Scientific Journal of Impact Factor (SJIF): 4.14

e-ISSN (O): 2348-4470
p-ISSN (P): 2348-6406

International Journal of Advance Engineering and Research Development

Volume 3, Issue 2, February -2016

Docker Management Using Libvirt API

Amit Chahar[1], Ajay Shekhawat[2], Ankit Mishra[3], Komal Kumari[4], Prof. P.R. Sonawane[5]

[1,2,3,4,5]*Department of Computer Engineering, Army Institute of Technology, Pune, India*

Abstract—Docker automates the deployment of applications inside software containers by providing an additional layer of abstraction. Docker implements a high-level API to provide lightweight containers that run processes in isolation. It uses resource isolation features of the Linux kernel such as cgroups and kernel namespaces to allow independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines.

Libvirt is used by the most clouds to give user access to the cloud and has bindings in other languages also like java, python, ruby. It support LXC (linux containers) but does not support docker containers. Initially docker used LXC as driver but because of it being managed by open source community, docker could not rely on it. So, docker developed its own driver libcontainer in go language.

Docker is todays emerging technology but clouds can support docker only by using their own apis specially designed for docker since libvirt does not support libcontainer. This increases the complexity of the cloud. Proposed solution is to implement docker api in c and integrate it with the libvirt api. Thus, clouds will have to give access to only libvirt without using any special api for docker. On the other hand, the docker interface will be generic for all clouds, thus user does not have to face the difficulty while migrating from one cloud to another.

Keywords— libvirt; linux containers; docker containers; libcontainer; cloud computing

I. INTRODUCTION

Most commercial cloud computing systems, both services and cloud operating system software products use hypervisors. Enterprise VMware installations, which can rightly be called early private clouds, use the ESXi Hypervisor. Some public clouds (Terremark, Savvis, and Bluelock, for example) use ESXi as well. Both Rackspace and Amazon Web Services (AWS) use the XEN Hypervisor, which gained tremendous popularity because of its early open source inclusion with Linux. Because Linux has now shifted to support KVM, another open source alternative, KVM has found its way into more recently constructed clouds (such as ATT, HP, Comcast, and Orange). KVM is also a favorite hypervisor of the OpenStack project and is used in most OpenStack distributions (such as RedHat, Cloudscaling, Piston, and Nebula). Microsoft uses its Hyper-V hypervisor underneath both Microsoft Azure and Microsoft Private Cloud.

However, not all well-known public clouds use hypervisors. For example, Google, IBM/Softlayer, and Joyent are all examples of extremely successful public cloud platforms using containers, not VMs. Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-systemlevel virtualization on Linux. Docker uses resource isolation features of the Linux kernel such as cgroups and kernel namespaces to allow independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. Docker implements a high-level API to provide lightweight containers that run processes in isolation. Building on top of facilities provided by the Linux kernel (primarily cgroups and namespaces), a Docker container, unlike a virtual machine, does not require or include a separate operating system. Instead, it relies on the kernels functionality and uses resource isolation (CPU, memory, block I/O, network, etc.) and separate namespaces to isolate the applications view of the operating system. Docker accesses the Linux kernels virtualization features either directly using the libcontainer library, which is available since Docker 0.9, or indirectly via libvirt, LXC (Linux Containers) or systemd-nspawn.

The libvirt project develops a virtualization abstraction layer, which is able to manage a set of virtual machines across different hypervisors. The goals of libvirt are to provide a library that offers all necessary operations for hypervisor management without implementing functionalities, which are tailored to a specific virtualization solutions and which might not be of general interest. Additionally, the long-term stability of the libvirt API helps these management solutions to be isolated from changes of hypervisor APIs.

Most of the clouds use libvirt api for hypervisor management as libvirt gives a common interface to the user for all hypervisors. Since docker has stopped supporting LXC drivers because LXC drivers are maintained by open source community and therefore docker developers had to made frequent changes to the implementation. Libvirt only supports LXC drivers, therefore it is difficult to use docker on clouds. Also some clouds have separate apis for docker management. Therefore having separate apis of docker and libvirt are inconvient to both the cloud service provider and the user.

Contributions of this paper: Today, there is still no support by libvirt api, which is able to manage docker containers. Administrators are either restricted to use docker separately on the cloud or they have to develop their own proprietary management suites for docker containers. This project offers an API that is able to manage docker containers over a stable interface. This paper presents the first implementation that integrates the docker containers into the libvirt library. Besides presenting the architecture of our integration, we discuss issues discovered during the implementation as well as limitations of API mapping in the context of virtualization management.

