Simulating a data center with a sharing System using a virtualization technique

Abhishek Kumar*, Elham Hojati, and Yong Chen,
Department of Computer Science, Texas Tech University, Lubbock, TX 79409, United States

Abstract*—There have been several technological novelties in the field of software development in the last decade. Cloud computing, Hadoop and NoSQL are some technologies that have gained widespread popularity and adoption. But in terms of long-term impact in the industry, these technologies are miles behind a relative newcomer – Docker. In this paper we have discussed in detail the virtualization techniques. Hypervisor-based virtualization is broadly classified into Type 1 bare metal hypervisor and Type 2 hosted hypervisor. Container based virtualization take advantages of small and light weight containers to run the applications. In our research, we build a multi node (3-node) Docker Swarm cluster and test it by creating an alpine Linux image from the shell of the leader node. We try to ping the Google DNS server and check whether our cluster has replicated the image successfully and all other privileges are granted to the manager and worker nodes. In the upcoming stage of our research we will be running a client application (RGB) on the head node and server application (DMTF/Redfish Mockup Server) on each of the agent node in our simulated data center.

***Keywords*—container-based virtualization, Docker, Docker Swarm, hypervisor, Linux containers, LXC, virtualization**

I. Introduction

VIRTUALIZATION is a technique to provide a virtual version of something. There are different virtualization techniques. Hypervisor, also sometimes called as a virtual machine monitor (VMM), is a type of virtualization. Hypervisor sits in between the guest operating system and the real physical hardware. Hypervisor controls the resource allocation to the guest operating system running on top of the physical hardware [4]. Container based virtualization is another type of virtualization technique, needed for the isolation of application's environment, resource allocation and sharing resources. Isolation using hypervisor-based approach comes with a huge overhead of managing the size of a VM. Container virtualization is done at the operating system level, rather than the hardware level. The main highlight of container virtualization is that each Linux container (LXC) or guest operating system shares the same kernel of the base system. Sharing in a virtualized environment (between guest operating system) is like sharing between independent systems, because virtualized hosts are not aware of each other, and the only method of sharing is traditional method of network of shared file system. As the container is sharing the kernel with the base system, one can see the processes that are running inside the container from the base system. However, inside the container, one will only be able to see its own processes.

II. Virtualization Techniques

Virtualization is a method of extending or switching an already existing interface to replicate the behavior of other system [4]. In simpler words, virtualization is a technique to provide a virtual version of something. In the 1960's, IBM introduced virtualization while developing CP/CMS (Control Program/Conversational Monitor System). Since then, CP/CMS has grown and is now used in the form of z/VM [38], a hypervisor for IBM System Z (mainframes) [5].

In our daily lives, virtualization and cloud computing are used interchangeably. Virtualization is certainly a component within cloud computing, but these two practices are not same. Cloud computing is separating the application from the underlying hardware. This is different from virtualization which separates the operating system from the underlying hardware. Thus, virtualization can be imagined as a subset of cloud computing. In the past, when we tried to install an operating system, it became inextricably linked to the hardware. For example, Windows Server 2008 R2 Standard is installed on top of a piece of hardware [8]. In such scenarios, migration of operating system was a tiring and tedious job. Also, if power supplies on such a physical server (or the processor, RAM) dies, the guest operating system dies. Virtualization allowed us to install a hypervisor that sits on top of this hardware. We now can easily install the operating system on the hypervisor. This operating system can be referred to as an instance of OS. The hypervisor is a software allowing the abstraction from the hardware [2]. The hardware running the hypervisor is called the host (and the operating system host operating system) whereas all emulated machines running inside them are referred to as guests and their operating systems are called guest operating systems [2].

With hypervisor coming into play, any instances of operating system can now be migrated from one machine (hypervisor 1 and physical server 1) to another machine (hypervisor 2 and

* Corresponding author.
E-mail address: abhishek.kumar@ttu.edu

This work was supported in part by the High-Performance Computing Center, Texas Tech University, Lubbock, TX 79409, United States.

physical server 2). A question that can arise in a reader's mind is why we really need virtualization. Firstly, it solves the major problem of migration. Once the operating system is no longer inextricable tied to the physical server, migrating these instances is just like migrating files or images. Secondly, a hypervisor helps us launch multiple instances (or guest operating systems). This can lead to major economic advantages at enterprise level where consolidating servers on individual machines is required. For example, organizations can purchase a single high-power server instead of ten cheap physical servers if they require to run suppose thirty guest operating systems on one virtual box. The hypervisor can be classified into two categories, type 1 and type 2 hypervisors.

