig veth0 192.168.100.1/24 up
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ifconfig veth1 192.168.100.2/24 up
root@prashanth-VPCCB15FG:/home/prashanth# ping 192.168.100.2
PING 192.168.100.2 (192.168.100.2) 56(84) bytes of data.
64 bytes from 192.168.100.2: icmp_seq=1 ttl=64 time=0.167 ms
64 bytes from 192.168.100.2: icmp_seq=2 ttl=64 time=0.095 ms
^C
--- 192.168.100.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.095/0.131/0.167/0.036 ms
root@prashanth-VPCCB15FG:/home/prashanth# ip netns exec netns1 ping 192.168.100.1
PING 192.168.100.1 (192.168.100.1) 56(84) bytes of data.
64 bytes from 192.168.100.1: icmp_seq=1 ttl=64 time=0.114 ms
64 bytes from 192.168.100.1: icmp_seq=2 ttl=64 time=0.109 ms
^C
--- 192.168.100.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.109/0.111/0.114/0.010 ms
root@prashanth-VPCCB15FG:/home/prashanth# █

```

Figure A.5: Creating new namespace

### A.3.1 Installing and running a simple container

This demo aims at installing lxc in your system and running a simple container. It is highly recommended to use Linux kernel version of 3.8+ as this is a requirement for unprivileged root containers discussed in the user namespace section and lxc currently supports this feature.

1. Switch to root user  
`sudo su`
2. To install lxc, run the following command  
`apt-get install lxc`
3. Create a new ubuntu container, this process might take time for the first time as it will have to fetch the ubuntu image from its repository. Once you have a local copy of your image, the time taken to deploy a container there after would be lesser. `-n` specifies the name of the container and `-t` specifies the image template to be used.  
`lxc-create -t ubuntu -n ubuntu-01`
4. Start the created container using, `-F` specifies to run the container in foreground  
`lxc-start -F -n ubuntu-01`
5. You would be directed to a terminal which asks for your username and password. Default username and password is ubuntu

```

Ubuntu 14.04.4 LTS ubuntu-01 console

ubuntu-01 login: * Stopping save kernel messages ...done.
ubuntu
Password:
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.1.3 x86_64)

 * Documentation:  https://help.ubuntu.com/

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

ubuntu@ubuntu-01:~$ cd /
ubuntu@ubuntu-01:/$ ls
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
ubuntu@ubuntu-01:/$ hostname
ubuntu-01
ubuntu@ubuntu-01:/$ sudo poweroff
[sudo] password for ubuntu:

Broadcast message from ubuntu@ubuntu-01
(/dev/lxc/console) at 11:37 ...

The system is going down for power off NOW!
ubuntu@ubuntu-01:/$ wait-for-state stop/waiting
 * Asking all remaining processes to terminate...
   ...done.
 * All processes ended within 1 seconds...
   ...done.
 * Deactivating swap...
   ...fail!
 * Unmounting local filesystems...
   ...done.
mount: / is busy
 * Will now halt
root@prashanth-VPCCB15FG:/home/prashanth# lxc-info -n ubuntu-01
Name:          ubuntu-01
State:         STOPPED
root@prashanth-VPCCB15FG:/home/prashanth# sudo lxc-destroy -n ubuntu-01
Destroyed container ubuntu-01
root@prashanth-VPCCB15FG:/home/prashanth# █

```

Figure A.6: Running a lxc container

6. Once you are logged in, now you are inside the ubuntu container. Browse through the container and you would realise that the whole container is isolated from the host system. You have your own ROOTFS all though your kernel is still shared with the host system.
7. Try running top and have a look at the processes running  
top
8. To come out of a container, stopping the container  
sudo poweroff
9. To destroy a container permanently, Refer Fig:A.6  
lxc-destroy -n ubuntu-01

### A.3.2 Some helpful commands in lxc

A beginner might be overwhelmed with the list of commands lxc provides. Here is demo of the useful commands in lxc.

1. To start and run a container in background  
`lxc-start -d -n ubuntu-01`
2. To attach a console to running container  
`lxc-console -n ubuntu-01`
3. To attach unattach shell from running container  
`Ctrl + a`  
`q`
4. To freeze a container  
`lxc-freeze -n ubuntu-01`
5. To unfreeze a container  
`lxc-unfreeze -n ubuntu-01`
6. To clone a container  
`lxc-clone -o ubuntu-01 -n ubuntu-01-cloned`

### A.3.3 Python API to use lxc

This demonstrates managing lxc using python API. It creates a container from python and runs the same.

1. Switch to python environment using sudo privileges  
`sudo python3`
2. Import lxc library  
`import lxc`
3. Initialize a new container class  
`container=lxc.Container("container-1")`
4. Create a new container  
`container.create("ubuntu")`
5. Start a new container  
`container.start()`
6. Stop the container  
`container.stop()`

You will notice that on running `top` Docker contains only about a couple of processes, where as running the same command in lxc shows many processes. This illustrates the difference between an system and application container.

### A.3.4 Disk Images in lxc

This demonstrates how disk images are handled in lxc. It walks through the host fs to show where the actual files lie in a default configuration. Fig:A.7 shows this.



