2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems

Container-based orchestration in cloud: state of the art and challenges

Andrea Tosatto
Istituto Superiore Mario Boella
Torino, Italy
andrea.tosy@gmail.com

Pietro Ruiu
Istituto Superiore Mario Boella
Torino, Italy
ruiu@ismb.it

Antonio Attanasio
Istituto Superiore Mario Boella
Torino, Italy
attanasio@ismb.it

***Abstract*—How to effectively manage increasingly complex enterprise computing environments is one of the hardest challenges that most organizations have to face in the era of cloud computing, big data and IoT. Advanced automation and orchestration systems are the most valuable solutions helping IT staff to handle large-scale cloud data centers. Containers are the new revolution in the cloud computing world, they are more lightweight than VMs, and can radically decrease both the start-up time of instances and the processing and storage overhead with respect to traditional VMs.**

The aim of this paper is to provide a comprehensive description of cloud orchestration approaches with containers, analyzing current research efforts, existing solutions and presenting issues and challenges facing this topic.

Keywords - cloud computing, orchestration, system-level virtualization, containers

I. Introduction

In the Cloud Computing era, the challenges of Cloud management organizations around the world are moving beyond traditional client-server architectures toward much more dynamic environments enabled by Cloud Computing, Big Data and Internet of Things (IoT). Business survey research indicates that IT decision makers in North America expect costs of renting and managing Cloud resources (public and private) will consume approximately half of their budgets by 2016 [1].

Automation and orchestration solutions will be more and more critical to the effective management of large-scale Cloud data centers, avoiding IT teams to struggle with rapidly changing business needs, costs and service-level agreements.

Cloud orchestration consists in coordinating, at software and hardware layer, the deployment of a set of virtualized services in order to fulfill operational and quality objectives of end users and Cloud providers. Orchestration in the Cloud poses still competitive challenges because of the scalability, resources heterogeneity and other constraints related to the environment itself [2]–[4].

Hardware virtualization has wider scope of usage but poor performances [5], thus in the last year system-level virtualization is gaining in popularity in the Cloud Computing community.

The next sections of the paper are structured as follows. Section II discusses about Cloud orchestration approaches and gives an overview of recent research on this topic. Section III is dedicated to system-level virtualization, compared to traditional virtualization approaches (i.e. hardware virtualization) and provides some details about existing solutions. Challenges and issues of the approaches and solutions examined in the paper are analyzed in Section V. Finally, in Section VI, conclusions and consideratons are detailed.

II. Cloud orchestration

Cloud orchestration solutions can be implemented following different approaches. A very intuitive classification can be made considering the type of resources to be orchestrated: software services or hardware services [6].

1) *Orchestration of software services (SaaSO)* orchestrates services as executable business process that interacts with other internal and external SaaS services. These services are also associated to the physical resources on which they will run, thus systems for the management of interrelated components are included.
2) *Orchestration of hardware services (IaaSO)* consists in the management of computing resources (i.e., memory, CPU, storage, network) in order to perform user requests or to satisfy operational needs of the cloud provider. Orchestration functions enable the costs reduction by consolidating physical resources and the fulfillment of Service Level Agreements (SLAs) by dynamically reallocating workloads and resources.

The requirements to deliver ongoing value in the Cloud are very heterogeneous and restricitive, since organizations need to be able to deploy critical workloads and services, to scale resources on the fly, to quickly react to changing demands, to operate interoperable solutions between different clouds or resources and standardized deployment configurations.

