MultiBox: Lightweight Containers for Vendor-independent Multi-cloud Deployments

James Hadley, Yehia Elkhatib, Gordon S. Blair, and Utz Roedig

Lancaster University, Lancaster LA1 4WA, UK
j.hadley1@lancaster.ac.uk

This is a pre-print. The final version is available at:
http://link.springer.com/chapter/10.1007/978-3-319-25043-4_8

Abstract. Cloud computing aims to make a large selection of sophisticated technologies available to users for deployment and migration. In reality, once a cloud service provider has been chosen, migration is often a costly and time-consuming process. This paper presents MultiBox, a lightweight container technology that facilitates flexible vendor-independent migration. Our framework allows its users to deploy and migrate almost any application in its normal state with minimal computational and network resource overheads. We show that the performance overhead of deploying within a lightweight container is 4.90% of the resources available to an average VM and downtime during a migration is less than the time needed to scale a server using provider-centric tools.

Keywords: cloud computing, multi-cloud systems, containers, workload migration

1 Introduction

The vision behind cloud computing is about liberating applications from the underlying resources, and allowing them to flexibly adapt according to their demand in order to optimise their operation. The current reality of cloud computing, however, is that such idealistic freedom does not exist. Cloud service providers (CSPs) do not support cross-provider workload management or migration. Some standardisation efforts subsist[1] but are yet largely disregarded by CSPs (for obvious commercial reasons) and by developers alike [1]. In face of such well-documented vendor-lockin (cf. [2, 3]), workload migration becomes a challenging, manual and bespoke effort to take into account the CSPs' divergent application programming interfaces (APIs), the multitude of heterogeneous services, and disparate pricing schemes.

Additionally, current workload migration approaches are inattentive to network overheads which makes them unsuitable for deployments over long-latency networks and, potentially, rather costly. This limits many application deployments both geographically and dynamically, posing serious restrictions over the

[1] http://www.occi-wg.org/

ability of an application to adapt to its demand. This is unsatisfactory at a time when application lifespans are becoming more volatile in light of agile development cycles of different resource usage profiles, and the use of social media that can cause sudden and dramatic unexpected demand fluctuations and geographical spread.

In this paper, we define workload fluidity as the capability to migrate one or more applications from one CSP to another at short notice with little or no human intervention. Our key contribution is a method of achieving workload fluidity by utilising extremely lightweight containers based on a relatively new addition to the Linux kernel, known as control groups (cgroups)[2].

We use the term 'container' to refer to a lightweight isolated environment wherein applications can be deployed, whilst staying decoupled from the host. Such execution environment is similar to a virtual machine (VM) but without a complete operating system (OS). Crucially, we focus on a highly flexible decoupling that supports an extremely low performance overhead, and reduces migration time and complexity to an absolute minimum.

The contributions of our MultiBox framework we present here are as follows:

1. A means of deploying both stateful and stateless applications (in their normal condition) to an environment decoupled from the host and other applications running inside it;
2. Our containers do not require the cooperation of CSPs as long as they provide Linux VMs, a novel contribution;
3. Reducing the impact of migrating an application when compared to other means including CSP-centric tools; and
4. Accommodating a wider range of applications with less complexity and performance overhead than other similar approaches.

These contributions are aimed at reducing development time spent on workload management, and at expanding a business's choice of CSPs by negating the need for CSP cooperation. They also support workload incubation in a local execution environment and cost-effective deployment and migration where connectivity is limited and/or costly.

The remainder of the paper is organised as follows: §2 outlines related work. §3 describes the MultiBox requirements and design. §4 details the technical implementation. §5 evaluates workload migration using MultiBox through use cases on two contrasting CSPs. §6 concludes and highlight future directions.