```

root@prashanth-VPCCB15FG:/home/prashanth# ls /var/cache/lxc/
trusty
root@prashanth-VPCCB15FG:/home/prashanth# ls /var/lib/lxc/
ubuntu-01
root@prashanth-VPCCB15FG:/home/prashanth# cat /var/lib/lxc/ubuntu-01/config
# Template used to create this container: /usr/share/lxc/templates/lxc-ubuntu
# Parameters passed to the template:
# For additional config options, please look at lxc.container.conf(5)

# Uncomment the following line to support nesting containers:
#lxc.include = /usr/share/lxc/config/nesting.conf
# (Be aware this has security implications)

# Common configuration
lxc.include = /usr/share/lxc/config/ubuntu.common.conf

# Container specific configuration
lxc.rootfs = /var/lib/lxc/ubuntu-01/rootfs
lxc.utsname = ubuntu-01
lxc.arch = amd64

# Network configuration
lxc.network.type = veth
lxc.network.link = lxcbr0
lxc.network.flags = up
lxc.network.hwaddr = 00:16:3e:28:d3:a7
root@prashanth-VPCCB15FG:/home/prashanth# ls /var/lib/lxc/ubuntu-01/rootfs
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys
root@prashanth-VPCCB15FG:/home/prashanth#

```

Figure A.7: Exploring lxc images

1. All the local cache for images exist at `/var/cache/lxc/`  
`ls /var/cache/lxc/`
2. Each of the container created exists at `/var/lib/lxc/`, and appear as the container name  
`ls /var/lib/lxc/`
3. The config file for a container can be found at `/var/lib/lxc/⟦container-name⟧/config`. The configurations relating to resource constrains, setup etc. will be specified here  
`cat /var/lib/lxc/ubuntu-01/config`
4. This is the actual directory containing the root mount for a container  
`ls /var/lib/lxc/ubuntu-01/rootfs`

Notice that the ROOTFS displayed here matches with the one running on the container. Hence lxc uses this same directory as the mount point to the container. For each created container there exists a ROOTFS at `/var/lib/lxc/⟦container-name⟧/rootfs`

## A.4 Docker

This section gives a set of demonstration on setting up of Docker and how to use it to manage application containers. The Docker version for the purposes of demonstration is 1.10 although, the version of Docker shouldn't be making much of a difference.

### A.4.1 Installing required dependencies for Docker

This demo aims at installing the dependencies for installing Docker in your system and running a simple container.

1. Switch to root user  
`sudo su`
2. Update your current system  
`apt-get update`
3. Make sure the APT works fine with https and CA certificates installed  
`apt-get install apt-transport-https ca-certificates`
4. Add new GPG (GNU Privacy Guard) key  
`apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80  
--recv-keys 58118E89F3A912897C070ADBF76221572C52609D`
5. Open `/etc/apt/sources.list.d/Docker.list` with any editor and remove all existing content and replace it with the the line given below (Make sure to use your ubuntu version) and now save and close vim `vim /etc/apt/sources.list.d/Docker.list`  
ADD `‘‘deb https://apt.Dockerproject.org/repo ubuntu-trusty main’’`
6. Again update your package index  
`apt-get update`
7. Purge the old repository if it exists  
`apt-get purge lxc-Docker`
8. Verify that pulling is from the right repository  
`apt-cache policy Docker-engine`
9. Again update your package index  
`apt-get update`
10. One last package  
`apt-get install linux-image-extra-$(uname -r)`

### A.4.2 Installing Docker

Now lets get to the actual installation of the Docker engine.

1. Update your package index  
`apt-get update`
2. Install Docker engine  
`apt-get install Docker-engine`
3. Start Docker service  
`service Docker start`
4. To verify installation run  
`Docker run hello-world`

