Containerization and the PaaS Cloud

Claus Pahl, Irish Centre for Cloud Computing and Commerce

Platform-as-a-service clouds can use containers to manage and orchestrate applications. This article discusses the requirements that arise from having to facilitate applications through distributed multicloud platforms.

The cloud relies on virtualization techniques to achieve elasticity of large-scale shared resources. Virtual machines (VMs) have been the backbone at the infrastructure layer providing virtualized operating systems (OSs). Containers are a similar but more lightweight virtualization concept; they're less resource and time-consuming, thus they've been suggested as a solution for more interoperable application packaging in the cloud.

Although VMs and containers are both virtualization techniques, they solve different problems. Containers are tools for delivering software—that is, they have a platform-as-a-service (PaaS) focus—in a portable way aiming at greater interoperability while still utilizing OS virtualization principles.[1] VMs, on the other hand, are about hardware allocation and management (machines that can be turned on/off and be provisioned)—that is, there's an infrastructure-as-a-service (IaaS) focus on hardware virtualization. Containers can be used as a replacement for VMs where the allocation of hardware resources is done through containers by componentizing workloads in between clouds.

For portable, interoperable applications in the cloud, we need a lightweight distribution of packaged applications for deployment and management.[2] One solution, *containerization*, provides

- a lightweight portable runtime;
- the capability to develop, test, and deploy applications to a large number of servers; and
- the capability to interconnect containers.

David Bernstein discusses the importance of container-based application deployment and cluster management for the cloud computing infrastructure.[3] This article reviews the virtualization principles behind containers, particularly in comparison with VMs. Specifically, I investigate the relevance of the new container technology for PaaS clouds, although containers also relate to the IaaS level through their sharing and isolation aspects. Because today's applications are distributed, I also discuss the resulting requirements for application

packaging and interoperable orchestration over clusters of containers. I aim to clarify how containers can change the PaaS cloud as a virtualization technique, specifically PaaS as a platform technology. I go beyond Bernstein,[3] addressing what's needed to evolve PaaS significantly further as a distributed cloud software platform, resulting in a discussion of achievements and limitations of the state of the art. To illustrate concepts, I'll discuss some example technologies that exemplify technology trends.

Virtualization and the Need for Containerization

Historically, virtualization technologies have developed out of the need for scheduling processes as manageable container units. The processes and resources in question are the file system, memory, network, and system information.

VMs as the cloud's core virtualization construct have been improved successively by addressing scheduling, packaging, and resource access (security) problems. VM instances acting as guests use large, isolated files on their hosts to store their entire file system and typically run a single, large process on the host. Although security concerns are usually addressed through isolation, several limitations remain. Full guest OS images are required for each VM in addition to the binaries and libraries necessary for the applications. Full images create a space concern that translates into RAM and disk storage requirements and is slow on startup (booting might take from 1 to more than 10 minutes[4]), as in Figure 1, which shows the different architectural settings.

Packaging and application management is a requirement that PaaS clouds need to address. In a virtualized environment, a solution must be grounded in technologies that allow the sharing of the underlying platform and infrastructure in a secure but also portable and interoperable way. Containers can meet these requirements, but a more in-depth elicitation of specific concerns is needed.

A container holds packaged, self-contained, ready-to-deploy parts of applications and, if necessary, middleware and business logic (in binaries and libraries) to run applications,[5] as Figure 1 illustrates. An example is a Web interface component with a Tomcat server. Successful tools like Docker are frameworks built around container engines that allow container engines to act as a portable mechanism to package and run applications as containers.[6] This means that a container covers an application tier or node in a tier, which results in the problem of
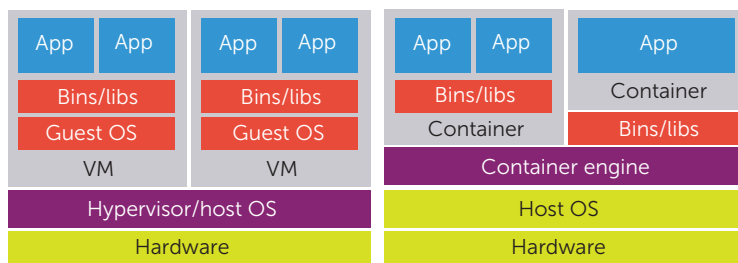