2 Related Work

Existing work on cloud workload management falls into two categories: *inside-out* and *outside-in*. Inside-out approaches focus on altering the application code to manage dependencies and enable cross-compatibility. This is highly complex as it requires the developers to be aware of the needs of the application before

[2] https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

it is deployed and to alter the programming process to meet the needs of the toolkit being used [4–6]. Outside-in approaches, on the other hand, focus on managing the cloud infrastructure that supports given applications so that the same business objectives can be achieved via cloud brokerage. This generally limits an environment to one stateless application. The role of the cloud broker can be highly complex as it needs to be aware of the application, the CSPs involved and the interfaces to connect to each CSP. It may also include a decision-making engine supporting policies, negotiation and enforcement [7–9].

What is considered 'traditional' cloud deployment management is done through VMs, a heavyweight task requiring substantial computational resources to run a complete guest OS as well as high bandwidth for migrating VMs. Current literature on workload migration deals in one way or another with the substantial overhead of shifting VM images either within (cf. [10, 11]) or between datacenters (cf. [12, 13]). However, all previous work avoids the real issue of inter-cloud migration by assuming a common API either of a single CSP (the former examples) or a federation of clouds (the latter).

Another approach is to avoid shifting heavy VMs and instead attempt to recreate deployments through Configuration Management Tools (CMTs). CMTs, such as Chef[3] and Ansible[4], allow for the definition of an infrastructure in the form of code. Apart from the fact that they are catered towards VM-based appliances not containers, CMTs are transitional and non-deterministic: They assume a certain initial state, usually a blank OS, then alter this state rather than defining it. This more often than expected leads to undesirable results in different environments (CSPs, OSs) [14].

To our knowledge, no other work has tackled workload migration as represented by containers as a resource-efficient and CSP-independent solution.

Previous efforts, however, have differed on defining a container. Some have a limited view and only encapsulate a certain application such as a web service [15, 16] as a monolithic appliance. The Elastic Application Container (EAC) [17] has a more generic definition that includes any application, but only allows for one instance per container. The EAC architecture prioritises container scalability and low overhead but not portability: EACs are managed by an Elastic Application Server which is analogous to a hypervisor running in the host OS. In contrast, our containers run using the Linux native cgroups, further reducing management overhead and supporting portability to any Linux host.

Our definition of a container agrees with those of technologies such as Docker[5], Linux-VServer[6] and OpenVZ[7]: a highly flexible abstraction of OS capabilities to enable applications to run in a virtual environment with low performance overhead, and at low migration time and complexity. Many of the mentioned solutions, however, are designed to cater for immutable appliances. They do not

[3] http://www.chef.io/
[4] http://www.ansible.com/
[5] http://www.docker.com/
[6] http://linux-vserver.org/
[7] http://openvz.org/

include system services and daemons necessary for running stateful applications or operating multiple thereof.

3 MultiBox Design

We opt for a minimalist container-driven design to offer the user flexibility, low performance overhead and the capability to migrate applications quickly. We draw a parallel between the exokernel[8] approach to OS kernels and our approach to containers in that we "give as much safe, efficient control over system resources as possible" [18]. To do this, we design a lightweight implementation that makes heavy use of Linux control groups, an existing feature in modern Linux kernels.

3.1 Requirements

Our container implementation must be lightweight so that it does not occupy noticeable resources on the VM. This is important for two reasons. First, resources occupied on the VM are unavailable to the container. A sizeable implementation could mean that a tradeoff emerges between cost savings in migratability and cost savings in VM size. Second, a smaller implementation is more easily portable. As our key aim is portability, this can be partially achieved by reducing the number of framework files copied and compiled on each VM.

In addition to its weight, the flexibility offered to the developer is significant. A flexible framework allows developers to deploy their applications without modifications. Typically an application may require system resources including users, devices and storage. All of these must be visible through the container abstraction layer.

Finally, we consider compatibility. CSPs offer heterogeneous operating system templates. There may be variance in the choice of offered distributions, versions, architectures and packages. To facilitate mutli-cloud deployments, we must ensure compatibility with as many CSPs as possible by connecting to the commonalities.

