Containerisation and the PaaS Cloud

Claus Pahl

Abstract— Containerisation is widely discussed as a lightweight virtualisation solution. Apart from exhibiting benefits over traditional virtual machines in the cloud, containers are especially relevant for Platform-as-a-Service (PaaS) clouds to manage and orchestrate applications through containers as an application packaging mechanism. We discuss the requirements that arise from having to facilitate applications through distributed multi-cloud platforms.

Index Terms—Cloud Computing, Cluster, Container, Docker, Kubernetes, Multi-cloud, PaaS, Virtualisation.

1 Introduction

THE cloud relies on virtualisation techniques to achieve elasticity of large-scale shared resources. Virtual machines (VMs) have been the backbone at the infrastructure layer providing virtualised operating systems. Containers are a similar, more lightweight virtualisation concept, i.e., less resource and time consuming. They have been suggested as a solution for more interoperable application packaging in the cloud.

VMs and containers are both virtualisation techniques, but solve different problems. The difference is that containers are tools for delivering software – i.e., there is a PaaS (Platform-as-a-Service) focus – in a portable way aiming at more interoperability [1] while still utilising operating systems (OS) virtualisation principles. VMs on the other hand are about hardware allocation and management (machines that can be turned on/off and be provisioned) – i.e., there is an IaaS (Infrastructure-as-a-Service) focus on hardware virtualisation. Containers as a replacement for VMs are only a specific use case where the allocation of hardware resources is done through containers by componentising workloads in-between clouds.

For portable, interoperable applications in the cloud, we need a lightweight distribution of packaged applications for deployment and management [2]. A solution is containerisation. The basic ideas of containerisation are

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

Bernstein [3] already proposes containers to address concerns at the cloud PaaS level. They also relate to the IaaS level through sharing and isolation aspects.

This article reviews the virtualisation principles behind containers, in particular in comparison with virtual machines. The relevance of the new container technology for PaaS cloud shall be specifically investigated. As applications are distributed today, the resulting requirements for application packaging and interoperable orchestration over clusters of containers are also discussed. We aim to clarify how containers can change the PaaS cloud as a virtualisation technique, specifically PaaS as a platform technology. We go beyond [3], addressing what is 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, some sample technologies will be discussed if they exemplify technology trends well.

2 Virtualisation and the Need for Containerisation

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

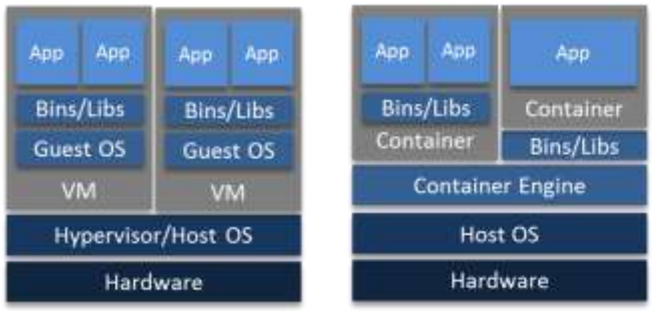

Fig. 1. Virtualisation architecture.

Virtual machines as the core virtualisation construct of the cloud have been improved successively by addressing scheduling, packaging and resource access (security) problems. VM instances as guests use isolated large files on their host to store their entire file system and run typically a single, large process on the host. While security concerns are largely addressed through isolation, a number of limitations remain. It needs full guest OS images for each VM in addition to the binaries and libraries necessary for the applications, i.e., a space concern that translates into RAM and disk storage requirements and is slow on startup (booting might take from one to more than 10 minutes [4]), see Fig. 1.

