Armstrong *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2015) 4:17
DOI 10.1186/s13677-015-0042-8

Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Contextualization: dynamic configuration of virtual machines

Django Armstrong[1*†], Daniel Espling[2†], Johan Tordsson[2], Karim Djemame[1] and Erik Elmroth[2]

Abstract

New VM instances are created from static templates that contain the basic configuration of the VM to achieve elasticity with regards to capacity. Instance specific settings can be injected into the VM during the deployment phase through means of contextualization. So far this is limited to a single data source and data remains static throughout the lifecycle of the VM.

We present a layered approach to contextualization that supports different classes of contextualization data available from several sources. The settings are made available to the VM through virtual devices. Inside each VM data from different classes are layered on top of each other to create a unified file hierarchy.

Context data can be modified during runtime by updating the contents of the virtual devices, making our approach the first contextualization approach to natively support recontextualization. Recontextualization enables runtime reconfiguration of an executing service and can act as a trigger and key enabler of self-* techniques. This trigger provides a service with a mechanism to adapt or optimize itself in response to a changing environment. The runtime reconfiguration using recontextualization and its potential gains are illustrated in an example with a distributed file system, demonstrating the feasibility of our approach.

Keywords: Cloud Computing; Contextualization; Recontextualization; Configuration; Virtual machine

Introduction

One of the key characteristics of cloud computing is rapid elasticity [1]; the ability to quickly provision or release resources assigned to a cloud service in order to respond to current demand. In the Infrastructure as a Service (IaaS) model, cloud services are normally comprised of a set of different components, each defined using a Virtual Machine (VM) template. To achieve elasticity, the capacity of a cloud service can be adapted during runtime by adjusting the number of running VM instances of each template. This makes it possible to scale each part of the service independently.

Each VM instance started from a template needs to be customized with some unique settings, e.g., networking configuration to ensure each instance is assigned a unique IP-address. The settings need to be applied dynamically, normally done as part of the VM boot process. This boot-time customization process is called contextualization [2, 3].

Several different usage scenarios that join isolated cloud infrastructures together into a larger unified infrastructure are being considered [4, 5]. Conceptually, a virtual market of cloud resources not limited by technological boundaries would offer diversity in terms of (but not limited to) pricing, availability, and a choice of different geographical locations to use for hosting. Cloud infrastructures commonly offer supporting services such as network based storage, and today's major infrastructure providers (IPs) such as Amazon wrap the infrastructure specific functionality into pre-configured VM templates. In a multi-cloud scenario the VM templates need to be generic enough to be deployed across a wide range of infrastructures. Therefore, the contextualization stage is needed to support interactions with any infrastructure-specific services or settings.

VMs can be migrated between physical hosts without being restarted using live-migration. This enables an IP to re-distribute the current VM load across available server resources during runtime. Because the VM is not

*Correspondence: scsdja@leeds.ac.uk
†Equal Contributors
[1]School of Computing, Faculty of Engineering, University of Leeds, LS2 9JT Leeds, UK
Full list of author information is available at the end of the article

Springer

© 2015 Armstrong et al. This is an Open Access article distributed under the terms of the Creative Commons Attribution License (http://creativecommons.org/licenses/by/4.0), which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly credited.

restarted during the migration, the contextualization procedure is not triggered post-migration. Therefore, any VM customization (such as loading infrastructure specific settings) will remain unchanged.

In our earlier work [6] we present and define recontextualization, an emerging technology to allow contextual data inside running VMs to be updated during runtime. This work extends on earlier work on recontextualization and contextualization [3] and presents a novel layered based approach to contextualization. In this approach, multiple independent sets of contextual data are merged into a unified structure realized by a custom file system whose data sets can be dynamically added or removed during runtime. This paper presents the requirements, design, and evaluation of a system that support this multi-layered approach to contextualization, which is the first contextualization solution to natively support recontextualization. It is worth noting that the research presented in this paper uses a scientific methodology based on functional prototyping and system performance analysis. We do not use mathematical modelling. Additionally and for clarity's sake, the results of this research aims to facilitate and enhance current cloud provider interoperability beyond the state of the art but we do not claim to solve all problems of VM or IaaS interoperability.

