Virtualization vs Containerization to support PaaS

Rajdeep Dua
VMWare
Bangalore, India
Email: rajdeepd@vmware.com

A Reddy Raja
IIIT
Hyderabad, India
Email: areddy@akrantha.com

Dharmesh Kakadia
IIIT
Hyderabad, India
Email: dharmesh.kakadia@research.iiit.ac.in

***Abstract*—PaaS vendors face challenges in efficiently providing services with the growth of their offerings. In this paper, we explore how PaaS vendors are using containers as a means of hosting Apps. The paper starts with a discussion of PaaS Use case and the current adoption of Container based PaaS architectures with the existing vendors. We explore various container implementations - Linux Containers, Docker, Warden Container, lmctfy and OpenVZ. We look at how each of this implementation handle Process, FileSystem and Namespace isolation. We look at some of the unique features of each container and how some of them reuse base Linux Container implementation or differ from it. We also explore how IaaSlayer itself has started providing support for container lifecycle management along with Virtual Machines. In the end, we look at factors affecting container implementation choices and some of the features missing from the existing implementations for the next generation PaaS.**

***Index Terms*—container; virtualization; paas**

I. Introduction

Platform as a Service has brought developer productivity to the forefront making it possible to create a layer of abstraction on IaaS for deployment of services easily and quickly. This has lead to unique challenges and opportunities for Infrastructure as a Service - IaaS layer.

In this paper, we look at various aspects of this abstraction and explore implementations, which use Host OS, Virtual Machines and Containers. Some of the PaaS vendors use containers to reduce overhead of creating a new VM for each application, hence reducing the cost of running a PaaS application while maintaining Isolation at Process, Network and FileSystem level.

In this paper, we explore various container technologies Warden Container[1], Openvz[4], Docker[5] and lmctfy[6]. We also look at benefits of adopting containers for todays PaaS implementations.

II. Virtual Machine and Container comparison

System Hypervisors allow Multiple Guest OS to run together on a single machine. Each Guest OS abstracts Compute, Storage and Network Components. Hypervisor itself could run on bare-metal (ESXi) or be part of an OS (KVM). Guest ISA is translated to Host ISA using multiple techniques like Hardware Virtualization or Binary Translation.

A container is a light weight operating system running inside the host system, running instructions native to the core CPU, eliminating the need for instruction level emulation or just in time compilation. Containers provide savings in resource consumption without the overhead of virtualization, while also providing isolation.

Table 1 compares Virtual Machines and Containers on multiple factors like performance, isolation security, networking, storage.

Parameter	Virtual Machines	Containers
Guest OS	Each VM runs on virtual hardware and Kernel is loaded into in its own memory region	All the guests share same OS and Kernel. Kernel image is loaded into the physical memory
Communication	Will be through Ethernet Devices	Standard IPC mechanisms like Signals, pipes, sockets etc.
Security	Depends on the implementation of Hypervisor	Mandatory access control can be leveraged
Performance	Virtual Machines suffer from a small overhead as the Machine instructions are translated from Guest to Host OS.	Containers provide near native performance as compared to the underlying Host OS.
Isolation	Sharing libraries, files etc between guests and between guests hosts not possible.	Subdirectories can be transparently mounted and can be shared.
Startup time	VMs take a few mins to boot up	Containers can be booted up in a few secs as compared to VMs.
Storage	VMs take much more storage as the whole OS kernel and its associated programs have to be installed and run	Containers take lower amount of storage as the base OS is shared

TABLE I
VM and Container feature comparision

III. PaaS Requirements

PaaS, focuses on developer productivity rather than managing Network, Storage and Compute. This makes it possible to abstract out the implementation of application runtime environment.

Some of the key requirements for a PaaS to host application are,

- Application isolation at Network, Compute and Storage level
- Lower Performance overhead - Virtual Machines are great for isolation but there is a performance overhead for the translation happening from Guest ISA to Host ISA.

