ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

In

COMPUTER NETWORKS M

Containerization in Cloud Computing: performance analysis of virtualization architectures

CANDIDATO
Amedeo Palopoli

RELATORE:
Chiar.mo Prof. Ing. Antonio Corradi

CORRELATORI:
Prof. Ing. Luca Foschini
Ing. Filippo Bosi
Dott. Ing. Stefano Monti

Anno Accademico 2016/17
Sessione II

Acknowledgements

- First and foremost, I would like to thank my supervisors at University of Bologna, Prof. Antonio Corradi and Prof. Luca Foschini, for this fantastic opportunity to work with them and develop this interesting thesis.

- I would also like to thank everybody affiliated with Imola Informatica S.p.A who directly or indirectly contributed in this work. In particular, I would like to thank Ing. Filippo Bosi who followed me with dedication. Surely, this experience has been very fundamental for my professional and educational career and so, I cannot forget everyone that gave me the possibility to work in this interesting project.

- Moreover, I have to thank everyone who allowed me to realize this little dream. In particular, my parents Carmine and Virginia. Without their support, I could never have completed this important step of my life.

- I consider myself a lucky person, because I am surrounded of special people. Of course, one of them is Maria, my girlfriend. You have always been present in every moment, and maybe thanking you is not enough…

- Lastly, but not the least, I would like to thank my brother Francesco and my grandmother Annuzza. I am grateful for the various pieces of good advice which have been given that, of course, helped me to grow up easier.

Abstract

La crescente adozione del cloud è fortemente influenzata dall'emergere di tecnologie che mirano a migliorare i processi di sviluppo e deployment di applicazioni di livello enterprise. L'obiettivo di questa tesi è analizzare una di queste soluzioni, chiamata "containerization" e di valutare nel dettaglio come questa tecnologia possa essere adottata in infrastrutture cloud in alternativa a soluzioni complementari come le macchine virtuali. Fino ad oggi, il modello tradizionale "virtual machine" è stata la soluzione predominante nel mercato. L'importante differenza architetturale che i container offrono ha portato questa tecnologia ad una rapida adozione poichè migliora di molto la gestione delle risorse, la loro condivisione e garantisce significativi miglioramenti in termini di provisioning delle singole istanze. Nella tesi, verrà esaminata la "containerization" sia dal punto di vista infrastrutturale che applicativo. Per quanto riguarda il primo aspetto, verranno analizzate le performances confrontando LXD, Docker e KVM, come hypervisor dell'infrastruttura cloud OpenStack, mentre il secondo punto concerne lo sviluppo di applicazioni di livello enterprise che devono essere installate su un insieme di server distribuiti. In tal caso, abbiamo bisogno di servizi di alto livello, come l'orchestrazione. Pertanto, verranno confrontate le performances delle seguenti soluzioni: Kubernetes, Docker Swarm, Apache Mesos e Cattle.

Abstract

The increasing adoption of cloud computing is strongly influenced by emerging of technologies whose aim is to improve the development and deployment processes of enterprise applications. The goal of this thesis is to investigate one of these solutions, called “containerization”, and deeply analyze how this solution can be included in cloud infrastructures as an alternative to complementary solutions like virtual machines. So far, the virtual machine model has been the predominant solution. The key differentiator nature that containers offer has stimulated an increasing adoption of this technology because improves resource management, resource sharing, and it guarantees substantial improvements regarding provisioning time of single instances. In this work, we will analyze the containerization paradigm from both infrastructure and application point of views. For the first one, we will investigate the performances by comparing LXD, Docker, and KVM, as hypervisor of OpenStack cloud infrastructure, while the second one concerns the development of enterprise applications that are distributed over a set of server hosts. In this case, we need to exploit high-level services such as orchestration. Therefore, we will compare the performances of the following container orchestrators: Kubernetes, Docker Swarm, Apache Mesos, and Cattle.

Summary

Introduction

Cloud computing is a paradigm where a large pool of computers is connected in private or public networks to provide resources or applications dynamically. These are available on the Internet, and five essential features characterize them: on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured services. In this way, consumers can focus on their core business without the need to buy or maintain the IT systems.

Virtualization is the core technology of cloud computing since it is the enabling solution allowing us to concurrently run multiple operating systems on the same server, thereby providing for efficient resource utilization and reducing costs. More precisely, this is called server virtualization, and the "virtual machine" model is the most known type of virtualization that is involved in cloud deployments. Server Virtualization introduces overhead to emulate the underlying hardware and load an entire operating system for each server instance and, as the demand for cloud computing increases day after day, the interest in traditional technologies is decreasing.

At the same time is increasing the demand for new virtualization technology as "containerization" or "lightweight virtualization". Unlike the existing virtualization approach, containerization does not involve to load a complete operating system by reducing, in this way, system overload. In fact, as opposed to traditional hypervisors, containers share the underlying kernel used by the operating system running the host machine. Furthermore, this also offers most of the benefits that are provided with the traditional virtual machine model.

The thesis was developed at Imola Informatica S.p.A. in Imola (Bo), a consulting and skill transfer company that works alongside customers, in management and development of mission critical projects. The objective of this work is to evaluate the adoption of containerization as cloud hypervisor in opposition to the traditional virtual machine model. In particular, we will investigate both cloud solutions in order to understand what is more suitable and particularly when is more appropriate to use containerization instead of machine virtualization in a cloud environment.

The first chapter provides an overview of virtualization technologies, how they are adopted in the IT industry and a comparison between containerization and traditional virtualization model.

The second chapter focuses on “container management”. This consists of providing users the possibility to deal with containers regardless to explore low-level details that are exploited through an independent API.

The third chapter concerns the usage of containers in cluster deployments by introducing a new higher level: orchestration. This is necessary to offer users other features that are architecture-dependent.

The fourth chapter is based on the highest level of abstraction: container-focused operating systems. This is the possibility to build a platform which provides services on a distributed architecture by exploiting the containerization paradigm.

The fifth chapter is dedicated to analyzing a cloud infrastructure solution that is OpenStack. This is introduced since it is the platform where this analysis will be performed on.

The sixth chapter is aimed to show the experimental results that addressed the containerization paradigm in cloud deployments and concern both infrastructure and application level.

The last chapter describes a case study that aims to provide a cloud-based service in order to evaluate the behavior of two types of server virtualization from the applicative point of view.

1 Virtualization and Containerization

1.1 Overview

Innovation is necessary to ride the inevitable tide of change and, for this reason, most of the enterprises are striving to reduce their computing cost. The most important answer to the demand of reducing the computing cost is the introduction of Cloud Computing [1]. This is rapidly becoming the standard for hosting and running software applications on the Internet. Cloud computing is not a new technology but an IT paradigm which offers the reduction of upfront capitalizations, rapid scalability, and ubiquitous accessibility. Cloud computing has been designed through the evolution of other existing solutions such as virtualization. This is one of the fundamental technologies that made up the cloud paradigm. Due to this new model, nowadays, consumers are able to concentrate more on the core application functionalities instead of focusing on not business aspects.

The term virtualization broadly describes the separation of a service request from the underlying physical delivery of that service [2]. Therefore, a new layer is included between client and service provider which is able to implement operation requests regardless the underlying physical infrastructure.

Modern application infrastructure techniques and methodologies incentivize an accelerated adoption of cloud computing technologies as well as various virtualization technologies. The most popular virtualization technique is about server virtualization, in which the virtualization layer allows multiple operating system instances to run concurrently on a single computer dynamically partitioning and sharing the available physical resources such as CPU, storage, memory and input/output devices.

However, the virtual machine is not the unique model solution used for cloud deployments. An alternative solution to machine virtualization is the containerization. This technology involves encapsulating an application in a container with its own operating environment and includes many of the benefits of using virtual machines as running on everything, from physical computers to virtual machines, bare-metal servers, private cloud clusters, public instances and more. Therefore, this chapter will investigate the state of the art of virtualization technologies.

1.2 Virtualization in cloud computing

In computing, virtualization means creating a virtual representation of a resource, such as a server, storage device, network or also an operating system where the middleware divides the resource into one or more execution environments [3]. Something as simple as partitioning a hard drive is considered virtualization because it is possible to take one storage device and partition it in order to create two different hard drives. This does not impact the behavior of devices, applications and human users which are able to interact with the virtual resources as if they were single logical resources. Even if server virtualization is the most known type, nowadays the term involves a number of computing technologies such as storage virtualization, operating system-level virtualization, network virtualization.

1.2.1 Storage virtualization

Companies are increasingly producing so much information that now IT figures are coming up with different approaches to consolidate their systems. Therefore, the increasing amount of data involves the increasing amount of storage devices in order not to lose data. However, managing all those devices can soon become difficult and of course is needed a solution to make easier the management of the storage service level. A new way to combine drives into one centrally manageable resource is possible due to the introduction of storage virtualization. Figure 1 shows a storage system which can be described as a storage array.

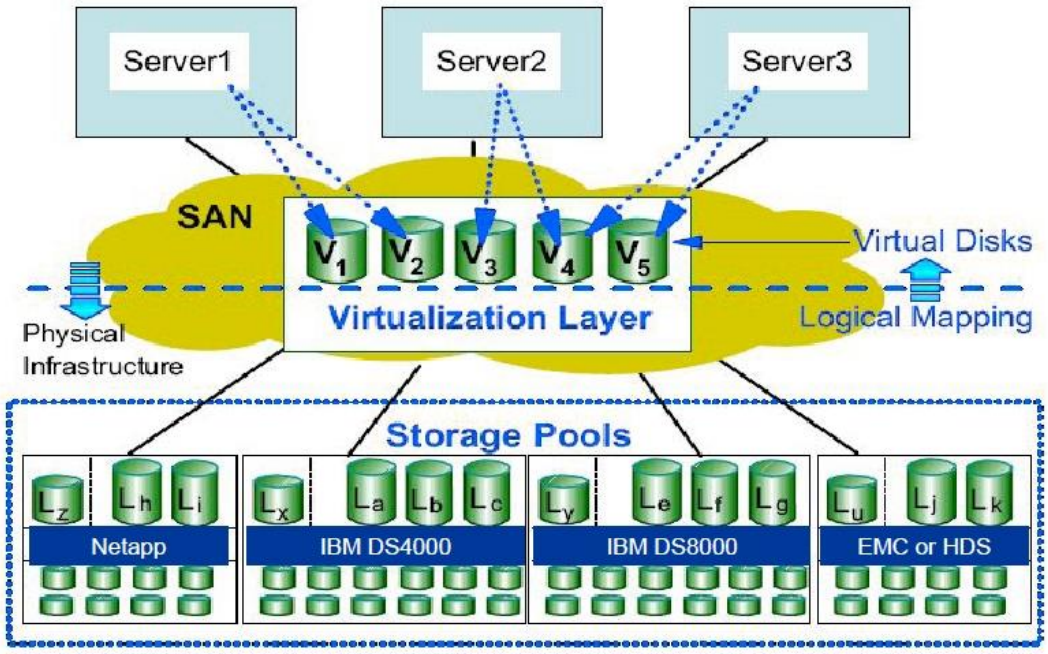

Figure 1 - Storage virtualization system

Typically, it is used special hardware and software along disk drives in order to provide fast and reliable storage for computing and data processing. This enables better functionalities and more advanced features in computer data storage systems, taking advantages as well as with server virtualization technologies.

Two storage types are provided through storage systems: block and file storage. Block level storage can be seen as the hard drive on a server but with the need to be installed in a remote chassis [4]. With this storage type, raw storage volumes are created, and then a server-based operating system connects to these volumes using them as individual hard drives. This makes block-level storage usable for almost any kind of application, including file storage, database storage, virtual machine file system (VMFS) volumes, and more. However, with block storage access, it is usual for an organization to be able to use operating system native backup tools or third-party backup tools such as Data Protection Manager(DPM) to back up files.

Block level storage is extremely flexible, but nothing beats the simplicity of file-level storage when all that is needed is a place to dump raw files. This consists of a solution with centralization, high-availability, and an accessible place to store files and folders. Usually, Network Attached Storage (NAS) devices are used in order to provide a lot of space at what is generally a lower cost than block level storage.

To conclude the comparison between the two storage types we can summarize that in the block level world, it is necessary to create a volume, deploy an OS, and then make an attachment to the created volume whereas, at the file level, the storage device handles the files and folders on the device.

1.2.2 Network virtualization

The introduction of cloud computing has put the focus on server virtualization even if there were other critical implications of that technology. In fact, the ability to create virtual environments means that we move resources within the cloud infrastructure. This elasticity and mobility have several implications for how network services are defined, managed and used to provide cloud services. Therefore, there are not just servers which benefit from virtualization but there also advantages that are necessary from the network point of view. This challenge was addressed due to the introduction of network virtualization.

Network virtualization is a process of abstraction which separates the logical network behavior from the underlying physical network resources [5]. This allows network aggregation and provisioning, combining different physical networks into a single virtual network, or breaking a physical network into multiple virtual networks that are isolated from each other.

As enterprises had moved from traditional server deployments to virtualized environments, the network issue was to provide security and segregation of sensitive data and applications. The solution to those requirements was to build the so-called multi-tenancy networking [6].

Multi-tenant networks are data center networks that are broken up and logically divided into smaller, isolated networks. They share the physical networking gear but operate on their own network without any visibility into the other logical networks. Often this requirement comes from business processes of the organization or from federal regulatory in which there is the need to isolate and control parts of the network system.

Network virtualization lends itself to cost savings, efficiency, security, and flexibility. In a virtual environment, logical switch ports are created and abstracted from the underlying physical ports. This allows more virtual switch ports to be added and connected to other switch ports quickly without having to commit real ports or cable them together in the data center. Figure 2 shows a typical network system used in cloud solutions.

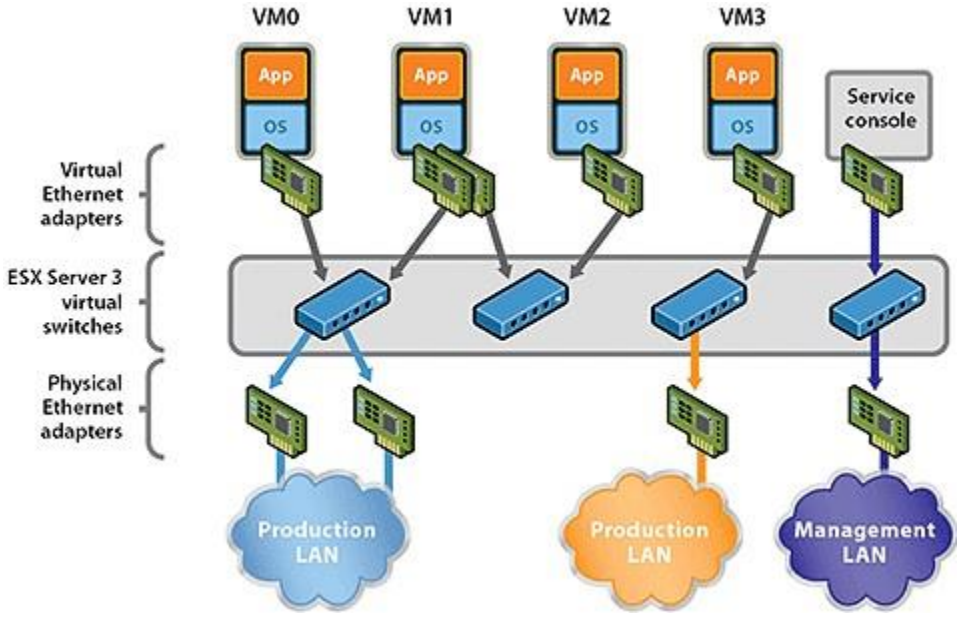

Figure 2 - Network virtualization system

It consists of using a layered architectural pattern in which the bottom level includes the physical devices used to communicate as well as in the classical network design. In the middle, we have software components that provide middleware services, in order to create multiplexer-functionalities to the top layer, the virtual network adapters.

Network virtualization can be split up into external and internal network virtualization. External network virtualization combines or subdivides one or more local area networks (LANs) into virtual networks to improve the efficiency of a large data center network. To do that, the key components are virtual local area network (VLAN) and network switches. This allows for a system administrator the possibility to configure systems physically attached to the same local network into separate virtual networks. Conversely, an administrator can combine systems on separate local area networks into a single VLAN spanning segments of a large network.

Internal network virtualization configures a single system with software containers. As it will be explained later, this approach consists of a method in which the kernel of an operating system allows the existence of multiple isolated user-space instances, instead of just one. Such instances, which are sometimes called containers, may look like real computers from the point of view of programs running in them.

With the usage of internal network solutions, a physical network can be emulated by software implementations. This can improve the efficiency of a single system by isolating applications to separate containers or pseudo-interfaces.

1.2.3 Server virtualization

The most common form of virtualization is the virtual machine (VM) [7]. A VM is an emulation of a computer system, and it is based on computer architectures providing functionalities of a physical computer. The implementation may involve specialized hardware, software, or a combination. With this improvement, a physical machine could spawn more virtual machines for each physical computer system. This allows better machine consolidation and augmented security. In fact, if the service becomes unstable or compromised, the other services on the physical host will not be affected.

Server virtualization hides server resources, including the number and identity of individual physical servers, processors, and operating systems, from server users

[8]. The server administrator uses a software application to divide one physical server into multiple isolated virtual environments.

In order to create, run and control VMs, a new software layer is required between the application and the hardware which is called hypervisor. The hypervisor presents the guest operating systems with a virtual operating platform and manages the execution of the guest operating systems. This software component uses a thin layer of code in software or firmware to allocate resources in real time. It is even the traffic cop that controls input/output and memory management.

As shown in Figure 3, numerous studies have classified the hypervisor solution with two models that are called respectively “Type One” and “Type Two”.

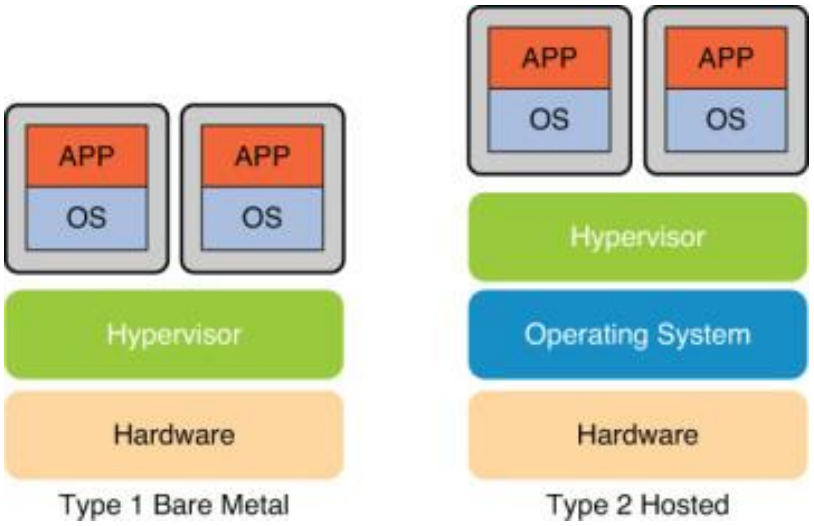

Figure 3 - Hypervisor Types

Figure 3 represents the architectural model of the two types of server virtualization implementations. The first is the solution in which the hypervisor software runs directly on the system hardware. It is often referred to as a “native”, “bare metal” or “embedded” hypervisor in vendor literature. On the contrary, type 2 hypervisors run on a host operating system. A Type One hypervisor provides better performance and greater flexibility because it operates as a thin layer designed to expose hardware resources to virtual machines, reducing the overhead required to run the hypervisor itself.

1.3 Types of virtualization with virtual machine

Virtualization is the technology that is rapidly transforming the IT landscape. It guarantees better hardware utilization, saves energy and makes it possible to concurrently run multiple applications and various operating systems on the same physical server. This increases the utilization, efficiency, and flexibility of

existing computer hardware. As anticipated in the previous section, there is a software that makes possible the virtualization of the server machine. This software is called hypervisor, and it is placed between the hardware and the operating system. Its role is to decide the access that applications and operating systems have with the processor and other hardware resources.

Partitioning a physical server into smaller virtual servers help to maximize resources but there are also some complexities which are included due to the competition with guests operating systems. By definition, the operating system is the unique component that is placed between the underlying hardware and the applications running on top. With the introduction of server virtualization, there is also the hypervisor which needs to be placed between the hardware and the operating system guests that share the underlying infrastructure.

One of the most important purposes of virtualization is to mask the service request from the physical implementation. This means that the operating systems, which are installed on top of the hypervisor, should be kept as well as they are running on top of physical resources. However, the mediation between the hypervisor and the operating system requires the introduction of an implementation which cannot be completely transparent. Therefore, the virtualization technologies that have emerged can be split up into three categories: full-virtualization, hardware-assisted virtualization, and para-virtualization.

1.3.1 Full virtualization

Full virtualization consists of an approach that relies on binary translation to trap, into the virtual machine monitor (hypervisor), and to virtualize certain sensitive and non-virtualizable instructions with new sequences of instructions that have the intended effect on the virtual hardware [9]. This guarantees the total isolation of guest operating systems from their hosts.

Meanwhile, the user-level code is directly executed on the processor for high performance, privileged commands which come from the guest operating system are trapped by the hypervisor because it is the unique component able to execute those operations.

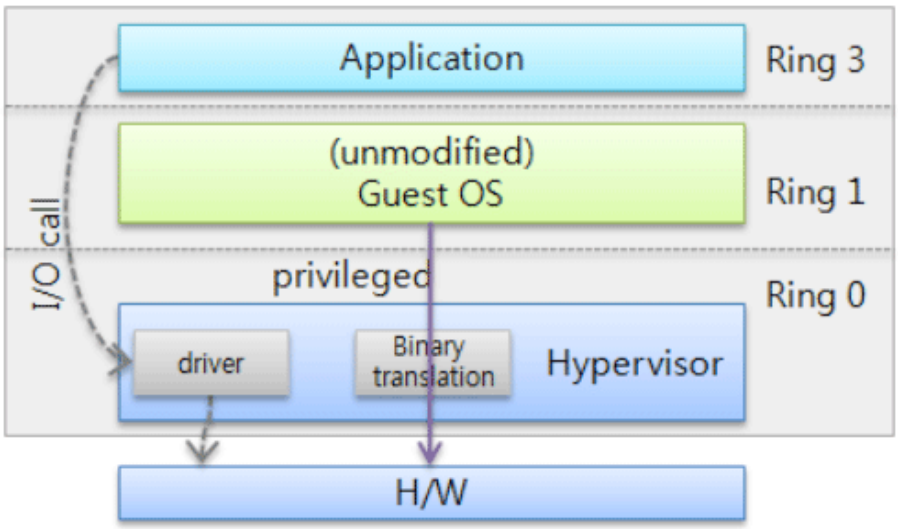

Figure 4 - Ring levels with full virtualization

Figure 4 shows us the representation of the ring level executing model with full virtualization. A variety of operating systems can run on the hypervisor without any modification, but the speed is somewhat low due to the machine code conversion process.

1.3.2 Hardware-assisted virtualization

One of the core elements of first-generation hardware-assisted virtualization was the introduction in the x86 CPU ring architecture, known as Ring -1. This allows hypervisors to run at Ring -1 in order to execute guest instructions at Ring 0, just as they normally would when running on a physical host [10]. Figure 5 shows an architecture that supports hardware-assisted virtualization.

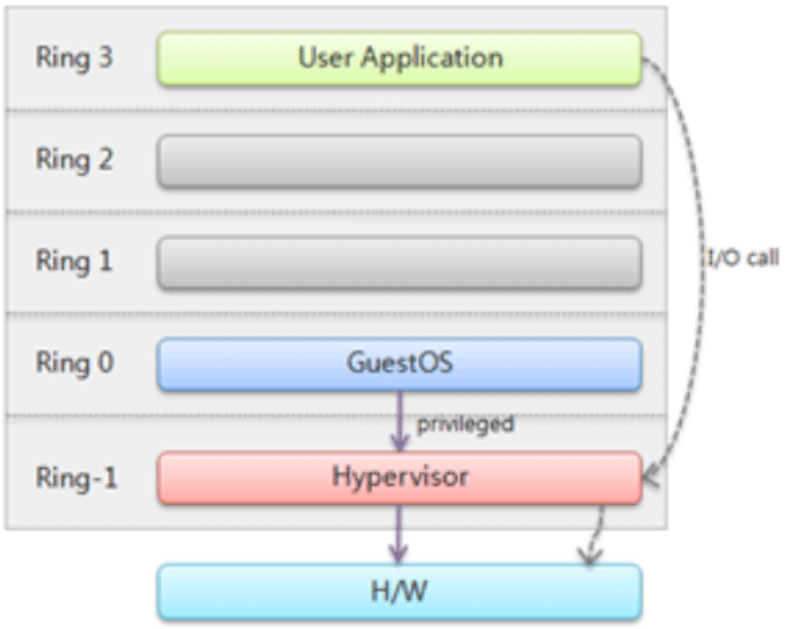

Figure 5 - Ring levels with Hardware-assisted virtualization

It provides a Ring-1 level, and the hypervisor runs on Ring-1 while the operating system runs on Ring 0. This does not require the process of binary translation for privileged commands, and a command is executed directly by hardware via the hypervisor with a notable performance improvement.

1.3.3 Paravirtualization

Paravirtualization is an enhancement of virtualization technology in which a guest operating system is recompiled before to install it inside a virtual machine [11]. It creates an interface layer to the guest operating systems that can differ somewhat from that of the underlying hardware. This capacity minimizes overhead and optimizes system performance by supporting the use of virtual machines.

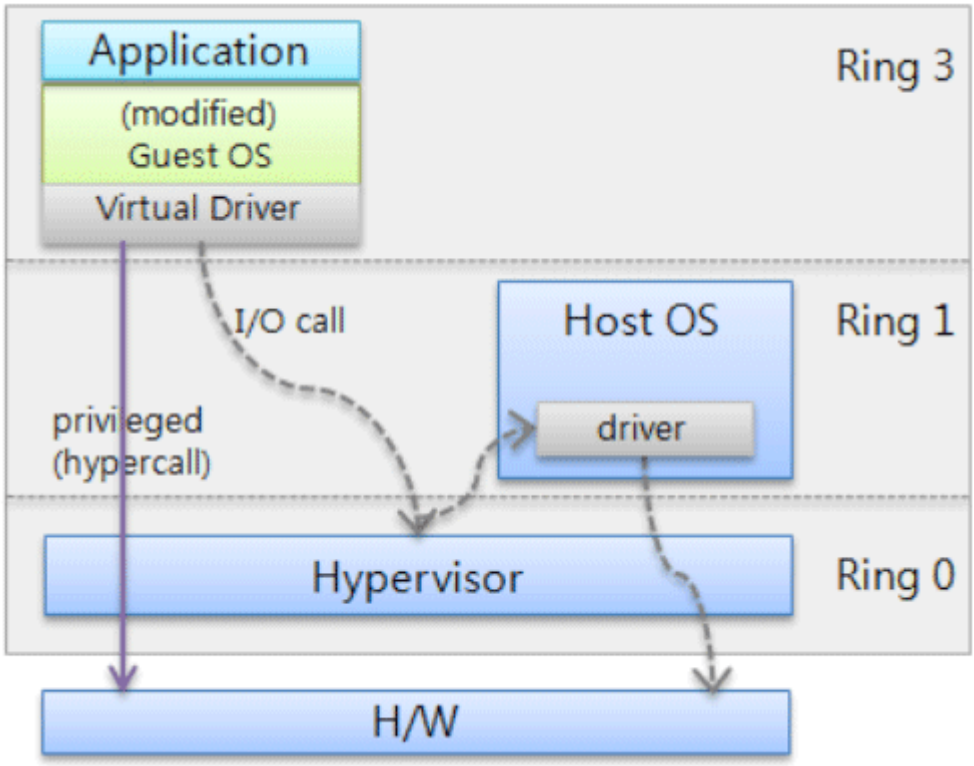

Figure 6 - Ring levels with Paravirtualization

Figure 6 shows the ring level view of a solution with paravirtualization. The main limitation of paravirtualization is the fact that the guest operating system must be tailored specifically to run on top of the virtual machine monitor. However, paravirtualization eliminates the need for the virtual machine to trap privileged instructions.

1.4 Container-based virtualization

Containerization, also known as container-based virtualization, is a virtualization method at the operating system level for deploying and running distributed applications [12]. Containers and virtual machines both allow to abstract the workload from the underlying hardware, but there are important differences in the two approaches that need to be analyzed. Moreover, the principle is the same and with a physical server, but using a single kernel, multiple isolated systems can be run. These are called containers.

Considering that containers share the same OS kernel, they can be more efficient than virtual machines. In fact, containers hold the components necessary to run the desired software, such as files, environment variables, and libraries. On the contrary, a virtual machine includes a complete operating system along with any drivers, binaries or libraries, and then the application. Each virtual machine is executed atop a hypervisor, which itself runs on a host operating system and in turn, operates the physical server hardware.

With containerization, the operating system is responsible for controlling containers access to physical resources, such as CPU and memory. This means that a single container cannot consume all physical resources of a host. A major factor in the interest in containers is they can be created much faster than hypervisor-based instances. This makes for a much agiler environment and facilities new approaches, such as microservices, continuous integration, and delivery.

The concept of containerization basically allows virtual instances to share a single host operating system and relevant binaries, libraries or drivers. This reduces wasted resources because each container only holds the application and related binaries or libraries. Furthermore, the role of a hypervisor is handled by a containerization engine, as it will be discussed further, which is installed on the host operating system and allows to users the possibility to manage containers, as well as a classical hypervisor, does with virtual machines.

However, there are not just advantages. In fact, a potential drawback of containerization is lack of isolation from the operating system running on the host. Therefore, security threats have easier access to the entire system when compared with hypervisor-based virtualization.

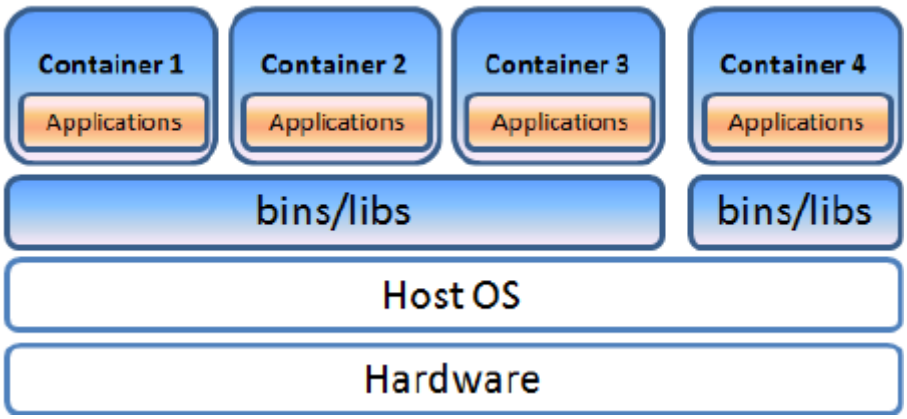

Figure 7 - Container-based virtualization architecture

Figure 7 presents the stack architecture of a container-based virtualization. It is also called as lightweight virtualization layer in which there is no virtualized driver. Therefore, a group of processes is put together in an isolated environment even if the underlying operating system is shared.

Containers are a solution to the problem of how to get the software to run reliably when moved from one computing environment to another. This is obviously quite suitable to be adopted in cloud computing environments because the mobility of resources is one of the most important features that the model provides to users.

The technology was introduced for the first time in 1979 [13]. The first mover was the so-called chroot UNIX, a system call that allows the change of the root directory of running processes. The idea was to provide a sort of isolation between applications. Of course, this was the precursor of Linux container, and later container solutions such as Linux VServer, OpenVZ, and LXC were introduced.

The purpose of this section is to investigate these proposals, how they implement the concept of containerization and particularly what are the key differences between them.

1.4.1 Linux-VServer

Linux-VServer is a virtual private server implementation that was created by adding operating system-level virtualization capabilities to the Linux kernel [14]. It is developed and distributed as open-source software and consists of a jail mechanism in which is possible to securely divide up resources on a computer system (such as the file system, CPU time, network addresses and memory) in such a way that processes cannot mount a denial-of-service attack on anything outside their logical resources. Each partition is called security context, and the

virtualized system within it is the virtual private server, nowadays correspondent to the concept of a container.

The context themselves are robust enough to boot many Linux distributions unmodified. To save space on such installations, each file system can be created as a tree of copy-on-write hard links to a “template” file system. The hard link is marked with a special file system attribute and when modified, is securely and transparently replaced with a real copy of the file.

Virtual servers share the same system call interface and do not have any emulation overhead. Furthermore, they do not have to be backed by opaque disk images but can share a common file system and common sets of files (through copy-on-write hard links).

Processes within the virtual server run as regular processes on the host system. Networking is based on isolation rather than virtualization, so there is no additional overhead for packets.

However, there are also disadvantages: VServer requires that the host kernel must be patched; no clustering or process migration capability is included, and this means that the host kernel and host computer is still a single point of failure for all virtual servers.

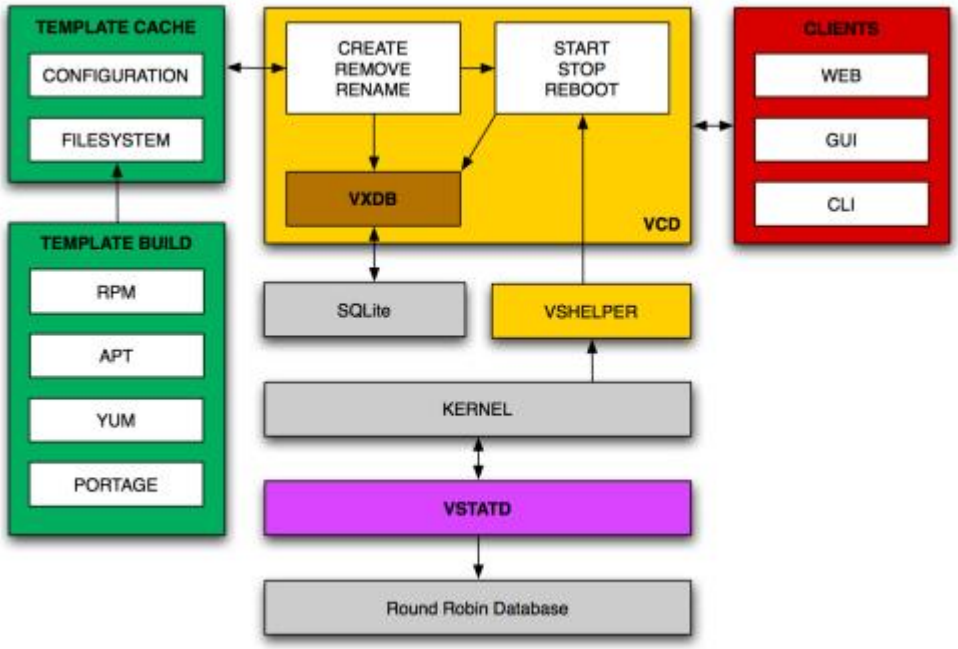

Figure 8 - VServer Control Daemon Architecture

Figure 8 illustrates the architecture of a VServer implementation.

The architecture consists of five major parts: the configuration database (VXDB), the XMLRPC Server, the XMLRPC Clients, the Template Management which acts as a lightweight statistic collector.

The configuration database stores all virtual private server related configuration data like disk limits, CPU scheduler or network addresses. Furthermore, the daemon stores information about its users and their permissions. This database is implemented using SQLite.

The XMLRPC Server is the core of VServer Control Daemon and implements the XMLRPC standard for Remote Procedure Calls (RPC). XMLRPC is a specification and a set of implementations that allow software running on different operating systems.

The XMLRPC protocol describes the serialization format that clients and servers use to pass remote procedure calls to each other. This protocol is characterized by two important features: the details of parsing the XML are hidden from the user, and there is no need to write both client and server in the same language.

The XMLRPC Clients, on the other hand, connect to the XMLRPC Server using the HTTP protocol. They need to pass authentication information and the connection between server and client is not persistent. This means that is required to pass authentication information with every method call.

The Template Management consists of various scripts and XMLRPC methods used to build and create new virtual private servers. The Template Build process assembles a complete root filesystem usable in virtual private servers and stores its content in a single tarball, the Template Cache.

The last component, the Statistic Collector, is a very lightweight daemon used to collect time-series data from running virtual private servers. The collected data is stored in Round Robin Database which is the industry standard data for logging and graphing applications.

1.4.2 OpenVZ

OpenVZ creates multiple secure, isolated containers on a single physical server enabling better utilization and ensuring that applications do not conflict [15]. These are even called virtual environments or virtual private servers, and therefore each container executes exactly like a stand-alone server.

A virtual environment is an isolated program execution environment, which looks and feels like a separate physical server. Multiple virtual environments co-exist within a physical server and, even if they can run different Linux distributions, all these environments operate under the same kernel.

It requires a modified Linux kernel with the augmenting of functionalities such as virtualization and isolation of various subsystems, resource management, checkpointing. While other concepts have already been discussed in complementary solutions, the checkpointing is a key concept of OpenVZ. It is a process of "freezing" a virtual environment, saving its complete state to a disk file, with the ability to "unfreeze" that state later. Checkpointing allows the "live" migration of a virtual environment to another physical server. Therefore, both temporary and persistent states can be transferred to another machine and the virtual environment can be "unfrozen" there.

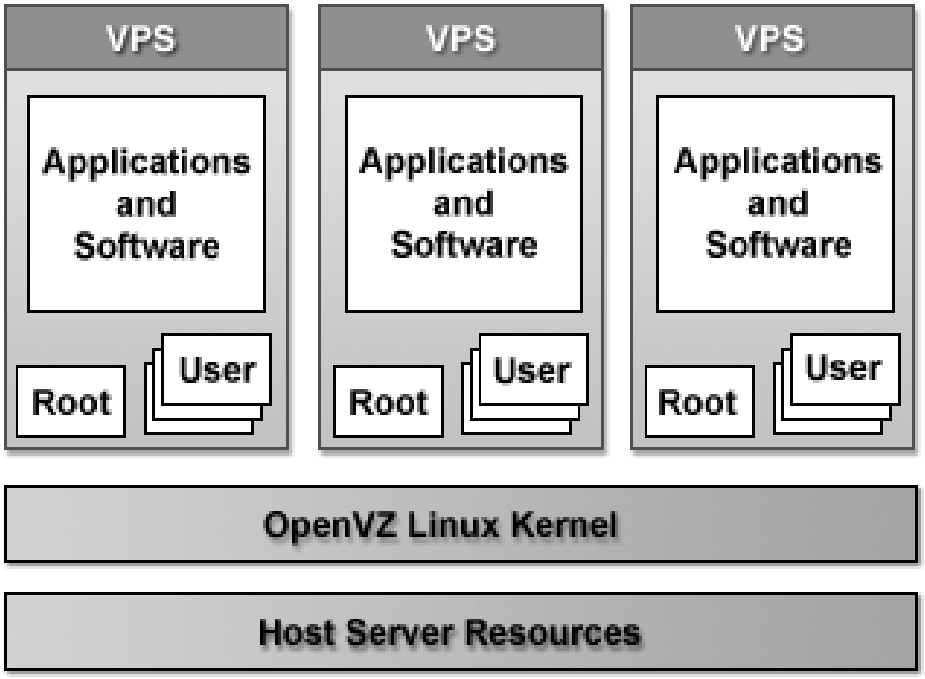

Figure 9 - OpenVZ General Architecture

Figure 9 illustrates the architecture of a system with the implementation of OpenVZ.

OpenVZ is distributed with a client utility, which implements a high-level command line interface to manage Virtual Environments. All the resources can be changed during runtime. This is usually impossible with other virtualization technologies, such as emulation or paravirtualization.

Virtual environments are created from templates, set of packages, and a template cache is an archive (tarball) of the chrooted environment with those packages installed.

Moreover, the concept of containerization is realized with techniques that are also used in complementary solutions such as Linux VServer. Each one provides its own features even considering the time in which they have been designed. Furthermore, the goal is always to abstract the way in which a physical server can be used to run multiple applications that do not need any shared level.

1.4.3 LXC

Linux containers (LXC) provides lightweight operating system virtualization and is relatively new [9]. LXC does not require hardware architecture support, and it is the successor of Linux VServer and OpenVZ.

The Container-paradigm allows for processes and their resources to be isolated without any hardware emulation or hardware requirements. For this reason, they provide a sort of virtualization platform where every container can run their own operating system. This means that each container has its own filesystem, network stack and running its own Linux distribution.

These abstractions make a container behave like a virtual machine with a separate filesystem, networking, and other operating system resources. Isolation is an important aspect of the container, and it is provided through two kernel features of the Linux operating system: groups and namespaces.

Cgroups, abbreviated from control groups, is a Linux feature that limits and isolates the resource usage of a collection of processes, while namespaces are responsible for isolating and virtualizing system resources of a set of processes. Namespaces are used to isolate resources like: filesystem, networking, user management and process ids; cgroups are used for resource allocation and management. In this way, it is possible to limit the number of resources that can be assigned to a specific container.

An important difference in resource allocation between LXC and virtual machine solutions is that CPU resources cannot be allocated on a per core basis, rather one should specify a priority. Originally, LXC containers were not as secure as other OS-level virtualization methods such as OpenVZ. The problem was that a root user of the guest system could run arbitrary code on the host system with root privileges, much like chroot jails.

Starting with the LXC 1.0 release, it is possible to run containers as regular users on the host using “unprivileged containers”. These are more limited in that they cannot access hardware directly. Nevertheless, even privileged containers should provide adequate isolation in the LXC 1.0 security model, if properly configured.

Privileged containers are defined as any container where the container root user identifier is mapped to the same root identifier of underlying host. In such containers, protection of the host and prevention of escape is entirely done through mandatory access control solutions.

They are still valuable in an environment where trusted workloads are running or where no untrusted task is running as root in the container. In contrast to OpenVZ, LXC works in the vanilla Linux kernel requiring no additional patches to be applied to the kernel sources.

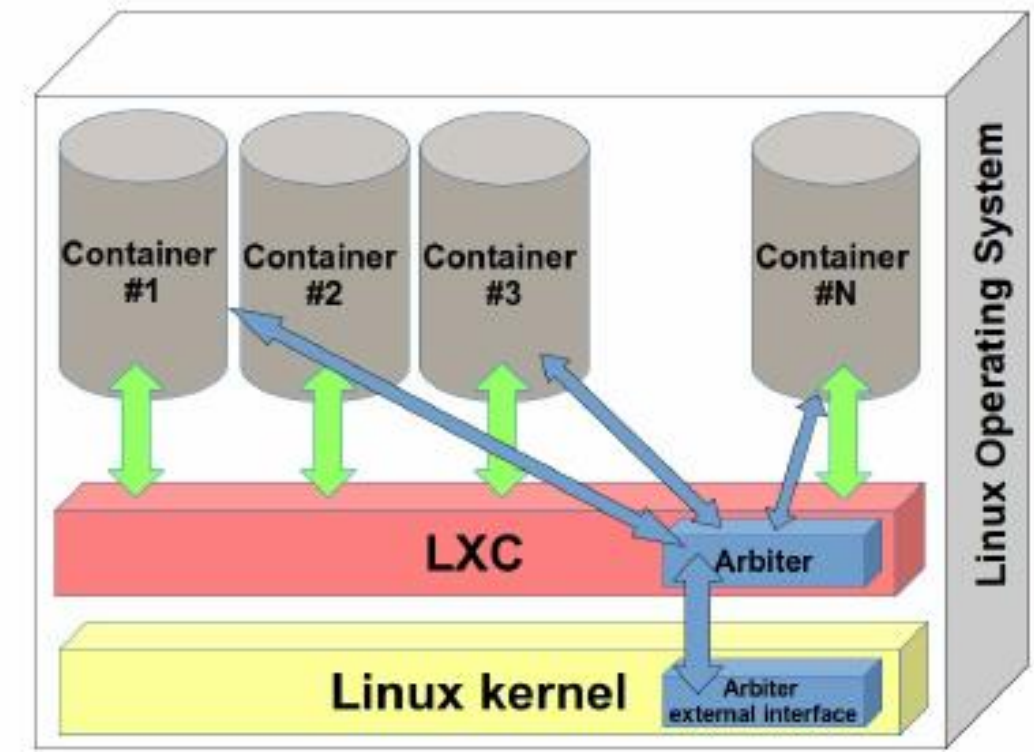

Figure 10 - LXC General Architecture

Figure 10 shows us the general architecture of LXC in which the software layer (LXC) intermediates between kernel features and containers. As explained before, LXC implements container primitives performing a sort of arbiter role, necessary to control and manage the underlying realization.

Unprivileged containers are safe by design. The root container identifier is mapped to an unprivileged user outside of the container and only has extra rights

on resources that it owns itself. With such container, the use of additional security solution is not necessary.

However, LXC will still adopt security solutions which may be handy in the event of a kernel security issue, but they do not enforce the security model.

1.5 Closing remarks

Although virtualization, as a form of technology has existed since the 1960s, only recently has become a staple in the IT industry [16]. Of course, the increasing of popularity is influenced by the introduction of cloud computing. By offloading hardware requirements and utility costs, it can rapidly transform the infrastructure and improve its efficiency by itself. In fact, Cloud computing takes advantages of virtualization allowing to run multiple applications and operating systems on the same server, thereby providing for efficient resource utilization and reducing costs.

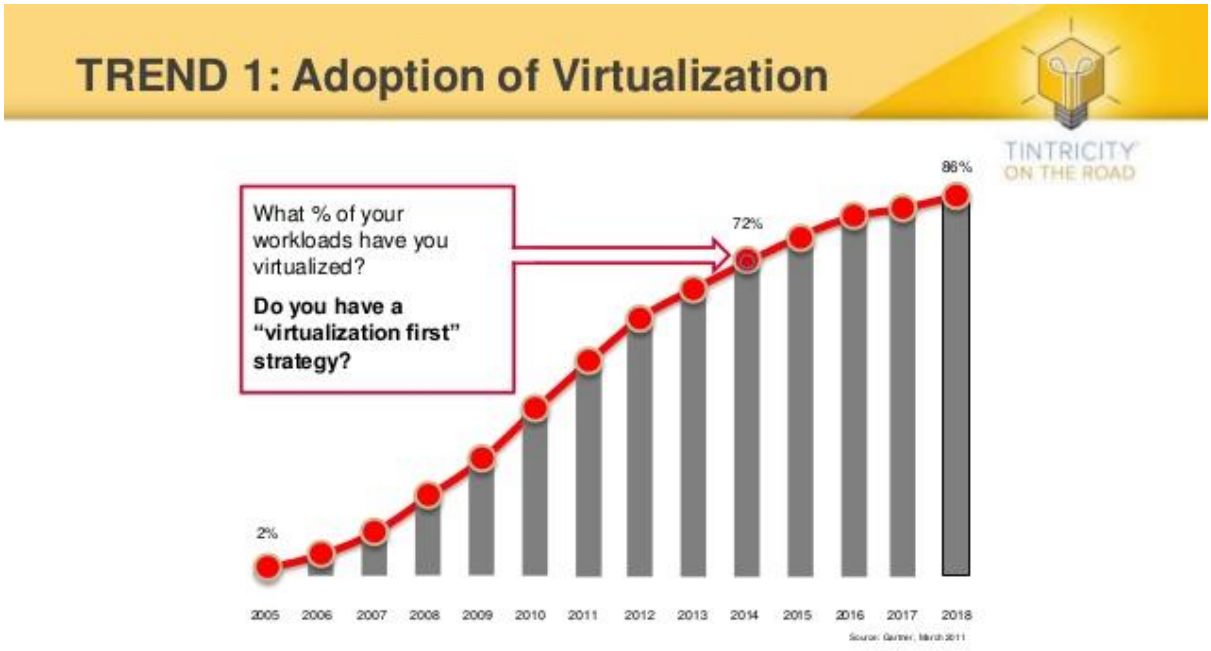

Figure 11 - Rate Adoption of Virtualization

Figure 11 presents the adoption of virtualization in IT industry.

Essentially, virtualization differs from cloud computing because it is a software that manipulates hardware, while cloud computing refers to a service that results from that manipulation. However, virtualization is a foundational element of cloud computing considering the fact that it helps deliver on the value of cloud computing.

As described in this chapter, the first and most known type of virtualization is about running multiple virtual machines on the same physical server. Determining whether or not this type of virtualization is the best solution for a business requires an in-depth analysis of the organization requirements.

Therefore, the purpose of this thesis is to perform an in-depth analysis in order to evaluate the differences between deployments with virtual machines and containers. The virtual machine model includes the introduction of software that exists outside of a guest operating system to intercept the commands sent to the underlying hardware. This is called "hypervisor" and, as mentioned in the correspondent section, could be deployed according to one of these following solutions: full virtualization, hardware-assisted virtualization, paravirtualization.

	Full Virtualization with Binary Translation	**Hardware Assisted Virtualization**	**OS Assisted Virtualization / Paravirtualization**
Technique	Binary Translation and Direct Execution	Exit to Root Mode on Privileged Instructions	Hypercalls
Guest Modification / Compatibility	Unmodified Guest OS Excellent compatibility	Unmodified Guest OS Excellent compatibility	Guest OS codified to issue Hypercalls so it can't run on Native Hardware or other Hypervisors Poor compatibility; Not available on Windows OSes
Performance	Good	Fair Current performance lags Binary Translation virtualization on various workloads but will improve over time	Better in certain cases
Used By	VMware, Microsoft, Parallels	VMware, Microsoft, Parallels, Xen	VMware, Xen
Guest OS Hypervisor Independent?	Yes	Yes	XenLinux runs only on Xen Hypervisor VMI-Linux is Hypervisor agnostic

Figure 12 - Comparison Virtualization Techniques

Figure 12 illustrates a comparison between the techniques involved with the virtual machine model. As it is possible to notice from the table above [17], sometime, performances should be sacrificed in favor of guest compatibility and hardware requirements.

Modern cloud infrastructure uses virtualization to isolate applications, optimize the utilization of hardware resources and provide flexibility. However, at the end of this chapter, we can assert that conventional virtualization comes at the cost of resource overhead.

As the virtual machine model requires the presence of a hypervisor, the containerization includes the introduction a software layer which is called container engine. This chapter has followed the historical path of containerization in which the most spread solutions were: Linux VServer, OpenVZ, and LXC.

	Linux VServer	**OpenVZ**	**LXC**
Works on non-patched kernel	✘	✘	✓
Limit memory usage	✓	✓	✓
Limit kernel memory usage	✓	✓	✘
Limit disk IO	✘	✘	✓
Limit disk usage	✓	✓	Partial
Checkpointing	✘	✓	✓ new
Live migration	✘	✓	✓ new

Table 1 - Comparison between Linux VServer, OpenVZ, and LXC

Table 1 illustrates a comparison between the solution of container engines which have been discussed in this chapter. LXC is newer than others, and several functionalities have been introduced later due to the increasing adoption of container paradigm. Furthermore, container-based virtualization could be an alternative as it potentially reduces overhead and thus improved the utilization of data centers. Both virtualization technologies are used to take advantages even if there is still no winner and both are the right choice for different use cases.

In conclusion, we have seen that containerization is a valid complementary solution that could also be adopted in cloud deployments. However, the container

engine is not enough and, as happened with virtual machines, the solution should include other points of view, such as management and orchestration of these instances.

Therefore, the next chapters will discuss the introduction of higher services which have been introduced in the containerization paradigm. Nevertheless, the question which we want to answer is if the containerization is a suitable alternative to server virtualization in cloud computing and of course what are the key concepts that an organization should consider an alternative to scenarios with server virtualization deployments.

2 Container Management

2.1 Overview

In the previous chapters, we investigated the possibility to create server virtualization using hypervisor-based virtual machines or container engines. Furthermore, containerization can also be used for higher services, according to another paradigm such as microservice [18]. This has led to distinguish between two categories: system and application containers. System containers are meant to be used as completely server virtualization in which the target is to run multiple operating systems on the same physical server concurrently. Application containers are the solution to the microservice paradigm in which consumers need to focus at a higher level.

Operating-system level virtualization is becoming increasingly fundamental for server applications since it provides containers as a foundation of the emerging microservice architecture, which enables agile application development, deployment, and operations. At the moment, cloud containers are a hot topic in the IT world, and the main idea is that containers are designed to virtualize a single application. So far, cloud containers have predominantly been the domain of Linux-based servers but nowadays even Microsoft, with Hyper-V container, will introduce the support to this new paradigm. However, there are still questions that need answers, such as how exactly are containers different to traditional hypervisor-based virtual machines. Therefore, the goal of this work is to investigate this technology comparing its own benefits with other well-adopted deployments, such as hypervisor-based virtual machines.

Today, many organizations strive to cope with rapid market changes, such as evolving customer requirements and new business processes. Agility in the microservice architecture depends on fast management operations for the container, such as create, start, and stop. A common design practice is to implement a service as a set of microservices, and the goal is to take advantages of existing solutions in order to get portability, isolation, and robustness. While system containers are designed to run multiple processes and services, application containers are meant to package and run a single service.

The most common application containerization technology is Docker, based on universal runtime runC, while the main competitive offering is CoreOS rkt container engine, which relies on the App Container specification. These solutions

are the most spread in the area of application containers. Furthermore, between application and system containers, a recently new solution, LXD, has been introduced in order to provide features that Docker or rkt handles less elegantly external to the container hosting. The primary value of LXD is simplicity. It is a container hypervisor that does not include the application delivery framework as included in both Docker and rkt. However, LXD is easier to integrate with virtualization frameworks, such as OpenStack, or with general DevOps tools, such as Chef and Ansible. Therefore, this chapter will investigate these solutions in order to evaluate a higher point of view of the containerization paradigm.

2.2 Docker

Docker comprises a command-line program, a background daemon, and a set of remote services that take a logistical approach to solve common software problems and simplify the experience of installing, running, publishing and removing software [19]. It accomplishes this through the containerization technology. Using containers has been a best practice for a long time, but manually building containers can be challenging and easy to do incorrectly.

This challenge has put them out of reach for some, and misconfigured containers have lulled others into a false sense of security. Docker is a helper in this scenario and any software, which is running with Docker, runs inside a container.

Docker uses Linux namespaces and cgroups, which have been part of Linux since 2007. Furthermore, it does not provide the container technology, but it specifically makes it simpler to use.

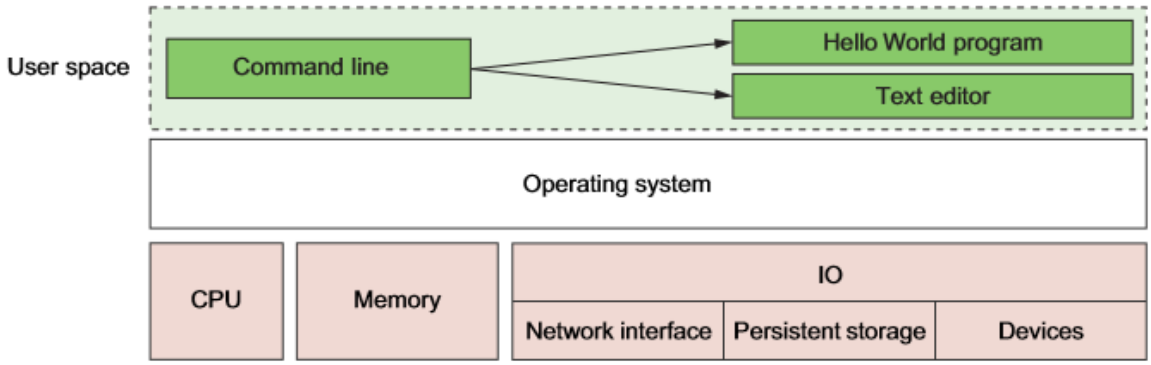

Figure 13 - A basic computer stack running two programs

Figure 13 represents an architecture stack with two running programs started from the command line. This case involves a command line interface which runs in the so-called user space, just like other programs that run on top of the operating system. Ideally, programs running in user space cannot modify kernel space

memory. In fact, the operating system is the interface between all user programs and the hardware that the applications are running on.

Until some time ago, Docker was built on top of LXC in order to create namespaces and all the components that go into building a container. As Docker matured and portability became a concern, a new container runtime called "libcontainer" [19] was built, replacing LXC. However, the ecosystem uses an interface layer so that users can change the container execution provider.

The idea behind application containers is that containers should be created for each component of the application. This is fundamental for a multi-component system using the microservice architecture.

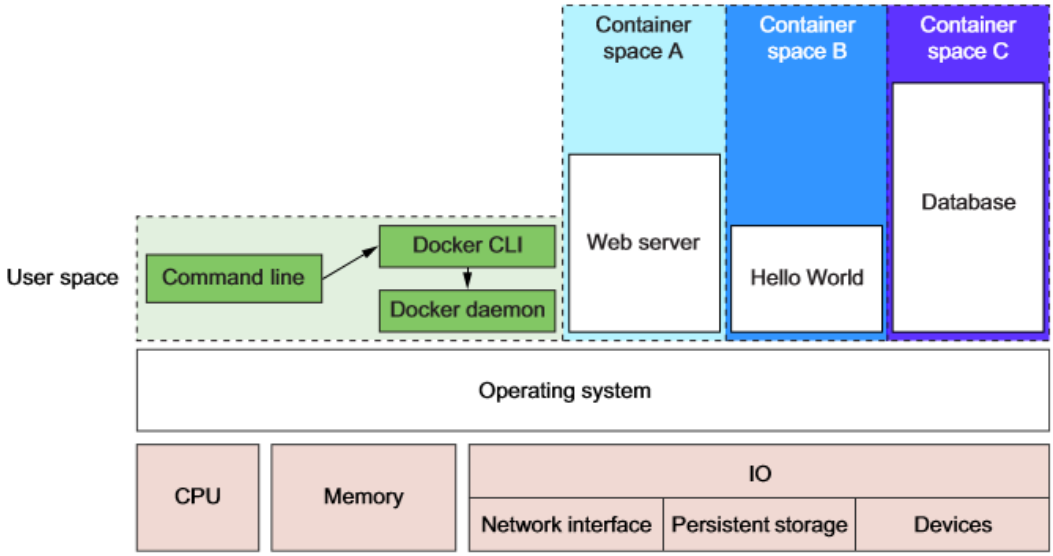

Figure 14 - Docker running three containers

Figure 14 illustrates the case in which three containers are deployed on a basic Linux operating system. The user space includes the command line interface in which the Docker client interacts with the Docker daemon. The architecture of the ecosystem will be detailed in the correspondent section, but the purpose of the picture is to demonstrate how the technology makes use of kernel features such as namespaces. Therefore, each container is completely isolated from other processes, such as the agent that created it.

Each container is running as a child process of the Docker daemon, wrapped with a container, and the delegate process is running in its own memory subspace of the user space. Programs running inside a container can access only their own memory and resources as scoped by the container. Therefore, the containers that Docker builds are isolated concerning some aspects.

Isolation aspect	Description
PID namespace	Process identifiers and capabilities
UTS namespace	Host and domain name
MNT namespace	File system access and structure
IPC namespace	Process communication over shared memory
NET namespace	Network access and structure
USR namespace	User names and identifiers
Chroot	Controls the location of the root file system
Cgroups	Resource protection

Table 2 - Isolation aspects of Docker containers

Table 2 shows the isolation aspects of Docker containers which are set by default to each Docker container. However, there is the possibility to extend the level of security enabling appropriate security solutions.

However, these are not the unique security measures adopted by Docker. Therefore, the ecosystem takes even care about operating system features access through involving capabilities [19], which are features of the Linux operating system. Whenever a process attempts to make a gated system call, the capabilities of that process are checked for the required capability. The call will succeed if the process has the required capability and fail otherwise.

At the creation of a container, by default, Docker drops a specific set of capabilities in order to further isolate the running process from the administrative functions of the operating system.

Sometimes there are cases in which we need to run a system administration task inside a container. Therefore, that container should be granted with privileged access to the underlying host operating system. As shown in LXC, these are called privileged containers. They maintain their file system and network isolation but have full access to shared memory and devices in addition to possess full system capabilities.

2.2.1 Architecture

The architecture is based on the client-server model. The Docker client talks to the Docker daemon, which is responsible for building, run and distribute Docker containers [19]. The ecosystem automates the repetitive tasks of setting up and configuring development environments so that developers can focus on just building software.

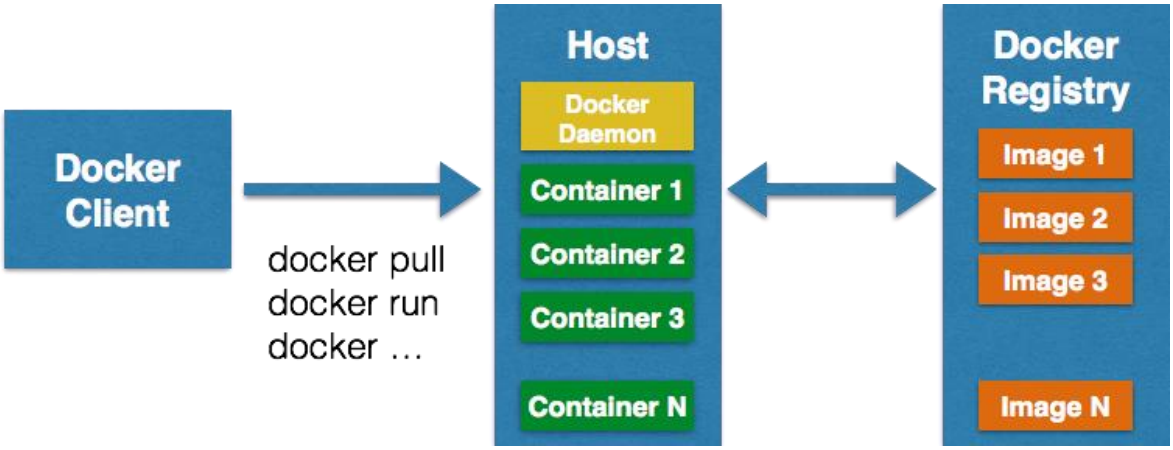

Figure 15 - Docker Architecture

Figure 15 shows us the components of the Docker ecosystem Architecture, which are structured following the client/server paradigm.

There is no constraint to have client and daemon running on the same host. They are designed to work without no difference if they were local or remote. They communicate using a REST API, over UNIX sockets or a network interface. When the processes are running on the same host, the implementation provides optimization in order to take advantages of inter-process communication.

Due to this level, the complexity is pushed into containers that are easily built, shared and run. The clear advantage of this architecture is that there is no need to have extra hardware for guest operating systems because everything works as Docker containers.

As shown in Figure 15 the architecture consists of the following main components:

- **Docker daemon** (dockerd) – it is the server which listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. It can also communicate with other daemons to manage Docker services.
- **Docker client** (docker) – it is the primary way that many Docker users interact with Docker. This sends HTTP REST requests to dockerd, which carries them out.
- **Docker registry** – it stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use.

By default, Docker is configured to look for images on Docker Hub. Each pull request is performed to a configured registry allowing anyone to buy and sell

Docker images or distribute them for free. Furthermore, this does not require any customization between different staging environments. In fact, it is always possible to upgrade the application by pulling the new version of the image and redeploying the containers.

2.2.2 Docker Images

The metaphor associated with Docker containers is that they are considered as physical shipping container [19]. It is a box where is possible to store and run the application and all of its dependencies. Moreover, to easily work with containers, it is necessary to provide a mechanism to distribute containers with ease. Docker completes the traditional container metaphor by including a way to package and distribute software. The component that fills the shipping container role is called an image.

A Docker image is a bundled snapshot of all the files that should be available to a program running inside a container. Many containers can be created from an image but, even if containers started from the same image, they do not share changes to their file system. Software with Docker is distributed through images, and this allows the receiving computers to create containers from them.

Images are the shippable units in the Docker ecosystem, and therefore a set of infrastructure components are coordinated to simply distributing Docker images. These components are registries and indexes, and it is even possible to use the publicly available infrastructure provided by Docker Inc., other hosting companies, or the own local registries and indexes.

A Docker image is a collection of layers. Each of which is an image that is related to at least one other image; images define parent/child relationships built from their parents to form layers. Therefore, each file available to a container is obtained by the union of all the layers in the lineage of the image that the container was created from. This aspect is quite different from the implementation in operating system containers because by default they do not include any layered file system.

Programs running inside containers know nothing about image layers. Furthermore, the file system operates as though it does not run in a container or operate on an image. From the perspective of the container, it has exclusive copies of the file provided by the image. This is made possible with something called a union file system.

