Editor: Yong Cui • cuiyong@tsinghua.edu.cn

Toward a Standard Interface for Cloud Providers

The Container as the Narrow Waist

Silvery Fu and Jiangchuan Liu • *Simon Fraser University*

Xiaowen Chu • *Hong Kong Baptist University*

Yueming Hu • *South China Agricultural University*

Containers underwent a whirlwind adoption across cloud providers recently, as the Open Container Initiative works toward standardizing the container format and configuration. The container could become the cloud infrastructure's "narrow waist," bridging a proliferation of existing and emerging cloud services.

Cloud computing has experienced a rapid growth in the past decade; it's estimated that 95 percent of companies have introduced cloud services to facilitate their business in 2015.[1] As a constellation of cloud technologies, applications, and business models keep emerging, there's a strong demand on standardization towards a set of core technical specifications, which are to be adopted and shared by cloud vendors.[2] Pioneering works include the Open Virtualization Format (OVF),[3] Open Cloud Computing Interface (OCCI; http://occi-wg.org), and IEEE's P2301 (Cloud Profiles) and P2302 (Intercloud) working drafts.[4] In particular, OVF is an established standard, adopted by the American National Standards Institute (ANSI) and International Organization for Standardization (ISO) that defines an open and portable format for packaging and distributing software run on virtual machines (VMs). A typical .ovf package includes descriptors for hardware requirements, network, storage, and security settings. IEEE P2301 aims to unify the variety of design options adopted in cloud computing systems, by organizing those options into profiles, which provide guidance on developing standard-based products for cloud vendors. These efforts (mostly *virtualization-oriented*) have helped improve the interoperability among cloud ecosystem participants. Enhancements and the establishment of new standards, however, are still required to better address application deployment and portability, as well as resource management with minimized overhead.

New opportunity arises from the advancement of cloud infrastructures. *Containerization*, a technology that enables fine-grained resource control and isolation by encapsulating applications inside containers, has undergone a whirlwind adoption and gained native support across major cloud providers, including Amazon Elastic Compute Cloud (EC2) Container Service (ECS; http://aws.amazon.com/ecs), Google Container Engine (https://goo.gl/oxBS2e), and Microsoft Azure Container Service.[5] The container, in its simplest form, is a collection of OS kernel utilities configured to manage the resources that an application uses. The Open Container Initiative (OCI; www.opencontainers.org), launched under the auspices of the Linux Foundation in mid-2015, aims to establish open industry standards for the container's runtime and format. Following the container's popularity, it has readily gained sponsorship from more than 30 companies and organizations, including leading cloud providers and application platforms.

With this in mind, here we provide a high-level overview of container technology, along with its standardization status. We discuss the container's integral modules and explain why we envision it becoming the "narrow waist" of the cloud infrastructure, bridging a proliferation of existing and emerging cloud services.

The Container: Rationale and Key Modules

In general, a container offers cloud providers a lightweight tool to achieve resource multiplexing and control, as an alternative to virtualization (particularly those hypervisor-based). A closer relative to the container is an operating system process, because both of them essentially encapsulate a (single) application runtime. What the container offers additionally is the capabilities of controlling and isolating OS resources assigned to the runtime, meanwhile including complete dependencies in a container instance. As such, we can also refer to a container as a *virtual environment* (VE).

The idea of the container and OS-level virtualization isn't new. Linux-VServer[6] and OpenVZ (https://openvz.org) are two previous container-based virtualization platforms. Yet the container has only come to the fore in recent years, for two reasons. First, it has shifted from the original role as a "hypervisor-free" VM, where a single container instance had to be built full-fledge to support a full OS (as in VServer and OpenVZ), to a lightweight runtime environment for applications. Second, recent platforms significantly simplify the procedure for container creation and management. These two advancements meet the growing need of deploying cloud-based distributed applications with "just enough" performance overhead and maintenance cost. They've been jointly achieved by Docker (www.docker.com), the most established and popular container by far.