FIGURE 1. Virtualization architecture. The two possible scenarios, a traditional hypervisor architecture on the left and a container-based architecture on the right, differ in their management of guest operating system components.

managing dependencies between containers in multitier applications. An orchestration plan describes components, their dependencies, and their life cycle. A PaaS then enacts the workflows from the plan through agents (which could be a container runtime engine). PaaSs can support the deployment of applications from containers.

In PaaSs, there's a need to define, deploy, and operate cross-platform-capable cloud services using lightweight virtualization, for which containers are a solution.[7] There's also a need to transfer cloud deployments between cloud providers, which requires lightweight virtualized clusters for container orchestration.[3] Some PaaSs are lightweight virtualization solutions in this sense.

Containerization for Lightweight Virtualization and Application Packaging

Recent OS advances have improved their multitenancy capabilities—that is, the capability to share a resource.

Linux Containers

As an example of OS virtualization advances, new Linux distributions provide kernel mechanisms such as namespaces and control groups to isolate processes on a shared OS, supported through the Linux Container (LXC) project.

Namespace isolation allows groups of processes to be separated, preventing them from seeing resources in other groups. Container technologies use different namespaces for process isolation, network interfaces, access to interprocess communication, and mount points, and for isolating kernel and version identifiers.

Control groups manage and limit resource access for process groups through limit enforcement, accounting, and isolation—for example, by limiting the memory available to a specific container.
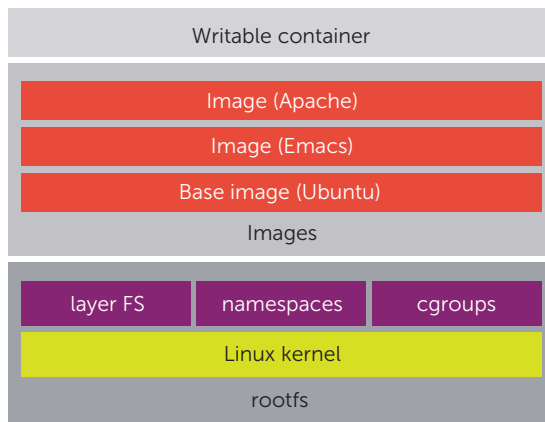
FIGURE 2. Container image architecture. Based on namespace and cgroup extensions of a Linux kernel, images are layered over each other, with a writable container image at the top.

This ensures that containers are good multitenant citizens on a host. It also provides better isolation between possibly large numbers of isolated applications on a host. Control groups allow containers to share available hardware resources and, if required, the control groups can set up limits and constraints.

Docker builds its solution on LXC techniques. A container-aware daemon, such as dockerd for Docker, can start containers as application processes and plays a key role as the root of the user space's process tree.

Docker Container Images

Containers are OS virtualization techniques based on namespaces and cgroups and are particularly suitable for application management in the PaaS cloud. A container is represented by lightweight images; VMs are also based on images but full, monolithic ones. Processes running in a container are almost fully isolated. Container images are the building blocks from which containers are launched.

Because it's currently the most popular container solution, I'll use Docker to illustrate how containerization works. A Docker image is made up of file systems layered over each other, similar to the Linux virtualization stack, using the LXC mechanisms, as Figure 2 illustrates.

In a traditional Linux boot, the kernel first mounts the root file system as read-only, then checks its integrity before switching the rootfs volume to read-write mode. Docker mounts the rootfs as read-only as in a traditional boot, but instead of changing the file system to read-write mode, it uses

a union mount to add a writable file system on top of the read-only file system.

There might be multiple read-only file systems stacked on top of each other. Using union mount, several file systems can be mounted on top of each other, which allows for creating new images by building on top of base images. Each of these file system layers is a separate image loaded by the container engine for execution.