978-1-4799-3766-0/14 $31.00 © 2014 IEEE
DOI 10.1109/IC2E.2014.41

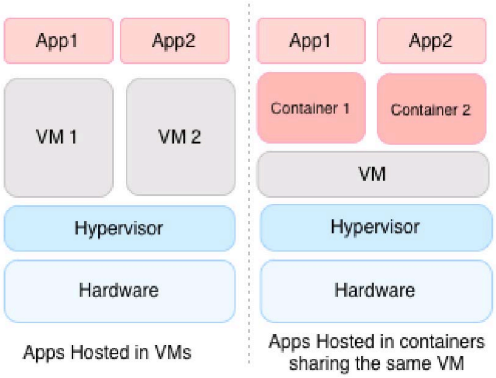

Fig. 1. Concept of Container in a VM

- Support multiple languages and runtimes.
- Fair share of CPU time
- Easy management of Application Lifecycle events
- Application state persistence and migration

A. PaaS Use Case

Platform as a Service focuses on providing Language Runtime and Services to developers leaving Infrastructure Provisioning and Management to the underlying Layer.

PaaS could use Host OS, VM or Container as hosting Environment for applications. PaaS can be implemented using one of the three scenarios.

- Option 1 : Each App runs in an Application level container directly on Host OS where PaaS layer runs
- Option 2 : Each App runs in a VM on PaaS : Each App runs in its own VM
- Option 3 : Each App runs in a container which shares a common VM or VMs for hosting Apps

Option 3 is one of the common implementation choice for most of the PaaS vendors as a combination of the VM and container meets some of the requirements of Isolation, Lifecycle management, fair share of CPU time.

B. Using VMs in Paas

Paas has an important use for VMs to abstract out the Runtime OS from the underlying hardware. Need for each application to have its own VM is the point of contention. Since containers tend to be much lighter in memory footprint, they are preferred architecture as compared to a VM for each application. VM is are used to host the containers. Other Services in Paas like Authentication, Routing, Cloud Controller, Persistent Storage is still run directly on Virtual Machines.

C. Container Adoption

Some of the PaaS implementations use container abstraction layer to optimize on resources. Table II provides the mapping between the PaaS Product and the container implementation used. Some of the large PaaS players don't seem to use any container relying on VMs or Application runtime containers for various reasons. As an example Google App Engine uses some form of Application Container to isolate tenant applications but does not use VMs or OS Level containers.

PaaS Provider	Implementation
OpenShift	Docker with LXC
Heroku	LXC
CloudFoundry	Warden Container
Stackato	Docker with Warden Container
AppFog	Warden Container
Virtuozzo	based on OpenVZ
dotCLoud	Docker with LXC

TABLE II
CONTAINER ADOPTION BY PASS PROVIDERS

IV. CONTAINER OVERVIEW

Containers are a means of providing isolation and resource management in Linux environment. The term is derived from shipping containersa standard method to store and ship any kind of cargo. An operating system container provides a generic way of isolating a process from the rest of the system. The containment applies to all the child processes. Hence, one can boot an entire operating system by spawning the init process.

It promises same level of isolation and security as a virtual machine and is more tightly integrated with the host operating system. No dependence on hardware emulation provides performance benefits over full virtualization but restricts the number of supported operating systems which can be spawned as guest operating systems; one cant boot Windows in a Linux/Unix Container. This is typically, not a requirement for PaaS providers and thus containers are well suited for providing low-overhead isolation.

A. Chroot

Chroot (change root) is a Linux command to change the root directory of the current process and its children to a new directory. Some of the containers use chroot to isolate and share the filesystem across virtual environments.

B. cgroups

Cgroups is part of kernel subsystem, provides a fine grained control over sharing of resources like CPU, memory, group of processes etc. Almost all the containers use cgroups as the underlying mechanisim for resource management. Cgroups set the limits on the usage of these resources.

