Scientific Online

OPEN JOURNAL OF MOBILE COMPUTING AND CLOUD COMPUTING
Volume 1, Number 2, November 2014

OPEN JOURNAL OF MOBILE COMPUTING AND CLOUD COMPUTING

A Lightweight Virtualization Cluster Reference Architecture Derived from Open Source PaaS Platforms

Nane Kratzke*

Lübeck University of Applied Sciences, Center of Excellence CoSA, 23562 Lübeck, Germany
***Corresponding author**: nane.kratzke@fh-luebeck.de

Abstract:
Actual state of the art of cloud service design does not deal systematically how to define, deploy and operate cross-platform capable cloud services. By simplifying and harmonizing the use of IaaS cloud infrastructures using lightweight virtualization approaches, the transfer of cloud deployments between a variety of cloud service providers becomes more frictionless. This article proposes operating system virtualization as an appropriate and already existing abstraction layer on top of public and private IaaS infrastructures, and derives an reference architecture for lightweight virtualization clusters. This reference architecture is reflected and derived from several existing (open source) projects for container-based virtualization like Docker, Jails, Zones, Workload Partitions of various Unix operating systems and open source PaaS platforms like CoreOS, Apache Mesos, OpenShift, CloudFoundry and Kubernetes.

Keywords:
Lightweight Virtualization; Cluster; Reference Architecture; PaaS; Cloud Computing

1. INTRODUCTION

If this paper uses common cloud computing terms like Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Software as a Service (SaaS) as well as public, hybrid and private cloud these terms are used according to the NIST definition of cloud computing [1].

The reader of this paper should be familiar with the term “reference architecture”. Reference architectures (RA) in the field of software architecture or enterprise architecture can be seen as a template for a particular system domain. It provides a common vocabulary with which implementations are discussable and commonalities being identifiable. An RA generalizes a set of solutions. These solutions are structured for the depiction of one or more concepts based on the harvesting of a set of patterns that have been observed in a number of successful implementations. RAs shall accelerate adoption processes by re-using effective solutions and provide a basis for governance to ensure the consistency and applicability of technology use within an organization.

Cloud computing can be seen as a new IT service delivery paradigm, often called a “programmable data center” [2]. These web accessible “programmable datacenters” provide several interesting features.

They allow to scale up and down resource usage on an as-needed basis. So cloud computing is likely to provide cost advantages (especially in peak load scenarios [3, 4]) due to its economical pay-as-you-go principle and technical quick scaling capability. Economically, it turn fixed into variable costs making it especially interesting for innovative start up companies, not able to do significant up-front investments. Nevertheless, beside above mentioned advantages, common security, governance, availability, cost concerns and especially technological vendor lock-in worries often come along with cloud computing which is especially true for small and medium-sized enterprises. While security and governance concerns often can be answered by encryption [3], cost concerns can be answered by cost based decision-making models [3, 4], vendor lock-in problems stay. From a technological point of view actual state of the art of cloud based software services are often characterized by a highly implicit technological dependency on underlying hosting cloud infrastructures.

Dustdar *et al.* [5] introduce a concept called meta-cloud to avoid vendor lock-in and furthermore, they postulate that all necessary tools and technologies for a meta-cloud already exist, but are not integrated or identified. To overcome subliminal generated vendor lock-ins, this paper proposes a reference architecture for an abstraction concept called lightweight virtualization cluster (LVC).

The remainder of this paper is structured as follows. Section 2 shows how inherent infrastructure dependencies are generated and what strategies exist to avoid, this is presented in a very condensed form. So the reader may also want to study [6] to get a better understanding how infrastructure dependencies are created in IaaS cloud computing. Nevertheless this paper should provide enough information for the reader to see that several shortcomings exist to understand vendor lock-in generating principles. Section 3 presents several popular open source projects for private PaaS platforms often mentioned by literature and community providing something, this paper calls a lightweight virtualization cluster abstraction to avoid remaining shortcomings identified in section 2. Section 4 takes a closer look on introduced open source projects for private PaaS platforms, and derives an reference architecture for a lightweight virtualization cluster concept as an abstraction layer on top of IaaS infrastructures. Finally, section 5 summarizes the key points how lightweight virtualization cluster abstractions are helpful to overcome vendor lock-in in IaaS cloud computing.

2. HOW VENDOR LOCK-IN EMERGES IN IAAS CLOUD COMPUTING

Vendor lock-in is defined to make a customer dependent on a vendor for products and services, unable to use another vendor without substantial switching costs. This is exactly the actual state of the art in IaaS (infrastructure as a service) cloud computing. On the one hand IaaS provides a commodity service. A commodity describes a class of demanded goods (in case of IaaS computing, storage and networking services) without qualitative differentiation. So the market treats instances of the goods as (nearly) equivalent. Or simpler: A Google Compute Engine provided virtual server instance is not better than an Amazon Web Services (AWS) provided virtual server instance. Both can be used to host user defined services. So IaaS provides commodities from a simple point of view.