Docker relies on utilities of the modern Linux kernel to create and manage container runtime. (Eventually Docker developed native implementation of these modules, as a solution for cross-platform support.[7]) First and foremost, the *control groups* (cgroups) module defines a collection of kernel resource controllers for (including but not limited to) CPU, memory, and network and disk I/O. User-level code is allowed to customize these controllers through cgroups' virtual file system. At the runtime, cgroups is assigned to a process through function hooking, by which resource accesses of the process will trigger the corresponding hooks. As such, without intervening performance-critical execution paths, cgroups is able to achieve resource tracking and control efficiently. Figure 1 gives an example showing the use of cgroups with container runtime. Here, `cgroups1` defines the control groups of two CPU cores 3 and 4, and a limited amount of memory shaded in light gray. It's assigned to ContainerB that encapsulates a database server runtime (denoted by *db-server*). During its life cycle, the CPU usage of db-server is limited to cores 3 and 4 (which in turn will be used exclusively by db-server) and the memory footprint is limited by the given amount. As `cgroups0` and `cgroups2` indicate, it's also possible to define the cgroups solely for CPU, memory, and other manageable system resources, or arrange them together in different combinations. The hardware resources here can be virtualized, too, as we detail later.

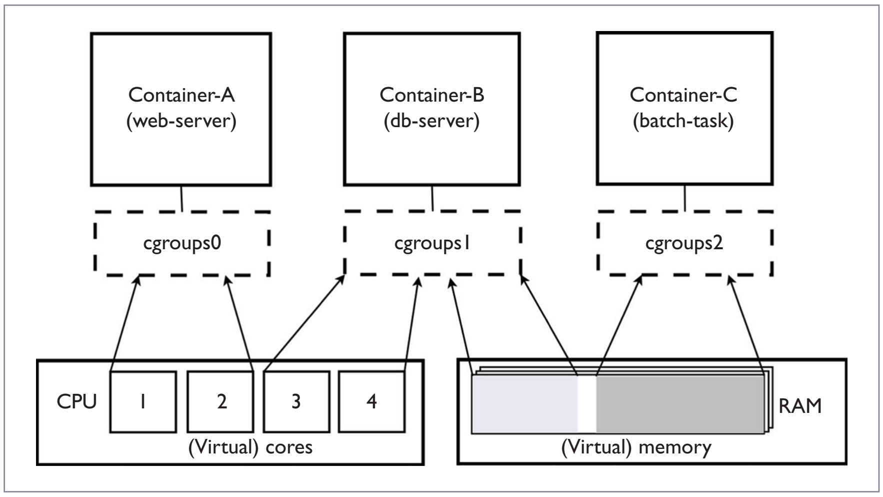

Figure 1. The control groups (cgroups) module for CPU and memory. The module defines a collection of kernel resource controllers.

Docker further leverages the namespaces isolation feature, forcing processes to have separate namespaces for system resources, including (but not limited to) a process identifier (PID), interprocess communication (IPC), and network. The resources allocated to the application runtime inside a container can't be addressed by the other containers, and vice-versa. With the use of cgroups and namespaces isolation, a container runtime can readily be hosted. Docker has donated the implementation of these modules to the OCI project in a collection called *runC*, serving as the cornerstone to a standardized container runtime. Notably, both cgroups and namespaces isolation have been used independently and flexibly to achieve resource control[8] or isolation,[9] and many other container platforms are built based on these modules (in addition to OCI, see https://github.com/coreos/rkt and https://linuxcontainers.org), making the container techniques versatile.

To facilitate container creation and management, Docker has also designed and implemented the container (image) format. Each container runtime is created from an image predefined, which includes all the dependencies that the target application requires. Besides, the images can be stored in publicly accessible repositories and conveniently dis-