Only the top layer is writable. This is the container itself, which can have state and is executable. It can be thought of as a directory that contains everything needed for execution. Containers can be made into stateless images (and reused in more complex builds), however.

A typical layering could include (top to bottom in Figure 2)

- a writable container image for applications,
- an Apache image and an Emacs image as sample platform components,
- a Linux image (a distribution such as Ubuntu), and
- the rootfs kernel image.

Containers are based on layers composed from individual images built on top of a base image that can be extended. Complete Docker images form portable application containers. They're also building blocks for application stacks. The approach is lightweight because single images can be changed and distributed easily.

Containerizing Applications and Managing Containers

The container ecosystem consists of an application container engine to run images and a repository or registry operated via push and pull operations to transfer images to and from host-based engines.

The repositories play a central role in providing access to possibly tens of thousands of reusable private and public container images, such as for platform components like MongoDB or Node.js. The container API allows creating, defining, composing, and distributing containers, running/starting images, and running commands in images.

Containers for applications can be created by assembling them from individual images, possibly based on base images from the repositories, as in Figure 2, which shows a containerized application. Containers can encapsulate several application components through the image layering and extension process. Different user applications and platform components can be combined in a container.
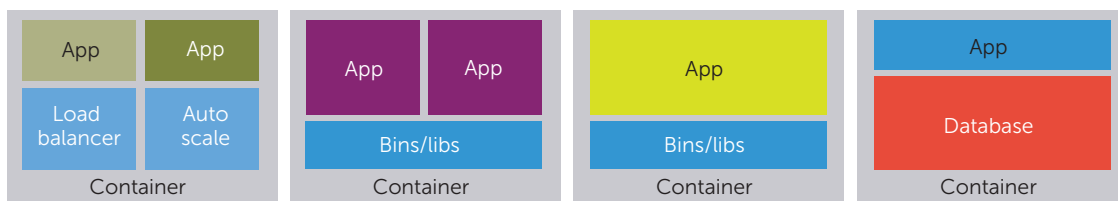
FIGURE 3. Container-based application architectures. These illustrate different architectural configurations with apps running on top of (a) management components such as load balancers and autoscalers, (b and c) binaries and libraries, and (d) databases.

Figure 3 shows different scenarios using the container capability of combining images for platform and application components.

The granularity of containers (that is, the number of applications inside a container) varies. Some favor the one-container-per-app approach, which still allows composing new stacks easily (for example, changing the webserver in an application) or reusing common components (for example, monitoring tools or a single storage service like memcached, either locally or predefined from a repository such as the Docker Hub). Apps can be built or rebuilt and managed easily. The downside is a larger number of containers with the respective interaction and management overhead compared to multi-app containers, although container efficiency should facilitate this.

Containers as application packages for interoperable and distributed contexts must facilitate storage and network management. There are two ways data is managed in Docker—data volumes and data volume containers. Data storage features can add data volumes to any container created from an image. A data volume is a specially designated directory within one or more containers that bypasses the union file system to provide features for persistent or shared data. Volumes can be shared and reused between containers, as Figure 4 illustrates. A data volume container enables sharing persistent data between application containers through a dedicated, separate data storage container.

Network management is based on two methods for assigning ports on a host—network port mappings and container linking. Applications can connect to a service or application running inside a Docker container via a network port. Container linking allows linking multiple containers together and sending information between them. Linked containers can transfer data about themselves via environment variables. To establish links and some relationship types, Docker relies on containers' names, which must be unique, meaning that links
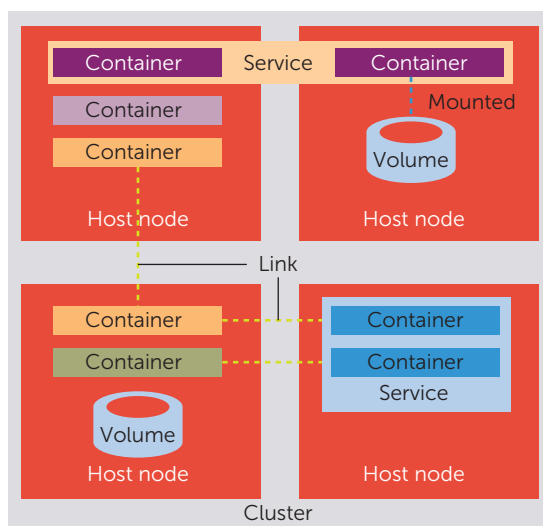