II. RELATED WORK

Some trace inspiration for containers back to the Unix chroot command, which was introduced as part of Unix version 7 in 1979. In 1998, an extended version of chroot was implemented in FreeBSD and called jail. In 2004, the capability was improved and released with Solaris 10 as zones. By Solaris 11, a full-blown capability based on zones was completed and called containers[1]. By that time, other proprietary Unix vendors offered similar capabilities for example, HP-UX containers and IBM AIX workload partitions. VMs as the clouds core virtualization construct have been improved successively by addressing scheduling, packaging, and resource access (security) problems. VM instances acting as guests use large, isolated files on their hosts to store their entire file system and typically run a single, large process on the host[2][3]. Although security concerns are usually addressed through isolation, several limitations remain. Full guest OS images are required for each VM in addition to the binaries and libraries necessary for the applications. Full images create a space concern that translates into RAM and disk storage requirements and is slow on startup (booting might take from 1 to more than 10 minutes)[4].

As an example of OS virtualization advances, new Linux distributions provide kernel mechanisms such as namespaces and control groups to isolate processes on a shared OS, supported through the Linux Container (LXC) project. Namespace isolation allows groups of processes to be separated, preventing them from seeing resources in other groups. Container technologies use different namespaces for process isolation, network interfaces, access to interprocess communication, and mount points, and for isolating kernel and version identifiers.Control groups manage and limit resource access for process groups through limit enforcement, accounting, and isolation for example, by limiting the memory available to a specific container. This ensures that containers are good multitenant citizens on a host[5]. It also provides better isolation between possibly large numbers of isolated applications on a host. Control groups allow containers to share available hardware resources and, if required, the control groups can set up limits and constraints[6][7].

Figure 1 compares application deployment using a hypervisor and a container. As the figure shows, the hypervisor-based deployment is ideal when applications on the same cloud require different operating systems or OS versions (for example, RHEL Linux, Debian Linux, Ubuntu Linux, Windows 2000, Windows 2008, Windows 2012). The abstraction must be at the VM level to provide this capability of running different OS versions. With containers, applications share an OS (and, where appropriate, binaries and libraries), and as a result these deployments will be significantly smaller in size than hypervisor deployments, making it possible to store hundreds of containers on a physical host (versus a strictly limited number of VMs). Because containers use the host OS, restarting a container doesnt mean restarting or rebooting the OS[1][4].

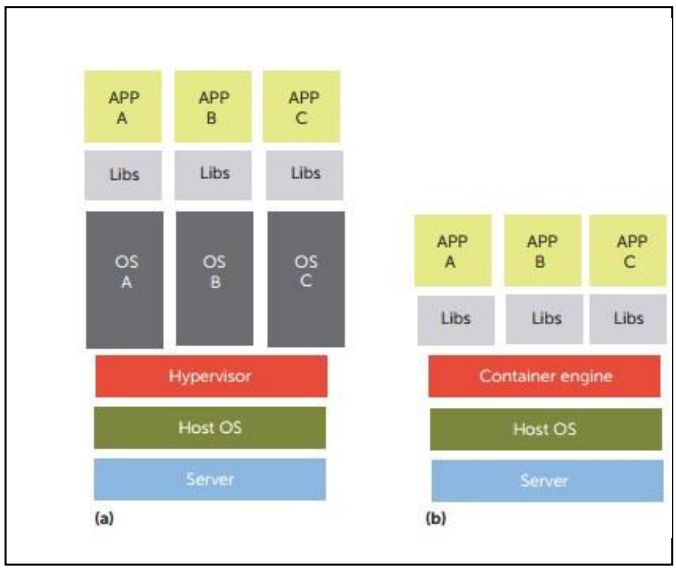

Fig. 1. Comparison of (a) hypervisor and (b) container-based deployments. A hypervisor-based deployment is ideal when applications on the same cloud require different operating systems or different OS versions; in containerbased systems, applications share an operating system, so these deployments can be significantly smaller in size.