A. State of the art

An overview of most relevant research about cloud orchestration is provided in the following lines. Many of these works are related to the use of descriptor files and DevOps solutions that are typically used in conjunction with Cloud Computing to automatically provision and manage (orchestrate) underlying resources such as storage or virtual machines. In [7] the authors propose a classification of DevOps artifacts, which are firstly split in two major classes: Node-Centric Artifacts (NCA) are limited to a single node, while Environment-Centric Artifacts (ECA) can potentially manage multiple nodes and their connections. ECAs typically use and orchestrate

978-1-4799-8870-9/15 $31.00 © 2015 IEEE
DOI 10.1109/CISIS.2015.35

NCAs. Artifacts belonging to both categories can be further characterized, according to other aspects:

- Level of dependencies: whether they depend on a specific service provider (provider-dependent) or on a certain tooling (tooling-dependent)
- Hypervisor-based (e.g. Amazon) vs container-based (e.g. Docker) virtualization
- For ECAs, Infrastructure-centric artifacts (e.g. AWS CloudFormation) focus on the infrastructural resources, while application-centric artifacts (e.g. Juju) focus on the configuration of middleware and application components.
- Definition-based artifacts (e.g. Chef, Puppet) define the configuration of resources while image based artifacts (e.g. Docker) capture and restore a persisted state of a resource.
- Definition-based artifacts can use declarative (e.g. Chef) or imperative (e.g. Unix Shell scripts) language or a combination of the two.

The study of [8] focuses more on the orchestration of dynamic application topologies, defined as "specifications of how distributed services are to be deployed and interconnected on any selected execution environment". More specifically, a tuple is associated to the topology which includes: the set of distributed services composing the application; the relationships among services; the logical regions mapped to resource pools at different datacenters; inter-region relationships with connectivity information and the set of rules to manipulate the state of services. Each rule is specified by a validation function, that contains a guaranteed state to be maintained. A guaranteed state can refer, for instance, to the level of availability, capability, efficiency, stability or security of the whole application or of a limited service/region.
The preliminary definition of SLAs fixes the set of guaranteed states of the topology. To evaluate the validity of these states, metrics are defined about user requests, application services state and resources utilization. If a validation function fails, an associated guaranteed action is executed. For instance, additional services may be started to redistribute the load, the metric of a service can be updated to change its state, resources can be scaled up and services can be mirrored or migrated to a different region to increase the availability. The most interesting consideration is that within this framework, the orchestration becomes a continuous process driven by the monitored metrics and the specified SLAs.
[9] describes an orchestration approach that follows the idea of converging toward a desired state of a resource (e.g., a middleware component deployed on a virtual machine) achieved by repeatedly executing idempotent scripts. Because of major drawbacks, the authors present an alternative solution based on compensation and fine-grained snapshots using container virtualization.
The main idea of compensation is to implement an undo strategy that is run in case an error occurs during the execution of a particular script or action. Depending on the level of implementing compensation, either compensation scripts can be implemented to roll back the work performed by a particular script or compensation actions can be implemented to undo a single action.
DevOps and containers communities provide techniques and tools to implement convergent deployment automation: its foundation is the implementation of idempotent scripts, meaning the script execution on a particular resource such as a virtualized instance can be repeated arbitrarily.
The authors discuss how to implement compensation on the level of scripts and on the level of individual actions. Moreover, they show how action-level compensation can be achieved using fine-grained snapshots to minimize the effort of implementing custom compensation actions. Docker is used as a container virtualization solution and Dockerfiles (construction plans for Docker containers) as scripts that can be compensated based on fine-grained container snapshots.
In the context of scientific workflow platforms, one of the main issues concerning software deployment is the use of heterogeneous applications with different, and sometimes incompatible, external packages dependencies. Moreover, updates and installations of software become difficult for unskilled users, especially when a specific version of a tool is required to exactly reproduce the same experimental results. Thus, the effective isolation of each applications runtime environment becomes a challenging issue. As stated by [10] this problem can be partially solved with the use of customized Virtual Machines, created from pre-existing images, that are configured in advance with the required tools and dependencies. However, virtualization with hypervisors introduces a not negligible overhead on resources utilization [5] and also the boot process of VMs affects the overall latency of the system. This issue becomes critical when considering dynamic allocation of computational resources based on the workload, with existing virtual VMs that have to be accordingly resized and rebooted. For these reasons, the use of containers seems to solve software deployment problems in scientific workflow platforms. The solution selected by the authors is based on the lightweight container-based virtualization. In particular, Docker containers are used by working nodes that are responsible for the execution of workflow tasks. A separate container is used for each application and Docker images are stored in the local storage system. When a working node has to perform a task, it downloads the corresponding Docker image and starts the container. In the proposed architecture, also the job scheduler on each working node runs on a separate container and can access the Docker APIs through socket.