3.2 Design Choices

By utilising key features present within the OS's kernel already, there is little duplication of code, little to install or ensure compatibility with above the kernel and little software to transport from one server to another during migration. We prioritise utilising kernel support and adopt a lightweight approach where this is not possible. We rely on a one-to-one relationship between hosts and containers to reduce the number of required costly features and restrictions. For example, we do not superimpose a virtualised file system (such as Ploop[9] or AUFS[10],

[8] Exokernel is an operating system kernel that forces as few abstractions as possible onto developers.
[9] http://openvz.org/Ploop
[10] http://aufs.sourceforge.net/aufs.html

which can offer additional resource management tools) on to that of the host. Additionally, we deploy our management toolkit as a single init script with a number of connected system ‘services’. The toolkit is thus simple to use, simple to call automatically and enables rapid installation on any Linux server.

We also rely on delta synchronisation for deployment and thus offer key system devices and file system read-write access where other container technologies, that rely on pre-specified metadata for deployment, do not. Specialised packages are not needed, which enables compatibility with a wider range of applications, including stateful applications, and enables application developers to deploy their application as they would normally.

We recognise that migration is necessary when moving any stateful application and aim to automate the process and reduce the migration time as much as possible. Migrating an application manually from one server to another involves installing an OS, adding common libraries, application dependencies and the application itself and finally installing any data upon which the application relies. However, by separating the running environment from the host’s OS, this process can be automated. Thus, the handover time from one server to another is reduced and no manual intervention is needed.

3.3 System Components

A MultiBox container is a versatile execution environment supporting various processes, which are unaware of the remainder of the system. By using cgroups, a kernel module that provides support for the separation of processes into namespaces, a container is created as a namespace within the host to isolate its own processes, system and network devices, and file system (as depicted in Figure 2). The functionality given to a container constitutes a smaller subset of an OS than that of other container managers to ensure that operation and migration overheads are minimised. It also includes the facilities for services, daemons, syslog, cron and running multiple applications. It makes use of the host’s file system to minimise complexity, maximise performance and support stateful applications.

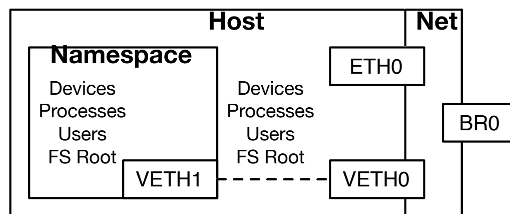

Fig. 1. The MultiBox Architecture

MultiBox containers are managed by a Container Manager that creates the namespace, routes network traffic to the host and its containers, and allocates resources to the container to support running applications. It also facilitates user interaction with the namespace. The Container Manager manages the few dependencies that are required outside the container. These include certain OS networking capabilities and a compiler.

A repository is maintained from which subsequent deployments will copy an OS template. Deltas, including the application and its dependencies, are then synchronised from a separate location in a second pass. The duration of this second pass depends partly on the type of application that is running within the container, though it is expected to be small relative to the data copied in the first pass. It also depends on the connection between the two VMs. In general, we expect that the first pass involves more data transferred at a higher speed than the second pass. This accounts for a potentially greater latency and lower bandwidth between the two servers at this stage, enabling a local or development server to synchronise with a production server elsewhere.

4 Implementation

The implementation consists of two key elements: kernel support for cgroups and the container manager. We now describe how these elements are implemented.

4.1 Kernel Support for Control Groups

Control groups allow for a system's resources to be divided into namespaces. They were designed to segregate concerns within a large system that, for example, may operate as a file server and a web server. These namespaces can then be treated as separate systems. This works via a kernel module, written in the C language, that isolates processes into groups. Allocated or unallocated resources, including memory, disk and network, can be assigned to a portion of the system. Devices can also be presented to the namespace. This portion of the system has a directory within the main system that it uses as its root, similar to a chroot environment. The capability to assign resources as well as processes to a namespace makes control groups suitable for containers.