Docker uses a variety of union file systems, which is part of a critical set of tools that combine to create the effective file system isolation. The other tools are MNT namespaces, and the chroot system call. The file system is used to create mount points on the host file system, which is responsible to abstract the use of layers. Therefore, these layers are bundled into a Docker image.

When a Docker image is installed, its layers are unpacked and appropriately configured for use by the specific file system provider chosen for the underlying host. The Linux kernel provides a namespace for the MNT system. When Docker creates a container, that new container will have its own MNT namespace, and a new mount point will be created for the container to the image. Lastly, chroot is used to make the root of the image file system the root in the container context. This prevents anything running inside the container from referencing another part of the host file system.

The most important benefit of this approach is that common layers need to be installed only once. This means that several specializations of a program are installed without storing redundant file or downloading redundant layers. By contrast, most virtual machine technologies will store the same files as many times as redundant virtual machines are installed on a computer.

2.2.3 Docker containers

A container is a runnable instance of an image. Using Docker API or CLI, it is possible to perform actions on Docker containers [19]. By default, a container is relatively well-isolated from other containers and its host machine. It is defined by an image as well as any configuration options provided when created or run it. When a container is stopped, any changes to its own state, which are not stored in persistent, will be disappeared.

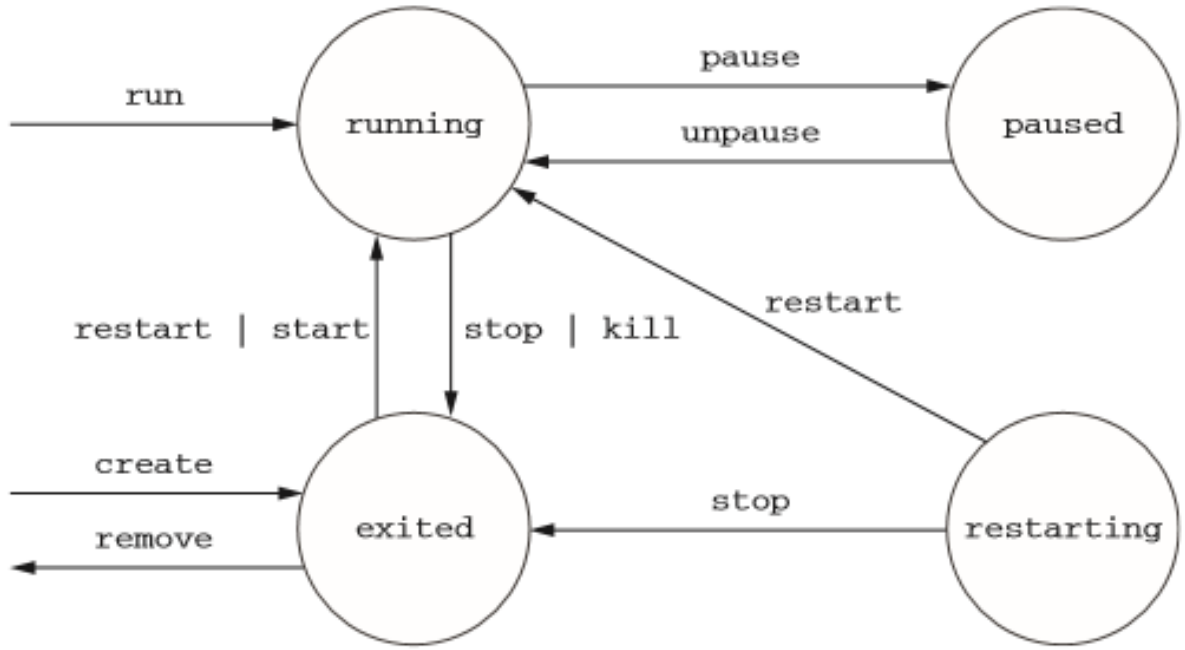

Figure 16 - The state transition diagram for Docker containers

Figure 16 represents the diagram state of a Docker container.

It consists of four states, and the execution phase is placed in the running state. When an image is built, the docker daemon prepares the content to launch a new instance of that image. This is accomplished by executing the run command of the Docker API. At this point, the container becomes a running process, and the execution is delegated to its main command. Therefore, Docker is considered an application container because the purpose is to execute a single command within the execution environment.

In addition to isolation features of the Linux kernel, such as cgroups and kernel namespaces, Docker makes use of a union-capable file system. This is a file system service for Linux which implements a union mount for other file systems. Furthermore, this allows files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. This allows independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines.

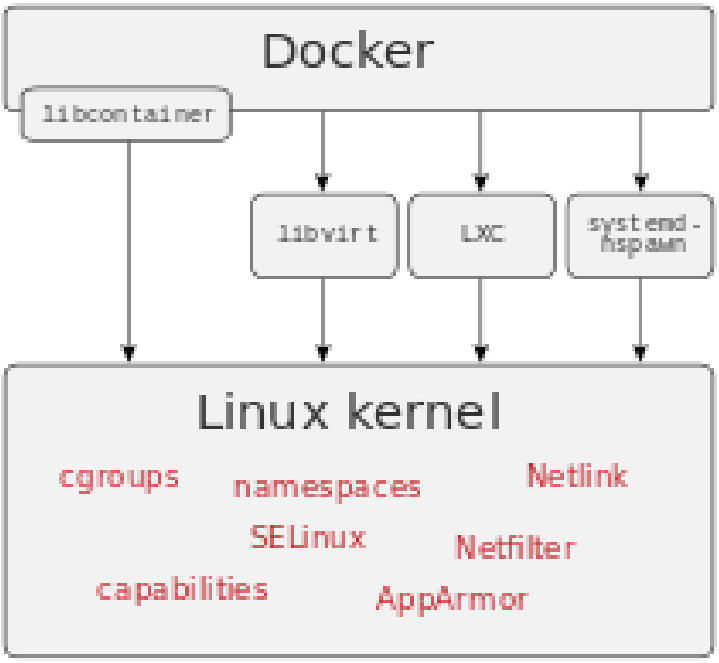

Figure 17 - Docker Linux Interfaces

Figure 17 illustrates the interaction model between Docker engine and the underlying Linux kernel. Since version 0.9, Docker includes the libcontainer library as its own way to directly use virtualization facilities provided by the Linux kernel, in addition to using abstracted virtualization interfaces via libvirt, LXC and system-nspawn.

Building on top of facilities provided by the Linux kernel, a Docker container, unlike a virtual machine, does not require or include a separate operating system. Instead, it relies on the functionalities of the kernel and uses resource isolation and separate namespaces to isolate the application point of view from the underlying operating system.

Docker containers running on a single machine share the kernel operating system; they start instantly and use less compute and RAM than virtual machines. Furthermore, they are based on open standards and run on all major Linux distributions, Microsoft Windows, and on any infrastructure including virtual machines, bare-metal and in the cloud. However, Docker is not a system container and therefore also the integration with cloud operating platforms such as OpenStack is no longer maintained. As it will be seen later, complementary solutions such as LXD or Zun, are more suitable to be integrated.

2.2.4 Network

Docker is concerned with two types of networking: single-host virtual networks and multi-host networks [19]. Local virtual networks are used to provide container isolation while multi-host virtual networks provide an overlay where any container on a participating host can have its own routable IP address from any container in the network. This section covers single-host virtual networks. This is crucial for the security-minded, and multi-host networking requires a broader understanding of other ecosystem tools in addition to understanding the material covering single-host networking.

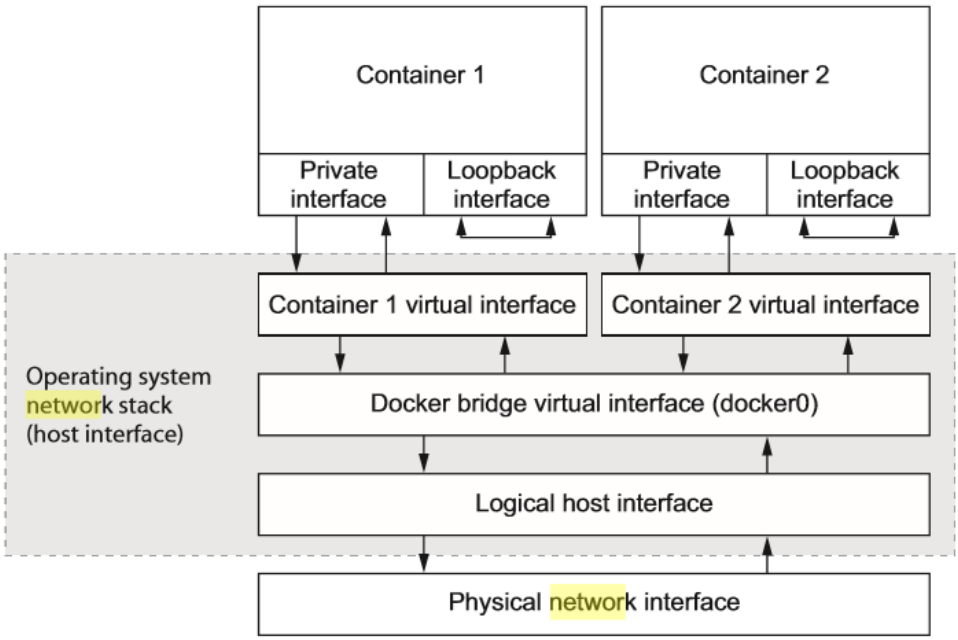

Figure 18 - The default local Docker network topology

Figure 18 illustrates the default network topology with two docker containers. As it is possible to see, those containers have their own private loopback interface and a separate Ethernet interface linked to another virtual interface in the namespace of the underlying kernel. These two linked interfaces make a link between the host network stack and the stack created for each container. Docker uses features of the underlying operating system to build a specific and customizable virtual network topology. The virtual network is local to the machine where Docker is installed and is made up of routes between participating containers and the wider network where the host is attached.

Each container gets a unique private IP address that is not directly reachable from the external network. Connections are routed through the Docker bridge interface called docker0 and the virtual interfaces created for containers are linked to that

bridge. Together they form a network, and this bridge interface is attached to the network where the host is connected to.

The connections between interfaces describe how exposed or isolated any specific network container is from the rest of the network. Docker uses kernel namespaces to create those private virtual interfaces, but the namespace itself does not provide the network isolation. Network exposure or isolation is provided by the firewall rules of the underlying host.

In order to define how a container interacts with the other network components, a so-called archetype is adopted. All Docker containers follow one of the following archetypes:

- Closed containers – with no communication possibilities.
- Bridged containers – communication with local containers.
- Joined containers – sharing network interfaces with another container.
- Open containers – with full access to the host network.

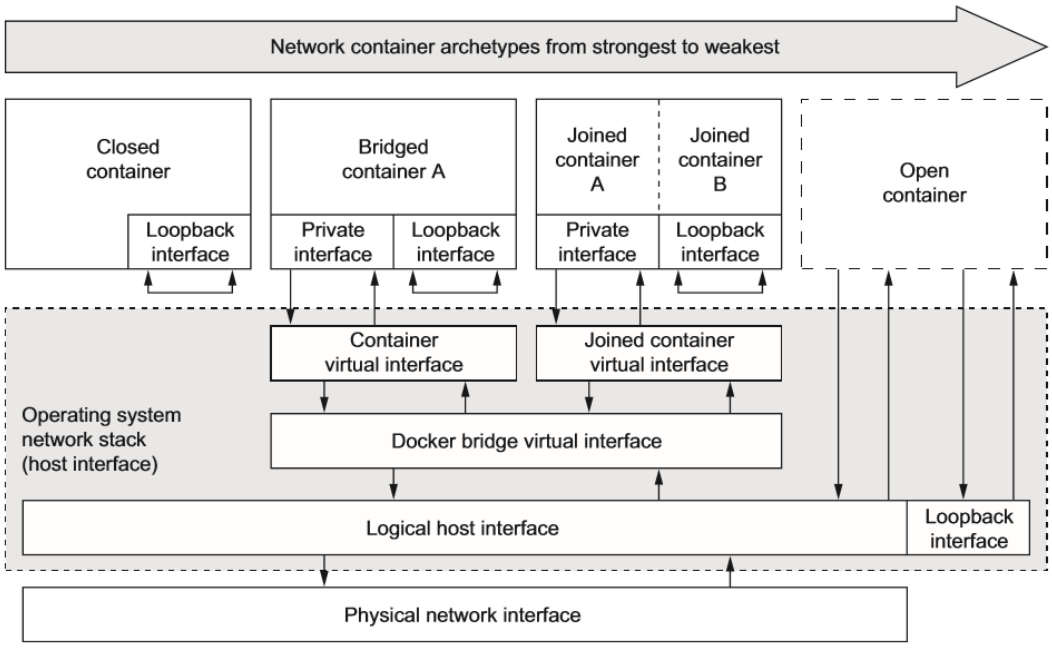

Figure 19 - Network archetypes and their interaction with the Docker network topology

Figure 19 illustrates the four archetypes of the Docker network subsystem. This is meant to the level of isolation that a user wants to reserve to its own containers.

Closed containers are the strongest type of network container. There is no network traffic allowed for this archetype. Processes running in such a container will have

access only to a loopback interface. If they need to communicate only with themselves or each other, this will be suitable whereas if the software needs to download updates, it will not be able to, considering the fact that it cannot use the network. Docker builds this type of container by simply skipping the step where an externally accessible network interface is created. In fact, there is no connection to the Docker bridge interface. Programs in these containers can talk only to themselves. Therefore, closed containers should be used when the need for network isolation is the highest or whenever a program does not require network access. In fact, this is not the default archetype for Docker containers.

By default, Docker creates bridged containers that relax network isolation in doing. They have a private loopback interface and another private interface that is connected to the rest of the host through a network bridge. All interfaces connected to the docker bridge are part of the same virtual subnet. This means that they can talk to each other and communicate with the larger network through the docker0 interface.

Bridged containers are not accessible from the host network by default. In fact, containers are protected by the firewall system of the underlying host. By default, there is no route from the external interface of the host to a container interface. Usually, containers would not be very useful if there were no way to get to them through the network. Moreover, the docker run command provides a flag (-p or –publish) that can be used to create a mapping between a port on the host network stack and the new container interface.

The next less isolated network container archetype is called joined container. These containers share a common network stack, and by this way, there is no isolation between joined containers. Docker builds this type of container by providing access to the interfaces created for a specific container to another new container. Therefore, network interfaces are shared but containers, joined in this way, will maintain other forms of isolation such as file system, memory, and more.

The last archetype of network container is open containers. They are not isolated because they have full access to the host network stack. This includes access to critical host services. Open containers provide absolutely no isolation and should be considered only in cases when no other option is suitable.

2.2.5 Storage

Docker provides the storage subsystem with the concept of volume. This is a mount point on the container directory tree where a portion of the host directory tree has been mounted. Without volumes, container users are limited to work with the union file system that provides image mounts, not providing the durability of data which should be held even after the execution of the container.

A volume is a tool for segmenting and sharing data that has a scope or life cycle that is independent of a single container. That makes volumes an important part of any containerized system design that shares or writes files.

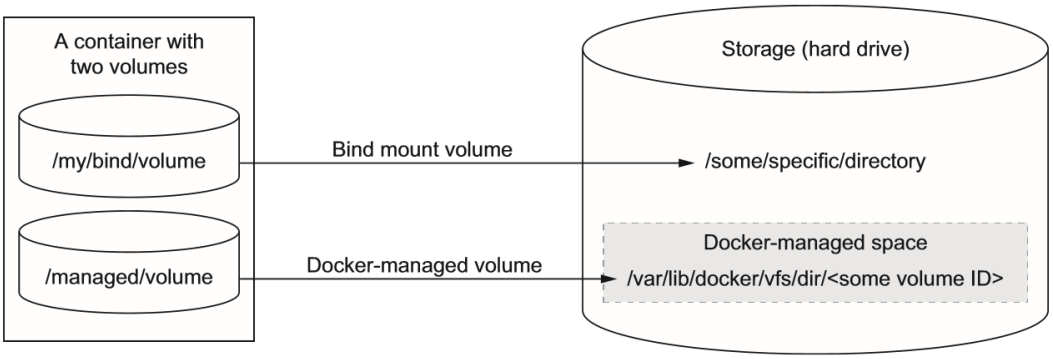

Figure 20 – Docker volume types

Figure 20 illustrates the two-volume types of Docker ecosystem. Every volume is a mount point on the container directory tree to a location on the host directory tree, but the types differ in where that location is on the host. The first type of volume is a bind mount whereas the second one is a managed volume.

Managed volumes use locations that are created by the Docker daemon in space controlled by itself, which is called Docker managed space. A “bind mount volume” is a volume that points to a user-specified location on the host file system. This is useful when the host provides some files or directories that need to be mounted into the container directory tree at a specific point. Furthermore, this is also fundamental when other processes, running outside a container, want to share data such as components of the host system itself. Moreover, it is possible to mount the volume as read-only, guaranteeing that any process inside the container cannot modify the content of the volume.

Bind mount volumes are not limited to directories, even if that is how they are frequently used. Therefore, it is possible to use a bind mount volumes to mount individual files. This provides the flexibility to create or link resources at a level that avoids conflict with other resources. The important thing to note in this case

is that the file must exist on the host before creating the container. Otherwise, Docker will assume that is needed to use a directory and so it will create it on the host, and it will be mounted at the desired location (even if a file occupies the location).

The first problem with bind mount volumes is that they tie portable container description to the filesystem of a specific host. Furthermore, this can be difficult to manage considering the fact that they create an opportunity for conflict with other containers. In fact, it would be a bad idea to start multiple instances of an image that all containers use the same host location as a volume. In that case, each of the instances would compete for the same set of files and therefore, without other tools such as file locks, that would likely result in corruption of the content.

Managed volumes are different from bind mount volumes because the Docker daemon creates managed volumes in a portion of the host file system that is owned by Docker. Using managed volumes is a method of decoupling volumes from specialized locations on the filesystem. This is useful when it is just needed to have a place where to put some data that can be thrown away after finished to work with them. Therefore, Docker can confidently remove any directories or files that are no longer being used by a container.

2.2.6 Docker Compose

Docker compose is a tool for defining, launching and managing services, where a service is defined as one or more replicas of a Docker container [19]. A simple client command-line program, docker-compose, is used to manage services and systems of services. These are defined in YAML (Yet Another Markup Language) files. It accomplishes the task of building Docker images, launching containerized applications as services, launching full systems of services, managing the state of individual services in a system and scale services up or down. This allows us to stop focusing on individual containers and instead of pointing out full environments and service component interactions. A compose file might describe four or five unique services that are interrelated but should maintain isolation and independent scaling. One of the most impressive and useful features of Compose is the ability to scale a service up and down. When performed, compose creates more replicas of the containers providing the service. These are automatically cleaned up when they are scaled down. This tool is useful for managing environments and iterating on projects are similar to docker command-line commands. In fact, all the operations that are available with a single docker container have the equivalent counterpart with Docker compose.

2.3 RKT

Rkt is a container engine designed for modern cloud-based environments [20]. It includes aspects not present in other solutions of container engine such as native integration of pod. It defines an execution environment strongly modular and an interface layer which simply allows the integration with other components. In Rkt the atomic unit of execution is the pod, the same concept introduced in the Kubernetes Orchestration System. Furthermore, it allows the possibility to specify low-level configurations in order to define a fine-grained behavior of each application.

The architecture is not so far different from other solutions such as Docker. Moreover, there is no central daemon, but it is self-contained, guaranteeing isolation by executing each pod directly within classical Unix processes. Rkt implements an open and standard format of container runtime, which is called App Container Specific, but it is also able to build Docker images.

It is the first challenger to Docker in the application container space. The key concept that differentiates it from Docker it is the security point of view, designed to alleviate many of the flaws inherent in the Docker, container model. Nevertheless, this justifies the fact that Docker has recently remediated some of its more critical aspects such as eliminating the need of running containers as root, addressing a longstanding security gripe among its adopters.

Furthermore, Rkt includes the support to check of container images cryptographically. RunC is the container runtime, an implementation of Open Container Initiative Specification but the disadvantage is that users need to know low-level features of the operating system and the overhead is increased with the responsibility to check security properties.

2.3.1 Architecture

The primary interface of the ecosystem is a command-line tool, rkt, which does not require a long-running daemon. This architecture [20] allows the project to be updated in-place without affecting application containers which are currently running. Furthermore, this allows separating the levels of privilege between different operations.

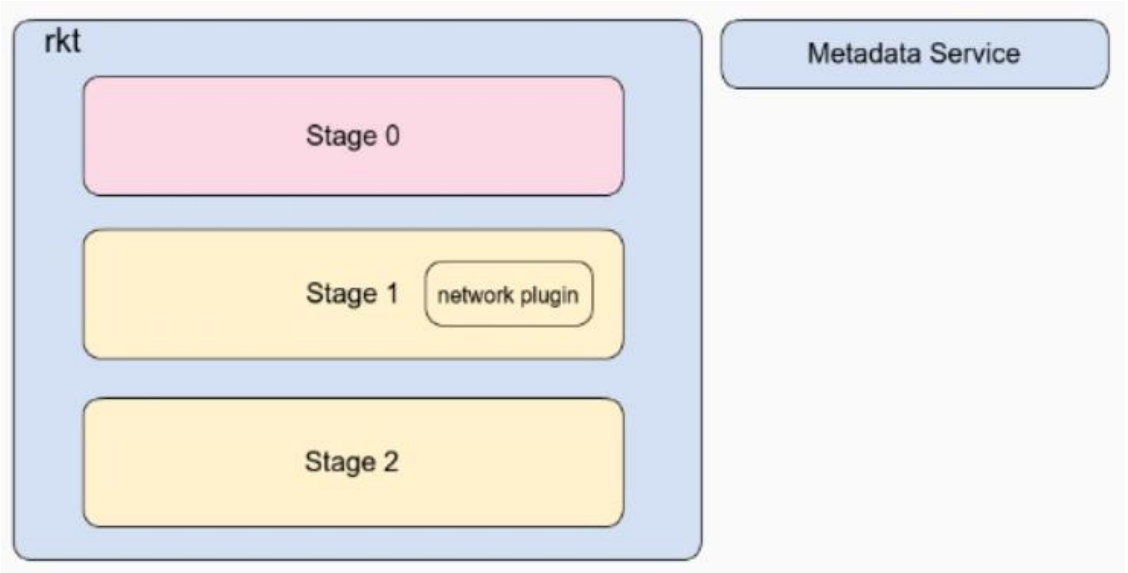

Figure 21 - RKT Architecture

Figure 21 describes the architecture of RKT ecosystem. The whole state of rkt is hand over through the file system, and the concurrent execution of the command uses kernel features such as file-locking in order to avoid competition access problems.

Execution with rkt is divided into several distinct stages, and the execution chain follows the numbering of stages. The first invokes involves the execution of the rkt command-line, which belongs to the so-called “stage0”. This is the state in which a process is invoked, and this operation is accomplished by a shell or a supervisor process. This state consists of a typical fork and exec, generating a child process of the process which has invoked it.

After completing the execution of the first stage, there is the transition to the new state in which the process is substituted with an exec to the entry point. Furthermore, this state is the intermediate between the ecosystem environment and the application which is wanted to run.

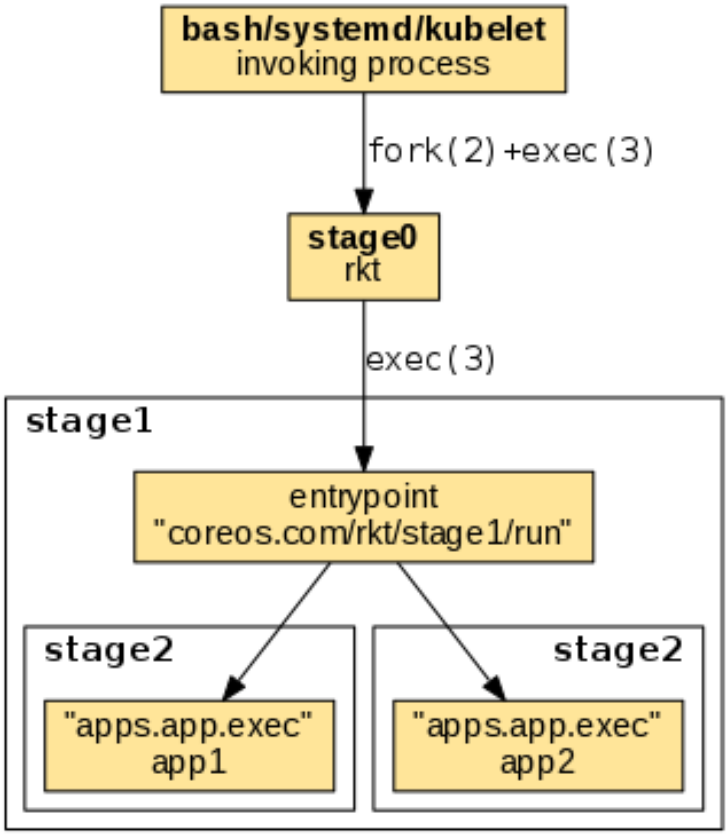

Figure 22 - RKT Execution Stages

Figure 22 illustrates the execution chain of the Rkt ecosystem. Stage1 has the duty to take the file system, created in the stage0 for the pod, define the network isolation and perform mounting to execute the pod. This consists of loading image and manifest of pod because inside the manifest there is specified the exec of each application. The isolation process requires the use of flavor.

The isolation process is accomplished through the use of flavors. Actually, there are three implemented flavors: fly, system-nspawn, and KVM. The last stage concerns about the environment in which is executed the application.

2.3.2 Process Model

Initially, Docker adopted a tightly coupled process model in which the docker daemon was responsible for acting as completely centralized process and therefore running with super-user privileges. Of course, this is a solution with some advantages from the point of view of managing containers. However, there are also other aspects which cannot be integrated such as the automation of some workloads. Therefore, from version 1.11, the docker engine does not manage containers delegating this responsibility to another process, which is called containerd. All of that is completely transparent to users who continue to use the

traditional Docker APIs. Furthermore, the docker daemon is just responsible for preparing the image as a bundle of the Open Container Image(OCI). After that, the containerd is invoked in order to start the OCI bundle. This produces the binary useful to the container runtime (runC) to create and launch container instances.

Rkt does not have a central process, but containers are directly launched within the client command. Moreover, it includes the same functionalities but does not expose the user to low-level details. The execution model of rkt is not so tightly coupled, and therefore other formats are supported such as docker images. This is possible due to the App Container Images standard that both Docker and rkt are based on.

With Rkt, container images are created with a proper build tool in order to define the manifest and the filesystem of a container. Subsequently, the container is ready to be distributed over HTTPS, without the need of specialized registries.

2.3.3 Network

Rkt is designed in the same period of the standardization of Container Network Interface (CNI) [21], a standard to configure network interfaces of Linux containers. Due to this standard, it is possible to define several network configurations that can be useful to different use cases. This implies the creation of a named network using a specific networking mode.

The network model of rkt ecosystem includes three different types of implementation: none, host and default. “None” does not include any network and by this way, a pod is completely isolated from the network. This means that the pod is created with just the loopback interface. On the contrary, host-network includes full access to the host network, sharing the network stack with the underlying host. However, there is no isolation guaranteed using this networking mode, and therefore this is suitable if pod processes belong to the same network namespace of the host.

If no mode is specified with the network option, rkt uses the default configuration type. In fact, in the “stage1” some plugins are configured in order to implement the CNI standard. By default, rkt uses “ptp” with host-local, even if there are also other integrated implementations. Nevertheless, this information will be sent to the execution phase which is the crucial state of the creating container.

2.3.4 Storage

By definition, a container file system is integrated with the image and the modifications applied during the execution are lost at the termination of the container. As in Docker, it is even possible to do that a volume can be mounted in a location of the container file system, necessary to allow persistent data storage.

There is no difference if the storage is provided with physical disks, cloud, and more. The most important aspect is to define which partitions have to be mounted. This responsibility is left to the systemd process which has the duty to make it sure of that.

The specification defines two types of volumes: host and empty. The first consists of exposing a host directory or a host file to the pod whereas the second one involves an initialized empty volume with a life-cycle linked to the pod, and so it will be deleted from a garbage collector.

2.4 LXD

As anticipated in the overview of this chapter, ecosystems based on the application container paradigm focus on application delivery from development to production. We have learned that these solutions offer a great way to deliver applications but the applicative point of view is not the unique layer of containerization. In fact, a current debate, which this work wants to investigate, is if containers could be an alternative to virtual machines. This is the case in which operating system containers are more suitable because users want to use them as well as hypervisor-based virtual machines.

Ubuntu has been one of the most supporters of this new paradigm, and since 2012 it gives us tools for container management and a wide choice of container operating system templates. The concept of container operating system is not new, and different solutions, such as LXC, OpenVZ e Linux VServer have been proposed. However, notwithstanding the fact that these are quite similar to virtual machines, a lot of users found difficulties due to the lower-level profile of implementations. Therefore, the company wanted to realize the potential of its own LXC project and promoted an enhancement which is called LXD.

LXD [22] builds on LXC capabilities to deliver multi-host container management with advanced features like live migration and online snapshotting including running state. It is written in the go language, providing a system daemon which is available to applications using a UNIX socket, or over the network via HTTPS.

Several advantages justify the introduction of LXD to support the LXC technology. With pure-play LXC, it is necessary to separate processes for each container in order to run many LXC containers using only a single system daemon. With LXD this is guaranteed, using a single system daemon, in order to make simpler the management and reduce the overhead.

Furthermore, on plain LXC, container security is more difficult. LXD uses unprivileged containers by default, and it provides more isolation and security than normal LXC containers. In fact, one of the purposes was to face multi-tenant workloads and other use cases that require more locked down environments.

Containers are just one element in an application delivery strategy, and the right type container choice depends on the business strategy, including deployment, security, and governance, but also application performance. LXD users can use the ecosystem as native or wrap the tool in their own higher application delivery framework. This simplicity has been characterizing the solution, and so recently it is available as hypervisor driver in the OpenStack cloud operating system.

2.4.1 Architecture

LXD contains a system daemon which provides an interface layer to drive LXC containers. Its main purpose is to provide user experiences as similar as to that of virtual machines but using Linux containers rather than hardware virtualization [23]. Figure 23 exposes the architecture of LXD ecosystem.

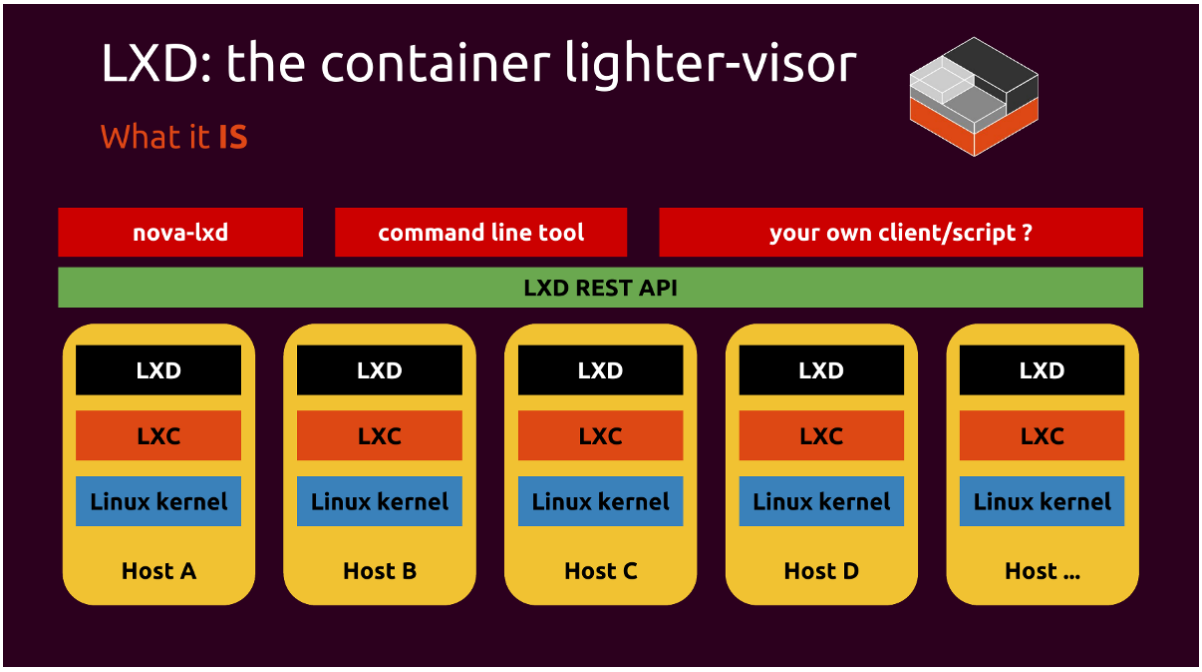

Figure 23 - LXD Architecture

The architecture purpose of the project is to take advantages of existing operating systems solutions, such as LXC, in order to provide the same ease of use already spread in the containerization market. Furthermore, keeping backward compatibility with older containers and deployment methods have also prevented LXC from using security features by default, leading to the more manual configuration for users. Therefore, LXD addressed this topic and provided a new solution which is quite accepted in the so-called container lighter-visor.

LXD focuses on system containers, which are long-running and based on a clean distribution image. In addition, traditional configuration management and deployment tools can be used as well as with virtual machines, cloud instances or bare-metal servers. Therefore, as shown in the picture below, a client can be a normal command line program but also a specialized component such as Nova of OpenStack.

There are some main components that make LXD, and those are typically visible in the LXD directory structure, in its command line client and in the API structure itself. Moreover, the architecture is even based on a client/server model in which the client requires the execution of an action by using a REST API layer. Therefore, a LXD daemon should act as a server, and when a client request is received, then it will be accomplished through the underlying subsystem.

LXD does not provide the containerization technology by itself, but it relies on existing solutions such as LXC. Furthermore, the mediation between clients and LXC allows the ecosystem to use the adapter design pattern in order to convert the server interface into that which is exposed to the clients. In fact, the underlying containerization technology forces its users to understand lower-level details about creating and managing containers. This requires a lot of initial knowledge to understand what they do and how they work. Of course, this limitation is faced by LXD that provides the additional feature with no difference according to the operating system paradigm. This is obviously quite important because of makes of the ecosystem a valid alternative to existing virtualization solutions such as VMs.

2.4.2 Containers

Containers objects are the core of LXD ecosystem. They can be created, updated and deleted. Most of the methods for operating on the container itself are asynchronous whereas the operations for getting information about the containers are synchronous. As other container implementation, even LXD container is made

of a file system, known as rootfs, which is useful to work with files within the container file system. Furthermore, these are operating system containers, and so there is no need to use a layered file system. Each container owns a bunch of devices such as UNIX disks, blocks, and network interfaces. The container execution environment is set through profiles from which the container inherits data. Furthermore, the configuration settings, such as resource limits, environment, and security options, are defined with a list that is associated with the container state.

2.4.3 Snapshots

Container snapshots are identical to containers except for the fact they are immutable and so they can be renamed, destroyed or restored but cannot be modified in any way. This allows us to store the container runtime state. This is the ability to roll back the container including its CPU and memory state at the time of the snapshots. Furthermore, this is an important feature, not well-included in previous solutions of container operating systems, that is fundamental to the so-called live-migration. In fact, even in cloud deployments, this allows us to move a virtual server from a compute node to another one while the instance continues running.

2.4.4 Images

LXD is image based, and so all containers come from an image. These are typically clean Linux distribution images similar to what is used for a virtual machine or a cloud instance. It is even possible to make an image from a container which can use then by the local or remote LXD hosts. Images are identified by their sha256 hash and can be referenced by using their full or partial hash. Aliases can also be set as one to one mapping between a unique user-friendly string and an image hash.

Now, LXD supports two image layouts, unified or split. The first is what LXD uses when generating images by itself. On the contrary, the split format consists of two distinct tarballs that are commonly used by users who are focused on rolling their own images with an existing compressed filesystem tarball. However, each of those two formats is effectively LXD-specific though the latter makes it easier to reuse the filesystem with other container or virtual machine runtimes. Furthermore, it is focused on system containers, and so it does not support any of the application container “standard” image formats.

2.4.5 Profiles

Profiles are information that describes the configuration state of a container. Any number of profiles can be applied to a container, and they are applied in the specification order. In any case, resource-specific configuration always overrides that coming from the profiles. Moreover, if the object is not defined, LXD creates a default profile which is set for any new containers.

2.4.6 Network

Initially, LXD came with no network defined at all [24]. The initialization command of the daemon provides the possibility to set one bridge up and attach it to all new containers by default. This bridge is called “lxdbr0”. While this certainly worked, it was a bit difficult because most of that bridge configuration was outside of LXD. None of this was exposed over the API, making remote configuration a bit of a pain. That was all until LXD 2.3 when it finally grew its own network management API and command line tools to match. An example is the possibility to define a network and attach it to a container. With LXD we have the support to DHCP and DNS server, which is run on the bridge. Furthermore, the network subsystem makes it very easy to define anything from a simple-host network to a very complex cross-host network for thousands of containers. In addition, it is also simple to define a new network for a few containers or add a second device to a container, connecting it to a separate private network.

2.4.7 Storage

For a long time, LXD has supported multiple storage drivers. Users could choose between zfs, btrfs, or plain directory storage pools but they could only ever use a single storage pool. With LXD it is possible to support not just a single storage pool but multiple storage pools. This is accomplished with its own storage management API [25].

A new LXD installation comes without any storage pool defined. However, by initialization, the lxd daemon creates a storage pool on which containers are created to. By default, LXD attaches storage volumes to the container with write capabilities. This means that the lxd process needs to change the ownership of the storage volume to the container identifier. However, storage volumes can also be attached to multiple containers at the same time. This is fundamental for sharing data among multiple containers, but it is possible only if all containers share the same identifier mapping.

The two best options for use LXD are ZFS and btrfs. They have about similar functionalities, but ZFS is more reliable. These have an internal send/receive mechanism which allows for optimized volume transfer. Therefore, LXD uses those features to transfer containers and snapshots between servers. This has led to LXD achieving great performance that can be evaluated in contrast to using the traditional server virtualization model. Moreover, understanding if containers can replace virtual machines is the goal of this work, and so a further dedicated chapter will discuss a performance analysis between the two deployment models.

2.4.8 Closing remarks

At the end of the chapter, we understood that containerization represents a valid approach to run systems also from the application point of view. In fact, as happened with cloud computing, the paradigm can be addressed from a higher level in which the focus is based on the platform service. This is what in containerization literature is called container management because the solutions involved relies on a containerization engine without showing the infrastructure level. This has influenced the classification between application and system containers. Application contaiers are used for a user who wants to simply package and deploy a single enterprise component, whereas system containers are meant to be treated like hypervisor-based virtual machines in which the workload can involve multiple services. Figure 24 shows us the increasing adoption rate of Docker ecosystem.

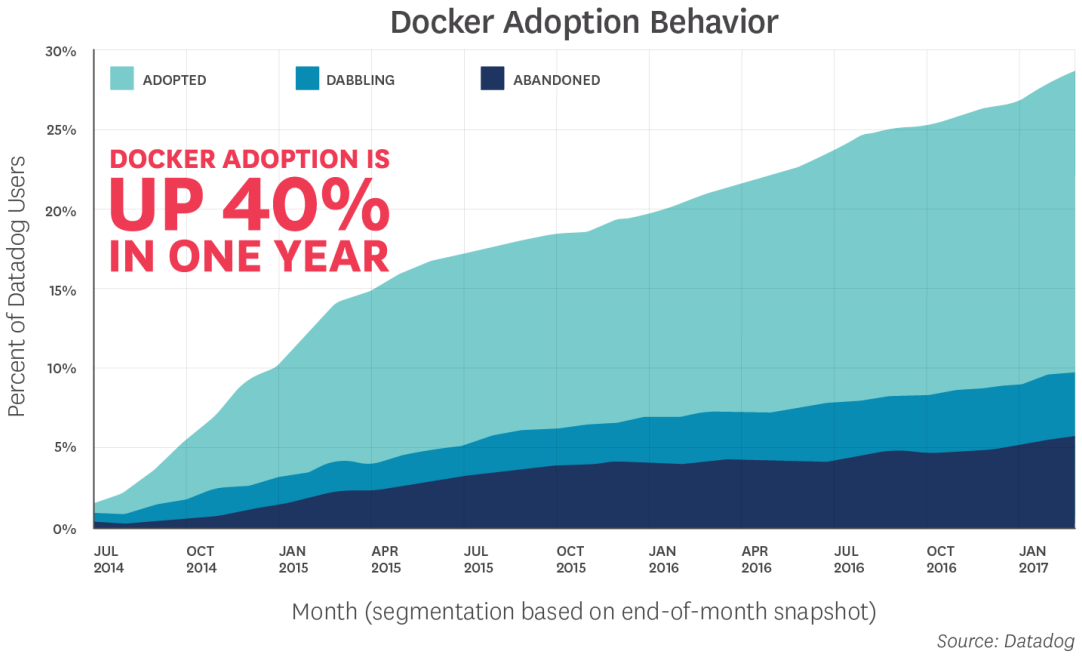

Figure 24 - Docker Adoption Trend

This interesting report [26] is coming from production use cases and the numbers here might look less than Docker findings. Docker is the most popular solution in the area of application containers even if it does not meet the needs of every IT requirements. In fact, when Docker was found to have several security issues, another solution, Rkt, was designed to be more secure, interoperable and according to an open format runtime.

Since then, Docker has covered a lot of ground in addressing all critical issues, but it is a worthwhile comparison to note how these two platforms differ in their various capabilities. So, the next table shows us a list of features and how these two platforms provide each one. Nevertheless, these are only the major container platforms available for production use now, and for this reason, at any point, users must evaluate available technologies against deployment factors, such as security and operating system overhead. Table 3 presents a functional comparison between Docker and Rkt.

	Docker	**Rkt**
Container image security	Docker Content Trust, introduced since Docker 1.8	By default, with Rkt signature verification
Root privilege attacks	There is the need to use security solutions such as SELinux or AppArmor	There is no need to enforce the security model because each container is never created from a privileged root process
Flexibility in publishing or sharing images Size of code base	There is the need to set up a special private registry or use a Docker paid account	It is enough to have a web server which is able to operate through HTTPS
Size of code base	Each new version causes the increasing of code lines of the whole program	The modular architecture allows confining modifications on the single new block
Portability to other container systems	Now it uses an open standard which is called “Open Container Initiative.”	Uses an open source format known as “Appc”

Table 3 - A comparison between Docker and Rkt ecosystems

As it is possible to see, even if Docker is quite spread in the container market, there are several issues that Rkt was able to face. This is also characterized from the architectural point of view in which Docker is based on a tight client/server model and therefore it is not suitable to be integrated with external supervisor system processes such as systemd. In fact, each container is always created from a client command, and a client fail is detected as well as the container was stopped, even if this is not properly the case. Rkt was designed to face Docker issues basically and for this reason, it gets over these problems.

Docker, Rkt and other application containers focus on ephemeral, stateless, minimal containers that are typically not involved in upgrading but instead just be replaced entirely. LXD focuses on system containers, also called infrastructure containers. Those containers will typically be long-running, based on a clean distribution image, and they are used as we would use them for a traditional virtual machine. That makes Docker and similar projects much closer to a software distribution mechanism than a machine management tool. Therefore, in cloud deployments, LXD is more suitable because it can be used in order to perform operations according to the specific service request level. In fact, recently it has been integrated into cloud solutions, such as OpenStack, whereas Docker is no longer maintained by the Nova project considering the fact that is properly focused on application containers.

The OpenStack integration of LXD allows us to create instances on that ecosystem in the same way that it would normally create virtual machines running on a traditional hypervisor such as KVM. Security has been a key principle from the design stage and, on the contrary of other solutions such as Docker, an excellent isolation was built from the start rather than as an afterthought. Furthermore, it provides scalability and trusted sources which are fundamental to create full filling systems.

3 Container Orchestration Engine

3.1 Overview

Until now, we focused on the different types of server virtualization and how containers can be used in production in order to provide the same benefits of the traditional virtual machines. The need for business agility [27] has led to commercial pressure for more frequent deployment of software. In order to support this, new software development techniques (known as 'agile') and operational cultures (such as 'DevOps') have taken hold. Therefore, applications increasingly tend to be built from existing components and a modern design involves the use of multiple components, even with a smaller number being written in-house. This has led to a new design trend in which the application is entirely composed of microservices, small independently deployable services which communicate over a network.

As seen in the previous chapter, containers provide an ideal vehicle for such components due to their low overhead and speed of deployment. Furthermore, they are also suitable for efficient horizontal scaling by deploying multiple identical containers of the relevant component. Modern applications thus might be built from hundreds or even thousands of containers, potentially with complex interdependencies. Nevertheless, according to this new trend of application design, the usage of container solutions, such as the application containers which we have already investigated, is quite limited and difficult to be adopted. Therefore, these issues were addressed with the introduction of a higher containerization level which is called "container orchestration". The purpose of this chapter is to investigate this new layer and the widespread solutions, considering the fact that they are also fundamental in production cases of cloud computing.

3.2 The need of Orchestration

When container solutions emerged, there were no orchestrators designed for containers. Today, there are a lot of such solutions, like Docker Swarm, Kubernetes, Mesos and more, that it can be difficult to know which one to adopt. At the most basic level, all container orchestrators do the same thing: they automate the provisioning and management of containerized infrastructure [28]. It was introduced in order to face with continuously automated scheduling, coordination, and management of complex systems of containerized components

and the resources they consume. It is also worth noting that orchestrators are not strictly limited to the container world. In fact, orchestration tools like Juju cloud orchestrator had existed for other types of infrastructure when containers became popular.

However, orchestrators are particularly important in a containerized environment because we have a lot of components, and managing things by hand is likely to fail. Therefore, the increasing adoption of container solutions stimulated the introduction of new capabilities that can be distinguished in functional and not-functional qualities. Among these, we can find scheduling, resource management, and service management. In fact, the scenario is a set of machines whose kernel holds a container engine in order to deal with containerized applications.

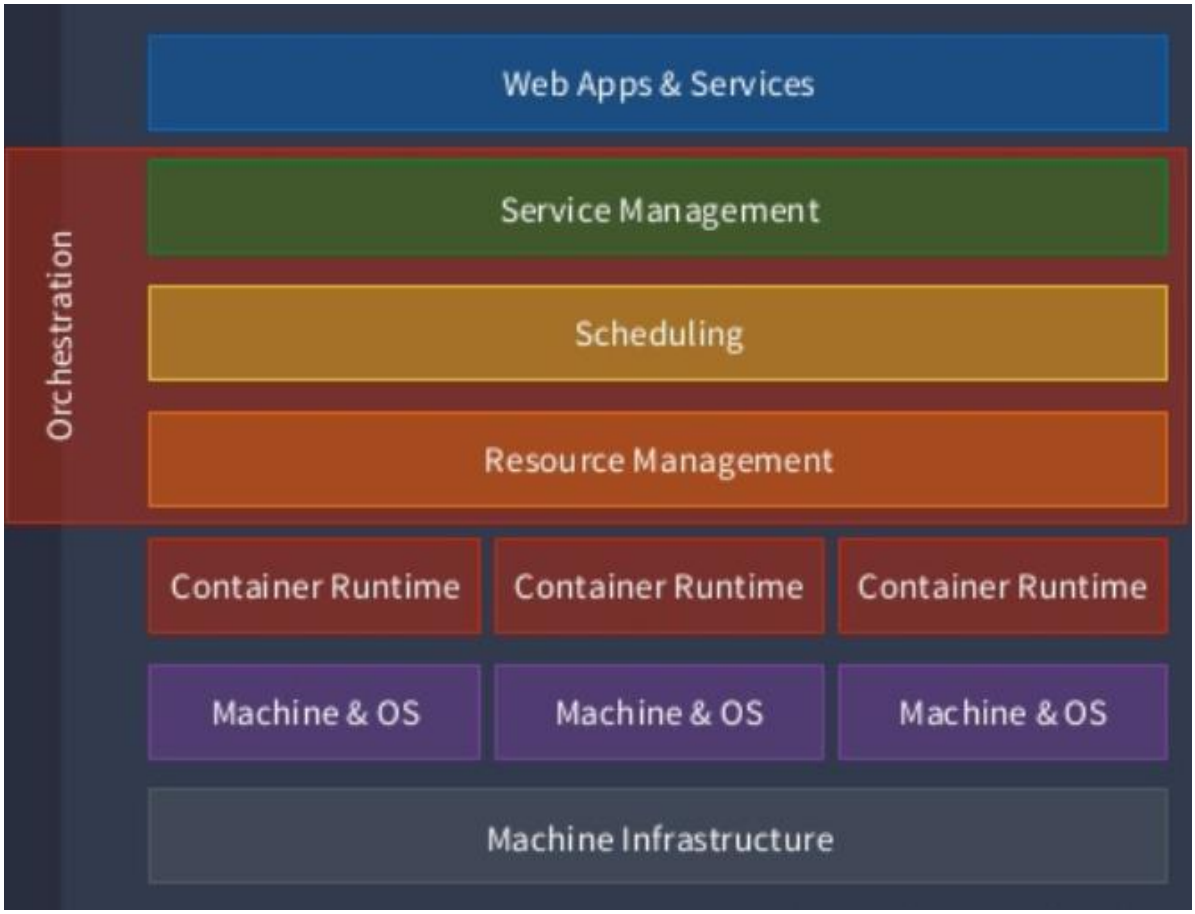

Figure 25 - Capabilities of Orchestration Layer

Figure 25 illustrates the layered structure of the container orchestrator.

Furthermore, the concept of orchestration is not new, and it is even present with managing systems which could be run on bare metal or virtual machines. The usage of orchestration is often discussed in other contexts such as service-oriented-architecture [29] or converged infrastructure [30]. The orchestration is

about aligning the business request with the applications, data, and infrastructure. It defines the policies and service levels through automated workflows, provisioning, and change management. This creates an application-aligned infrastructure that can be scaled up or down based on the needs of each application.

As usual in distributed computing, it is fundamental to trade off the right overhead in order not to make heavy the system. For this reason, it is necessary a resource management layer to manage low-level resources, such as Memory, CPU, Volumes, and more. Lastly, considering the fact that the focus is addressed to the application and not to the infrastructure, it is important to include a service management functionality in order to provide functional capabilities to build and deploy enterprise applications quickly.

3.3 Docker Swarm

Docker Swarm is a clustering and scheduling tool for Docker containers. This allows IT operators to manage a cluster of Docker nodes as a single virtual system. Nowadays, even containerization needs the important feature of clustering [31]. This is an important aspect because it creates a cooperative group of systems that provide redundancy and enable the failover mechanism if one or more nodes experience an outage. Furthermore, the orchestration tool provides to administrators the centralization where to manage and control the whole system.

As it will be seen later, the orchestrator is based on the master/slave model in which the master is the main component of the whole cluster. It is that node which is responsible for scheduling containers whereas a slave node is responsible for launching the received containers. Also, Docker Swarm provides the ability to add or subtract container iterations as computing demands change. This is obviously important in cloud environments where the elasticity is one of the most important features. Moreover, Docker Swarm continues to use the standard Docker application programming interface to interact with other tools, such as Docker Machine. This means that a Docker user does not find any difference to work with a single machine or with an entire cluster.

3.3.1 Docker Clustering

A cluster is a group of servers and other resources that act like a single system enabling high availability and, in some cases, load balancing and parallel processing. Since a cluster is a logical rather than a physical unit, the size of a cluster can be varied.

Working with distributed systems involve meeting long latencies and unexpected failures. Building a cluster is a solution that can be used in order to prevent these problems using more robust hardware and better network interconnections.

With Docker, this strategy requires reasons deeply because organizing containers to run across a fleet of machines is not a trivial task [19]. It used to be the case that we would deploy different pieces of software to different machines. Furthermore, with Linux containers for isolation and Docker for container management, the remaining major concerns are the efficiency of resource usage, the performance characteristics of hardware, network locality.

3.3.2 Architecture

A Swarm cluster is made up of two types of machines: a machine running Swarm in management mode called a manager and another one, which is called docker node, that runs a Swarm agent [19]. Both types of nodes are just like any other Docker machines. These agents require no special installation or privileged access to the machines, but they run in Docker containers.

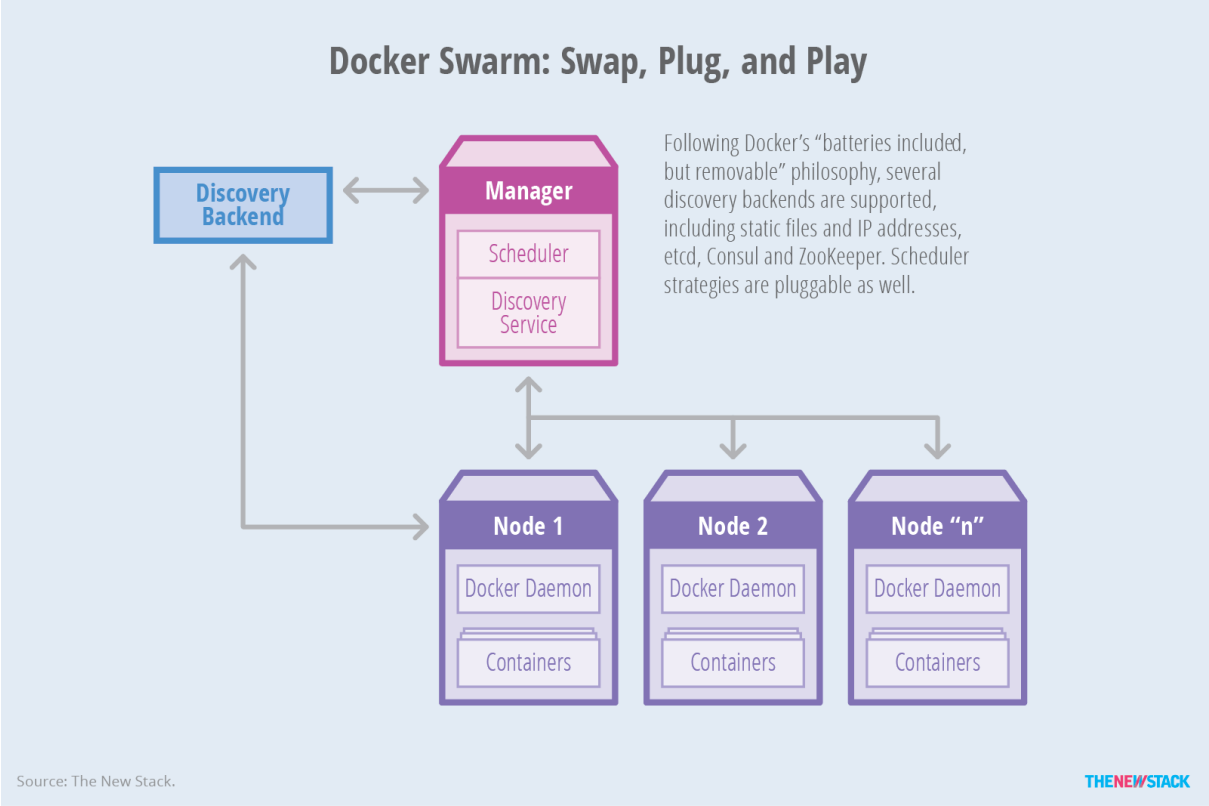

Figure 26 - Docker Swarm Architecture

Figure 26 shows us the architecture of Docker Swarm. The only difference between Docker Swarm and Docker standalone is a small set of additional command-line parameters that are included with the "create" subcommand.

Building up a swarm cluster requires specifying the machine which should act as Swarm manager. This means that a particular agent will be placed in order to enable additional functionalities to provide the cluster mode. After that, each slave node needs to be joined with the swarm manager through an agent component which is running on. Furthermore, every type of machine in a Swarm cluster requires a way to locate and identify the cluster it is joining.

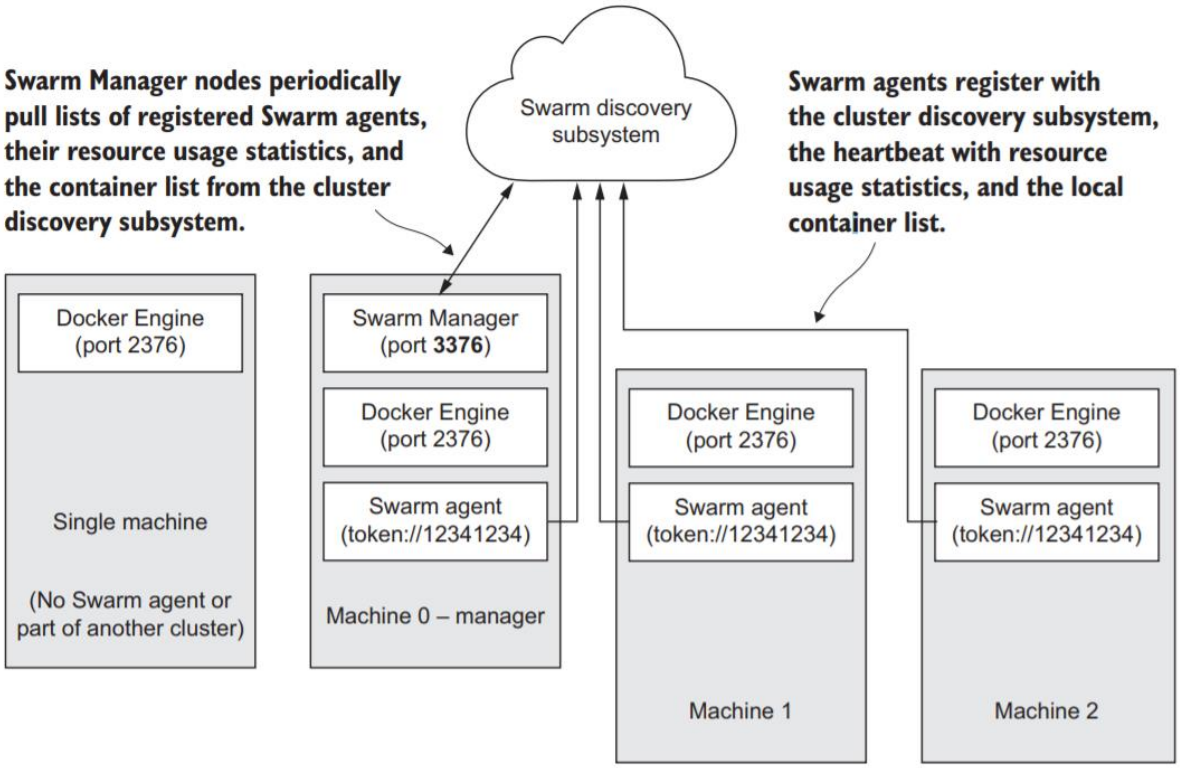

Figure 27 - The discovery subsystem in Docker Swarm

Figure 27 represents the way in which a swarm manager and agents interact to discover available services on the built cluster. This involves that manager nodes need to check the lists of registered Swarm agents periodically. However, this is not enough because to fit other mechanisms, such as scheduling, it is fundamental to catch information on resource usage, and a container list. On the contrary, Swarm agents need to register with the cluster discovery subsystem.

Like other Docker projects, Docker Swarm follows the “swap, plug and play” principle [32]. Therefore, it is possible to swap out the pre-defined scheduling backend using an out-of-the-box solution. This feature is quite important considering that it is suitable for most use cases, and is suitable for large-scale production deployments for more powerful backends, like Mesos.

3.3.3 Docker Swarm API

Docker Swarm Manager endpoints expose the Swarm API. Clients can use that API to manage or inspect a cluster. Moreover, the Swarm API is an extension to the Docker Remote API [19]. In fact, any Docker client can directly connect to a Swarm endpoint as well as if it were a single machine.

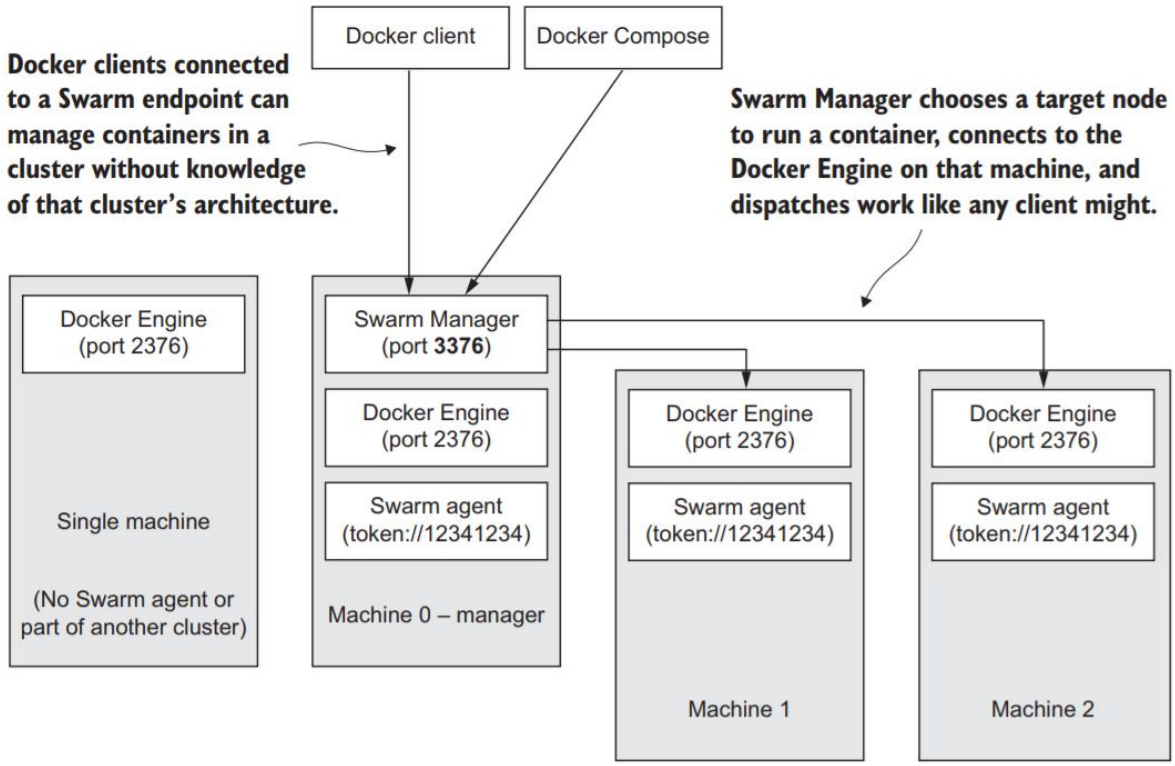

Figure 28 – A Swarm cluster with a simple Docker client

Figure 28 shows us how it is possible to use the same docker client to interact with a swarm cluster. However, the implementation of the Docker Remote API is different from the Docker Engine. In fact, depending on the specific operation, a single request from a client may impact one or many Swarm nodes.

3.3.4 Swarm scheduling

Docker Swarm provides three different scheduling [19] algorithms. Each of which has its own advantages and disadvantages. The algorithm is set at the moment in which a Swarm manager is created, and the user can tune the scheduling algorithms for a given Swarm cluster by providing constraints for specific containers.

The first one is the so-called spread algorithm. It will try to schedule containers on under-used nodes and spread a workload over all nodes equally. The algorithm specifically ranks all the nodes in the fleet by their resource usage and then ranks

those with the same resource rank, according to the number of containers which are running on. This algorithm works best in situations where resources reservations have been set in containers, and there is a low degree of variance in those limits. As the resources required by containers those provided by nodes diversify, the Spread Algorithm can cause issues.

The second one has fine-tuned scheduling with filters. Before the Swarm schedule applies the algorithm, it gathers and filters a set of candidate nodes, according to the Swarm configuration and the needs of the container. Each candidate node will pass through each filter of the configured cluster, which is used to customize every scheduling algorithm.

The last ones are BinPack and Random. The BinPack scheduling algorithm prefers to make the most efficient use of each node before scheduling work on another. This algorithm uses the fewest number of nodes to support the workload.

Random provides a distribution that can be a compromise between Spread and BinPack. Each node in the candidate pool has the same opportunity of being selected, but that does not guarantee that the distribution will realize evenly across that pool.

BinPack is particularly useful if the containers in the system have high variance in resource requirements or the project requires a minimal fleet and the option of automatically downsizing. Whereas the Spread algorithm makes the most sense in systems with a dedicated fleet, BinPack makes the most sense in a wholly virtual machine fleet with scale-on-demand features. This flexibility is gained at the cost of reliability.