The rest of this article is organized as follows; In Section 'Background and related work' the background of topics related to the field of contextualization and recontextualization are presented. Section 'Contextualization' presents an overview of contextualization including challenges and a summary of our earlier work on the subject. The corresponding information for recontextualization is presented in Section 'Recontextualization'. Section 'Context-aware lifecycle management' shows how the two techniques are used in conjunction to provide adaptable usage throughout the lifecycle of a service. The unified approach is demonstrated and evaluated in Section 'Functional evaluation'. Finally, conclusions and future work are presented in Section 'Conclusions and future work'.

Background and related work

This section of the paper introduces the topics of Virtualization, Service Lifecycle, Configuration Management and Autonomic Computing in relation to our research. Additionally, the differences between configuration management, contextualization and recontextulization, are explicitly defined using the life-cycle of a cloud application in Section 'Service lifecycle'.

Virtualization

Hardware virtualization techniques [7, 8] provide means of dynamically segmenting the physical hardware into smaller virtual compartments. This makes it possible to run several different VMs in parallel on the same physical hardware. Each VM is a self-contained unit, including operating system and (virtual) hardware drivers. The physical resources are subdivided, managed, and made available to the executing VMs through a Hypervisor.

Virtualization is the underlying abstraction technology for most IaaS clouds. VMs create natural barriers between multiple tenants and offer an environment practically indistinguishable from running on dedicated physical hardware. One of the major benefits of virtualization is that it enables some of the fundamental assumptions of physical hardware hosting to be relaxed, which creates new interesting research areas. For example, the hardware configuration of a VM can dynamically change during runtime as the number of CPUs or the amount of RAM memory assigned is adjusted by the hypervisor [9, 10]. These events have no correspondance when running on physical hardware, as the hardware setup is assumed to remain constant from boot-time throughout the uptime of the server. The lifecycle of a VM therefore is more complicated in comparison to a regular server.

Another interesting property of virtualization is that VMs can be relocated (migrated) during runtime from one physical host to another. This opens up a wide range of management possibilities from the infrastructure side, as the mapping between physical and virtual resources can change dynamically in response to demand and availability. From the perspective of the system running inside the VM, migration is not noticeable (apart from a slight pause as control over the VM is finally passed on from the source to the target hypervisor). Migrations may therefore invalidate decisions made during initialization. Such decisions may include algorithmic decisions based on nearest-neighbour or geographical hosting location, where the algorithms assume that the location of the service remains unchanged throughout the uptime of the system. How the layered contextualization approach presented in this work can be used to mitigate the effects of migration is discussed and described in Section 'Functional evaluation'.

Service lifecycle

The lifecycle of a cloud service is related to the lifecycle of the underlying VM templates, rather than to the lifecycle of individual VM instances. The VM templates and a service manifest [11] are created during the *Definition* phase. The service manifest contains all necessary meta data required to tie the individual VM templates together into a unified service. This includes rules, initial values, and lower and upper bounds for automated scaling of VM instances. When the service is submitted for execution to a cloud IP the service enters the *Deployment* phase. During deployment, the initial number of VM instances are started and assigned to physical resources. The service executes during the *Operations* phase, during which

the placement and constitution (size) of the service can change dynamically.

Configuration and contextualization are done during different parts of the service lifecycle. This is illustrated in Table 1. Configuration takes place during the definition phase of the service and includes all customization required to make the VM templates function as a unified service. Settings specific to the platform to which the service is deployed are normally not available during the definition phase unless the service is designed with a single platform in mind. Instead, it is made available to the VMs during the deployment phase through contextualization. Recontextualization offers means to update customized data during the operations phase, enabling migration of VMs in scenarios where such data is fundamental to the operations of the service.

Configuration management

The long lasting deployment of systems such as cloud services create a need for maintenance and management of the application during the operations phase. Such maintenance may include, e.g., applying system security updates across all associated nodes. For small scale systems this can be done manually, but the process is time consuming and error-prone. The rapid and automated elasticity of cloud services further limits the feasibility of manual system management as instances of VMs may be added or removed automatically during any period of the day. *Configuration management* is a well established concept for managing distributed systems during runtime, not just specific to the field of cloud computing. Aiello et al. [12] defines configuration management as:

Definition. *Configuration Management.* A management process that focuses on establishing and maintaining consistent system performance and functional attributes using requirements, design, and operational information throughout a system's lifecycle. [12]

Computing oriented configuration management tools such as CFEngine, Puppet, or Chef are commonly used in large scale hosting on physical platforms. These tools provide a number of benefits including (I) the reproducibility and automation of software configuration across an unlimited number of (virtual) machines, (II) the continuous vigilance over running systems with automated repairs and alert mechanisms, (III) enhanced control over and rationalisation of large scale deployments, and (IV) the ability to build up a knowledge base to document and trace the history of a system as it evolves.