On the other hand reality looks more complex. Although cloud computing provided services must be classified as commodity, a switch from one service provider to another service provider is in most cases not as easy as buying coffee from a different supplier. This is mainly due to inherent dependencies on underlying cloud infrastructures. These dependencies are often subliminal generated by cloud service provider specific (non standardized) service APIs.

Cloud computing vendor lock-in problems can be made obvious by looking at an example. Let us take a look on provided services by Amazon Web Services (AWS). We see that AWS provides over 30 services but only nine of them can be rated as commodity (see Kratzke [6] for a more detailed analysis). Using cloud computing means it is likely that (horizontal) scalable and elastic systems have to be deployed. So in most cases scalable distributed systems will be deployed to cloud infrastructures. Components of distributed systems need to interact and this can be tricky in details for system engineers (especially with scalable system designs). While the core components (virtualized server instances) of these distributed systems can be deployed using commodity services (that is in case of AWS mainly done using the EC2 service), all further services to integrate these virtualized server instances in an elastic and scalable manner are provided by non-commodity services. In case of AWS it is very likely that services like elastic load balancing (ELB), auto scaling, message queue system (SQS) are needed to design an elastic and scalable system on an AWS infrastructure. But especially these services are non-commodity services binding a deployment to vendor specific infrastructure (in case of our example to AWS).

2.1 Non-commodity Services Create Vendor Lock-in

So vendor lock-in emerges using non-commodity services provided by IaaS cloud service providers. IaaS cloud customers are systematically aided (by tutorials, training, white papers, documentation, support, convenience APIs and so on) to use non-commodity (convenience) services (like non standardized load balancing, messaging or auto scaling services) to solve hard problems of (horizontal scalable) distributed service design [6]. And exactly these non-commodity services are poor standardized and therefore hardly transferable between different cloud service infrastructures (IaaS).

To overcome these vendor lock-in problems paper [6] provides a more detailed analysis and derives the following seven features to be fulfilled by lightweight virtualization clusters.

- Feature #1: Hosting highly available deployed services
- Feature #2: Providing secure communication between services
- Feature #3: Providing convenient auto scalability
- Feature #4: Providing convenient load balancing
- Feature #5: Providing distributed and scalable service discovery and orchestration
- Feature #6: Enabling transfer of service deployments between LVCs
- Feature #7: Using formal description of service deployments

2.2 Existing Strategies to Avoid Vendor Lock-in

Reflecting actual research literature the following strategies can be identified to overcome vendor lock-in (see [6] for a more detailed discussion).

- **Industrial top down standardization approaches** like CIMI (Cloud Infrastructure Management Interface [7]) try to define a standardized API to different cloud service providers. So if it is

possible to deploy to a provider A via a standardized API it is also possible to deploy to a (standard compliant) provider B.

- **Open source bottom up approaches** try to do the same like the industrial standardization approach but applying a bottom up strategy. *e.g.* fog.io [8] is a Ruby-based API to access over 30 cloud service providers.
- More formal **research approaches** try to formalize the deployment description by defining mostly XML–based deployment languages (compare [9–12]).

Looking at industrial top down approaches (like CIMI) show that typical standardization problems arise like the least common denominator problem. The same is true for Open Source bottom up approaches like fog.io. A standardization or a bottom up developed open source library (abstraction layer) seems only to work for commodity, but not for non-commodity services necessary for deployments of typical complexity (load balanced, elastic and distributed). Furthermore companies like AWS are not known to be very standard oriented ("premature standardization considered harmful" is a famous statement from Werner Vogels, CTO of Amazon). And finally, formal deployment description language approaches proposed by several research institutions (see [9–12] for an impression) look promising but these deployed systems are not elastic (so do not provide automatic up and down scaling for instance). Furthermore they do not face the problem how to move a deployed distributed and elastic system from one service provider to another. So in fact vendor lock-in stays.

So far all of the above mentioned strategies seem to have shortcomings, especially in handling dynamic cloud deployments of typical complexity (so deployments which are elastic and load balanced).

3. LIGHTWEIGHT VIRTUALIZATION APPROACHES

Other approaches rely on so called lightweight virtualization approaches. These approaches are often referenced as PaaS platforms. Typical platforms are Apaches Mesos, Googles Kubernetes, RedHats OpenShift or CloudFoundry (see **Table 2**). This paper uses the term lightweight virtualization in the meaning of a convenient form to handle operating system virtualization.

3.1 Operating System Virtualization

Recently the tool Docker [13] and its container concept made operating system virtualization very popular especially for Linux based operating systems. **Table 1** shows that this approach has not been invented by Docker but exist almost every unixoid operating system. Docker (due to its convenient container concept) just made operating system virtualization popular and manageable for the DevOps community. But operating system virtualization approaches exist for almost every popular operating systems..