```

root@prashanth-VPCCB15FG:/home/prashanth# docker run --name my-container -i -t ubuntu /bin/bash
root@953b2e3ac6c3:/# ls /
bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var
root@953b2e3ac6c3:/# hostname
953b2e3ac6c3
root@953b2e3ac6c3:/# exit
exit
root@prashanth-VPCCB15FG:/home/prashanth# hostname
prashanth-VPCCB15FG
root@prashanth-VPCCB15FG:/home/prashanth# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
953b2e3ac6c3        ubuntu             "/bin/bash"        35 seconds ago     Exited (0) 19 seconds ago
root@prashanth-VPCCB15FG:/home/prashanth# docker stop my-container
my-container
root@prashanth-VPCCB15FG:/home/prashanth# docker rm my-container
my-container
root@prashanth-VPCCB15FG:/home/prashanth# docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
root@prashanth-VPCCB15FG:/home/prashanth#

```

Figure A.8: Running Docker container

### A.4.3 Running a Docker Container

This demonstrates creating, running, stopping and removing a Docker container running ubuntu distro. Fig:A.8 shows this.

1. This command creates and runs a new Docker container in a single command.  
Docker run --name my-container -i -t ubuntu /bin/bash
2. Now you will be switched into the container. Now navigate around to verify the same. Run simple commands like  
ls /  
hostname
3. Try running top and have a look at the processes running top
4. To leave to container, exit the shell  
exit
5. Now run hostname again and you would find it return the host machine's hostname  
hostname
6. To look at all containers present on Docker  
Docker stop my-container
7. To delete the container permanently  
Docker rm my-container

Running top in lxc displayed several processes, but Docker shows a very less number. This illustrates the difference between application and system container.

### A.4.4 Disk Image layers in Docker

Lets move on to one of the final and one of the most import demonstration of this discussion, which is the Docker images. This demonstrates how disk image layers are handled in Docker.

```

root@prashanth-VPCCB15FG:/home/prashanth# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
trusty               latest             bd3d156d2df9       5 weeks ago        228.3 MB
kvmprashanth/static_web latest             c67bfae4022e       5 weeks ago        227.6 MB
kvmprashanth/apache2 latest             2a8ea991d44a       5 weeks ago        223.8 MB
changed-ubuntu       latest             3e6d9a528b6a       7 weeks ago        187.9 MB
fedora               latest             760a896a323f       7 weeks ago        204.5 MB
ubuntu               latest             14b59d36bae0       7 weeks ago        187.9 MB
busybox              latest             3240943c9ea3       7 weeks ago        1.114 MB
hello-world          latest             690ed74de00f       5 months ago       960 B
root@prashanth-VPCCB15FG:/home/prashanth# docker pull ubuntu:15.04
15.04: Pulling from library/ubuntu
9502adfb7f1: Pull complete
4332ffb06e4b: Pull complete
2f937cc07b5f: Pull complete
a3ed95cae02: Pull complete
Digest: sha256:2fb27e433b3ecccea2a14e794875b086711f5d49953ef173d8a03e870f1510f
Status: Downloaded newer image for ubuntu:15.04
root@prashanth-VPCCB15FG:/home/prashanth# docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
trusty               latest             bd3d156d2df9       5 weeks ago        228.3 MB
kvmprashanth/static_web latest             c67bfae4022e       5 weeks ago        227.6 MB
kvmprashanth/apache2 latest             2a8ea991d44a       5 weeks ago        223.8 MB
changed-ubuntu       latest             3e6d9a528b6a       7 weeks ago        187.9 MB
fedora               latest             760a896a323f       7 weeks ago        204.5 MB
ubuntu               latest             14b59d36bae0       7 weeks ago        187.9 MB
busybox              latest             3240943c9ea3       7 weeks ago        1.114 MB
ubuntu               15.04              d1b55fd07600       10 weeks ago       131.3 MB
hello-world          latest             690ed74de00f       5 months ago       960 B

```

Figure A.9: Docker pulling images