Table 1 Lifecycle phases of a cloud service

Definition phase	Deployment phase	Operations phase
Develop	Select Provider	Monitor/Optimize
Compose	Deploy	Execute
Configure	*Contextualize*	*Recontextualize*

The CFEngine [13] project provides automated configuration management of large networked systems. CFEngine can be deployed to manage many different types of computer system such as servers, desktops and mobile/embedded devices. The project was started in 1993 by Mark Burgess at Oslo University as a way to automate the management of dissimilar Unix workstations. In the work by Burgess the foundations of self-healing systems were developed and as a precursor heavily influenced the ideas of Autonomic Computing developed later by IBM (see Section 'Autonomic computing').

Puppet [14] is a configuration management system originally forked from CFEngine. Puppet provides graph-based and model-driven approaches to configuration management, through a simplified declarative domain specific language that was designed to be human readable. The model driven solution enables the configuration of software components as a class, a collection of related resources where a resource is a unit of configuration. Resources can be compiled into a catalogue that defines resource dependencies using a directed acyclic graph. A catalogue can be applied to a given system to configure it.

Chef [15] rose out of the Ruby-on-Rails community out of dissatisfaction with Puppet's non-deterministic graph-based ordering of resources. In contrast to Puppet, Chef places emphasis on starting up services from newly provisioned clean systems, where the sequence and execution of configuration tasks is fixed and known by the user. This makes Chef particularly well suited to the paradigm of cloud computing where VM instances are short-lived and new instances are spawned from a newly provisioned base image. Chef uses the analogy of cooking and creates "recipes" that are bundles of installation steps or scripts to be executed.

Although existing configuration management tools are well suited to cloud computing they do not resolve all the issues surrounding configuring an application in an dynamic environment. Most notably, these systems operate on the application level for automated management of runtime reconfigurations for a large number of system nodes. Contextualization, as described in Section 'Contextualization', operates on a lower layer of the system stack. Contextualization offers a multi-purpose mechanism for adapting a generically configured VM to a specific and dynamically assigned execution environment. For example, the required settings to connect a booting VM to a VPN can be supplied by the infrastructure at boot time. Configuration management tools and contextualization are complementary techniques for dynamic reconfiguration. Contextualization deals with lower level

IP specifics such as network configuration and platform specific settings, while configuration management can be used to manage updates at the application level.

Autonomic computing

Autonomic computing has been a topic of interest for many years with research starting in 2001 by IBM's autonomic computing initiative [16]. The initiative aimed to create a self-managing computing environment, capable of handling increasingly complex systems. Autonomic computing has been defined as:

Definition. *Autonomic Computing.* "Computing systems that can manage themselves given high-level objectives from administrators." [16]

Autonomic computing involves the creation of systems that run diagnostics checks and compensate for any irregularities that are discovered. Multiple control loops adjust the system to maintain its state within a number of specific bounded criteria. The ever growing complexity of distributed systems provide motivation for the use of autonomous systems as manual control is expensive, prone to errors and time consuming. Autonomic computing reduces the need for system maintenance with aspects such as security or software configuration maintained in an unattended fashion. Administrators, instead of controlling entire systems by hand, define general rule based policies that guide "*self-**" management processes in four functional areas:

- **Self-configuration**: Automatic configuration of components during runtime.
- **Self-healing**: Automatic discovery, and correction of faults during runtime.
- **Self-optimization**: Automatic monitoring and control of resources to ensure the optimal function with respect to the defined requirements.
- **Self-protection**: Proactive identification and protection from arbitrary attacks.