FIGURE 4. Container-based cluster architecture. Clusters assemble host nodes with container and data volumes, joined through links.

are often limited to containers of the same host (managed by the same daemon).

Comparison
Table 1 compares traditional VMs and containers. Some sources are also concerned about security, suggesting that it's preferable to run, for instance, only one Docker instance per host to avoid isolation limitations.[3]

Different Container Models
A range of other container technologies exist for different operating systems types (I single out Linux and Windows here) as well as specific or generic solutions for PaaS platforms[8]:

- Linux (Docker, LXC, OpenVZ, and others for variants such as BSD, HP-UX, and Solaris),
- Windows (Sandboxie), and
- Cloud PaaS (Warden/Garden (in Cloud Foundry) and LXC (in OpenShift).

Table 1. Virtual machine versus container-based application architectures.

Feature	Virtual machines	Containers
Standardization	Fairly standardized system images with capabilities similar to bare-metal computers (for example, Distributed Management Task Force's Open Virtualization Format, or OVF)	Not well standardized, OS- and kernel-specific with varying degrees of complexity
Host/guest architecture	Can run guest kernels that are different from the host, with consequently more limited insight into host storage and memory management	Run host kernels at guest level only but do so possibly with a different package tree or distribution such that the container kernel operates almost like the host
Boot process	Started through standard boot process, resulting in a number of hypervisor processes on the host	Can start containerized applications directly or through a container-aware init daemon, such as systemd, which appear as normal processes on the host

There's still an ongoing evolution of OS virtualization and containerization, aiming at providing OS support through standard APIs and tools for container management, network management, and more visible and manageable resource utilization.

The tool landscape is equally in evolution. For example, Rocket is a new container runtime from the CoreOS project (CoreOS is Linux for massive server deployments), which is an alternative to the Docker runtime. It's specifically designed for composability, security, and speed. These concerns highlight the teething concerns that the community is still engaged with.

Containerization in PaaS Clouds

Although VMs are ultimately the medium to provision PaaS platform and application components at the infrastructure layer, containers appear to be more suitable for application packaging and management in PaaS clouds.

PaaS Features

A PaaS generally provides mechanisms for deploying applications, designing applications for the cloud, pushing applications to their deployment environment, using services, migrating databases, mapping custom domains, IDE plugins, or a build integration tool. PaaSs have features such as built farms, routing layers, or schedulers that dispatch workloads to VMs. A container solution supports these problems through interoperable, lightweight, and virtualized packaging. Containers for application building, deployment, and management (through a runtime) provide interoperability. Containers produced outside a PaaS can be moved into the PaaS so that the container encapsulates the application. Existing PaaSs have embraced the momentum caused by containerization and standardized application packaging driven by Docker. Many PaaSs have a container foundation for running platform tools.

PaaS Evolution

The evolution of PaaS is moving toward container-based, interoperable PaaSs. The first generation consisted of classical fixed proprietary platforms such as Azure or Heroku. The second generation was built around open source solutions such as Cloud Foundry and OpenShift, which let users run their own PaaS (on-premise or in the cloud), already built around containers. OpenShift has now adopted the Docker container model, as has Cloud Foundry through its internal Diego solution. The current third generation includes platforms such as Dawn, Deis, Flynn, Octohost, and Tsuru, which are built on Docker from scratch and are deployable on a company's own servers or on public IaaS clouds.

Open PaaSs such as Cloud Foundry and OpenShift treat containers differently, however. Whereas Cloud Foundry supports stateless applications through containers, stateful services run in VMs. OpenShift doesn't distinguish these.