Table 1. Examples of Lightweight Virtualization Mechanisms on Various Operating Systems

Operating System	Virtualization mechanism	Link for further informations (last access 30th June 2014)
Linux	Docker	http://www.docker.io
	LXC	http://linuxcontainers.org
	OpenVZ	http://openvz.org
Solaris	Solaris Zones	http://docs.oracle.com/cd/E26502_01/html/E29024/toc.html
FreeBSD	Jails	http://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/jails.html
AIX	Workload Partitions	http://www.ibm.com/developerworks/aix/library/au-workload/index.html
HP-UX	Containers (SRP)	https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HP-UX-SRP
Windows	Sandboxie	http://www.sandboxie.com

Nevertheless IaaS cloud computing is heavily affected by hardware virtualization. Multiple virtual servers providing different operating systems can be run on one physical host system. So customers can run the operating system they want. Upcoming lightweight virtualization technologies like Docker [13] seem auspicious but rely on operating system virtualization. Maybe that is why operation system virtualization is seldom used for avoiding vendor lock-in so far. Using lightweight (operating system) virtualization the virtualized operating system has to be the same like the hosting operating system. So lightweight virtualization is limited on the one hand but one the other hand generates a better hosting density. In general it is possible to host more lightweight virtualizations on a physical host than with hypervisor based virtualizations. Especially unixoid systems support this form of lightweight virtualization (**Table 1**).

Operating system virtualization can be deployed to any public or private cloud service provider due to its lightweightness. Operating system virtualization can be frictionless applied on top of hypervisor virtualization and is therefore deployable to the commodity part of any public or private cloud service infrastructure.

So lightweight virtualization is a perfect abstraction layer for the commodity part of any IaaS cloud service infrastructure. Unlike industrial top down or open source bottom up standardization approaches, lightweight virtualization defines an abstraction layer out of the box only depending on commodity services (virtual servers). On the other hand this approach is restricted to operating system virtualization and therefore only applicable to unixoid operating systems from a pragmatic point of view. According to common server operating market shares Unix based servers are used by almost 2/3 of all public accessible websites. So the here presented approach is likely to be applicable to 2/3 of the relevant problem domain which is good from authors point of view.

3.2 PaaS Platforms on top of Operating System Virtualization

Table 2 shows several open source PaaS platforms providing such an abstraction layer. All projects have in common that they integrate multiple single servers into a cluster concept on a higher abstraction level (see **Figure 1**).

Table 2. Some Open Source Projects Providing PaaS Plattforms

Project	**Organization**	**Link for further informations (last access 16th July 2014)**
Mesos	Apache	http://mesos.apache.org
Kubernetes	Google	https://github.com/GoogleCloudPlatform/kubernetes
CoreOS	CoreOS	http://coreos.com
OpenShift	Red Hat	http://www.openshift.com
Cloud Foundry	Cloud Foundry Foundation	http://cloudfoundry.org

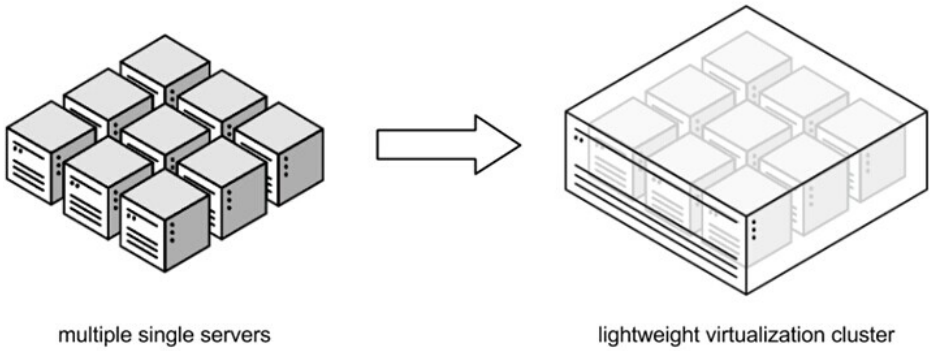

Figure 1. Lightweight Virtualization Cluster Approach (visualization inspired by and taken from mesosphere.io,http://mesosphere.io

Due to the higher abstraction level the deployment of distributed services does not have to deal with a multitude of single server nodes but can be run against a "virtual single but scalable server node" (consisting of several single servers). This virtual single and scalable server node is normally called a cluster and provides on the one hand a higher internal complexity than a single server node but also a reduced external complexity for deployment purposes. The cluster concept hides complexity-creating concepts like scaling, load balancing, fail-over and so on. These hidden complexity creating concepts are provided via internal components of the cluster and does not rely on any IaaS provided non-commodity services like the AWS auto-scaling or elastic load-balancing (ELB) service for instance.