Within the context of cloud computing autonomic computing can be applied to and reasoned about both at the infrastructure level and at the service level. At the infrastructure level, self-management actions primarily within self-optimization can be used with regards to resource assignment and control. An example is re-evaluating the current mapping between virtual and physical hardware to reach higher-level objectives, such as improved application availability. Work in this area has been done by, e.g., Dautov et al. [17], Karakostas [18], Fargo et al. [19], Maurer et al. [20], Wood et al. [21], and Shrivastava et al. [22]. At the service level, autonomic computing approaches can be used to manage the operations of the applications themselves. This includes triggering self-optimization routines to adapt to the current service constitution or self-configuration approaches to adapt to a new execution environment. The work presented in this paper focuses on the service-level. We show how the approach can be used as triggers that allow self-configuration and other *self-** processes to react to changes in the service context. This is done by having mechanisms to make dynamical generated and context-aware data available to processes running inside the VMs of each service.

Athreya et al. [23] investigate the application of self-* approaches to the Internet of Things (IoT), a vision of having billions of devices all conneted to the Internet. With so many connected devices, more autonomic management systems are required. In their work Athreya et al. present a number of challenges for self-configurable IoT systems. These challenges include, e.g., addressing and clarifying the incentives for self-management (including weighing the development costs to the potential benefits), and also discuss the great need of having integrated applications to fully utilize the advantages granted by a self-* enabled infrastructures. These challenges clearly applies to our work as well, and their work provides a solid ground for further discussions.

Self-configuration can conceptually be subdivided into two different subtasks; how to generate the new configuration data and how to allow the software to transition from using the old data to using the new data. For this work we consider the generation of configuration data to be out of scope, since the specifics of the data is closely associated with any given situation.

Finally, Windows Azure [24] provides some basic SaaS level self-configuration functionality through the RoleEnvironment Class [25]. This makes use of Windows environment variables but is limited in scope to Microsoft's proprietary programming languages and the Azure platform. This prevents interoperability amongst other IaaS providers. Our approach strives to enhance this over the current state of the art. Additionally, the RoleEnvironment approach does not consider the self-configuration of IaaS and PaaS level software components, which our solution does.

Contextualization

Our research on contextualization of cloud applications started with an evaluation into the limitations of the current cloud infrastructures. It quickly became apparent that, although virtualization brings several key benefits and is a critical enabler of cloud computing, it also increases the complexity of managing an application deployed in a cloud.

Modern virtualization technologies enable rapid provisioning of VMs and thus allow cloud services to scale up and down on-demand. This elasticity [26, 27] comes with

a new set of challenges for dynamic service configuration. The focus of this research is on horizontal elasticity where scaling is achieved by adding or removing VMs to a service during its operation. The related case of vertical elasticity, i.e. application scaling through VM resizing, is much easier from a contextualization perspective. For horizontal elasticity scenarios, predefined yet flexible contextualization mechanisms enable the manipulation of VM images during the development of a software service. Previously, this required the complex and time consuming configuration of software within base images.

For clarity, our definition of contextualization is as follows:

Definition. *Contextualization.* The autonomous configuration of individual components of an application and supporting software stack during deployment to a specific environment. [3]

In this definition, an environment is considered to be the specific composition of underlying physical and/or virtual hardware in addition to any value added services provisioned by an IP.

Challenges

There are a number of challenges that make the contextualization of cloud services a non-trivial affair. In our previous work we argued that contextualization in cloud computing is a highly pervasive key technological requirement of any cloud service, where elastic resource management is critical to the on-demand scalability of a service [3]. The holistic nature of the services deployed on clouds makes it difficult to provide flexible generic and open tools without limiting the heterogeneity of supported services. We identified three inherent challenges to providing elasticity through contextualization where VMs are added and removed during service operation.

The first challenge to overcome is the complete contextualization of cloud services across all classifications within the cloud ecosystem: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [1, 28]. These classifications refer to:

- SaaS: web-based applications and services hosted on-line, usually reachable from standard interfaces such as a web browser.
- PaaS: systems that offer deployment of applications designed for execution on a specific platform or software environment. For example, the opportunity to upload a Java archive and have it executed on remote resources.
- IaaS: infrastructures that offer the provisioning of remote resources on which the consumer can execute arbitrary software as if the resources were dedicated hardware servers. Virtualization is usually used as a layer between the hardware and the consumer software systems, which makes it possible for the consumer to design the software stack from the operating system upwards.