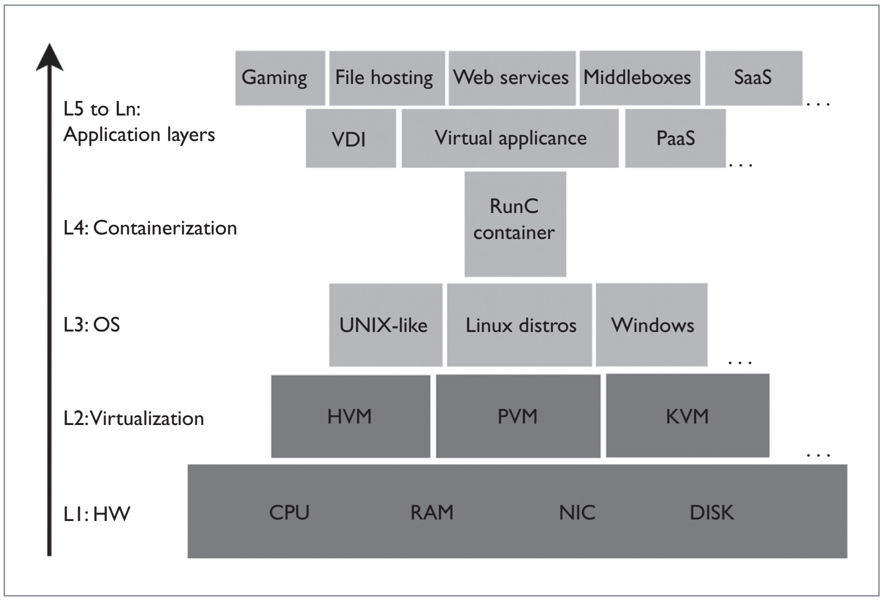

Figure 2. Cloud layered model: we envision the container (L4) as the narrow waist. It will bridge the underlying infrastructure layers and the application layers above.

tributed. Finally, Docker utilizes a layered file system to allow efficient sharing between container images, which significantly reduces the storage overhead.

Containerization versus Virtualization

To date, machine virtualization remains the most common way to manage hardware resources for cloud providers (for example, Xen[10] for Amazon EC2 public cloud), attracting significant standardization efforts (including OVF). Containers share such common design goals and features with VMs as resource isolation and imaging. Yet the new generation of containers represented by Docker, are built with important, distinct tenets.[6]

At the high level, containerization is upward-facing and application-driven, while virtualization is downward-facing and hardware-driven. Hypervisor-based virtualization (such as Xen) enables multiple users to create VMs that share the same physical hardware, where distinct OSs, ranging from the proprietary to open sourced, are hosted in an isolated fashion. Containerization permits only applications to be encapsulated in containers, which leads to greatly reduced deployment overhead and much higher instance density on a single machine. Unfortunately, it disallows a full OS stack to be run separately from the host OS, prohibiting a multi-OS setting.

At the low level, containerization leverages the host OS utilities to achieve resource encapsulation and management. Hypervisors, on the contrary, run directly on top of the hardware in the most-privileged mode, taking charge of accessing and managing the underlying hardware resources, akin to the role of an operating system kernel. The VMs and guest OS kernels now run in a less-privileged mode, such that any privileged system calls from guest OSs will be trapped to the hypervisor's kernel and executed in isolation. This process can be done in two ways: either through modifying the guest OS kernel and drivers to enforce privileged calls being sent to the hypervisor directly — which is known as *para-virtualization* (PV); or trapping those calls by special hardware extensions, known as *hardware-assisted virtualization* (HVM). As such, virtualization functions at the border of hardware and OS. It's able to provide strong performance isolation and security guarantees with the narrowed interface between VMs and hypervisors. Containerization, which sits in between the OS and applications, incurs lower overhead, but potentially introduces greater security vulnerabilities, such as namespace-agnostic system calls.[11]

In short, containerization isn't necessarily a replacement to virtualization; rather, these two complement each other, and are to be placed into a unified framework for cloud vendors and users.

The Container as the Narrow Waist

The renowned OSI model (ISO/IEC 7498-1) has helped delineate and standardize the Internet. As Figure 2 shows, we envision that a similar standardized reference model for the cloud infrastructure will emerge eventually. Such a model will be layered, with each layer emphasizing distinct infrastructural issues and functionalities, including resource multiplexing, isolation, orchestration, and application supports, respectively. An important design consideration for a layered model is the disposition of heterogeneity. For instance, the heterogeneity of network protocols converges at the IP layer, such that any transport layer protocols considering only the semantics of IP protocol will still be able to run on today's Internet infrastructure. Likewise, we must decide where a unified interface (that this, the "narrow waist") should be placed in the layered cloud infrastructure model. The key insight here is that a better part of the heterogeneity is introduced by the OS layer (L3) and application layers (L5 and above), where different applications rely on a variety of dependencies and OS supports. OSs themselves are also largely distinct from each other. Therefore, we envision container (L4) to become the narrow waist of this layered cloud model, bridging the underlying infrastructure layers and the application layers above.