A. *Type 1 Hypervisor*

Type 1 hypervisor run directly on hardware and is sometimes called as native or bare metal hypervisor. The diagram for this is show in Fig (mention figure number later). It has no underlying operating system and the hypervisor is installed directly on the server. For example, a Xen hypervisor (type 1) installed on these hardware and instances of guest operating systems are running on top of such hypervisor. On rebooting the system, one cannot see a proper management console in type 1 hypervisor and thus we see a blinking screen. This is usually because of the absence of additional console management tools. Such come as a service package with enterprise type 1 hypervisors. Some of the examples of Type 1 hypervisors are VMware vSphere/ESXi, Microsoft Windows Server 2012 Hyper-V (or the free Hyper-V Server 2012), Xen /Citrix XenServer, Red Hat Enterprise Virtualization (RHEV), KVM (or Kernel-Based Virtual Machine) etc. To install our required guest operating systems on type 1 hypervisor, we request access so that the management console software installed on a computer configures to the hypervisor. An important role of these management console software is that they allow guest operating systems to easily migrate based on current resource needs of physical servers. In enterprise level operations, organizations often face situations when during specific hours, a specific guest operating system is hit by a massive number of requests. Such situations are handled by these management console tools provided by respective vendors that move these instances on a different physical server. This technique can save companies millions of dollars on electricity consumption. We can often handle situations of fault tolerance with virtualization. Imagine a physical hardware machine that is running plenty of guest operating system on top of its hypervisor suddenly fails, such running instances within no time are migrated to servers that have sufficient resources. Type 1 hypervisors allow overallocation of random access memory to its guest operating system. A physical server with 16 GB of RAM can handle three instances of 12 GB, 10 GB and 8 GB (a total of 30 GB) with Type 1 hypervisor. Type 1 hypervisor intelligently moves the RAM based on the requirements of the individual guest operating system.

B. *Type 2 Hypervisor*

Type 2 or hosted hypervisors require a host operating system whose resources are used to perform their operations. The diagram for this is show in Fig (mention figure number later). What makes type 2 hypervisor easy to work with is that it does not require a management console software to configure and move the instances of an operating system. It can also make it easier to do things like monitoring or backups from the host OS. Type 2 hypervisors are the first choice if we want to run a small number of guest OS on a physical server. Thus, they have the required driver support. Examples of type 2 hypervisor include VMware Fusion, Oracle Virtual Box, Oracle VM for x86, Solaris Zones, Parallels and VMware Workstation. Type 2 hypervisors does come with some warnings. Firstly, overallocation of RAM to the guests is not permitted. For most part, if a guest OS requires a certain amount of RAM, the hypervisor automatically allots that from the host. For a host of 8GB that requires at least 4 GB to run, we cannot launch instances such that the sum of RAM of the guests turns out to be greater than 4GB. This would result in crashing of the host. Secondly, network traffic can sometimes affect the performance of the host. There may be a situation that after launching multiple instances, the web browser and file transfer works smoothly but the mailing services is having some issues. Thirdly, we must be sure that we have operating system add-ons with type 2 virtualization software such that operating system has performs efficiently.