Kernel support is needed to run control groups within a VM. The lightweight cgroups package provides this functionality that can be compiled into a standard 3.x kernel. To facilitate easy deployments, we compiled Kernel 3.14.22 with support for control groups. We also included support for paravirtualised environments to enable compatibility with a larger number of CSPs. The built kernel was compiled into an RPM package to enable easy installation on RedHat Linux systems such as those running RHEL, CentOS and Scientific Linux. We chose this type of package to support a wide range of OSs easily, though other packages could also be created to support other distributions.

4.2 Container Manager

The container manager utilises cgroups to prepare a VM to run a container and to migrate a container from a separate VM. The container manager was implemented as an init script supporting the five following commands.

The `prepare` command runs on a fresh VM to download and install the prebuilt kernel, toolkit and other packages necessary to support containers from a central repository. It also makes configuration changes to run the new kernel, support the OS-level IP forwarding needed to forward connections from the VM to the container and creates the base directory for the container.

The `create` command initiates a fresh container the first time that containers are implemented in an environment. It downloads and extracts a blank OS template file into the container's root directory from a remote repository.

When a container is ready to be run, `boot` is invoked. It also runs as part of the VM's subsequent start-up sequences. It begins by detaching any existing containers running on the VM by analysing the running processes and active devices on the VM. It then creates a new container within a background process on the VM, utilising the OS files located in the container's root directory. A virtual Ethernet and network bridge are created to support network communication between the container and the Internet. We assume that the VM has one IP address allocated to it and nothing else running on it. Thus, we move SSH on the VM to port 65535 and forward traffic on all other ports to the container. Then, a virtual Ethernet device is moved to the container's namespace and traffic is routed through it. Finally, a subset of the standard OS initialisation commands are run within the container to start application-dependent services.

`sync` creates a new SSH key pair if one does not exist on the VM. It uploads the public key to the old VM and runs `resync` for the first time. This is intended to be run when a migration is requested. Finally, the `resync` command synchronises the container's root directory from one host to another. It does this by downloading deltas from the old VM using the `rsync` utility present in Linux.

4.3 Overall Workflow

The container manager was integrated with a backend PHP program to make API calls to CSPs. The overall process is as depicted in Figure 2. We have adopted a final offline synchronisation pass as live migration techniques used in deployments such as [19] would not have improved efficiency where an IP change is necessary. Each step in the workflow diagram is performed once although file deltas could be copied several times where the ratio of network throughput to file changes is low.

We now examine each step in detail.

Allocation: A new VM is created at a CSP. This is performed by the backend application using calls to the CSP's API. Most CSPs expose powerful APIs, many of which can also be used to collect data about the CSP's offerings. As

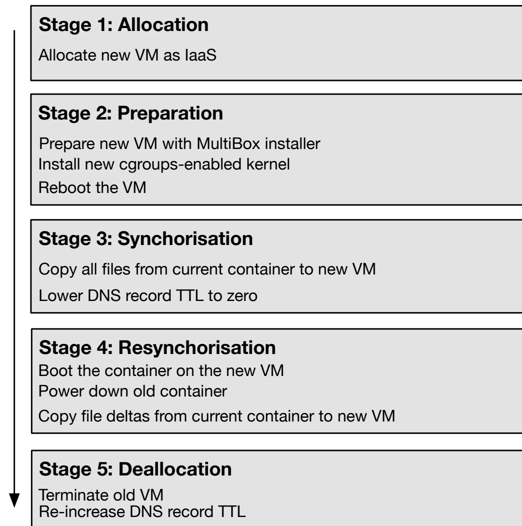

Fig. 2. The MultiBox Workflow

such, decisions could be made in real time concerning which CSP to deploy to, based on usage data already collected and/or user preferences.

Preparation: Preparing a new VM with the MultiBox installer consists of a single call to the MultiBox program. This command is also run by the backend application. The MultiBox program downloads and installs system dependencies, including the cgroups-enabled kernel. It also makes changes to the operating system to support IP forwarding.