Service Orchestration

Development and architecture are central PaaS concerns. Recently developed microservice architectures break monolithic application architectures into service-oriented architecture (SOA)-style independently deployable services, which are well supported by container architectures. Services are loosely coupled, independent, and can be rapidly called and mapped to whatever business process is required. The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms. These services are independently deployable by a fully automated deployment and orchestration

framework. They must be able to deploy often and independently at arbitrary schedules, instead of requiring synchronized deployments at fixed times. Containerization provides an ideal mechanism for their deployment and orchestration, particularly, if they're to be PaaS-provisioned.

Container Orchestration and Clustering

Containerization facilitates the step from a single host to clusters of container hosts to run containerized applications over multiple clusters in multiple clouds.[9] The built-in interoperability makes this possible.

Container Clusters

A container-based cluster architecture groups hosts in clusters.[10] Figure 4 illustrates an abstract architectural scenario based on common container and cluster concepts. Container hosts are linked in a cluster configuration:

- Each cluster consists of several (host) nodes, where nodes are virtual servers on hypervisors or possibly bare-metal servers. Each host node holds several containers with common services such as scheduling, load balancing, and applications.
- Each container can hold continually provided services such as their payload service, which are one-off services (for example, print) or functional (middleware service) components.
- Application services are logical groups of containers from the same image. Application services allow scaling an application across nodes.
- Volumes are used for applications requiring data persistence. Containers can mount volumes. Data stored in these volumes persists even after a container is terminated.
- Links allow two or more containers, typically on a single host, to connect and communicate.

This configuration creates an abstraction layer for cluster-based service management that goes beyond container solutions like Docker.

A cluster management architecture has the following components:

- The deployment of distributed applications through containers is supported using a virtual scalable service node (cluster) with high internal complexity (supporting scaling, load balancing, failover) and reduced external complexity.
- An API allows operating clusters from the creation of services and container sets to other life-cycle functions.

- A platform service manager looks after the software packaging and management.
- An agent manages the container life cycles (at each host).
- A cluster head node service is the master that receives commands from the outside and relays them to container hosts.

This architecture allows development without regard to the network topology and requires no manual configuration.[11]

A cluster architecture is composed of engines to share service discovery (for example, through shared distributed key-value stores) and orchestration/deployment (load balancing, monitoring, scaling, and also file storage, deployment, pushing, and pulling).

This satisfies some of the requirements Nane Kratzke lists for cluster architectures.[8] A lightweight virtualized cluster architecture should provide several management features as part of the abstraction on top of the container hosts:

- hosting containerized services and providing secure communication between these services,
- autoscalability and load-balancing support,
- distributed and scalable service discovery and orchestration, and
- transfer/migration of service deployments between clusters.

Mesos is an example of a cluster management platform. This Apache project binds distributed hardware resources into a single pool of resources. Application frameworks can use Mesos to efficiently manage workload distribution. Mesos is a distributed systems kernel following the same principles as the Linux kernel but at a different level of abstraction. The Mesos kernel runs on all cluster machines and provides applications with APIs for resource management and scheduling across cloud environments. It natively supports LXC and also supports Docker.