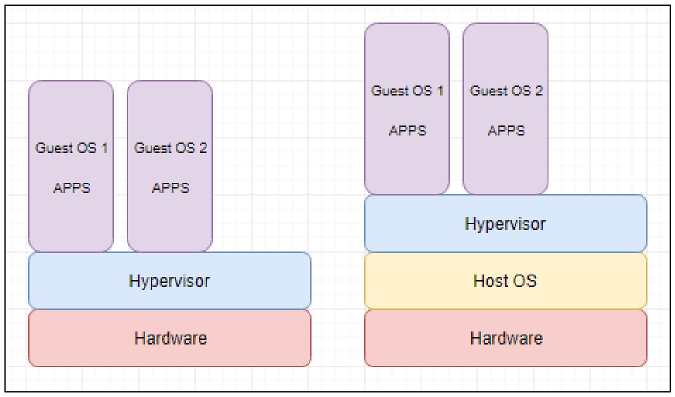

Fig 1: Type 1 Bare Metal Hypervisor and Type 2 Hosted Hypervisor

C. *Container-based virtualization*

Container-based virtualization does not emulate an entire computer. An operating system is providing certain features to the container software to isolate processes from other processes and containers [1]. A common feature in both bare metal and hosted virtualization is that they are virtualizing hardware resources. Containers on the other hand virtualize at operating system level, rather than the hardware level. The unique feature about container virtualization is that each container (or guest operating system) shares the same kernel of the base system. This is an advantage because as each container is sitting on top of the same kernel, and sharing most of the base operating system, containers are much smaller and lightweight compared to a virtualized guest operating system. As they are light weight an operating system can have many containers running on top of it, compared to the limited number of guest operating system that you can run.

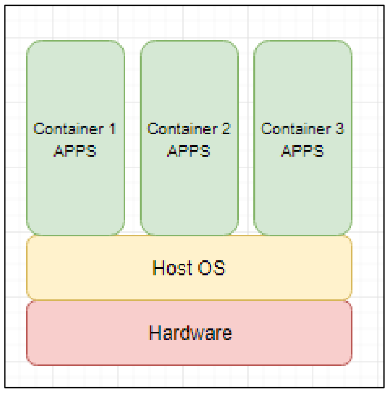

Fig 2: Container based virtualization

III. Docker

Linux containers [12] (LXC) is a project that aims to build a toolkit to isolate processes from other processes. LXC is required to avoid spending a lot of time reading the kernel documentation to understand and use the features to set up a container. For someone who wants to run processes in an isolated environment without spending too much time figuring out how the toolkit works, LXC is not a viable approach. This is when Docker comes into picture.

A. *Importance of Docker*

Docker was released as open source in March 2013 and since then has gained a lot of attention resulting in the rapid growth of the project. It got a lot of consideration from the open source community and major players in the enterprise level are adopting it in their upcoming releases. Docker solves one of the major issues that developers and system administrators have faced for years. Often something that works perfectly in development environment and quality assurance (QA) surprisingly fails to work in production environment. The problem could be that some packages are not properly installed or there is a version mismatch. Docker makes an image of the entire application, with all its dependencies, and ships it to the required target environment. In this way, an application that runs perfectly on a local machine will run anywhere as we have shipped the exact copy. If we try to solve this problem with hypervisor-based virtualization we must take an image of the entire virtual host and launch a new virtual instance from it. This is not an effective approach as one must go through the hassle of setting up a new host. Thus, being so light weight is a huge advantage of a Docker container.

B. *Linux kernel namespaces*

The Linux kernel provides a feature called namespaces that helps in isolation of each container. As the container is sharing the kernel with the base system, we can see the processes that are running inside the container from the base system. However, when we are inside the container, you will only be able to see its own processes. Without a virtual hardware, containers can provide a separated environment, like virtualization, where every container can run their own operating system by sharing the lone kernel. Each container has their own network stack and file system. In simple words, namespaces enable applications to run in their own isolated environments with separate process ids (PID), file system, network devices, user lists etc. If we look at the containers from the host, it appears to be a process, but inside the container it appears like a separate virtual machine (separate from the entire system). This is because we are in an entirely new namespace of its own. Since each of the container resides in its own namespace, we can have the same PID (say 123) inside container 1, 2, 3..., n. This is analogous to naming of files in a directory. One cannot have the same filename for different files in a single directory but, we can have same filename inside two separate directories.

IV. Methodology

We used play-with-docker lab to build our multi-node cluster. We use Docker Swarm to build and run the cluster.

V. Test And Results

How far can we scale out the data center simulation? Although it depends on different aspects of the environment that the simulation runs on, But the quick answer to that question is that for a data center simulator with hundreds of emulated equipment you need single-node version and for a data center simulator with thousands of emulated equipment you need multi-nodes version.