III. System-level virtualization (Containers)

Orchestration solutions need to interact with all the stacks of an IT infrastructure from cloud services to data center resources. For this reason orchestration systems need highly elastic and scalable infrastructures that allow the dynamic allocation of different resources (such as compute, storage, networking, software or a service) in the right location and in short times, enabling the deployment of applications.
The elasticity in cloud environments is obtained abstracting

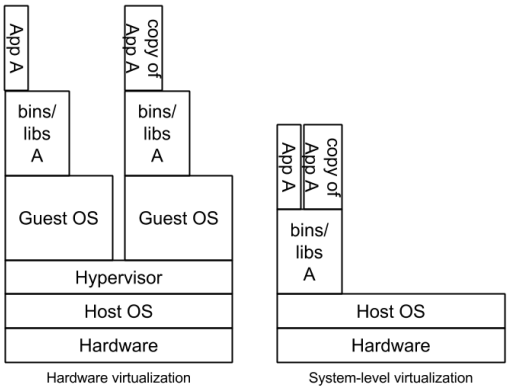

Fig. 1. Comparison of containers vs VMs footprint on the host system

physical resources form underlying layer by means of virtualization. There are different virtualization technologies, but the two most relevant in the cloud computing landscape are Hardware virtualization and System-level virtualization.

- *Hardware virtualization* (aka Hypervisors) abstracts the underlying hardware layers to enable complete operating systems to run inside the hypervisor as if they were an application. Paravirtualization solutions (Xen) and hardware virtualization solutions (KVM), in combination with hardware specific support integrated in modern CPU (Intel VT-x and AMD-V), can achieve low level of overhead due to the new layer added between the virtual instance and the hardware.
- *System-level virtualization* (aka Containers) is based on fast and lightweight process virtualization and allows to tie up an entire application with its dependencies in a virtual container that can run on every Linux distribution. It provides its users an environment as close as possible to a standard Linux distribution. Due to the fact that containers are more lightweight than VMs (see Fig. 1), the same host can achieve higher densities with containers than with VMs. This approach has radically decreased both the start-up time of instances and the processing and storage overhead, that are typical drawbacks of Hypervisor-based virtualization [11].

The two approaches differ in complexity of implementation, breadth of OS support, performance in comparison with standalone server, and level of access to common resources. Hardware virtualization has wider scope of usage but poorer performances; containers provide the best performance and scalability, are usually also much simpler to be managed as all of them can be accessed and administered from the host system [12].

Currently, most of the several different implementations of system-level virtualization relies mainly of two kernel features, namespaces and cgroups:

- *namespaces* provide per-process isolation of the operating system resources, in other words create barriers between processes. There are six namespaces, each covering a different resource: pid, net, ipc, mnt, uts and user;
- *cgroups* is a kernel feature that isolates resource usage and provides resource management and accounting. The resources that can be controlled through this feature are memory, CPU and block I/O.

Neither namespaces nor cgroups intervene in critical paths of the kernel, and thus, in general, they do not incur a high performance penalty. The only exception is the memory cgroup, which can incur significant overhead under some workloads [11].

A. Existing solutions

All container-based systems have a near-native performance of CPU, memory, disk and network, but there are more differences in the resource management implementation: some projects implement their own capabilities beyond standard Linux features, introducing more resource limits, such as the number of processes, which give more security to the whole system [13]. Below a list of some of the most known container-based systems:

- *OpenVZ* is an open-source solution used for providing hosting and cloud services, and it is the basis of the Parallels Cloud Server. It is based on a modified Linux kernel. In addition, it has command-line tools (primarily vzctl) for management of containers, and it makes use of templates to create containers for various Linux distributions. OpenVZ also can run on some unmodified kernels, but with a reduced feature set [12].
- *LXC (LinuX Containers)* project provides a set of userspace tools and utilities to manage Linux containers. As opposed to OpenVZ, it runs on an unmodified kernel. LXC is fully written in userspace and supports bindings in other programming languages like Python and Go. Current LXC uses different kernel features to control processes, such as Kernel namespaces (ipc, uts, mount, pid, network and user), Apparmor and SELinux profiles, Seccomp policies, Chroots, Kernel capabilities, CGroups. LXC containers are often considered as something in the middle between a chroot and a fully-fledged virtual machine. The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel. It is available in the most popular distributions. It is also possible to run Linux containers on architectures other than x86, such as ARM [14].
- *Linux-VServer* is a soft partitioning concept based on Security Contexts which permits the creation of independent virtual instances that run simultaneously on a single physical server. The guest runs on an almost identical operating environment as a conventional Linux server. Each Virtual Private Server (VPS) has its own user account database and root password and is isolated from other virtual servers, except for the fact that they share the same hardware resources. The basic concept of the Linux-VServer solution is to separate the user-space

environment into distinct units in such a way that each VPS looks and feels like a real server to the processes contained within. Linux-VServer uses a patched kernel to allow several distributions to run simultaneously on a single, shared kernel, without direct access to the hardware, and to share the resources in a very efficient way [15].

IV. Docker, container-based orchestrator

Docker [16] is one of the most popular Cloud orchestrators for system-level virtualization solutions, that uses Linux containers as a way of isolating data and computing on shared resources and facilitates the deployment of applications between different Linux-based platforms. It is not only a tool for controlling and isolating processes but provides other services that allow to quickly build or assemble applications and to easily distribute them between Docker hosts. Docker solved the issues of portability and consistency between environments. Portability in Docker is not represented by the possibility to migrate VMs or OSs but it makes possible to ship only the code of the application.

The main method used by Docker to interact with the underlying OS is the LXC driver, but recently a new built-in execution driver has been introduced. This driver is based on libcontainer, a pure Go library to access the kernels container APIs directly, without any other dependencies. Furthermore Docker includes an execution driver API which can be used to customize the execution environment surrounding each container, allowing to take advantage of the numerous isolation tools available (such as OpenVZ, systemd-nspawn, libvirt-lxc, libvirt-sandbox, qemu/kvm, BSD Jails, Solaris Zones and chroot).

Docker is composed by three main components: the Docker daemon, the Docker client, and the Docker.io registry.

The Docker client is the primary interface of the system, it accepts commands from the user and communicates with a Docker daemon to manage the container lifecycle on any host. The Docker daemon runs on the host machine and controls resources. The Docker.io registry is the global archive of container images.

In Docker each container is based on an image and a set of configuration data. Images are static snapshots of the containers' configuration. Images are read only and incremental, meaning that all changes can be made only in top-most writable layer and that each image depends on one or more parent images. Every time a container is launched from an image, instead of changing the file system to read-write mode as normal Linux boot, Docker uses union mount to add a read-write file system over the base image. If the container is committed (using commit command) the changes are stored adding a new image layer. The Dockerfile is the file that contains simple instructions that build Docker images.

When a Docker client tells the Docker daemon to run a container, the latter checks for the presence of the required image and if it doesn't exist locally on the host, it is downloaded from the public registry (Fig. 2). Once Docker has the image

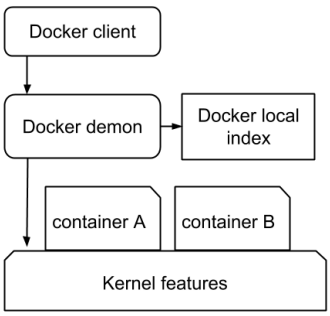

Fig. 2. Docker overall architecture

it creates a container, allocates a filesystem and mounts a read-write layer. Finally it creates a network interface for interacting with the local host and assigns an available IP address from an existing pool. As soon as the environment is ready, Docker can run the application and logs standard inputs, outputs and errors to monitor the application behaviour.