Synchronisation: Files are copied using the `rsync` Linux utility, which is based on the SSH protocol. The TTL on the DNS record is lowered by the PHP application via a call to the DNS provider's API. A TTL of zero ensures that the record is not cached, increasing the overhead on the DNS service but ensuring the fastest transition.

Resynchronisation: The boot process on the new container depends on the operating system to be booted. For example, on Red Hat Linux, the `chkconfig` utility provides a list of services to be started at runtime. This is used to generate a list of commands for execution. These commands are run in a shell in the container's namespace. Commands are also run on the new VM, outside the container to forward traffic to and from the container. The container on the old VM is shut down via a Shell command.

Deallocation: The old VM is terminated via a call to the CSP's API and the DNS record is re-increased via a call to the DNS provider's API.

5 Evaluation

We confirm the efficacy of our framework in migrating stateful and stateless applications between clouds by successfully migrating a running Minecraft game server and an Apache web server from Softlayer to Vultr. In the case of the stateful application, a client connected to the game server was able to reconnect within a few seconds without changes to the connection settings or the game's state. In the case of the stateless application, a client browsing the hosted website did not notice any interruption.

The result of only migrating *moving parts*, i.e. application-specific logic, data and dependencies, offers improvements over full VM migration and recipe-based deployment. In full VM migration, the whole OS and shared libraries are moved. Such migration process moves *gigabytes* of data across the network and thus cannot be achieved without noticeable interruption (and cost). By contrast, the moving parts generally account for *megabytes* of data. In recipe-based deployment, there is no support for the deployment and migration of stateful applications and developer time is under greater demand with regard to recipe and repository creation and maintenance.

We continue to evaluate the efficacy of our framework by analysing two metrics. First, we compare the migration-related delay and downtime with a stateful and a stateless application by scaling the existing VM using provider-centric tools. Second, we measure the performance penalty of deploying and running an application within our framework relative to deploying directly to the VM.

5.1 Deployment Overhead

To measure the performance impact of running our framework, we deployed an instance on the Softlayer cloud[11]. We first ran the Geekbench Linux benchmarking utility to obtain a score for the memory and CPU performance in 32-bit mode. We then created a container and ran the same test within the container.

The Geekbench benchmarking utility produces scores against a 2500 baseline score; the higher the score the better the performance. Geekbench yielded a multi-threaded score of 7709 outside the container and 7331 inside the container. Whereas both scores are acceptable for a VM of 4 CPU cores and 8GB memory, there is a reduction in performance of 4.90% inside the container. As the container's overhead is fixed, this overhead is expected to form a larger percentage of available resources on a small VM and a smaller percentage on a larger VM.

5.2 Migration Time

To accurately measure the migration delay and downtime, we install a stateless web server in one container and a stateful client-server game, Minecraft, in another. We then construct a simulator in PHP to initialise the migration process

[11] We ran this test on Softlayer only as resources on the Vultr cloud are subject to the 'noisy neighbours' phenomenon.

using our framework and to record the time at different points. Both the stateful and stateless servers are accessible via a DNS "address" record with its TTL set to zero. The DNS server is external.

The simulator initially prompts the user for the VM's current IP address, and the destination CSP of choice from Softlayer and Vultr. Softlayer, IBM's public cloud offering, represents larger clouds with greater elasticity, complexity and cost, while Vultr represents smaller clouds at the other end of the respective spectrums. Each CSP has a different delay and downtime profile.

The simulator then creates a new VM with the chosen CSP, connects to the new VM and runs `prepare` and `sync`. Then, it connects to the DNS server to update the corresponding record and runs `resync`. Timestamps are taken at the beginning, after the VM creation, after running `sync` and upon completion. Thus, we obtain three time durations: the time to create the VM at the CSP; the time to prepare, sync and boot the container; and the final switchover time.