The benefits of having such a unified/standardized interface are many-fold. Underneath, different host OSs can be utilized for the container, ranging from Unix-like ones to Windows-like proprietary ones, as long as they implement the container interface. Depending on user requirements, these host OSs are either placed on top of bare-metal non-virtualized machines or hypervisor-based VMs, where the first option gives the highest and close-to-native performance for demanding applications such as HPC workloads. Private clouds with managed user access could also benefit from this thinner model. With an additional hypervisor, a cluster yields stronger isolation and security guarantees, thereby being particularly appealing for public cloud providers. Moreover, the providers might opt for different hypervisors — ranging from full-virtualization to PV, and HVM to Kernel Virtual Machines (KVMs) — to meet various operational requirements.

On top of the interface, deploying and shipping an application runtime across distinct OSs and hardware architectures will become straightforward, because applications are agnostic to all of the underlying infrastructure details. Developers are spared from redundant and repetitive runtime environment configurations, system administrators are granted the ability of lightweight resource tracking and control, and cloud providers are allowed to support higher-level services such as platform as a service (PaaS) effortlessly. From the resource orchestration perspective, both the open source and proprietary cluster managers have been relying on the container as the basic scheduling unit.[12,13] In particular, the idea of two-level scheduling separates the cloud provider's concern in achieving efficient resource multiplexing[14] from the user's concern in attaining optimized workload locality, resource utilization, as well as cost savings. Further, the container interface will facilitate cloud services' transition beyond infrastructure as a service (IaaS), PaaS, and software as a service (SaaS), to highly modularized and distributed architectures such as microservice and "serverless" applications (see AWS Lambda; https://aws.amazon.com/lambda). Through defining a standard format of declaring container configuration and runtime, the promising OCI is a project that aims to establish the interface so that any host OS that implements it accordingly can support the container, bringing the aforementioned benefits to the cloud ecosystem. Earlier we detailed the key modules for implementing this interface, and it's worth noting here that the reference container implementation provided by OCI, *runC*, originates from Docker's `libcontainer` project.

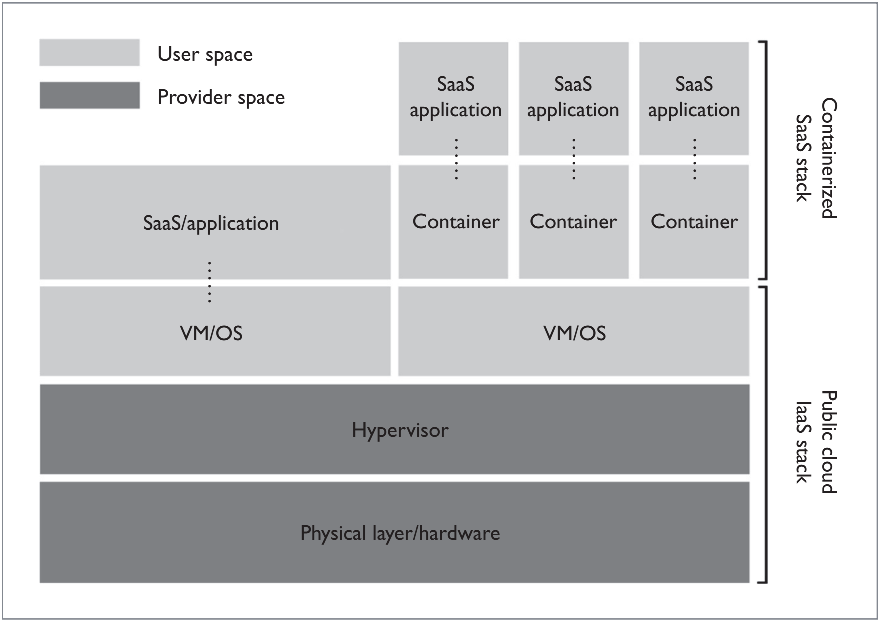

Figure 3. Hybrid virtualization layering in a public cloud. IaaS stands for infrastructure as a service, PaaS stands for platform as a service, and SaaS stands for software as a service.