3.3 Lightweight Virtualization Cluster Abstraction

So PaaS platforms (see **Table 2**) are using commodity services only. That is why these clusters are frictionless deployable to a multitude of IaaS cloud service providers or hypervisor based systems like vSphere by VMWare or even bare-metal systems. Platforms mentioned in **Table 2** provide something this paper calls a **lightweight virtualization cluster (LVC)** abstraction.

From the understanding of this paper a lightweight virtualization cluster abstraction layer must provide several features normally provided in a non-commodity way by cloud service providers (especially auto scaling, load balancing) but not provided by basic operating system virtualization tools like Docker [13] out of the box. A LVC has the job to host services, enable (load balanced and secure) communication between services, up and down scale services according to actual workloads. A LVC is a cluster and therefore consists of several hosts capable to host single services or complex service deployments (services

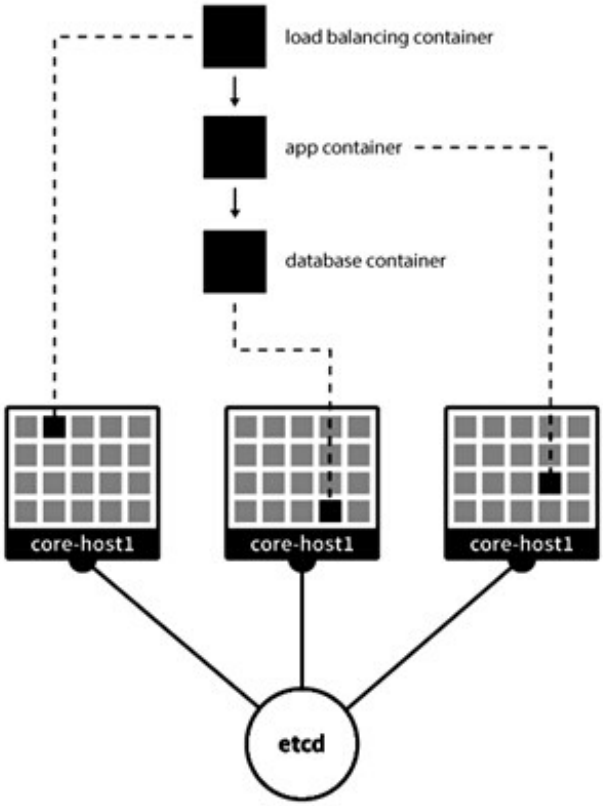

Figure 2. Service Deploying Using Core OS on Three Different Hosts (Visualization Taken from CoreOS, https://coreos.com/using-coreos/

consisting of other services). And furthermore a LVC must be deployable to any IaaS public or private cloud service or bare-metal infrastructure. **Figure 2** shows this by example of a classical three tier web app deployed to a Core OS cluster (as an example for a lightweight virtualization cluster). The three hosts shown in **Figure 2** could be provided by public or private IaaS, hypervisor hosted virtual machines or by bare-metal servers.

3.4 Core Features of a Lightweight Virtualization Cluster

LVCs support the following features which can be derived from actual strategies (see section 2 and [6]) and several open source projects providing lightweight virtualization cluster abstractions (see **Table 3**) to overcome vendor lock-in present in actual IaaS based cloud computing (see [6] for a more detailed discussion). LVCs should address furthermore common security, governance, and availability as well as vendor lock-in worries (see section 1).

Very often substantial showstoppers for cloud computing are due to higher order than pure technical considerations [3]. Due to legal regulatory it sometimes might be not possible to store and process data on an infrastructure not controlled and administered by the service provider itself. That is why transferability feature #6 is so important. Feature #6 states that it should be possible to transfer any deployed service to

Table 3. Core Features of a Lightweight Virtualization Cluster

		Common Cloud Worries [3, 5]			
Nr.	**Feature**	**Security**	**Availability**	**Governance**	**Vendor Lock-In**
1.	High available deployment of services		✓		
2.	Secure communication between services	✓		✓	
3.	Convenient auto scalability		✓		
4.	Convenient load balancing		✓		
5.	Distributed and scalable service discovery and orchestration		✓		
6.	Transfer of service deployments between LVCs			✓	✓
7.	Formal description of service deployments deployable				✓

any LVC (wherever it is hosted). So feature #6 and #7 provide the option in case of doubt to return every deployed service from a public cloud service provider back home[1] (into the on-premise data center to be in accordance to common governance rules).

4. A REFERENCE ARCHITECTURE FOR LIGHTWEIGHT VIRTUALIZATION CLUSTERS

A reference architecture (RA) provides a common vocabulary and generalizes a set of solutions. These solutions are structured for the depiction of one or more concepts based on the harvesting of a set of patterns that have been observed in a number of successful implementations. This is exactly done in this section. Section 4.1 defines a terminology used to derive functional components of a reference architecture in Section 4.2. Generalized solutions are reflected in Section 4.3 and aligned with the concepts identified and defined in paragraphs Section 4.1 and Section 4.2.