3.3.5 Swarm service discovery

A distributed system needs some mechanisms to find its pieces. When processes are placed on the same machine, some named shared memory pool or queue will be used. However, if the components are designed to interact over a network, they need to agree on names for each other and decide on a mechanism to resolve those names. Most of the time, networked applications rely on DNS for name-to-IP address resolution. Container links are managed by Docker through static configuration into the name-resolution system of the container [19]. However, this is not suitable in distributed environments where the docker engine has no visibility of services running on other hosts. Therefore, the goal of the Swarm project is to provide a “batteries-included” but the optional solution for clustering

containers. However, this needs the development of several technologies and enhancements to the underlying Docker Engine.

3.3.6 Swarm and single-host networking

The Docker Engine creates a local network behind a network bridge on every machine where it is installed on. This means that on a Swarm cluster, deployed on machines that operate with the single-host network, containers can discover each one running on the same host. Nowadays, is more popular to have clustered applications. These are viable for some use cases in spite of this limitation, but the most common scenarios are underserved. Therefore, server software typically requires multi-host distribution and service discovery.

3.3.7 Swarm and multi-host networking

Actually, the Docker network system is implemented by three types of drivers which are: bridge, host, overlay. While bridge and host are used to implement single-host networking features, the overlay driver implements an overlay network with IP encapsulation or VXLAN.

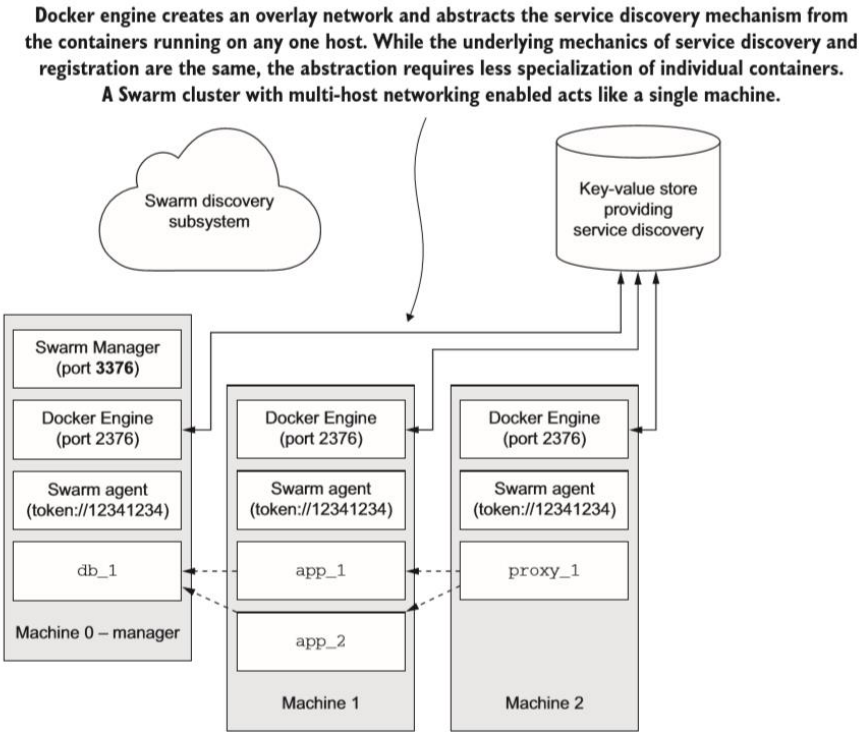

Figure 29 - MultiHost Networking with Docker Swarm on top of Docker Engine

Figure 29 shows us a swarm deployment with the support of multi-host networking. By this way, each container gets a unique IP address that is routable from any other container in the overlay network. All that work is performed in the

infrastructure layer provided by Docker and the integrated key-value store. This abstracts container locality from the concerns of the applications within Docker containers. Each container will act as a host on the overlay network.

3.4 Kubernetes

We have already learned that, as the number of deployable application components grows, it starts to become hard to manage them all. Google was the first company that realized it needed a much better way of deploying and managing their software components and their infrastructure if they were going to scale globally. However, this was enforced by the fact that the company ever faces system execution on a great number of servers. This has led them to build solutions for making the development and deployment of thousands of software components manageable and cost-efficient [33]. Initially, they developed an internal system called Borg (later changed to Omega), which helped both application developers and system administrators to manage the huge amount of applications and services. After having kept Borg and Omega secret for a whole decade, in 2014, Google introduced Kubernetes, an open source system based on the experiences gathered through Borg, Omega, and other internal Google systems.

Kubernetes is a system for managing containerized applications across a cluster of machines [34]. It was designed to address the lack between how modern clustered infrastructures are designed and some of the assumptions that most applications have about their environments. Therefore, it enables to run software applications on thousands of server nodes as if all those nodes were a single computer.

Users do not need to see the infrastructure level because the platform abstracts it. Therefore, deploying applications through Kubernetes is always the same with no difference if the size of the cluster is continuously changing. As Docker Swarm, Kubernetes is based on a master/slave architectural pattern in which the developer submits a list of apps to the master and, subsequently, the platform takes care to deploy them across the worker nodes.

For this reason, Kubernetes is considered as an operating system for the cluster because it shows users the whole set of resources as a single and central point management. Furthermore, application developers do not need to implement certain infrastructure-related services because they can rely on Kubernetes to provide these services. This includes functionalities such as service discovery,

scaling, load-balancing, self-healing and also leader election. This makes of Kubernetes the most adopted solution in the orchestration market. In fact, due to this useful set of features, developers are able to focus on the business core and not waste time figuring out how to integrate it with the infrastructure.

3.4.1 Architecture

Kubernetes is an open-source platform for deployment and management of applications based on containers executed on a cluster of machines. Even if it is a container orchestration system, the whole project consists of a complex architecture which aims to expose container-centric APIs to users. Those do not need to care about infrastructure management and low-level components such as compute, storage and network. The architecture is based on master-slave architectural pattern, and it is also designed with an open layer interface. Therefore, it is possible to customize the platform extending the behavior of such components. To do that, components do not directly interact, but they have been designed to have decoupled interactions. The project is quite modular, and each function is made up of components. Each entity provides services and uses APIs of the Kubernetes core. Furthermore, the asynchronous communication pattern guarantees flexibility and the possibility to use the project in a customizable way.

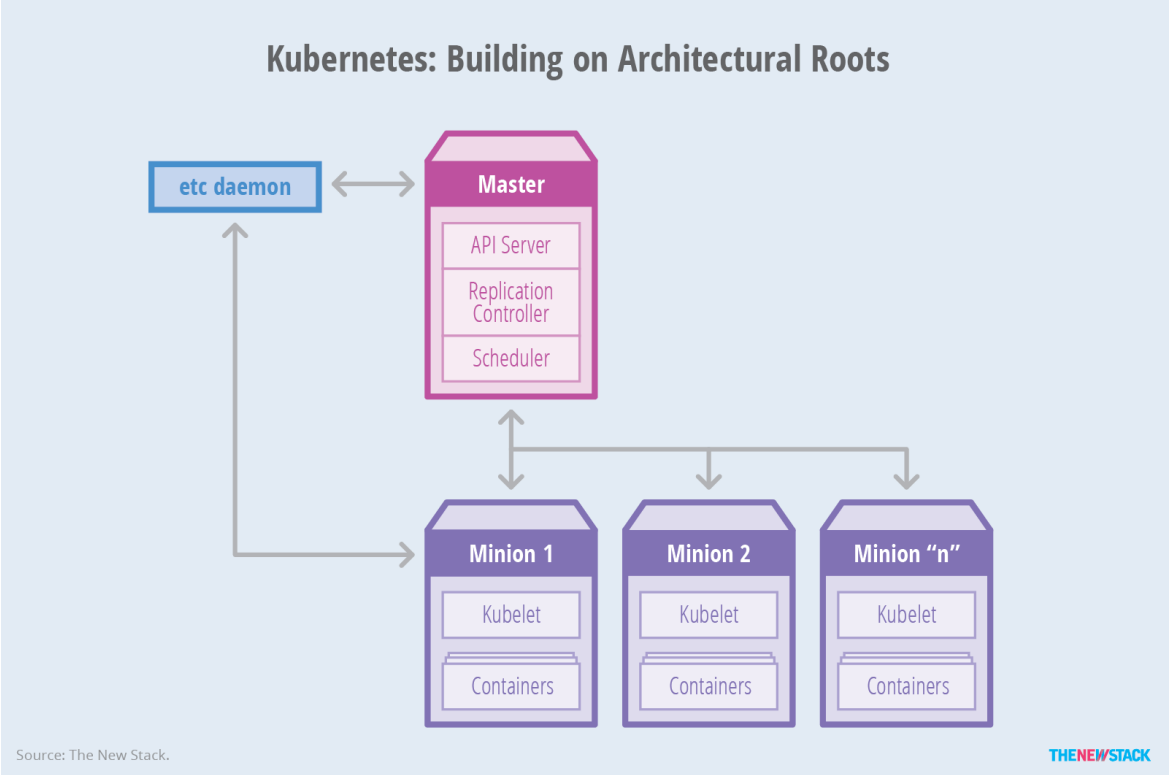

Figure 30 - Kubernetes Architecture

Figure 30 illustrates the architecture components of a Kubernetes cluster.

As it is possible to see, there are three main entities: a master node, one or more minion nodes, and a persistent data storage system. The master represents the control plane of the cluster, and it can be replicated to guarantee high-availability and fault-tolerance. It is composed of different components, each of which is used to implement a specific functionality. One of those is the module that exposes REST-APIs. These are dedicated to providing the four basic functions of persistent storage, properly known as creating, read, update, delete (CRUD) [35].

User access these APIs and, after succeeding the authentication phase, are able to work with Kubernetes objects to orchestrate them on the whole cluster. Therefore, the REST layer represents a shared point that every user accesses to work with the platform. Considering that Kubernetes can be easily customized, it is possible to modify some specific parts such as the container engine or the replication module.

At runtime, there will be components whose job is to guarantee a specific quality of service. Therefore, it is necessary a monitoring process that will be used to check if the current state is the same of the desired state specified by the user. Of course, the interaction between these components is completely decoupled and without a single centralization point of failure.

Another fundamental component of the master node is the cluster state data store. Typically, etcd is used, and its job is to store data necessary for components which have to check the current state of the system. Moreover, to guarantee the desired state, there is a server component which is called Controller-Manager. The major application functionalities are included in that component. It is a separated process, and its responsibilities are lifecycle and business logic management. On the contrary of other components, it is more monolithic.

The focus of the Kubernetes is multi-container applications, typically enterprise service components. The concept of multi-container is implemented on a platform object which is called “pod”. By design, Kubernetes provides a containerized application as a set of containers, each of which is specific for a single microservice. A pod is a set of containers which are involved to be deployed as the smallest atomic unit of the architecture. This means that containers placed in a pod will be located on the same machine of the cluster. In fact, the platform takes care of the whole pod, and the chosen minion will receive the whole structure of the Kubernetes object.

Users communicate to Kubernetes that a pod needs to be located on a machine running on the cluster. Which decides to place the pod is the scheduler, another component of the master node. Furthermore, the framework supports user-provided schedulers which are useful when customers want to adapt the scheduling behavior to their needs.

The master node is the control plane of the cluster, but there is no pod executed on that because it is reserved to hold just management functionalities. Containers can be executed on slave nodes, called minions or Kubernetes nodes. Therefore, these must contain the necessary components to guarantee the execution of the Kubernetes applications. Nevertheless, it is even important that the control plane obtains information about the cluster state. For this reason, there should be a functionality that communicates to the master information about the current state of each minion node.

A minion node consists of three main components: Kubelet, a container runtime, Kube Proxy. Kubelet is the most important component of each slave node, and it is an agent whose purpose is to perform platform-specific actions. Without the presence of that component, the project cannot work as a cluster orchestration system.

Furthermore, Kubernetes uses containers with complete isolation. However, this does not follow the principle of the design model used in traditional operating systems. In fact, isolation is guaranteed not only between containers but also between containers and the underlying host.

Kubelet is the slave component which is responsible for deciding, accordingly to its own strategy, which pods can be executed on the host which it is running on. This means that, even if the master node takes the scheduling decisions, the final arrangement it is up to the Kubelet component.

In order to get information about containers, which are currently running on the host, Kubelet is linked with another component which is called cAdvisor. Data collected in this way are fundamental to build atop a monitoring system that can be used by the Controller-Manager, but also to perform fine-tuned scheduling strategies.

As mentioned before, Kubelet is not the unique infrastructure component of a Kubernetes slave. In fact, there is also a container runtime that is necessary to manage the lifecycle of containers. This represents the container management

layer of the solution. For this reason, it is responsible for downloading images and to execute containers.

There is no strong relationship between Kubelet and the container engine. Thus, an open layer definition of the container runtime has been designed in order to make modular the underlying infrastructure. This allows us to perform evaluation tests between different providers. Actually, the supported container engines are docker, rkt, cri-o, and frakti. This was also influenced by the competition between different container solutions such as Docker and rkt.

Kubernetes was one of the first mover supporting the introduction of a different model like Rkt. Considering the fact that design integrates the concept of the pod, such solution guarantees to the platform a better way to manage the architecture without no need to create specialized infrastructure components. Moreover, Kubernetes has always been quite opened to the container solutions by defining an open infrastructure level able to support multiple implementations. However, this was not the case of other competitors such as Docker Swarm, which does not provide any solution outside the Docker world. Of course, this has influenced the Kubernetes strategy to be more accepted due to its own openness and flexibility.

The last service component in the slave node is the so-called Kube Proxy. It consists of a simple process which executes on every node to configure the iptables firewall. As it will be seen further, Kubernetes specifies the concept of "service" to represent an applicative component which is reachable by other clients.

In fact, the relationship between a server component and clients is not fixed and, even if the server component can change its own IP address, there should be a mechanism that allows both pairs to communicate. Therefore, Kubernetes uses this strategy in order to configure the underlying firewall that is aware of the current state of service components. Furthermore, this component is responsible for performing load balancing in order to choose the destination of a service request between multiple instances of the same pod.

This is the high level of a Kubernetes deployment, but it is not completely the whole platform. In fact, there are other functionalities that can be included such as the following:

- DNS – local domain name system to resolve the association between the name of services and the associated network address.
- Ingress Controller – component to route external service requests in order to centralize some internal services into a single-entry point.
- Heapster – component to enable container cluster monitoring and performance analysis.
- Dashboard Graphic User Interface – a web-based user interface which is used to manage the cluster obtaining an overview of the whole cluster resources.

3.4.2 Network

Kubernetes networking is approached somewhat differently than Docker does by default. In fact, the network sub-system defines distinct aspects of dealing with:

- Highly-coupled container-to-container links: solved by pods and localhost communications.
- Pod-to-Pod communications: the primary focus of this section.
- Pod-to-Service and External-to-Service communications: covered by the Kubernetes object “Service”.

By design, Kubernetes allows to pods the possibility to communicate with other pods, regardless of which host they land on [36]. Every pod gets its own IP address, so it is not necessary to explicitly create links between pods by mapping container and host ports. This creates a clean, and backward-compatible, a model where pods can be treated much like virtual machines or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

To do that, Kubernetes [37] has defined a specific networking model imposing the following fundamental requirements on any network implementation (barring any intentional network segmentation policies): each container can communicate with other containers without techniques of network-address-translation (NAT); all nodes can communicate with all containers (and vice-versa) without NAT; the IP that a container sees by itself is the same network address that others see it has.

This means that with two Docker-compliant hosts, it is not guaranteed that Kubernetes works well. In fact, it should be ensured that the fundamental requirements are primarily met. Furthermore, this model is achieved through a quite number of implementations that can be adopted. Each pod gets its own IP address but, as mentioned before, a typical Kubernetes deployment involves different use cases in which is not easy to maintain an association between the logic name of the service and the pod IP address. This concerns about multiple instances of a specific pod which implement the same service or the case in which a pod has been scheduled on a different node following a host outage. In that case, it is difficult to maintain the binding between the logic and the physical name of that service component.

Therefore, Kubernetes introduces another resource: the concept of "Service". It is important to make easier the interaction with pods, regardless of the networking model adopted by the underlying implementation. A typical component exposes services to outside, and so it is fundamental to have a sort of mechanism that allows anyone to reach the component without no difference if the client is inside or outside the cluster.

There are many cases in which the micro-kernel design pattern is adopted. This means that a pod can be a client of another pod, currently executing inside the cluster. Therefore, it is necessary to have a name resolution system that is independent of the underlying infrastructure.

Thus, a service in Kubernetes is just a mechanism on top of the underlying communication infrastructure. In fact, it identifies a specific component that can be implemented by many pods. Each service is created with an IP address and a port that does not change. This allows any client to reach the server pointing out the same IP address and port, even if the location of the underlying pods change during execution. This is completely hidden to clients who just see the concept of a high-level service, and no constraint involves a specific host to run pods which are associated with a service.

Furthermore, this creates a sort of load-balancing mechanism if the service is implemented by many pods, each of which is distributed all over the cluster.

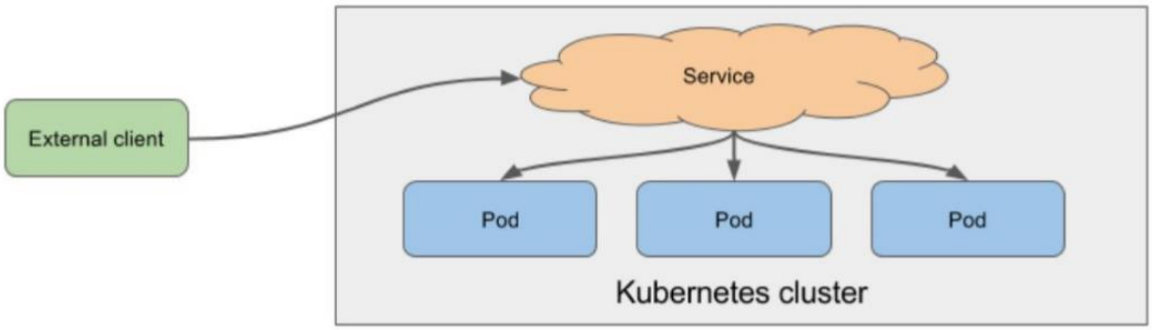

Figure 31 – A Kubernetes Service exposed to application clients

Figure 31 shows us a high-level view of the service concept introduced in Kubernetes. The mechanism used to be aware which pods implement a service is also involved in the behavior of the Replication Controller. This is the Kubernetes component which is responsible for making sure that a certain number of pods are currently providing a specific service. Therefore, a service is built with a label and the implementation pods are characterized with the same labeling system. These labels are used by the so-called Kubernetes selectors which are responsible for investigating the association between an object and a specific label.

After completing the creation phase, a service gets two principal addresses, which are respectively called Cluster-IP and External-IP. The first one is the address accessible just inside the cluster while the second one is exposed to external clients. Obviously, while a cluster-IP is always included with a service definition, an External-IP depends on the infrastructure provider on which the Kubernetes cluster is built on. Therefore, a Cluster-IP is not a routable address, and so it makes sense just inside the local deployment. Therefore, when a request is addressed to a Cluster-IP, this is intercepted by the local kube-proxy, which opportunely forwards the message to an endpoint of that service.

Kubernetes was the first project to propose this different network model, and therefore other competitors decided after to follow the principle of multi-host networking. Of course, it is needed to consider that application developers and operators are quite linked to the traditional orchestration model and so different solutions, such as that originally adopted by Docker, was not so properly suitable. In fact, Kubernetes started to move towards the direction in which users need to focus just on the application features, and for this reason, the definition of this type of architecture was completely considered a success.

3.4.3 Storage

Kubernetes introduces the support even for the storage sub-system. As seen in Docker, this involves the concept of "Volume" [35]. A volume is not an application business resource such as services and pods. For this reason, they cannot be directly created and deleted through HTTP requests to the API server of the Kubernetes master. The concept of Volume is strongly linked to the pod, which is quite similar to a virtual machine. Moreover, containers that are executed inside a pod share different namespaces, such as the network stack. Nevertheless, there are also cases in which it is important to share storage devices, as well as between different processes which are executing on the same virtual machine.

By definition, a container file system comes from the image which the container has been created from [37]. They are isolated, and so there is no mechanism to share data between containers that belong to the same pod. Kubernetes faces this lack through the concept of "Volume". Considering the fact that a volume is not a high-level resource, its own lifecycle is strongly dependent on the pod which is associated to. This means that a volume is initialized when a pod is created and evicted at the destruction of that pod. A specific type characterizes a Kubernetes volume and the definition of these types is explained in the table below.

Volume Types	**Description**
EmptyDir	Simple empty directory used for storing transient data
HostPath	Useful for mounting directories from the underlying node file system into the pod
GitRepo	Volume initialized by checking out the content of a Git repository
NFS	A network file system type of volume which is mounted into the pod
Cloud disk format	Useful for mounting cloud provider specific storage (Google, AWS, Azure)
Network storage	Other types of network storage (Cinder, Ceph, Gluster, and more)
Security types	Used to expose certain Kubernetes resources and cluster info to the pod
PersistentVolumeClaim	A way to use a static or dynamically provisioned persistent storage

Table 4 - Kubernetes Volume Types

Table 4 shows the different volumes types of Kubernetes.

This is a fundamental information that characterizes a volume, each of which is suitable for a different use case and can be useful for several applicative scenarios. It is possible to mount a volume at a specific part of the container file system. This guarantees that data written by containers will not be lost even if containers are started again.

In Kubernetes, volumes are classified into two types: temporary and persistent. The first ones are “emptyDir” and “gitRepo” whereas the remaining are all persistent volume types. As the name suggests, temporary volumes concern about a directory created at the initialization and destroyed when the pod will be deleted. This is not suitable for components whose state needs to be maintained even after the execution of the pod. Therefore, it is important to use the other volume types which allow having long-persistent data.

Furthermore, these different types of the volume are characterized by data which are stored locally and others which rely on a network infrastructure. This requires that the developer needs to be exposed at infrastructure level services. However, this is not the Kubernetes principle, whose aim is to leave developers to focus only on business aspects.

In fact, the infrastructure management should be reserved for the cluster administrators. Thus, the platform should include the possibility by which the developer just specifies a specific amount of storage that the application needs. Which decides to provision that service is responsibility of Kubernetes that needs to match the request with the underlying infrastructure provisioning. Nevertheless, the principle is the same used in an application process that requires hardware resources such as CPU, memory, and more.

For this reason, Kubernetes has introduced two other important concepts: Persistent Volumes and Persistent Volume Claims. The first is the Kubernetes object corresponding to the underlying physical resource whereas the other one is an object which is associated with an applicative service request. Therefore, the cluster administrator takes care about registration of persistent Volumes while users will use them through the concept of PersistentVolumeClaim.

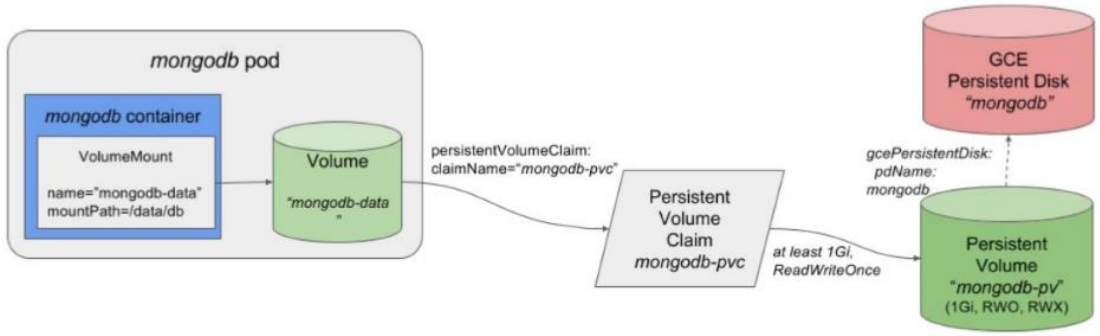

Figure 32 - A Kubernetes application whose storage is taken by Google Cloud,

Figure 32 shows us the case in which a Kubernetes user defines a specific storage object. On the contrary of other solution like Docker Swarm, users specify just the minimum amount of storage and the access mode of that resource. After sent this information to the Kubernetes API server, if it is possible, the association will be dynamically accomplished by the mediation of the Kubernetes middleware.

Furthermore, Kubernetes supports the possibility to relax the constraint to have always a cluster administrator that needs to create the correspondent Kubernetes resources. In fact, there is the possibility to delegate this duty to Kubernetes itself, through the use of a dynamical provisioning of persistent volumes. As shown in Figure 33, in this case, the cluster administrator needs to deploy a so-called "PersistentVolumeProvisioner" and define one or more "StorageClass". StorageClass are used by users to specify what types of persistent volumes they need, but the component which takes care of provisioning is the "Persistent Volume Provisioner". Nowadays, Kubernetes is well-adopted in cloud deployments, and therefore it statically includes the provisioning support for the major types of cloud providers.

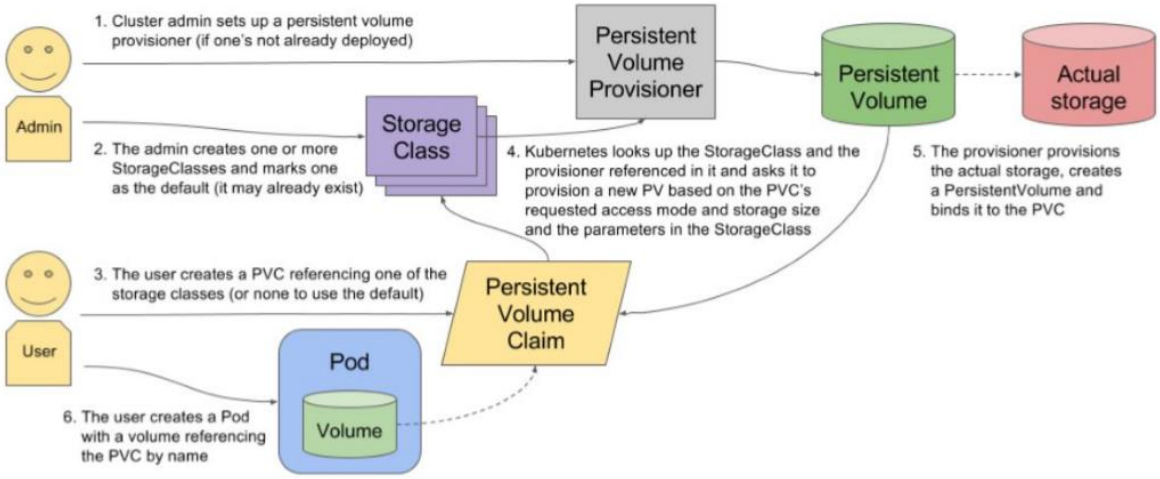

Figure 33 - Kubernetes Storage Provisioning with cloud features

This completely integrates a Kubernetes deployment through a weak-coupled relation with the cloud provider. In fact, it is possible to change the underlying storage provisioning with no effects to the application point of view of the business component. Of course, this makes of Kubernetes a sort of featured player because it was able to face these issues that existing solutions have never thought to deal with.

3.4.4 Scheduling

The Kubernetes scheduler is a policy-rich, topology-aware, workload-specific function that significantly impacts availability, performance, and capacity [37]. It requires getting an in-depth individual account and collective resource requirements. Furthermore, it is designed to consider different aspects such as quality of service, applicative constraints, affinity specifications, data locality, inter-workload interference, deadlines, and so on. These requirements are exposed through the APIs as necessary. The scheduler is not just an admission controller; for each pod that is created, it finds the best candidate machine for that pod, and if no machine is suitable, the pod remains unscheduled until a machine becomes suitable.

Furthermore, the scheduler component is quite configurable [35]. Basically, it supports two policy types, which are respectively called “FitPredicate” and “PriorityFunction”. “FitPredicate” requires rules to satisfy a specific request even if there is no available candidate who is able to face it. In fact, in these cases, the building pod is not scheduled on any nodes. This is the case in which the pod remains in “Pending” state until it can be satisfied by one of the Kubernetes slaves.

Furthermore, if the scheduler finds that multiple machines are able to host the pod, there is the possibility to work with a fine-grained strategy. That is where priority functions come in. Basically, the scheduler ranks the machines that meet all of the fit predicates and then chooses the best one. This allows users to choose a specific policy according to the infrastructure requirements. Nevertheless, scheduling component is able to be dynamically modified. Therefore, users can decide which “fit predicates” and “priority functions” are desired to apply.

3.4.5 High-availability

One of the most important features of Kubernetes Orchestration System is the possibility to design an application with high-availability, without requiring no action of the system administrator. An example is the case of a web application.

If the main process crashes, Kubernetes takes care about that and performs a recovery action to fail over the problem. However, another failure type is the outage node where the web process is running on. In this case, it is necessary that the system is able to detect the event and subsequently schedule the component on a different node.

Kubernetes includes the failover mechanisms [37] to deal with both cases. When a fault concerns about the main process of a container, the event is detected by the kubelet agent, which is installed on the slave node. After that, the agent relaunches the container through the local container engine. However, there are cases in which is not possible to detect the event of process-fault such as deadlock, etc. In this case, the agent cannot detect the event unless there is a communication mechanism between the agent and the application.

Nevertheless, Kubernetes deals with this issue by introducing a communication system between the agent and the application. This mechanism is called "stay-alive," and they are messages that are sent by the kubelet agent in order to know the health state of the container. Furthermore, these messages are distinguished between three probe types: HTTP GET, TCP Socket, and exec command. The first one consists of an HTTP message sent to an HTTP server which is listening on the container. The return status code allows the agent to be aware of the container service soundness.

The second one, TCP Socket consists of the initialization of a TCP connection with a server that is listening on the application container. If the connection is established, the agent recognizes this event as a good health state of the container. The last one possibility consists of executing a command inside the container, and the agent knows the health state through the exit status code of the process.

Moreover, it is even possible to configure the probe delay if the container needs an amount of time, before becoming available to response probe messages. If no probe message is declared, the kubelet agent will detect just the crash state of a container. Of course, this requires the evaluation of a right trade-off in order not to stress the system because the monitoring process includes a not negligible overhead.

However, if the whole slave node crashes, there is no way to failover the service if that component will not be scheduled on a different slave node. Therefore, Kubernetes introduces the definition of a management component that is called

"Replication Controller". It is a framework object whose task is to maintain the desired number of pod instances.

If the current state is not the same of that desired, like in cases of an outage node, automatically the Replication Controller will start the scheduling of a new instance on a different slave node. This is possible because the creation of that pod is delegated to this component which holds a template of that application service.

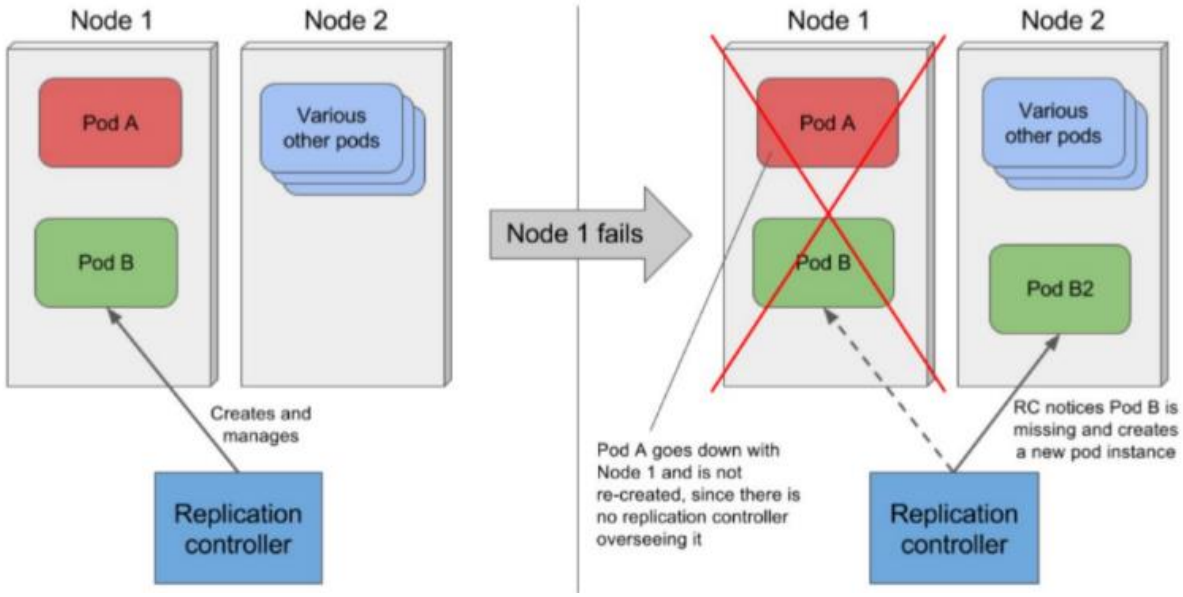

Figure 34 - Kubernetes Replication Controller with node failures

Figure 34 shows us the case in which a pod is under control of a specific Replication Controller. Therefore, each pod is distinguished in managed and not. Pods not managed by a ReplicationController will not survive a crash of the slave node which they are running on. Thus, in order to guarantee the desired state, the unique possibility to face these cases is to perform manual actions.

Furthermore, the controller mechanism of the heartbeat is not too aggressive and it is designed with a subscriber notification system that notifies the execution of some actions on the API server.

These fundamental settings characterize a ReplicationController: label selector, a specific number of instances desired, a pod template. The label select is necessary to point out the pod to be monitored. The template is the structure of the pod to be used when is necessary a new creation of that instance, while the "replica count" gives the controller the desired state for that specific pod. There is no difference between controller and other framework resources. This means that is possible to change the label selector or the template. Changes on label selectors

imply that the set of managed pods will change, whereas changing template will influence updates just on newer instances, which are created by the ReplicationController.

It is always possible to take pods from the managed state out of the scope of a ReplicationController. To do that, it is necessary to modify the label selector. An important update concerns about the new version of a specific component. This is about changing the template of the ReplicationController which manages its own pods. However, even in this case, the updates will be applied after the creation of new instances, and so it is necessary that the controller detects a variation between the desired and the current state of the system.

Another important resource is the so-called ReplicaSet. It is quite similar to the ReplicationController because it is a specialization. The unique difference is that the behavior of label selectors can be customized with sophisticated expressions. The controllers saw until now concern just failover to maintain the desired state of a specific application component.

Requirements of high-availability interest even other aspects such as component whose aim is to perform a job or others whose instances need to be just one for each slave node. To deal with that, Kubernetes introduces the concept of Job, DaemonSet, and CronJob. Job takes care of controlling that a task can be correctly executed without no constraint to maintain it long-running.

On the contrary, the second one is used to guarantee that, for each slave node, a specific task is running on. This is quite useful to system components such as framework agents whose presence is fundamental for the soundness of the whole platform. Lastly, CronJob interests tasks whose behavior is finished but also repeatable. It is an extension of the Job Controller whose aim is to guarantee that a specific task completes the execution. Furthermore, it is suitable for a periodic scheduling of the job that is fundamental to perform particular actions at specific time intervals.

3.4.6 Service Discovery

Each service gets an IP address and a port, regardless of the implementation pods are continuously changing their endpoints. Therefore, clients should have an access mode always to reach the service. For this reason, it is necessary a mechanism to discover the IP address and the port. Kubernetes gives different ways to do that [33]. An example consists of using environment variables. When

a pod is initialized, a set of environment variables are associated with the pod. One of these variables contains even information on services. This makes sure that a pod is always able to find out the address and the port of a service through a classical mechanism of discovering. This is not the standard form of traditional discovering such as DNS.

Nevertheless, Kubernetes includes this resolution possibility. In fact, within system pods, there is a particular pod, which is called kube-dns. This is a DNS server that automatically configures the executing pods on the cluster, involving the update of the local DNS file. This is an important feature because clients are able to reach the service, knowing just the fully qualified domain name of the component. Furthermore, this architectural design pattern has been recently included in complementary solutions, like Docker Swarm but the way in which the discovery process is performed is quite different.

3.4.7 Quality of Service

Kubernetes takes care of health state of pods by sending liveness probes to containers with the purpose to detect a crash state. This allows starting a container again when a liveness probe fails. However, a fundamental requirement is not just high-availability but also the quality of service. An example concerns about services whose pods need to be accessible when a client performs a request. When a pod implements a service, it is quickly added to the list of endpoints that are available.

Furthermore, there are many cases in which pods need a setup time in order to be ready to respond. For this reason, Kubernetes has introduced the so-called “readiness probes”. They are similar to liveness probes, and the classification includes the following: HTTP request, TCP connection, and exec command. On the contrary of liveness probes, a failure, in this case, does not imply a failure and consequently reboot of the container. Simply, this means that the pod cannot be associated with an endpoint in the list of the services. This allows to forward messages just to ready pod and clients will not be served from not ready components.

3.4.8 Cluster Federation

One of the most important benefits of Kubernetes is to maintain high-availability even if a node failure, or local congestion, happens. If the organization maintains its own services over different datacenters, the solution seen until now do not allow us to work guaranteeing the same requirements. Therefore, Kubernetes has

introduced the support to different cluster installed over different locality and even with an underlying infrastructure provided by a different cloud provider. This is called “Cluster Federation”. Due to this support, we can maintain requirements even if a fault hits a specific datacenter such as disaster recovery events. Furthermore, another suitable use case is about the fidelity level that a cloud administrator has with a cloud provider. It is more likely that an organization do not want to install every data on a specific cloud provider. Therefore, a sort of backup services is created, and so a good level of uptime will be guaranteed.

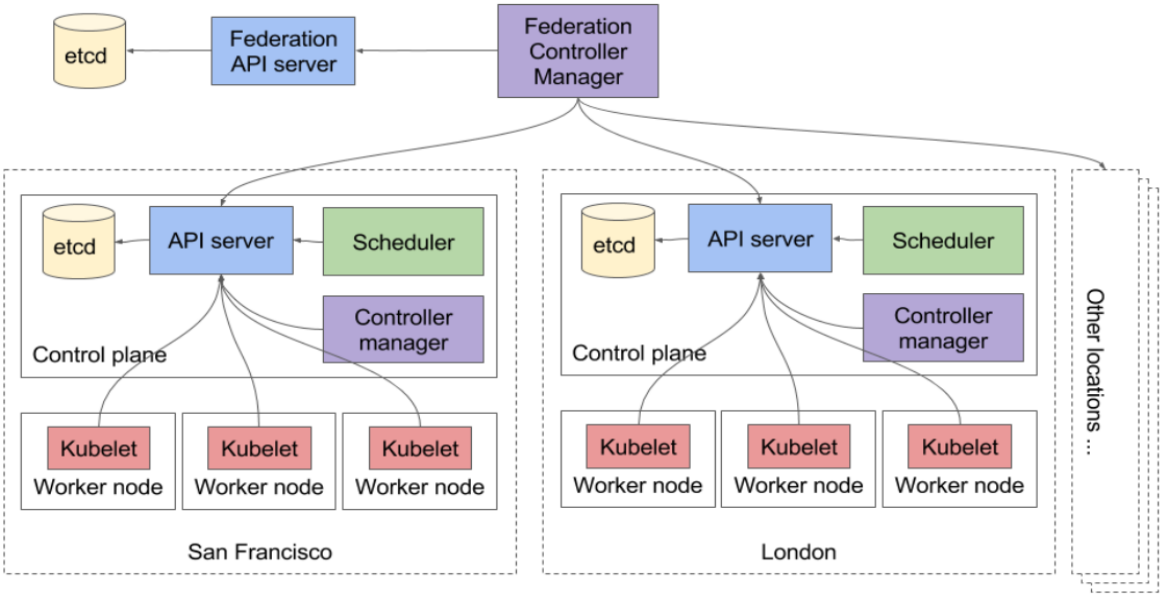

Figure 35 - Kubernetes Cluster Federation System

Figure 35 illustrates the architecture of cluster federation in Kubernetes and, as it is possible to see, it is not so different from the standalone architecture. In fact, it is simply an enhancement of the existing solution; whose aim is to manage the federation between different clusters that belong to the federated cluster. In this case, we have a high-level master, which is called Federated Control Plane, while every single cluster is known as cluster worker.

The Federated Control Plane consists of the following three parts:

- Etcd, to store persistent objects through federated API;
- Federation API server;
- Federation Controller Manager.

Etcd is used to store federated objects which are registered through the REST endpoint (Federation API Server). The Federation Controller Manager [33] contains a list of federation controllers, each one of these executes operations

depending on the federated object involved in the invoked API. Users interact with the Federation API Server [37] which creates federated objects. These are stored in the storage federated etcd. Furthermore, federation Controllers subscribe to the control plane in order to receive events associated to the correspondent federated objects. When it is necessary to execute an action, the federation controller interacts with the API server of underlying clusters to propagate the creation of regular Kubernetes resources. When a ReplicationController is used, the desired behavior is not to create a number of objects for each underlying cluster. The principle is to extend the strategy at the federation level, and so the desired state is expressed at the federated level and not single-cluster.

Nevertheless, the synchronization is only one-directional, from the federation server to the underlying cluster. Modifying the resource on the underlying cluster, changes will not be notified to the API Server, taking to a not consistent state. When an entire cluster faults, the federated control plane detects the event and, if some fault-tolerant strategy is configured, the system will make sure that on the remaining clusters available the resource will be installed.

However, federating clusters has not just advantages. In fact, it is possible to point out disadvantages that in a single-cluster deployment these are not present. Surely, this involves higher overhead because the system uses a greater amount of network. The federation control plane should check every cluster and control that the desired state is the same of the current state. So, if clusters are placed on the different and far region, this includes a not negligible overhead.

Another disadvantage is that a bug on the federation control plane causes a fault on the whole system. For this reason, it’s fundamental that the federation control plane should act as simple front-end and delegate requests to correspondent single-clusters.

3.4.9 APIs to extend Kubernetes

Kubernetes was designed to provide a set of defined resources in order to model an application using pods, volumes, Replication Controllers, and more. Each resource type is defined by a manifest, described with a YAML or JSON format. Furthermore, the implementation consists of a controller that registers itself on the API Server in order to receive events associated to the correspondent managed resource [33]. This implements the desired behavior executing low-level actions, characterizing the most popular way to use Kubernetes.

Nevertheless, one of the most important features of Kubernetes is the possibility to extend the platform customizing the open source project. In fact, it is possible to create custom objects and define the business logic of controllers, whose aim is to manage custom objects. Kubernetes is composed of components completely decoupled and independent, with a weak relationship necessary to guarantee flexibility. This allows us to modify the runtime, even choosing a different container-engine. All of that has influenced the adoption of Kubernetes in may projects, such as OpenShift, Deis, WorkFlow, and Helm.

3.4.10 Platform as a Service and OpenShift

Since Kubernetes operates at the application level rather than at the hardware level, it provides some generally applicable features common to the paradigm of Platform as a Service (PaaS), such as deployment, scaling, load balancing, logging, monitoring. However, the decoupled architecture makes it able to be pluggable and extensible. This results in a system that is easier to use and more powerful.

Red Hat OpenShift is a Platform-as-a-Service with the purpose to allow the developer to focus only on designing services. This means that the target is to quickly provide results obtaining at the same time scaling and long-term maintenance of applications. The original versions of the platform did not have any link to Kubernetes. However, the last release(III) was completely designed with a distributed platform atop of Kubernetes. There is a strong separation in the responsibility model to design services: Kubernetes takes care of managing runtime changes and application scaling, while OpenShift concerns about building images and correspondent deployments, without integrating any Continuous Integration solutions such as Jenkins, etc.

Furthermore, OpenShift introduces even the concept of user and group to deal with multi-tenancy of Kubernetes clusters. This means that users have access to their own namespace. The application which is executing in their namespace are completely network isolated. Basically, OpenShift is an extension of Kubernetes with the introduction of the following resources: Users and Groups, Projects, Templates, BuildConfigs, DeploymentConfigs, ImageStreams, Routes. Furthermore, this extension opened the integration with cloud computing feature that Kubernetes has never met yet. One of these is multi-tenancy. So, the platform service includes the concept of users and projects, as a workspace to host single tenant workload. Each user has access to the project, practically speaking

Kubernetes namespaces. Furthermore, user accesses to projects are managed by cluster administrators.

OpenShift allows to users the possibility to define configurable manifests. These are called Templates, which are a list of objects whose definitions can include several parameters. These values are set when the template is initialized, and so developers can have a sort of existing solutions which are spread for the most applicative use cases.

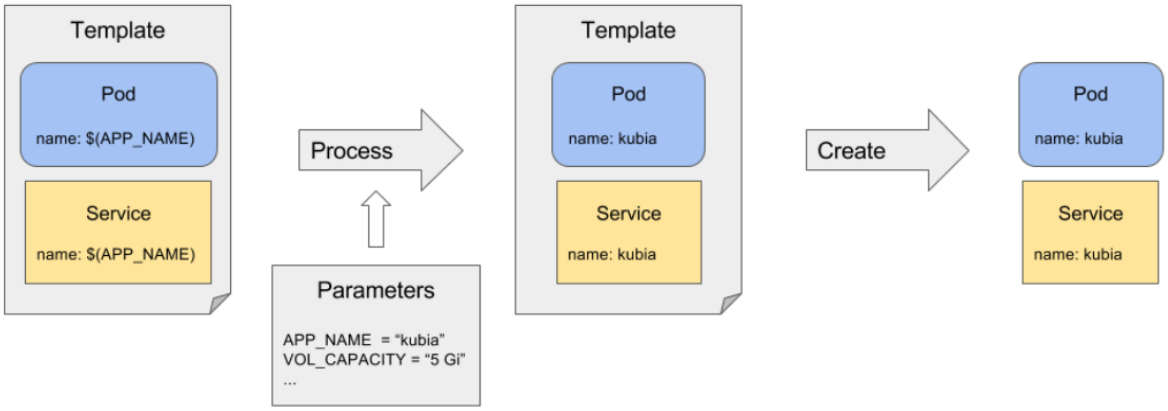

Figure 36 - OpenShift Templates

Figure 36 shows us the process of building Kubernetes application through the high-level concept of OpenShift Template. These are simply pre-defined manifest, with a set of parameters, that is given to a logic component in order to build a set of common solutions.

One of the most benefits of OpenShift is the possibility to enable the automatically run after completed the build process of the application. In this case, there is no need to execute the build with the container image. This is possible due to the OpenShift resource which is called "BuildConfig". To do that, a git repository is used, but the implementation does not monitor that repository. So, this is accomplished by a hook to that repository that notifies OpenShift the execution of a new commit command. After that, OpenShift will execute the pull operation from the repository and start the execution of the entire process.

Furthermore, another important OpenShift resource is the so-called "DeploymentConfig". When a container image is built, this will be added to an ImageStream. This consists of a notification queue that holds each image after completing the build process. When the DeploymentConfig detects that a build has been done, it starts the rollout of the new image. This is the case when an application needs an upgrade or update modifications.

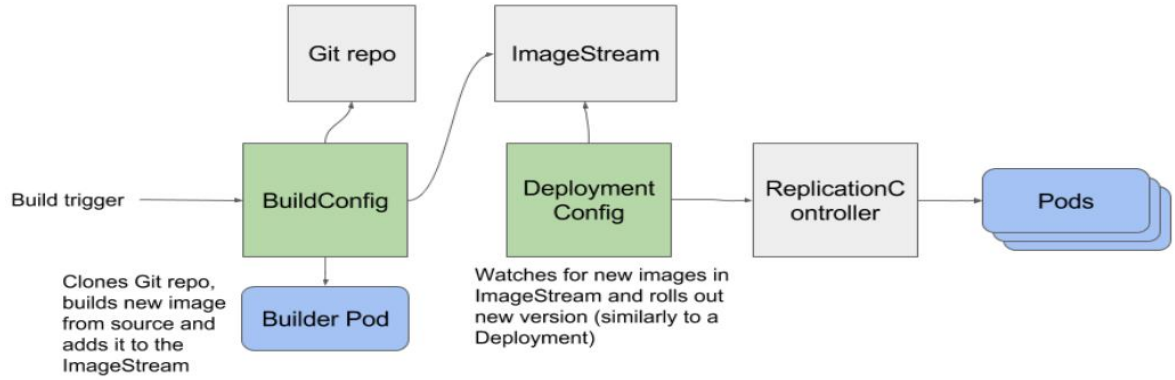

Figure 37 - BuildConfig and DeploymentConfigs in OpenShift

Figure 37 recaps what we have just discussed about the deployment configuration in OpenShift. This consist of a notification system that is used to detect the definition of a different application component. This involves a chain execution that as a result yields to the rollout of the application, without no need to have management downtime.

3.5 Apache Mesos

Apache Mesos is an open source project designed before Docker. Basically, it is a platform whose aim is to manage computer clusters using Linux Cgroups [38]. In fact, the purpose of the system is to provide CPU, I/O, filesystem, and memory isolated resources. It is described as a cluster platform which provides computing resources to frameworks. The primary benefit of this project is the level of abstraction it provides. In fact, there is no lock-in with a cloud provider or datacenter vendor infrastructure. Basically, it joins the whole set of resources as a centralized logic control system.

It was built to support mixed workloads of long-running (application) and short-running (batch processing) processes and jobs. The main principle of Mesos [39] is to provide computing resources to frameworks, such as Hadoop, Spark and so on. In fact, there may be multiple frameworks running on a Mesos cluster for different kinds of task and users interact with frameworks rather than directly with Mesos. Sharing improves cluster utilization and avoids per-framework data replication. Studies have asserted that Mesos can achieve near-optimal data locality when sharing the cluster among frameworks. Furthermore, it is quite resilient to failures.

Mesos [40] has a different philosophy than Docker Swarm and Kubernetes, which are both container management tools with a cluster-to-cluster relationship. Mesos is more of a resource allocation manager that allows users to manage both Docker

and non-Docker jobs. In fact, the execution of framework jobs is accomplished using native operating system features, and not necessarily with Docker containers. This offers us the possibility to build a specific executor binary in order to launch an isolated process within the Mesos infrastructure. That is properly what many cluster frameworks, such as Hadoop, Spark, and more have already done. Furthermore, there is no tight-coupled relation between the application level and the underlying infrastructure. In fact, Mesos provides a framework to launch heterogeneous workloads onto the same cluster.

Nevertheless, Mesos works between the operating system and the application layer. The aim is to provide dynamic allocation of resources of the underlying data center, and this has not led the project to be quite spread in the orchestration market. For this reason, in the segment of container orchestration, Mesos is now used in conjunction with another framework which is called Marathon. This serves as a container orchestration platform on top of Mesos, providing functionalities such as scaling and self-healing for containerized workloads. In fact, it is focused on application level, and users can interact with the platform as well as with other existing solutions such as Docker Swarm or Kubernetes.

3.5.1 Architecture

Mesos has an architecture that is based on the master/slave model. Furthermore, in addition to the two types of node, it comprises frameworks which are placed atop of the entire infrastructure. This introduces another issue that concerns about the process in which it is needed to provide scheduling. As it will be seen in the next section, Mesos uses a multi-level process in which frameworks continue to take scheduling decisions while the core of Mesos provides to them resource offers [41].

The architecture consists of a master/slave design pattern, in which the execution of tasks is delegated to slave nodes. The master process, running on the manager node of the cluster, takes care of management and monitoring of the whole cluster architecture. Therefore, it needs to communicate with frameworks whose aim is to schedule jobs on the slave nodes. This means that after an explicit declaration of the local component to the framework, the master has to launch the task execution on a slave node. To do that, it is necessary that each framework gives to the master a special component able to instantiate jobs for that solution. These components are called executors, simply implemented as processes running on slave nodes.

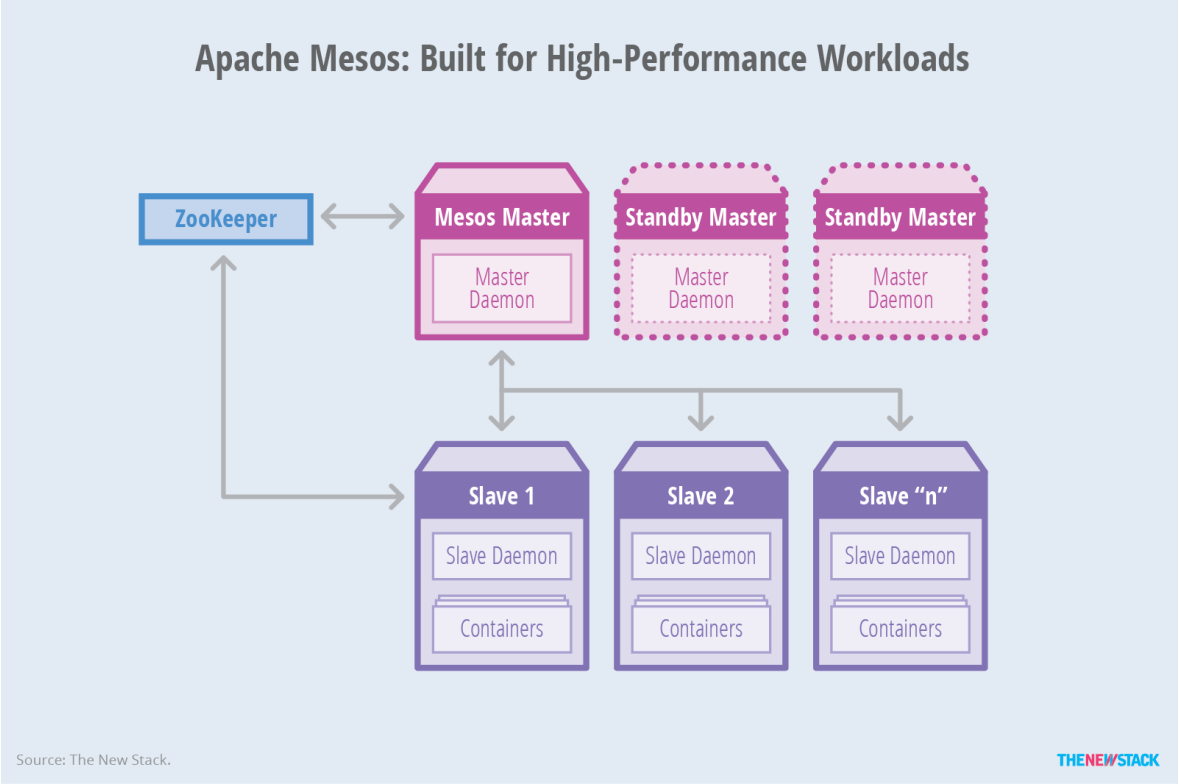

Figure 38 - Mesos Architecture

Figure 38 shows us the architectural model of a Mesos.

The resource sharing strategy [42] is applied to the master through a sophisticated mechanism: it decides to offer a set of resources to each framework. The strategies are two: fair sharing and strict priority. However, Mesos gives us the possibility to extend its own strategy because the master is designed through a modular architectural pattern. Furthermore, Mesos is suitable to offer also high-availability. This consists of replicating master nodes in order to provide failover mechanisms in case of master failures. To do that, many solutions can be used even if the most adopted implementation is Zookeeper. This consists of an election algorithm which determines the new node which has to play the master role.

3.5.2 Scheduling

A framework installed on Mesos consists of two components: a scheduler and an executor process [28]. The scheduler is the component responsible for deciding if accepting or not the resource offers. In that case, it will specify what resources will be used among those offered by Mesos platform. The executor is the framework-dependent process which has to instantiate scheduled tasks on the node which it is running on. Furthermore, even with Apache Mesos, there is a possibility to define a custom Mesos scheduler. Basically, there it can be built in

two different ways: by using a C++ interface or by using the new definition of HTTP API. Nevertheless, those schedulers, built on the notification model, are expected to keep the subscription connection open as long as possible.

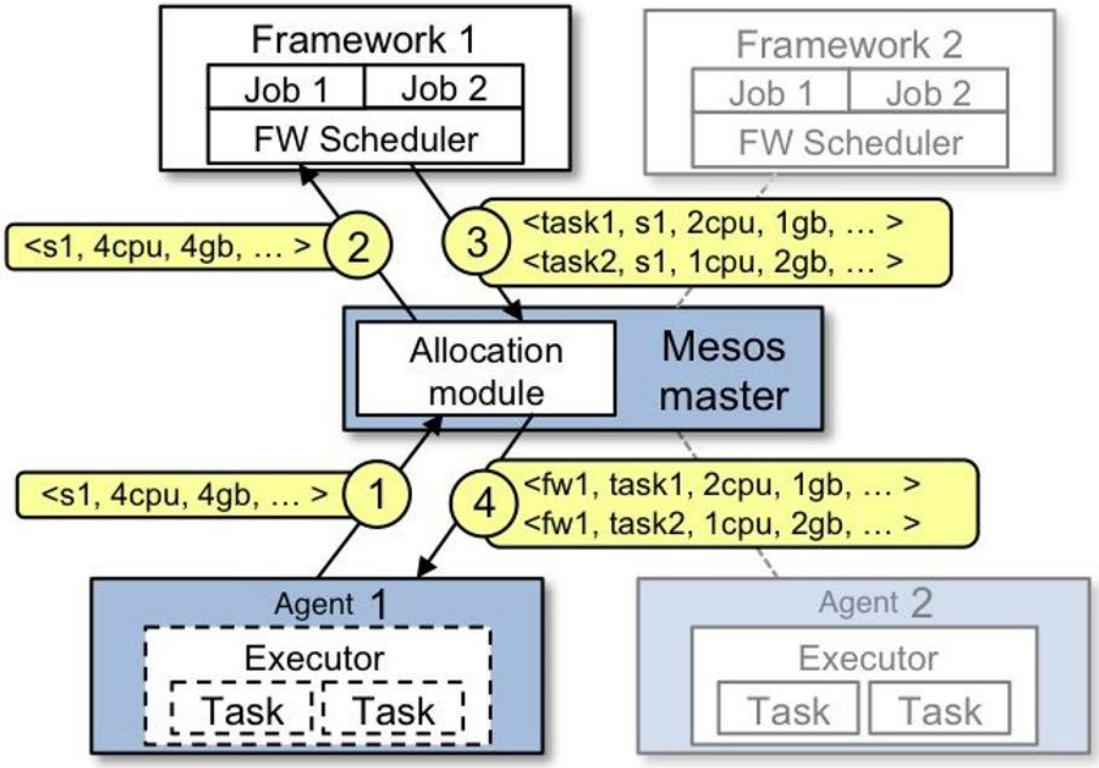

Figure 39 - Mesos Scheduling

Figure 39 shows an example of resource offer. The first step concerns about a slave node that communicates to the master an amount of availability. Subsequently, the master decides how many resources and to which these resources need to be offered. This means that it will perform as an offer to the framework which has to respond if accepting or not the offer.

Each framework has its own internal strategy in order to accept even an amount of resources less than the proposal offered. Obviously, the master is not aware of local information to frameworks. However, statistics shown as using that strategy had not influenced the reach of good performances.

The design principle [43] to locally choose the accepting of a resource offer allows satisfying application requirements because the solution was designed to adopt a locality principle that is independent of underlying Mesos core. However, the assign of resources has not an unlimited time, and so Mesos can decide to preempt the offer after an expiration timeout, which is established in conjunction with the resource offer.

That preemption can be done through a simple communication with the correspondent executor or killing the process associated with that framework. However, this is not an elegant solution because it can cause inconsistencies, considering the fact that an event of this type cannot be handled. Thus, a reserved quota has been designed in order to guarantee a minimum allocation to the framework.

3.5.3 Executors isolation

Executors are run on slave nodes but, considering Mesos principles, it is possible to imagine the case in which multiple executors concurrently run on the same slave. Therefore, in order to get isolation between executors, Mesos adopts two mechanisms: Linux Containers and Solaris Project. These technologies limit the usage of resources such as CPU, bandwidth, memory, I/O devices, and processes. Furthermore, the platform supports a dynamic configuration of limits associated with containers.

At state of the art, Mesos uses container technologies, but in the future, it is possible that complementary solution will be investigated, as virtual machines. Of course, the adoption of container mechanisms allows us to take advantages of isolation already built on frameworks, such as Hadoop. In this framework, tasks are different jobs in the same machine but practically executed as separated processes.

In conclusion, we can affirm that Mesos is a lightweight solution to enable the efficient resource sharing between frameworks in cluster computing. This is enforced by two most important principles: a sophisticated resource sharing strategy and a decentralized scheduling mechanism, which is called resource offers. These make of Mesos a tool that guarantees a fit-for-all cluster solution able to deal with dynamic changes to the system, without no loss of simplicity and scalability.

3.5.4 Marathon

Until now, we have learned what a Mesos cluster focusing on infrastructure management in order to get efficiency and scalability is. However, many solutions have been developed in order to build on top of Mesos cluster a sort of application-level management (orchestrator) [43].

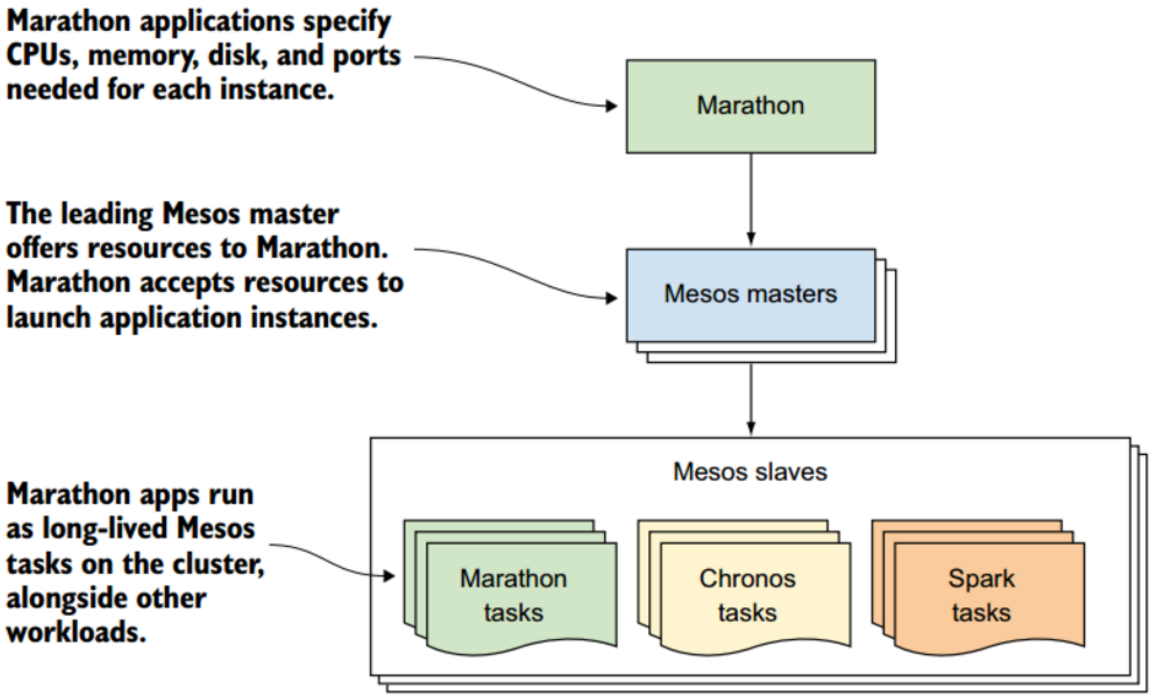

Figure 40 - Marathon Framework on top of Mesos Cluster

Figure 40 shows an architecture overview of the Marathon framework built atop of a Mesos cluster. It allows us to start applications by using Mesos and underlying technologies, such as Linux containers or Docker.

In literature, it is even considered as a sort of private Platform as a Service, which allows us to configure the deployment of a generic application. Furthermore, Marathon is used to specify desired resources for each instance of a generic application. So, it is even possible to define the number of instances that are willing to launch.

As each Mesos framework, Marathon interacts with the master component. The idea of Marathon is to provide orchestration functionalities to the whole Mesos cluster. Thus, if a slave faults, Marathon will start a new instance in order to guarantee the fault-tolerance. Furthermore, this offers high-availability and the support to developers, who can focus on business logic problems and not on underlying infrastructure.

3.5.5 Service Discovery and Load Balancing

When an application is placed on the cluster, it is necessary to manage the addressed traffic to that service. This should be done regardless if the sender belongs to the same cluster or not. Furthermore, in modern microservice applications, this is much more important because service instances have dynamically assigned network locations. Moreover, the set of service instances

changes dynamically because of auto-scaling, failures, and upgrades. Consequently, the client code needs to use a more elaborate service discovery mechanism that is completely infrastructure-agnostic [41].

Apache Mesos has included three solutions to face with these issues, and they are Mesos-DNS, Marathon-lb, and HaProxy-marathon-bridge. Mesos-DNS is a service discovery that uses the domain name system (DNS). It works directly with Mesos, and it is independent of Marathon. Furthermore, it is possible to integrate Mesos-DNS with outside implementation that can be useful when a request cannot be locally resolved.