III. DOCKER CONTAINERS

Docker is an open source project providing a systematic way to automate the faster deployment of Linux applications inside portable containers. Basically, Docker extends LXC with a kernel-and application-level API that together run processes in isolation: CPU, memory, I/O, network, and so on[8]. Docker also uses namespaces to completely isolate an applications view of the underlying operating environment, including process trees, network, user IDs, and file systems. Docker containers are created using base images. A Docker image can include just the OS fundamentals, or it can consist of a sophisticated prebuilt application stack ready for launch. When building images with Docker, each action taken (that is, command executed, such as apt-get install) forms a new layer on top of the previous one. Commands can be executed manually or automatically using Dockerfiles.

Each Dockerfile is a script composed of various commands (instructions) and arguments listed successively to automatically perform actions on a base image to create (or form) a new image. Theyre used to organize deployment artifacts and simplify the deployment process from start to finish. Containers can run on VMs too. If a cloud has the right native container runtime (such as some of the clouds mentioned) a container can run directly on the VM. If the cloud only supports hypervisor-based VMs, theres no problemthe entire application, container, and OS stack can be placed on a VM and run just like any other application to the OS stack[7][4]. In a traditional Linux boot, the kernel first mounts the root file system as read-only, then checks its integrity before switching the rootfs volume to read-write mode. Docker mounts the rootfs as read-only as in a traditional boot, but instead of changing the file system to read-write mode, it uses a union mount to add a writable file system on top of the read-only file system. There might be multiple read-only file systems stacked on top of each other. Using union mount, several file systems can be mounted on top of each other, which allows for creating new images by building on top of base images. Each of these file system layers is a separate image loaded by the container engine for execution. Only the top layer is writable. This is the container itself, which can have state and is executable. It can be thought of as a directory that contains everything needed for execution[5][6]. Containers can be made into stateless images (and reused in more complex builds), however.

IV. LIBVIRT API

The libvirt library is a Linux API over the virtualization capabilities of Linux that supports a variety of hypervisors, including Xen and KVM, as well as QEMU and some virtualization products for other operating systems. This article explores libvirt, its use, and its architecture.When it comes to scale-out computing (such as cloud computing), libvirt may be one of the most important libraries youve never heard of. Libvirt provides a hypervisor-agnostic API to securely manage guest operating systems running on a host. Libvirt isn't a tool per se but an API to build tools to manage guest operating systems. Libvirt itself is built on the idea of abstraction. It provides a common API for common functionality that the supported hypervisors implement. Libvirt was originally designed as a management API for Xen, but it has since been extended to support a number of hypervisors. Libvirt exists as a set of APIs designed to be used by a management application. Libvirt, through a hypervisor-specific mechanism, communicates with each available hypervisor to perform the API requests[9][10].

With libvirt, you have two distinct means of control. The first is demonstrated in figure 2, where the management application and domains exist on the same node. In this case, the management application works through libvirt to control the local domains. The other means of control exist when the management application and the domains are on separate nodes. In this case, remote communication is required. This mode uses a special daemon called libvirtd that runs on remote nodes. This daemon is started automatically when libvirt is installed on a new node and can automatically determine the local hypervisors and set up drivers for them. The management application communicates through the local libvirt to the remote libvirtd through a custom protocol. For QEMU, the protocol ends at the QEMU monitor. QEMU includes a monitor console that allows you to inspect a running guest operating system as well as control various aspects of the virtual machine (VM).

To support extensibility over a wide variety of hypervisors, libvirt implements a driver-based architecture, which allows a common API to service a large number of underlying hypervisors in a common fashion[3]. This means that certain specialized functionality of some hypervisors is not visible through the API. Additionally, some hypervisors may not implement all API functions, which are then defined as unsupported within the specific driver. Figure 2 illustrates the layering of the libvirt API and associated drivers. Note also here that libvirtd provides the means to access local domains from remote applications.

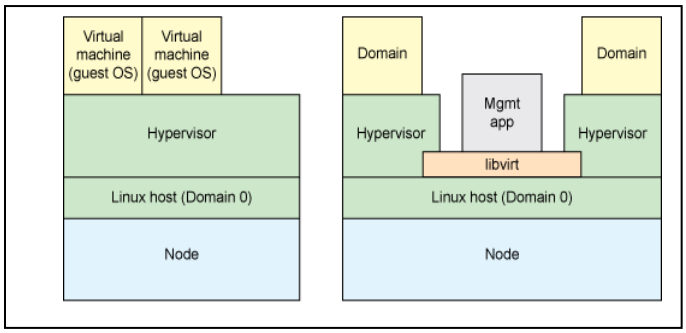

Fig. 2. Comparison and use model of libvirt

V. LIBVIRT EXTENSION FOR DOCKER

The scope of the libvirt API is intended to extend to all functions necessary for deployment and management of virtual machines. This entails management of both the core hypervisor functions and host resources that are required by virtual machines, such as networking, storage and PCI/USB devices. Most of the APIs exposed by libvirt have a pluggable internal backend, allowing support for different underlying virtualization technologies and operating systems. Thus, the extent of the functionality available from an particular API is determined by the specific hypervisor driver in use and the capabilities of the underlying virtualization technology.