4.1 Terminology of a Reference Architecture for Lightweight Virtualization Clusters

Most readers will be not familiar with ontology/terminology design but with software design. Therefore this paper uses a UML class diagram (see **Figure 3**) to define and introduce core concepts and relations between these concepts of a lightweight virtualization cluster. Authors assume the reader to be familiar with UML notation. For better understanding concepts introduced in **Figure 3** are marked **bold** in the following explanations. The same is true for relations between concepts (UML associations in **Figure 3**). These are marked *italic* for better understanding.

[1] *Of course the transfer from in-house provided to public cloud provided is also possible.*

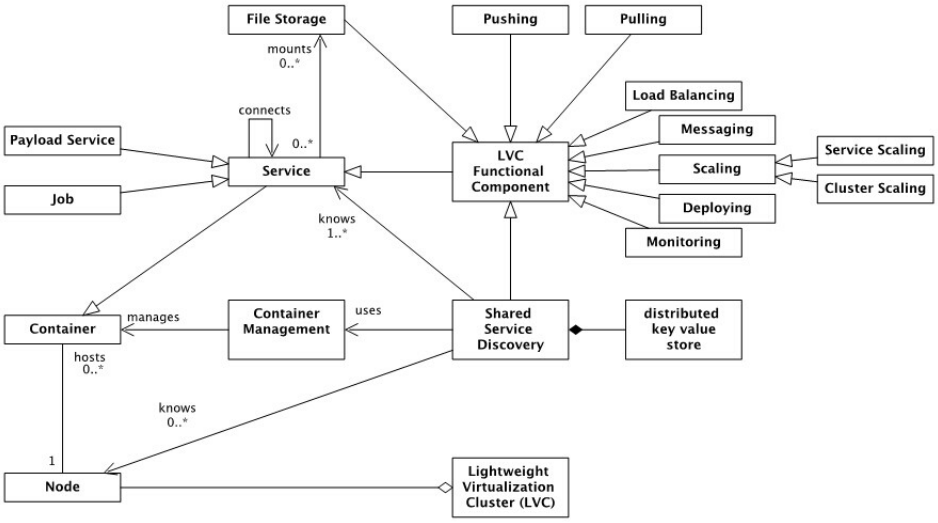

Figure 3. Terminology for a Reference Architecture of Lightweight Virtualization Clusters

A Lightweight Virtualization Cluster (**LVC**) consists of several nodes (**Node**) capable to *host* containers (**Container**) by operating system virtualization. Nodes can be seen as (virtual) servers provided via public or private IaaS cloud infrastructures, via hypervisor based virtualization, or even via bare-metal servers. Containers comprise arbitrary applications and their dependencies in a runnable form as an isolated user process on the host operating system.

A **Service** is any container registered with a LVC. A service can establish connections (*connects*) with other services. A service can *mount* file storage to store persistent data. Services can be grouped into three service categories.

1. A **Payload Service** is a service deployed to a LVC in order to provide continual service outcomes based on incoming requests (*e.g.* a web server hosting a website).
2. A **Job** is also a service deployed to a LVC but to provide a one-time service outcome (*e.g.* a print job).
3. A **Functional Component** is a service providing specific features necessary to provide complex scalable, elastic, distributed and transferable cloud deployments in an IaaS infrastructure independent manner. All functional components build the feature set of a LVC. Such functional components provide features like load balancing, auto scaling, file storage, monitoring, service discovery and orchestration, messaging and deploying.

4.2 Components of a Reference Architecture for Lightweight Virtualization Clusters

Above mentioned main concepts are typically layered in a lightweight virtualization cluster approach (see **Figure 4**). The hosting layer provides nodes on which containers can be run. A variable number of nodes therefore provide the hosting layer. Nodes can be provided via public or private IaaS cloud infrastructures (as commodities!), with hypervisors or as bare-metal. The amount of "physical" nodes is determined by **Cluster Scaling.** On these "physical" nodes containers are operated. **Container**s are

executed by an appropriate **Container Management** solution like the actual very popular Docker [13] tool.

The LVC layer integrates a variable number of "physical" nodes into a logical cluster. On top of this cluster jobs or user-defined payload services are executed without the necessity to know how these jobs and payload services are mapped to "physical" nodes. This is done transparently by LVC. LVC provides therefore the following functional components.

Heart of a LVC is a concept called **Shared Service Discovery** often realized by **distributed key value stores** like serf [14, 15] or etcd [16]. Shared Service Discovery has the responsibility to detect the deployment, shutdown or change of service containers in the LVC as well as to do or to trigger appropriate reconfigurations to keep user defined payload services of LVC functional components working.