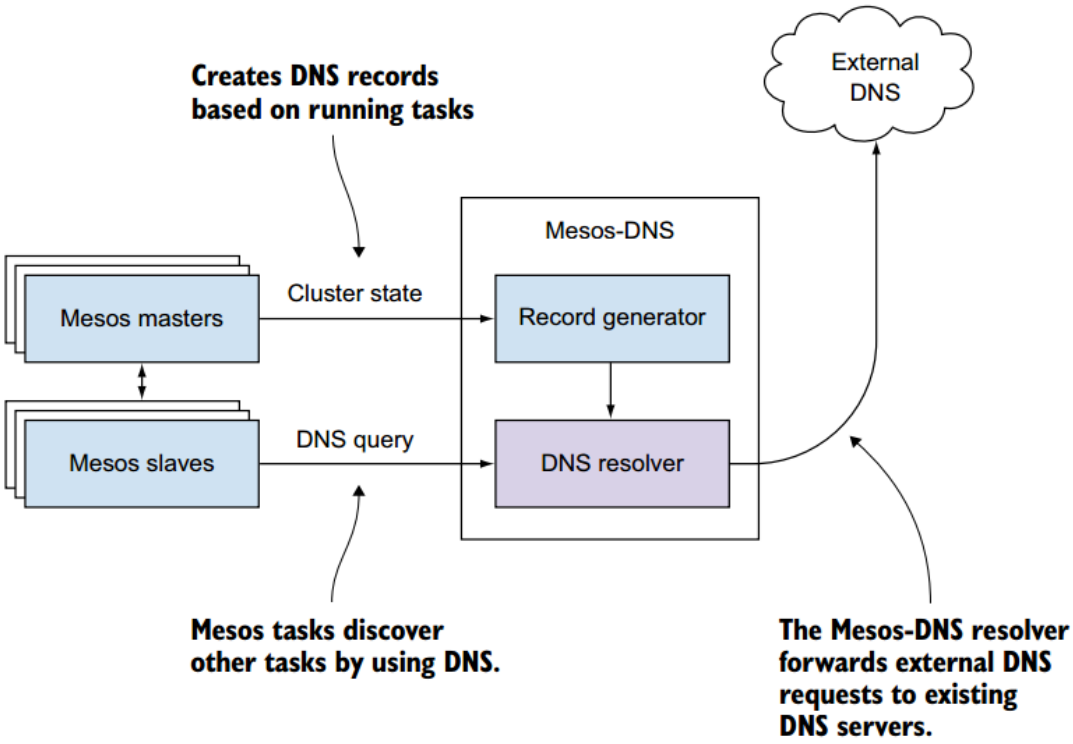

Figure 41 - Mesos Service Discovery

Figure 41 represents the case in which Mesos is hosted with A DNS resolution system. As it is possible to see, the master is responsible for generating a record for each service whereas slaves reach services by querying the Mesos DNS. Furthermore, there is the possibility to federate the Mesos DNS which can forward the request to an external existing DNS server.

Marathon-lb is port-based using a lightweight TCP/HTTP proxy, which is called HAProxy. It is an alternative way to implement a discovery service. The proxy is installed on each cluster host in order to forward requests statically addressed on a specific port. This allows for clients to connect to a specific port without no

knowledge about the underlying discovery process. However, this approach is useful only if all applications are launched by using Marathon.

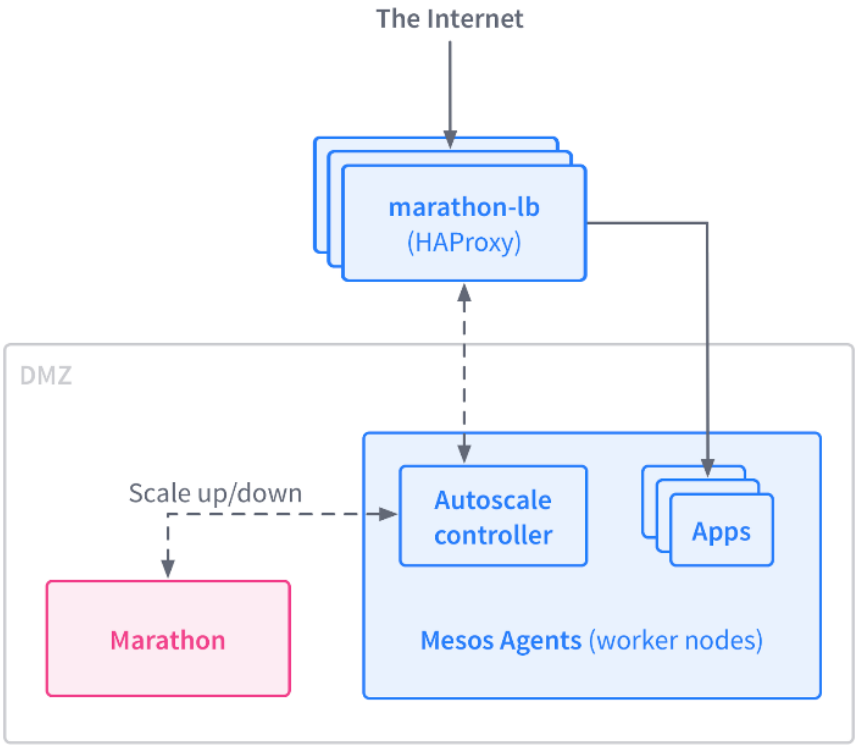

Figure 42 - Marathon Load Balancing

Figure 42 shows us the domain resolution system offered by Marathon atop of a Mesos cluster. In this way, Marathon sets up a proxy which is opened to the Internet. This is the component which stores the resolution name space in order to find where is placed is application component. Nevertheless, there are also other solutions such HaProxy-marathon-bridge, even if it is no longer supported. This is a script which configures a local installation of HAProxy, an advanced load-balancer, which is installed on each slave node. Furthermore, applications running on a slave should listen on a specific port on the localhost network interface. This allows HAProxy to guarantee the intra-cluster communication. In fact, it is periodically configured by a script that gets from Marathon APIs information about applications which are currently executing.

3.5.6 Chronos

Mesos is an elegant project which allows us to have a high-level representation of the underlying data center infrastructure. However, popular requirements have influenced the production of new frameworks that work on top of Mesos. Chronos is another framework example that works on Mesos to execute similar functionalities as the cron daemon, installed on the kernel of a UNIX-like operating system. The purpose is to get the possibility to schedule activities to be

executed in a repeatable way. This is typically realized with the configuration of a cron job, which executes a particular script accordingly to a temporal configuration. Furthermore, the Chronos project is an augmented cron daemon. Due to this introduction, there is the possibility to specify a task and the underlying middleware takes care to start execution on the infrastructure architecture. Furthermore, with classical cron scripts, we are not able to explain the dependent task. Therefore, Chronos allows us to support that in order to create interacting activities, such as a pipeline Extract-Transform-Load (ETL).

3.6 Rancher

Containers make software development easier by making code portable across development, test, and production environments. Furthermore, once in production, organizations focus on an orchestration tool in order to manage their containerized applications and service components. As we have already anticipated, a lot of solutions have been introduced, each of which faces specific aspects that, in part or completely, are not included in the complementary solutions. Even if, such projects are based on a modular architecture, customizing and setting up the orchestrators can be challenging and with the need to include a steep learning curve [44]. This is why Rancher was introduced for.

Rancher is not a container orchestrator but a complete container management platform that includes everything is needed to manage containers in production. So, users can quickly deploy and run multiple clusters across multiple clouds with a click of a bottom. This makes easier the management of all aspects of running containers. Furthermore, Rancher supports existing orchestration tools and so it works at a higher level than orchestration perspective. For this reason, users can use Rancher to set up different deployment environments, each of which can be launched in a matter of times.

This allows users not to face with orchestration-specific configurations, also providing the possibility to stay up-to-date with new stable releases easily. In fact, Rancher offers the possibility to set up an environment, using a specific orchestration tool, and as a result, there is no difference using that environment. So, clients can interact with the system as if it were built with the standalone configuration tools.

Also, Rancher makes the underlying orchestrations easy to be adopted, including enhancement features such as corporate security and multi-tenant environments. This is quite suitable for cloud scenarios where the multi-tenancy is a fundamental feature that characterizes each cloud infrastructure. Another important functionality that Rancher provides is the support of multi-clustering and multi-cloud deployments. This means that is possible to have running containers on a single on-premises as the same as containers which are running on multiple clusters and cloud service providers.

Rancher includes a distribution of all popular container orchestration and scheduling frameworks today, including Docker Swarm, Kubernetes, and Apache Mesos. This allows users the possibility to create multiple orchestration clusters by using the same underlying infrastructure. Also, Rancher supports its own container orchestration and scheduling framework, which is called Cattle. Rancher optimized this framework to orchestrate infrastructure services as well as setting up, managing, and upgrading existing orchestration clusters. So, the purpose of this section is to investigate this alternative solution and its own native orchestration tool.

3.6.1 Architecture

Rancher is designed to run Docker containers immediately on top of the kernel, namely those providing core Linux services to the users. Those services run inside containers, and so, users can create their own Docker (user-level) containers, as in any other Linux distribution [45]. Figure 43 shows us the layered architecture of Rancher platform.

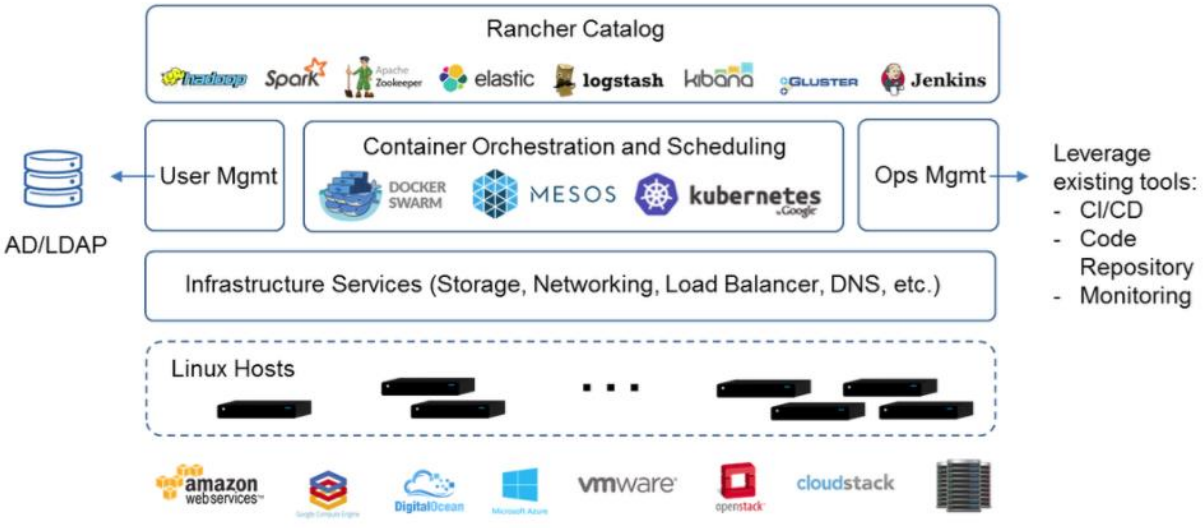

Figure 43 - Rancher Architecture

As it is possible to notice, the solution has no tight-coupled relationships. It is organized into three main levels: infrastructure services, container orchestration, and application catalog. Due to this modular organization, containerized applications are deployed on an infrastructure which is completely agnostic of the services that are built atop. This is guaranteed by the Rancher components that constitute the so-called “Infrastructure Services”. At this level, the platform takes care of low-level services that involve storage, processing or networking.

Furthermore, there is a standardization way that enables the architecture to be integrated with on-premises and cloud solutions. The unique requirement is to have machines whose kernel is Linux-compliant. Hosts can be even heterogeneous in terms of CPU, memory, storage, and network. To do that, this layer was defined including also other features, such as load balancer, DNS, and security. The flexibility is guaranteed because these components are installed in a container and so the infrastructure is completely agnostic.

As mentioned in the overview of this chapter, it is fundamental to have a container orchestration system in order to take advantages of external components that support managing and deploying containers along a cluster of machines. So, Rancher includes a layer of container orchestration in which is possible to continue to use native-clients in order to communicate with the correspondent container orchestration.

Actually, the supported solutions are Docker Swarm, Kubernetes, Mesos and the native Rancher orchestration system, Cattle. This allows the support of different cluster instances, each one with a specific orchestration system. Therefore, users create their own clusters, starting from the concept of “Rancher Environment”. Each environment is created from a template and, in order to build it, is necessary to specify the orchestration system which is preferred to use.

Lastly, the application catalog is a Rancher service which provides to users a set of pre-defined multi-container clustered applications. They can be installed with a simple click and can also be updated at runtime with new versions and configuration settings. Furthermore, Rancher includes an additional functionality that is called Enterprise-Grade control. This consists of a set of services that can be installed as a sort of plug-in like, for example, different authentication mechanisms, such as Active Directory, LDAP, and more.

3.6.2 Network

For the networking subsystem, Rancher supports a Common Network Interface (CNI) layer [46]. This allows the platform to enable the openness to other lower-level mechanisms, which can be integrated inside the Rancher platform. Basically, it consists of specifying which network services are installed on the underlying architecture. This is information is given by the Environment resource. However, it is necessary to select which driver type is needed to use the underlying infrastructure services. Therefore, for each network provider, the platform gives us a sort of plug-in catalog in which is possible to find the supported implementation.

By default, Rancher uses the driver which called IPsec. This defines an overlay network using IPsec tunneling. When a network driver is launched in the environment, a default network will be created. Therefore, each service installed on top will be using this network. Originally, Rancher used the managed network, using the docker bridge. With the adoption of the CNI layer, each container started on top of the network infrastructure sees just the Rancher managed IP. This information is not present in metadata of the underlying Docker engine and so, it is not possible to get them through a Docker inspection. Rancher included this new functionality extending the behavior of Docker ecosystem. Furthermore, each container is launched with just two network interfaces: loopback and the underlying managed by Rancher. Users interact with Rancher using the command line interface or the web interface. However, a container is always started using the managed network, the overlay network supported by Rancher. This allows to containers to communicate with each other, without no loss if they are running on the same host or not. Furthermore, this characteristic is important for other important services, are built on top of network infrastructures, such as load-balancer and DNS.

3.6.3 Storage

Rancher includes the possibility to use different storage services to expose the concept of “Volume”. The principle is the same used in other solutions like Docker. Therefore, this service type is also declared in the definition stage of an “Environment Template”. A template is another Rancher resource which is used to create an environment without no need to specify each functionality included. In fact, users are able to create or use an existing environment that can be easily made up by clicking the correspondent bottom. It is also possible to select and launch a storage service from the catalog. In fact, the application catalog provides

to the user the possibility to use existing storage solutions such as those that are particularly spread in the current segment target. However, there is no guarantee that will be a full compatibility with storage systems of some container orchestration, such as Kubernetes.

Volumes are able to have different scopes, which refer to the level at which the volume is managed by Rancher. Currently, using Rancher Compose files, there is the support to create different types of volumes. These are called scoped volumes and they must be defined in the correspondent section of a Docker compose the file. Actually, the scope definitions are stack and environment. By default, a stack scoped volume is created, but different scopes can be created on modifiers in the top-level definition. As the name suggests, the first one is confined to a single stack of services while the second one is completely visible to each component that is defined in the environment. For this reason, users need to evaluate the visibility trade-off in order to take advantages of both models. Furthermore, this makes of Rancher a featured platform that includes a feature not usual in other solution. In fact, the concept of environment is basically a synonym of the tenant and so we can conclude that the architecture is well-suitable for cloud infrastructure deployments.

3.6.4 Cattle

Cattle was the first container orchestration system available with Rancher and so it represents a solution quite stable inside the platform. It is much similar to Docker orchestration, considering the fact that is based on Docker commands. In fact, applications are defined using docker-compose. Furthermore, the application deployment is based on the concept of “Rancher Stack”. This is a set of components that together compose the application. It is very useful to adopt the microservice paradigm and so that resource has been also included with the supporting of the other orchestration tools. A stack can be directly launched by an application catalog, or through a docker-compose file with the augmenting of a possible rancher-c1ompose, basically, a Rancher extension of docker compose.

Each stack is composed of services. These are docker images, characterized by application requirements, such as scaling, health checks, service discovery links and configuration parameters. It is even possible to include a load balancer service and other external solutions within a cattle stack. This principle is outside other implementations such as Kubernetes and so on. All of that allows a quick deployment, simply based on single-click instead of defining docker-compose and rancher-compose.

However, as we have already explained, this is not the main feature of Rancher. In fact, the good integrating of other existing solutions and their functionalities has given to Rancher a good popularity that is influencing the rate adoption of that platform to run containers in production. Due to this modular architecture, users can easily take advantages of different solutions while maintaining a single management experience. Furthermore, it is cloud-agnostic and so it is possible to work across cloud and multiple data center without no loss of visibility and deployment reliability.

3.6.5 Cattle Scheduling

The scheduling subsystem is the core of Cattle. It handles port conflicts and ability to schedule through labels on host and containers. The concept of the label is the same that we have already introduced with Kubernetes. As in other solutions, Rancher does not force users to adopt a specific scheduling strategy [47]. In fact, it is possible to easily integrate other solutions, such as one chosen from the Rancher catalog. Furthermore, in Rancher the scheduling mechanism has been distinguished in three cases: multiple IPs on the host, resource constraints, and services that can be scheduled on a host. These are the main aspects that the platform considers when a scheduling decision is needed.

By default, Rancher assumes that one host has its own unique IP address. Moreover, if no address can be used, it is necessary to configure the system to notify the Rancher scheduler about which network addresses are being used. A common scenario is when a load balancer or a service needs a port to be externally exposed. In this case, Rancher will schedule against all the available scheduler IPs otherwise it will report the so-called “port conflict” [44]. The other aspect is about the configuration of a host inside the Rancher platform. In this case, the host is configured to the infrastructure with automatically assigned resource limits, which are based on the host characteristics.

Therefore, when users need to deploy an application, Cattle considers parameters, such as memory or CPU that can be used on the host. Nevertheless, most of the container scheduling is defined on the service. In fact, a service is defined with specific rules or host restrictions that the containers can be scheduled with. An example concerns about a container to be scheduled onto a host that has a specific host label. This is the last aspect which Cattle allows users to consider when containers should be run in production environments.

3.6.6 Rancher WebHook

WebHooks are components that can be created in Rancher in order to trap events that are useful to be handled to either provide a different behavior or to react when something happens. These are uniform resource locators (URLs) which can be used to start an execution action within Rancher. An example is a receiver hook [48] used to integrate a monitoring system to scale up or down the number of container instances for a specific service. It is composed of a name for the receiver, a type and the action associated with the receiver. After defined a WebHook, an URL is obtained.

Cattle introduces three types of receiver hook: “Scale a Service”, “Scale the number of Hosts” and “Upgrade a Service based on Docker Hub Tag Updates”. As the name suggests, we can scale a service. This requires to configure the WebHook to define the intention, the service involved and the maximum number of containers at the time. A possible usage of this receiver hook is to scale a service by implementing an auto-scaling integration with an outside process. This is another feature that distinguishes Rancher from the explained complementary solutions. However, each one has its own pros and cons and for this reason, at the end of this chapter, we will discuss a comparison of these solutions.

3.7 Amazon EC2 Container Service

Amazon EC2 Container Service (Amazon ECS) is a highly scalable and fast container orchestration service that makes easy to run, stop and manage Docker containers on a cluster of Amazon Elastic Compute Cloud (Amazon EC2) instances. By this way, it is possible to get the state of a cluster from a centralized service, accessing many familiar Amazon EC2 features [49]. It eliminates the need to operate on cluster management, configuration or worry about scaling the infrastructure. Basically, it represents the Docker-compatible orchestration solution from Amazon Web services. So, each amazon cluster consists of tasks which run in Docker containers, and container instances, among many other components. Furthermore, the solution manages just amazon container workloads, resulting in vendor lock-in. In fact, there is no support to run containers on infrastructure outside of EC2, including physical infrastructure or other clouds such as Google Cloud Platform and Microsoft Azure. Nevertheless, the solution is provided by Amazon as a service and so there is the ability to work with all the other AWS instances services like Elastic Load Balancers, CloudTrail, CloudWatch, and more.

3.7.1 AWS Elastic Beanstalk

Amazon Web Services is an elastic, secure, flexible and developer-centric ecosystem that serves as an ideal platform for Docker deployments. It provides the scalable infrastructure, APIs, and SDKs that integrate tightly into a development lifecycle and accentuate the benefits of the lightweight and portable container that Docker offers to its users. AWS Elastic Beanstalk [50] is a management solution for AWS services, such as Amazon Elastic Compute Cloud (Amazon EC2), Amazon Relational Database Service (Amazon RDS), and Elastic Load Balancing.

By this way, there is no requirement to manually launch the AWS resources to start the application. Therefore, it is AWS Elastic Beanstalk which handles the details of capacity provisioning, load balancing, scaling and health monitoring.

In addition, it provides the possibility to deploy and manage containerized applications, and a command line interface (web tool) that can be used to deploy both the AWS Elastic Beanstalk environment and Docker containers. Furthermore, there is the possibility to easily deploy and scale containerized web applications avoiding the complexities of provisioning the underlying infrastructure. In fact, we can affirm that: if more granular control over containers or custom application architectures is needed, it is better to consider working directly with Amazon ECS.

3.7.2 Amazon ECS

AWS Elastic Beanstalk is useful for deploying a limited number of containers and the way to run and operate container-enabled applications is quite flexible. Amazon Elastic Container Service (ECS) is designed to run and manage containers across a number of hosts that are grouped into clusters [50]. In fact, managing containers, as the number being run increases, becomes difficult and makes a not negligible overhead. This takes operators not to focus on core businesses. So, Amazon ECS provides a way to deal with containers and easily run distributed applications on a managed cluster of EC2 instances. Basically, it offers three possibilities to use Docker containers. Firstly, by simple API calls, with no need to install and operate with the cluster management infrastructure. Secondly, Amazon ECS is designed for use with other AWS services and includes access to many familiar features like Elastic Load Balancing, EBS volumes, EC2 security groups, and IAM roles. Lastly, there is the possibility to manage container scheduling through a variety of solution in order to support a wide set of applications.

3.7.3 Architecture

Clusters are made up of container instances, which are EC2 instances running the Amazon ECS container agent. This is responsible to communicates instances and container state information to the cluster manager and dockerd (the Docker daemon). Computation resources executing the Amazon ECS container agent will automatically register with the default or specified cluster. The Amazon ECS container agent is open source and freely available, and as such, can be built into any AMI intended for use with Amazon ECS [50]. Furthermore, when a cluster is created using the Amazon ECS console, an Auto Scaling group is also associated with the cluster. This ensures that the cluster grows in response to the needs of the container workload.

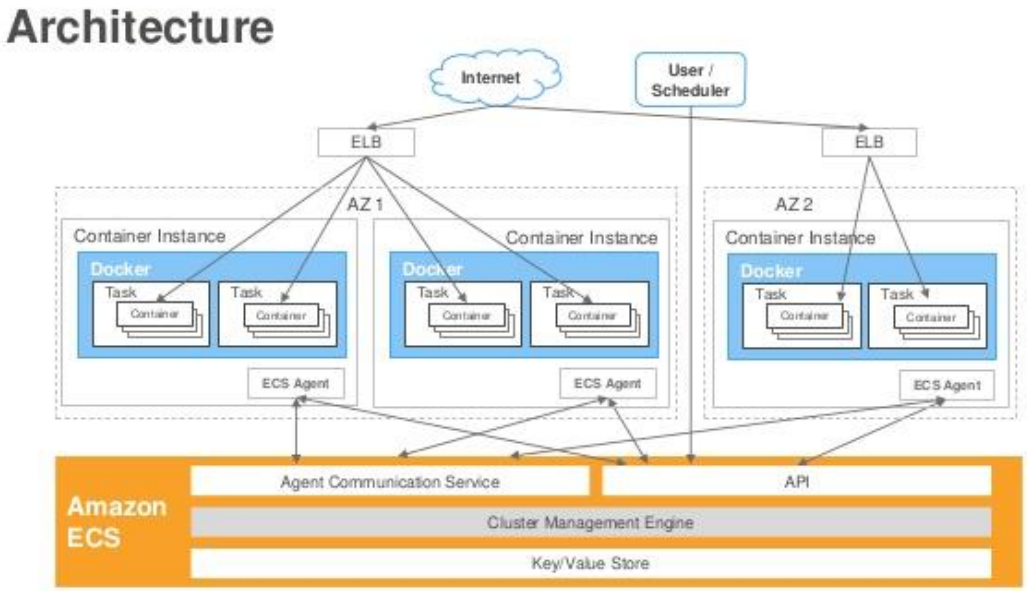

Figure 44 - Amazon ECS Architecture

Figure 44 illustrates the architecture of a typical amazon cluster with ECS as container orchestration engine. After instances have been deployed to a cluster, a task definition is used to define application or service to run. This is accomplished defining the containers and volumes that are deployed to a host.

Task definitions are one or more container definitions. These specify the name and location of Docker images, how to allocate resources to each container, and any links to other placement constraints. Furthermore, if it is needed, it is possible to specify any volume requirements. These are specified the definition of a single task, which is the minimum unit of work in Amazon ECS.

3.7.4 Scheduling

Amazon ECS is a shared state, optimistic concurrency system and provides very flexible scheduling capabilities. Schedulers use cluster state information provided by the Amazon ECS API actions [50]. Of course, this information is necessary to make appropriate placement decisions. Actually, Amazon ECS provides two scheduler options: RunTask and CreateService.

RunTask randomly distributes tasks across the cluster and tries to minimize the fact in which a single cluster instance will get a disproportionate number of tasks. The other one is ideally suited to long-running stateless services. This ensures that an appropriate number of tasks are constantly running and reschedules tasks when a fault occurs. In addition to default schedulers, Amazon ECS allows for integration with both custom schedulers and existing third-party schedulers, such as Apache Mesos Framework.

3.7.5 Network

Amazon ECS takes advantages of native Docker features like port mapping and container linking while building on host-level Amazon EC2 networking features [50] such as security groups and IP addresses. In addition, Amazon ECS supports Docker links, including the usage of injected variables and configuration host files. This is important to guarantee a simple discovery of other linked containers. Furthermore, for more advanced users, it is possible to specify the whole definition that a security group, network interface, and IP addresses should be used by a single container.

3.7.6 Storage

Data volumes are used to store and share information between containers. These are used as a persistent data store that can be shared between different containers on a host, as empty, non-persistent scratch space for containers, or as an exported volume from one container to be mounted by other containers. ECS task definitions allow us to reference the location of an appropriate location on the host (either on instance storage or using EBS volumes) [51]. Then, it is possible to reference the underlying volume from specific container definitions and let Docker managing the volumes within containers.

There are different options to use data volumes. An example is the so-called “sourcePath” that is a reference point to a directory on the underlying host. If a sourcePath is not provided, docker treats the defined data volumes as scratch space, and the data is not persisted past the life of the container. Furthermore, data

volumes can also define the storage relationship between two containers by using the “volumesFrom” parameter. This configuration allows getting data by referencing a data volume presented in a different container. This feature is quite useful when it is needed to export persistent state. However, the mount point for the exported volume is defined by the container that is exporting the shared volume.

3.8 Kontena

Kontena is an alternative container orchestrator. Compared to the current big players, such as Kubernetes and Mesos, it has a little different approach. For this reason, it is quite popular nowadays. An important difference is about a separate authentication server. The user context, necessary to interact with Kontena, is provided by an authentication provider, which can either be self-hosted or the one centrally hosted by Kontena. This would allow easy integration with an enterprise infrastructure, such as LDAP. Furthermore, Kontena separates authentication from authorization and each master does access control based on roles and users. In addition, there are several other features such as audit logs and supporting different existing solutions, such as Overlay network and OpenVPN.

The Kontena platform may be deployed on any infrastructure: private, public and hybrid cloud. This is influenced by the way in which the project is structured. In fact, the software is packaged as a container and so it works on any Linux machine capable of running privileged mode Docker container. The principle is the same used in solutions like Kubernetes: “all batteries included”. An example is about high-availability. It is designed to guarantee a stable architecture in which the rate of downtime is continuously reduced. Furthermore, it is based on a declarative service model through which the behavior of various applications is defined. An important difference, compared to competitor solutions, is the “Desired State Reconciliation”. In fact, in Kontena there is a strong monitoring of the grid state and when the desired state differs from the actual, a reconciliation process will be performed.

Nevertheless, Kontena is thought to include features already offered by a traditional orchestration system. For this reason, it is suitable to deploy stateful services with the possibility to have a complete management of those components. In fact, the underlying health check mechanism guarantees that a failure is automatically detected and faced with a new instance of the failed component.

To conclude this brief overview, we have learned that Kontena has all the basic ingredients to become a successful container orchestration platform for enterprises. However, compared to other solutions, it is not still well known and this might prevent that from reaching feature parity or implement better features in the orchestration space.

3.8.1 Architecture

The purpose is to run applications composed of multiple containers, such as elastic, distributed micro-services. To do that, the user starts by telling the Kontena system to run a service that is composed of one or more containers.

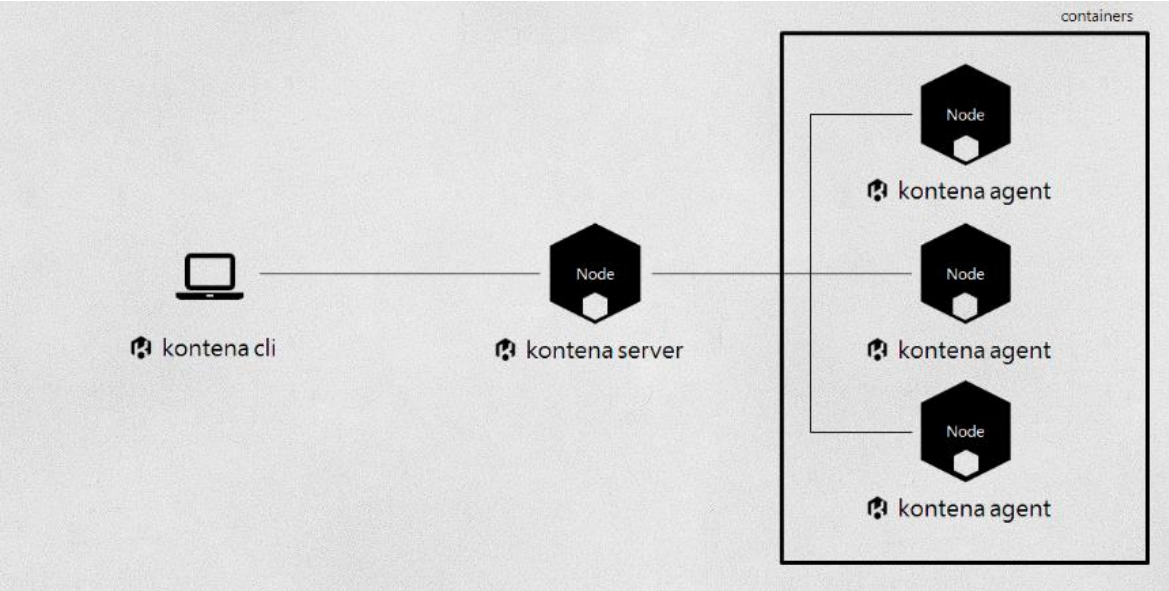

Figure 45 - Kontena Architecture

Figure 45 shows us the architecture of a Kontena deployment [51]. As in other solutions, it consists of a client/server model in which the server is organized according to a master/slave architectural pattern. In fact, there is a special node which works as manager of the whole cluster. This node provides an interface to manage Kontena object. The specific distinguishes these objects in Grids, Nodes, and Services. In addition, the master collects log streams and statistics from the Host Nodes and Services.

As anticipated in the overview of the orchestration tool, the user has to pass the authentication phase before interacting with the master API. Moreover, each master node might be used to manage multiple Grids, each of which assigned to a dedicated set of nodes to provide the computing power. Furthermore, as seen in Kubernetes, the master node by itself does not provide any computing power for any of the services.

On the contrary, the slave nodes are known in Kontena as “Host Nodes”. They are designed to deliver the computing power to the Grid. The architecture is completely infrastructure agnostic and so these nodes can be hosted by virtual or physical machines whose kernel is Linux-compliant.

A Grid receives a number of host nodes. The platform allows us to increase the available capacity by scaling up the number of host nodes to a Grid. Furthermore, the communication between the master and the host node is performed via a secure WebSocket [52] channel. This is an important difference, compared to other implementation, because this medium is used for all services and so: service orchestration, management, statistics and log streams.

3.8.2 Network

The network model is based on the Kontena grid, which spans a set of host nodes. In fact, a Grid object uses an overlay network to provide connectivity between service containers, even running on different nodes. Furthermore, the Kontena Agent establishes the overlay network mesh between the nodes and the grid network [51] provides service discovery for each deployed microservice component.

As anticipated before, each host node runs the Kontena agent, which establishes a WebSocket connection to a Kontena master. While the master can manage multiple grids, each grid has an isolated overlay network with its own address space. This means that nodes and containers, attached to different Grids, cannot communicate with each other. Of course, this is an important feature that, as seen with solutions like Rancher, goes towards the direction of multi-tenancy in an enterprise deployment.

3.8.3 Storage

Kontena provides an experimental support for managing persistent service data by using the same concept of “Docker volume”. In the Kontena master, volumes are supported as first-class objects and can be referred to from stack YAML definitions [53]. Furthermore, a volume can be used by multiple service instances, that can be deployed to different host nodes. However, in this case, the scheduler will automatically create multiple separate volume instances for each Kontena volume. In fact, these correspond to a specific Docker volume on a specific host node.

Furthermore, volumes are defined with a scope that can be distinguished in an instance, stack or grid. The suitable scope depends highly on the service that relies on data and to provide the desired durability. Instance scoped volumes are created per service instance and so each service instance will get its own volume. On the contrary, stack scoped volumes are created once per stack per node. This means that services, within the same stack and running on the same node, will use the same Docker volume. Lastly, grid scoped is used once per grid per node and the principle is the same to the others complementary scopes.

3.8.4 Scheduling

Kontena has a built-in advanced scheduler [54] that takes care of running and managing service instances on multiple host nodes. Furthermore, it is guaranteed an automatic failover and rebalance when the cluster has changes that will affect services. An important difference between Kontena and other solutions is that the scheduler is aware of the service nature. In fact, it distinguishes stateless and stateful services, not migrating stateful services to another node.

The scheduling can be described with deployment strategies and functionality conditions. Deployment strategies allow users to adopt different scheduling algorithms. At the moment the supported strategies are High Availability (HA), Daemon, Random. A service with HA strategy will deploy its instances to different host nodes. On the contrary, the daemon strategy will deploy a given number of instances to all nodes and the last one, Random, will deploy service containers to host nodes randomly.

There is also the possibility for users to provide several conditions and rules to drive the scheduler in order to determine how and where to deploy service instances. The currently supported definitions are: “Wait for port”, “Min health”, and “Affinity”. When a service has multiple instances and the so-called “wait_for_port” definition, the scheduler waits until the container responds to a specific port, before starting to deploy another instance. This is performed in order to achieve zero-downtime deploys. “Min health” is useful to Kontena that will make sure that at any point in time a number of healthy instances are up. Lastly, an affinity condition is when Kontena is trying to find a field that matches given value. Furthermore, Kontena has the ability to compare values against node name, labels and service name.

3.8.5 Kontena Objects

Kontena provides the complete environment for orchestrating and running containerized workloads. In fact, the platform abstracts all available compute resources and data volumes as a single unified resource pool. Furthermore, these resources are used by containerized workloads describes through the high-level Kontena objects. These are classified as described in the following list.

- **Grid** - the top-level abstraction that consists of a set of nodes. It is created and managed by Master Node. Furthermore, the creation of a Grid [51] implies the automatic creation of an overlay network with VPN access available. Moreover, each node is automatically connected to this overlay network. Therefore, service may communicate with each other in multi-host environments just like in a local area network.
- **Service** – a logical set of containers. In fact, containers are ephemeral environments that come and go. Furthermore, they get their own IP addresses and those cannot be predicted in advance. Therefore, a service [51] defines a logical group of correlated containers by building also a logical central point to configure and specify the desired runtime state. As in other orchestration solutions, Kontena provides the support for both stateful and stateless services but in addition, it offers the possibility to work also with batch and data streaming processing.
- **Stacks** – the same concept of Rancher Stack an used to distribute, deploy and run the pre-packaged application. Furthermore, these are reusable collections of multiple services with any associated configuration.

3.9 Nomad

Nomad is a cluster manager and scheduler solution that can be included in different related categories. Until now, we have learned several orchestration tools whose aim is to provide all the features needed to run application containers, including additional functionalities such as cluster management, scheduling, service discovery, and more.

Nomad only aims to provide cluster management and scheduling. It is based on the Unix philosophy of having a small scope while composing with other solutions like Consul for service discovery and Vault for secret management. Furthermore, Nomad is more general purpose and so it supports virtualized, containerized and standalone applications, including Docker.

Compared to the solutions which we have just discussed, Nomad is architecturally much simpler. It is based on a client/server model and does not require any external services for coordination or storage. This is quite different because, as we have already seen, solutions like Kubernetes are designed as a collection of more than a half-dozen interoperating services which together provide the full functionality.

On the contrary, Nomad combines a lightweight resource manager with a sophisticated scheduler into a single system. Furthermore, it even supports working with huge clusters and multi-datacenter deployments. For this reason, it is a good chance to investigate it as alternative solutions to those we have discussed before.

3.9.1 Architecture

Nomad is a free and open-source solution which comes from HashiCorp Software Company. This platform is based on a client-server model through which users deploy applications in order to take advantages of a well-structured cluster management.

Before describing the architecture, it is needed to list a set of terms that are used by a single Nomad deployment.

- **Job** – a specification provided by users that declare a Nomad workload. This is a form of the desired state and so the responsibility of Nomad is to make sure that the cluster state matches the user desired state.
- **Task Group** – a set of tasks that must be run together. A single job is composed of one or more task groups. This is the unit of scheduling and so the entire group must run on the same node.
- **Driver** – the basic means of executing a single task. An example of the driver is Docker, Qemu, Java and static binaries.
- **Task –** the smallest unit of work in Nomad. They are executed by drivers which allow Nomad to be flexible in the types of tasks it supports.

Nomad infrastructure [55] is classified in regions and data centers. A region can contain multiple datacenters. The architectural pattern adopted by the platform is a simple master/slave. However, these are respectively known as server and clients. Servers are assigned to a specific region and for each region, they manage the whole state making scheduling decisions. Furthermore, there is the possibility to federate multiple regions together.

As seen with other solutions, the master is the brain of the cluster. Each region containers a cluster of master and data are replicated in order to ensure high-availability. Therefore, Nomad makes use of an election algorithm to make sure that at every moment just one node acts as a master. On the contrary, the slave, but properly called client, is a machine that tasks can be executed. To do that is necessary to run the Nomad agent. This agent is a long-lived process and it is responsible to interact with the servers and executing application tasks.

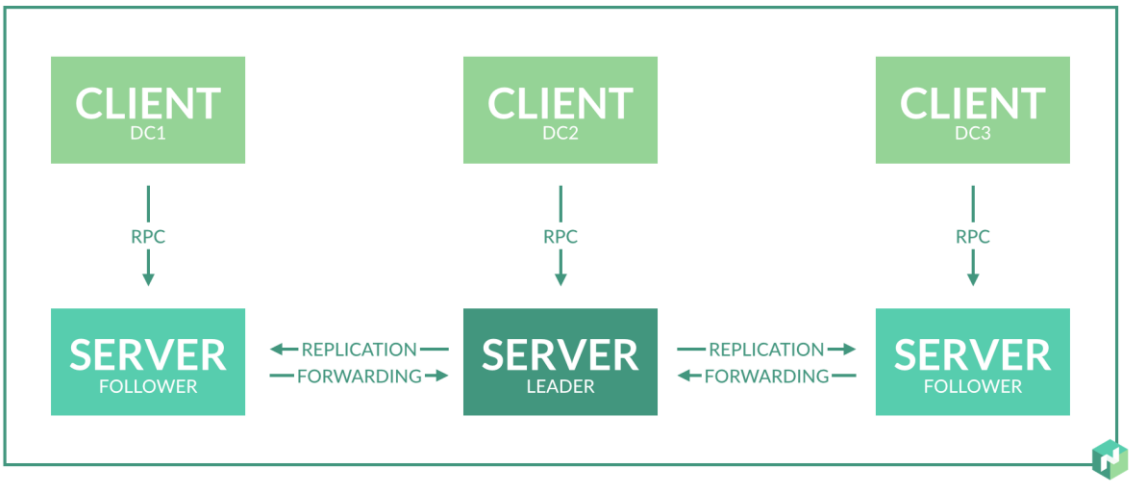

Figure 46 - Nomad Architecture

Figure 46 illustrates the architecture of Nomad [56], based on a single region. As it possible to see, each region holds both clients and servers. Servers are responsible for accepting jobs from users, managing clients, and computing task placements. Each region is fully independent of each other and does not share jobs, clients or state. Therefore, they are loosely-coupled using a gossip protocol, which allows users to submit jobs to any region or query the state of any region transparently.

The servers in each region are all part of a single consensus group. For this reason, they work together to elect a single leader which has extra responsibilities. Furthermore, Nomad is optimistically concurrent, all servers participate in making scheduling decisions in parallel. Of course, the leader provides the additional coordination needed to do this safely and to ensure clients are not oversubscribed.

Clients communicate with their regional servers using remote procedure calls (RPC). This interaction includes registering, sending heartbeats for liveness, waiting for new allocations and updating the status of allocations. Obviously, this

information is taken by servers which need to deal with that in order to perform scheduling decisions and create allocations to assign work to clients.

Even if it is not shown in the picture, Nomad provides a command line interface client by using APIs to allow users to submit jobs to the servers. Furthermore, Resource utilization is maximized by the so-called “bin packing”, in which the scheduling tries to make use of all resources of a machine without exhausting any dimension. This is the most important feature of Nomad that, compared to other solutions, gives to that a good value also used in orchestration target.

3.9.2 Scheduling

The scheduling process must respect the constraints as declared in the job and optimize resource utilization. The design is heavily inspired by the work of Google on both Omega and Borg. Therefore, it takes care to be flexible and scalable but also to deal with a large-scale cluster in order to offer a well-designed service management.

As anticipated before, the high-level resources of Nomad are jobs, nodes, allocations, and evaluations. Tasks can be scheduled on nodes in the cluster running the Nomad client. The mapping is done by using allocations. So, an allocation is used to declare that a set of tasks in a job should be run on a particular node while scheduling is the process of determining the appropriate allocations and is done as part of an evaluation. Furthermore, an evaluation begins with an event causing the process to be created. These are created in the pending state and are queued into the evaluation broker.

The evaluation broker is unique in the cluster and runs on the leader server. Its responsibility is to manage the queue of pending evaluations, provide priority ordering and ensure at least once delivery. Each server runs scheduling workers, one per CPU core, in order to process evaluations. From the broker, the workers pull from the queue evaluations and then invoke the appropriate scheduler as specified by the job.

Nevertheless, Nomad schedulers are classified in service, batch, system, and core. The first is used for long-lived services; batch is used for fast placement of batch jobs; a system to run jobs on every node and core is used for internal maintenance. Furthermore, Nomad can be extended to support custom schedulers as well.

The output result of a scheduler process is an allocation plan [55]. This is the set of allocations to evict, update or create. Placing allocation is split into two distinct

phases, feasibility checking, and ranking. Once the scheduler has ranked enough nodes, the highest-ranking node is selected and added to the allocation plan. When planning is complete, the scheduler submits the plan to the leader which adds the plan to the planning queue. This allows the leader node to protect against from resource over-subscription and for this reason, it performs partial or complete rejections of a plan.

Once the scheduler has finished processing an evaluation, it updates the status of the evaluation and acknowledges delivery with the evaluation broker. This completes the lifecycle of an evaluation and the created allocations are picked up by client nodes which starts the execution.

3.9.3 Use cases

Nomad is well-designed to act as Microservices Platform, Hybrid Cloud Deployments, and E-Commerce service application. Microservices [40], or Service Oriented Architectures (SOA), are design paradigm in which an application is structured as a collection of loosely coupled services. These should be fine-grained and the protocols, used to interact, should be lightweight.

Of course, this improves modularity and makes the application easier to be understood, developed and ready to production. However, they add an operational challenge of managing hundreds or thousands of services instead of a few large applications.

Nomad provides a platform for managing microservice components, making it easier to adopt the paradigm. In fact, Nomad is designed to handle multi-datacenter and multi-region deployments, being cloud agnostic, and it can be seen as an enabler to hybrid cloud deployments.

This is useful if servers are set to be executed in private datacenters running bare metal, OpenStack, or alongside AWS, Azure o Google Cloud. Therefore, it is easier to incrementally migrate workloads or to utilize the cloud for bursting. Furthermore, the last case is about a typical E-Commerce web application. Nomad allows all typical workloads to share an underlying cluster, increasing utilization, reducing cost, simplifying scaling and providing a clean abstraction for developers.

3.10 Closing remarks

As expected in the overview, the orchestration level is an important containerization feature that is fundamental to deal with continuously automated

scheduling, coordination, and management of complex systems of containerized components and the resources they consume. On top of the underlying infrastructure, we have machines (physical or virtual) whose operating system has to support the execution of a container runtime. Furthermore, this is not enough because of the automated arrangement, coordination, and management of complex systems requires the introduction of a middleware layer able to support these features. That is why the orchestration layer was introduced for.

Moreover, as explained in the correspondent section, an orchestration system is characterized by three important functionalities: service management, scheduling, and resource management. Surely, these are functional capabilities that are used to design and quickly implement a containerized system whose components are spread over the cluster. Furthermore, there are also other non-functional qualities which are often required for the following: scalability, usability, availability, portability flexibility and security. Choosing the right containerization and the cloud computing cluster management tool can be challenging. Each tool has a different function even if they can be broken down in application container schedulers and infrastructure management platforms.

While the container runtime format is largely settled, the real differentiation is in how to deploy and manage those containers. Therefore, several solutions have been implemented in order to provide and meet some specific aspects. In fact, in each solution, the included approaches and features vary enough that comparing them is necessary to choose the right option for the specific use case. For this reason, we conclude this chapter describing how these solutions are different in the context of resource management, scheduling and service management.

Resource Management

Resource Management is the functional set of capabilities that are logically below the scheduling module. Therefore, scheduling cannot avoid considering resource management in order to efficiently place components at the orchestration level. As explained in this chapter, an important point of view is to provide a service which is infrastructure-agnostic. For this reason, managing objects, such as Memory, CPU, IPs etc., can be indispensable. Furthermore, this is not just a functional requirement but also a helper for non-functional qualities, such as usability, availability, and flexibility.

Nevertheless, analyzing the different solutions, we can make a set of resources that need to be considered. Surely, this includes Memory, CPU, GPU, Disk Space, Volumes, Persistent Volumes, Ports, and IPs.

✓ *yes* O *part.*	*Kubernetes*	*Mesos/ Marathon*	*ECS*	*Swarm*	*Nomad*	*Cattle*	*Kontena*
Memory	✓	✓	✓	✓	✓	✓	✓
CPU	✓	✓	✓	✓	✓	✓	✓
GPU		O					
Disk Space		✓					
Volumes	✓	✓	✓	✓			✓
Persist. Volumes	O	O	O				
Ports	✓	✓	✓	✓	✓	✓	✓
IPs	O	O		O			O

Table 5 - Container Orchestration Engine: Resource Management Comparison

Table 5 shows us the support comparison for resource management. In this case, Mesos is better. This is quite acceptable because it was designed for abstracting the whole infrastructure of the data center.

Scheduling

When applications are scaled out across multiple host systems, the ability to manage each node and abstract away the complexity of the underlying platform becomes attractive. The orchestration is a broad term that refers to container scheduling in order to get the ability for an administrator to load a service file onto a host system that establishes how to run a specific container. Furthermore, a cluster scheduler has multiple goals: using efficiently the cluster resources, working with user-supplied placement constraints, scheduling applications rapidly not to let them in a pending state, being robust to errors, and guaranteeing high-availability.

Actually, there are three main scheduler architectures that are adopted by the solutions which are spread in the scheduling market: monolithic, two-levels, and

shared state. Monolithic consists of a solution in which the scheduling decision is performed with no concurrency. This is obviously the simplest even if, often, it does not guarantee the best performance. A two-level scheduler, as seen in Mesos, adjusts the allocation of resources to each scheduler dynamically using a central coordinator to decide how many resources each sub-cluster can have. Lastly, the shared-state consists of a scheduling module in which there is no central resource allocator. Table 6 shows us a functional evaluation of the scheduling process in the solutions that have been discussed in this chapter. We can summarize the important features of a scheduling system, considering the following capabilities:

- **Placement** - the main capabilities that allow to users to load a service file and automatically seeing the execution of scheduling decisions by the architecture.
- **Replication/Scaling -** needed to make more than one instance, in order to provide high-availability and reduce latency.
- **Readiness Checking -** it allows to include the service only when the component is ready to answer. This consists of a simple request/response protocol. In fact, the purpose is to have not just alive components but also ready service pieces of the application.
- **Resurrection -** the capability that is useful for long-lived processes whose job requires to be always up.
- **Rescheduling** - it consists of being tolerant when a node fails.
- **Rolling Deployment**–it is important when upgrades/downgrades are performed and the application will show no downtime
- **Collocation** – it consists of deploying more than one container on the same host. This is fundamental to take advantages of local inter-process communication instead of deploying components on different hosts.

	Kubernetes	*Mesos/ Marathon*	*ECS*	*Swarm*	*Nomad*	*Cattle*	*Kontena*
Placement	✓	✓	✓	✓	✓	✓	✓
Replication/ Scaling	✓	✓	✓	✓	✓		✓
Readiness checking	✓	✓	✓		✓	✓	✓
Resurrection	✓			✓	✓	✓	✓
Rescheduling	✓	✓	✓	✓			✓
Rolling Deployment	✓	✓			✓	✓	
Collocation	✓						

Table 6 - Container Orchestration Engine: Scheduling Comparison

As it is possible to notice, Kubernetes is the most featured project It is considered a pure application container scheduler and for this reason, it has recently moved on Rancher with a more comprehensive infrastructure management platform.

Service Management

This is the highest capability level of each orchestration system. It provides the functionalities to manage high-level services such as load-balancing, multi-tenancy, high-availability and more. This is led by the spectrum of cloud computing servers. The key difference is about the container deployment and lifecycle management. This allows us to quickly deploy containers in production through a management layer which suits the rapid change applications undergo in a DevOps strategy. However, as happened for scheduling and resource management, any type of tool automates the management of the underlying infrastructure according to a specific business strategy.

✓ *yes* O *ext/part*	*Kubenetes*	*Mesos/ Marathon*	*ECS*	*Swarm*	*Nomad*	*Cattle*	*Kontena*
Labels	✓	✓	✓	✓	✓	✓	✓
Groups / Namespaces	✓	✓					✓
Dependencies		✓					
Load Balancing	✓	O	✓		✓	✓	O
Readiness checking	✓	✓					

Table 7 - Container Orchestration Engine: Service Management Comparison

Table 7 shows us the comparison, from the service management point of view, between the analyzed orchestration tools. Mesos, in conjunction with Marathon, is almost completely featured. However, the focus is not based on the high-level of container applications and so complementary solutions such as Kubernetes are designed to include also components like load balancing and more. To conclude, we understood that there are three key differentiators: the level of abstraction, the specific container-centric and the integration with external services. However, there is no solution which is a fit-for-all purpose because, as anticipated before, an orchestration tool can be distinguished in container and infrastructure focused. Therefore, users must pay attention at the functional comparison, which we have just presented, and choose the solution that is more suitable for the specific business use case.

4 Container-focused Operating System

4.1 Overview

There has been a lot of recent excitement around containers and orchestration tools. In particular, the containerization focus has been moved to another point of view: container-focused operating systems [39]. This is a new reality and also important as much as containers and orchestration systems since the paradigm analyzed until now, did not include the set of requirements that a containerization technology nowadays can deal with.

For example, a production environment can require to directly install a specific component on systems or also mobile devices that do not need the usage of starting and stop containers. Applications are often built on distributed systems comprised of a lot of individual services, engines, and data-processing tools. This is the case for also cloud-based applications inside a large enterprise in which the consumer needs to point out a much larger software system.

These applications require handling the actual demands around performance, features, reliability and continuous improvement. So, it is better than all these pieces work together in order to augment the experience of users, operators, and developers. The introduction of a distributed operating system is able to face with these issues in order to allow users to easily deploy a single platform for running everything that modern applications require. This includes Docker containers but also every modern infrastructure that makes extensive use of open source projects. So, containerization contributed to enhancing this model to take existing advantages to provide other types of services.

4.2 The need of a Container Operating System

Since the launch of Docker, there has been an explosion of new container-centric operating systems [57]. The success of these new container-centric operating systems apart is given by their lightness compared with a traditional Linux distribution. Containers are run somewhere and the host, which container will be executed on, needs to have an operating system as well. This has led the rise of these new container-centric operating systems, also identified as “micro OSes”.

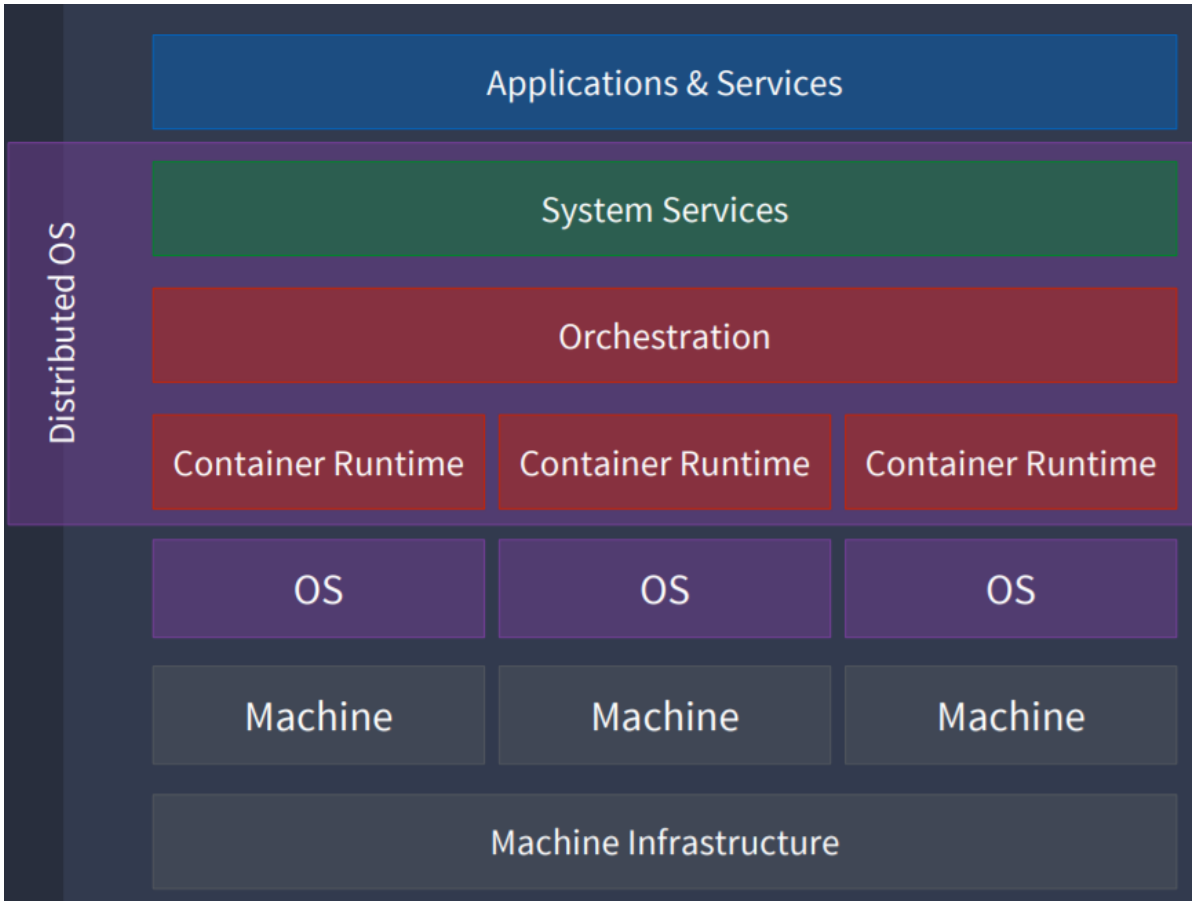

Figure 47 - Architectural model of a Distributed Operating System

Figure 47 shows us the layered structure of a Distributed operating system. This allows us to build a set of the cluster whose machines are primarily fixed to be managed by a container platform.

However, the idea is not new. In fact, stripped-down operating systems have long been embedded in electronic systems, ranging from traffic lights to digital video recordings. Initially, these operating systems were designed to run on a single node. Today, micro OSes [58] are designed to run in distributed environments, in which the entire data center is treated as one giant operating system that spans hundreds or even thousands of nodes. Furthermore, this can be treated with the containerization paradigm, because the existing solutions allow us to quickly deploy and run microservice components without no need to face the underlying configuration problems.

4.3 CoreOS

Nowadays, a plenty of distributions support Docker, but not in a way that seems designed for large-scale production use. CoreOS [59] is an operating system designed from the ground up to facilitate container operationalization at any scale.

In fact, it is extremely lightweight and designed to guarantee high-availability and fault tolerance.

It is the pioneer of the micro OS paradigm [60]. The idea is to build an environment where changes do not involve any propagation. This is a consequence of the traditional case in which is difficult to change things in a Linux server environment. CoreOS is a platform in which the operating system is treated more like a web browser, that is automatically updated as new components are released. Furthermore, CoreOS aims to be the general-purpose choice, and so the company supports numerous deployment options.

The purpose is to provide a container-based Platform as a Service, and so it takes care of infrastructure and architecture problems. It is a Linux distribution based, in a way, on Gentoo Linux. Furthermore, it is an important part of many container stacks and runs on almost any platform, including Vagrant, Amazon EC2, QUEMU/KVM, VMware, OpenStack and bare-metal hardware.

4.3.1 Architecture

CoreOS is not just a container management system. In fact, it is an entire Linux-based operating system. It consists of a few critical systems and services that manage all the scalability and fault tolerance it claims to facilitate.

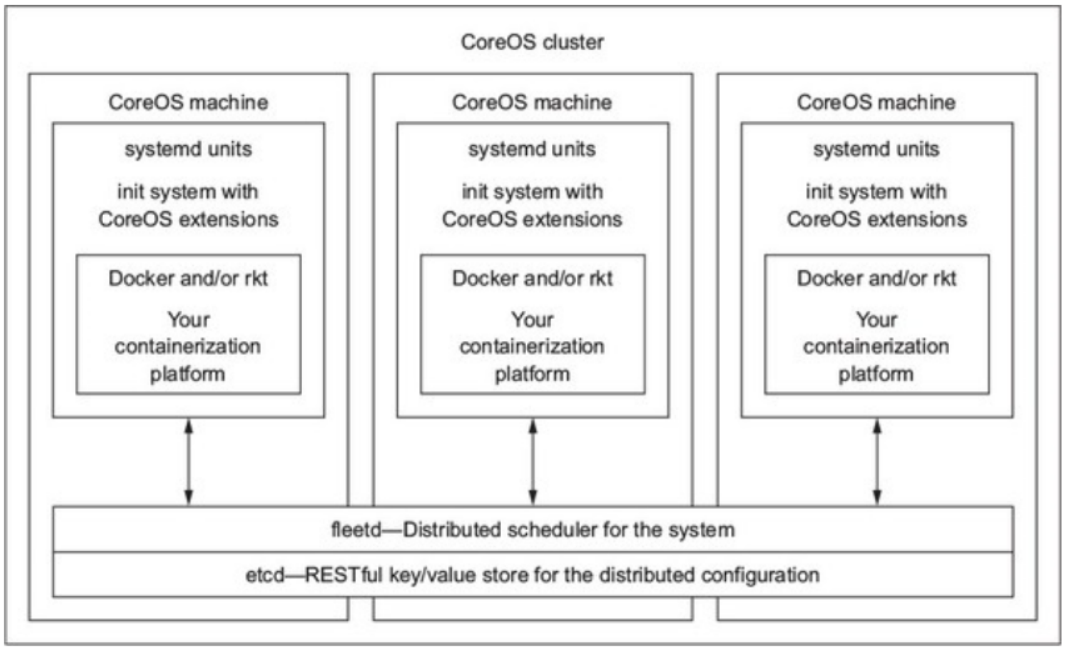

Figure 48 - CoreOS Architecture

Figure 48 shows the architectural point of view of a typical CoreOS cluster.

The basic components [59] of CoreOS are etcd, fleet, system processes and a declarative specification, which is called cloud-config. Etcd is the key/value store that is useful to the API Server in order to provide the RESTful state for the distributed configuration. Fleet is the agent that is responsible to act as a distributed scheduler for the system. The others are used to set up the entire cluster in order to guarantee consistency and synchronization between all nodes of the cluster.

In the example, there are tree CoreOS machines, each of which, with a container engine. Actually, the architecture supports the integration with Docker, Rkt but also future implementations. This is true due to the standardization of the underlying container runtime infrastructure.

4.3.2 Configuration and Service Discovery

Etcd is a highly-reliability distributed key/value [61] store. It focuses on distributed consistency and availability over performance. In order to access to it, the project provides a command line interface which interacts with the core of the project. It was designed to distribute system and service configurations. As will be seen soon, etcd is the data store for the CoreOS distributed scheduler, fleet. Therefore, due to etcd, nodes exchange configuration with other, and they are aware of what services are available across the cluster. Furthermore, data can be read from etcd via a command line utility or via an HTTP endpoint.

4.3.3 Application Management and Scheduling

Fleet is the other important component of CoreOS. It helps the platform to act as a single machine by distributing system units intelligently across the cluster. To do that, it makes use of etcd in order to distribute the whole state. Fleet gets requests to start up a number of service units, and it will perform actions across the cluster. Furthermore, the fleet is considered as a cluster-wide init (the first process that runs all other processes) system that interacts with the underlying system processes of each individual node. This allows the project to manage individual processes on each node from a single central point. However, the fleet is no longer actively developed or maintained by CoreOS. In fact, it recommends Kubernetes for cluster orchestration.

4.3.4 Container runtime

The last part of the layout is the container runtime. CoreOS supports Docker and Rkt. However, as it has already been seen in the correspondent chapter, Rkt is

able to run Docker containers. This is true due to the App Container (appc) specification through which container images (ACIs) were defined.

4.3.5 Applications

There is no package manager in CoreOS; all applications run inside containers. To do that, the platform makes use of Docker or the native container engine, Rkt. The goal of CoreOS is security, consistency, and reliability. Therefore, updates are automatically done using an active/passive dual-partition scheme [62] to update CoreOS as a single unit, instead of using a package-by-package method. So, it uses Linux containers to manage services at a higher level of abstraction. Furthermore, there is no installing package via yum or apt. In fact, the code of a single and all its own dependencies are packaged within a container that can be run on a single CoreOS machine or on the cluster. Considering the fact that CoreOS is only designed to run application containers, many fewer system-level packages are required and installed. This means lower CPU and more efficient memory usage if compared to a typical Linux server.

4.4 RedHat Project Atomic

Red Hat Enterprise Linux Atomic Host is a variation of the Red Hat Enterprise Linux 7 because it is optimized to run Linux containers. Basically, it is lighter than a traditional operating system even if it is not as small as some of its competitors. The idea was to counteract the other solutions by the fact that most of them integrated several system containers in addition to the application container. Therefore, the purpose was to integrate just the set of services that address the most use cases for container applications. These are put in a sort of middleware, which is called Atomic. So, this is a platform to deploy and manage containers on bare-metal, virtual, or cloud-based servers.

In fact, there is no requirement to have specific server hosts, considering the fact that the operating system comes with built-in functions with Docker and the related system components. Roughly speaking, it is designed to be minimally focused on the delivery of container services.

The project contains Docker, Flannel, and Kubernetes to build clusters for container-based services. Docker provides the container runtime, Flannel the overlay networking while Kubernetes the scheduling and the coordination of host containers. In addition, it makes use of properly security implementations in order to secure the deployed containers as well as manage accesses to and from them.