A. Hypervisor Connections

A connection is the primary or top level object in the libvirt API. An instance of this object is required before attempting to use almost any of the APIs. A connection is associated with a particular hypervisor, which may be running locally on the same machine as the libvirt client application, or on a remote machine over the network. In all cases, the connection is represented with the *virConnectPtr* object and identified by a URI. The URI scheme and path defines the hypervisor to connect to, while the host part of the URI determines where it is located.

B. Guest Domains

A guest domain can refer to either a running virtual machine or a configuration that can be used to launch a virtual machine. The connection object provides APIs to enumerate the guest domains, create new guest domains and manage existing domains. A guest domain is represented with the *virDomainPtr* object and has a number of unique identifiers. Unique identifiers :

- ID: positive integer, unique amongst running guest domains on a single host. An inactive domain does not have an ID.
- name: short string, unique amongst all guest domains on a single host, both running and inactive.

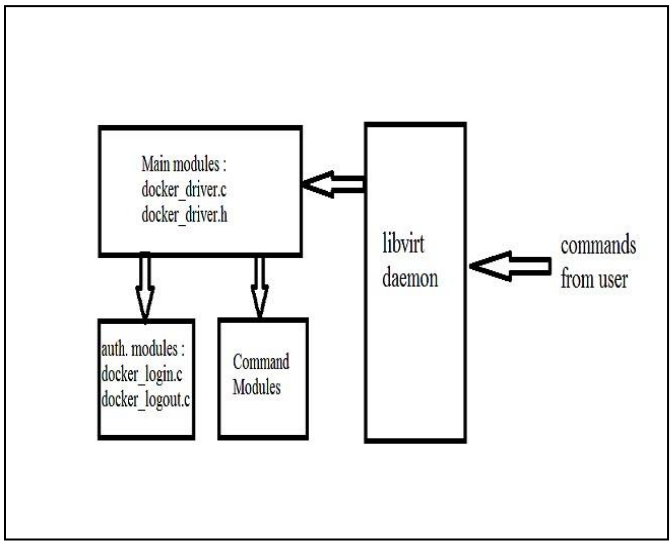

Fig. 3. Fig. 3. Connecting modules in docker and libvirt API. Commands are given to the libvirt daemon by the user. Libvirt daemon calls main modules. Main modules calls auth. modules or command modules according to the commands passed.

- UUID: 16 unsigned bytes, guaranteed to be unique amongst all guest domains on any host. RFC 4122 defines the format for UUIDs and provides a recommended algorithm for generating UUIDs with guaranteed uniqueness.

A guest domain may be transient or persistent. A transient guest domain can only be managed while it is running on the host. Once it is powered off, all trace of it will disappear. A persistent guest domain has its configuration maintained in a data store on the host by the hypervisor, in an implementation defined format. Thus when a persistent guest is powered off, it is still possible to manage its inactive configuration. A transient guest can be turned into a persistent guest while it is running by defining a configuration for it[9].

C. Integrating Docker API with Libvirt

The main modules *docker driver.h* and *docker driver.c* are implemented according to the specification given by the libvirt development guide. These modules contain the connection objects with the main api. *docker driver.h* contains the driver number which is used in the libvirt api to call the docker api. It is included in *libvirt.c* file to run the driver as daemon. In the configuration file of the libvirt, support for the docker package is added. *docker driver.c* is added to the make file for the compilation and the whole libvirt api is compiled again using make to integrate the docker api.

D. Module Development

The main modules are *docker driver.h* and *docker driver.c*. These are used to call the other modules according to the commands passed by the user. Modules *docker login.c* and *docker logout.c* authenticate the user with the docker engine. There can be two step authorization: one in libvirt and second after entering in the docker engine. This increases the security of the docker containers. Also, same credentials for both libvirt and docker engine can be used for ease. All the commands of the docker are implemented in the command modules. These modules contain all the commands and their options which can be passed from command line. Command moudules are called by the main modules according to the command passed by the user. Docker Remote API is used to execute the commands on the docker engine.