C. kernel namespaces

Namespaces create barriers between containers at various levels. The **pid** namespace allows each container its own process id. The **net namespace** allows each container to have its network artifacts like routing rable, iptables, loopback interface. The **IPC** namespace provides isolation for various IPC mechanisms namely semaphore, message queues and shared memory segments. The **mnt** namespace provides each

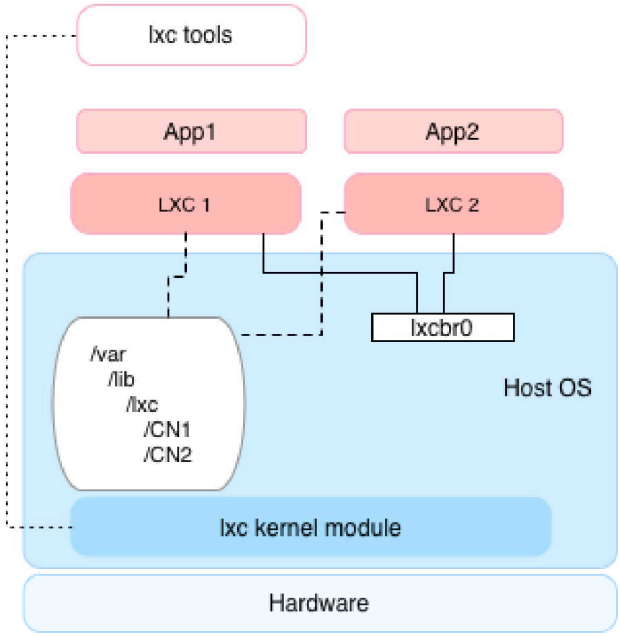

Fig. 2. Linux Container running in a Linux Host

container with its own mountpoint. The **UTS** namespace ensures that different containers can view and change hostnames specifically to them.

V. Containers - Current State

In this section, we provide a brief summary of popular container implementations.

A. Linux Containers

Linux Containers (LXC) are light weight kernel containment implementation supported on few flavors or Linux like Ubuntu and Oracle Linux.

Key Characteristics of the Linux Containers are,

- **Process** Each container is assigned a unique PID. Each container can run a single process.
- **Resource Isolation** Uses cgroups and namespaces to isolate resources
- **Network Isolation** Containers get a private IP address and veth interface connecting to a linux bridge on the host
- **File System Isolation** Each container gets a private file system by leveraging chroot.

Linux containers use Apparmor Security profile for hardening of the host. It also uses cgroups to limit device access for the containers. LXC uses a separate file system for each container which can be backed by an LVM.

Figure 2 shows a HostOS running two Linux containers LXC 1 and LXC 2, having their own file system under /var/lib/lxc. In this figure both the containers are connected to a linux bridge lxcbro and assigned IP and assigned private network components.

The key advantage of a LXC is a lightweight implementation which performs at native speeds which providing better network and filsystem isolation. Currently, LXC suffers from following the limitations:

- Containers use a Shared kernel
- Limitations in terms of secure containment environment [3]
- Limited to Linux based environments
- Implementation closely tied to a Linux Kernel

B. Warden Container

Warden container provides a kernel independent containment implementation which can be plugged to multiple underlying Host OS. This container implementation is used by Cloud Foundry project to host applications.

Some of the Key attributes of Warden container are,

- Uses namespaces for isolation of Process and Network.
- Uses cgroups concept from Linux to isolate resources
- Each container can run multiple processes.
- Supported only on ubuntu but the design makes it OS neutral.

C. Docker

Docker is a daemon which provides ability to manage Linux containers as self contained images. It utilizes LXC (Linux Containers) for the container implementation and adds image management and Union File System capability to it.