Packaging and application management is a requirement that PaaS clouds need to answer. In a virtualised environment, this has to be grounded in technologies that

• *C. Pahl is with the Irish Centre for Cloud Computing and Commerce IC4 and the Irish Software Research Centre Lero, School of Computing, Dublin City University, Dublin 9, Ireland. E-mail: cpahl@computing.dcu.ie .*

allow the sharing of the underlying platform and infrastructure in a secure, but also portable and interoperable way. Containers can match 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], see Fig. 1. An example would be a Web interface component with a Tomcat server. Successful tools like Docker are frameworks built around container engines [6] that allow containers to act as a portable way to package applications to run in containers. This means that a container covers an application tier or node in a tier, which results in the problem of managing dependencies between containers in multi-tier applications. An orchestration plan describes components, their dependencies and their lifecycle in a layered plan. 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 is a need to define, deploy and operate cross-platform capable cloud services [7] using lightweight virtualisation, for which containers are a solution. There is also a need to transfer cloud deployments between cloud providers, which requires lightweight virtualised clusters for container orchestration [3]. Some PaaS are lightweight virtualisation solutions in this sense.

3 Containerisation for Lightweight Virtualisation and Application Packaging

Recent OS advances have improved their multi-tenancy capabilities, i.e., the capability to share a resource.

3.1 Linux Containers

As an example of OS virtualisation advances, new Linux distributions provide kernel mechanisms such as namespaces and cgroups to isolate processes on a shared OS – supported through the Linux container project LXC.

- Namespace isolation allows groups of processes to be separated not allowing them to see resources in other groups. Different namespaces are used by container technologies for process isolation, network interfaces, access to inter-process communication, mount-points or for isolating kernel and version identifiers.
- cgroups (control groups) manage and limit resource access for process groups through limit enforcement, accounting and isolation, e.g., limiting the memory available to a specific container. This ensures containers are good multi-tenant citizens on a host. It provides better isolation between possibly large numbers of isolated applications on a host. Control groups allow sharing available hardware resources between containers and, if required, setting up limits and constraints.

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

3.2 Docker Container Images

Based on these mechanisms, containers are OS virtualisation techniques 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.

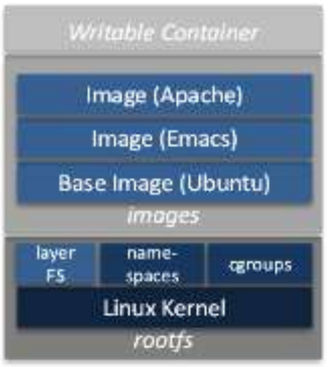

Fig. 2. Container Image Architecture.

As it is currently the most popular container solution, Docker shall illustrate how containerisation works. A Docker image is made up of file systems layered over each other, similar to the Linux virtualisation stack, using the LXC mechanisms, see Fig. 2.

- 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 may actually 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 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), though.

A typical layering could include (top to bottom, see Fig. 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 are also building blocks for application stacks. The approach is lightweight as single images can be changed and distributed easily.

3.3 Containerising 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, e.g., for platform components such as MongoDB or Node.js. The container API allows creating, defining, composing, distributing containers, running/starting images and running commands in images.

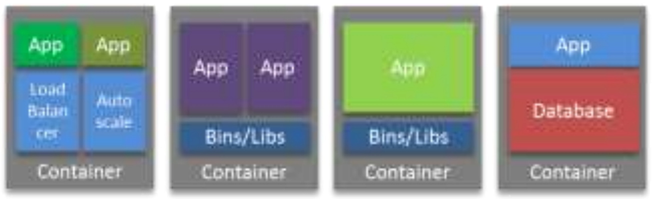

Fig. 3. Container-based Application Architecture.

Containers for applications can be created by assembling them from individual images, possibly based on base images from the repositories, which can be seen in Fig. 2 that shows a containerised application. Containers can encapsulate a number of application components through the image layering and extension process. Different user applications and platform components can be combined in a container. Fig. 3 illustrates different scenarios using the container capability of combining images for platform and application components.

The granular of containers, i.e., the number of applications inside, varies. Some favour the one-container-per-app approach, which still allows composing new stacks easily (e.g., changing the Web server in an application) or reuse common components (e.g., monitoring tools or a sinlge storage service like memchached - either locally or predefined from a repository such as the Docker Hub). Apps can be built/rebuilt and managed easily. The downside is a larger number of containers with the respective interaction and management overhead compared to multi-app containers, though the container efficiency should faciliate this.