In this work, we focus on IaaS contextualization of VMs comprising a cloud service. We touch upon using the suggested approach to access platform software services such as network based storage or database services that could be considered stand-alone PaaS services. In IaaS, the challenge pertains to low-level contextualization of virtual resources, as found in IPs, where virtual devices require context to enable VMs to be bootstrapped to existing virtual infrastructures. This approach has been partially explored by RESERVOIR [5].

The second challenge to overcome is contextualization across multiple IaaS domains for reasons of interoperability. Many IaaS providers offer platform services that are not interoperable with those of other providers. Also, as contextualization of VMs is performed as part of service development, the service will be customized to a single provider only. To solve this challenge, we need to extend contextualization to support run-time recontextualization.

The third challenge pertains to a set of functional requirements for real world clouds and their impact on contextualization. Notable among these are end-to-end security through contextualization mechanisms that support a Virtual Private Network (VPN) overlay and software license management systems. Both of these have unique contextualization requirements: contextualization must be secure with no VPN keys stored unless in use; and contextualization that is able to accommodate license protected software and licensing tokens.

In our previous work [6] we discuss the nature of contextualization in the light of the OPTIMIS project [4]. We present details of the architecture and implementation of our tool for the contextualization of platform level services as well as virtual infrastructure. We discuss the implication of contextualization in clouds, the motivation behind our work and suggest a landscape for the evolution of contextualization tools across all classes of clouds within the ecosystem of the future. We contribute to both the image and instance level contextualization of VM's and illustrate the potential effectiveness of our tool through a simple use case.

In addition, as part of our previous work on contextualization, a proof of concept contextualization tool was developed and its performance tested. This prototype was used to confirm the validity of our contextualisation approach on a cloud testbed. This provided evidence on the potential performance of our approach for contextualization with regards to preparing generic VM images of

sizes in the range of 1-5 GB in increments of 1 GB and with varying numbers of concurrent user requests from 10-100, for the purpose of creating ISO CD images containing 1 MB of context data. The results showed adequate scalability and response time over ten iterations of the experiment with minimal variance.

Recontextualization

Recontextualization can be used to adapt a running service to any system changes, including making newly migrated VMs operate properly in the (potentially different) system environment of a new host. A definition of recontextualization is as follows:

Definition. *Recontextualization.* The autonomous updating of configuration for individual components of an application and supporting software stack during runtime for the purpose of adapting to a new environment [6].

In this definition, a new environment is considered to be a change in the underlying physical or virtual hardware, for example when a VM is migrated from one host to another. In addition, changes in infrastructure or platform level services that are in active use by a cloud application would also be considered to be a new environment.

An analogy to recontextualization can be made with printers and printer drivers. Imagine a laptop user moving from one physical location to another (i.e. a new environment) wanting to access a printer at the new location. Instead of manually installing a driver for an associated printer and having to configure the driver before the printing device is usable, the environment itself could detect that the laptop has entered the printer domain, and supply the needed software and settings to the laptop operating system automatically.

Using recontextualization, VM instances can be automatically offered settings and context data to adapt to a new execution environment. Without the support of recontextualization, adapting the instances to their environment may require a time-consuming manual process that limits the flexibility and scalability of the cloud. When migrating VM instances, parts of the virtual infrastructure (e.g., virtual networks, storage areas, databases) may have to be migrated as well. For example, assume that a cloud service is running on the resources of an IP and that it employs a network based storage service offered by that provider. When migrating to another IP, there are three main tasks; migrating the VM itself, relocating the data on the network based storage service to the corresponding service offered by the new IP (to avoid cross-network transfer costs), and ensuring that the VM is able to communicate with the new storage system without manual intervention. Recontextualization can be used to achieve the latter of these three.

The key benefits of our approach to recontextualization are: i) minimal changes to the existing cloud infrastructure, i.e. there is no need to make alterations to the hypervisor and ii) the preservation of security through the selection of a recontextualization mechanism that gathers contextualization data from a secured source. In addition to these benefits, there are a number of challenges that must be overcome before recontextualization can reach widespread adoption. These are due in part to a lack of IP interoperability and the difficulties in creating an approach to recontextualization that can be applied to the wide diversity of applications deployed into cloud computing environments.