4.4.1 Architecture

Red Hat Atomic Enterprise Platform [63] is an optimized container infrastructure platform for deploying, running, and managing multi-container based applications at scale. It is used to provide a scale-out cluster of instances that together form an enterprise-class foundation for delivering traditional and cloud-native enterprise applications.

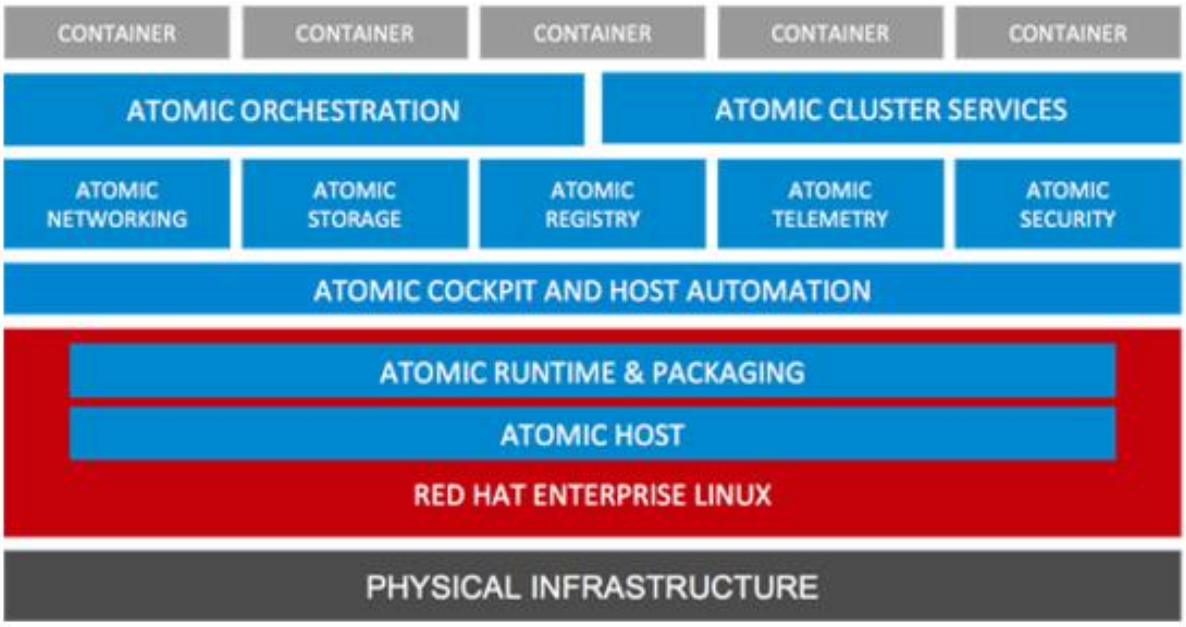

Figure 49 – RedHat Project Atomic Architecture

Figure 49 shows us the architectural point of view of a typical Atomic cluster. The primary building block is the Atomic Host, a lightweight container operating system which implements the target idea.

This is immutable since it is imaged from an upstream repository, supporting mass deployment and applications that are executed by containers. Currently, the host comes out the box with the orchestration system Kubernetes. However, it is being analyzed to support different versions of the same host, such as the third version of OpenShift. Furthermore, the project makes use of several Kubernetes utilities like etcd and flannel.

The host system is managed via rpm-ostree, an open source tool for managing bootable, immutable, versioned filesystem trees which come from upstream RPM content [58]. This and several other components are wrapped in the atomic command which provides a unified entry point. In addition, many other tools are included in the container-based infrastructure, such as the following: “Cockpit”, to give visibility to the hosts and container cluster; “Extensions to Docker”, for better security and monitoring system integration; “AtomicApp” and “Nulecule”

for composing multi-container applications; “Commissaire”, to provide a better API for Kubernetes hosts, and “Atomic Developer Bundle” to make easier the development of containerized applications.

4.4.2 Network

Atomic makes use of Docker and Kubernetes and so it takes advantages of both solutions. As seen in the correspondent chapter, Docker hosts, by default, give to each container a network address that is taken from an unused private address range. This enables containers on the same host to communicate with each other, by using assigned IP address and their exposed ports. However, with the single-host-networking model, container linking does not span multiple docker hosts, and it is difficult for applications running inside containers to advertise their external IP and port, considering the fact that these are not available to them.

Multi-host docker deployments [64] will benefit from using the additional stack components that ship with Atomic. In fact, an Atomic cluster comes out with the possibility to configure flannel and Kubernetes. We have already discussed the multi-host-networking support of Kubernetes and that addresses this issue by giving to each application component a sort of network visibility that is independent of the underlying host operating system.

So, each cluster machine receives a full subnet, in order to reduce the complexity of doing port mapping. Therefore, Atomic hosts include the networking driver which provides an overlay network by defining an independent local subnet in a Kubernetes cluster. By this way, Atomic combines the advantages of Kubernetes and Docker in order to provide a cluster platform that is able to run and manage production services.

4.4.3 Storage

RedHat Atomic Enterprise Platform makes use of the concept of “Volume” to provide a persistent-data-storage subsystem. This is set by default partitioning through the docker-storage-setup service, which creates a Logical-Volume-Management (LVM) thin pool to be used by the container images. Firstly, a root logical volume is created and then that service sets up a LVM pool [65], which is called docker-pool. This takes 60% of the whole space and the remaining can be used for extending the root volume or the docker-pool. Furthermore, it is also possible to override the behavior of the service during the boot process.

When Atomic Host is installed from a cloud image, by default, it makes use of a partitioning with a set of two logical volumes that is called “Volume Group”. Furthermore, the underlying host makes use of XFS file system [66]. It is the default file system for Red Hat Enterprise Linux 7. This is influenced by the fact that, due to the support of “metadata journaling”, it is highly scalable and performance. Furthermore, the XFS file system can also be defragmented and enlarged while mounted and active. However, if the Atomic LVM thin pool runs out of space, it will lead to a failure because the XFS file system underlying will be retrying indefinitely in response to any I/O errors. For this reason, it is very important to monitor the free space in the docker-pool and not to allow it to run out of space.

Nevertheless, in addition to a LVM thin pool, it is possible to use the so-called OverlayFS, that we have already discussed it with Docker. This is a copy-on-write file system that features page-cache sharing between snapshot volumes. Therefore, it supports efficient data storage and, compared to LVM thin pool, the container creation and destruction is more performant because it makes use of less memory

4.5 Mesosphere DCOS

Mesosphere DCOS is a very robust and innovative way of looking at how to manage containers. The most interesting thing about it is that it is not just limited to container management but it has cluster computing solutions built-in, such as Hadoop, Cassandra etc. This is one of the key differentiators from the other container operating systems that make Mesosphere DCOS [67] very successful. Furthermore, to do that, the project makes use of other open source projects, such as Apache Mesos, Marathon, Zookeeper, and a few other services.

The platform abstracts the cluster hardware and software resources by providing just common services to the applications layer. Similar to Linux, DC/OS has both system and user spaces. The system space is a protected area that is not accessible to users and involves low-level operations such as resource allocation, security, and process isolation. On the contrary, the user space is where the user applications, jobs, and services are running on. Basically, Mesosphere DCOS is not a host operating system but it spans multiple machines and relies on each of which to have its own host operating system and kernel module.

Though containerization offers massive scale, as we have already discussed, it has one significant gap: lack of tight integration with existing stateful applications.

Therefore, DC/OS is designed to manage both stateful and stateless workloads, within the same environment. In fact, due to the integration with atop frameworks like Hadoop and Cassandra, it can handle the scale-out web tier running in containers that talk to them. Furthermore, customers can use Marathon for orchestrating applications while they use Chronos for scheduling long-running tasks. The fact that it is able to run stateful applications, along with the scale-out of containerized applications, gives to the platform a key differentiator factor, compared to other container-focused operating systems. Therefore, the purpose of this section is to investigate the implementation of that solution.

4.5.1 Architecture

The project is a platform for running containerized software. It is infrastructure-agnostic and so may consist of virtual or physical hardware as long as it provides compute, storage and networking. The architecture can be split into three most important layers [43]: software, platform, and infrastructure.

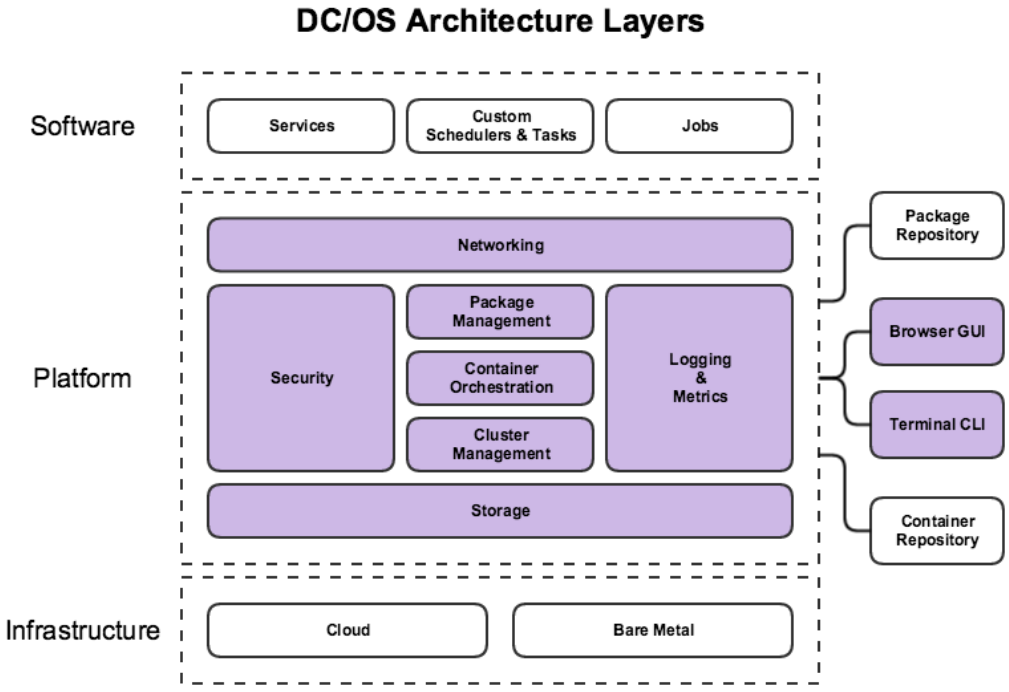

Figure 50 - Mesosphere DCOS Architecture

Figure 50 shows us the layered architecture of Mesosphere DCOS.

The lowest layer is about the underlying resources on which is built. As mentioned above, DC/OS can be installed on public clouds or private clouds but also on-premises hardware. At the platform layer, we can find dozens of components grouped in categories. However, these components are divided across multiple

node types: master nodes, private and public agent nodes. Therefore, each node must already be provisioned with one of the supported host operating systems.

The highest layer is the one which is shown to users. This provides package management and a package repository to easily install and manage multiple types of services. In addition to these packaged applications and services, the user may install their own custom apps, services, and scheduled jobs. Furthermore, the project includes and integrates several external components, such as a graphical user interface (GUI), a client command line interface (CLI), a package repository, and a container registry. These are used to build a set of application stacks that are designed by focusing just on the business core of the use case requirements.

This is one of the most adopted solutions in this segment market. This is influenced by the fact that the sauce behind the platform is the design strategy used to power such robust applications.

However, it is a sort of fit-for-all because takes advantages of existing solutions like Apache Mesos. This has led to the introduction of the so-called Container 2.0 workloads. Mesosphere goes toward this direction in order to provide additional functionalities such as: simultaneously running multiple schedulers and supporting the multi-tenancy. Container 1.0 systems do not optimize these workloads and users end up with non-optimal operating constraints, including being forced to separate clusters for each service. In conclusion, we can affirm that Mesosphere DC/OS is the best of all the aspects that users can experience according to the possibilities of the containerization paradigm. In fact, it is open source and easy to build each type of application that can propel organization business into the next evolution of the digital age.

4.5.2 Network

The platform provides a number of tools out-of-the-box, ranging from basic network connectivity between containers to more advanced features, such as load balancing and discovery. The so-called “IP Per Container” functionality [39] allows containers to run on any type of IP-based virtual networks. The project supports this capability for the Universal container runtime (UCR) by using the Container Network Interface (CNI). Furthermore, it can also use the Docker container runtime by using the Container network model (CNM). This consists of a virtual networking solution that works both with UCR and Docker container runtimes. Basically, it is an overlay network and it makes use of the underlying Mesos support to provide a unique network address to each container.

Furthermore, DC/OS includes other services such as a resolution name system and load-balancing. The resolution name system is accomplished by two components: a centralized Mesos DNS, which runs on every master; a distributed component called Spartan that runs on every agent. Lastly, the load balancing option is provided out-of-the-box by three implementations: Minuteman, Edge-LB, and Marathon-LB. However, there is a difference with the concepts already introduced in the Mesos section.

4.5.3 Storage

Applications lose their state when they terminate and are started again. This is not a suitable case for all scenarios, such as a database or a stateful service like Kafka or Cassandra. So, in order to preserve the durability, it is necessary to configure Mesos to mount disk resources to enable users to create tasks that can be restarted without data loss.

Disk Mount Resources [42] are primarily for stateful services and consists of a dedicated storage available throughout the cluster. However, it is still important to consider the performance and reliability requirements for the applicative use case. In fact, they are built by taking advantages of the underlying storage and so it is not its responsibility to provide data replication services.

Furthermore, in DC/OS, there are other types of persistent resources that are classified in: local and external persistent volumes. In such way, tasks and their associated data are stored to the node and will not be lost, even if the container on that node will terminate. Nevertheless, this guarantees to the application reserving its own persistent state.

4.5.4 Container Orchestration

DC/OS provides easy-to-use container orchestration right out of the box. It includes Marathon [39] as a core component, giving to us a production-grade, battle-hardened scheduler that is capable of orchestrating both containerized and non-containerized workloads. This allows to us the ability to reach extreme scale, scheduling tasks across a several numbers of nodes. Moreover, the application definitions are configurable using a declarative approach to enforce advanced placement constraints with node, cluster and grouping affinities.

4.6 Snappy Ubuntu Core

The Snappy Ubuntu Core Operating System comes with a new type of application manager, that is called snappy, and focuses on running applications and containers. It abstracts the lower-level functionalities that are introduced by etcd, Consul, fleet, Kubernetes, and all the other tools. The base of the system is the "Ubuntu Core" [67]. On top of that, applications are realized by read-only images that can be transitionally updated. This means that it is not needed to download an entire application to deploy a new version. In fact, it is enough to just download the changes that have been made.

Snappy is a very tiny and thin operating system. It is the result of a long work that Canonical performed in order to create a tiny-yet-robust operating system for mobile devices. Furthermore, the increasing demand of users for reliable systems and application updates, the target of this solution is to build the so-called "transaction, image-based delta updates". This consists of transmitting only differences to keep downloads small and ensure that upgrades can always be rolled back.

Therefore, to enhance the security of mobile devices, they created a containment mechanism that isolates each application running on the device. This is the same capability that is available using Docker standalone. In addition, for security improvements, Snappy took advantages of LXD. However, Canonical continues to recommend Docker for packaging and running applications. In conclusion, Snappy Ubuntu Core is not a pure container OS but has some interesting aspects to it. For this reason, the purpose of this chapter is to investigate this solution.

4.6.1 Architecture

The target is Internet-of-Things (IoT) and mobile devices that need support in order to provide high-quality services. In the classic Ubuntu, any package consists of writing to any file. The Snappy approach is different because there are two types of package units: read-only and writable spaces.

This architecture guarantees automatic updates, backups, rollback and by design the system is secure. In fact, these system packages are confined and isolated and so, the changes are not spread all over the system such as in classical ubuntu systems. As it will be seen soon, the build process packages everything needed into a single "snap" file. For example, "python runtime env" needs to be packaged into the snap package for python applications.

Snappy is different from a traditional package-based Ubuntu server and desktop OS. In fact, it guarantees the isolation of each system part in a separate read-only file and does the same for each application. This allows developers to confidently update their applications without worrying about breaking other installed software.

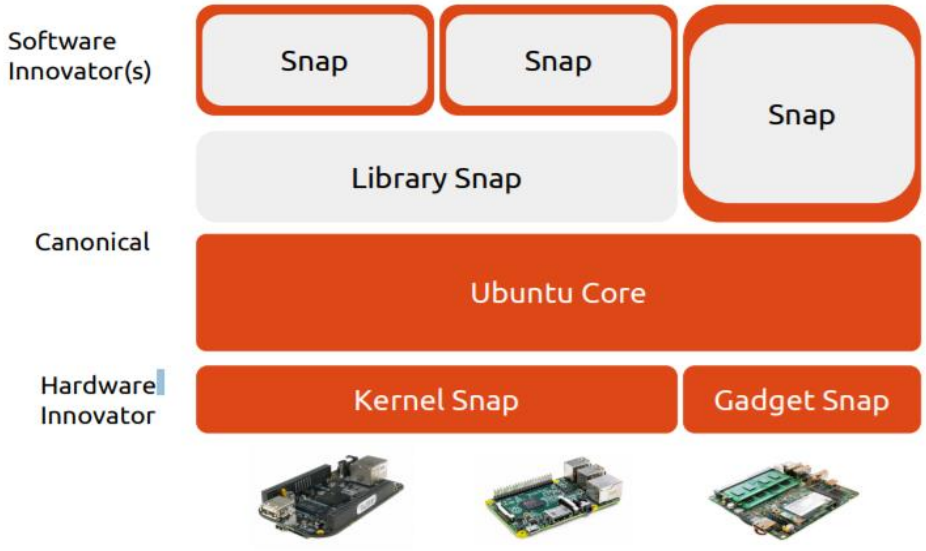

Figure 51 - Ubuntu Snappy Architecture

Figure 51 gives us the layered architecture of Snappy Ubuntu Core.

As it is possible to notice, the underlying target resources concern about IoT devices in which the purpose is to quickly and securely deploy system components. So, there is the need to build a sort of middleware which is able to spread applications, without worrying about system configurations. For this reason, Ubuntu Snappy Core adopted the principle of "snap packages", which are much similar to Docker containers and so they do not expose operators and developers to face with system-related configuration problems.

4.6.2 **Containerization with Internet of Things**

Numerous factors are affecting the complexity of modern enterprise applications. Of course, this is influenced by the surging adoption rates of mobile technologies, distributed environments, big data and its near instantaneous transmission. Furthermore, these have considerably complicated enterprise architecture. These factors are exemplified by the Internet of Things (IoT) [68], which is augured to involve tens of millions of connected devices by the outset of the subsequent decade.

Nevertheless, organizations are increasingly attempting to remedy these complexities by adopting virtualization technologies, in which data is made available as an abstraction layer accessible to different parties from distinct location. Containerization represents the next level of virtualization solution by exploiting the possibility to provide the benefits of real-time application data in a post-IoT world.

Furthermore, running applications as microservices could very well be the best means of creating and deploying them in time to account for the extreme volumes of IoT and velocities of data, especially when they are leveraged within containers.

Microservices are especially well suited for the IoT because of the machine-to-machine capacity of the latter. In particular, numerous IoT deployments involve machine learning. The intersection of the IoT, microservices and containerization revolves about this fact. In fact, once people get that part of the architectural thinking down they can realize that microservices will have configuration files, perhaps task-specific libraries associated with them. In such case, containerizing that makes it even easier for the DevOps folks to deploy those containers across the infrastructure.

Nevertheless, the benefits do not concern just development features. In fact, one of the most important issue that containers aim to solve is the security. Snappy Ubuntu Core is the version of Ubuntu that is built around container to address the IoT world. The core mechanism, snap, offers automatic updates and helps blocking unauthorized updates. Using transactional systems management [69], snaps ensure that updates either deploy as intended or not at all. In Ubuntu Core, security is further strengthened with other security-target solutions, like AppArmor. Therefore, all application files are kept in separate silos, and these are just read-only. This characterizes the ability run snap packages on any major distribution, including Ubuntu Server and Ubuntu Cloud, by allowing users to provide a coherent experience. In literature, this is meant as the relevance from edge to gateway to the cloud. On the contrary, by applying virtual machines to IoT, the performances issue and restrictions on direct hardware access is quite limited. Using container technologies, like Docker, may be natural for enterprise developers by exploiting the features of the containerization paradigm.

4.6.3 Package build

Snapcraft is a tool which lets developers package their software as the so-called "snap" file. This allows to incorporate components from different sources and build well-designed technologies and solutions. However, the tool needs to run on an Ubuntu OS distribution. In Snappy Ubuntu Core "snaps" is the packaging mechanism that is used to build and deploy applications. They are self-contained and made from reusable components, that are called "parts". Thus, developers can include all required dependencies in their snaps in order to remove any dependency on system libraries. Furthermore, they can leverage existing open source projects by integrating them as part of their snap.

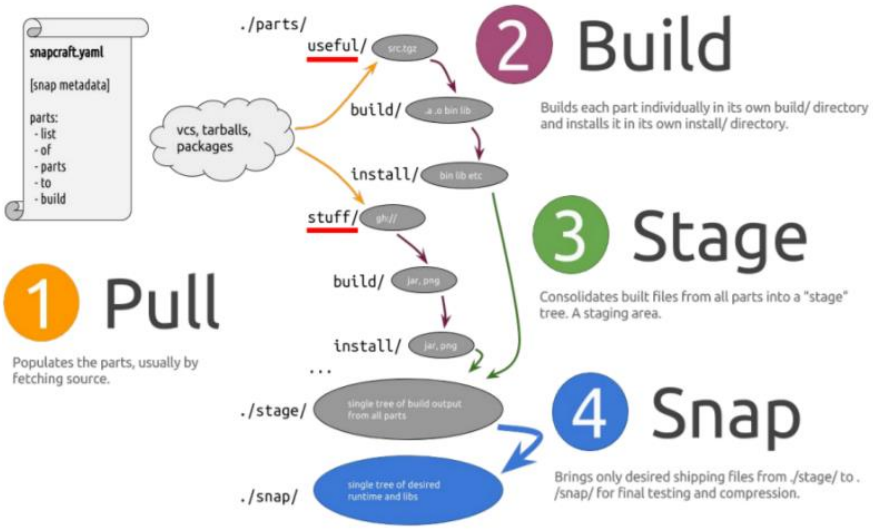

Figure 52 - Snap Package Build

Figure 52 shows us the building process of snap. These are designed to be secure, sandboxed, and isolated containerized applications. A snap consists of a fancy zip file containing an application together with its dependencies, and a description of how it should safely run on the system. As described in the figure, the building process is made up of four phases: pull, build, stage, and snap. Pull consists of fetching the package source. After that, it is needed to configure the local system with the installation of each subcomponent. By this way, the output files of the previous phase are consolidated in a sort of tree through the usage of the "stage" process. Lastly, the desired components are put in a snap. This is read-only for security because the aim is to prevent a hostile party from sneakily changing the software on the underlying machine. Therefore, it is not possible to modify a snap once that it is installed on the system. Furthermore, it is even possible to check the snap signature to make sure that it still exactly the intended software.

4.7 Closing remarks

Container Operating System is the last level of the layered point of view of the containerization paradigm. This is very important for combine advantages of containerized applications and distributed operating systems. Table 8 shows us a comparison of the analyzed solutions.

	CoreOS	**RHEL Atomic**	**Mesosphere DCOS**	**Snappy Ubuntu Core**
Use case	Large-scale deployments	Large-scale deployments	Large-scale deployments	IoT and mobile devices
Auto-updates	Yes	Yes	It relies on the underlying kernel host	Yes
Rollback version system	Yes	Yes	No	No
Container Orchestration	Fleet	Kubernetes	Marathon	Not included
Application services	Service Discovery, Container networking, Resource scheduling	Service Discovery, Telemetry, Security	Stateful scale-out, Service Discovery and High-Availability	Usage of snaps

Table 8 - A comparison of container-focused operating systems

As expected, each implementation has its own key differentiator features and so it is more suitable in some applicative use cases. Even in this case, there is not a fit-for-all solution but each proposal is suitable for the correspondent use cases. Therefore, the analysis phase requires the indispensable evaluation of the right choices. In conclusion, this means that users should perform an in-depth analysis in order to choose the right one which is more suitable for the specific requirements.

5 Containers with OpenStack

5.1 Overview

OpenStack is the leading cloud framework for adopting and adapting new technologies. So, the community, influenced by containerization success, decided to support this new virtualization paradigm. This has led several projects to ensure that containers, and the third-party of ecosystems, are completely supported in OpenStack deployments.

For example, OpenStack compute service manages to compute resources which may be virtual machines but also containers. These are suitable for use cases with the requirement to treat a container like a lightweight virtual machine, allowing the usage in a similar way to on-demand virtual machines. Of course, this is the case of operating system containers.

The goal is to allow users to create and manage containers similarly to how they use the Nova service to get virtual machines. For this reason, the focus of OpenStack is based on three important areas: supporting containerized workloads, simplifying the setup to run a production multi-tenant container service and offering a modular choice to operators who have not established a definitive containers strategy yet.

In order to deal with those areas, several projects have been designed with the aim to easily embrace the containerization paradigm in cloud deployments like OpenStack. So, this chapter will investigate these solutions in order to provide an in-depth overview of which implementations are now available to adopt and what are the correspondent use cases to be associated.

5.2 Cloud Computing

Cloud computing has changed business models and design patterns used to develop applications. The introduction of this new buzzword has produced the spread of several opinions and politics on when is better to adopt that model. However, the most spread definition is from NIST, which defines the cloud as a useful model to access resources on-demand. These concern about servers, networks, storage and services which can be rapidly provisioned with a minimum interaction between users and service provider. Furthermore, another important feature, that has influenced the cloud adoption, is the so-called pay-per-use

concept. This means that users pay just for the usage time avoiding to waste money when services are not needed.

Cloud computing was promoted with different provisioning and deployment models. Considering service provisioning, there are three proposals: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). Using IaaS, users have the responsibility to manage infrastructure components. At the PaaS layer, there is no visibility on the underlying infrastructure but, the system provides a set of services that can be used in order to build a distributed cloud application. Lastly, the highest level, Software as a Service, provides users the unique possibility to interact with the application without no visibility on the underlying services.

Considering the cloud deployment, the adopted models are classified in: public, private or hybrid. A public solution consists of directly using resources provided by an external provider. Even if there are many advantages, often this solution becomes difficult to be adopted because organizations should completely rely on cloud providers. Therefore, it is often deployed an on-premise cloud solution, which is called private cloud. The last option is to combine advantages of both solutions and the result is called hybrid cloud.

5.3 OpenStack

OpenStack is an open-source IaaS cloud platform composed by a set of services for building and managing resources. Each service handles a specific offer to users. For example, Nova manages and spawn virtual machines, Neutron creates virtual network resources, and Swift manages different kinds of storage. OpenStack makes easy horizontal scaling adding new compute nodes on demand without any specific software requirement. The OpenStack project benefits from a huge community who decide to focus on a solution without no need to spend money on commercial licenses. The interaction mechanism with the platform can be performed through REST APIs or a web-based interface.

5.3.1 Architecture

An OpenStack deployment consists of three main services: compute, networking, and storage. Compute is designed to provision compute resources. The storage module provides the concept of “Volume” to store persistent data but also virtual machines images. The last one, networking, manages two types of communication: services among virtual machines

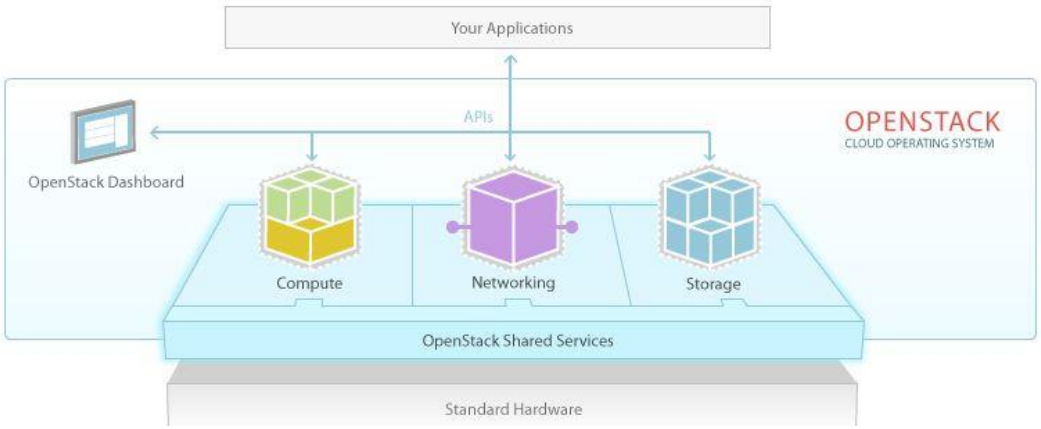

Figure 53 - OpenStack Architecture

Figure 53 shows us the architecture of a basic OpenStack deployment.

OpenStack is based on seven key components, which are shown in Table 9. They are a part of the OpenStack core and maintained by the OpenStack community.

Service Type	**Service Name**
Compute	Nova
Object Storage	Swift
Identity	Keystone
Dashboard	Horizon
Block Storage	Cinder
Network	Neutron
Image Service	Glance

Table 9 - OpenStack Projects

Nova manages computing resources and is the main project of OpenStack with the purpose to guarantee: scalability, fault-tolerance, and compatibility with APIs of other solutions, such as Amazon EC2.

Object Storage is useful to store and retrieve data and, it is based on Cloud Files of Rackspace. This is a safe, efficient, and convenient data-storage system.

Keystone is the module responsible to provide authentication and authorization. It adopts a role-based access control strategy to avoid untrusted accesses to cloud services. After completed the authentication process, users get an authorization token that is used to make aware services about which role the user is running with. Keystone, as other OpenStack services, makes use of external solutions like MySQL to store persistent management-data.

Horizon is a web-based interface that allows users to interact with OpenStack services, without no need to install the client command of an OpenStack

component. This is useful to start and stop a virtual machine or interacting with other services in order to manage networking, storage, and so on.

Block Storage provides to virtual machines a way to store persistent data. To do that, the concept of volume is introduced also in OpenStack. Each volume consists of a sort of virtual hard drive and it is attached to a virtual machine. So, information is saved on blocks of a fixed size. Cinder is the most popular component to provide the block-storage service. It guarantees high-availability and fault-tolerance.

Neutron is an OpenStack project to provide "network connectivity as a service" and so, it provides an API that allows users to set up and define network connectivity and addressing in the cloud. This handles the creation and management of a virtual networking infrastructure, including networks, switches, and routers for devices managed by the OpenStack Compute Service (Nova). Neutron consists of a neutron-server, a database for persistent storage, and several plug-in agents, which provide other services such as interfacing with native Linux networking mechanism, external devices, or SDN controllers.

The last component is an Image Service called Glance. Images are necessary to spawn a virtual machine. They are disc types with a pre-installed operating system that, at the booting phase, will be attached to the compute instance.

5.3.2 Nova System Architecture

Nova is a project which is composed of different processes, each of which is dedicated to performing a specific functionality. Users interact with Nova by using REST APIs or Horizon, while components inside Nova communicate through a remote-procedure-call (RPC) message passing mechanism.

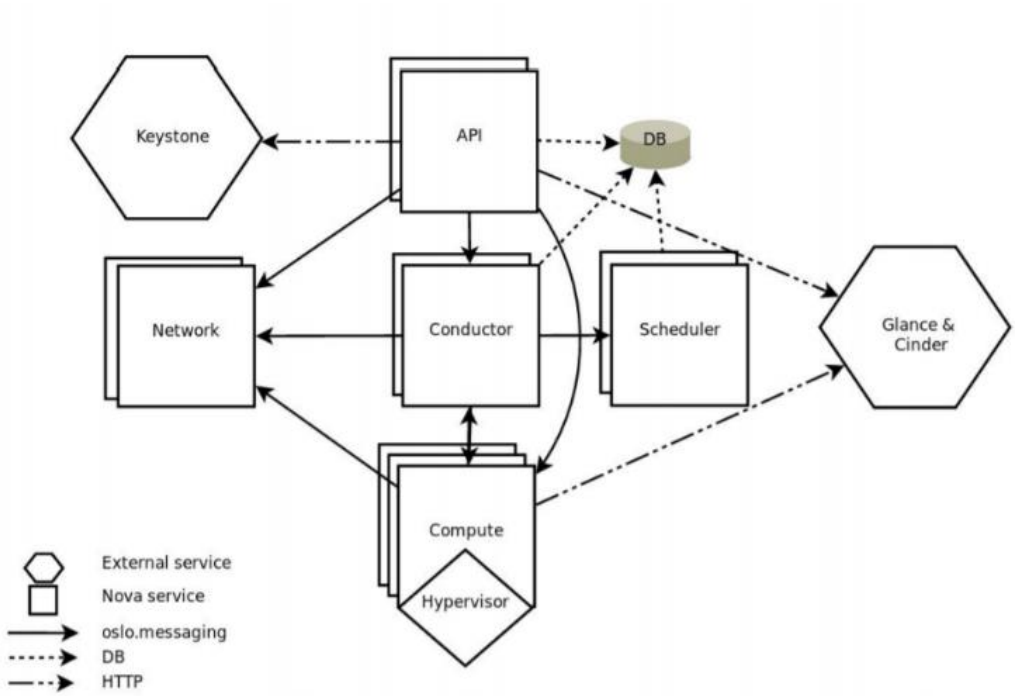

Figure 54 - Nova System Architecture

Figure 54 describes the architecture of the Nova system. The main component is the API server process that receives client requests, which typically, does read and write from and to a database. Furthermore, this process can interact, by sending messages, with other Nova components and the cooperation flow determines the response to the REST invocation.

The RPC messaging communication is done by using the “oslo.messaging” library. This is an abstraction made up on a message queue and used to guarantee both time and space decoupling between involved components. Furthermore, it is possible to install nova components on different servers with a manager that listens to RPC messages. The unique exception is about nova-compute, a single process that interacts with the underlying hypervisor to manage compute instances. Nova uses a central database logically shared between different components. The access to this structure is performed through nova-compute, in order to maintain independence from eventual updates of the whole service. Furthermore, Nova-compute sends RPC requests to a central manager, which is called nova-conductor.

Nowadays, each cloud computing platform consists of three compute instance types: virtual machine, container, and bare-metal server. Nevertheless, it is necessary to analyze when an organization can spend money on a traditional virtual machine infrastructure or other complementary solutions. It is also difficult to migrate from a current deployment type to another one and so it is important to evaluate the introduced overhead by migrating applications. Therefore, these scenarios should be investigated determining which parameters should be considered when a cloud-based infrastructure is needed. For this reason, at the end of this work, a performance analysis between containers and virtual machines will be discussed.

We have already learned that containerization does not meet everything and so we need to investigate different use cases in order to understand when is more suitable to use virtual machines instead of container-based deployments.

In OpenStack, each deployment model is associated to a different sub-project:

1. **OpenStack Hypervisor based**, which uses solutions like KVM to manage the whole lifecycle of virtual machines. The communication between OpenStack and KVM is managed by the driver nova-libvirt.
2. **OpenStack Container-based**, which uses a container engine to manage containers. The communication between OpenStack and the underlying container engine is managed by a driver, like nova-docker or other newer solutions, such as nova-lxd.

   In this case, Nova is responsible to make use of drivers in order to spawn instances and interact with Neutron for the networking service.

3. **OpenStack bare-metal**, which uses the Ironic project to directly request physical server instances.

   In contrast to the previous case, with OpenStack bare-metal, Nova delegates this responsibility to Ironic which has the duty to provide a physical server instance and interact with the other OpenStack projects.

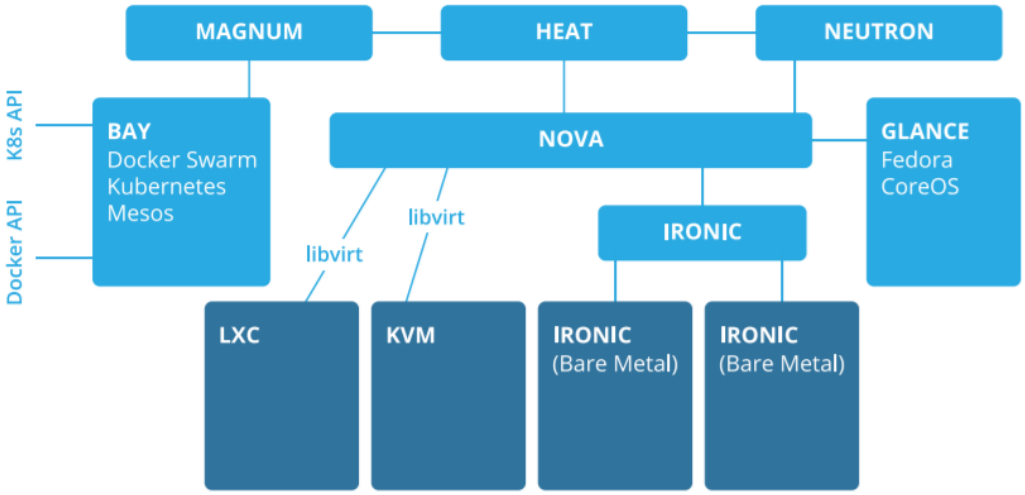

Figure 55 – Virtualization Architectures with OpenStack

Figure 55 shows us an overview of different virtualization solutions that are integrated with OpenStack. The Nova service can be structured by supporting one of these three computational models. LXC is used for containers, KVM for virtual machines and IRONIC for physical servers. To do that, Ironic uses Preboot Execution Environment (PXE) and Intelligent Platform Management Interface (IPMI). PXE is a standardized client-server environment that boots a software, retrieved from a network, on PXE-enabled clients. IPMI defines a set of interfaces used by system administrators for out-of-band management of computer systems and monitoring their operation. This is important to manage a computer that may be powered off or otherwise unresponsive by using a network connection to the hardware rather than to an operating system or login shell. In this way, Ironic is able to switch on a physical server and install on that an operating system that is booted though a network connection. After completed this phase, Ironic interacts with the other projects in order to complete the provisioning process.

5.3.3 The adoption of software-container in OpenStack

OpenStack supports containers on bare metal or virtual machines [70]. However, this requires the attention of operators who must be aware that containers do not have the same security isolation capabilities as virtual machines. Therefore, service providers often run container in virtual machines in order to provide robust protection of processes, which belong to the same tenant, from poorly behaved or malicious code in other containers. Another way consists of the introduction of the bay concept. In this case, a set of virtual machines make a bay, which is a cluster of instances, that is only used by one tenant to address this risk.

OpenStack provides support for all of these configurations in the role of the overall data center manager. There are multiple OpenStack projects leveraging

container technology to augment the OpenStack quality of usage: Magnum, Kolla, and Murano.

Magnum is designed to offer container specific APIs for multi-tenant containers-as-a-service. Murano is an application catalog solution that offers users the possibility to quickly deploy packaged applications whereas Kolla is the solution to offer a dynamic OpenStack control plane where each OpenStack service runs in a Docker container.

5.4 OpenStack Magnum

The adoption of the software-container paradigm has influenced the production of a new OpenStack project, Magnum. Its target is to provide a sort of API layer to implement the so-called "Container as a Service". The principle is the same as in other projects, such as Nova to provision compute instances and so on. As described in the correspondent chapter, the introduction of containers used to get more complexity in management operations. For this reason, a new layer called container orchestration was designed for. This has influenced the production of several solutions and the idea of Magnum was to integrate into OpenStack a subset of these platforms. Furthermore, the purpose is to combine both advantages and using OpenStack principles, such as multi-tenancy, lifecycle management and so on. Moreover, users should be able to continue to use the client native tools to deploy their own applications.

Actually, the supported orchestrations are Docker Swarm, Kubernetes and Apache Mesos. This project was not designed to be a complementary solution, but it focuses on requirements that are not still supported, such as multi-tenancy. This is accomplished by taking advantages of existing projects like Keystone which implements the fundamental concept of multi-tenancy. This is not at all, in fact, Magnum allows to users the possibility to contemporarily use multiple orchestration instances, even from different implementations. This is an important feature, considering the fact that a generic solution has several use cases but also other aspects not properly covered.

5.4.1 Architecture

The idea of Magnum is creating a cluster of servers, each one configured with a container orchestration engine in order to deploy containerized applications. Furthermore, the project takes advantages of existing solutions like Nova and Neutron in order to provide the same features as multi-tenancy and automated management of the underlying resources. For this reason, the project makes use

of the concept of “Bay”. This comprises a set of compute instances, logically interconnected due to the capability of Neutron, in order to quickly provide a cluster solution that users can exploit with a container orchestration system.

The architecture of Magnum is structured as in other OpenStack services. So, even in this case, there is an API server which is responsible to take care of the service requests invoked by clients. Basically, it consists of a component that implements its functionalities according to a simple create-read-update-delete (CRUD) model. This allows users the possibility to work with Magnum resources like “Bay” and “BayModel”.

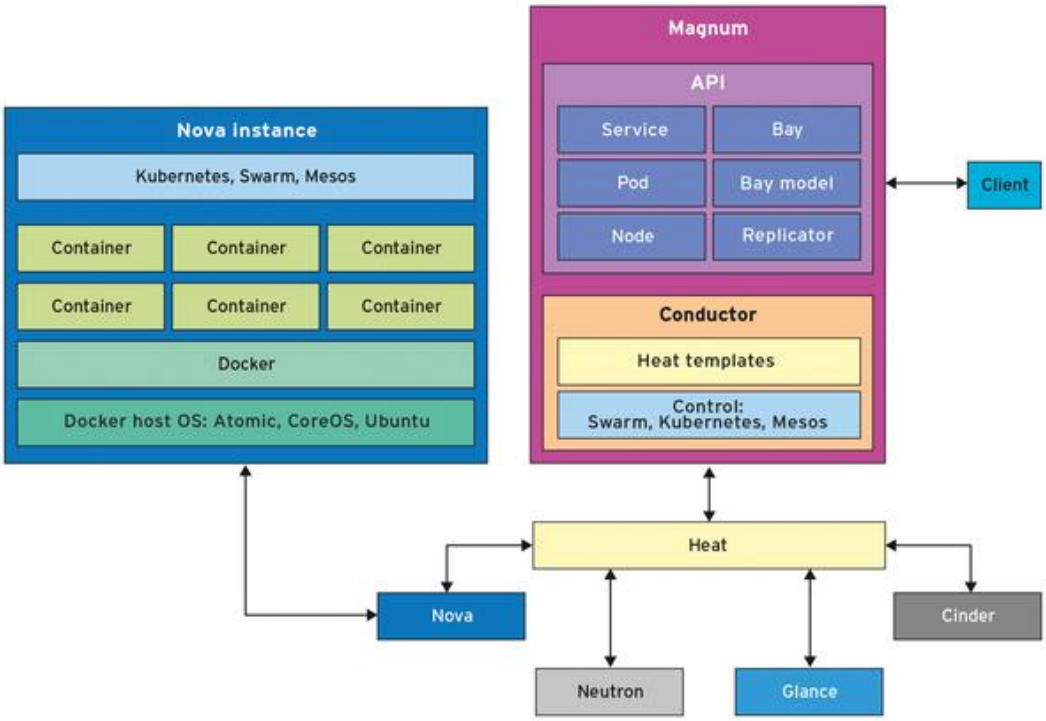

Figure 56 - Magnum Architecture

Figure 56 illustrates the architecture point of view of a Magnum solution that is integrated with an OpenStack deployment. As it is possible to notice, the service core is placed on the controller node, as the other components, and interact with them in order to provide a server cluster able to make use of a container orchestration system.

Therefore, the Magnum API server offers the possibility to work with COE-dependent resources, such as Service, Pod, and Nodes. The invocation of those primitives notifies the action of a component inside Magnum, which is called Magnum-Conductor. This is responsible to interact with Heat in order to orchestrate resources to be managed for cluster provisioning. Practically

speaking, this involves Nova to create instances and even Neutron to create the private network inside the cluster. Then, Glance is used to loading distro images and Cinder to provide the possibility to create mount points on containers. In fact, each nova instance is deployed with a container runtime, configured according to the specific orchestration system defined in the BayModel. However, these magnum resources will be deeply discussed in the correspondent section whereas the purpose of this part is to provide an overview of the whole Magnum architecture.

5.4.2 Network

As mentioned before, for network services Magnum is based on existing OpenStack solutions. So, it makes use of Neutron to configure the network of the whole cluster. This allows each instance the possibility to communicate with the others that belong to the same cluster. Of course, the implementation differs according to the specific orchestration engine that is willing to use. An example is about Kubernetes that is based on the "Flannel" driver, which establishes an overlay network to assign an IP per pod, regardless of the host which is currently executing on. On the contrary, if the chosen orchestration system does not support the multi-host networking, the network subsystem will avoid implementing such functionalities.

5.4.3 Security and Multi-tenancy

Magnum resources are started and will be accessible only to users which belong to the same tenant, in which the compute instances have been created. Bays are not shared and so containers will not be running on the same kernel of neighboring tenants. This is a fundamental feature in terms of security because containers of different tenants will be executed on separate nova instances. Of course, this is quite different from orchestration solutions, such as Kubernetes. In fact, without the Magnum support, there is no possibility to include the multi-tenancy considering that Kubernetes was not designed to offer multi-tenancy and so it leaves this responsibility to users.

Magnum does not include isolation functionalities but it takes exploits the capability offered by Nova when virtual machines will be instantiated. Therefore, if Nova allows isolation between different tenants, Magnum would be considered secure on providing an isolation level between containers.

5.4.4 Magnum API Objects

According to the Magnum terminology, the cluster is called “Bay” (actually changed with Cluster). A Bay is a Magnum resource, which is created with a dedicated container orchestration system. Basically, it consists of a set of Nova instances that are provided with a pre-defined configuration containing network capabilities, security groups and so on. However, as seen for multi-tenancy, Magnum does not include an orchestration functionality and so, it exploits existing orchestration projects like OpenStack Heat. The orchestration component is based on templates. As anticipated in the correspondent chapter, the cluster can scale up or down according to the desired state. Magnum uses Heat templates, that are pre-defined or manually created by users, in order to define an architectural design useful to repeat the scenario in similar use cases.

Heat Stack Templates are what Magnum passes to Heat to create a cluster. For each template, a Heat Stack is created to arrange all of the cloud resources needed to support the container orchestration environment. The purpose is to provide a mapping of Magnum object attributes to the Heat Stack Template. This allows Heat service to create the so-called “Heat Stack” that is the enabler to the output Container Orchestration Environment.

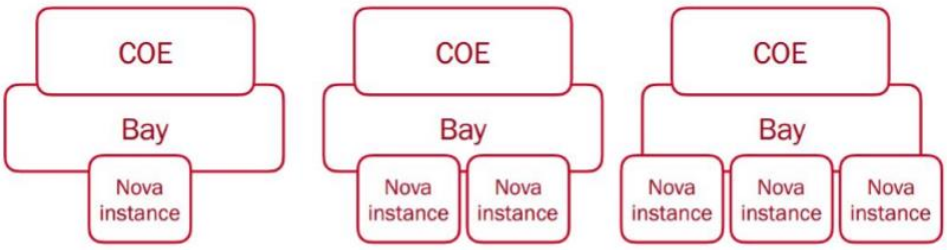

Figure 57 - Magnum Bays

Figure 57 shows us the concept of the Bay resource which is the minimum Magnum deployment unit to guarantee the container as a Service paradigm. Therefore, clusters that belong to different tenants will be configured in such a way that containers will not be running on the same kernel host. This is the responsibility of Nova and Neutron. Furthermore, as anticipated before, they are not the unique projects that are used to implement Magnum services. The idea is to provide high-level APIs without reinventing the wheel and so the principle is to combine features that are already provided in OpenStack, due to the presence of several existing projects.

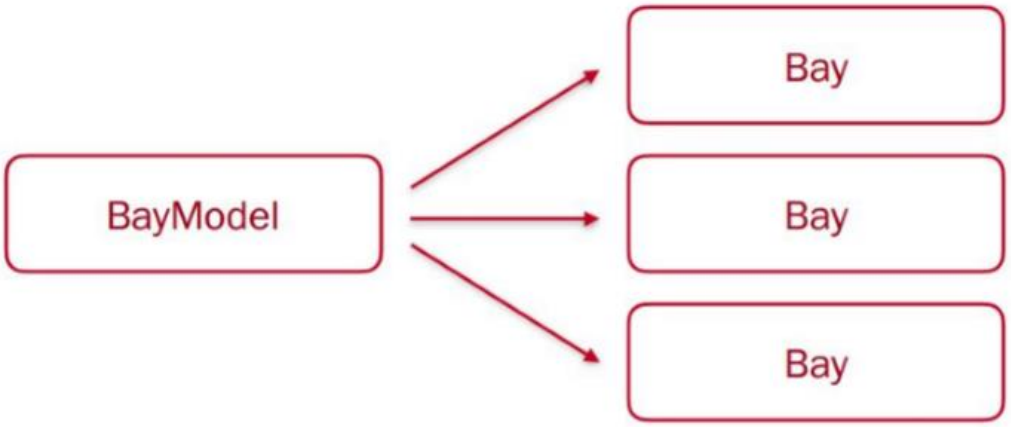

Figure 58 - Magnum BayModel

Figure 58 shows us the concept of a BayModel.

This is dependent on the container orchestration engine (COE) that is willing to be installed on the cluster instances. In order to create a Bay, it is needed to specify a BayModel. This is the template which contains the definition of a set of parameters that are used by Magnum to configure the whole cluster. Therefore, this resource is shared between bays that are created from the same template.

Each COE implementation supports at least a cluster driver. In fact, as mentioned before, Magnum does not include everything but makes use of existing capabilities. So, a cluster driver consists of a set of resources that are the following: heat templates, configuration scripts, cloud images and documents referred to a particular container-orchestration-engine (COE). The cluster driver is used to enable the infrastructure for a specific container orchestration.

COE	**Distro**
Kubernetes	Fedora Atomic, CoreOS
Swarm	Fedora Atomic
Mesos	Ubuntu

Table 10 - Container Orchestration Engines with Magnum

Table 10 shows us the pre-installed drivers that Magnum includes for each COE. These are associated with the distributions that are already included in the other OpenStack services. In fact, Magnum makes use of the OpenStack principle that is completely based on the integration strategy. So, it is even possible to integrate other cluster drivers. To do that, it is necessary to define the driver as a directory which contains some specific files in order to support the building of a container orchestration system.

Furthermore, in the cluster driver is specified the association mapping between the BayModel definitions and the correspondent Heat template. The workflow uses Glance to load a compute image to be installed on Nova instances. These instances are built with a pre-defined container orchestration engine that is used by clients, according to the specific declarations of the BayModel.

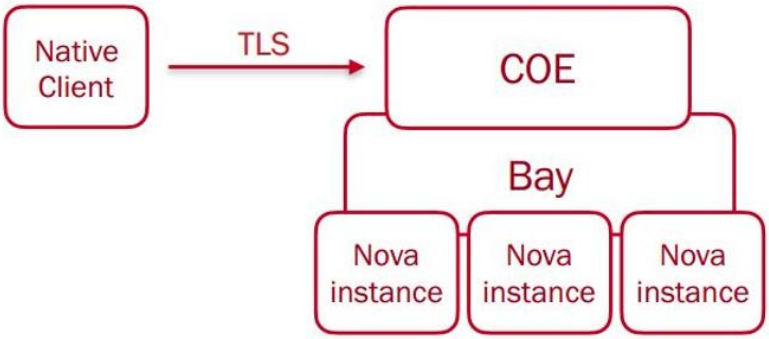

Figure 59 - Magnum API services

Figure 60 shows the API services which, by default, they make use of secure communication based on Transport-Layer-Security (TLS). However, it is even possible to disable this feature configuring the proper configuration in the cluster definition. Nevertheless, we understood that Magnum takes care only to provision cloud resources that are indispensable for executing the container orchestration engine. Furthermore, the remaining is implemented according to politics and mechanisms that each existing solution provides.

5.4.5 Resource lifecycle

Magnum is an OpenStack API service that offers the possibility to integrate an existing container orchestration engine such as Docker Swarm, Kubernetes and Apache Mesos. Furthermore, it takes care of the complete lifecycle management of each container orchestration engine. Therefore, the purpose of this section is to investigate how this fundamental feature is managed by Magnum. Of course, this includes each Magnum resource and the entire flow of the management process is described by a state diagram, as shown in the picture below. Figure 60 the diagram state that characterizes a Magnum Bay.

Figure 60 - Magnum Bay Lifecycle

Figure 60 presents the resource lifecycle. Each operation is asynchronous and so it is possible to instantiate another cluster, before that the previous has reached the "completed" state. Of course, it is possible that the creation process can fault. Currently, cluster drivers make use of Heat templates and so resources will be automatically detected due to the correlation between Magnum and Heat.

There is no difference with the compute instances created through Magnum and those created through other projects, such as Nova. This means that they are accessible to the cluster owner and with the possibility to perform any actions. Furthermore, by modifying directly a single resource is not well considering that this behavior is completely unexpected from the Magnum components. In fact, Nova instances, created outside the Magnum service, will not be considered as those, created by Magnum. Therefore, if a "cluster-delete" operation is invoked, the private network created for Magnum cannot be deleted until there are instances, created outside Magnum, that are attached to that network.

5.5 OpenStack Zun

Zun (ex. Higgins) is a Container Management service for OpenStack. It aims to provide an OpenStack API for launching and managing containers backed by different container technologies. In fact, the aim is to abstract the whole container lifecycle management. The solution is fairly new but it deeply integrates with other OpenStack solutions, such as Keystone, Nova, Neutron, Glance, and Horizon.

Zun and Magnum are two independent solutions. Magnum provides APIs to provision and manage Container Orchestration Engines (COEs), such as Kubernetes. On the other hand, Zun [71] is not specifically based on users who

want to adopt a specific orchestration solution in OpenStack. In fact, it focuses on users who want to create and manage containers as an OpenStack-managed resource. This means that users can manage containers without the need to explore the complexities of different container technologies.

Furthermore, Magnum is for users who want a self-service solution to provision and manage an orchestration cluster. On the contrary, Zun is completely based on OpenStack container provisioning in order to manage and perform the basic operations within the OpenStack container management platform. This is one of the motivations that has led the OpenStack community to suspend the support of Nova-docker. In fact, Zun interacts with container compute instances without the need to have a tight-coupled relationship with Nova.

5.5.1 Architecture

Basically, Zun is designed as other OpenStack projects. There is a server process whose aim is to receive and process client requests. Furthermore, it is responsible to interact with another important process that is called "Zun Compute". This process is responsible to launch containers and manage the underlying compute resources. Moreover, it may interact with Nova in order to create a sort of sandbox, that is actually a docker container. Nevertheless, the concept of "Zun Sandbox" will be deeply illustrated in the next section.

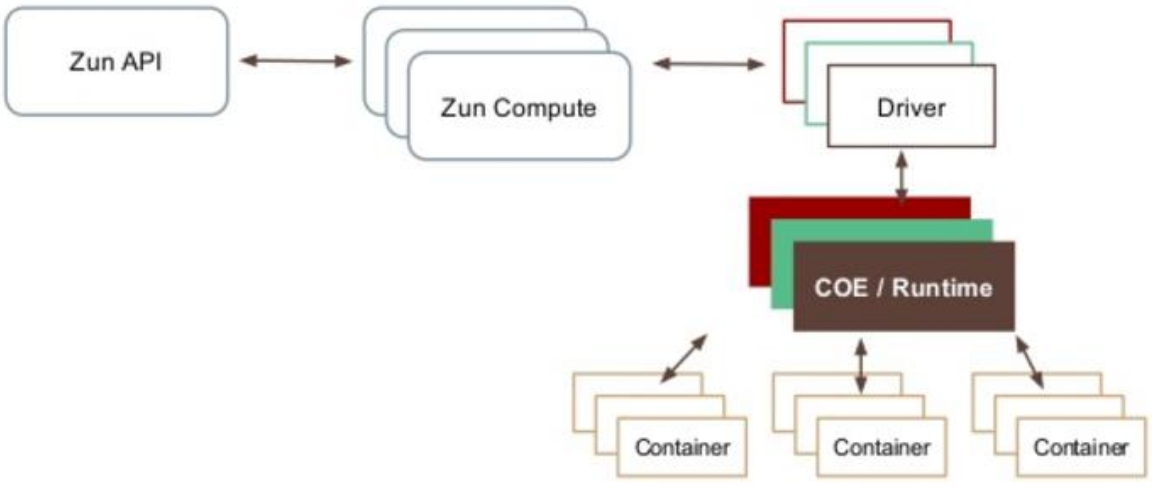

Figure 61 - Zun Architecture

Figure 61 shows us the whole architecture of the Zun project. As seen in Magnum, this involves Nova instances, that are scheduled by Nova scheduler, with Neutron ports attached for providing networking capabilities. Containers are created by Zun and will run inside the context of a sandbox. All containers that belong to the

same sandbox will be located on the same host in order to share the Linux namespaces of the entire sandbox.

The key aspect of the Zun project [72] is to support various container technologies in OpenStack. Such container technologies include Container runtimes (i.e. Docker, Rkt, Clear Container) and COEs (i.e. Kubernetes, Docker Swarm etc.). However, these two groups look very different from each other and so it is hard to abstract all of them into a common set of APIs.

Therefore, the decision of Zun was to separate the support of these two groups of technologies. Firstly, Zun deeply integrates with existing COEs. In fact, the exposed APIs provide the common feature set among prevailing COEs, such as deploying an application to one or multiple containers, and more. On the contrary, it is needed to provide a sort of API layer that is specific for Zun. So, the project focuses on the basic management of a single container and integrates those containers with existing OpenStack primitives (like networking, storage, authentication, monitoring etc.).

5.5.2 Comparison between Zun and Magnum

The basic promise of Zun is to fill the gap from where the project Magnum ends. Magnum is really just a system for deploying a container orchestration system like Kubernetes, Apache Mesos or Docker Swarm. It provides provisioning as well as scaling capabilities and security feature, serving as a certificate authority and generating OpenStack Keystone users.

As we have already seen, originally, Magnum was introduced with the mission to be a container service. In fact, the official mission statement was to provide a set of services for management application containers in a multi-tenant cloud environment. Then, Magnum has been changed and now is considered as a Container Infrastructure Management Service with the mission to provide a set of capabilities for provisioning, scaling and managing a Container Orchestration Engine (COE). On the contrary, Zun provides container-specific APIs across different container technologies. So, this is an interesting idea though it is all very confusing. Moreover, Zun basically represents the evolution of the older Nova-Docker API.

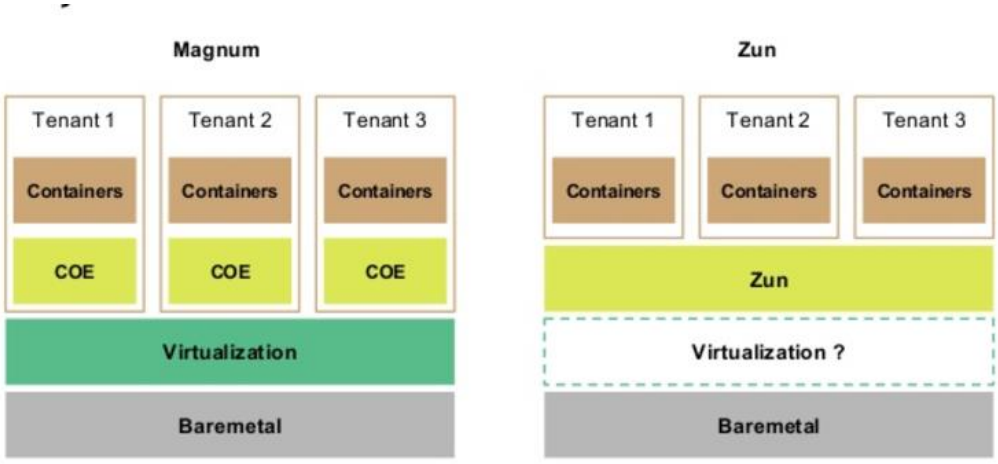

Figure 62 - Magnum Vs Zun

Figure 62 shows us the architecture differences between Magnum and Zun. As it is possible to notice, Zun is not tight-coupled with the underlying virtualization infrastructure and so it can make use of different container runtime.

However, this is not the unique OpenStack container-centric solution. In fact, another container-focused OpenStack project is Nova-LXD. This makes use of LXD and represents a properly integrated solution with the whole OpenStack architecture.

5.5.3 Zun Concepts

Zun needs to manage containers as well as their associated underlying infrastructure-related resources, such as network addresses, security groups, ports, volumes, and more. However, the adopted principle was to decouple the management of containers from their associated resources. To do that, they introduced the concept of “sandbox”.

A sandbox is an isolated environment for one or multiple containers. Its responsibility is to provision and manage infrastructure resources associated with a container or a group of containers. By this way, each container must have a sandbox, and resources (such as Neutron ports) are attached to sandboxes (instead of directly being attached to containers).

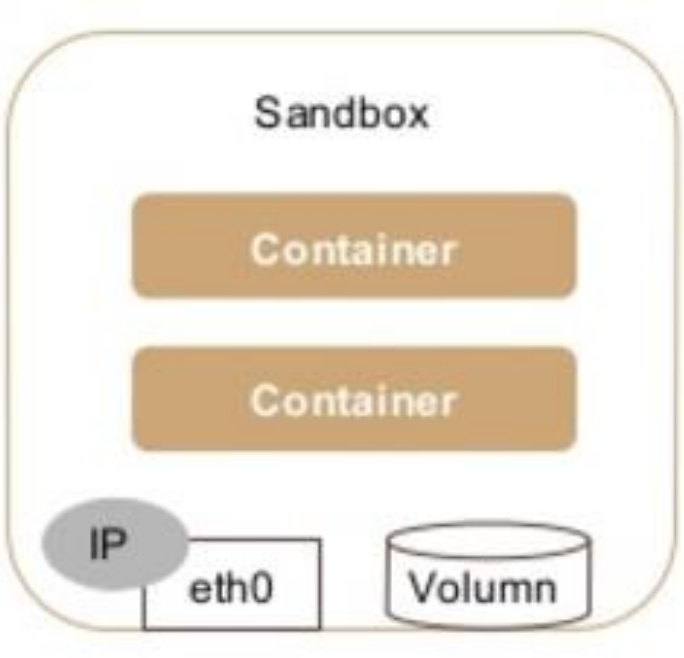

Figure 63 - Zun SandBox

Figure 63 shows us the architectural model of a Zun sandbox. In fact, it is defined a Sandbox interface and the implementation is driver-dependent. So, each driver needs to implement the sandbox interface. For docker driver, the sandbox can be implemented by using docker container itself. Furthermore, the design is extensible so that operators can plug-in their own drivers if they are not satisfied by the built-in sandbox implementations.

5.6 OpenStack Kolla

Kolla is a project designed to make easier the installation and update of the whole OpenStack. Releases of OpenStack foundation are published according to a specific timespan. This introduces a strong flexibility but also much complexities to deployment operations. Kolla is a new way to configure OpenStack inside containers, taking advantages of rapidly obtaining a reliable and composable installation.

In fact, Kolla simplifies the configuration of each service, that is seen as a micro-service installed through a Docker container. Therefore, in order to update a service, it is needed to build a new Docker container by using containerized micro-services and orchestration tool such as Ansible. This mechanism allows us to install OpenStack without no need to properly configure other system components. This guarantees the immutability of the entire project because the unique part which needs to change is the configuration module to load inside

containers. So, it represents a declarative-based system through which is possible to deploy a cloud environment with no need to spend time in configuration issues.

Kolla is implemented according to the so-called “data container” model. Data containers are a separate technology from virtualization, though they are based on some of the same theories. With virtualization, an entire machine is replicated. By contrast, a data container shares the underlying host kernel by storing only applicative data. In addition, there is no need for data containers to be provided with a virtual memory, meaning they consume less processing power when running. So, a data container can be mounted on the underlying operating system and every stateful component, such as a database, virtual machines and so on, can be included in data containers. Each of which can be individually used without no problem for eventually backup and recovery processes.

Kolla is considered as a deployment system which interacts with the configuration of four main parts. The leader distribution is CentOS, even if containers are also available for Fedora, Oracle Linux, Red Hat Enterprise Linux and Ubuntu. Currently, the project is on experimental phase and still not considered for production-ready.

5.6.1 Architecture

Kolla makes easier the work of operators who, even with minimal experience, can quickly deploy OpenStack. In fact, the object is to replace the inflexible, painful, resource-intensive deployment process of OpenStack with a flexible and inexpensive deployment process. Finding people experienced in OpenStack deployment is very difficult and expensive and so Kolla seeks to remedy this set of problems by simplifying the deployment process by enabling a flexible deployment model.