Storage and network management are two specific issues that containers as application packages for interoperable and distributed contexts must facilitate.

- 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, see Fig. 4. 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 the names of containers. Container names have to be unique, which means that links are often limited to containers of the same host (managed by the same daemon).

3.4 Comparison

Both traditional VMs and containers shall be compared in order to summarise the two technologies, see Table 1. Some sources are also concerned about security, suggesting to run for instance only one Docker instance per host to avoid isolation limitations [3].

TABLE 1. Container-based Application Architecture.

	VMs	Containers
Standardisation	Fairly standardised system images with capabilities similar to bare-metal computers (e.g., OVF from DMTF).	Not well standardised, OS- and kernel-specific with varying degrees of complexity.
Host/ guest architecture	Can run guest kernels that are different from the host, with consequent more limited insight into host storage and memory management.	Run host kernels at guest level only, but can 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 containerised application directly or through container-aware init daemon like systemd. These appear as normal processes on the host.

3.5 Different Container Models

We use Docker to illustrate some core concepts, but a range of other container technologies exist for different operating systems types (we single out Linux and Windows below) and also specific or generic solutions for PaaS platforms [8]:

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

There is still an ongoing evolution of OS virtualisation and containerisation, aiming at providing OS support through standard APIs and tools for container management, network management and making resource utilisation more visible and manageable.

The tool landscape is equally in evolution. As one 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 is specifically designed for composability, security, and speed. These concerns highlight the teething concerns that the community is still engaged with.

4 CONTAINERISATION IN PAAS CLOUDS

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

4.1 PaaS Features

PaaS generally provide 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. PaaS have features like built farms, routing layers, or schedulers that dispatch workloads to VMs. A container solution supports these problems through interoperable, lightweight and virtualised packaging. Containers for application building, deployment and management (through a runtime) provide interoperability. Containers produced outside a PaaS can be moved in – the container encapsulates the application. Existing PaaS have embraced the momentum caused by containerisation and standardised application packaging driven by Docker. Many PaaS have a container foundation for running platform tools.

4.2 PaaS Evolution

The evolution of PaaS is moving towards container-based, interoperable PaaS.

- The first generation was made up of classical fixed proprietary platforms such as Azure or Heroku.
- The second generation was built around open-source solutions such as Cloud Foundry or OpenShift that allow users to run their own PaaS (on-premise or in the cloud), already built around containers. Openshift moves now from its own container model to the Docker container model, as does Cloud Foundry through its internal Diego solution.
- The current third generation includes platforms like Dawn, Deis, Flynn, Octohost and Tsuru, which are built on Docker from scratch and are deployable on own servers or on public IaaS clouds.

Open PaaS like Cloud Foundry and OpenShift treat containers differently, though. While Cloud Foundry supports state-less applications through containers, stateful services run in VMs. Openshift does not distinguish these.

4.3 Service Orchestration

Development and architecture are central PaaS concerns. Recently, microservice architectures are discussed. This is an approach to breaking monolithic application architectures into SOA-style independently deployable services, which are well supported by container architectures. Services are loosely coupled, independent services that can be rapidly called and mapped to whatever business process is required. The microservices 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 fully automated deployment and orchestration framework. They require the ability to deploy often and independently at arbitrary schedules, instead of requiring synchronized deployments at fixed times. Containerisation provides an ideal mechanism for their deployment and orchestration, particularly, if these are to be PaaS-provisioned.