Challenges

A motivational factor behind the need for runtime recontextualization stems from VM migration in clouds [29, 30]. Using migration, a VM can be transferred from one physical host to another without explicitly shutting down and subsequently restarting the VM [31]. The entire state of the VM, including e.g., memory pages, are transferred to the new host and the VM can resume its execution from its state prior to migration. As a consequence of this, no contextualization is triggered again when the VM is resumed, as the level of abstraction provided by virtualization is insufficient for platform services. In this research migration from and to identical hypervisor technology is considered. The topic of hypervisor interoperability for migration is out of scope but is discussed in work by Liu et al. [32]. The idea of reconfiguring computing services is by all means not a new subject area. Mohamed et al. [33] present work on reconfiguration of Web Services during runtime. In their approach, remote APIs for modifying values in running services are compiled into Java-based Web Services and are made accessible using remote method invocation.

As presented by Ferrer et al. [4], there are several different cloud scenarios that present a number of challenges to development and deployment of an application. The ability to perform VM migration is a necessity in all these cloud scenarios, e.g., for the purpose of consolidating resources and maintaining levels of QoS. Three of these scenarios are cloud bursting, cloud federations, and brokered cloud deployment. For each scenario, the ability to migrate VMs during runtime is important:

- Bursting - The partial or full migration of an application to a third party IP. This may occur when local resources are near exhaustion.
- Federation - The migration of an applications workload between a group of IPs. This normally occurs when a single IP's resources are insufficient for maintaining the high availability of an application through redundancy.

- Brokering - The automated migration of an application's VMs across IPs. This is normally done for the purpose of maintaining an agreed Quality of Service (QoS) in the case of an end-user utilizing a broker to select an IP given a set of selection criteria.

These scenarios have also been used to guide the defining of requirements for any potential recontextualization mechanism. In the bursting scenario, if an IaaS provider is not obligated to divulge third party providers used for outsourcing of computational resources, an application may end up deployed on to a third party's infrastructure that requires use of their local infrastructure services. A dynamic federation of IaaS providers created during negotiation time that alters during the operation phase requires infrastructure services to be discovered dynamically. The same is applicable in the case of a broker, knowledge of a provider's local infrastructure services is not available during deployment until after the broker has selected a provider.

The following general requirements have been extracted from the scenarios and are considered as imperative:

i. A triggering mechanism for recontextualization on VM migration or other events.
ii. A secure process to gather and recreate contextualization data after migration or other events.
iii. A hypervisor agnostic solution that maintains IaaS provider interoperability.
iv. An approach that is non-pervasive and minimizes modifications at the IaaS level.

The lack of knowledge on the attributes of an IaaS provider's local infrastructure service available during deployment time further motivates this research. An example of such a service that exhibits configuration issues after resource migration is application-level monitoring and is discussed in detail in our previous work [6].

A recontextualization mechanism

With the requirements in mind, we have considered several approaches for mechanisms that would support recontextualization [6]. An approach using dynamic virtual device mounting is found to be the most promising solution for recontextualization due to inherent interoperability and support in all major operating systems. Dynamic virtual device mounting is based on dynamically mounting virtual media containing newly generated content in a running VM via the reuse of existing hypervisor interfaces and procedures. Interoperability is achieved by reusing existing drivers for removable media such as USB disks or CD-ROM drives. Recontextualization can be detected by the guest OS by reacting to events triggered when new USB or CD-ROM media is available. The ability to manage virtual devices is also offered by the libvirt API [34], inferring that there is fundamental support for these operations in most major hypervisors.

The suggested approached has been implemented and the performance has been assessed to confirm the feasibility of the approach in a real cloud infrastructure. For all tests, libvirt version 0.9.9 is used to monitor and manage the VMs. QEMU-KVM version 1.0.50 and Xen version 4.0.0 are used as hypervisors, both running on the same hardware using CentOS 5.5 (final) with kernel version 2.6.32.24. The hosts used in these tests are on the same subnet, have shared storage and are comprised of a quad core Intel Xeon X3430 CPU 2.40 GHz, 4 GB DDR3 @ 1333 MHz, 1 GBit NIC and a 250 GB 7200 RPM WD RE3 HDD.