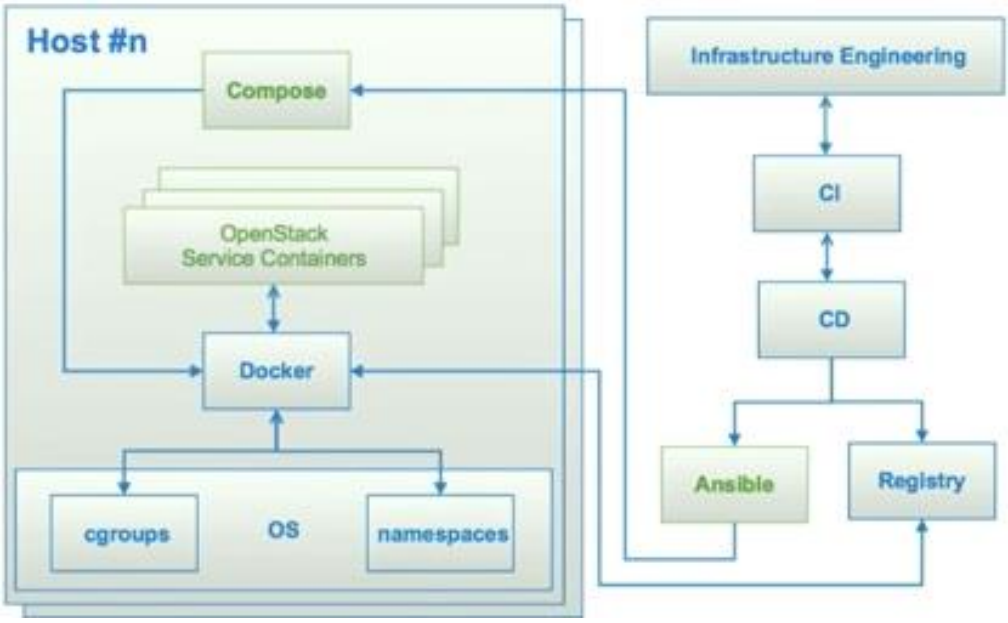

Figure 64 - Kolla Architecture

Figure 64 shows us the architecture point of view of a Kolla deployment. It makes use of Docker containers while it exploits other systems to perform other types of operations like orchestration and configuration. In this case, Ansible is used to deploy OpenStack on bare metal or virtual machines, but it is also possible to use Kubernetes templates to deploy OpenStack on a whole Kubernetes cluster. In fact, the mission of Kolla is to provide production-ready containers for the whole OpenStack deployment and management process.

5.6.2 Benefits of using containerized deployment

Usual the system configuration consists of making use of package-based components through which an entire platform can be deployed. However, this is now getting replaced with an image-based management due to the introduction of the so-called software-container paradigm. This is helping to solve the availability, management and scalability aspects of deployment systems. In fact, considering operations are atomic, there is minimal interruption of service and it is even possible to perform full rollback actions [73].

Using containers provides to operators several benefits taking advantages of the guaranteed isolation and performance feature. Working with the server-based environment is not as easy as with containers. In fact, what is enough is to start and stop docker containers on server-nodes, which can be scaled up and down as compute nodes to OpenStack. Surely, the new way is able to run on any platform, regardless of the physical host operating system. However, the unique requirement is to support the container technology.

By the way, upgrading and patching operations are atomic, meaning that they will either successfully complete or fail. Due to this new configuration approach, deployment of OpenStack takes on an average about ten minutes, much faster than any other deployment tools, such as Puppet or Salt. Furthermore, there is no need to rolling-updates. In fact, when a new container-image is available, it is possible to simply stop the old-container and start the new one with the latest image. Moreover, in case of problems, it is possible to fall back to the old image. This makes everything containerized and so managing services consists of starting and stopping the related containers.

As we have already discussed, in order to manage those containers, many orchestration tools can be used, like Kubernetes or Docker Swarm, and so failed containers can be automatically restarted. This results in a self-healing deployment and mages, once built, do not change over time. Hence, it is possible to recreate the same setup on different environments with the exact same piece of code running that is running on the other one. This is very important because allows us to easily move everything from a development to the production environment with ease and no difference.

5.6.3 Deployment

A Kolla Deployment [73] is made of a node which is responsible to manage the whole provisioning. This is called provisioning node. It makes use of a private Docker registry and so the correspondent OpenStack nodes are created as Docker containers.

The major part of OpenStack services is deployed on specific hosts which are responsible to take care of the whole management of the cluster. These are called Controller nodes and can hold every service except the nova-compute. In fact, this is the component responsible to interact with the underlying compute hypervisor. For this reason, it is deployed on compute nodes where the instances will be created. Furthermore, these nodes contain also the agent of openvswitch that is the component responsible to enable the communication by guaranteeing the multi-tenancy OpenStack feature.

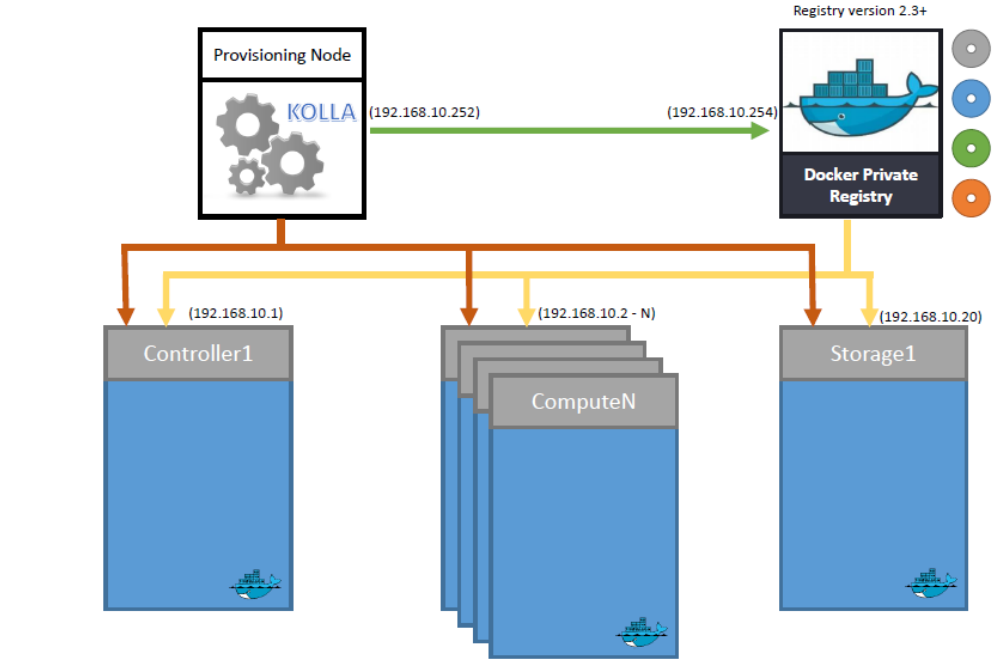

Figure 65 - Kolla Deployment

Figure 65 shows us a typical deployment of OpenStack using Kolla. As it is possible to see, the provisioning node is responsible for building images. To do that, the private registry holds docker images that are used to deploy the whole OpenStack cluster. For this reason, the private Docker registry acts as a central store from where every node can download the requested Docker-image.

5.6.4 Network

In a usual OpenStack deployment, Neutron manages networking by creating bridges, namespaces, ports and tying them together. Kolla is based on the same networking model [73].

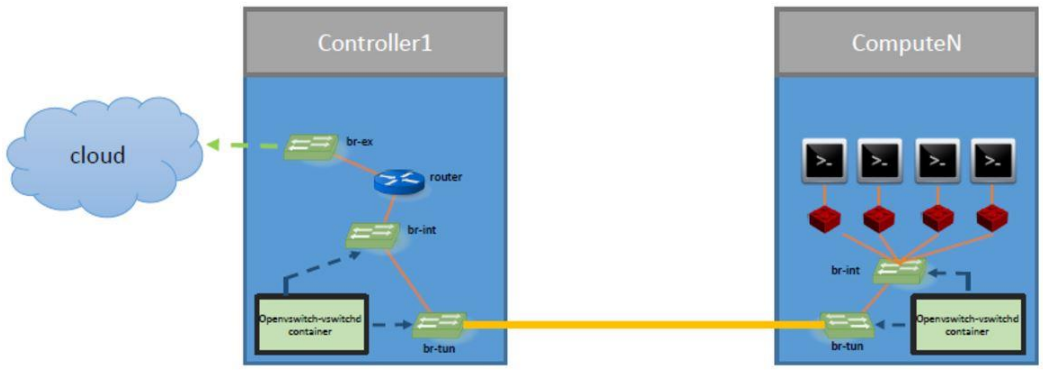

Figure 66 - Kolla Network

Figure 66 shows us a network overview of an OpenStack deployment using the project Kolla. Furthermore, to completely support all neutron models, it is

necessary that bridges and namespaces created, which are created by some containers, are even visible on the underlying host network stack. Therefore, the docker engine is started with a networking host type which, as seen in the correspondent section, provides to containers the possibility of sharing the same network namespace with the underlying kernel host.

5.7 Murano

Murano is an OpenStack project which aims to provide an application catalog, in order to be useful to both developers and operators. These applications are categorized on a repository accessible through the OpenStack dashboard. Administrators use public repositories to obtain additional services, such as OpenStack Community App Catalog, Google Container Repository, and Docker Hub. Furthermore, this project provides to users the possibility to have the full control of applications lifecycle. This is the same principle which has been adopted in other complementary solutions such as Rancher. In fact, users can quickly install reliable applications, simply by pressing a click-button.

Murano enables the provisioning of typical applications with container-based environments and PaaS solutions, including Kubernetes, Apache Mesos, Cloud Foundry and Docker Swarm atop of OpenStack. Furthermore, it coordinates the usage of all Docker drivers inside the context of an application. To do that, as seen with Magnum, it makes use of existing solutions like Heat orchestration system and additionally python plugins.

Murano provides a high-level service in order to make easier operations, such as upgrade, scale up/down, backup and recovery processes. The purpose is to use the abstraction of service-management to compose and configure environments through a web-based interface or REST APIs. The execution environments are simply virtual machines or multi-tier applications with the enhancement of auto-scaling and self-healing. Therefore, users, who are not able to interact with IT-specific operators, can easily carry out their own work by simply deploying the correspondent applications packages. This allows people to focus only on business parts and organization requirements. In fact, Murano is designed to solve the problem of integrating third-party components in OpenStack. Nevertheless, this is quite important because enables the provisioning of a service model that can be rapidly exchanged between Infrastructure and Platform as a Service.

5.7.1 Architecture

As seen in Magnum, the project makes use other existing OpenStack services to exploit their target capabilities. These are the orchestration system and the identity service. Moreover, the interaction between Murano and those services is performed by using REST invocations.

Heat is used to orchestrate infrastructural resources, such as servers, volumes, and networks. Based on the application definitions, Murano creates heat templates and performs the correspondent invocations on the orchestration client. On the other hand, Keystone is used to make Murano APIs available to all OpenStack users and so the project has to integrate its own functionalities with the OpenStack principle that is properly based on multi-tenancy.

This constitutes the principle of the platform which aims to make easier the work of developers and cloud administrators. In fact, developers want to simply use applications as opposed to installing and managing them. On the contrary, cloud administrators focus on offering a well-tested set of on-demand self-service applications. For this reason, Murano is considered as the project that solves this problem for both constituents.

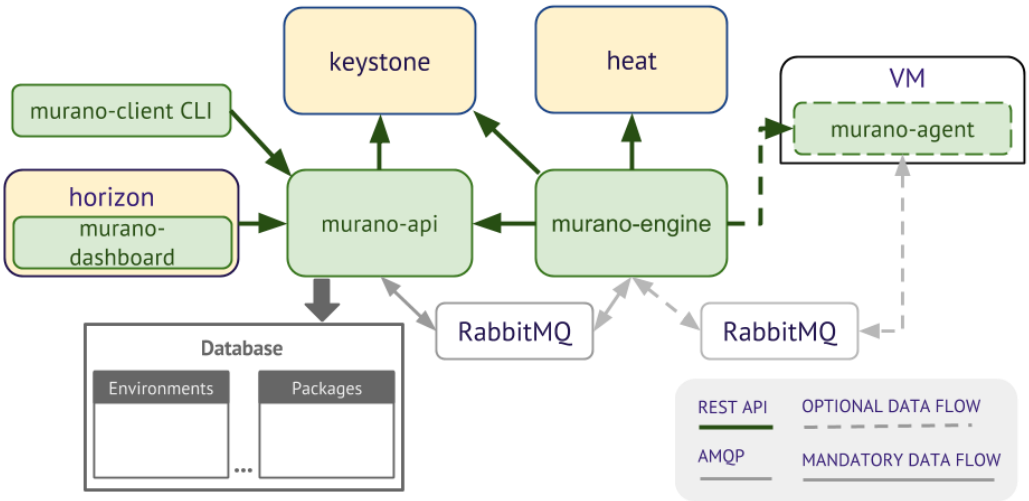

Figure 67 - Murano Architecture

Figure 67 illustrates the architectural point of view [74] of Murano. As it is possible to see, all operations are carried out through an Advanced-Message-Queuing-Protocol (AMQP) queue that in this case is RabbitMQ. This component mediates the communication between the API server, the Murano-Engine, and the Murano-agent. The API server is responsible to process external client requests. Murano-engine is the component that takes care of the core functionalities, that

are implemented in the controller node, whereas the Murano-agent is the process that is deployed in each compute instances. Furthermore, the architecture consists of additional components like a command-line-interface, to perform natively client requests, and a horizon plug-in, that is used to integrate this important catalog in a web-based interface like Horizon.

5.7.2 Network

Murano is able to work in various networking environments. The system consists of a process which is able to detect the deployment network configuration and automatically chooses the appropriate settings. However, additional actions are required to support some advanced scenarios [75].

The simplest solution is to use Nova-network. It has limited capabilities but is available on any OpenStack deployment with no need to install additional components. When a new Murano environment is created, it checks if a dedicated networking service exists in the current deployment. If such a service is not included all the compute instances, that are spawned by Murano, will be joining the same network.

On the contrary, Murano enables the advanced networking features that are taken out of user responsibilities. Furthermore, by default, each environment is configured with an isolated network. Therefore, it is necessary to put a gateway in order to expose the applications in the spawned virtual machines.

5.7.3 Advantages to using Murano

Murano by itself is not a container environment, but it is an application catalog [76] that makes use of Kubernetes for deploying application containers. Furthermore, this guarantees the advantage not to manage Kubernetes even if the underlying infrastructure makes use of that. In fact, users may be unaware that applications are running on containers.

By this way, the internal provisioning is handled by Murano and users focus just on the business aspects through an easy-to-use User Interface. Therefore, Murano is quite accepted by developers, who are writing applications for other people in order to exploit a self-service manner in cloud deployments. Furthermore, Murano and Magnum communities are getting together with a plan to create a Magnum application for Murano. This allows users to combine advantages of both solutions, including access to a generic container orchestrator beyond Kubernetes.

5.8 Closing remarks

OpenStack is the leading cloud framework for adopting and adapting new technologies. For this reason, the community decided that containers were an important technology to support and that decision has resulted in several projects to ensure containers, and the third-party ecosystem around containers are supported in OpenStack clouds.

Moreover, as described in this chapter, the focus on containerization is based on different requirements that containers are suitable to deal with. OpenStack Compute Nova manages the compute resources for an OpenStack cloud. Nowadays, due to the support of containerization in OpenStack, we are able to spawn compute resources that are running as containers. To do that, we need to use a specific driver in order to allow the interaction between Nova service and the underlying container runtime. Initially, Docker was integrated with a project called Nova-Docker. Nowadays, this is no longer maintained and so, OpenStack Zun is more suitable for use cases with the requirement to treat a container like a lightweight virtual machine, allowing use in similar way to on-demand virtual machines.

Furthermore, containers are excellent for encapsulation of microservices. In build/continuous integration environments, containers enable organizations to rapidly test more system permutations as well as deliver increased parallelism, increasing innovation and feature velocity. In this case, there is the need to work at a higher level with functionalities like orchestration. OpenStack has even promoted this integration with the introduction of OpenStack Magnum. This is designed to offer container specific APIs for multi-tenant containers-as-a-service by integrating the cloud platform with well-established container orchestrators such as Kubernetes, Docker Swarm, and Apache Mesos. In addition, there is the support for packaged applications to be deployed on OpenStack through Murano. This is a catalog solution that allows users to quickly deploy and configure enterprise applications by using an existing container orchestrator.

In conclusion, we have seen that containers are supported in OpenStack also for the deployment point of view. To do that, OpenStack Kolla takes advantages of containerization to make easier the deployment process of a complex architecture by running software components as containers.

6 Experimental Results

6.1 Overview

Today, the cloud is a proven delivery model, with a growing number of enterprises realizing impressive agility and efficiency benefits. However, as the technology matures, the trend is to extend cloud deployments to even more flexible solutions that promise exciting new ways to expand the value of enterprise services [77]. So, the purpose is always focused on what organizations need to do in order to get the most value. In fact, the value proposition of cloud computing is quite stable but, of course, users need to choose the best way to simplify the delivery of their own cloud services.

Cloud computing provides a variety of services with the growth of their offerings. However, this implies numerous challenges to be faced. As seen before, cloud computing is based on virtualization, which provides users a plenty of computing resources without managing any component of the underlying virtualization middleware. Nevertheless, sometimes the abstraction level involved in virtualization has been reducing the workload performances [78] that are quite fundamental in a cloud infrastructure.

For this reason, new solutions have been proposed to provide benefits where the classical virtual machine model ends up. An example is the fundamental concept of Containerization, which has been deeply introduced in the previous chapters. The key differentiator is that containerization provides server instances by sharing a single kernel whereas, in a virtualized server, each virtual “guest” includes a complete operating system with drivers, binaries, and the same application. Various related works analyze regarding performances the behavior of virtual machines and containers. However, notwithstanding the existing solutions, just with Docker this paradigm has obtained a significative adoption rate. Moreover, as mentioned in the previous chapter, Docker cannot be considered a full complementary solution of virtual machines. In fact, it aims to application containers and not system containers.

Considering the area of system containers, we analyzed the newer proposal of Canonical, LXD, which is thought to provide what Docker did not. Furthermore, it is quite integrated into cloud operating systems like OpenStack, and so, in this chapter, one of the purposes is to propose a performance analysis between two OpenStack deployments, each of which provides compute instances through a

different virtualization technology. This investigation will be extended to also analyze the behavior of a quite adopted solution, like Docker, even if this study cannot be performed with an OpenStack deployment, considering that the solution does not belong to the category of system containers.

For this reason, subsequently, we will discuss solutions that focus on the applicative point of view: container orchestrators. As discussed in the correspondent chapter, they are fundamental to provide high-level services that are designed to deal with application-deployment and cluster-management. Thereby, they are also suitable to be adopted in cloud scenarios where the user requirements are focused on higher services.

6.2 Requirements

Cloud computing services provide resources using virtualization and Containerization. Virtualization is a crucial part of the Cloud computing definition, and so we need to manage virtualized resources more efficiently. Furthermore, the Cloud Infrastructure [78] provides the abstraction to make sure that application, or the business service model, is completely independent of the underlying hardware such as servers, storage or networks. In fact, as seen before, the cloud model depends on virtualization technology by which a bare-metal server is used to spawn virtual compute instances. These can be virtual machines or containers, and therefore we need to investigate the concept of reducing the number of resources wasted during the computational process.

Until now, we discussed cloud computing features and how they are implemented with virtual machines and containers. Surely, everything is strongly dependent on infrastructure resources like CPU, Memory, Disk space, Input/Output, and so on. These are shared among multiple users and so an efficient management is a key differentiator to success. Therefore, we need to perform a comparison between containers and virtual machines regarding resources overload. In particular, we need to point out the so-called “density”. This means how these resources are influenced by increasing the number of compute instances per server host. Moreover, a fundamental cloud feature is the “elasticity”, that is the possibility to complete a service request by introducing the shortest possible time. Therefore, it is necessary that we also investigate the time needed to create virtual machine-based and container instances.

This work is not just focused on the infrastructure point of view and so, another important purpose is to investigate the applicative perspective of the

containerization paradigm. An example concerns the orchestration level that is referred to the process of managing any kind of infrastructure. This is the case to provision a distributed enterprise application that, by definition, requires flexibility and high-availability. For this reason, the goal of this analysis is to evaluate the behavior of these solutions in terms of service provisioning and Mean Time to Repair (MTTR).

6.3 Test Plan

6.3.1 Virtualization and Containerization

Containers enable users to pack more applications into a single physical server, because they share the underlying kernel. Virtual machines do not run just a full copy of an operating system, but a virtual copy of all the hardware that the operating system needs to run. Surely, this quickly adds up to a lot of RAM and CPU cycles. In contrast, all that a container requires is an operating system, supporting programs, libraries, and system resources to run a specific program.

In order to perform a comparison between system containers and virtual machines, we chose to deploy two OpenStack implementations: one that makes use of KVM as hypervisor and the other one that exploits LXD to provide compute instances as system containers. Furthermore, to evaluate the behavior of Docker, we chose to build a cluster infrastructure that makes use of that container solution as compute hypervisor.

The analysis consists of monitoring the performances of the following resources: CPU, Network, and Disk I/O. In order to reduce the error probability, each experiment is repeated ten times and the average value is recorded. The aim is to take out what is better and particularly when is more appropriate to adopt a container-based deployment instead of the traditional virtual machine model.

CPU Analysis

The CPU analysis is split into two benchmarks: CPU-Power and CPU-Contention. The first is meant to get the whole execution time of a single process that exploits the number of existing cores through the concept of Multi-Threading. On the contrary, the CPU-Contention test consists of analyzing the behavior of the execution time by putting together multiple compute instances that compete to access the same resources. For the CPU-Power test, we need some compute instances, each of which with a different number of vCPUs. In OpenStack, a flavor defines the resources of a compute instance such as memory, number of

cores and storage. Therefore, in our test, four different OpenStack flavors are created with a variable number of vCPUs (1, 2, 4, and 6). The dual representation in Docker is obtained by exploiting the resource limit definition of a Docker container.

Network Analysis

Network performance refers to measure the network service quality as seen by the customer. This test includes the analysis of bandwidth, latency, and throughput. Bandwidth is the maximum rate that information can be transferred. As mentioned before, the time for a request is a fundamental aspect that needs to be considered in cloud deployments. For the throughput and bandwidth analysis, the purpose is to analyze the behavior of these key performance indicators by exploiting UDP and TCP communications. To do that, we decided to use "iperf". Furthermore, just for UDP messages, this gives us the possibility to set a variable target bandwidth in order to observe the network behavior at different rates. Lastly, the latency cannot be avoided, and so this test includes even an analysis of how the performances change between KVM, Docker, and LXD instances.

There are several ways to measure the performances of a network, because every solution is different in nature and design. For this reason, we chose to split this test into three parts, each of which refers to a different deployment scenario. The first concerns the communication between a cloud instance and another one which runs outside the cluster. The second one is between two pairs that run on different hosts of the same cluster infrastructure, whereas the last one is between two pairs that execute on the same physical server.

Input/Output Analysis

A virtual machine replicates an entire server, including the operating system and the associated drivers. On the contrary, a container makes use of the underlying kernel host to access external resources, such as I/O devices and networks. Therefore, an important analysis is the file system that is present within each compute instance. There are a lot of designed benchmarks, and in our test, we decided to choose "Bonnie++". To do that, we need to stress the input/output system that operates by writing and reading from and to big chunks of data. Furthermore, it is important for the Input/Output benchmark to limit RAM involvement. In fact, a big amount of memory would mean that caching would be a predominant factor and therefore would affect the real results. However, there

is a way to bypass this problem, and so we need to use files and data which are larger than the amount of system memory. So, for these experiments, the compute machines are set to 512MB of RAM and the files that they are working with have a size of 24GB (forty-eight times the size of the RAM).

Density Analysis

Containers are one of the hottest topics in the IT world today, largely due to their adoption by many web-scale companies like Facebook and Twitter. By the way, a key differentiator aspect is the density that is the number of compute instances that a server host is able to execute, without degrading performances. Containers also enable better workload density within an infrastructure, considering that they require less memory overhead per instance. In fact, each application is loaded into a kernel host that is shared across all containers. As already explained, operating systems and kernel "guests" do not need to be loaded per container and so, more applications and workloads can be squeezed into the same hardware or infrastructure footprint. For this reason, the purpose of this benchmark is to analyze the behavior of CPU and Memory usage, by increasing the number of compute instances per host. To do that, we decided to monitor the same physical server in order to get the behavior of CPU-Load and memory usage.

System Analysis

A company, offering cloud-computing services, accomplishes any customer requests of the computing resource, with the purpose to satisfy two competitive needs: to provide the proper hardware to fulfill any request and the necessity not to imply a larger time to complete the service request. Therefore, it is necessary that this performance analysis is extended to evaluate other system aspects that are involved in the provisioning of a service request. These are the time needed to provision a whole instance, the time to boot up a single machine and the whole time that is required to complete a snapshot of a compute instance. Furthermore, we also consider, as system aspect, the size impact of an operating system image that is needed to create the compute instances. Therefore, the purpose of this test is to compare compute instances, according to these specific aspects, each of which is taken by a different hypervisor implementation.

6.3.2 Container Orchestration

All container orchestrators do the same thing: automate the provisioning and management of containerized infrastructure. It is worth noting that orchestrators

are not strictly limited to the container world. These are solutions that exist for other types of infrastructure with the main purpose to allow developers and DevOps people to forget about the detail of what needs to happen. Similarly, containers allow us to standardize the environment and abstract away the specifics of the underlying operating system and hardware. Thereby, a container orchestrator does the same job for a whole data center: it allows us the freedom not to think about what server will host a particular container or how that container will be started, monitored and killed.

As seen in the correspondent chapter, a container orchestrator is composed of three modules: resource management, scheduling, and service management. Nevertheless, the core is the scheduling part of the solution because this is the crucial responsibility by which a user relies on. In fact, this allows developers to focus on business aspects regardless of the distributed nature of the enterprise applications. For this reason, in this test, we want to investigate the behavior of these solutions in terms of time measurements to provide the following functionalities: service provisioning, and failover.

Service Provisioning

In cloud computing, the elasticity is defined as the degree to which system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at every time the available resources match the current demand as closely as possible. Therefore, an important aspect is to analyze the time needed to provision an application. To do that, we decided to split this test in two scenarios: one that requires the deployment of a simple web application and the other one that exploits enterprise applications of different complexities. Moreover, the purpose is to determine the behavior of the provisioning time by incrementally stressing the workload of the container orchestrator. Thereby, we chose to study the deployment of the first web application by increasing the number of container replicas, whereas in the other one we analyze the performances by increasing the number of microservices that compose a single application.

Failover

The failover system mechanism is used to increase the reliability and availability of IT resources by using clustering technology to provide redundant implementations. A failover system is configured to automatically switch over to

a redundant resource instance whenever the currently active IT resources become unavailable. These are commonly used for mission-critical programs, and this mechanism may rely on the resource replication mechanism to supply the redundant IT resource instances, which are actively monitored for the detection of errors and unavailability conditions. For this reason, as will be illustrated in the system description, the architecture is cluster-based by which is possible to schedule again a service that unexpectedly was stopped. A failure can involve the process or the entire server host. Therefore, in this analysis we will investigate, in terms of time measurements, both types of failures.

6.4 Deployment Tools

6.4.1 Infrastructure as Code: why do we need it?

The proliferation of virtualization coupled with the increasing number power of industry-standard servers, and the availability of cloud computing, has led to a significant uptick in the number of servers that need to be managed within an organization [79]. Therefore, data center orchestration and configuration management tools come into play. In many cases, we manage groups of identical servers, running the same applications and services. Often, these are deployed on virtualization frameworks within the company, but increasingly they are running as cloud or hosted instances in remote data centers. To deal with those, different solutions were built with a single goal in mind: to configure and maintain several severs much easier. This offers benefits to clusters of different sizes because the idea is to have a model that is architecture-agnostic in order to facilitate the provisioning and deployment of a whole enterprise infrastructure. For this reason, we decided to embrace this pattern and so, we now introduce the solutions used to deploy our systems.

6.4.2 Maas

Metal-As-A-Service (MaaS) is hardware provisioning software from Canonical intended to quickly commission and deploy physical servers to run a wide array of software services or workloads via Juju charms [80]. By this way, servers can be dynamically associated or connected together to scale up services, and can also be disconnected to scale down as demand requires it. Furthermore, MAAS treats physical servers as compute commodities that can be quickly manipulated to meet customer demand, similar to how a cloud environment creates and removes virtual resources to adjust to computing demands.

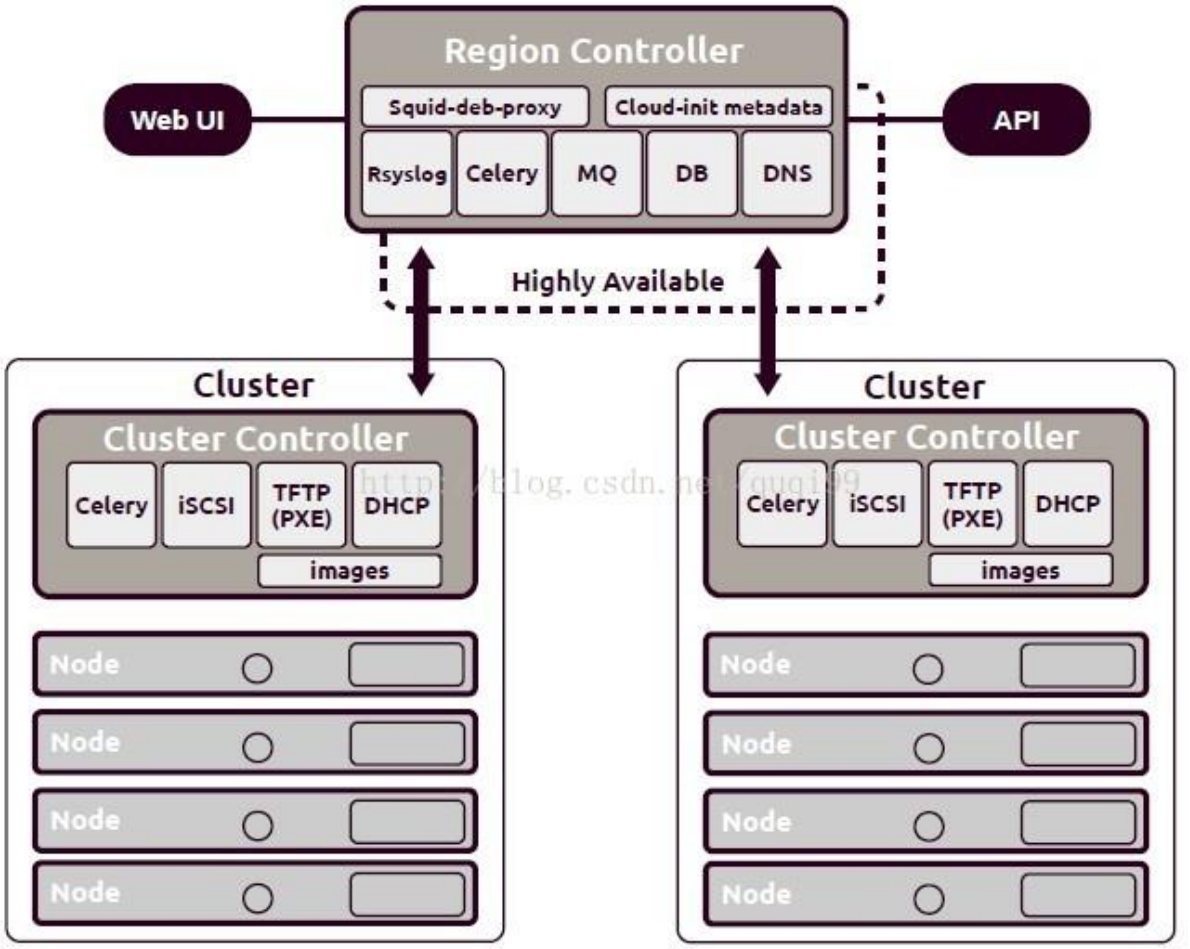

Figure 68 - An example of MAAS deployment

Figure 68 shows us a deployment with a Region Controller and two Cluster Controllers that are the two types of controllers involved in MAAS. A region controller is responsible for a data center, or a single region. On the contrary, the cluster controller (or Rack controller) is the responsible for each server node in a single data center. Furthermore, both the region and the cluster controller can be scaled-out as well as made highly available.

Basically, MAAS turns our bare metal data center into an elastic cloud-like resource. By this way, machines can be quickly provisioned and then destroyed again as easily as we can with instances in other clouds like Amazon AWS, OpenStack, and more. Also, it can act as a standalone Preboot Execution Environment (PXE), or it can be integrated with other technologies. In particular, it is designed to work well with Juju, the service, and model management orchestration. This makes a perfect arrangement: MAAS manages the machines and Juju take care of services running on those machines. So, the next section discusses this orchestration tool, because the implemented solution for the

performance analysis between virtualization and containerization is properly based on this deployment pattern.

6.4.3 Juju

In modern environments, applications are rarely deployed in isolation. In fact, even simple applications may require several other services in order to function well. For modeling a more complex system, like OpenStack, many more service components need to be installed, configured and connected to each other. Juju [81] is an orchestration service that provides tools to express the intent of how to deploy some applications and to subsequently scale and manage them.

Usually, IT operators make use of traditional configuration management tools like Chef [79] and Puppet [82], or even general scripting languages as Python or bash, to automate the configuration of machines to a particular specification. Juju works at a higher level providing the possibility to create a model of the relationships between components that together constitute an entire complex system. By this way, application-specific knowledge such as dependencies, scale-out practices, operational events like backups and updates, and integration options with other pieces of software are encapsulated in the so-called "Juju charm"

A charm is a definition of everything is needed for a specific component that can be integrated into a solution. It is possible to use pre-existing pieces, or otherwise write them, to deploy a service component in seconds, on any cloud instance or bare-metal server. Furthermore, it is also possible to integrate Puppet, Chef, and others with Juju. In fact, it works a layer above by focusing on the service the application delivers, regardless of which it runs. So, the Juju charm includes all the logic for writing configuration files for an application that can be written in whatever language or tool the author prefers.

6.5 System Description

6.5.1 System specification

At its most fundamental, OpenStack is a common API abstraction layer for infrastructure. This means that OpenStack is essentially a way of enabling developers to address datacenter infrastructure though a standard set of instructions, regardless of what that actual infrastructure is. This is quite useful because there is no need to perform custom integrations for every type of hardware. Furthermore, this allows us to swap out components with less need to worry about compatibility issues. The same is about Rancher that, as described in

the correspondent section, does not include any infrastructure requirement but just Linux systems able to execute Docker containers.

In order to set up two different OpenStack clusters, we made use of two groups of servers, moreover quite similar in order not to affect the result of our test. The physical servers are eight MicroServer Hewlett Packard Enterprise (HPE) Proliant Gen 8. They include 2x Intel (R) Celeron(R) CPU G16610T @ 2.30 GHz processors with 12 GB di RAM. The unique difference is that the cluster used for KVM and Rancher is built with two hard drives of 750GB whereas the other one with two hard drives of 500 GB. Furthermore, both devices work with 7200 rpm.

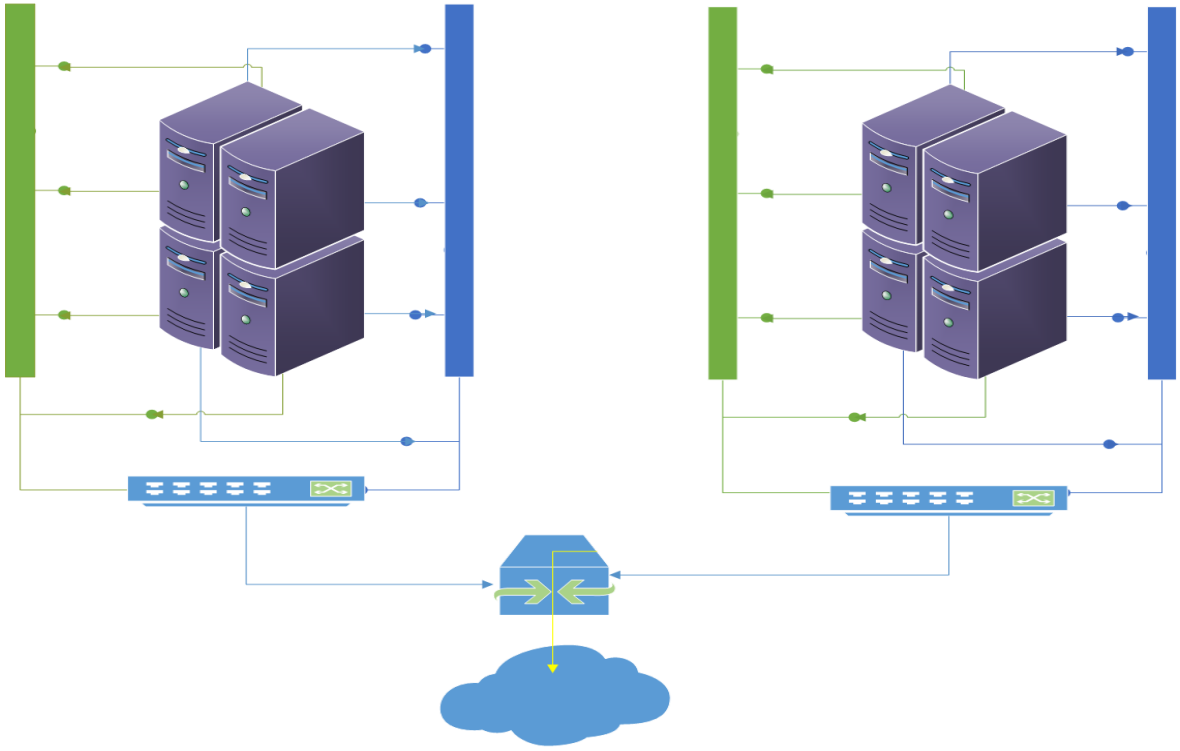

Figure 69 – Layout of the physical system

Figure 69 shows us the layout of the physical system that has been used for the test. As it is possible to see, the machines of each cluster are connected with the aid of two Network-Interface-Cards (NICs) to a 1000 Mbit/s network. In addition, we used two GS724TV4 ProSafe 24-port Gigabit Ethernet Smart Switch. Both switches are connected to a single gateway which represents the front-end of each cluster infrastructure.

6.5.2 Virtualization and Containerization

This section aims to describe the implemented solution in order to achieve the deployment of two cloud models: one by using KVM as hypervisor and the other one with LXD, as container management engine. However, the two clouds are

completely independent and running on different server hosts. Deploying and upgrading a basic OpenStack environment has always been a complex task. Containerized micro-services and orchestration tools, now allow operators to upgrade a service by building a new container and redeploying the entire system. In addition, this allows us to take advantages by supporting different versions and package mechanisms. For this reason, the implemented solution followed this principle with the usage of MAAS and Juju. By this way, the environment is completely immutable, because the only things that change are the configuration files loaded into a container and how those changes modify the behavior of the OpenStack services. So, the purpose of the following sections is to describe how we achieved the deployed solution.

OpenStack Deployment

As seen in the previous chapter, OpenStack is a cloud-operating system with a lot of service components that interact each other to perform the implementation of several types of service requests. Furthermore, even with the support of automated tools like Juju, this solution involves a lot of charms that need to be deployed with the correspondent relationships. In fact, the principle of each "Juju charm" follows the idea of each OpenStack service: completely decoupled and asynchronous communication. Therefore, a single OpenStack deployment needs the usage of a lot of charms, each of which is properly configured to interact with those that represent a single OpenStack service. However, the sponsoring company of this new design model has faced this issue with the introduction of the so-called concept of "Juju bundle". This is a set of Juju charms, properly configured, to work together in order to deploy a single OpenStack cloud environment quickly.

The purpose of this work is to perform a performance analysis of two cloud deployments: one that makes use of virtual machines and the other one that exploits LXD containers. So, we thought to divide the implementation into two different solutions: a single cloud with the compute service (Nova) organized to interact with KVM as a hypervisor, and the other one that provides Nova instances through LXD. To do that, we made use of two different Juju bundles: one for virtual machines [83] and the other properly built to support Nova-LXD [84].

Furthermore, these bundles are designed to run on bare metal using Juju with MAAS, and so we need to set up a MAAS deployment with a minimum number of physical servers before using this bundle.

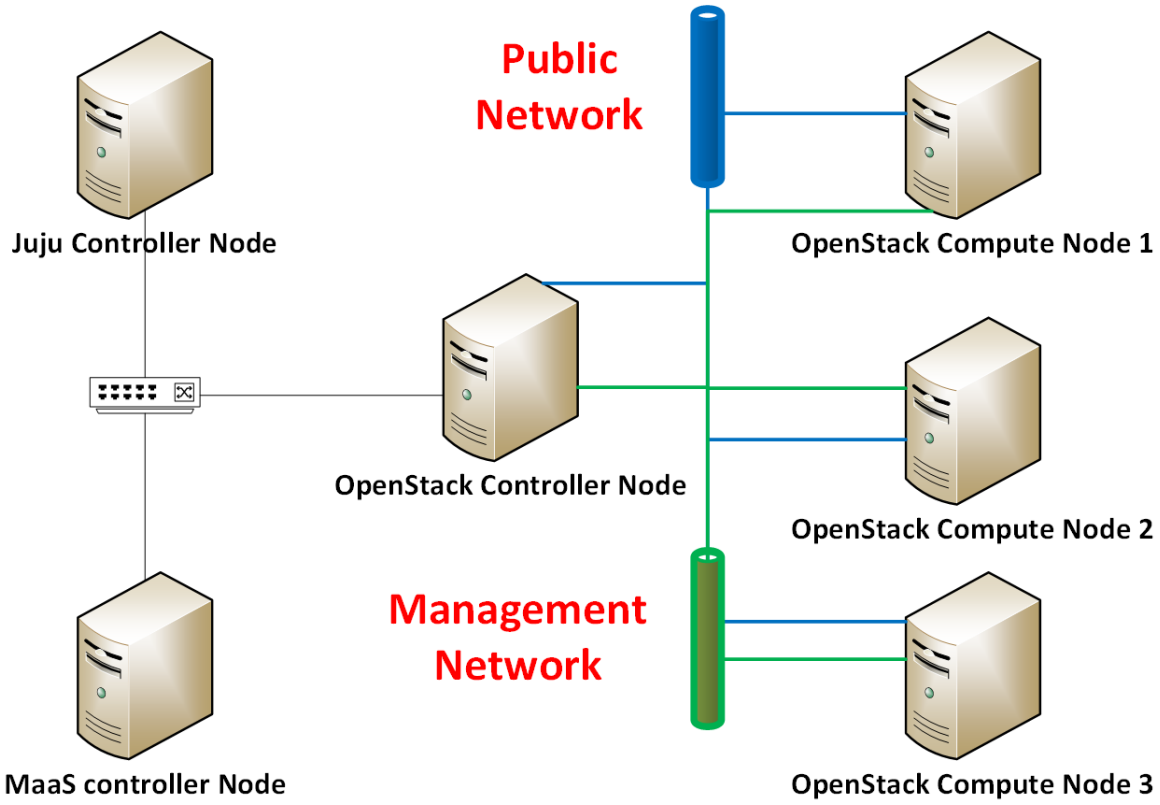

Figure 70 –A layout of the deployed OpenStack cloud

Figure 70 illustrates the layout of the deployed OpenStack. Furthermore, this is the same for both clouds: hypervisor virtual machine and container-based. As it is possible to notice, each OpenStack consists of two types of nodes: Controller and Compute. Controller is responsible for hosting the management part of the entire architecture whereas the second one is responsible for hosting and executing the compute instances. Therefore, the key difference between the two clouds is that the compute nodes make use of a different Nova libvirt implementation. In fact, the first one has a driver that provides compute instances by interacting with the underlying KVM hypervisor, while the second one does the same but by making use of LXD as underlying hypervisor.

6.5.3 Container Orchestration

As discussed in the previous chapter, Rancher has grown very quickly and now includes support for multiple orchestration frameworks in addition to Cattle. The support for these different orchestration platforms is delivered by creating isolated "environments". Each of which is composed of different hosts, which are just Linux physical or virtual machines that run Docker and the Rancher agent. In particular, the Rancher agent is simply a Docker container. This allows users to quickly deploy and test different orchestration solutions by exploiting the power of Rancher platform. In fact, from a user perspective, it is not any more complex

to deploy the different platforms, as Rancher automates all the deployment and configuration of the orchestration platforms.

It is commonly believed that a container orchestrator is a key to successfully operationalizing containers at scale. This is true if we are running a single cluster in the cloud or with reasonably homogenous infrastructure. However, many organizations have a diverse application portfolio and user requirements and therefore have more expansive and diverse needs. In these situations, setting up and configuring a cluster like Kubernetes gives rise to several challenges. An example concerns the customization to the DevOps team or to automate the upgrade of the whole Kubernetes cluster. For this reason, we decided to build a Rancher cluster to evaluate the behavior of each orchestration platform.

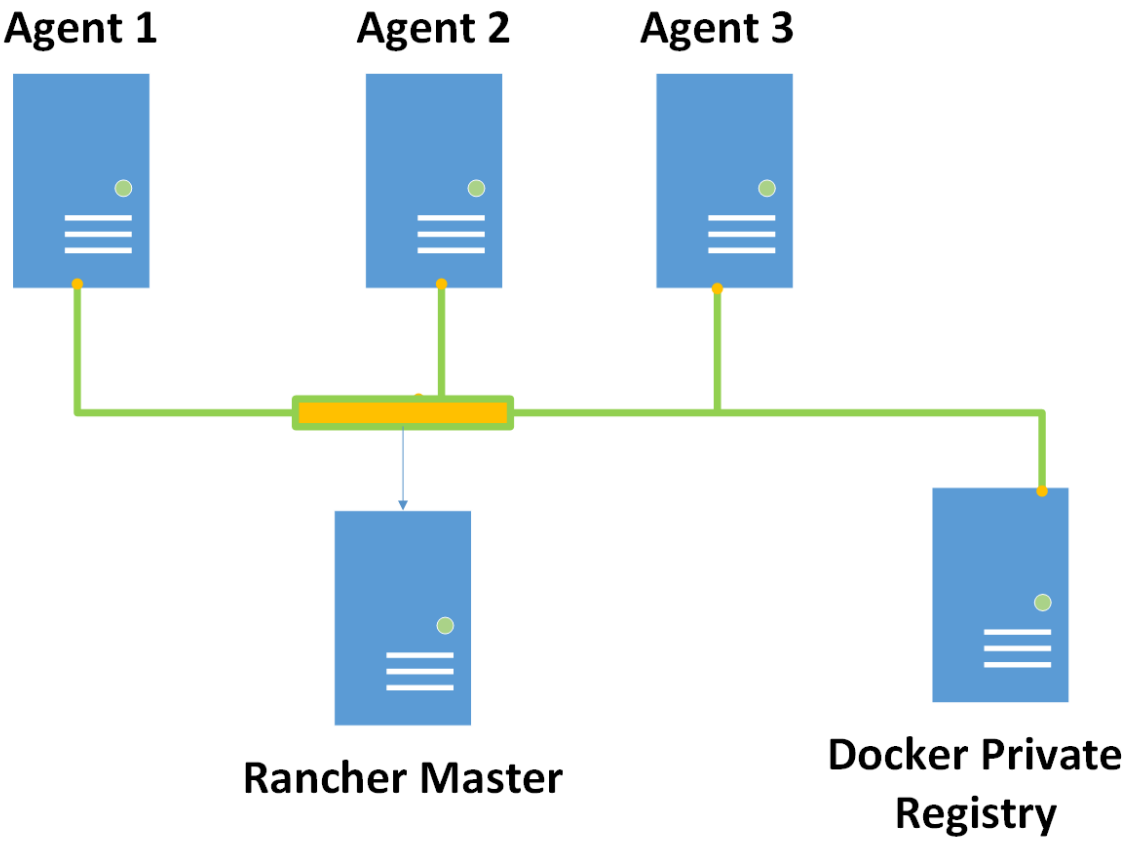

Figure 71 - A layout of the deployed Rancher cluster

Figure 71 shows us the infrastructure layout of the Rancher cluster that we used to execute the comparison between some container orchestrations.

As mentioned in the correspondent chapter, just Kubernetes, Docker Swarm, Apache Mesos, and Cattle are supported. Nevertheless, they are the most known solutions in the orchestration market, and so the purpose of this work is to analyze

these different orchestration solutions. Furthermore, this architecture is even used to test containerization at infrastructure level with Docker. Rancher is quite used for Docker containers in production, and so we decided to adopt this architecture to repeat the same experiments that we decided to perform on OpenStack.

6.6 Benchmarking Tools

6.6.1 Ganglia monitoring system

Ganglia is a scalable distributed monitoring system for high-performance computing systems like clusters and Grids. It widely makes use of technologies such as XML for data representation; XDR for data transport and RRDTool for data storage and visualization. Furthermore, the used data structures and algorithms allow that to achieve very low per-node overheads and high concurrency.

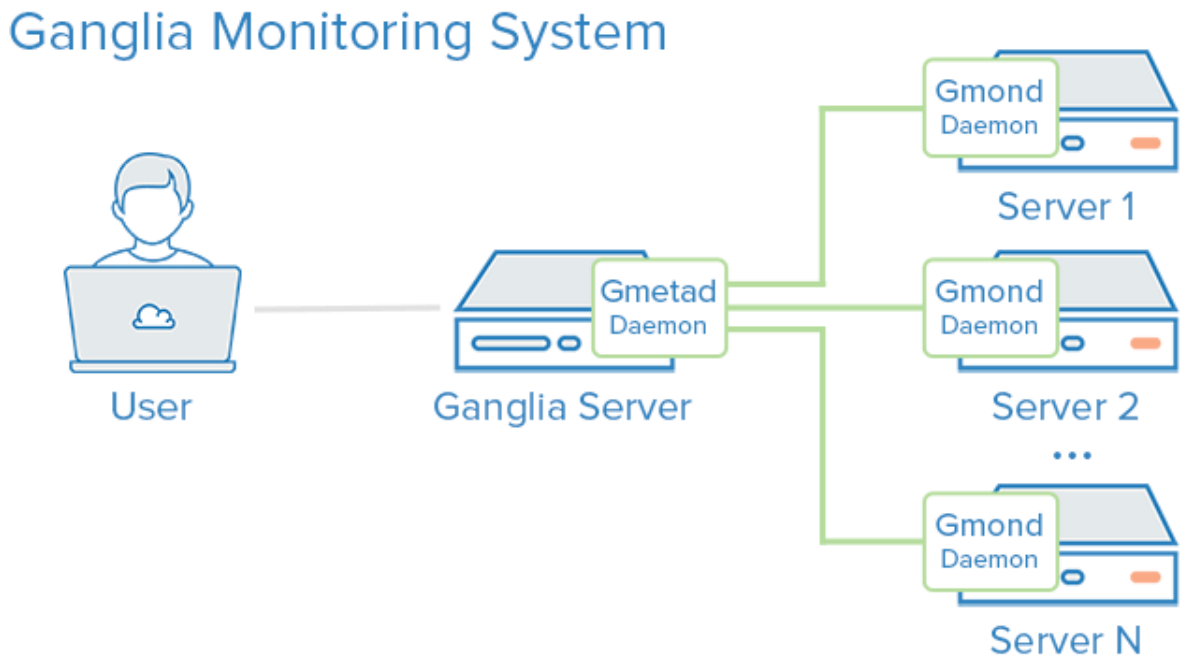

Figure 72 - The Architecture of Ganglia Monitoring System

Figure 72 shows us a diagram of a functional Ganglia cluster [85] that, as it is possible to notice, it is based on the master/slave architectural pattern.

To do that, each node holds a specific agent that is necessary to act as the configured role. The master is called “Gmetad” while the slave runs “Gmond”. Basically, the master agent is responsible for collecting data received by each slave node. These are then shown by a web-based interface that is used by external users to exploit this well-defined solution. Furthermore, we made use of Ganglia to collect data about CPU and Memory usage in the context of the density

benchmark. As explained before, this test consists of using a single server host and atop create multiple compute instances. Ganglia is installed on the physical hosts and so, we can collect data about the underlying hypervisor, by increasing the number of compute instances running on that node.

6.6.2 PXZ

Parallel XZ (PXZ) is a compression utility that takes advantage of running the classic Lempel-Ziv-Markov algorithm [86]. This tool is a parallel lossless compression that can be easily configured to run in any number of cores, therefore making it easy to run on machines with different CPU capabilities [7]. So, we made use of that by creating different OpenStack flavors in order to exploit an increasing number of cores to measure the whole execution time of a compression process. As input for PXZ, we chose to give 300 MB of the random data dump, that has been replicated on the other machines to execute the same process by using the same file. The file is fed into the PXZ algorithm while varying the number of cores. Lastly, the wall time that PXZ takes to compress the file is then recorded and considered to compare the performances between KVM, Docker, and LXD compute instances.

6.6.3 Iperf

Iperf is a widely-used tool for network performance measurement and tuning. It is open source and written in C but it runs on various platforms including Linux, Unix, and Windows. Usually, it is used to produce standardized performance measurements for any types of the network [87]. It is based on client/server model, and the benchmarks are accomplished by creating data streams that are sent from the client to the server. These can be either Transmission Control Protocol (TCP) or User Data Protocol (UDP). This allows us to measure several key aspects such as bandwidth and the throughput between the two ends in one or both directions. In contrast to other solutions, like ping, iperf tries to consume all the bandwidth that the medium can support. Furthermore, there is even the possibility to set the target bandwidth by using UDP data streams. Therefore, in the UDP associated network test, we measured the throughput and bandwidth also considering an increased data rate (from 250 Mbit/s to 1000 Mbit/s).

6.6.4 Ping

Ping is a simple system tool that is commonly used to determine connectivity. Furthermore, it can serve to measure latency – how long it takes one packet to get from X to Y. In fact, for each ping reply received, a round trip time is reported. It

is measured by the local clock in the pinging computer, from when the request left to when the reply arrived.

6.6.5 Bonnie++

Bonnie++ is a free file system benchmarking tool for Unix-like operating systems. It is aimed at performing some simple tests of hard drive and file system performance [88]. Furthermore, Bonnie++ benchmarks three things: data read and write speed, number of seeks that can be performed per second and number of file metadata operations that can be performed per second.

As seen in the correspondent section, LXD is not able to exploit distributed file systems like Ceph. As of release 2.16, it comes with a Ceph storage driver but, considering the time release of this new powerful storage API (August/September 2017), it is not included in the used Juju bundle for OpenStack deployment. Therefore, the OpenStack LXD-based does not support Ceph as block storage, and it makes use of ZFS that is properly a local file system. So, we decided to make a comparison between these two block storage implementations.

6.7 Results

6.7.1 Virtualization and Containerization

This section is aimed at showing the results of the performance analysis that have been executed to compare the usage of containers instead of virtual machines. In OpenStack, as it will be soon demonstrated, LXD is quite suitable to replace the classical virtual machine model, due to the less resource overhead that is involved. However, there are not just advantages, and so, as we have already discussed in the previous chapters, there are also some aspects that do not make of the containerization a fit-for-all implementation. In addition, we even included Docker to repeat the same experiments in a distributed cluster infrastructure with Rancher. However, even if Docker is not a system container, and therefore it is not suitable to completely replace a virtual machine, it presents some benefits. Thereby, if the scenario is suitable, there are cases in which Docker can also be used in production environments.

CPU

To briefly recap what has been done, this test is split into two benchmarks: one to measure the execution time of a single process, by exploiting the paradigm of Multi-threading, and the other one to analyze how this time is influenced in scenarios where the compute instances compete by accessing the same resources.

Flavors	vCPU	RAM(MB)	Root Disk (GB)
A	1	2048	20
B	2	2048	20
C	4	2048	20
D	6	2048	20

Table 11 - Description of the used Flavors

Table 11 shows us the adopted flavors in order to create different compute instances.

However, the definition of flavors is about the two OpenStack clouds that we used to execute compute instances through KVM and LXD. To do the same for Docker, we exploited the functionality of Rancher to limit the resource capabilities of each Docker container. Furthermore, considering that this study requires more resources, we included another physical server host, that is equipped with 64 GB of RAM and 6 processors. This host was used as compute node just for the CPU-Power test and for density for every type of virtualization hypervisor.

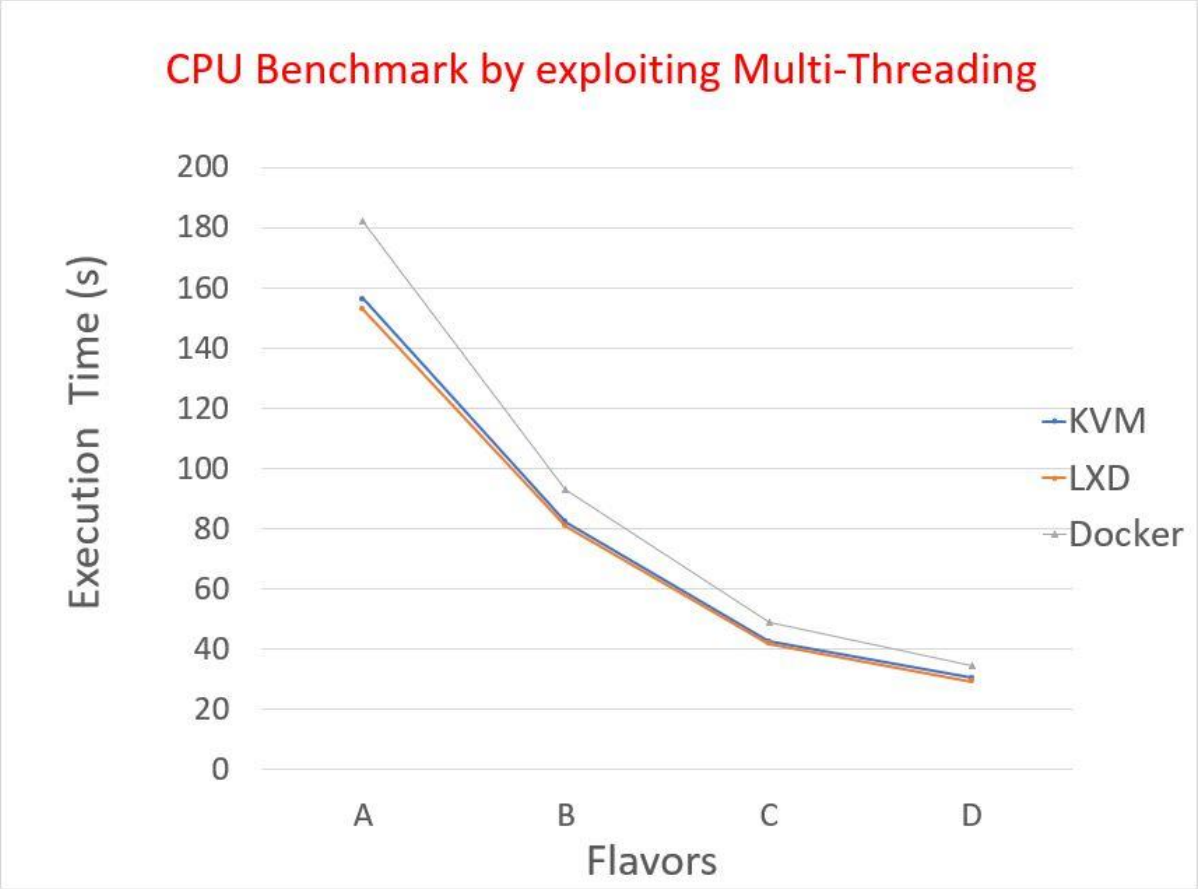

Figure 73 - CPU Benchmark by exploiting Multi-Threading

Figure 73 illustrates the execution time by increasing the number of threads that execute the same algorithm. This consists of compressing a file of 300MB that is

replaced in each compute instance. As it is possible to notice, there is not much difference between the KVM and LXD even if the last one is a bit better. The same is not for Docker that in this experiment presents the worst results.

The purpose of the other experiment is to analyze the behavior of the different virtualization solutions that compete to access the shared resources.

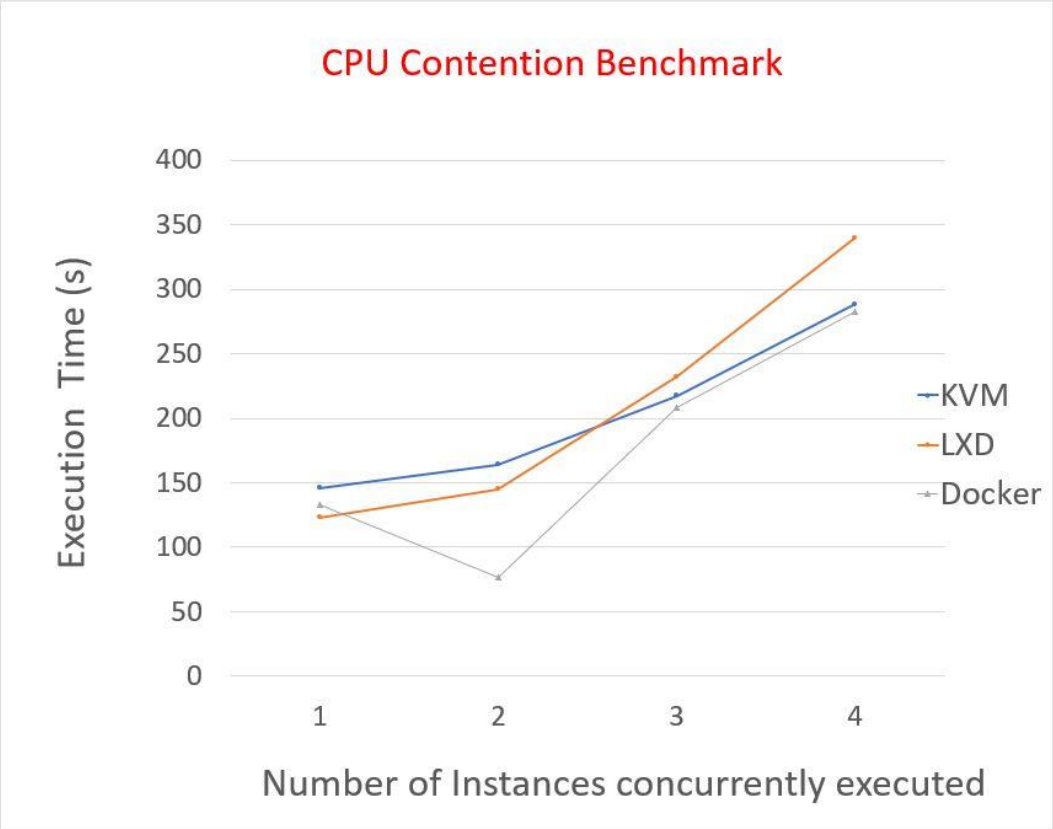

Figure 74 - CPU Contention Benchmark

Figure 74 shows us the result of the CPU Contention Benchmark. We made use of dual-core systems and so we can notice that the execution time of each server hypervisor presents a curve after the contemporary execution of two instances. In a nutshell, Docker is a bit better. Up to two instances, container-based machines imply less time while then KVM is better than LXD. This is not true for Docker that is not affected by the increasing of contemporary instances. Surely, this is influenced by the fact that the underlying file system of LXD introduces an overhead that causes the degradation of the whole performances. On the contrary, the OpenStack virtual machine-based makes use of CEPH that does not include a huge difference by increasing the number of compute instances that compete to access the same physical resources.

Network

This test involves stressing the network subsystem in order to analyze the behavior of bandwidth, latency, and throughput. To measure bandwidth and throughput, we used iperf that is based on a client/server model. For each experiment, the system to be tested acts as a client while another compute instance works as Iperf server. Therefore, as anticipated in the test plan, we split this analysis to investigate three different scenarios: “InterCloud”, “IntraCloud”, and “IntraNode”.

- **InterCloud**: is the case in which the client is a compute machine that runs inside the cluster whereas the Iperf Server is another pair that executes outside the cluster.
- **IntraCloud**: is the case in which the client is a compute machine that runs inside the cluster and the Iperf Server is another pair that executes in the same cluster but on a different physical server host.
- **IntraNode**: is the case in which the client is a compute machine that runs inside the cluster and the Iperf Server is another compute machine that runs on the same physical server host.