Elastic service deployments rely on the capability to scale horizontal and to balance the load across multiple nodes. This is done normally by three interacting components of a LVC.

1. **Load Balancing** is responsible to distribute workloads across multiple Payload Services or LVC Functional Components. So it needs access to shared service discovery to identify necessary Services. Think about HAProxy as an example software.

2. **Monitoring** is responsible to monitor the processing, memory, disk and network performance of Payload Services or LVC Functional Components. Therefore it needs access to hosting nodes and their actual processing, memory, network and disk utilization metrics to provide adequate data for scaling decisions.

3. **Scaling** is responsible to handle growing or decreasing amount of processing, memory, disk or network resources of Payload Services or LVC Functional Components in order to provide additional (up scaling) or to shut down (down scaling) Payload Service or LVC Functional Component instances. Scaling needs access to metrics data provided by monitoring. Scaling can be done on two levels. **Service Scaling** can be realized within a given cluster size. **Cluster Scaling** is used to add or remove nodes to a cluster according to actual demand of resources of Payload Services or LVC Functional Components.

The other components of the LVC include the following:

The **File Storage** component has the responsibility to provide mountable and persistent storage of files for Payload Services or Jobs. Think about NFS mountable file servers as an example.

Deploying component has the responsibility to deploy services into the cluster. It needs access to shared service discovery to provide appropriate orchestration for the deployed service. Think about adding a new backend server to a load balancing service for example.

Pushing component has the responsibility to insert a service deployment into a LVC. The internal LVC deploying is done by the deploying component. In actual LVC implementations (see **Table 2**) pushing and deploying are considered to be identical (which might be correct from a pure technical point of view but not from a conceptual one). The reason for a pushing component is only obvious when we look at the conceptual counterpart (which is the pulling component).

Pulling component has the responsibility to extract a service deployment from a LVC. It is the counterpart of the pushing component. Using both components it is possible to extract a given service deployment from a LVC A using the pulling component and insert the extract using the pushing component of a LVC B. A and B can be hosted on completely different public or private cloud service provider

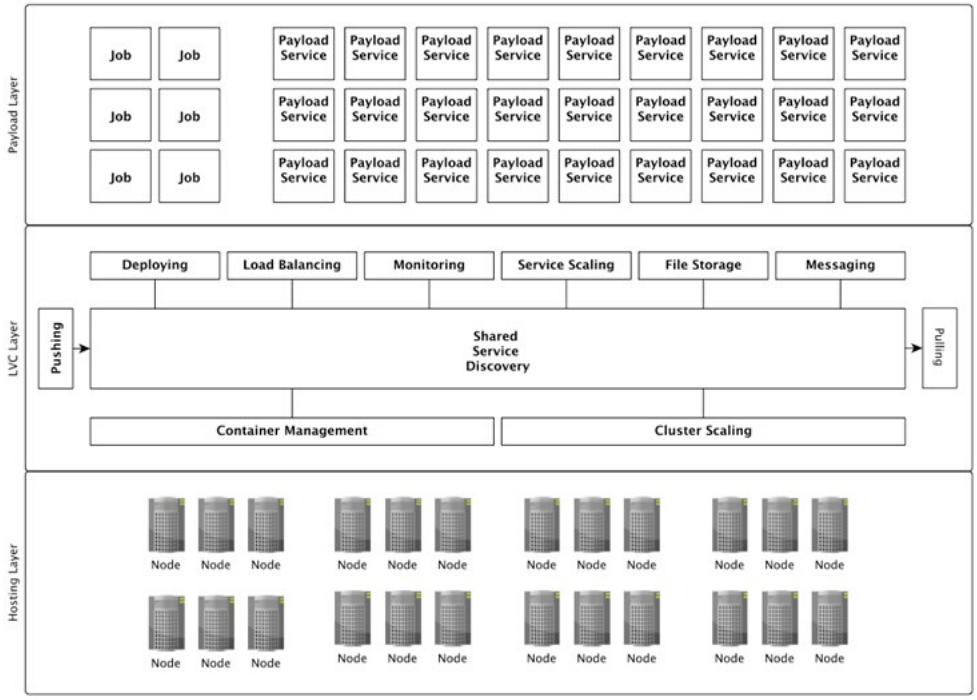

Figure 4. Reference Architecture of Lightweight Virtualization Clusters

infrastructures. Pushing and pulling component realize a transfer capability of cloud service deployments between arbitrary cloud service infrastructures (see **Table 4** feature 6).

Messaging component has the responsibility to provide publish subscribe features for messages necessary to being passed between services of the LVC. These messages can be used internally for cluster features provided by LVC functional components but also for user defined payload services. Some LVC realize messaging features by using a distributed key value store to pass messages (put a key with a value, the message) and react on messages by listening on changes of keys and/or values.