Docker relies heavily on various features in the kernel for isolating resources but also for efficiently storing containers images on filesystem. One of the strengths of Docker is the ability to handle containers filesystems as a set of writables layers where to put changes to original images. This allows to limit the overhead and the size of new containers. Docker can use different storage backends, but the most adopted is Advanced Multi Layered Unification Filesystem (AUFS). AUFS is a stackable unification filesystem, which unifies several directories and provides a merged single directory [17]. It is based on the concept of Unionfs, but while keeping the basic features it was entirely re-implemented. The backend represents each layer of the container as an regular directory, adding metadata for all files that are unique for that layer, and keeping tracks about changes with respect to the previous layer.

V. Challenges and issues

This section explores the main challenges faced by the currently available container orchestrators and some of the still open issues in the widespread adoption of such technologies in production environments.

Services containerization enhances the abstraction of the application execution enviroment with respect to the actual system on which the software is running. The only constraint involves the hosts operating system that, as explained in Section III, has to be Linux. One of the main benefits in the adoption of system-level virtualization technologies, is that containerized services are portable: the whole application environment, in fact, is persisted into container images that can be easily snapshotted and restored. CoreOS [18], a Linux distribution that natively supports Docker, integrates an orchestrator, called fleet, that offers a full support to containers mobility. Relying on etcd, a distributed key-value store, fleet supports live container migration from one host to another

and the possibility to modify at runtime the environment of running containers.

Lightweight-virtualization is also very useful to manage very large mission critical distribuited systems. As an example Google uses lmctfy [19] and Docker to run F1 [20], a distribuited relational database used to store AdWords data. Using containers, distribuited systems can scale up and down according to their load more rapidly than with traditional VMs. Moreover, some system-level virtualization orchestrator such as Google Kubernetes [21] adopt a mixed approach to the virtualization that allows to scale both the computing resources available to containers and the number containers available to applications. In that way, application can rapidely scale up and down according to their needs. In the Kubernetes jargon, servers providing computing resources are called minionis and the containerized services are called pods. Kubernets automatically takes care of scaling minions provisioning new VMs or bare metal hosts everytime is needed to scale up pods.

System-level virtualization technologies are the natural companion of Service Oriented Architectures (SOA). Creating development or staging environments for SOA based systems could be a really hard work because of the application stack heterogeneity and the complex interactions between services. Containerization technologies offer native tools to support both these SOA challenges. In Docker, for example, the services stack can be described in Dockerfiles and the interactions could be managed using the link primitive. Furthermore, Docker offers an orchestration solution, called Docker Compose [22], that is very suitable for the provisioning of testing environments. Using that tool, developers can specify services dependecies and topologies and provision the whole infrastructure with a simple command.

As we have seen, container orchestration technologies address many challenges thrown by modern software systems. Although these technologies are really young, many large IT companies are successfully using them in production, letting them handle their workflow. Currently many off-the-shelf different solutions are available as open-source software but what we observe is that mostly every company is developing its own custom orchestration solution to address its specific use cases. The result of this approach is a very fragmented offer of container orchestrators that makes difficult to new small IT companies to deploy them in production. For these reasons a lot of work must be done in the standardization of both system-level virtualization technologies and orchestrators.

Furthermore, the container monitoring challenge is still unaddressed by most of the available orchestration solutions. IT companies are used to monitor their services with systems like Zabbix os Nagios. Even if system-level virtualization relies on cGroups, monitoring systems do not yet integrate with containers. The currently available solutions such as Google cAdvisor [23] or RedHat ProjectAtomic [24] are far from the stability. As a result, it is very difficult for small IT companies to find an effective solution to monitor containers, limiting the adoption of such technologies in mission critical production services.

VI. Conclusions

As we have discussed in Section V, although containers are a very recent technology they are widespreading among large IT companies beacause they allow an effective management of distrbuted software systems. The lack of standardization is currently preventing smaller companies from the provisioning of containers in production because they can't be easily monitored using the standard monitoring services.