5 CONTAINER ORCHESTRATION AND CLUSTERING

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

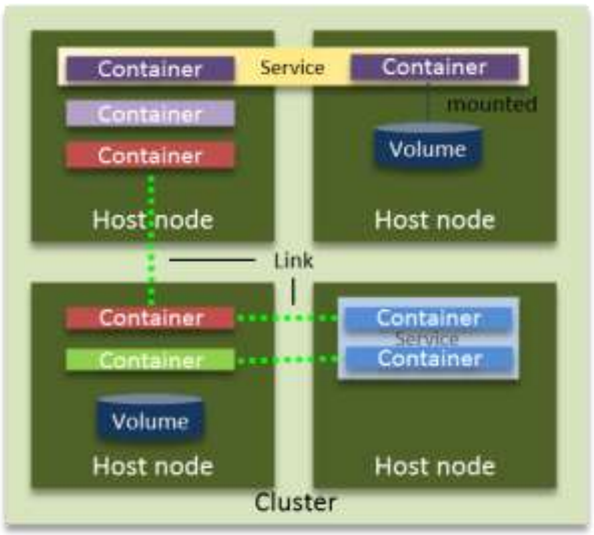

Fig. 4. Container-based Cluster Architecture.

5.1 Container Clusters

A container-based cluster architecture groups hosts into clusters [10]. Fig. 4 that illustrates an abstract architectural scenario based on common container and cluster concepts. Container hosts are linked into 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, so-called jobs, which are once-off services (e.g., 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 that require 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 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 lifecycle functions.
- A platform service manager looks after the software packaging and management.
- An agent manages the container lifecycles (at each host).
- A cluster head node service is the master that receives commands from the outside and relays them to container hosts.

This 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 (e.g., through shared distributed key value stores) and orchestration/deployment (load balancing, monitoring, scaling, and also file storage, deployment, pushing, pulling).

This satisfies some of the requirements listed by Kratzke [8] for cluster architectures. A lightweight virtualised cluster architecture should provide a number of management features as part of the abstraction on top of the container hosts:

- Hosting containerised services and providing secure communication between these services,
- Auto-scalability and load balancing support,
- Distributed and scalable service discovery and orchestration,
- Transfer/migration of service deployments between clusters.

A sample cluster management platform is Mesos, an Apache project that binds distributed hardware resources into a single pool of resources. Mesos can be used by application frameworks to efficiently manage workload distribution. It 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.

A sample clustering management solution that is 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, i.e., sets of containers on the same host. Openshift has adopted Kubernetes. Expertise by Google incorporated in Kubernetes competes here with platform-specific evolution towards container-based orchestration. Cloud Foundry, for instance, uses Diego as a new orchestration engine for containers.

5.2 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 (called pods) receives its own unique IP address, reachable from any other pod in the cluster, whether co-located 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 besides the network aspect. Managing containers in Kubernetes clusters might be hampered in terms of flexibility and efficiency by the need for pods to co-locate with their data. What is needed is to pair up a container with a storage volume that, regardless of the container location in the cluster, follows it to the physical machine.

5.3 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, see Fig. 5.

While there is no accepted solution for the orchestration problems, its relevance shall briefly be illustrated using a possible solution. While Docker has started to develop its own orchestration solution and Kubernetes is another relevant project, a more comprehensive solution that would tackle orchestration of complex application stacks could involve Docker orchestration based on the topology-based service orchestration standard TOSCA, which is for instance supported by the Cloudify PaaS. Cloudify uses TOSCA (Topology and Orchestration Specification for Cloud Applications [12]) to enhance the portability of cloud applications and services, see Fig. 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 Fig. 4,
- the operational behaviour of these services (e.g., deploy, patch, shutdown) in an orchestration plan.

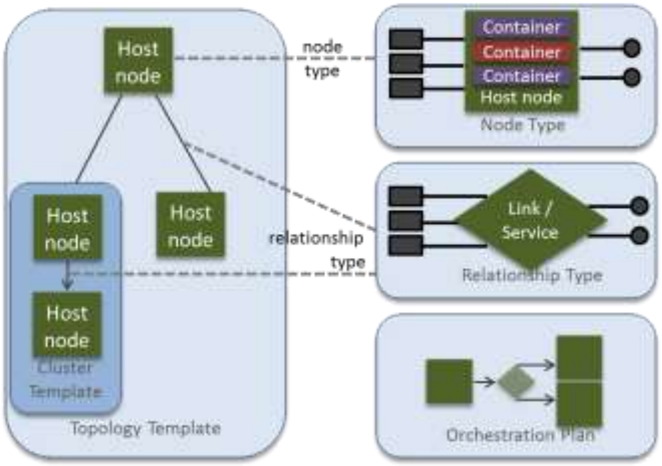

Fig. 5. Cluster Topology Orchestration [adapted from TOSCA].