Note that both virtualization and containerization are included in the reference model. When they're used together (for example, in a public cloud), with the underlying physical hardware and the OS in between, they form the hybrid-virtualization layers sitting at the bottom of the standard model (analogous to the link layers in the OSI model). This layering essentially places containers inside hypervisor-based VMs, which are then run on top of the underlying hardware; the rationale is that these two technologies, with distinct design goals and characteristics (as illustrated earlier), are in fact complementary to each other.

This hybrid layering is intuitive yet powerful, by which cloud users are allowed to orchestrate their resources provisioned without any assistance from the underlying infrastructure providers. As Figure 3 shows, the

Table 1. Performance of hybrid virtualization on a public Elastic Compute Cloud (EC2).

Resource type (benchmark)	Bare VM*	Hybrid VM
CPU (7z compression)	92.18 Mbytes/s	92.26 Mbytes/s
Memory (Sysbench, read)	10.84 Gbytes/s	10.85 Gbytes/s
Memory (Sysbench, write)	10.49 Gbytes/s	10.08 Gbytes/s
Disk (Bonnie++, rewrite)	119.95 Mbyte/s	118.22 Mbytes/s
Network (Iperf, TCP Send)	126.60 Mbytes/s	126.59 Mbytes/s
Network (Iperf, TCP Recv)	126.53 Mbytes/s	126.50 Mbytes/s

* VM = virtual machine.

cloud resource stack is, more often than not, separated into the user's and provider's space. For security and overall cloud performance, (public) providers don't let users freely launch any operations in their space, making resource consolidation and orchestration in the user space a headache (left half in the user space). In the hybrid layering (right half), the container adds another layer of abstraction in between the VM instance and applications, decoupling application-specific scheduling and VM scheduling. A direct use of this model is PaaS, which relies on the container to establish any service runtime environment effortlessly, and in the meantime the PaaS providers are exempted from handling the physical infrastructure. IaaS providers (such as Amazon ECS) can also benefit from this paradigm. Without involving redundant and unnecessary OS processes, the scaling out is more efficient for containers than VMs (see, for example, the Google Container Engine), which enables finer-grained billing. Noticeably, the hybrid layering is able to coexist with the virtualization-only solution, as Figure 3 indicates.

The additional layer might cause performance overhead and thus reduce the public cloud's cost-effectiveness. Previous works[11] have demonstrated that the container is able to achieve much-improved close-to-native performance, as compared to the hypervisor counterpart. However, the joint performance of the hypervisor and the container has yet to be explored. To quantitatively evaluate the overhead, we conducted experiments in the Amazon EC2 public cloud. Our metrics included CPU, memory, disk, and network I/O. To alleviate interference from the underlying cloud infrastructure, we repeated the experiments in four cloud instances, provisioned with the same resource offerings but at different times of the day. We containerized each EC2 instance with the newest version of the Docker container, and used microbenchmarks to measure and compare the performance inside and outside the container. As Table 1 shows, using the container introduces only negligible overhead with respect to CPU, memory, and disk and network I/O. (In our experiments, the CPU and memory-read performance are slightly higher for hybrid layering; but we conjecture that this is caused by the public cloud performance variance.) As we discussed, these resources are managed using cgroups through efficient function hooking (except for the network module, in which the Docker tags each network packet with a class identifier, and uses the *Linux traffic controller* to manage the packets from and to a particular container).

A container is a lightweight, flexible, and application-driven tool for fine-grained resource control and OS-level isolation. It augments the cloud infrastructure by addressing separate issues as compared to virtualization, which is a necessity when security is still one of the biggest concerns for cloud users.[1] The explored hybrid virtualization layering combines these two technologies with negligible performance overhead, setting the basis for the layered cloud reference model. In practice, we can place tens of or even hundreds of containers on the same physical or virtual machine.

Our experimental results indicate that using a container won't penalize the application performance when there's relatively low resource contention at the OS level. There is, however, contention at the hypervisor level from other tenants. It remains to be discovered — when we raise the density of co-located containers — exactly how the performance overhead will increase. Joining forces with major cloud vendors, the OCI project will further tackle these implementation and compatibility issues for the container. As today's applications are growing more distributed and cloud-based, it's expected that the container as the narrow-waist interface will improve the cloud infrastructure.