4.3 Using the Reference Architecture to Evaluate existing Approaches

Table 5 maps existing open source implementations of lightweight virtualization cluster concept to the reference architecture derived and visualized by **Figure 3** and **Figure 4**. As table 5 indicates the Cloud Foundry project seems very complete from the reference architecture point of view. Nevertheless if no load balancing, scaling or messaging features are required for a particular use case this very complex strategy might be overkill. CoreOS seems to follow a more minimal but less complex project and might be a better solution for less complex use cases. Other open source solutions like Mesos, OpenShift and Kubernetes seem to play somewhere between Core OS (minimal featured) and Cloud Foundry (maximum featured).

Nevertheless **Table 5** shows in particular that no one of the popular open source PaaS platforms (see **Table 2**) is providing a pulling component. According to our definition a pulling component has the responsibility to extract a service deployment from a LVC and is therefore the counterpart of the pushing component. Pushing and pulling components realize together a transfer capability of cloud service

Table 4. Core Feature To LVC Component Mapping

		LVC Components										
Nr.	**Feature**	**Deploying**	**Load Balancing**	**Monitoring**	**Service Scaling**	**Cluster Scaling**	**File Storage**	**Messaging**	**Container Management**	**Shared Service Discovery**	**Pushing**	**Pulling**
1.	High available deployment of services	✓	✓		✓	✓						
2.	Secure communication between services	✓					✓	✓	✓	✓		
3.	Convenient auto scalability			✓	✓	✓						
4.	Convenient load balancing		✓	✓					✓	✓		
5.	Distributed and scalable service discovery and orchestration	✓							✓	✓		
6.	Transfer of service deployments between LVCs	✓									✓	✓
7.	Formal description of service deployments deployable	✓									✓	✓

deployments between arbitrary cloud service infrastructures (see **Table 4** feature 6). So actual PaaS platforms provide the feature to bring services onto the platform (they support pushing) but provide little support to extract deployed services from a platform (they do not support pulling). It is like picking a tourist up a hotel to give him a site seeing tour but refusing him to show the way back to his hotel.

5. CONCLUSION

This paper derived a defined terminology and reference architecture for core components of lightweight virtualization approaches necessary to provide transferable and vendor lock-in free deployments of cloud-based service deployments.

Although the introduced reference architecture is subject to ongoing research it is in its actual state already a valuable instrument to evaluate different lightweight virtualization approaches for their completeness and provided features (see **Table 5**). It can be used to identify appropriate LVC solutions for cloud vendor lock-in avoiding strategies. One aspect we identified that popular PaaS platforms (see **Table 2**) do not provide features to extract service deployments from one cluster installation to insert the extract into another installation of cluster (hosted on different IaaS infrastructure provided by a different service provider). So no “copy” commands of service deployments can be executed in a convenient manner.

Most of the time the question is answered “How to get into the cloud?” but seldom the questions “How to get out of the cloud?” or “How to transfer service deployments between clouds?” are asked or even answered.

Ongoing research will address exactly this shortcoming and investigate how pushing and pulling components of LVCs have to be implemented in order to handle formal service deployment descriptions (*e.g.* presented in [9–11]) and provide the capability to transfer service deployments from one IaaS service

Table 5. Mapping of Example Open Source Projects Concepts to LVC Concepts

LVC Functional Component	**Mesos**	**Core OS**	**OpenShift**	**Cloud Foundry**	**Kubernetes**	**Comments**
Payload Service	Mesos slave, several frameworks	Docker	Cartridges	DEA, Buildpacks	Docker	Tools/technology used to capsule user defined services or jobs
Shared Service Discovery	ZooKeeper	etcd	geard	NATS, Cloud Controller, Service Broker	etcd	Tools used to provide service discovery and orchestration
File Storage	-	-	-	-	-	Tools and technology used to provide file storage for payload services or jobs
Load Balancing	-	-	HAProxy	Router	-	Tools used to provide load balancing over horizontal scalable payload services
Messaging	-	-	-	NATS	-	Tools used to provide publish subscribe based messaging for payload services
Scaling	-	-	Yes	yes	Kubernetes Controller Manager Server	Tools used to provide auto scaling of payload services
Deploying	Marathon, Mesos master	fleet	gear	Cloud Controller, Buildpacks, bosh	Kubernetes API Server	Tools used to deploy payload services within a LVC
Monitoring	Marathon	-	Yes	Health Manager	-	Tools used to monitor processing, network, memory and disk metrics of payload services
Pushing	Marathon, Mesos master	fleet	gear	Cloud Controller, Buildpacks, bosh	Kubernetes API Server	Tools to deploy payload services or jobs into the LVC
Pulling	-	-	-	-	-	Tools to extract payload services or jobs from the LVC

provider to another. This would enable to develop tools in order to overcome vendor lock-in problems actual present in IaaS cloud computing.