An example clustering management solution at a higher level than Mesos is the Kubernetes architecture, which is supported by Google. Kubernetes can be configured to allow orchestrating Docker containers on Mesos at scale. Kubernetes is based on processes that run on Docker hosts that bind hosts into clusters and manage containers. Minions are container hosts that run pods (that is, sets of containers) on the same host. OpenShift has adopted Kubernetes. Google expertise incorporated in Kubernetes competes here with platform-specific evolution toward container-based orchestration. Cloud
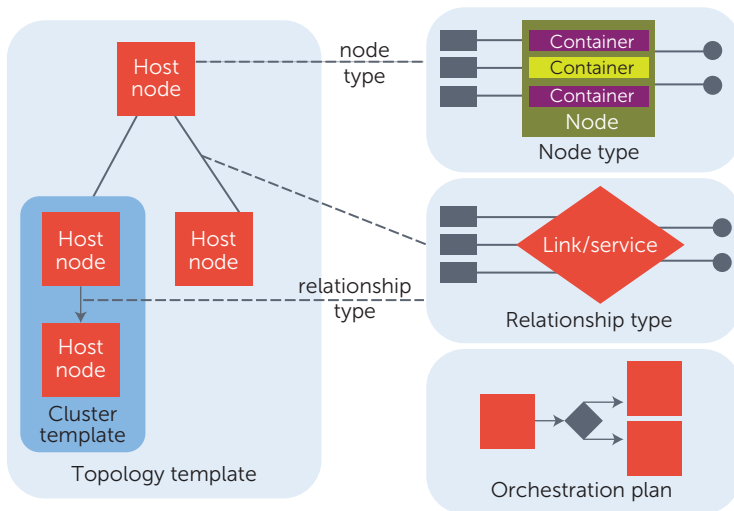
FIGURE 5. Cluster topology orchestration (adapted from the Topology and Orchestration Specification for Cloud Applications [TOSCA] by applying the generic TOSCA service template to the container and cluster technology context).

Foundry, for instance, uses Diego as an orchestration engine for containers.

Network and Data Challenges

Containers in distributed systems require advanced network support. Containers provide an abstraction that makes each container a self-contained unit of computation. Traditionally, containers were exposed on the network via the shared host machine's address. In Kubernetes, each group of containers (or *pods*) receives its own unique IP address, reachable from any other pod in the cluster, whether colocated on the same physical machine or not. This requires advanced routing features based on network virtualization.

Data storage is another problem in distributed container management. Managing containers in Kubernetes clusters might be hampered in terms of flexibility and efficiency by the need for pods to colocate with their data. What is needed is to pair a container with a storage volume that, regardless of the container's location in the cluster, follows it to the physical machine.

Orchestration Scenarios

Container cluster-based multi-PaaS is a solution for managing distributed software applications in the cloud, but this technology still faces challenges. These include formal descriptions or user-defined metadata for containers beyond image tagging with simple IDs but also clusters of containers and their

orchestration. The topology of distributed container architectures needs to be specified and its deployment and execution orchestrated, as Figure 5 illustrates.

There's currently no accepted solution for the orchestration problem; however, I briefly illustrate its relevance using a possible solution. Although Docker has started to develop its own orchestration solution and Kubernetes also provides an orchestration mechanism for containers onto nodes, a more comprehensive solution that would tackle orchestration of complex application stacks could involve Docker orchestration based on the Topology and Orchestration Specification for Cloud Applications (TOSCA),[12] a topology-based service orchestration standard that's supported, for example, by the Cloudify PaaS. Cloudify uses TOSCA to enhance the portability of cloud applications and services (see Figure 5). TOSCA enables

- the interoperable description of application and infrastructure cloud services (here, containers hosted on nodes),
- the relationships between parts of the service (here, service compositions and links, as illustrated in Figure 4), and
- the operational behavior of these services (for example, deploy, patch, and shutdown) in an orchestration plan.

The TOSCA framework is independent of the supplier creating the service and any particular cloud provider or hosting technology. TOSCA will also make it possible to associate higher-level operational behavior with cloud infrastructure management. Using TOSCA templates for container clusters and abstract node and relationship types, an application stack template can be specified.

Observations

Some PaaSs have started to address limitations in the context of programming (such as orchestration) and DevOps for clusters. The examples I've used allow for some observations. First, containers are by now largely adopted for PaaS clouds.[3] Second, standardization through adoption of emerging de facto standards such as Docker or Kubernetes is also taking place, although at a slower pace. Third, development and operations are still at an early stage.

Cloud management platforms are still at an earlier stage than the container platforms they build on. Whereas clusters in general are about distribution, the question emerges as to what extent this distribution reaches the edge of the cloud with small devices

and embedded systems and whether devices running small Linux distributions such as the Debian-based DSL (which requires around 50 Mbytes of storage) can support container host and cluster management.