References

1. Harvard Business Review Analytic Services, "Cloud Computing Comes of Age," *Harvard Business Rev. Analytic Services Report*, tech. report, 2015; https://goo.gl/nZ4dIC.
2. S. Ortiz, "The Problem with Cloud-Computing Standardization," *Computer*, vol. 44, no. 7, 2011, pp. 13–16.
3. *Open Virtualization Format White Paper*, white paper, Distributed Management Task Force, v. 2.0.0a, document DSP2017, 3 July 2013; www.dmtf.org/sites/default/files/standards/documents/DSP2017_2.0.0a.pdf.
4. IEEE Cloud Computing Initiative, "Standards in Cloud Computing," IEEE Standards Assoc., 2016; http://cloudcomputing.ieee.org/standards.
5. R. Gardler, *Azure Container Service: Now and the Future*, blog, Microsoft, 29 Sept. 2015; https://goo.gl/QaDvQU.

6. S. Soltesz et al., "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors," *Proc. European Conf. Computer Systems*, vol. 41, no. 3, 2007, pp. 275–287.
7. N. Peterson, *Windows Containers*, Microsoft, 11 Dec. 2015; https://goo.gl/YAsGmo.
8. D. Lo et al., "Heracles: Improving Resource Efficiency at Scale," *Proc. Int'l Symp. Computer Architecture*, 2015, pp. 450–462.
9. B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," *Proc. Sigcomm Workshop on Hot Topics in Networks*, 2010, article no. 19.
10. P. Barham et al., "Xen and the Art of Virtualization," *Proc. Sigops*, vol. 37, no. 5, 2003, pp. 164–177.
11. W. Felter et al., "An Updated Performance Comparison of Virtual Machines and Linux Containers," *IBM Research Report*, tech. report RC25482, IBM Research Division, IBM, 21 July 2014.
12. B. Hindman et al., "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," *Proc. Usenix Conf. Networked Systems Design and Implementation*, 2011, pp. 295–308.
13. A. Verma et al., "Large-Scale Cluster Management at Google with Borg," *Proc. European Conf. Computer Systems*, 2015, article no. 18.
14. M. Armbrust et al., "A View of Cloud Computing," *Comm. ACM*, vol. 53, no. 4, 2010, pp. 50–58.

Silvery Fu is concurrently a master's student at Simon Fraser University and an undergraduate student in the dual degree program from Simon Fraser University and Zhejiang University. His interests focus on cloud computing topics surrounding virtualization, networking, and online gaming. Contact him at dif@sfu.ca.

Jiangchuan Liu is a university professor in the School of Computing Science, Simon Fraser University, and an NSERC E.W.R. Steacie Memorial Fellow. He's also the EMC-Endowed Visiting Chair Professor of Tsinghua University. His research interests include multimedia systems and networks, cloud computing, social networking, online gaming, Big Data, wireless sensor networks, and peer-to-peer networks. Liu has a PhD in computer science from the Hong Kong University of Science and Technology. He has served on the editorial boards of IEEE Transactions on Big Data, *IEEE Transactions on Multimedia, IEEE Communications Surveys and Tutorials, IEEE Access, IEEE Internet of Things Journal,* and *Computer Communications*. Contact him at jcliu@sfu.ca.

Xiaowen Chu is an associate professor in the Department of Computer Science, Hong Kong Baptist University. His research interests include distributed and parallel computing and wireless networks. Chu has a PhD in computer science from the Hong Kong University of Science and Technology. He's a senior member of IEEE and an associate editor of *IEEE Access*. Contact him at chxw@comp.hkbu.edu.hk.

Yueming Hu (corresponding author) is a professor of land information technology at South China Agricultural University. He's also the director of the Guangdong Province Key Laboratory of Land use and consolidation, and the chairman of the Guangdong Mapping and Geoinformation Industry Technology Innovation Alliance. Yueming has a PhD in soil geography from Zhejiang University. Contact him at ymhu@scau.edu.cn.