We also continuously connect to the server running within each container. For the stateless web server, this is achieved by periodically fetching a web page. For the stateful game, it is achieved by running an instance of the game's client elsewhere. In both cases, the server is contacted via the DNS name. We record the downtime, i.e. the time during which no server is available at that address.

We compare these times with the times taken to scale the server at the current CSP. This is acquired by starting a timer when an upgrade process is initialised using the CSP's own toolkit for greater CPU, memory and disk resources. The time taken after the request is created but before the server is shutdown is recorded. Additionally, the time taken for the server to be booted with the new resource set is recorded.

Figure 2 shows a comparison between these metrics after 500 seconds. Note that in each case, the delay before taking a server offline was greater with our migration framework but that period during which service was unavailable was greater with the CSP-provided tools. This result is quite significant as the delay can often be predicted and planned to reduce the impact on business operation. Furthermore, errors during the migration process can be easily recovered from via re-deployment, whereas errors during the upgrade process are more complex to recover from.

5.3 Human Costs and Portability

The cost of deploying and re-deploying (i.e. migrating) MultiBox containers on the developer is very low. The process is deterministic as dependencies are explicitly defined and natively provisioned. This is in comparison to CMTs where the desired execution environment is surprisingly not guaranteed across CSPs and OSs [14].

MultiBox supports portability of containers by design. The MultiBox implementation (kernel and command line tools) are OS agnostic, enabling easy deployment to any Linux-based host. This is significant as CSPs offers different sets of Linux distributions. Such Linux-native support sets MultiBox apart in its support for container migration from all other similar efforts.

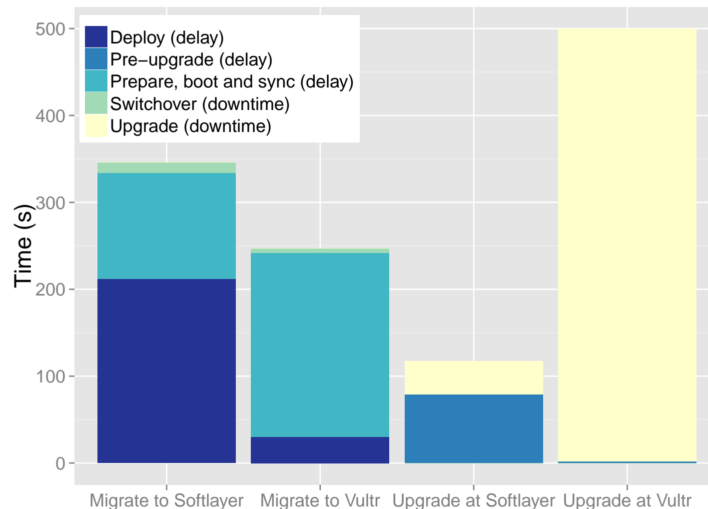

Fig. 3. Plot of migration and upgrade-related delays at two CSPs

6 Conclusion

This paper presented MultiBox, a means of creating and migrating containers. MultiBox containers are isolated form the rest of the host OS through the use of Linux cgroups to create namespaces. MultiBox containers can support multiple stateful processes as well as other OS-level services and file system.

The Linux-native aspect of MultiBox containers offer great advantages: they are transferable to any CSP infrastructure that supports any Linux variant. This means that CSP cooperation is *not* required, which is a groundbreaking advancement in the area of cross-cloud computing. Furthermore, MultiBox containers are lightweight by design and migrating them is significantly more resource efficient than other cloud workload migration approaches.

Also presented in the paper is a preliminary evaluation of MultiBox management and migration overheads. More experiments using larger cloud deployments and a thorough analysis of the different overheads is planned as future work.