Container technology has a huge potential to substantially advance PaaS technology toward distributed heterogeneous clouds through lightweightness and interoperability, as Bernstein and other have recognized.[3] However, we still need significant improvements to deal with data and network management aspects as well as an abstract development and architecture layer. •••

Acknowledgments
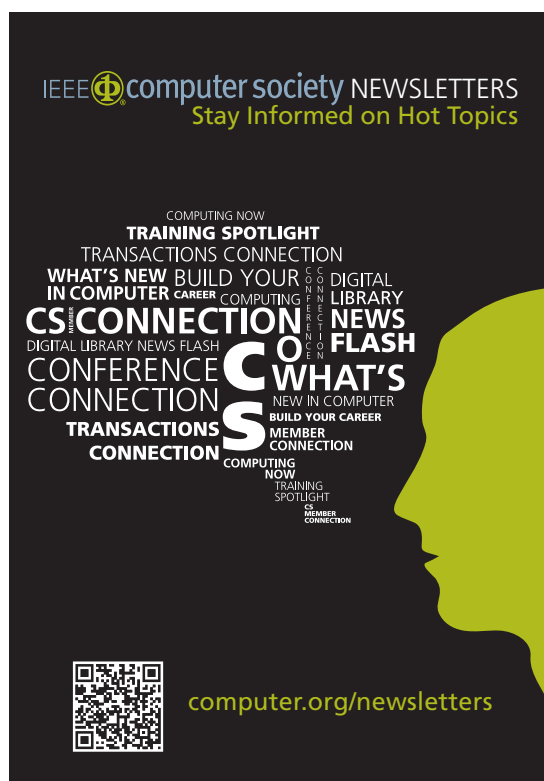
References

1. R. Ranjan, "The Cloud Interoperability Challenge," *IEEE Cloud Computing*, vol. 1, no. 2, 2014, pp. 20–24.
2. B. Di Martino, "Applications Portability and Services Interoperability among Multiple Clouds," *IEEE Cloud Computing*, vol. 1, no. 1, 2014, pp. 74–77.
3. D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, 2014, pp. 81–84.
4. M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," *Proc. IEEE 5th Int'l Conf. Cloud Computing* (Cloud 12), 2012, pp. 423–430.
5. S. Soltesz et al., "Container-Based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors," *ACM SIGOPS Operating Systems Rev.*, vol. 41, no. 3, 2007, pp. 275–287.
6. J. Turnbull, *The Docker Book*, 2014; www.dockerbook.com.
7. T.H. Noor et al., "Analysis of Web-Scale Cloud Services," *IEEE Internet Computing*, vol.18, no. 4, 2014, pp. 55–61.
8. N. Kratzke, "A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms," *Open J. Mobile Computing and Cloud Computing*, vol. 1, no. 2, 2014, pp. 17–30.
9. B. Satzger et al., "Winds of Change: From Vendor Lock-In to the Meta Cloud," *IEEE Internet Computing*, vol. 17, no. 1, 2013, pp. 69–73.
10. V. Koukis, C. Venetsanopoulos, and N. Koziris, "~okeanos: Building a Cloud, Cluster by Cluster," *IEEE Internet Computing*, vol. 17, no. 3, 2013, pp. 67–71.
11. O. Gass, H. Meth, and A. Maedche, "PaaS Characteristics for Productive Software Development: An Evaluation Framework," *IEEE Internet Computing*, vol. 18, no. 1, 2014, pp. 56–64.
12. T. Binz et al., "Portable Cloud Services Using TOSCA," *IEEE Internet Computing*, vol. 16, no. 3, 2012, pp. 80–85.

CLAUS PAHL *is the lead principal investigator of the Irish Centre for Cloud Computing and Commerce (IC4) and a funded investigator and executive member of the Irish Software Research Centre, Lero. His research interests include software engineering in service and cloud computing, specifically migration and scalability concerns. Pahl has a PhD in computing from the University of Dortmund. Contact him at cpahl@computing.dcu.ie.*