A. *Single-Node Data Center Simulator*

In a single-node data center simulator there is only one machine (no matter a VM or bare metal) that runs Docker Swarm and is called the Swarm manager. The Swarm manager is not only responsible for Swarm cluster management and orchestration, but also is the Swarm node that all the containers run on.

```
[ C:\Users\Abhishek] # docker machine
create node1
Running pre-create checks…
(node1) Default Boot2Docker ISO is out-of-
date, downloading the latest release…
Creating machine…
Waiting for machine to be running, this
make take a few minutes…
Checking connection to Docker…
Docker is up and running!
To see how to connect your Docker Client to
the Docker Engine running on this virtual
machine, run: docker-machine env node1
```

Fig 3: Starting node 1 in swarm cluster

We can enter the node 1 in the swarm cluster using either docker-machine env or docker-machine ssh.

```
[ C:\Users\Abhishek] # docker-machine ssh
node1
            ##         .
              ## ## ##        ==
           ## ## ## ##       ===
       /"""""""""""""""""\___/ ===
  ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
       \______ o          __/
         \    \        __/
          \____\______/

Boot2Docker version 18.03.1-ce, build HEAD:
cb77972
Docker version 18.03/1-ce, build 9ee9f40
docker@node1: ~%$
```

Fig 4: Entering node 1 in swarm cluster

B. *Multi-Nodes Data Center Simulator*

In the Multi-nodes data center simulator, Figure 4.7, except from the Swarm manager node, there are also some other machines that are parts of the Swarm cluster and are called Swarm worker nodes. The Swarm manager node is only responsible for managing the Swarm cluster and the orchestration and the Swarm worker nodes are the nodes that the containers (emulated equipment) run on. The Swarm manager node can run the containers as well, but we decided not to do that for decreasing its load. One of the advantages of using Multi-nodes data center simulator is that if you need to scale out the simulator up to a certain level that by the current number of the Swarm worker nodes you are not able to achieve, you can easily add more Swarm worker nodes to your cluster and increase its capacity. The host option that we choose for our research was play-with-docker.com. Figure 5 shows how to initialize a leader node.

```
[node 1] (local) root@192.168.0.33 ~
$ docker swarm init –advertise-addr
192.168.0.33
Swarm initialized: current node
(yhm8u85r38vb41dp2a5kay6qi) is now a
manager.

To add a worker to this swarm, run the
following command:
  docker swarm join –token SWMTKN-1-
28y0yhyw923pcfvgv2ba0i0e14p05532g9tpiou79vh
kgcq11d-5or9p5bw9mzzyhwhim679vjob
192.168.0.33: 2377

To add a manager to this swarm, run ‘docker
swarm join-token manager’ and follow the
instructions.
```

Fig 5: Initializing swarm on node 1

The worker nodes are not privileged to access the swarm command. This can be shown in Fig 6. The leader node grants permission to node 2 to join the cluster as a worker. This is shown in figure 7.

```
[node 2] (local) root@192.168.0.32 ~
$ docker node ls

Error response from daemon: This node is
not a swarm manager. Worker nodes can’t be
used to view or modify cluster state.
Please run this command on a manger node or
promote the current node to a manger.
```

Fig 6: Workers are not privileged to access on manger mode

```
[node 2] (local) root@192.168.0.32 ~
$ docker swarm join –token SWMTKN-1-
28y0yhyw923pcfvgv2ba0i0e14p05532g9tpiou79vh
kgcq11d-5or9p5bw9mzzyhwhim679vjob
192.168.0.33: 2377

This node joined a swarm as a worker.
```

Fig 7: Node 2 joined the swarm cluster

We update the role of the node 2 as manager using the privileges of node 1. This can be shown in the figure 8. Using the docker node ls command we can see the status of our nodes.

```
[node 1] (local) root@192.168.0.32 ~
$ docker node update –role manager node2
node2
[node 1] (local) root@192.168.0.32 ~
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
yhm	node1	Ready	Active	Leader
iaa	node2	Ready	Active	Reachable

Fig 8: Updating role of node 2 as a manager

Next, we generate the token for node 3 to join the cluster from node 1 as shown in figure 9.

```
[node 1] (local) root@192.168.0.33 ~
$ docker swarm join-token manager

To add a manager to this swarm, run the
following command:
  docker swarm join –token SWMTKN-1-
28y0yhyw923pcfvgv2ba0i0e14p05532g9tpiou79vh
kgcq11d-2ydi8kz03hejrcsm8f58hwf3a
192.168.0.33: 2377
```

Fig 9: Generating manager token for node 3

This key is different from the command to add a worker in Fig 5. We access manager privileges to node 3 as shown in the figure 10.