E. Platform Used

Platform used for the development of the extended libvirt api is ubuntu, linux distribution. Ubuntu supports both docker and libvirt API. Libvirt API works better on ubuntu as compared to the other linux distributions, since it was originally developed on ubuntu and tools used for the development of the libvirt api are already present in the basic installation whereas this can't be the case for other linux distributions. Docker API is developed in C language since original Libvirt API was also developed in C language. Also, using C language make the execution faster but other language binding are also present for the libvirt. Make tool of ubuntu is used for the compilation of the API.

VI. CONCLUSION AND FUTURE WORK

Docker container management is one of todays most important problems in data center management. Libvirt is the most prominent solution of an abstraction layer, being used by a number of management solutions and offering a stable interface to hypervisor and container management.

In this paper, Libvirt api is extended to support docker containers and libcontainer driver. The language used to implement the api was C. After updating the libvirt apis on clouds, no separate api is needed in order to support libcontainer driver and docker can be *installed* with libcontainer driver. Therefore, enhanced reliability and portability across clouds. Future work includes development of the C language api in other languages and integrate it with the libvirt api binding already present in that language. Also, this api can be extended to enable to run test suites for various virtualization environments. A graphical user interface can also be developed to simplify user interaction with the api.

ACKNOWLEDGMENT

We would like to express our deep gratitude to our Guide,Prof. P.R. Sonawane, Faculty of the Department of Computer Engineering, Army Institute of Technology and Mr. Mohsin Khazi, Calsoft Inc. for all the valuable guidance and intellectual stimuli that he provided during the progress of this work.It was a privilege to work under his valuable guidance and supervision.We are also grateful Prof. M.B. Lonare, Prof. Praveen Hore, Faculty of the Department of Computer Engineering for helping us from start,Prof. Sunil Dhore, HOD of the Department of Computer Engineering and other faculty members of Department of Computer Engineering for timely guidance and encouragement to complete this work.

REFERENCES

[1] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," Cloud Computing, IEEE, vol. 1, no. 3, pp. 81–84, Sept 2014.

[2] M. Bolte, M. Sievers, G. Birkenheuer, O. Niehorster, and A. Brinkmann, "Non-intrusive virtualization management using libvirt," in Design, Automation Test in Europe Conference Exhibition (DATE), 2010, March 2010, pp. 574–579.

[3] V. Danciu, N. Felde, M. Kasch, and M. Metzker, "Bottom-up harmonisation of management attributes describing hypervisors and virtual machines," in Systems and Virtualization Management (SVM), 2011 5th International DMTF Academic Alliance Workshop on, Oct 2011, pp. 1–10.

[4] A. Joy, "Performance comparison between linux containers and virtual machines," in Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in, March 2015, pp. 342–346.

[5] C. Pahl, "Containerization and the paas cloud," Cloud Computing, IEEE, vol. 2, no. 3, pp. 24–31, May 2015.

[6] L. Li, T. Tang, and W. Chou, "A rest service framework for fine-grained resource management in container-based cloud," in Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on, June 2015, pp. 645–652.

[7] T. Adufu, J. Choi, and Y. Kim, "Is container-based technology a winner for high performance scientific applications?" in Network Operations and Management Symposium (APNOMS), 2015 17th Asia-Pacific, Aug 2015, pp. 507–510.

[8] "Docker documentation," https://docs.docker.com/.

[9] "libvirt api documentation," https://libvirt.org/docs.html.

[10] "libvirt github repository," https://github.com/libvirt/libvirt.

[11] "Docker github repository," https://github.com/docker/docker.

[12] D. Liu and L. Zhao, "The research and implementation of cloud computing platform based on docker," in Wavelet Active Media Technology and Information Processing (ICCWAMTIP), 2014 11th International Computer Conference on, Dec 2014, pp. 475–478.

[13] L. Affetti, G. Bresciani, and S. Guinea, "adock: A cloud infrastructure experimentation environment based on open stack and docker," in Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on, June 2015, pp. 203–210.

[14] R. Zhang, M. Li, and D. Hildebrand, "Finding the big data sweet spot: Towards automatically recommending configurations for hadoop clusters on docker containers," in Cloud Engineering (IC2E), 2015 IEEE International Conference on, March 2015, pp. 365–368.