References

1. Johnston, S.: Simple workload & application portability (swap). In: The 1st international workshop on CrossCloud computing (CrossCloud). (2014) 37–42
2. Nguyen, D.K., Taher, Y., Papazoglou, M., van den Heuvel, W.: Service-based application development on the cloud: State-of-the-art and shortcoming analysis. In: Int'l Conf. on Cloud Computing and Services Science (CLOSER'12). (2012)
3. Petcu, D., Macariu, G., Panica, S., Crăciun, C.: Portable cloud applications—from theory to practice. Future Generation Computer Systems (2013) 1417–1430

4. Vadhiyar, S.S., Dongarra, J.J.: SRS: A Framework For Developing Malleable and Migrateable Parallel Applications For Distributed Systems. Parallel Processing Letters **13** (2003) 291–312
5. Guillén, J., Miranda, J., Murillo, J.M., Canal, C.: A service-oriented framework for developing cross cloud migratable software. The Journal of Systems & Software (2013) 2294–2308
6. Miranda, J., Guillén, J., Murillo, J.M., Canal, C.: Enough about standardization, let's build cloud applications. In: Proceedings of the ACM WICSA/ECSA 2012 Companion Volume. (2012) 74–77
7. Pawluk, P., Simmons, B., Smit, M., Litoiu, M., Mankovski, S.: Introducing STRATOS: A Cloud Broker Service. In: Int'l Conf. on Cloud Computing (IEEE CLOUD). (2012) 891–898
8. Nair, S., Porwal, S., Dimitrakos, T., Ferrer, A., Tordsson, J., Sharif, T., Sheridan, C., Rajarajan, M., Khan, A.: Towards secure cloud bursting, brokerage and aggregation. In: European Conf. on Web Services (IEEE ECOWS). (2010) 189–196
9. Samreen, F., Blair, G.S., Rowe, M.: Adaptive decision making in multi-cloud management. In: The 2nd international workshop on CrossCloud computing (CrossCloud). CCB'14, ACM (2014) 4:1–4:6
10. Hirofuchi, T., Nakada, H., Itoh, S., Sekiguchi, S.: Enabling instantaneous relocation of virtual machines with a lightweight vmm extension. In: Int'l Conf. on Cluster, Cloud and Grid Computing (IEEE/ACM CCGrid). (2010) 73–83
11. Han, R., Guo, L., Ghanem, M., Guo, Y.: Lightweight resource scaling for cloud applications. In: International Symposium on Cluster, Cloud and Grid Computing (IEEE/ACM CCGrid). (2012) 644–651
12. Celesti, A., Tusa, F., Villari, M., Puliafito, A.: Improving virtual machine migration in federated cloud environments. In: Int'l Conf. on Evolving Internet (INTERNET). (2010) 61–67
13. Cerroni, W.: Multiple virtual machine live migration in federated cloud systems. In: The 1st international workshop on CrossCloud computing (CrossCloud). (2014) 25–30
14. Zhu, L., Xu, D., Xu, X.S., Tran, A.B., Weber, I., Bass, L.: Challenges in practicing high frequency releases in cloud environments. In: Int'l Workshop on Release Engineering, Mountain View, USA (2014) 21–24
15. Mohamed, M., Yangui, S., Moalla, S., Tata, S.: Web service micro-container for service-based applications in cloud environments. In: Int'l Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (IEEE WETICE). (2011) 61–66
16. Yangui, S., Mohamed, M., Tata, S., Moalla, S.: Scalable service containers. In: Int'l Conf. on Cloud Computing Technology and Science (IEEE CloudCom). (2011) 348–356
17. He, S., Guo, L., Guo, Y., Wu, C., Ghanem, M., Han, R.: Elastic application container: A lightweight approach for cloud resource provisioning. In: Int'l Conf. on Advanced Information Networking and Applications (AINA). (2012) 15–22
18. Engler, D.R.: The Exokernel Operating System Architecture. PhD thesis, Massachusetts Institute of Technology (1998)
19. Clark, C., Fraser, K., Hand, S., Hansen, J.G., Jul, E., Limpach, C., Pratt, I., Warfield, A.: Live migration of virtual machines. In: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2, USENIX Association (2005) 273–286