Key attribute of Docker Containers are

- **Process** - Each Container is assigned a unique process id and a private IP. Cannot run multiple processes in a single container
- **Resource Isolation** - Uses cgroups and namespaces concept from Linux Containers
- **Network Isolation** - Leverages LXC functionality
- **File System Isolation** Leverages LXC functionality
- **Container Lifecycle** - Managed using a daemon and command line tool
- **Container State** - Docker allows ability to store and retrieve container state

High level Goals of Docker project to improve security (which are limitations of Linux Containers)

- Map root user of the container to non-root user of docker
- Make docker daemon run as a non root user

D. Google lmctfy

Google lmctfy provides a resource configuration API which simplifies container management. It provides intent based configuration without the need to understand cgroups and removes the difficulties of unstable LXC APIs. It also improves the resource sharing and can support performance guarantees.

Key attribute of lmctfy Containers are

- **Process** - Use of PID namespace. Recursive containers are supported.
- **Resource Isolation** - Resources isolation using cgroups. Hierarchy splitting for easy management and higher resource utilization.

- **Network Isolation** - Plan to use net namespace in next version CL2
- **File System Isolation** - Plan to use mnt namespace in next version CL2
- **Container Lifecycle** - Plan to support freeze, chckpoint/restore and disk import/export.
- **Monitoring** - To be supported through higher level.

E. OpenVZ

OpenVZ uses a modified Linux Kernel with a set of extensions. OpenVZ manages multiple physical and Virtual servers, by using dynamic realtime partitioning. Like containers, OpenVZ has little overhead and offers higher performance and can be managed better than Hypervisor technologies. Just like other containers, OpenVZ uses cgroups and Namespaces. OpenVZ additionally provides templates that help in precreated Virtual environments.

- **Process** - Each container has it own PID namespace, IPC namespace with its own shared memory, semaphores and messages.
- **Resource Management** - Manages resource sharing using Bean Counters, Fair Share CPU Scheduler, disk quotas based on users and Containers, and can manage per continer disk I/O priority
- **Network Isolation** - Uses net namespace, has its own virtual network device, its own IP Address, filters and routing tables
- **File System Isolation** - Provides isolation to Application Files, System Libraries etc
- **Container Lifecycle** - Supports create, start, change and stop functions as part the lifecycle. Can be backed up, migrated. Supports remote management with the Libvirt API
- **Container State** - Provides Checkpointing feature for storing and recovering the last known state. The Complete state of the container like running processes, openfiles, network connections, buffers, memory segments can be stored on a file. This enables Live Migration of OpenVZ containers

VI. Container Comparison

TABLE III summarizes the containers discussed above on the following paramters: Process , Network, Filesystem Isolation. We did not cover lmctfy as the container implementation is still a work in progress.

It also discusses differences employed in managing these containers. It is clear from the table that all the implementations rely on cgroups for resource limits, chroot for process, namespaces for network and resource isolation.

VII. Choosing the Right Container

Choosing the right container for most of the PaaS implementations will be based on the following factors:

- Ecosystem for prebuilt containers
- Hardened layer for isolation of Process, Network, CPU and Filesystem