ACKNOWLEDGMENTS

This paper was written conducting the research project Cloud TRANSIT. Project Cloud TRANSIT is funded by German Federal Ministry of Education and Research (FHProfUnt2014-31P9227). Author thanks Lübeck University (Institute for Telematics) and fat IT solution GmbH for accompanying and supporting project Cloud TRANSIT as well.

References

[1] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” *NIST Special Publication 800-145*. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[2] J. Barr, A. Tetlaw, and L. Simoneau, *Host your website in the cloud: Amazon Web Services mad*

easy. SitePoint, 2010.

[3] N. Kratzke, "Cloud Computing Costs and Benefits An IT Management Point of View," *Cloud Computing and Services Sciences (Eds. Ivanov, van Sinderen, Shishkov)*, pp. 185–203, 2012.

[4] N. Kratzke, "Overcoming Ex Ante Cost Intransparency of Clouds Using system analogies and a corresponding cost estimation model," in *CLOSER 2011 1st International Conference on Cloud Computing and Services Science (Special session on Business Systems and Aligned IT Services BITS 2011)*, pp. 707–716, 2011.

[5] 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.

[6] N. Kratzke, "Lightweight Virtualization Cluster: How to overcome Cloud Vendor Lock-in," *Journal of Computer and Communications*, vol. 2, no. 12, 2014. to be published.

[7] DMTF, "CIMI (Cloud Infrastructure Management Interface)," http://dmtf.org/standards/cloud.

[8] Fog.io (The Ruby cloud services library) http://fog.io.

[9] C. de Alfonso, M. Caballer, F. Alvarruiz, G. Moltó, and V. Hernández, "Infrastructure Deployment Over the Cloud," in *Cloud Computing Technology and Science (CloudCom)*, pp. 517–521, 2011.

[10] G. Juve and E. Deelman, "Automating Application Deployment in Infrastructure Clouds," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pp. 658–665, IEEE, 2011.

[11] A. Lenk, C. Danschel, M. Klems, D. Bermbach, and T. Kurze, "Requirements for an IaaS deployment language in federated Clouds," in *Service-Oriented Computing and Applications (SOCA), 2011 IEEE International Conference on*, pp. 1–4, IEEE, 2011.

[12] S. Murphy, S. Gallant, C. Gaughan, and M. Diego, "U.S. Army Modeling and Simulation Executable Architecture Deployment Cloud Virtualization Strategy," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pp. 880–885, IEEE, 2012.

[13] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[14] "Serf," http://www.serfdom.io/docs/index.html.

[15] A. Das, I. Gupta, and A. Motivala, "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol," in *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pp. 303–312, IEEE, 2002.

[16] etcd https://github.com/coreos/etcd.

About This Journal

MCCC is an open access journal published by Scientific Online Publishing. This journal focus on the following scopes (but not limited to):

- Autonomic Business Process and Workflow Management in Clouds
- Cloud Composition, Federation, Bridging and Bursting
- Cloud Computing Consulting
- Cloud Configuration, Performance and Capacity Management
- Cloud DevOps
- Cloud Game Design
- Cloud Migration
- Cloud Programming Models and Paradigms
- Cloud Provisioning Orchestration
- Cloud Quality Management and Service Level Agreement (SLA)
- Cloud Resource Virtualization and Composition
- Cloud Software Patch and License Management
- Cloud Workload Profiling and Deployment Control
- Cloud Video and Audio Applications
- Economic, Business and ROI Models for Cloud Computing
- Green Cloud Computing
- High Performance Cloud Computing
- Infrastructure, Platform, Application, Business, Social and Mobile Clouds
- Innovative Cloud Applications and Experiences
- Security, Privacy and Compliance Management for Public, Private and Hybrid Clouds
- Self-service Cloud Portal, Dashboard and Analytics
- Storage, Data and Analytics Clouds

Welcome to submit your original manuscripts to us. For more information, please visit our website: http://www.scipublish.com/journals/MCCC/

You can click the bellows to follow us:

- Facebook: https://www.facebook.com/scipublish
- Twitter: https://twitter.com/scionlinepub
- LinkedIn: https://www.linkedin.com/company/scientific-online-publishing-usa
- Google+: https://google.com/+ScipublishSOP

SOP welcomes authors to contribute their research outcomes under the following rules:

- Although glad to publish all original and new research achievements, SOP can't bear any misbehavior: plagiarism, forgery or manipulation of experimental data.
- As an international publisher, SOP highly values different cultures and adopts cautious attitude towards religion, politics, race, war and ethics.
- SOP helps to propagate scientific results but shares no responsibility of any legal risks or harmful effects caused by article along with the authors.
- SOP maintains the strictest peer review, but holds a neutral attitude for all the published articles.
- SOP is an open platform, waiting for senior experts serving on the editorial boards to advance the progress of research together.