Bandwidth and Throughput

As described in the test plan, we stressed the network system by sending UDP packets with a variable bandwidth target. Considering that the NICs are set to work with 1000 Mbit/s, we made tests between 250 and 1000 Mbit/s within an interval of 30s. This is performed just for UDP network traffic. In fact, a key point to remember when testing bandwidth with Iperf is that it consumes all bandwidth available between client/server via TCP, regardless of LAN, WAN, or VPN connection. Furthermore, there is no possibility to define a target bandwidth, and therefore we did not perform the same experiment with TCP. To do that, we measured both inbound and outbound throughput.

InterCloud Scenario

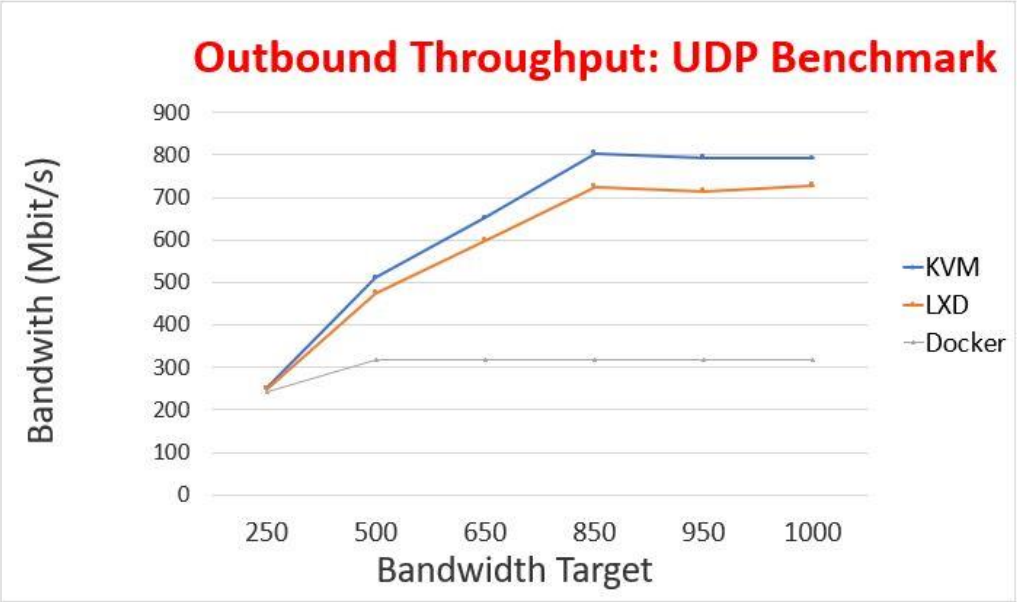

Figure 75 – InterCloud: Outbound Bandwidth - UDP Benchmark

Figure 75 is the behavior of the UDP bandwidth by varying the data-rate in the scenario of InterCloud. As it is possible to notice, KVM is more performant than LXD. On the other hand, Docker is the worst, and it does not achieve a bandwidth target more than 300 Mbit/s. Nevertheless, this is expected because the network subsystem of the container management is implemented with a Network Address Translator mechanism by exploiting features of the underlying kernel host.

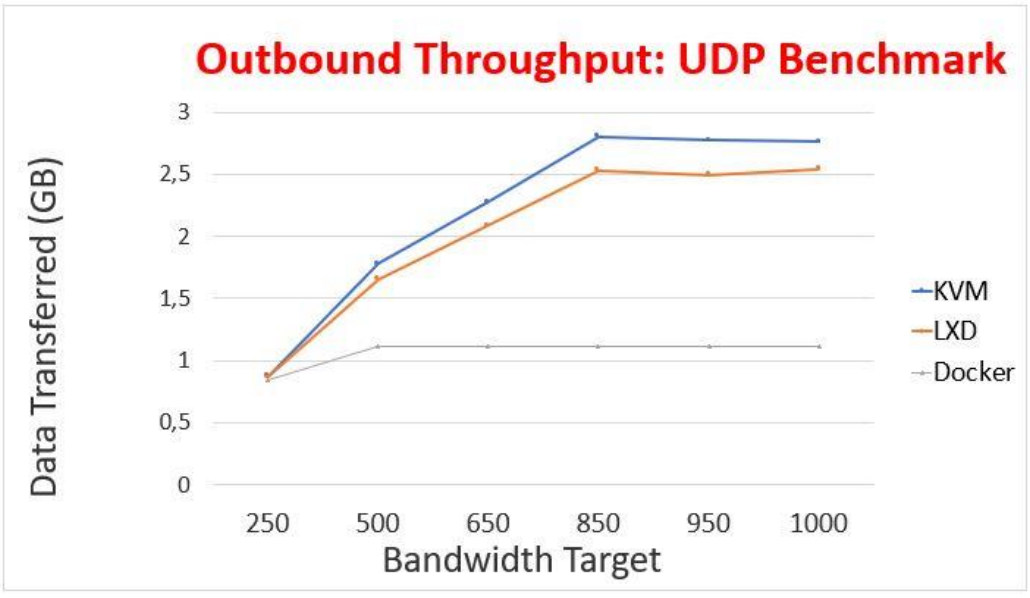

Figure 76 – InterCloud: Outbound Throughput – UDP Benchmark

Figure 76 shows the results of throughput transferred in the scenario of "InterCloud". As seen for bandwidth, KVM network subsystem is able to move a

greater data amount, close enough to 3GB whereas in LXD the peak is about 2.5GB. As expected, considering that Docker is the worst regarding bandwidth, the amount of data transferred is much less than KVM and LXD. On the contrary, we also tested the inbound network subsystem. As it will be shown in Figure 78 and Figure 79, the results are quite similar between both LXD and KVM and, but the same is not for Docker. This technology suffers the external network communication very much and even for inbound communication, the performances are much worse than KVM and LXD. In fact, KVM and LXD saturate the bandwidth around 800 Mbit/s while the peak amount of data transferred is about 2.8 GB.

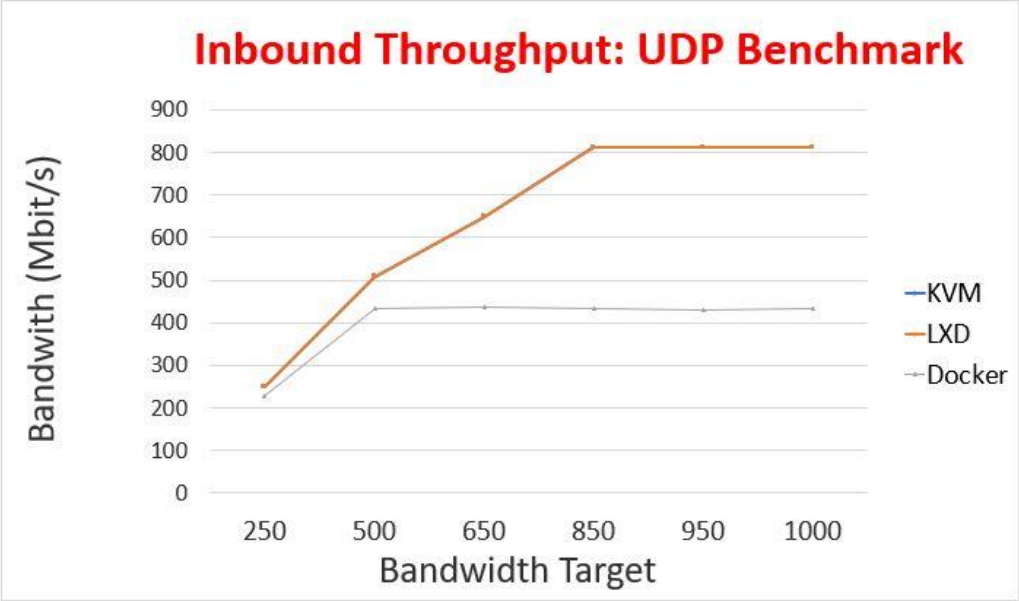

Figure 77 – InterCloud: Inbound Bandwidth – UDP Benchmark

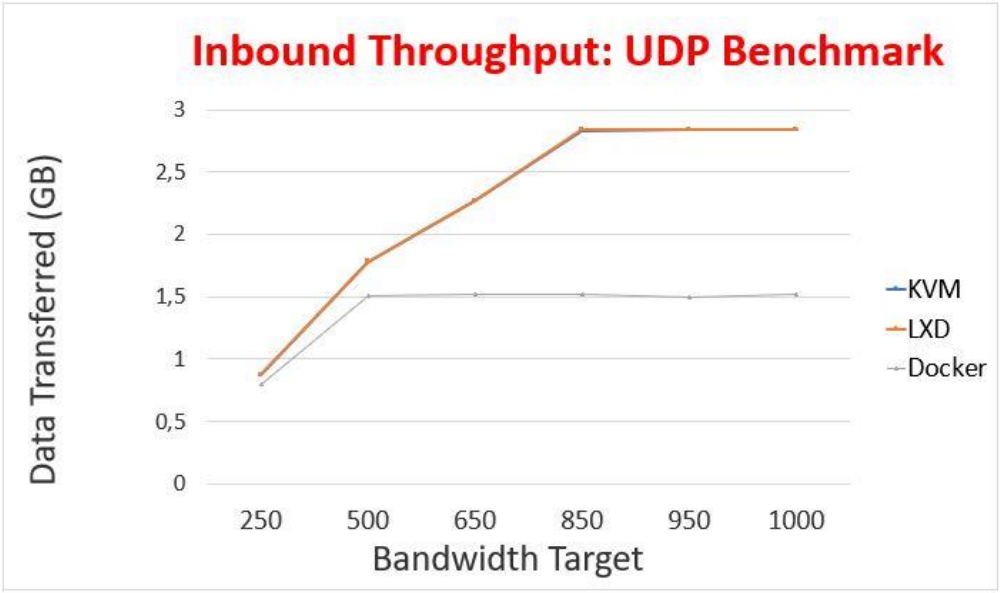

Figure 78 – InterCloud: Inbound Throughput - UDP Benchmark

As anticipated, for the TCP network traffic, it is not possible to stress the system by varying the bandwidth data rate, and so we recorded just the information about a single test by consuming the whole available bandwidth.

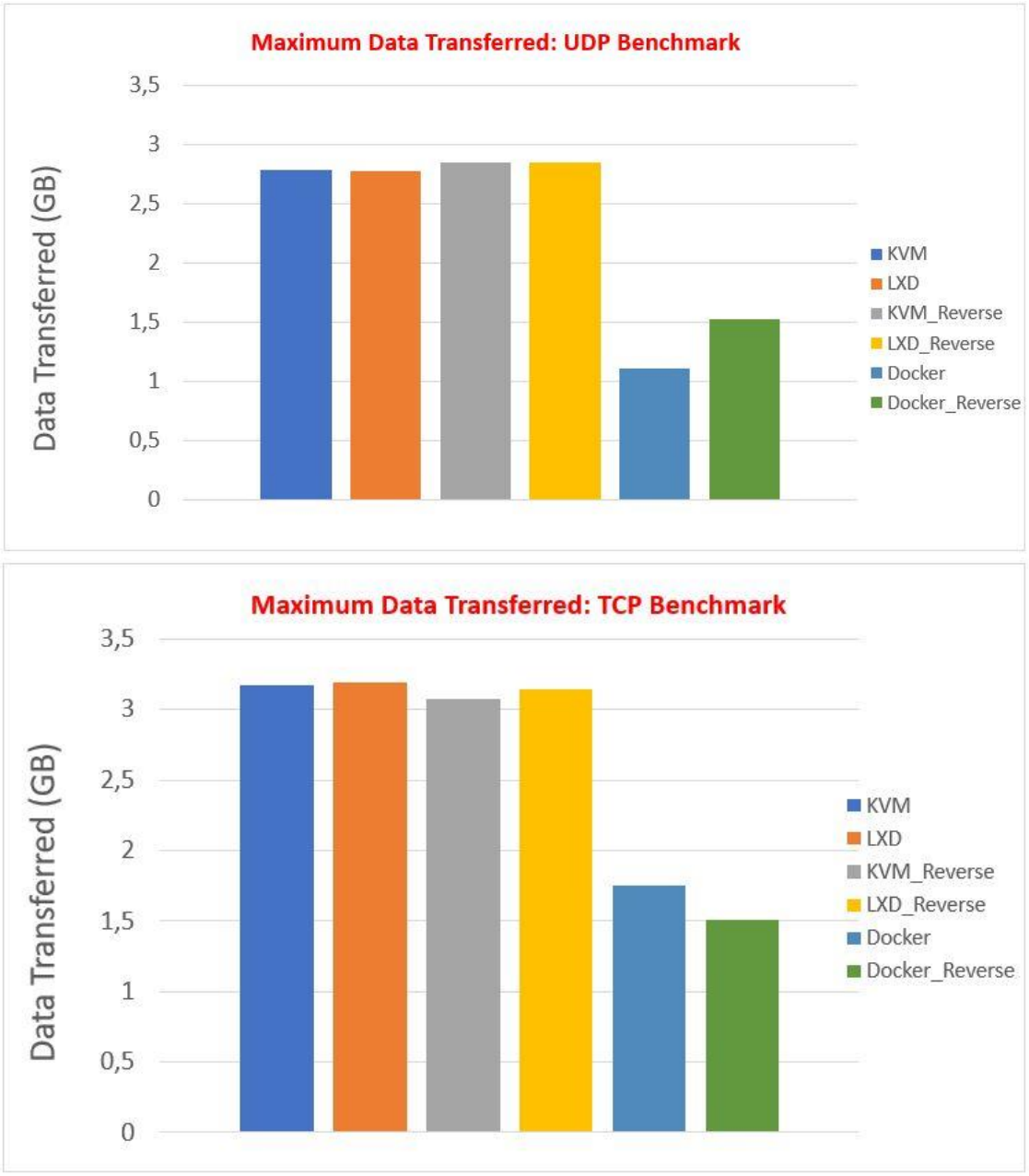

Figure 79 –InterCloud: Throughput Analysis between TCP and UDP

Figure 79 demonstrates that we achieved better performances with TCP by comparing performances with UDP throughput. This is not a surprise. UDP traffic is often rate-limited by network devices because of the lack of inherent flow control. This is the contrary to TCP and therefore we obtained a better result by using a connection-oriented communication with TCP.

IntraCloud Scenario

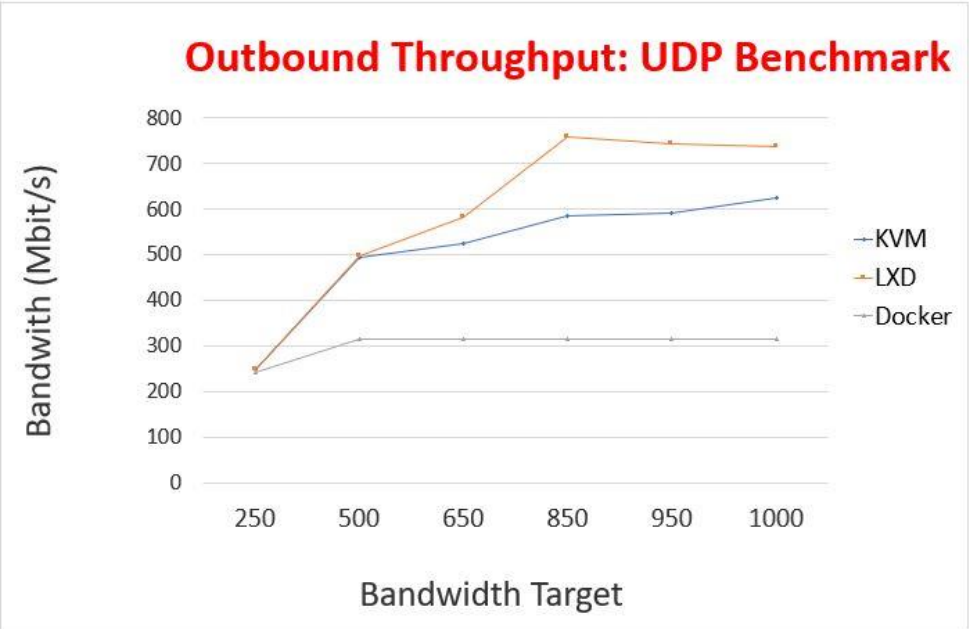

Figure 80 - IntraCloud: Outbound Bandwidth Benchmark

Figure 80 is the behavior of the outbound UDP bandwidth by varying the data-rate in the scenario of IntraCloud. As it is possible to notice, in this case, KVM is less performant than LXD. This means that, as opposed to the deployment LXD-based, the OpenStack implementation with KVM introduces a greater amount of network overhead. Furthermore, even in this case, Docker is the worst. As seen with the previous scenario, the Docker-based compute instances do not achieve a bandwidth target more than 300 Mbit/s. Figure 81 shows the results of the amount of data that the system was able to transfer in the scenario of IntraCloud.

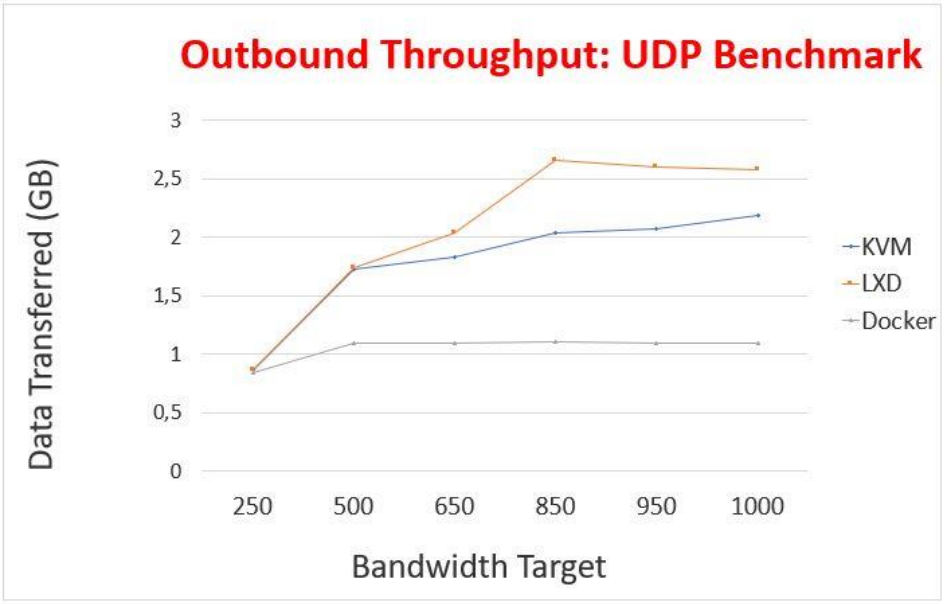

Figure 81 - IntraCloud: Outbound Throughput Benchmark

Surely, this behavior is related to the bandwidth analysis seen in the previous picture. KVM network subsystem moves a smaller data amount, a bit more than 2GB. On the contrary, the LXD behavior is not affected by the different nature of the test, and the peak is about 2.5GB. As regards Docker, as seen with LXD, there is no difference with this other scenario, and more nor less there is no distinction between the two tests.

Even in this case, we analyzed the inbound behavior and, as shown in the next figure, the evolution of the performances is quite similar to that we have just analyzed with the Outbound network system.

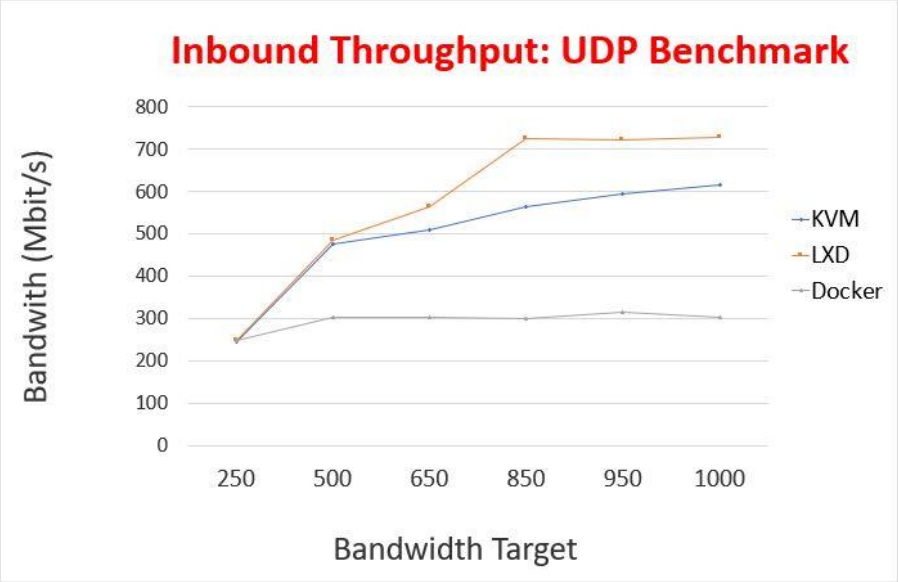

Figure 82 - IntraCloud: Inbound Bandwidth - UDP Benchmark

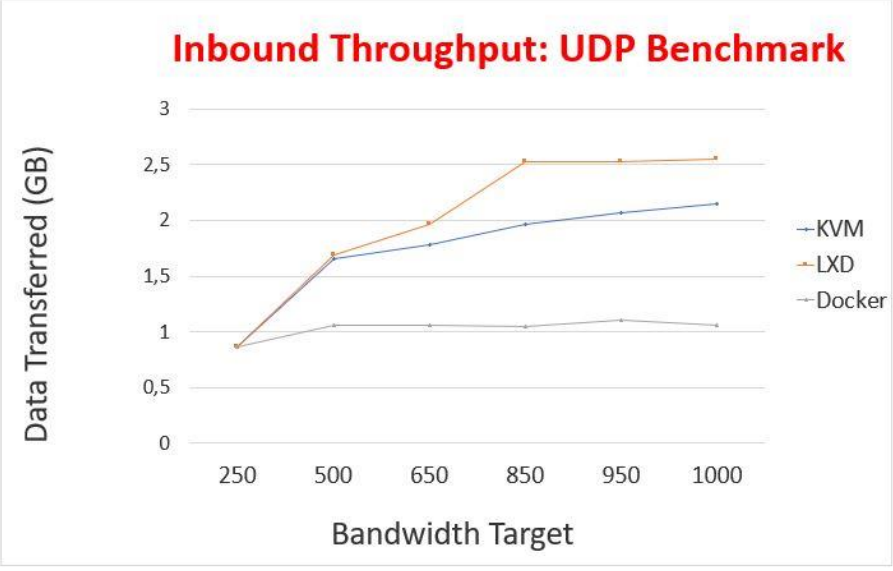

Figure 83 - IntraCloud: Inbound Throughput - UDP Benchmark

Figure 82 and Figure 83 shows us the analysis of the Inbound network.

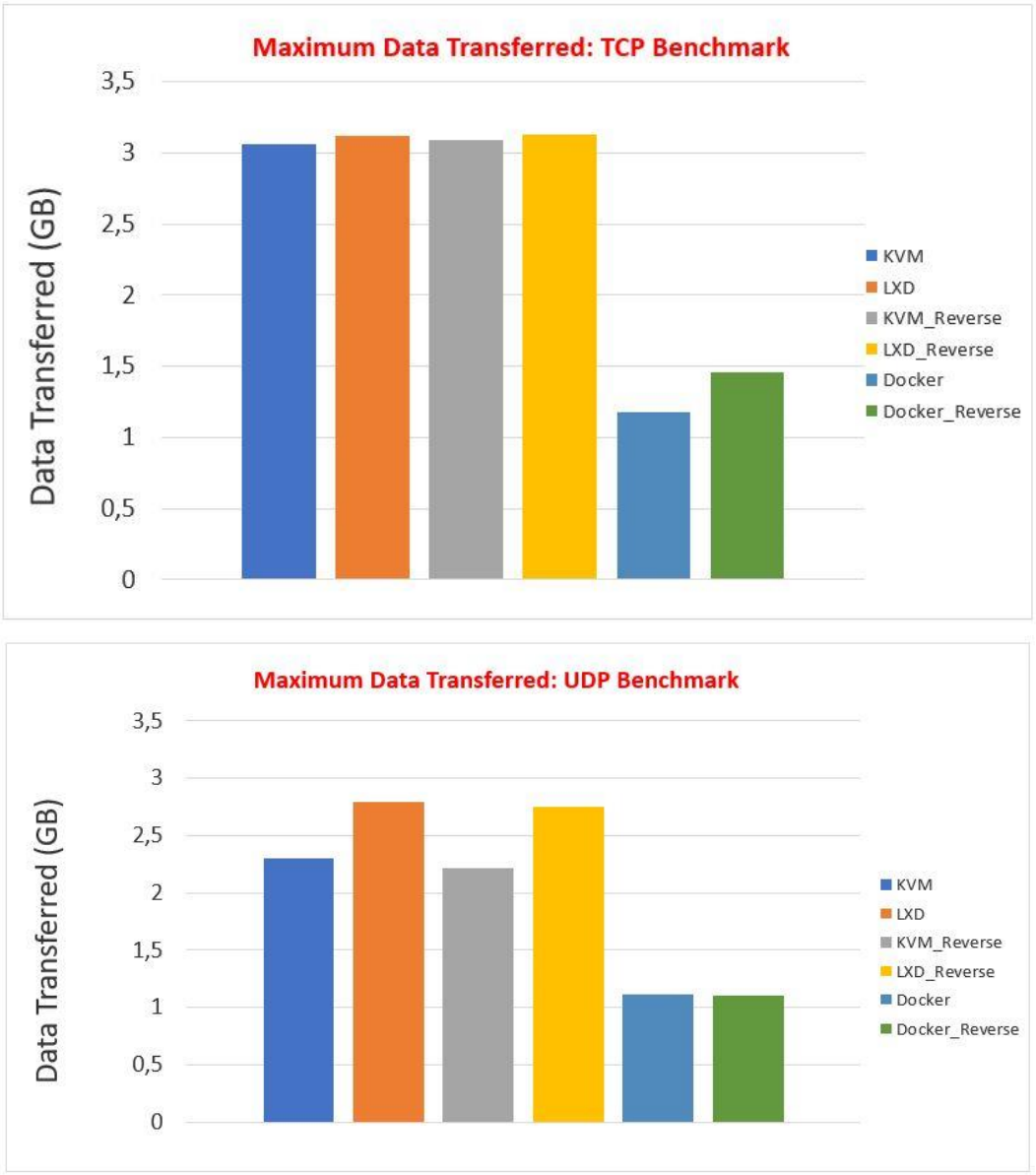

Figure 84 - IntraCloud: Throughput Analysis between TCP and UDP

Figure 84 illustrates us a comparison between TCP and UDP Throughput in the context of "IntraCloud". As expected, LXD can achieve better results whereas Docker is the most affected virtualization technology which is not able to transfer more than 1 GB.

IntraNode Scenario

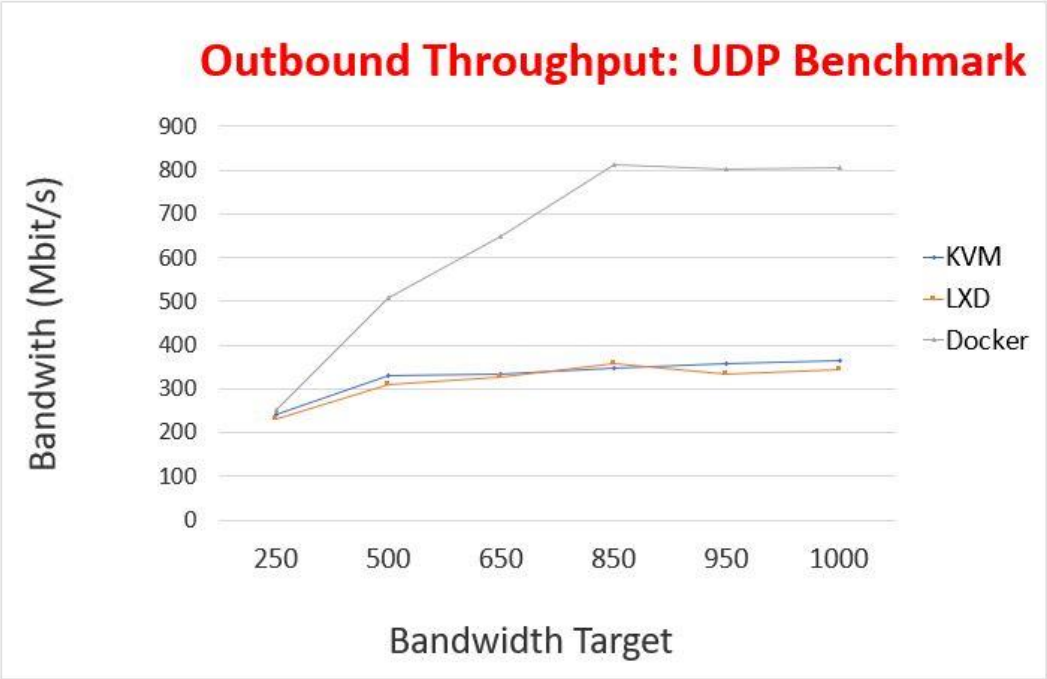

Figure 85 - IntraNode: Outbound Bandwidth - UDP Benchmark

Figure 85 shows us the behavior of the bandwidth benchmark by analyzing the scenario of "IntraNode". This is the case in which the involved compute instances are running on the same physical server host. As it is possible to notice from the picture, there is a substantial difference between Docker and the others. From what we have seen so far, this result appears quite strange. However, this is not surprising because the compute instances obtained from LXD and KVM suffer very much the network overhead introduced by the OpenStack services. On the contrary, Docker has been used with a Rancher platform by which the complementary overhead is not so high, considering that the platform itself is built from Docker containers. So, by this way, we can state that for local communications Docker is more performant because the underlying network subsystem exploits the fact that is about inter-process communication.

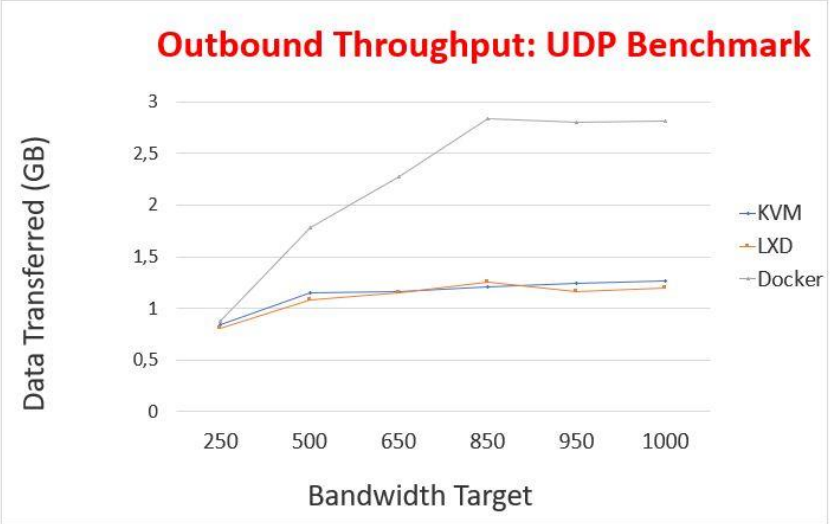

Figure 86 - IntraNode: Outbound Throughput - UDP Benchmark

Figure 86 shows us the dual behavior for the throughput benchmark in the context of two compute machines running on the same physical host. The previous analysis shown us the number of bits per second that the link can send or receive, including all flows. Of course, this does not refer to data usage.

In fact, the key indicator for that is the throughput. Moreover, these are quite related and so, as expected, even in this case Docker is able to achieve better results as opposed to LXD and KVM because they are affected by the network overhead introduced by OpenStack services. Furthermore, as shown in Figure 87, we analyzed the behavior of bandwidth and throughput for the inbound network by using UDP messages. This is after all the same result that we obtained from the outbound network benchmark.

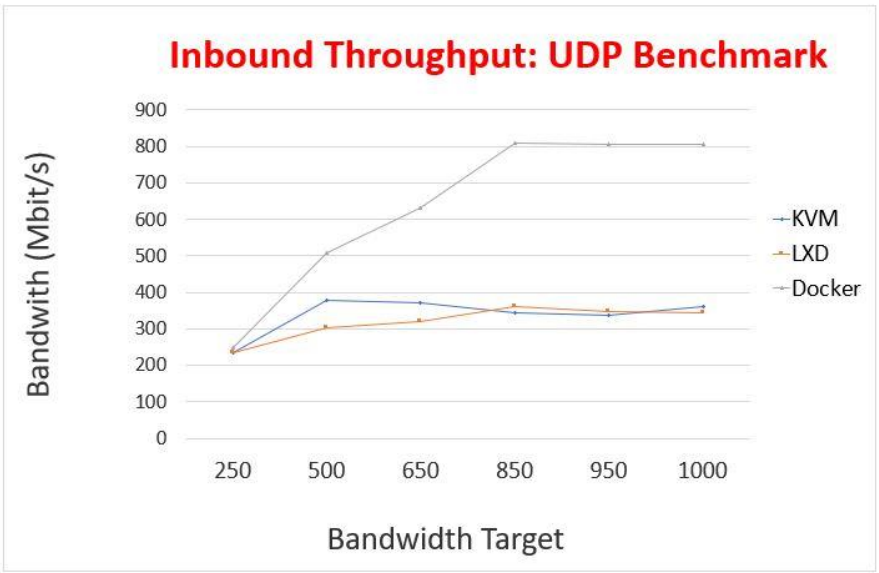

Figure 87 - IntraNode: Inbound Bandwidth - UDP Benchmark

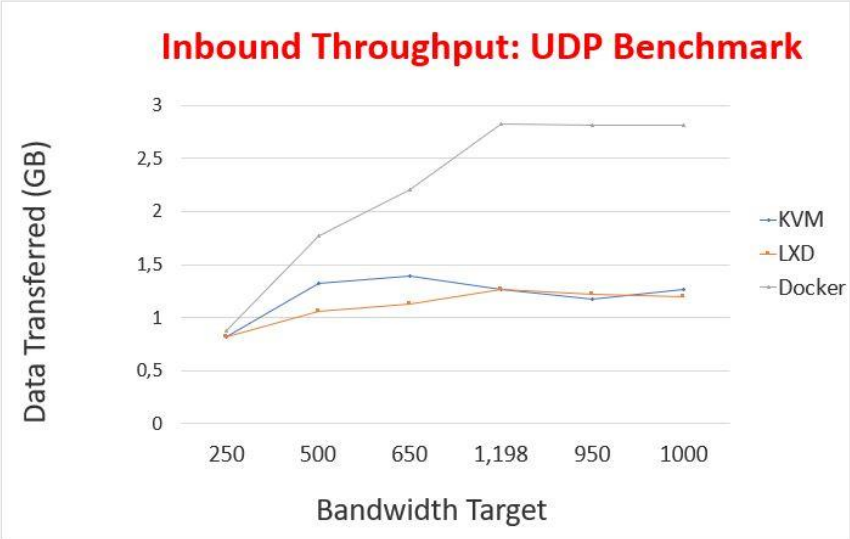

Figure 88 - IntraNode: Inbound Throughput - UDP Benchmark

Figure 88 shows us the analysis of the Inbound network.

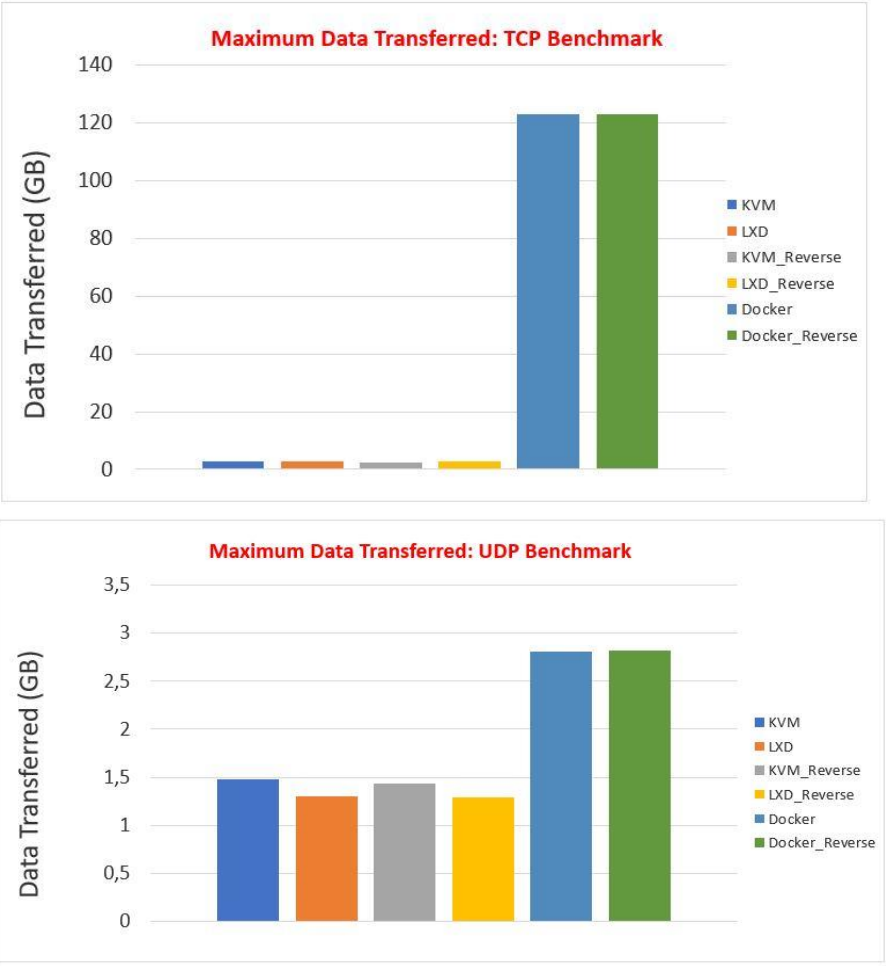

Figure 89 - IntraCloud: Throughput Analysis between TCP and UDP

Figure 89 illustrates us a comparison between TCP and UDP Throughput in the context of "IntraCloud". As expected, Docker is better performant, but the most

surprising result is the data transferred with TCP. In fact, as anticipated before, Iperf tries to consume all the available bandwidth, and therefore the result is quite different as opposed to that obtained with UDP.

Latency

Latency is a time delay between the moment the request is initiated, and the time in which the response is received. The word derives from the fact that during the period of latency the effects of an action are latent, meaning potential or not yet observed. This is quite important because it is quite visible to people who need to wait the time necessary to complete a service request. However, latency is imparted by each element involved in the transmission of data. Therefore, we made use of the ping system utility by sending ten packets and reporting the average round-trip-time (RTT) needed to complete the service request.

InterCloud Scenario

Figure 90 – InterCloud: Latency Benchmark

Figure 90 shows us the latency between the different types of server virtualization. As mentioned in the plan, even this benchmark is based on a client/server model. Moreover, the obtained result is expected, considering that latency-sensitive workloads imply less latency under LXD and Docker, as opposed to KVM. This makes of container-based instances an important new technology in the move to network function virtualization in telecommunications and media, and the convergence of cloud and high-performance computing. Furthermore, in contrast with the previous experiment to test bandwidth and throughput, this benchmark uses the cloud instance as a server and an external machine as the client.

IntraCloud Scenario

Figure 91 - IntraCloud: Latency Benchmark

Figure 91 shows us the latency between KVM, LXD, and Docker. As a result, even in this scenario, the container-based compute instances provides us the possibility to exploit a less amount of time with a complete route to achieve the destination from the client pair.

IntraNode Scenario

Figure 92 - IntraCloud: Latency

Figure 92 presents the analysis of latency with the last scenario, between two compute nodes running on the same physical server host. As expected, Docker is better than others even if there is no much difference between this result and the

previous one. On the contrary, as it is possible to notice from the picture, for KVM and LXD there is a substantial improvement.

Input/output

This test is quite important, considering that the OpenStack KVM-based makes use of a distributed file system whereas the other one exploits just a local file system (ZFS). Therefore, we made a performance analysis also for the file system inside each compute instance to analyze how sensitive workloads can impact the input/output subsystem. Moreover, we considered also Docker that, as specified in the correspondent section, it makes use of a union file system. To do that, we used Bonnie++ that operates by copying and reading from and to big chunks of data. As already mentioned in the planning section, it is important that the used files and data are larger than the amount of system memory. Therefore, we designed an OpenStack flavor with 512 MB of RAM and the benchmark tool makes use of files that have a size of 24GB. Even in this case, regarding Docker, we exploited the possibility to limit the amount of resources of a single Docker container.

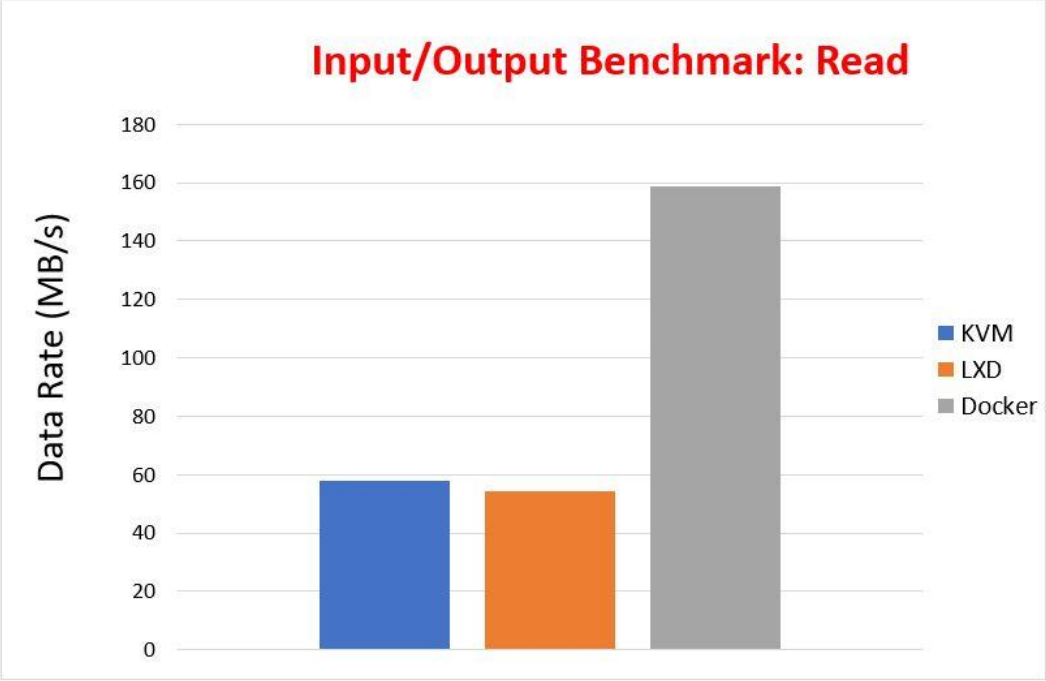

Figure 93 - Input/Output Benchmark Read

Figure 93 shows us the performance achieved with reading benchmarks.

Docker is able to achieve the best performances, considering that the average data rate is much higher than others. The OpenStack KVM-based instance is a bit better than LXD, surely influenced by the fact that the first one exploits CEPH

while the second one makes use of ZFS. In fact, ZFS is a local file system that, on the contrary of the most other file systems, includes a mechanism for snapshots and replication, including snapshot cloning. Furthermore, this represents an overhead by reading data that probably is the cause to make LXD less performant than KVM

Figure 94 - Input/output Benchmark Write

Figure 94 represents the performance results by writing data. As seen in the other case Docker is better than LXD and KVM. However, as opposed to the previous case, LXD is much better than KVM. This is expected, considering that KVM introduces the overhead to distribute the information across the whole cluster, while ZFS is completely local and so there is no synchronization overhead.

Density

An important virtualization feature is the so-called: server consolidation. This consists of an approach to the efficient use of physical server resources in order to reduce the total number of servers or server locations that an organization requires. This was faced with the fact that there were situations in which multiple servers were under-utilized and so they took up more space and consumed more resources instead of using them for their workloads. Containerization is a new technology that can provide better server consolidation, and so the purpose of this benchmark is how these resources are influenced by increasing the amount of compute instances that a physical server host can spawn. To do that, the benchmark was structured with the usage of the “stress” Linux utility that is

simultaneously executed across each compute instance. Furthermore, the behavior of the underlying physical resources has been monitored with the support of Ganglia.

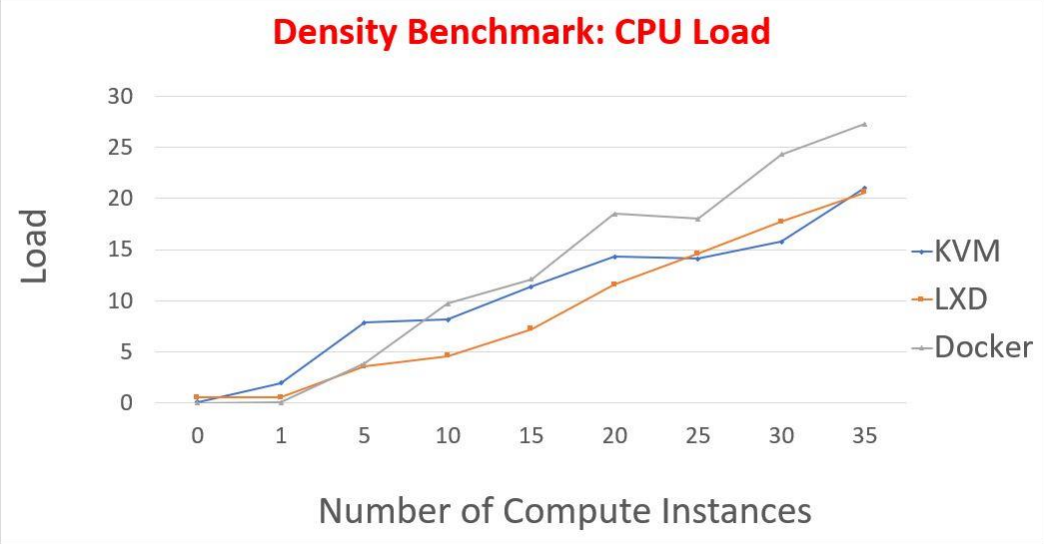

Figure 95 - Density Benchmark with CPU Load

Figure 95 shows us the behavior of CPU load by increasing the number of compute instances that are scheduled on a single bare-metal OpenStack Compute Node. As expected, LXD-based OpenStack instances allow us to reduce the CPU load drastically. Moreover, there is a point where Docker is at most better than KVM. After that, Docker-based compute instances involved a higher value of CPU-Load.

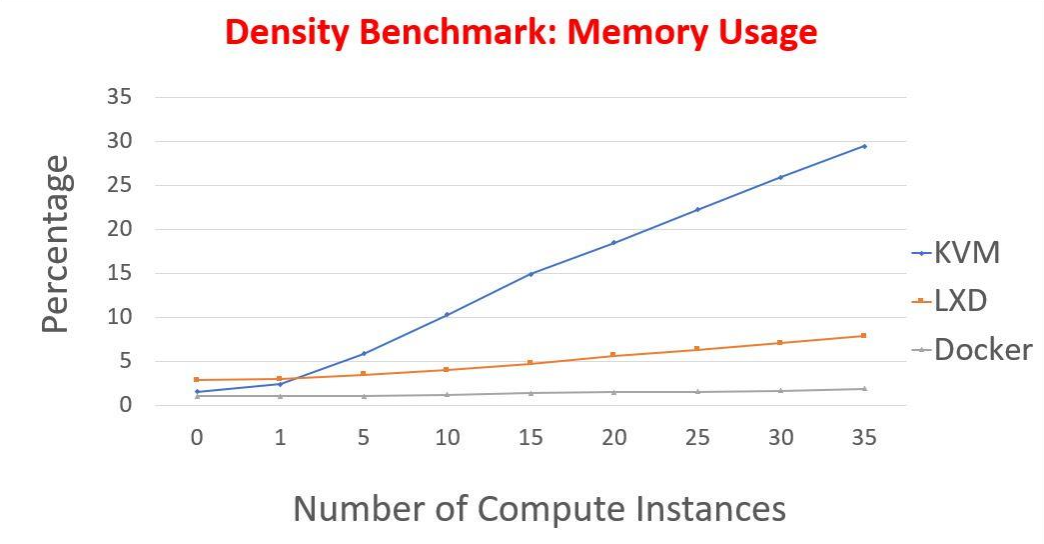

Figure 96 - Density Benchmark with Memory Usage

Figure 96 illustrates the behavior of the system memory by increasing the number of compute instances. As it is possible to see, KVM implies a greater amount of

memory used, considering the fact that a single virtual machine consists of the inclusion of an entire guest operating system. Furthermore, as seen for the CPU Load, the fact that each physical resource is emulated in the virtual machine model justifies this result. This demonstrates how LXD crushes KVM in density and so this constitutes a dramatic improvement on traditional virtualization. This is particularly valuable for large hosting environments that can be hosted on a fraction of the hardware using LXD instead of KVM. In addition, we can notice the key differentiator of Docker-based instances: memory usage. In fact, as it is possible to notice from the picture, Docker does not involve a huge amount of memory by increasing the number of compute instances per host. This is expected, not just for the overhead introduced by OpenStack components, but particularly dependent on the underlying architecture of Docker runtime.

System

Lastly, we considered also the impact of containerization on system aspects such as the time necessary to complete system processes or the size that a single image requires being hosted by a physical server host. Therefore, we structured this test to measure such times: boot, snapshot, and provisioning. Furthermore, we evaluated the disk impact of a single operating system image between different Linux distributions.

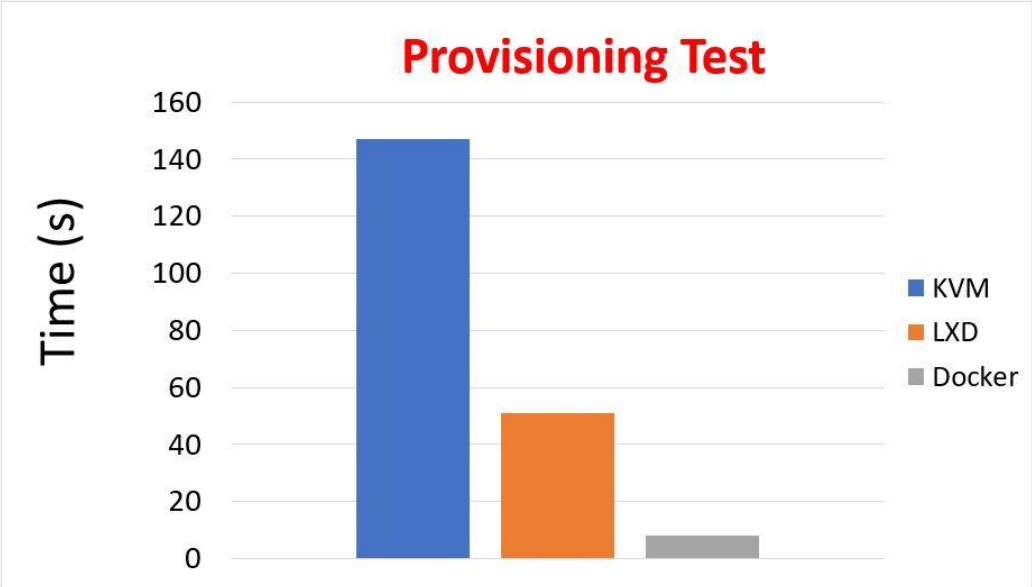

Figure 97 - Provisioning benchmark

Figure 97 illustrates the result of the comparison between the KVM, Docker, and LXD provisioning. As expected, LXD completes the whole process around 50s whereas KVM requires a bit greater than 140s. Furthermore, the shortest time is

achieved by Docker that is also more performant than LXD. To do this experiment, we made use of a private Docker registry, running on a different server host of the same cluster, that was exploited by the local Docker daemon to instantiate a single container. This gave us the possibility to emulate the same OpenStack deployment which has been used to evaluate LXD and KVM. Moreover, one of the key differentiators of container-based instances is the provisioning time. This is influenced by the fact that a single virtual machine, to be provisioned, requires loading the whole operating system, whereas a container is basically an isolated process, which includes some system functionalities due to features of the underlying kernel host. Furthermore, as seen for density, this is one of the key differentiators that make of LXD a more appropriate cloud hypervisor solution.

Figure 98 - Boot Benchmark

Figure 98 shows us the execution time needed to complete a single boot process. This is even expected because there is no need to involve scheduling and more, but it consists of communicating to the hypervisor to start up the compute instance.

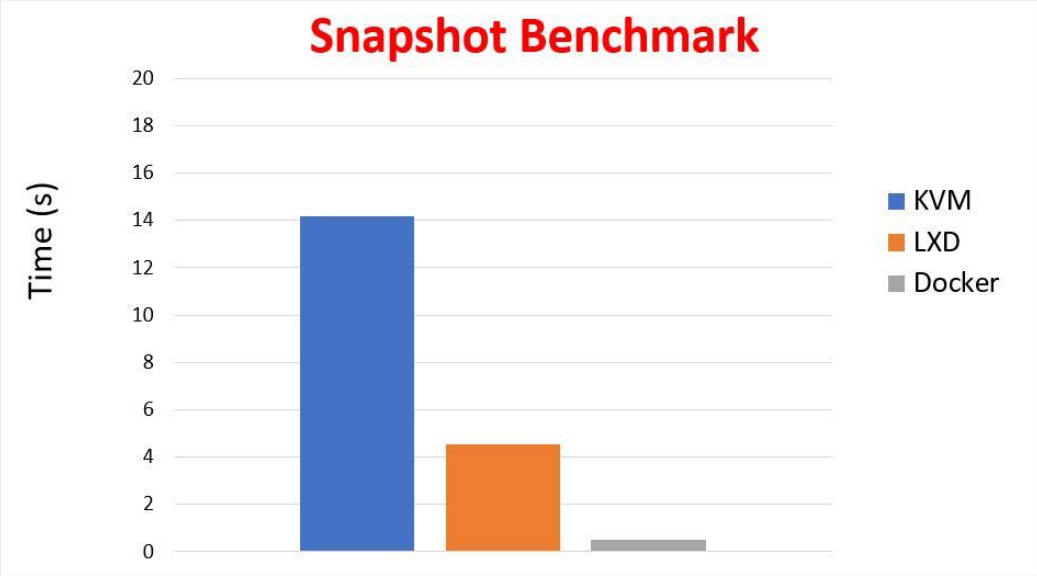

Figure 99 - Snapshot Benchmark

Figure 99 shows us the time needed to complete a single snapshot. This is useful when we need to preserve the state of the entire compute instance in order to return to the same state repeatedly. Snapshot technology is commonly defined as a virtual copy of a set of files, directories or volumes, at a particular point in time. These are often used in storage systems to enhance data protection and efficiency. This is fundamental to solve several data backup problems, and so, it is important in cloud deployments. Moreover, even in this case, container-based instances offer us the possibility to complete the service request with the shortest time.

Lastly, we performed an analysis of the impact that is involved with the operating system images. In fact, they are fundamental to create server instances and so, this test is useful if we want to analyze the overhead introduced by the different virtualization technologies.

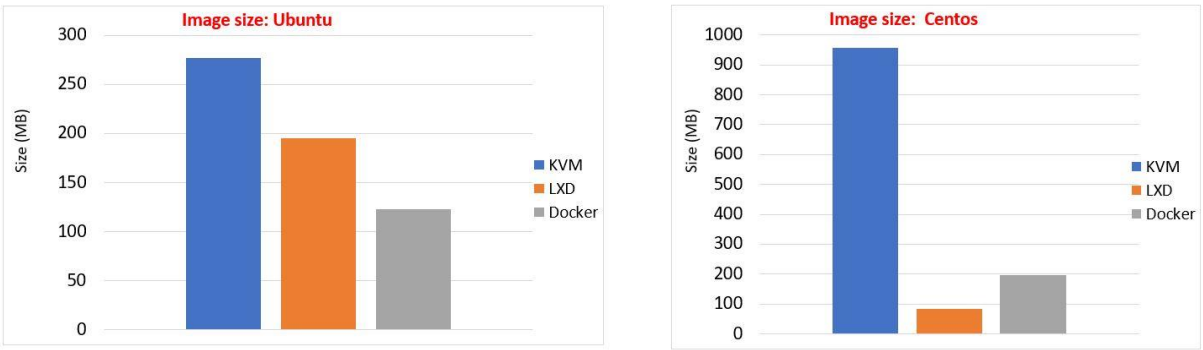

Figure 100 - Comparison of size between Virtualization Images

Figure 100 shows us the impact on disk of the operating system images that are needed to host in order to create each compute instance. As expected, even in this

case, we can demonstrate how a single container introduces less overhead than a traditional virtual machine.

6.7.2 Container Orchestration

Until now, we analyzed the behavior of containerization at the infrastructure level, with the purpose to evaluate its performances as opposed to those obtained by using the traditional virtual model. However, this is not enough to properly adopt the technology in distributed environments and particularly to cloud infrastructures. In fact, as seen in the second chapter, the container management layer is responsible just to manage the lifecycle of one container on one host. When there is the need to manage multiple containers, deployed on distributed hosts, this model falls short. Therefore, we must turn to orchestration tools. These extend lifecycle management capabilities to complex, multi-container workloads that can be deployed on a cluster of machines. Furthermore, by abstracting the host infrastructure, orchestration tools allow users to treat the entire cluster as a single deployment target.

Containers make software development easier, enabling us to write code faster and run it better. However, running containers, and the related orchestration tools, in production can be hard. In fact, there are a wide variety of technologies to integrate, and new tools are emerging every day. Rancher makes it easy to manage all aspects of running containers in production. We do not need to develop the technical skills required to integrate a complex set of open source projects. In fact, it includes everything we need to manage software-containers in production, with no need to build a specific container management platform from scratch by using multiple open source technologies. This is quite important not just for the time to provision the entire cluster, but also from the flexibility point of view.

As seen in the correspondent chapter, each container orchestration solution presents something that is different from the others. Therefore, users should be able to manage different possibilities according to their needs. Rancher gives us the capability to exploit the same infrastructure level by quickly deploying an orchestration tool. For this reason, we decided to evaluate the performances of container orchestrations by provisioning a Rancher cluster. However, at the moment Rancher does not support every orchestration. In fact, some projects belong to a specific cloud provider and others that, even if they are open source, are not quite mature to be adopted in production environments. Therefore, in this experiment, we tested Kubernetes, Docker Swarm, Apache Mesos, and Cattle. Furthermore, just Docker is supported at the level of container management, and

so, we performed this test by evaluating different orchestration tools with the same container runtime.

Considering that we use a single Rancher cluster to deploy each container orchestration, an important key indicator is the time needed to build the so-called "environment" with a specific container orchestration. Furthermore, this approach is affected by the fact that an orchestration model is even used to deploy a single architecture. This is the same principle of what we have seen for Juju with OpenStack. Therefore, the purpose of the next picture is to analyze what are the differences, in terms of time measurements, to provision a single cluster with the adoption of a supported container orchestrator.

Figure 101 - Cluster Provisioning Time in Rancher

Figure 101 presents the cluster provisioning time to deploy a single orchestration tool by using the Rancher platform. As it is possible to notice, Cattle requires the shortest time to deploy a single cluster. This is not surprising. In fact, that is the native orchestration tool of Rancher platform, and so, the entire architecture is optimized to deploy the container orchestrator that is provided by default. Another important aspect is that Kubernetes introduces the highest provisioning time. Of course, this is affected by the fact that the architecture is more complex than others. As seen in the correspondent chapter, Kubernetes is the solution that provides the most capabilities to manage and deploy services in production. However, it is based on a different design model that introduces a not negligible overhead, and so, this justifies the result that Rancher takes more time to provision a Kubernetes cluster.

Nevertheless, an orchestration tool is important to manage enterprise applications in production, and so, the purpose of this analysis is to evaluate the behavior of each orchestration tool to provision different container-based applications.

The first analysis concerns the deployment of a simple web application. This is composed of two microservices: one to host a database to store persistent data whereas the other one to provide the front-end of the web application. As explained in the planned test, the goal of this test is to analyze how the provisioning time is influenced by increasing the number of replicas of the front-end container.
To do that, we split this investigation into two scenarios: one that makes use of Docker images pre-installed on each Docker host and the other one that does not have the Docker image. In this last case, they need to download the Docker image, and so we designed to download the Docker image from a private Docker registry that is built on a different physical server on the same cluster.

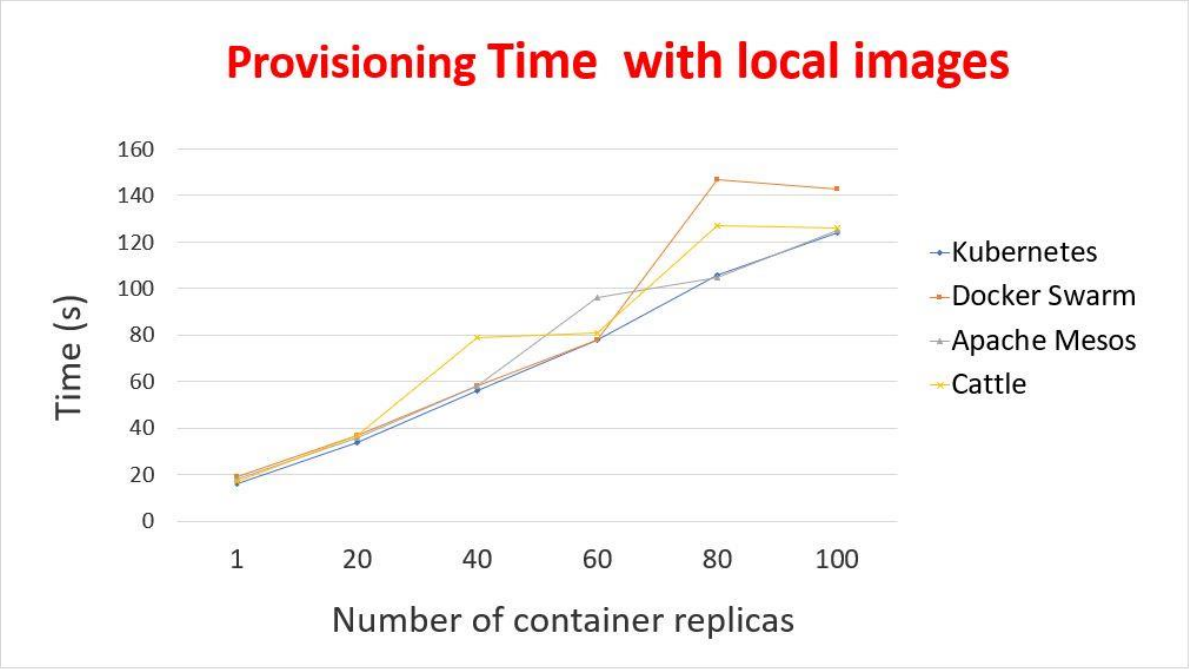

Figure 102 - Provisioning Time with local images

Figure 102 shows us the provisioning time of the web application that exploits the fact that each Docker image is present on each server node.

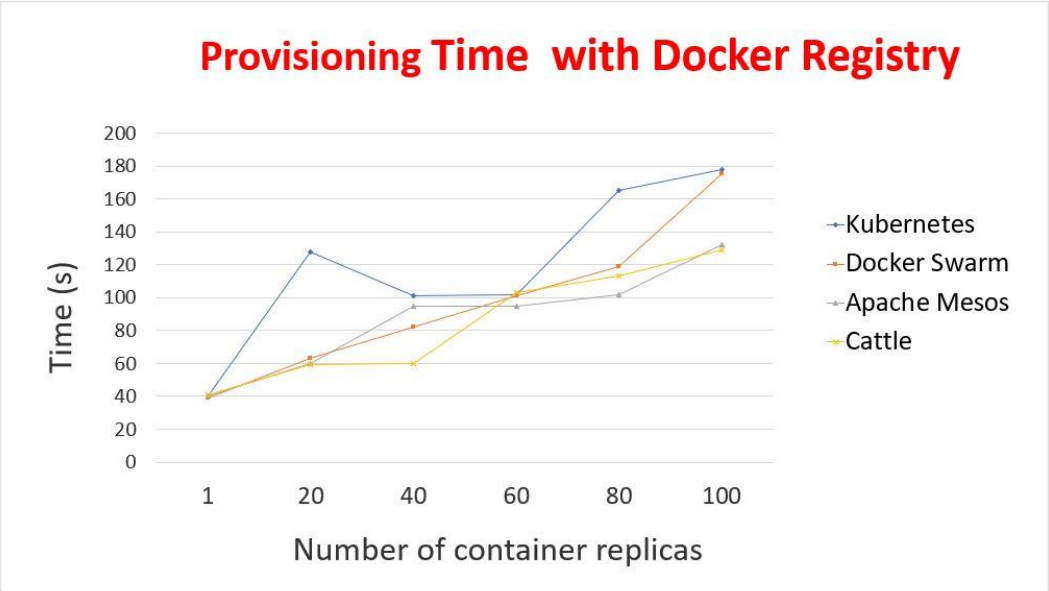

Figure 103 - Provisioning Time with Docker Registry

Figure 103 shows us the provisioning time of the web application that consists of downloading the Docker image from the Docker private registry that is installed on a different physical server host of the cluster.