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

5.4 Observations

Some PaaS have started to address limitations in the context of programming (such as orchestration) and DevOps for clusters. The examples used above allow some observations. Firstly, containers are by now largely adopted for PaaS clouds [3]. Secondly, standardisation by adopting emerging de-facto standards like Docker or Kubernetes is also happening, though at a slower pace. Thirdly, development and operations are still at an early stage.

Cloud management platforms are still at an earlier stage than the container platforms that they build on. While clusters in general are about distribution, the question emerges as to which 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 50MB storage) can support container host and cluster management.

In conclusion, container technology has a huge potential to substantially advance PaaS technology towards distributed heterogeneous clouds through lightweightness and interoperability – which has also been recognised by Bernstein and others [3]. However, significant improvements are still required to deal with data and network management aspects as is providing an abstract development and architecture layer.

Acknowledgment

This work was supported in part by the Irish Centre for Cloud Computing and Commerce (IC4), an Irish national Technology Centre funded by Enterprise Ireland and the Irish Industrial Development Authority, and by Science Foundation Ireland grant 13/RC/2094 to Lero - the Irish Software Research Centre.

References

[1] R. Ranjan, "The Cloud Interoperability Challenge", IEEE Cloud Computing, vol. 1, no. 2, pp. 20-24, 2014.
[2] B. Di Martino, "Applications Portability and Services Interoperability among Multiple Clouds", IEEE Cloud Computing , vol. 1, no. 1, pp. 74-77, 2014.
[3] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014.
[4] M. Mao and M. Humphrey, "A Performance Study on the VM Startup Time in the Cloud," 5th International Conference on Cloud Computing (CLOUD), IEEE, pp. 423-430, 2012.
[5] S. Soltesz, H. Pötzl, M.E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors". ACM SIGOPS Operating Systems Review, vol. 41, no. 3, pp. 275-287, 2007.
[6] J. Turnbull, "The Docker Book". http://www.dockerbook.com/. 2014.
[7] T.H. Noor, Q.Z. Sheng, A.H.H. Ngu, and S. Dustdar, "Analysis of Web-Scale Cloud Services", IEEE Internet Computing, vol.18, no. 4, pp. 55-61, 2014.
[8] N. Kratzke, "A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms", Open Journal of Mobile Computing and Cloud Computing vol. 1, no. 2, 2014.
[9] B. Satzger, W. Hummer, C. Inzinger, P. Leitner, and S. Dustdar, "Winds of Change: From Vendor Lock-In to the Meta Cloud", IEEE Internet Computing, vol. 17, no. 1, pp. 69-73, 2013.
[10] V. Koukis, C. Venetsanopoulos, and N. Koziris, "~okeanos: Building a Cloud, Cluster by Cluster", IEEE Internet Computing, vol. 17, no. 3, pp. 67-71, 2013.
[11] O. Gass, H. Meth, and A. Maedche, "PaaS Characteristics for Productive Software Development: An Evaluation Framework", IEEE Internet Computing, vol. 18, no. 1, pp. 56-64, 2014.
[12] T. Binz, G. Breiter, F. Leymann, and T. Spatzier, "Portable Cloud Services Using TOSCA", IEEE Internet Computing, vol. 16, no. 3, pp. 80-85, 2012.

Biography

Claus Pahl. Claus Pahl is the Lead Principal Investigator of the Irish Centre for Cloud Computing and Commerce IC4 and a Funded Investigator and an 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. He holds a Ph.D. in computing from the University of Dortmund and an M.Sc. from the University of Technology in Braunschweig.