```
[node 3] (local) root@192.168.0.31 ~
$ docker swarm join –token SWMTKN-1-
28y0yhyw923pcfvgv2ba0i0e14p05532g9tpiou79vh
kgcq11d-2ydi8kz03hejrcsm8f58hwf3a

This node joined a swarm as a manager.
```

Fig 10: Node 3 joins the cluster

We can test the 3-Node Swarm cluster by creating an alpine Linux image from the shell of the leader node. Here, we try to ping the Google DNS server. Fig11 shows that the service is up and running for all the three nodes.

```
[node 1] (local) root@192.168.0.33 ~
$ docker service create –replicas 3 alpine
ping 8.8.8.8
Il5hdyu9nt9ds1c2boyqmarze
Overall progress: 3out of 3 tasks
1/3: running
2/3: running
3/3: running
Verify: Service converged
[node 1] (local) root@192.168.0.33 ~
$ docker service ls
```

ID	NAME	MODE	REPLICAS	IMAGE
il5	romantic_ elion	replicated	3/3	alpine: latest

```
[node 1] (local) root@192.168.0.33 ~
$ docker service ps romatic_elion
```

ID	NAME	IMAGE	NODE	CURRENT STATE
zae	romantic_ elion.1	alpine: latest	node3	Running
dcf	romantic_ elion.2	alpine: latest	node1	Running
5ff	romantic_ elion.3	alpine: latest	node2	Running

Fig 11: Creating replicas of alpine Linux image and pinging the Google DNS

In figure 12, we perform a lookup on our node to see how all the servers are currently behaving.

.

```
[node 1] (local) root@192.168.0.32 ~
$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
yhm	node1	Ready	Active	Leader
iaa	node2	Ready	Active	Reachable
f7t	node3	Ready	Active	Reachable

Fig 12: Checking swarm status

VI. Conclusion

Our multi-node Docker Swarm cluster was successfully built. This cluster contains 3 nodes. Node 1 is the manager node with all the privileges where as Node 2 is the worker node. Node 1 has granted manager privileges to Node 3. We build a multi node (3-node) Docker Swarm cluster and tested it by creating an alpine Linux image from the shell of the leader node. We try to ping the Google DNS server and check whether our cluster has replicated the image and all the privileges are granted to the manager and worker nodes. In the upcoming stage of our research we will be running a client application on the head node and server application on each of the agent node in our simulated data center.

References

[1] C. Boettiger, "An introduction to Docker for reproducible research" ACM SIGOPS Operating Systems Review - Special Issue on Repeatability and Sharing of Experimental Artifacts archive, Volume 49 Issue 1, January 2015.

[2] M. Eder, "Hypervisor vs Container-based Virtualization," Seminar Future Internet WS2015/16 Lehrstuhl Netzarchitekturen und Netzdienste Fakultät für Informatik, Technische Universität München

[3] Pillai, S. (2014, October 21). Difference between Hypervisor Virtualization and Container Virtualization. Retrieved from https://www.slashroot.in/difference-between-hypervisor-virtualization-and-container-virtualization

[4] A. S. Tanenbaum and M. V. Steen, "Distributed Systems: Principles and Paradigms," in *2nd* ed. Pearson Prentice Hall, 2016.

[5] A. Sampathkumar, "Virtualizing Intelligent River: A comparative study of Alternative Virtualization Technologies", Computer Sciences Commons.

[6] A. A. Nosrati, "On Simulating Data Centers with Redfish^{TM}–Enabled Equipment", Retrieved from https://ttu-ir.tdl.org/ttu-ir/handle/2346/521

[7] M. J. Scheepwrs , "Virtualization and Containerization of Application Infrastructure: A Comparison" 21st Twente Student Conference on IT June 23rd, 2014, Enschede, The Netherlands. Copyright 2014, University of Twente, Faculty of Electrical Engineering.

[8] Docker project homepage, https://www.docker.com/, Retrieved: September 13, 2015

[9] KVM project homepage, http://www.linux-kvm.org/ page/Main_Page, Retrieved: September 16, 2015

[10] XEN project homepage, http://www.xenproject.org/, Retrieved: September 16, 2015

[11] Microsoft TechNet Hyper-V overview, https://technet.microsoft.com/en-us/library/hh831531.aspx, Retrieved: September 16, 2015

[12] VMware Homepage, http://www.vmware.com/, Retrieved: September 16, 2015

[13] VirtualBox project homepage, https://www.virtualbox.org/, Retrieved: September 16, 2015