The results are quite different because the two scenarios involve different operations to perform each user request. In the first case, Kubernetes is the solution that takes the shortest time to provision the web application. On the contrary, in the second case, it is the worst. This is affected by the fact that the interaction between the Kubernetes agent and the Docker registry introduces a not negligible overhead that makes Kubernetes the solution with the highest time to provision the entire web application. In fact, in this case, Cattle is more or less the best solution. This is not surprising because it is the Rancher-native orchestration tool and so, the interaction between each Cattle agent is optimized to interact with an external private Docker Registry.

Furthermore, we analyzed another case study to evaluate the behavior of the provisioning time by deploying different applications that are characterized by an increasing complexity rate. To do that, we chose three applications: Jenkins, WordPress, and GitLab.

- **Jenkins:** is a continuous integration server. This is the practice of running our tests on a non-developer machine automatically every time someone pushes new code into the source repository. A single microservice composes this project.

- **WordPress:** is a Content Management System (CMS) based on PHP and MySql. This supports the creation and modification of web sites. Two microservices compose this application.
- **Gitlab:** is a web-based Git repository manager with wiki and issue tracking features. This is important for tracking changes in computer files and coordinating work on those files among multiple users and teams. This is composed of four microservices.

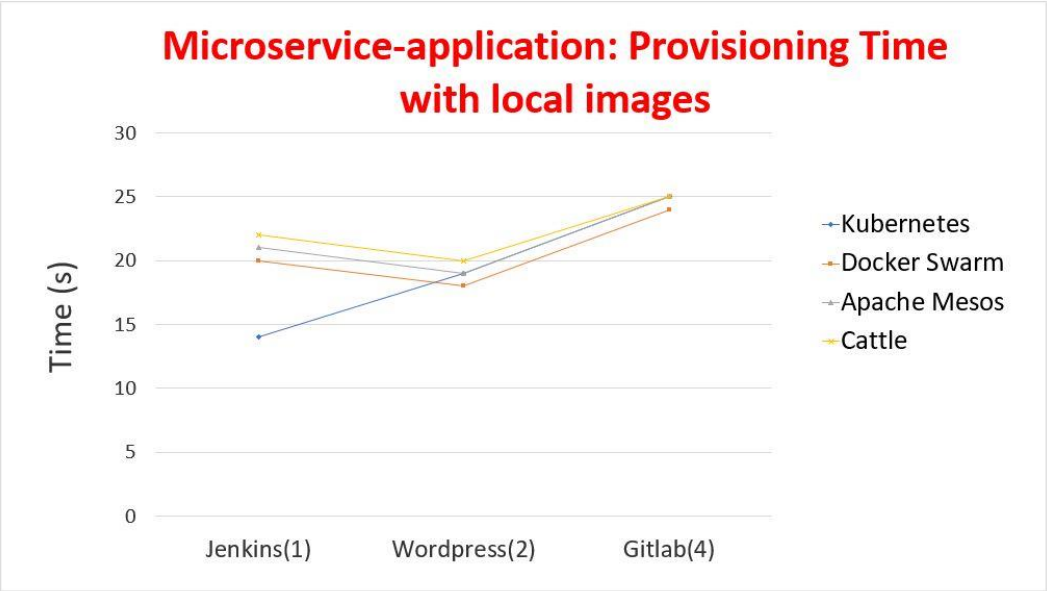

Figure 104 - Microservice-application: Provisioning Time with local images

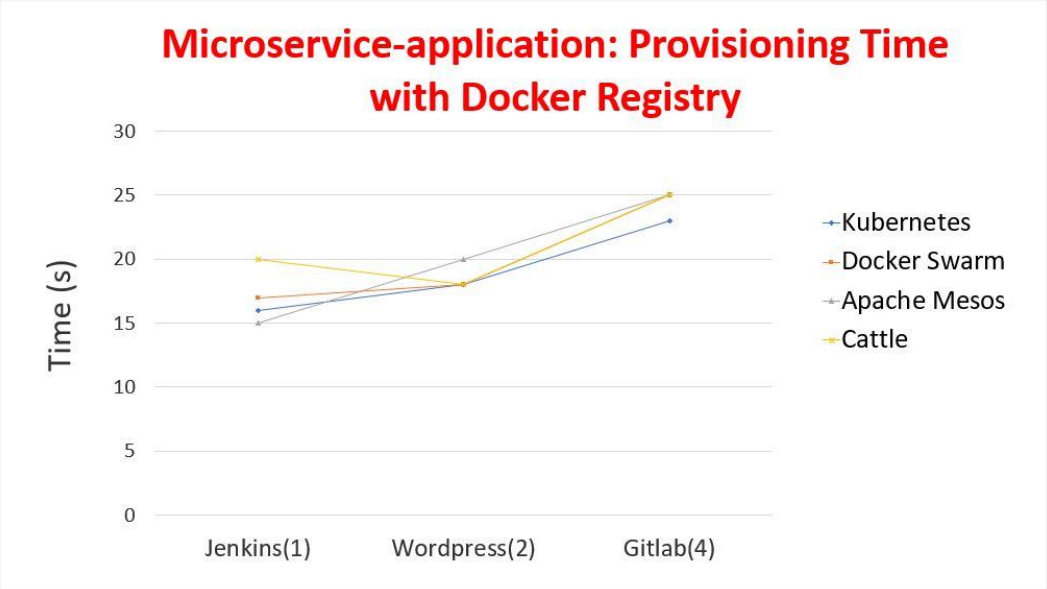

Figure 105 - Microservice-application: Provisioning Time with Docker Registry

Figure 104 shows us the provisioning time with local Docker images while Figure 105 illustrates the case in which the microservice-based applications are deployed

by interacting with a private Docker registry. As seen in the previous experiment, even in this case Kubernetes is affected by the interaction with a remote Docker registry. Of course, this is not just network-dependent. In fact, as explained in the associated chapter, the atomic unit of Kubernetes is the pod and not the single Docker container.

This is another abstraction level that is needed to co-locate a set of Docker containers that want to exploit the inter-process communication. However, even if the pod is composed of a single Docker container there is no constraint to avoid the creation of that component, and so the number of applicative components increases, as well as the deployed applications, are built with multiple containers in a single Kubernetes pod.

The last case concerns the failover time that is needed to guarantee a service level request that is aimed to provide the highest uptime. This is quite important with enterprise applications because users do not like to see their applications not available and so the orchestration capability should guarantee that in every time the services must be running even if the failure concerns the physical server host on which the application is executing on.

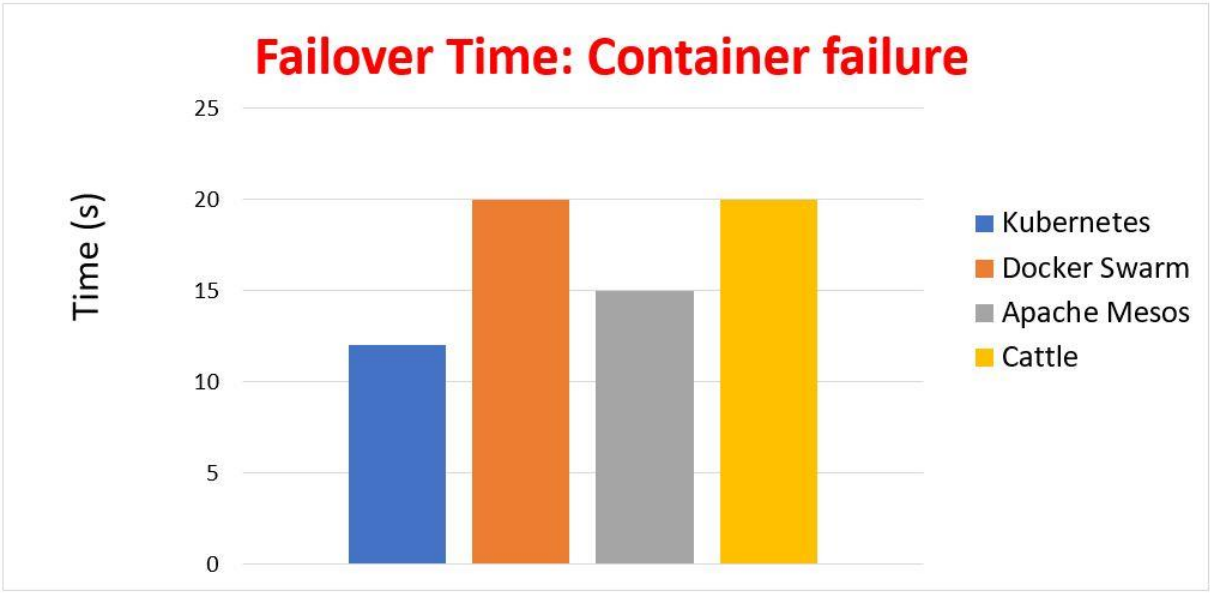

Figure 106 - Failover Time: container failure

Figure 106 shows us the failover time with a container failure.

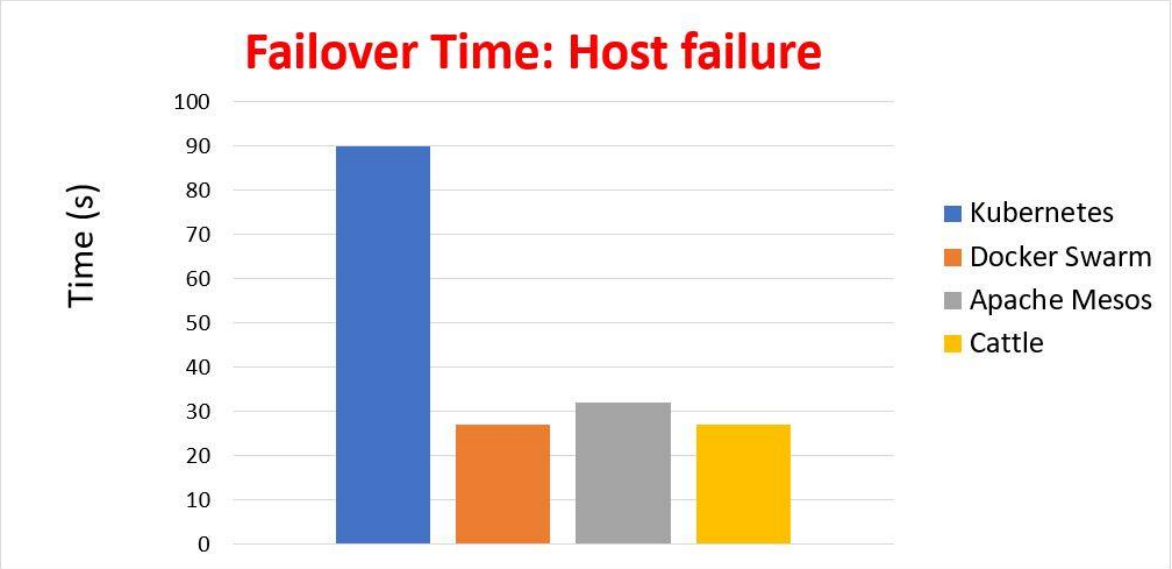

Figure 107 - Failover Time: Host failure

Figure 109 shows us the failover time of the single container whereas Figure 110 is related to the host failure.

This feature is the so-called "rescheduling" that is natively supported by Kubernetes and Docker Swarm. However, Rancher introduces the possibility to define the minimum number of running containers for each microservice. Therefore, there is a dedicated Rancher functionality that provides this feature, even if the underlying container orchestrator does not offer this capability. For this reason, we used the native feature for Kubernetes and Docker Swarm, whereas for Cattle and Apache Mesos we exploited the Rancher capability.

As it is possible to see from the pictures, Kubernetes is the best solution to the failure of the single container. On the contrary, it is the worst in the case of a node failure. Nevertheless, this is not surprising. In fact, in the first case, the failure is detected by the local Kubernetes agent that makes sure to guarantee the availability of the running containers. On the other hand, with the node failure, there is the so-called "Replication Controller" that is responsible for guaranteeing the rescheduling of the failed pods. This approach is event-based because the Kubernetes object is notified by the Kubernetes Controller when the number of pods changes. Therefore, with Kubernetes we obtained the highest value of failover time. The same is not for Docker Swarm or Cattle because they exploit the heartbeat functionality of the Docker architecture and so, they provide the failover mechanism with the shortest time.

6.8 Closing remarks

As expected, we can confirm that also in cloud environments the containerization represents a valid alternative to the traditional virtual machine model. However, there are also open questions that emerge from this work as what about LXD when the new storage API will be integrated into OpenStack, guaranteeing the same interoperability with a distributed file system. This paradigm is still not mature, and so the interoperability with other solutions is a little hampered. As we anticipated before, the storage API of LXD does not support the adoption of a distributed file system like CEPH. Therefore, it cannot be integrated into OpenStack with Cinder, but it makes use of a local file system like ZFS. This is the most recommended backend storage system because it supports all the features LXD needs to offer the container experience. Moreover, the purpose of this thesis is to investigate the performance behavior between compute instances obtained by virtual machines and containers. So, we just made use of the available implementation of LXD in the OpenStack architecture.

Therefore, the next chapter discusses a case study in which the purpose is to design an applicative case in order to provide fault-tolerance with the available implementation of LXD in OpenStack. In fact, as already mentioned before, by this way, users cannot exploit low-level services of cloud operating system and so, there is the necessity to manually provide an ad-hoc solution in order to bypass this limit.

In the context of container management, we have learned that containerization has led the classification between application and system containers. LXD focuses on system containers by trying to provide the same capabilities of the traditional virtual machine model. On the contrary, application containers like Docker provides good performances in terms of provisioning time and resources overload. Nevertheless, application containers cannot be used to replace an entire virtual machine because they are focused on simplifying the management and deployment of enterprise applications. Therefore, we then evaluated the behavior of some container orchestrators to perform a comparison analysis in terms of performance. They are useful to manage container-based applications that are spawn along with a whole cluster. As expected, Kubernetes is the best solution also regarding performances, even if the complex architecture introduces a not negligible overhead that does not make of that a fit-for-all solution.

7 Case study: a Fault-Tolerant Cloud-based application by comparing virtualization architectures

7.1 Overview

Recent development and optimization in virtualization technologies have led to their wingspread adoption and a growing trend toward hosting workload in virtualized platforms. In fact, as virtualization promise a reduction in cost and complexity, it also raises the concerns on the availability of applications hosted on virtualized platforms. In particular, high compute instance density on a single host may have a negative impact on the availability of applications encapsulated, since those compute instances, and the services provided, will fail upon an event of host failure.

Therefore, in order to guarantee high availability [89] of such applications, there have been introduced several mechanisms like live migration and checkpoint/restore. Furthermore, such solutions were even integrated into cloud operating systems like OpenStack but the lack of distributed file system integration with LXD does not allow us to face these events. In fact, OpenStack makes use of internal services to support low-level functionalities such as live migration. That is implemented by exploiting OpenStack Cinder and so, if the file system is local, there is no possibility to move the persistent state from a server host to another.

Nevertheless, we can design an application that stores its own state in a replicated distributed storage that can be deployed behind load balancers and so, their failures are masked by redirecting the traffic to other healthy replicas. Therefore, the purpose of this chapter is to design a fault-tolerant application and analyze what is the behavior of the performance by exploiting virtual machines for the first type of replication and LXD containers for the other one.

7.2 Requirements

As already anticipated in the overview of this chapter, the aim of this application is to design a web-based service that interacts with a persistent data storage system like a relational database. The purpose is to deploy this application on a cloud infrastructure such as OpenStack. In the previous chapter, we analyzed the behavior of an OpenStack deployment between two types of hypervisor solutions: KVM and LXD. Furthermore, we found that the virtual machine-based

installation supports the integration with a distributed file system like CEPH. This is the implementation of the block storage service of the OpenStack architecture.

On the contrary, with LXD, there is no possibility to use distributed file systems because it is based on ZFS, that is a purely local file system. Therefore, the target of this chapter is to design a fault-tolerant application that is deployed atop of OpenStack, considering both KVM and LXD as possible hypervisor implementations. However, the main requirement is to guarantee fault-tolerance and so we need to design a robust architecture by comparing the performances between the virtual machine and the container-based implementation.

7.3 System Analysis

As seen in the previous chapter, we have two OpenStack deployments: one for virtual machines and the other one for LXD containers. However, the OpenStack container-based does not support the integration with a distributed file system and so we need to provide an ad-hoc solution to guarantee fault-tolerance and high-availability. In this case, as already mentioned in the overview of this chapter, there are two possible design models: one that exploits the underlying distributed file systems and the other one that makes use of a replicated distributed data storage system.

For this reason, we decided to adopt the first model for the implementation with KVM as hypervisor and the other one for the LXD-based OpenStack deployment. However, considering that it is a cloud-based application, we need to guarantee other features such as high-availability. So, we designed the solution by following a common cloud computing pattern that is focused on distributing the workload through the introduction of a load-balancer. To do that, we made use of some software applications that they will be deeply illustrated in the following sections.

7.3.1 Apache Web Server

Apache is the most widely used web server software. It is an open source software and maintained by Apache Software Foundation. Considering that it is open, it is possible to customize it to meet the needs of many different environments by including extensions and external modules. Our aim is to provide a web-based application and so we need the integration of a process that is able to exchange HTTP messages. This is called web server and Apache Web Server is used just for that.

7.3.2 Apache Tomcat

Tomcat is an application server from the Apache Server Foundation that executes Java servlets [90] and renders Web pages that include Java Server Page coding. Furthermore, Tomcat integrates an internal web server and so it can be used as either a standalone product or together with other Web servers, including Apache, Netscape Enterprise Server, and more. The requirement is to have a Java Runtime Enterprise Environment because Tomcat follows that standard.

7.3.3 Apache mod_jk

In computing, load balancing improves the distribution of workloads across multiple servers. This is aimed to maximize throughput, minimize response time and avoid overload of a single resource. Furthermore, this increases reliability and availability if the correspondent back-end system is designed with a replication grade able to guarantee redundancy. Usually, this involves dedicated software or hardware, and it differs in the strategy that is adopted to distribute the workload across the servers behind. Apache mod_jk is a software implementation of a computing load balancer that follows the Round-Robin principle by using the arithmetic mod operator.

7.3.4 MySQL RDBMS

MySQL is an open source relational database management system (RDBMS) that is based on Structured Query Language (SQL). It is cross-platform and so can run on any platform regardless if it is Linux or Windows-based. Usually, it is associated with a web-based application. In fact, it is an important component of the enterprise stack that is called LAMP (Linux, Apache, MySQL, PHP). In this case, we made use of MySQL as persistent data storage system but not following the LAMP stack considering that we use Tomcat as application server.

7.3.5 Galera Cluster for MySQL

Galera Cluster is a synchronous multi-master replication plug-in for the MySQL engine, which is called InnoDB. It varies the regular MySQL Replication and addresses a number of issues including write conflicts when writing on multiple masters, replication lag and slaves being out of sync with the master. This allows users to rely on Galera notwithstanding to know which server they can write to and which servers they can read from.

By this way, an application can write to any node, and transaction commits are then applied on all servers, via a certification-based replication [91]. This is an alternative approach to synchronous database replication using Group

Communication and transaction ordering techniques. Furthermore, a minimal Galera cluster consists of three nodes and the number of nodes should guarantee the possibility to proceed with the transaction commit in case of there is a problem with a transaction on one node. This is true because the consistency is guaranteed by a quorum mechanism that allows overcoming a number of failures.

7.4 Test Plan

This phase is aimed to provide information about the quality of the service designed. In this case, we have two solutions: one is virtual-machine based whereas the other is container-based. Furthermore, we need to evaluate other aspects instead of just performance analysis between containers and virtual machine. In fact, considering that in a solution we use a distributed file system and in the other a replicated data storage, we want to evaluate the impact of both design models on the underlying physical resources. So, we need to monitor the impact on the physical server hosts that run the deployed compute instances. To do that, we used Ganglia as a monitoring system and Apache JMeter to stress the entire web-application.

7.4.1 Apache JMeter

Apache JMeter is an open source and Java-based application that is aimed to perform various testing types like performance, load, stress, regression and functional testing [92]. This is quite important to get accurate performance metrics against our web applications. An important feature of JMeter is that it interacts with a target server by simulating a group of users. Furthermore, it subsequently collects data to calculate statistics and display performance metrics through various formats. Therefore, it was used to test our applications in order to get information about the behavior of the underlying server hosts.

7.5 Design and Implementation

The purpose of this application is to guarantee fault-tolerance and high-availability. As we have already anticipated, to ensure the fault-tolerance, we need to adopt two different design models: one that exploits a cloud-based data storage and the other that is statically built with an ad-hoc solution by using a replicated and distributed data storage. Furthermore, to guarantee high-availability we need to exploit a mechanism that is able to detect when a compute instance fails and then forward the request to another instance. Therefore, we adopted an external load-balancer that acts as the front-end of the entire web application. Moreover, we need to guarantee the fault-tolerance even if the underlying physical host

breaks down. So, we decided to deploy each compute instance on a different server host.

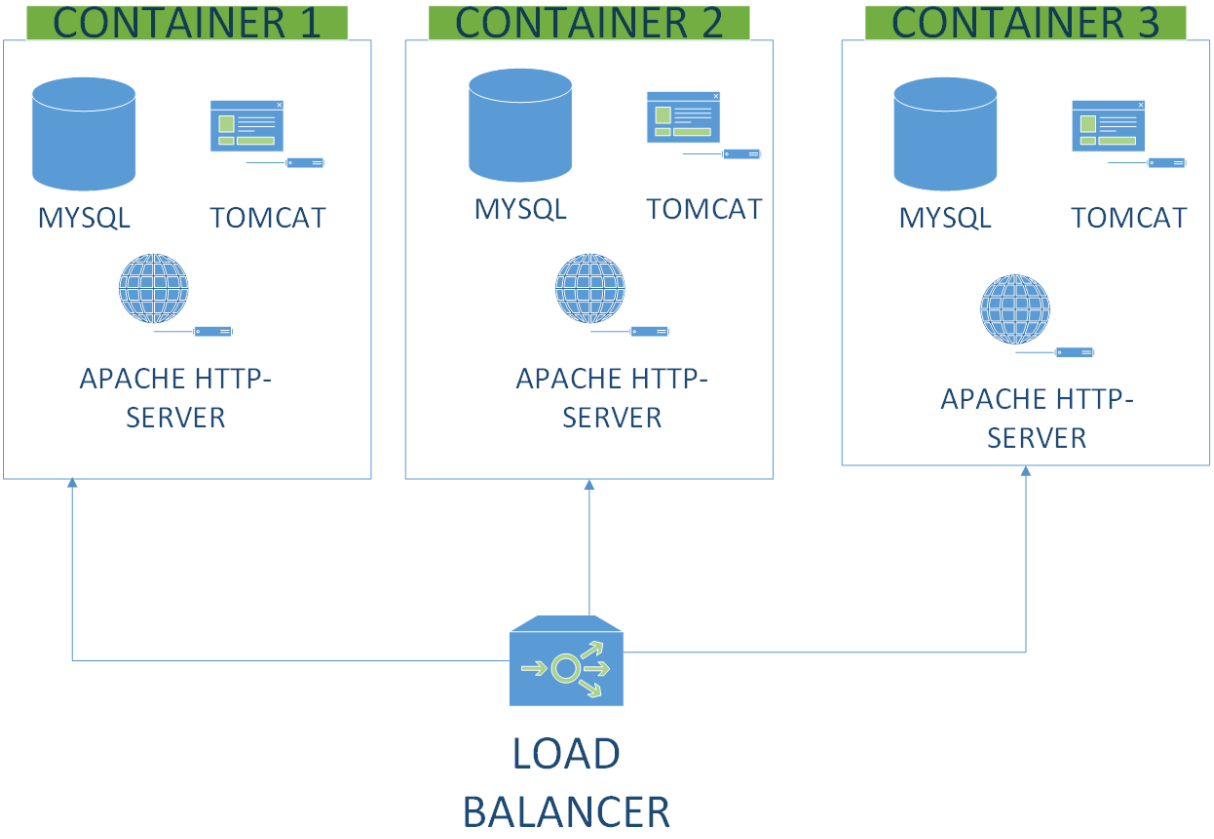

Figure 108 - The implementation of the container-based service

Figure 108 shows us the layout of the container-based web application.

As it is possible to notice, we have three containers. Each of which is deployed on a different compute node. Moreover, these are operating system containers and so they can host multiple services. In this case, we have each container that runs two processes: an application server and a database server. Furthermore, the entire database is locally stored because we do not have the support to store that in a distributed object storage. Lastly, the load-balancer is another container instance that provides the integration with the existing application servers.

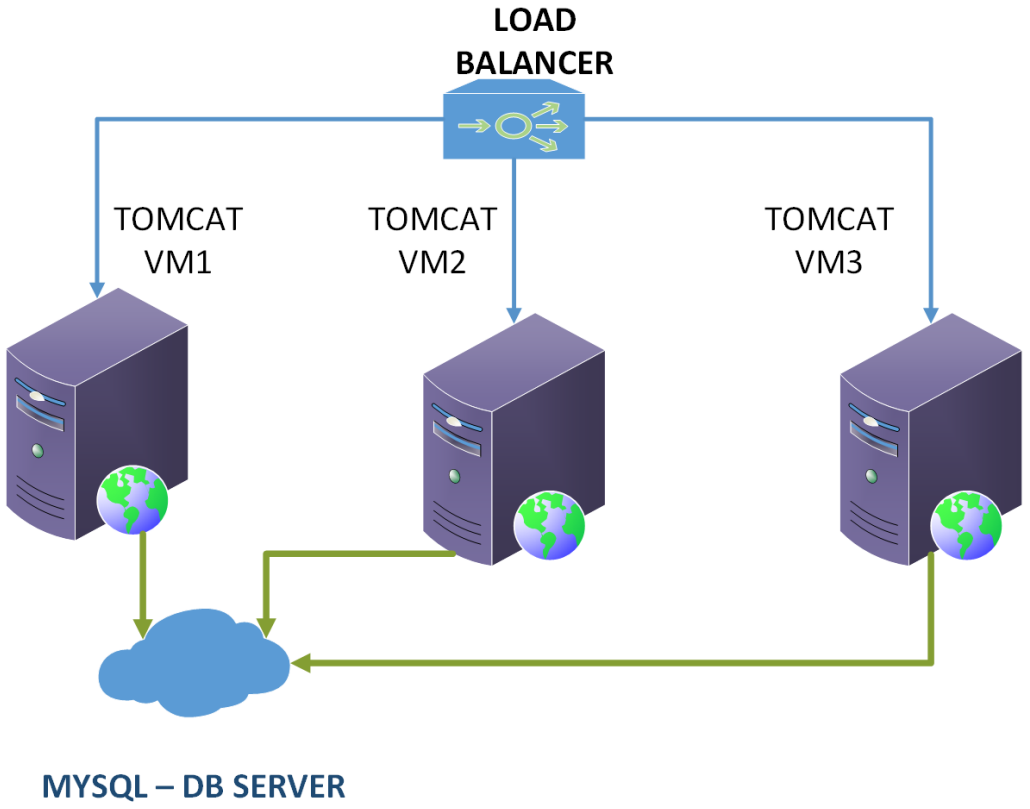

Figure 109 - The Implementation of the virtual machine-based service

Figure 109 shows us the layout of the virtual machine based implementation. As the name suggests, in this case, each compute instance consists of a single virtual machine. Moreover, also this solution consists of a deployment in different physical server host. Furthermore, as it is possible to notice, the database server is not included in each "business instance". So, it is a different virtual machine that relies on an object storage that is stored within the cloud infrastructure. This allows us to guarantee the fault-tolerant because, in case of a fault, it is just needed to migrate the database server process, considering the fact that the persistent volume is completely distributed across the OpenStack cluster.

Therefore, the purpose of the next sections is to investigate how these solutions impact the underlying physical resources by increasing the number of client requests that currently involve a different workload.

7.6 Results

The testing phase is organized by monitoring the physical server hosts that run the clustered application. In this case, we have two systems, each of which is built on a different hypervisor model. So, the aim of this work is to analyze how performances are influenced by deploying an applicative workload on both VM and container-based instances. Furthermore, considering that we adopted two

different models, we need to evaluate the behavior of these solutions whose aim is to guarantee fault-tolerance and high-availability.

So, we decided to split the test into two parts: one aimed to analyze the impact of the system on the underlying physical resources; the other one, with the purpose to understand what is the behavior of the high-level components by comparing the two alternative solutions. In fact, as demonstrated in this work, we need to evaluate if the containerization is an alternative solution also for cloud environments. Therefore, this chapter is focused on a case study that combines the analysis of infrastructure and high-level services. Nevertheless, it should be stated from the outset that the underlying infrastructure is quite different between both solutions and so we need to consider even this aspect with the analysis of the related benchmarks.

7.6.1 CPU Analysis

The idea of this benchmark is to analyze the system load. It is a measure of the amount of computational work that a computer system performs. Furthermore, for all tests, we provide a global analysis of the entire cluster. To do that, we made use of Ganglia monitoring system that provides us a global view of the monitored resources. The load average represents the average load over a period of time. Therefore, we organized each benchmark to perform an increasing number of request in a minute and then collect data by using the monitoring system.

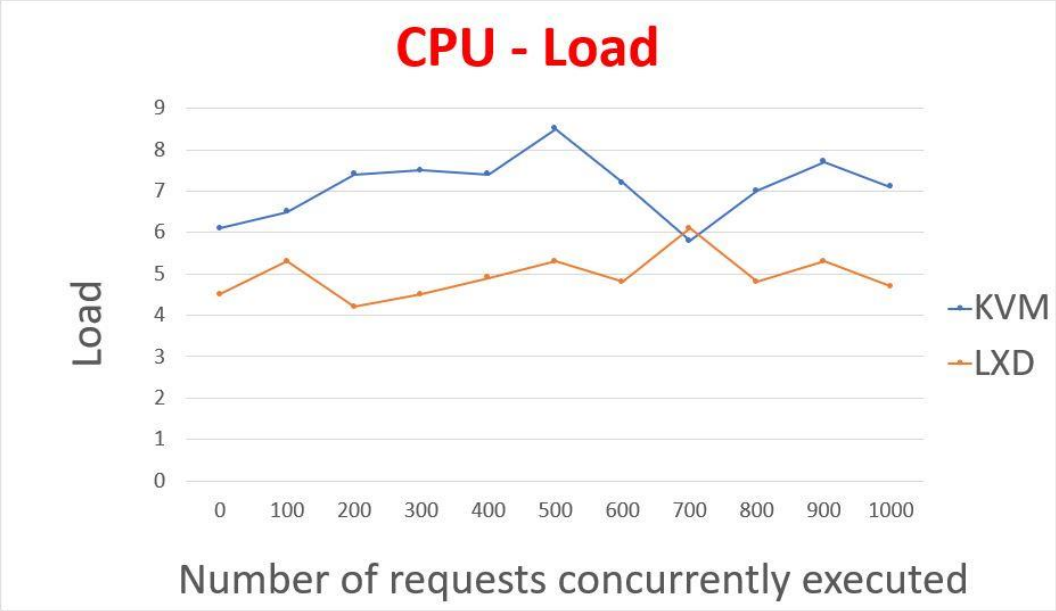

Figure 110 - Case study results: CPU Load

Figure 110 shows us the behavior of the CPU Load by stressing the related physical resource with an increasing number of requests that are executed within

a time interval of 60s. In this case, notwithstanding the fact that both clusters make use of different resources overhead, the CPU Load is greater with the implementation virtual-machine based. Furthermore, as we have already anticipated, this analysis depends also on the underlying different OpenStack implementation.

7.6.2 Memory Analysis

As seen for the CPU Load, even with system memory we want to get the behavior of the underlying physical resources by increasing the number of client requests within a time interval of 30s.

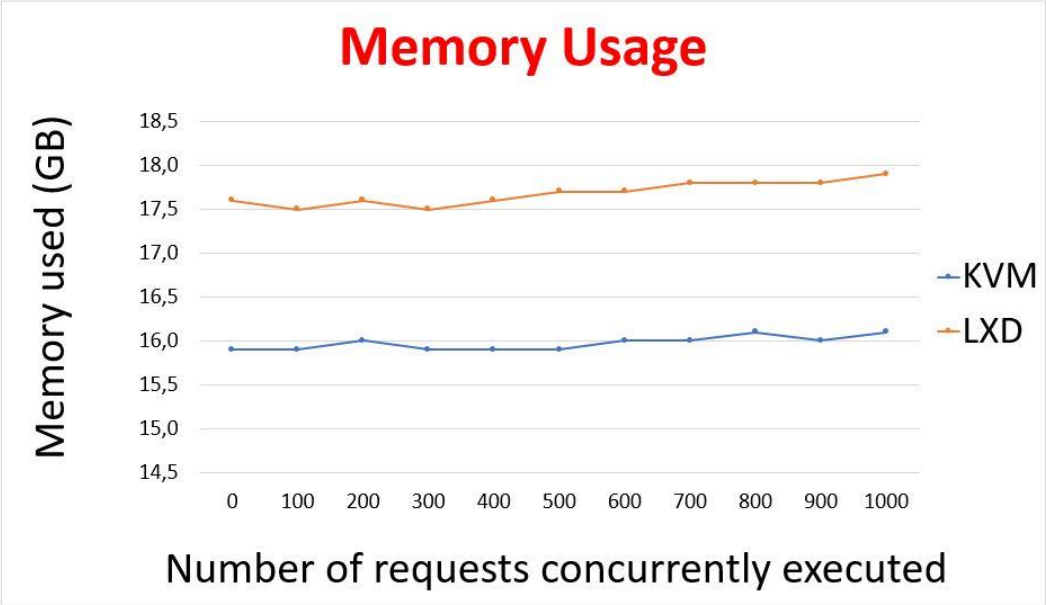

Figure 111 - Case study results: Memory Usage

Figure 111 shows us the behavior of the system memory by stressing the underlying physical resource. As it is possible to observe, this benchmark shows us that the LXD-based implementation implies a greater amount of memory that is occupied. Probably, this is quite influenced by the fact that the two implementations are different. In fact, with the LXD-based solution, we have the overhead introduced by the synchronization between the database servers and the usage of ZFS volume pools that locally includes a huge system overhead.

7.6.3 Network Analysis

The analysis of network in cluster computing is quite important. In fact, the implementation consists of a set of connected servers that work together so that, in many respects, they can be viewed as a single system. In this case, we have two solutions that both make use of several network resources. The KVM-based implementation introduces the overhead to replicate the object storage across the

compute nodes of the OpenStack cluster. On the contrary, the LXD-based solution guarantees the same requirement by including an active-active replication that is performed by three database server processes.

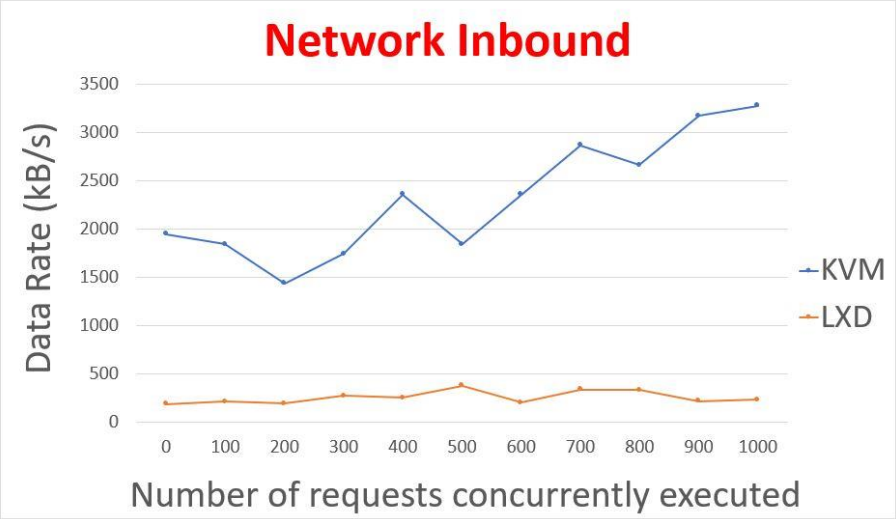

Figure 112 - Case study results: Network Inbound

Figure 112 illustrates the monitoring process of the network inbound. The KVM-based solution provides a greater data rate. As seen for memory, even in this case, an important aspect is that the LXD-based solution makes use of a different model that involves a different overhead. However, we have just seen that KVM provides better results in network analysis. In fact, the LXD network subsystem is purely implemented due to the network address translation that the underlying kernel provides.

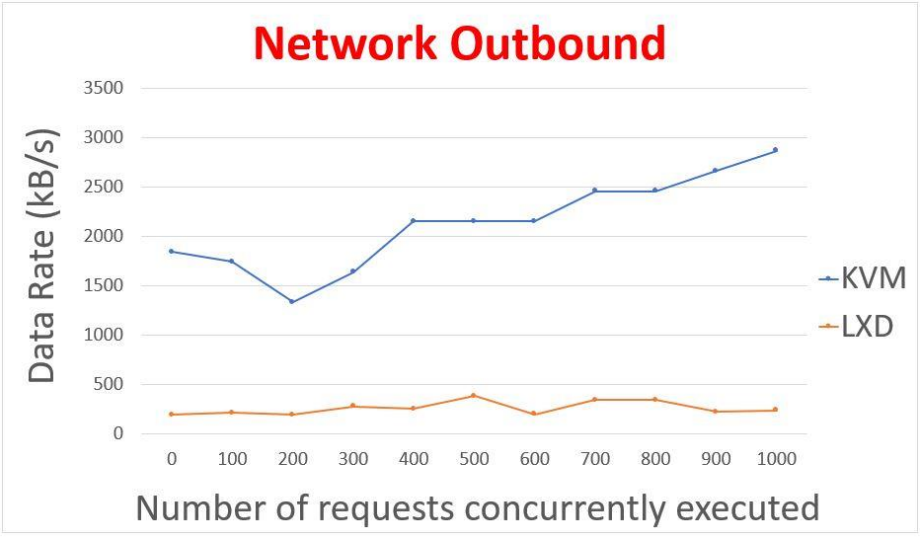

Figure 113 - Case study results: Network Outbound

Figure 113 shows us the monitoring of the outbound network. The result is the same of the inbound network: the KVM-based solution is able to achieve a greater

data rate. This demonstrates that the ad-hoc solution, that was built to guarantee fault-tolerance, is not as better as the other one for the network purpose. Furthermore, this is true due to the greater rate of competition within the underlying network subsystem. Moreover, this is also influenced by the fact that, as anticipated in the previous chapter, the two OpenStack deployments make use of a different architecture and probably, this affects also the obtained results.

7.6.4 Input/Output Database Server Analysis

The analysis of the database server behavior, with the input/output subsystem, had been the main factor that led us to build this case study. According to the LXD-state of the art, we thought that a typical solution with this container management is still aimed to rely on users' responsibilities. Therefore, we need to evaluate if an enterprise application can benefit from that or not. Usually, this involves to work with a database system and so we need to do an analysis of the execution time between a "SQL-Select" and "SQL-Insert".

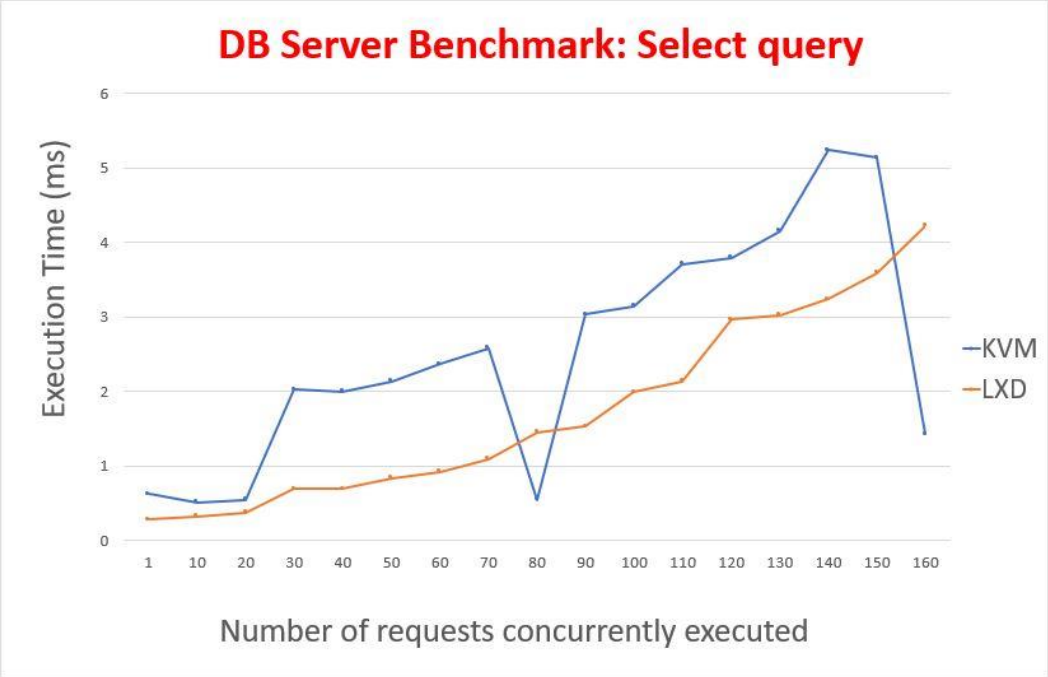

Figure 114 - Case study results: DB Server Benchmark - Select Query

Figure 114 shows us the execution time of a Select query by increasing the number of concurrent requests that are executed within a time interval of 60s. The KVM-based solution implies a greater amount of time for the most experiments. This means that considering both design models, the cloud object storage is not as performant as the local data storage with LXD containers. The distributed and replication solution that we used (Galera cluster) is optimized to guarantee read scalability. In fact, the certification based replication is applied with transactional

commit and, as demonstrated in the graphic above, the solution does not degrade the performance of the underlying physical infrastructure.

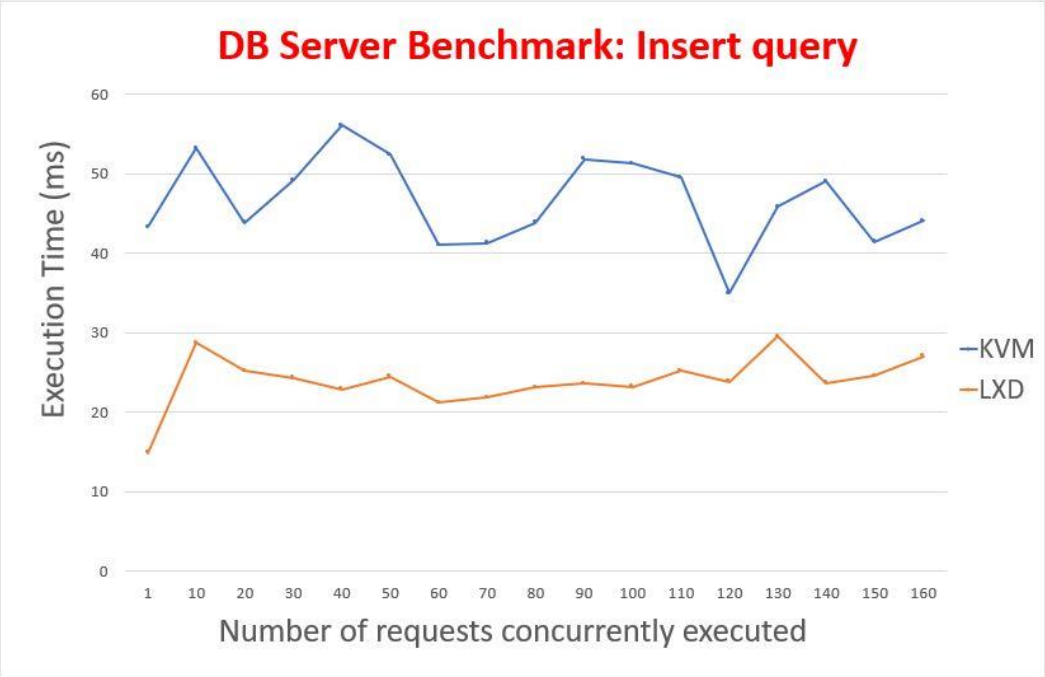

Figure 115 - Case Study results: DB Server Benchmark - Insert Query

Figure 115 shows us the results for the "SQL Insert Query". As expected, compared to the previous experiment, the execution time is greater for both solutions. Furthermore, we can notice that the KVM-based solution is not as performant as the LXD-based. This means that for the perspective of the database server, a simple insert involves much workload with the distributed cloud storage. This data was recorded due to the information collected by the local schema provided from the MySQL Engine. Therefore, they are referred to the implementation that gives us the information after completing the local input/output process.

7.6.5 Apache Web Server Benchmark

Lastly, we decided to monitor also the performances of the Apache Web Server installed in each compute instances. In fact, as seen for the execution time of MySQL queries, the sense of this experiment is to investigate the behavior of both applications from the perspective of the high-level services.

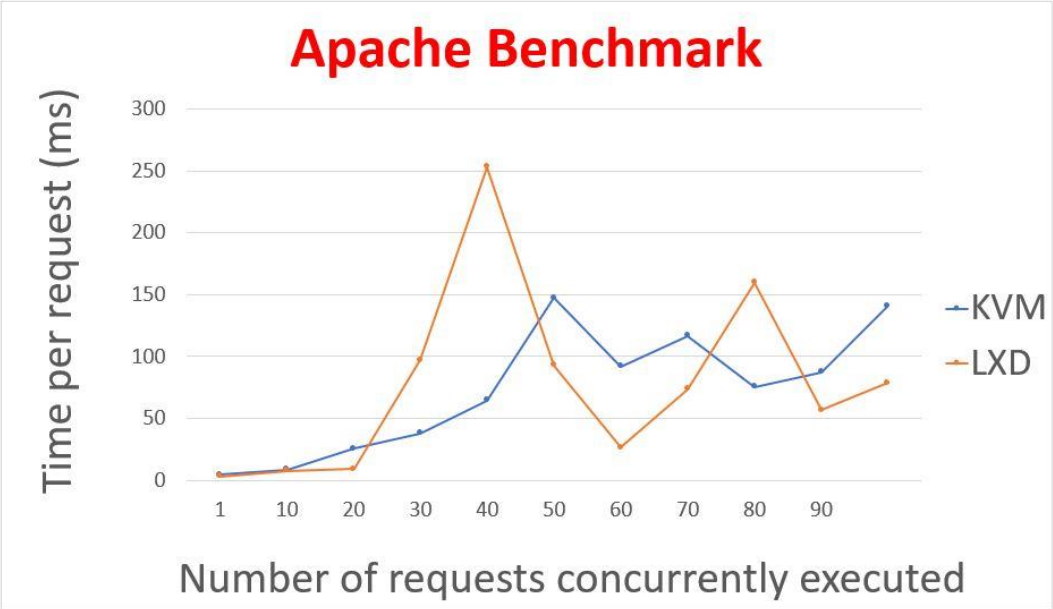

Figure 116 - Case Study results: Apache Benchmark

Figure 116 illustrates the behavior of Apache Web Server by monitoring the time needed to execute a single request notwithstanding the increasing number of requests that are executed within a time interval of 60s. As it is possible to notice, we did not get a stable information because there are experiments in which the KVM-based solution is better and other in which it is worse. However, this was important to evaluate the healthy state of the web servers and so, this test demonstrated us that with both solutions we can make use of such applicative systems.

7.7 Closing remarks

From this case study, we had another possibility to demonstrate how containers and virtual machines are complementary solutions that are suitable for cloud deployments. However, as seen with the performance analysis in the previous chapter, the containerization has not reached a mature state yet. Therefore, there are still open questions that need to be solved, and an example is the integration of LXD with a distributed file system such as CEPH. Of course, this was influenced also by the recent classification between application and operating system containers. LXD belong to the last one and so it tries to provide the same capabilities of the traditional server virtualization. Furthermore, an ad-hoc solution, as shown in this chapter, allows us to achieve the same goal even if the introduced overhead is not negligible. By this way, we have learned that several aspects are the key differentiators of the new containerization paradigm but still with other issues to be faced.

Nevertheless, this case showed us different approaches to solve a common enterprise problem with both hypervisor implementations. Surely, when OpenStack integrates LXD with the same architecture components introduced for KVM, it is quite important to repeat this experiment in order to get more accurate performance behaviors. This requires a strong refactor of the container management solution because it is based on the feature of the underlying file system. Moreover, this depends on the target proposal that can be integrated with the LXD implementation, as recently introduced with the new storage API to support the integration with a featured solution like CEPH.

Conclusions

The idea of using cloud computing as a utility is attracting companies of all sectors to adopt this environment in order to cope with a vigorously altering business environment. The possibility to maintain scalable IT infrastructures allows business agility by exploiting the principle of "pay-per-use". This lets consumers to use resources without worrying about infrastructure cost and processing power. Therefore, organizations can offload their IT infrastructure in the cloud and gain from fast scalability.

Depending on the capability that is provided to the consumer, Cloud computing has been categorized into three models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS). SaaS consists of providing to users an entire application as cloud resource, and so, the client does not have to manage low-level components like servers, storage, and network. PaaS solutions are addressed to developers because they have the control over their deployed applications, but with no need to manage the underlying infrastructure resources. This capability is offered with an Infrastructure as a Service model in which the user has the responsibility to control every application resource. At this level, the "virtual machine" model is the most prevailing technology, even if container-based solutions are challenging this traditional principle.

Today, containers are being used for two major purposes: system and application containers. The first one is aimed to run an entire system, whose use cases are effectively similar to those of virtual machines while the second one is a way of bundling and running applications in a more portable way in order to break down and isolate parts of applications.

We started analyzing how containers can be used as alternative to virtual machines, considering that the key differentiator is the minimalist nature of their deployment. Then we focused those from the application point of view. In this case, Docker is the principle actor to this phenomenon. This is also influenced by the emerging application paradigm of "microservices". Therefore, we discussed about a layered structure of the paradigm that is dependent on which capability is provided to consumers.

At the lowest level, we concentrated on system containers in order to compare how cloud instances are different if we use containers instead of virtual machines. To do that, we used two OpenStack deployments: one that makes use of LXD, to provision compute instances as containers, and the other one with KVM for virtual machines. In addition, we analyzed also the behavior of the predominant solution that is Docker.

The results showed that containers achieve better performances due to the key differentiator nature of the technology. In fact, they involve less resources overload by providing greater density and requiring less time to boot and provision a single instance. However, there are not just advantages because the containerization has not reached a mature state yet. Therefore, this does not make of the technology a fit-for-all implementation.

In addition, we focused on the application point of view in order to evaluate the development process of enterprise applications. In this case, high-availability is one of the most important purposes. So, there is the need to introduce high-level services like orchestration, and therefore we compared different container orchestrators. To conclude, Kubernetes is the richest solution in terms of functionalities, even if the complex architecture involves a not negligible overhead that does not make of Kubernetes the best solution for every deployment.

References

[1] Z. Mahmood, Cloud Computing: challenges, limitations and R&D solutions, London: Springer.

[2] VMWare, Virtualization Overview.

[3] V. Beal, "virtualization," Webopedia.

[4] S. Low, «Block level storage vs. file level storage: A comparison,» *The Enterprise Cloud,* February 2011.

[5] S. T. W. Stephen J. Bigelow, «Network virtualization explained,» [Online]. Available: http://searchitchannel.techtarget.com/feature/Network-virtualization-explained.

[6] «Primer: Multi-tenant network for the private cloud,» *TechTarget,* August 2010.

[7] Charalampos, «Performance analysis of different virtualization architectures using OpenStack,» January, 2017.

[8] M. Rouse, «Server Virtualization,» [Online]. Available: http://searchservervirtualization.techtarget.com/definition/server-virtualization.

[9] M. J. Scheepers, «Virtualization and Containerization of Application Infrastructure,» *IEEE*, 2014.

[10] C. Wolf, «Hardware-Assisted Virtualization».*Virtualization & Cloud Review.*

[11] «Paravirtualization,» [Online]. Available: http://searchservervirtualization.techtarget.com/definition/paravirtualization.

[12] «containerization,» [Online]. Available: http://searchservervirtualization.techtarget.com/definition/container-based-virtualization-operating-system-level-virtualization.

[13] T. Banerjee, «Understanding the key differences between LXC and Docker,» *Flockport*, 19 August 2014.

[14] «Linux-VServer,» [Online].

[15] [Online]. Available: https://openvz.org/Main_Page.

[16] «The Different Types of Virtualization in Cloud Computing,» *RedSwitches*, 1 June 2017.

[17] P. A. P. D. Dr. Sanjay, «School of Computing, UNF,» [Online]. Available: https://www.unf.edu/~sahuja/cloudcourse/Fullandparavirtualization.pdf. [Consultato il giorno 29 August 2017].

[18] Y. U. M. O. Tatsushi Inagaki, «Container management as emerging workload for operating systems,» *IEEE*, 26 September 2016.

[19] J. Nickoloff, in *Docker in Action*, Mannig.

[20] «rkt Container Engine,» [Online]. Available: https://coreos.com/rkt.

[21] «rkt Network Modes and Default CNI Configurations,» [Online]. Available: https://coreos.com/blog/rkt-cni-networking.html.

[22] S. Graber, «LXD 2.0: Introduction to LXD,» 11 03 2016. [Online]. Available: https://stgraber.org/2016/03/11/lxd-2-0-introduction-to-lxd-112/.

[23] S. Graber, «Insights Ubuntu,» 14 March 2016. [Online]. Available: https://insights.ubuntu.com/2016/03/14/lxd-2-0-introduction-to-lxd/.

[24] «Ubuntu Insights,» 14 February 2017. [Online]. Available: https://insights.ubuntu.com/2017/02/14/network-management-with-lxd-2-3/.

[25] «Storage management in LXD 2.15,» 12 July 2017. [Online]. Available: https://insights.ubuntu.com/2017/07/12/storage-management-in-lxd-2-15/.

[26] «8 Surprising facts about real Docker,» DataDog, April 2017. [Online].

[27] C. Company, «For CTO's: the no-nonsense way to accelerate your business with containers,» February 2017.

[28] Mesosphere, «Container Orchestration Wars,» 2016.

[29] M. L. R. Perrey, «Service-oriented architecture».*IEEE.*

[30] A. S. Prateek Sinha, «Converged infrastructure for enterprise exchange environment».*IEEE.*

[31] M. Rouse, «What is Docker Swarm?,» TechTarget, August 2016. [Online]. Available: http://searchitoperations.techtarget.com/definition/Docker-Swarm. [Consultato il giorno 30 August 2017].

[32] «Docker Swarm,» [Online]. Available: https://docs.docker.com/v1.9/swarm/.

[33] M. Luksa, Kubernetes in Action, Manning.

[34] J. Ellingwood, «An Introduction to Kubernetes,» Digital Ocean, 14 October 2016. [Online]. Available: https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes. [Consultato il giorno 30 August 2017].

[35] «The Kubernetes API - Kubernetes,» Kubernetes, [Online]. Available: https://kubernetes.io/docs/concepts/overview/kubernetes-api/.

[36] «Cluster Networking,» [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking.

[37] «Kubernetes,» [Online]. Available: https://kubernetes.io/docs/admin/kube-scheduler/.

[38] A. Lichtigstein, «DevOps Kubernetes vs. Docker Swarm vs. Apache Mesos: Container Orchestration Comparison,» loom, 19 June 2017. [Online]. Available: https://www.loomsystems.com/blog/single-post/2017/06/19/kubernetes-vs-docker-swarm-vs-apache-mesos-container-orchestration-comparison.

[39] «Mesosphere,» [Online]. Available: https://docs.mesosphere.com/1.9/networking/.

[40] R. Howard, «Orchestration With Kubernetes, Docker Swarm, and Mesos,» Dzone, July 2017. [Online]. Available: https://dzone.com/articles/orchestration-with-kubernetes-docker-swarm-and-mesos.

[41] R. Ignazio, Mesos in action, Manning.

[42] «Mesosphere,» [Online]. Available: https://docs.mesosphere.com/1.9/storage/.

[43] «Mesosphere,» [Online]. Available: https://docs.mesosphere.com/1.9/overview/architecture/.

[44] R. LABS, «Easily deploy and scale Kubernetes with Rancher».

[45] M. Pais, «RancerOS: A minimal OS for Docker in Production,» *Infoq,* 31 Mar 2015.

[46] «Rancher,» [Online]. Available: http://rancher.com/docs/rancher/v1.6/en/rancher-services/scheduler/.

[47] «Rancher,» [Online]. Available: http://rancher.com/docs/os/v1.0/en/.

[48] «Rancher Labs,» [Online]. Available: http://rancher.com/cattle-swarm-kubernetes-side-side/.

[49] «Kubernetes Vs Amazon ECS,» *Platform9,* 20 July 2017.

[50] B. C. a. T. Jones, «Docker on AWS,» 2015.

[51] «Kontena,» [Online]. Available: https://www.kontena.io/docs/core-concepts/architecture.html.

[52] «What are WebSockets?,» Pusher, [Online]. Available: https://pusher.com/websockets.

[53] «Kontena,» [Online]. Available: https://www.kontena.io/docs/using-kontena/volumes.html.

[54] «Docker Orchestrating and Scheduling,» [Online]. Available: http://blog.kontena.io/docker-orchestrating-and-scheduling/.

[55] K. Team, «Choosing the Right Containerization and Cluster Management Tool,» Kublr, [Online]. Available: https://blog.kublr.com/choosing-the-right-containerization-and-cluster-management-tool-fdfcec5700df.

[56] HashiCorp, «Architecture - Nomad by HashiCorp,» HashiCorp, [Online]. Available: https://www.nomadproject.io/docs/internals/architecture.html.

[57] S. Hall, «RISE OF THE CONTAINER-FOCUSED OPERATING SYSTEMS,» *TheNewStack* , 27 January 2016.

[58] Kapal, «MicroOses for containers,» [Online]. Available: https://kalpacg.wordpress.com/2016/09/03/microoses-for-containers/.

[59] M. Bailey, «CoreOS in action,» Manning, 2017.

[60] «Rise of the container-focused operating systems,» *The New Stack,* 27 January 2016.

[61] C. Ward, «An Introduction to CoreOS,» *CODESHIP,* 5 5 2017.

[62] R. Mocevicius, «DEIS,» [Online]. Available: https://deis.com/blog/2016/coreos-overview-p1/.

[63] «ProjectAtomic,» [Online]. Available: http://www.projectatomic.io/docs/gettingstarted/.

[64] «Atomic,» [Online]. Available: http://www.projectatomic.io/docs/atomic-host-networking/.

[65] «RedHat,» [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/managing_containers/managing_storage_with_docker_formatted_containers.

[66] RedHat, «CHAPTER 3. THE XFS FILE SYSTEM,» RedHat, [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/ch-xfs.

[67] J. Rosland, «Container OS comparison,» *CodeShip,* 16 6 2017.

[68] J. Harper, «Enterprise Architecture for the Internet of Things: Containerization and Microservices,» *AnalyticsWeek,* 27 February 2017.

[69] E. Brown, «The Future of IoT: Containers Aim to Solve Security Crisis,» *LiNUX.COM,* 10 November 2016.

[70] P. C. S. D. J. G. B. H. J. J. A. M. A. O. C. P. B. E. Kathy Cacciatore, «Exploring Opportunities: Containers and OpenStack,» 2015. [Online]. Available: https://www.openstack.org/assets/pdf-downloads/Containers-and-OpenStack.pdf.

[71] «Zun - OpenStack,» OpenStack, [Online]. Available: https://wiki.openstack.org/wiki/Zun.

[72] [Online]. Available: https://openstack.nimeyo.com/89062/openstack-dev-zun-higgins-the-design-of-zun.

[73] Rahul, «OpenStack and Container (project Kolla) | A new beginning,» 20 September 2016. [Online]. Available: https://rahulait.wordpress.com/2016/09/20/openstack-and-containers-project-kolla/.

[74] «Architecture - Murano,» [Online]. Available: https://murano.readthedocs.io/en/stable-liberty/intro/architecture.html.

[75] «Configuration - Murano,» [Online]. Available: https://murano.readthedocs.io/en/stable-liberty/administrator-guide/configuration.html#network-configuration.

[76] «Magnum vs Murano: An OpenStack container strategy,» [Online]. Available: https://www.mirantis.com/blog/magnum-vs-murano-openstack-container-strategy/.

[77] Intel, «Virtualization and Cloud Computing,» August 2013. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/guides/cloud-computing-virtualization-building-private-iaas-guide.pdf.

[78] R. K. L. K. R. R. D. G. Rabindra K. Barik, «Performance Analysis of Virtual Machines and».*IEEE*.

[79] «Chef - Automate IT Infrastructure,» [Online]. Available: https://www.chef.io/chef/.

[80] J. D. l. R. M. W. Kent Baxley, «Deploying workloads with Juju and MAAS in Ubuntu 14.04 LTS».

[81] «What is Juju?,» [Online]. Available: https://jujucharms.com/docs/2.0/about-juju.

[82] «How Puppet works,» Puppet, [Online]. Available: https://puppet.com/products/how-puppet-works.

[83] «OpenStack base | Juju,» Juju, [Online]. Available: https://jujucharms.com/openstack-base/.

[84] «OpenStack LXD | JUJU,» Juju, [Online]. Available: https://jujucharms.com/u/openstack-charmers-next/openstack-lxd/.

[85] «Introduction to Ganglia on Ubuntu 14.04,» DigitalOcean, [Online]. Available: https://www.digitalocean.com/community/tutorials/introduction-to-ganglia-on-ubuntu-14-04.

[86] D. A. G. S. E.Jebamalar Leavline, «Hardware Implementation of LZMA Data Compression Algorithm».*International Journal of Applied Information Systems (IJAIS), Foundation of Computer Science FCS, NewYork, USA.*

[87] K. Tolly, «How to use Iperf to measure throughput,» [Online]. Available: http://searchenterprisewan.techtarget.com/tip/How-to-use-iPerf-to-measure-throughput. [Consultato il giorno 24 September 2017].

[88] J. Coyle, «Benchmark disk IO with DD and Bonnie++,» 11 September 2013. [Online]. Available: https://www.jamescoyle.net/how-to/599-benchmark-disk-io-with-dd-and-bonnie?v=cd32106bcb6d.

[89] E. R. E. M. C. Wubin Li and Ali Kanso, «Comparing Containers versus Virtual Machines for,» *IEEE*, 2015.

[90] M. Rouse, «What is Tomcat».*TheServerSide.*

[91] «Galera Cluster for MySQL - Tutorial,» Serveral nines, [Online]. Available: https://severalnines.com/resources/tutorials/galera-cluster-mysql-tutorial.

[92] TOOLSQA, «What is Apache JMeter,» [Online]. Available: http://toolsqa.com/jmeter/what-is-apache-jmeter/.

[93] V. Beal, «operating system-level virtualization,» [Online]. Available: http://www.webopedia.com/TERM/O/operating_system_level_virtualization.html.

[94] «Full Virtualization,» [Online].

[95] H. Jain, 25 November 2016. [Online]. Available: https://www.sumologic.com/blog/code/lxc-lxd-explaining-linux-containers/.

[96] «rkt vs Other Projects,» [Online]. Available: https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html#process-model.

[97] R. Mocevicius. [Online].

[98] M. Hoogendoorn, «Kontena: an alternative container orchestrator,» [Online]. Available: http://container-solutions.com/kontena-alternative-container-orchestrator/.

[99] «What is Amazon EC2 Container Service,» Amazon Web Services, [Online]. Available: http://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html.

[100] «Architecture - Nomad by HashiCorp,» HashiCorp, [Online]. Available: https://www.nomadproject.io/docs/internals/architecture.html.

[101] D. S. A. @. M. I. Karl Isenberg, Mesosphere, 23 June 2016. [Online]. Available:

https://www.slideshare.net/KarlIsenberg/container-orchestration-wars.

[102] G. Amorim, «The Importance of SOA to Cloud Computing,» *Service Technology Magazine*, 2 December 2014.

[103] G. Nag, «Operating System Containers vs. Application Containers,» *RisingStack*, 2015.