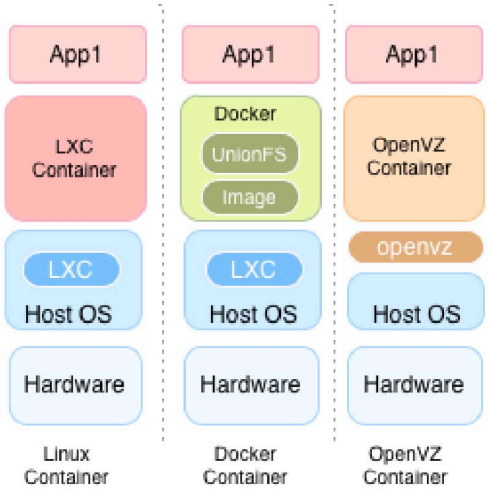

Fig. 3. Comparing various containers

Parameter	LXC	Warden	Docker	OpenVZ
Process Isolation	Uses pid namespace	Uses pid namespace	Uses pid namespace	Uses pid namespace
Resource Isolation	Uses cgroups	Uses cgroups	Uses cgroups	Uses cgroups
Network Isolation	Uses net namespace	Uses net namespace	Uses net namespace	Uses net namespace
Filesystem Isolation	Using chroot	Overlay File system using overlayfs	Using chroot	Using chroot
Container Lifecycle	Tools lxc-create, lxc-stop, lxc-start to create, start, stop a container	Containers are managed by running commands on a warden client which talks to warden server	Uses Docker daemon and a client to manage the containers	uses vzctl to manage container lifecycle

TABLE III
Container Implementation Comparison

- Tools to manage lifecycle of a container
- Ability to migrate containers between hosts
- Support for multiple OS and kernels

VIII. Cloud Orchestration support for Containers

Most of the PaaS products are built on top of existing Cloud Orchestration products like AWS, vSphere and OpenStack. In the current implementations Containers lie at a layer above IaaS as a part of PaaS implementation. These containers typically run on top of a Virtual Machine rather than running

directly on Host OS.

Openstack with its Havana release supports managing Containers (docker in particular) along with VMs using a libvirt driver for docker. We could see containers being part of the IaaS layer and being managed by the Cloud Orchestration layer rather than the PaaS implementation in some of the new PaaS implementations

This approach will make the containers integration with PaaS pluggable and not tied to a particular container implementation.

IX. Next Generation PaaS support

The usage of containers in PaaS is becoming mainstream, and the Linux Containers are becoming a defacto standard but have some way to go before widespread adoption. Some of the Cloud Providers are already using Containers to provide a fine-grained control over resource sharing.

Some of the features listed below would help improve adoption of containers in PaaS platforms

- **Standardization** It is very important to come up with a container file format - similar to OVF in VMsso that there is interoperability between container formats. This will also help in building an ecosystem of pre-built container images.
- **Security** Containers need to be more secure both from the perspective of filesystem, network and memory isolation. Most popular container implementations like LXC and Docker seem to be lacking in this regard.
- **OS Independence** Containers should have abstraction layer so that they are not tied to a particular Kernel or its userspace. Warden Container seems to be in the right direction in this regard, but we doubt it will get wide adoption beyond Cloud Foundry.

X. Conclusion

Containers have an inherent advantage over VMs because of Performance improvements and reduced startup time. There are multiple flavors of container implementations available, all of them open source. Each implementation having its own pros and cons. Common Linux containment principles like chroot or namespaces are being used by all of them. Docker adds additional layers on top on Linux Containers making it little different and perhaps most relevant for PaaS implementors. OpenVZ is perhaps the most secure though it needs a modified kernel. Since OpenVZ is working on bringing some of these features into Linux kernel, at some point in time OpenVZ and LXC will converge.

Only few PaaS implementations notably the ones which are open source or are new are using containers as opposed to older ones like GAE, Azure which do not use this concept.

Containers have a bright future specially in the PaaS use case provided there is more standardization and abstraction from the underlying kernel and Host OS.

References

[1] *Cloudfoundry warden* manages isolated, ephemeral, and resource controlled environments. https://github.com/cloudfoundry/warden.
[2] *LXC - Linux Containers*. http://linuxcontainers.org/.
[3] *Evading from linux containers*. http://blog.bofh.it/debian/id_413.
[4] *Openvz linux containers*. http://openvz.org/.
[5] *Docker : the linux container engine* http://www.docker.io/.
[6] *lmctfy - let me contain that for you* https://github.com/google/lmctfy.
[7] *Comparing LXC and OpenVZ* http://www.janoszen.com/2013/01/22/lxc-vs-openvz/.
[8] *Docker Security*. http://blog.docker.io/2013/08/containers-docker-how-secure-are-they/.
[9] *OpenVZ Overview* http://www.slideshare.net/kolyshkin/openvz-linux-containers