Since many different technologies are actually being used, a joint standardization effort is of primary importance to achieve interoperability. As an example, the App Container Specification [25] proposed by CoreOS tries to define a standard image format, runtime, and discovery protocol for running applications in containers. This could also help to better comprehend the containerization technologies, promoting the development of new missing features such as monitoring.

References

[1] "Orchestration simplifies and streamlines virtual and cloud data center management," *IDC Technology Spotlight*, Jan. 2013. [Online]. Available: http://public.dhe.ibm.com/common/ssi/ecm/en/til14057usen/TIL14057USEN.PDF

[2] D. Griffin, "Report on the public consultation for h2020 wp 2016-17: Cloud computing and software," European Commission, Tech. Rep., Dec. 2014. [Online]. Available: http://ec.europa.eu/information_society/newsroom/cf/dae/document.cfm?doc_id=8161

[3] J. Hodges, "Nfv: What exactly can be virtualized?" Light Reading University, Tech. Rep., 2013.

[4] "Network functions virtualisation (nfv) - network operator perspectives on industry progress," ETSI, White Paper, 2014. [Online]. Available: http://portal.etsi.org/NFV/NFV_White_Paper3.pdf

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," IBM, IBM Research Report, 2014.

[6] K. Bousselmi, Z. Brahmi, and M. Gammoudi, "Cloud services orchestration: A comparative study of existing approaches," in *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*, May 2014, pp. 410–416.

[7] J. Wettinger, U. Breitenbücher, and F. Leymann, "Standards-based devops automation and integration using tosca," in *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC 2014)*. IEEE Computer Society, 2014, pp. 59–68.

[8] A.-F. Antonescu, P. Robinson, and T. Braun, "Dynamic topology orchestration for distributed cloud-based applications," in *Network Cloud Computing and Applications (NCCA), 2012 Second Symposium on*, Dec. 2012, pp. 116–123.

[9] J. Wettinger, U. Breitenbücher, and F. Leymann, "Compensation-based vs. convergent deployment automation for services operated in the cloud," in *Service-Oriented Computing*. Springer, 2014, pp. 336–350.

[10] W. Gerlach, W. Tang, K. Keegan, T. Harrison, A. Wilke, J. Bischof, M. D'Souza, S. Devoid, D. Murphy-Olson, N. Desai, and F. Meyer, "Skyport - container-based execution environment management for multi-cloud scientific workflows," in *DataCloud '14*. IEEE, Nov. 2014.

[11] R. Rosen. Linux containers and the future cloud. [Online]. Available: http://www.linuxjournal.com/content/linux-containers-and-future-cloud

[12] Openvz. [Online]. Available: http://openvz.org

[13] M. Xavier, M. Neves, F. Rossi, T. Ferreto, T. Lange, and C. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, Feb. 2013, pp. 233–240.

[14] Lxc. [Online]. Available: https://linuxcontainers.org

[15] Linux vserver. [Online]. Available: http://linux-vserver.org

[16] Docker. [Online]. Available: https://www.docker.com/

[17] Aufs. [Online]. Available: http://aufs.sourceforge.net/aufs2/man.html

[18] Coreos. [Online]. Available: https://coreos.com/

[19] Lmctfy. [Online]. Available: https://github.com/google/lmctfy

[20] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, "F1: A distributed sql database that scales," *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1068–1079, Aug. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536222.2536232
[21] Kubernets. [Online]. Available: http://kubernetes.io/
[22] Docker compose. [Online]. Available: http://blog.docker.com/2015/01/dockercon-eu-introducing-docker-compose/#more-4169
[23] cadvisor. [Online]. Available: https://github.com/google/cadvisor
[24] Project atomic. [Online]. Available: http://www.projectatomic.io/
[25] App container specification. [Online]. Available: https://github.com/coreos/rocket/blob/master/Documentation/app-container.md
[26] Soa manifesto. [Online]. Available: http://www.soa-manifesto.org/
[27] Rocket - app container runtime. [Online]. Available: https://github.com/coreos/rocket