```

root@prashanth-VPCCB15FG:/home/prashanth# docker history ubuntu:15.04
IMAGE               CREATED             CREATED BY          SIZE
d1b55fd07600        10 weeks ago       /bin/sh -c #(nop)  CMD ["/bin/bash"] 0 B
<missing>            10 weeks ago       /bin/sh -c sed -i 's/^#\s*(deb.*universe\)$/ 1.879 kB
<missing>            10 weeks ago       /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic 701 B
<missing>            10 weeks ago       /bin/sh -c #(nop)  ADD file:3f4708cf445dc1b537 131.3 MB
root@prashanth-VPCCB15FG:/home/prashanth# docker history ubuntu:15.10
IMAGE               CREATED             CREATED BY          SIZE
4e3b13c8a266        3 days ago         /bin/sh -c #(nop)  CMD ["/bin/bash"] 0 B
<missing>            3 days ago         /bin/sh -c sed -i 's/^#\s*(deb.*universe\)$/ 1.863 kB
<missing>            3 days ago         /bin/sh -c set -xe  && echo '#!/bin/sh' > /u 701 B
<missing>            3 days ago         /bin/sh -c #(nop)  ADD file:43cb048516c6b80f22 136.3 MB
root@prashanth-VPCCB15FG:/home/prashanth# █

```

Figure A.10: Difference between layers of a same image

1. You can list all the cached images on your local machine using, I have a lot of images displayed but you may have one or none  
Docker images
2. Now lets pull a new image  
Docker pull ubuntu:15.04
3. Now list the images again and you will find the new image in the cache as shown in Fig:A.9  
Docker images
4. This is the actual directory containing the root mount for a container  
ls /var/cache/lxc/ubuntu-01/rootfs
5. Now lets pull another new image  
Docker pull ubuntu:15.10
6. Now to show that Docker uses reusable layers, try the following commands

```
Docker history ubuntu:15.04
Docker history ubuntu:15.10
```

Looking at Fig:[A.10](#) you will notice that image layer-3 between ubuntu:15.04 and ubuntu:15.10 is the same and it reuses this. This proves Docker using COW layers for images.

## B. Appendix: Benchmarks

The following is the list of benchmarks used in the different experiments taken up by the case studies.

### B.1 Memory

Table B.1: Benchmarks used on memory based experiments

Benchmark	Description	Measurement Metric
STREAM	Perform simple operations on vectors. Four components - copy, add, scale & triad	Throughput (GB/s)
Random Access	Random 8-Byte words are read, modified using XOR, and written back	Throughput: Giga-updates per second (GUPS)

### B.2 CPU

Table B.2: Benchmarks used on CPU based experiments

Benchmark	Description	Measurement Metric
Y-Cruncher	Multi-threaded benchmark for multi-core system used to calculate the value of PI, and other constants	Time(ms) & Multi-core efficiency (%)
PXZ	Parallel lossless data compression using Lempel Ziv Markov chain algorithm (LZMA)	Compression Throughput (B/s)
NBENCH	CPU utilization, Floating point unit (FPU)	Benchmark indexes
Noploop	Measuring clock speed using an unrolled NOP loop	Execution time (ms)
Linpack	Dense system of linear equations using LU factorization	Instructions per second (FLOPS)

## B.3 Network

Table B.3: Benchmarks used on network based experiments

Benchmark	Description	Measurement Metric
Nuttcp	Unidirectional bulk transfer, client-server architecture	CPU cycles per byte
Netperf	Bulk data transfer and request/response performance using either TCP or UDP and the Berkeley Sockets interface	Round Trip Time (ms)
NetPIPE	Series of ping-pong tests over a range of message sizes to provide a complete measure of the performance of a network	Throughput (B/s) & Latency (ms)

## B.4 Disk I/O

Table B.4: Benchmarks used on disk I/O based experiments

Benchmark	Description	Measurement Metric
FIO	Spawn a number of threads or processes doing a particular type of I/O action	Throughput (B/s)
Filebench	Performs various file operations like open, read, write	Operations per second (op/s)
dd	Simple command line tool used to copy block size data	Throughput (b/s)

## B.5 Application

Table B.5: Application Benchmark used

Benchmark	Description	Measurement Metric
SysBench	Generates database transactions at a specified rate to target machine	Throughput (transactions/ second) & latency (ms)