The results of the evaluation are shown in Fig. 1. The first set of bars illustrate the time to migrate a VM from one host to another with recontextualization running and context data attached. The second set of columns illustrate the same migrations with recontextualization turned off and no virtual devices mounted. The third column illustrates the time spent within the recontextualizer software during the tests from the first column, measured from when the event for migration was received in the recontextualizer until the device had been removed and reattached. The values shown are the averages from ten runs, and all columns have error bars with the (marginal) standard deviations which are all in the 0.03 to 0.07 second range.

Based on the evaluation we conclude that the recontextualization process adds about an 18 % overhead using either hypervisor compared to normal migrations [6]. For KVM, most of the extra time required for recontextualization is spent outside the bounds of our component. The delay is likely associated with processing events and extra overhead imposed by preparing migration with virtual devices attached. In the case of Xen the device management functionality in libvirt proved unreliable and we therefore had to bypass the libvirt API. We instead had to rely on sub-process calls from the recontextualizer to Xen using the *xm* utility. This workaround increases the time needed for recontextualization in the Xen case. The internal logic of the recontextualizer is discussed in full detail as part of Section 'Design and architecture' and within Procedure 1.

There are four major phases associated with the recontextualization process. First, information about the VM corresponding to the event is resolved using libvirt when the migration event is received. In the second phase, any current virtual contextualization device is identified and detached. Third, new contextualization information is prepared and bundled into a virtual device (ISO9660) image. Finally, the new virtual device is attached to the VM. A detailed breakdown of the time spent in different

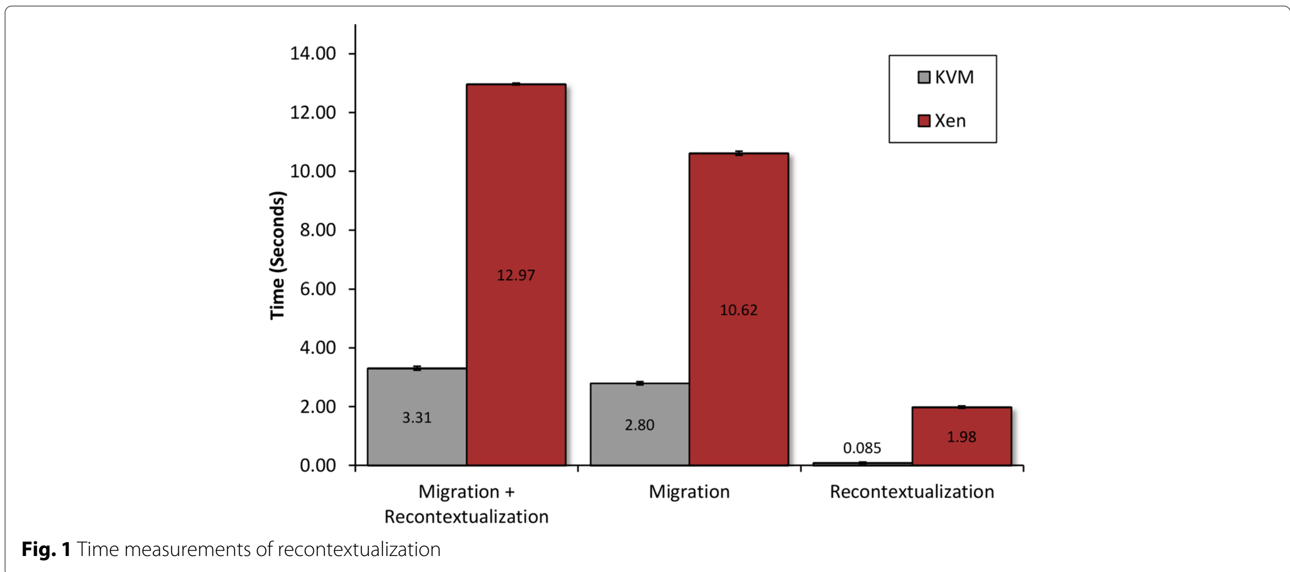

Fig. 1 Time measurements of recontextualization

phases of recontextualization is presented in Fig. 2. The above mentioned workaround for Xen interactions affects the second and fourth phase (detaching and attaching of devices), most likely increasing the time required for processing. In the first and third phases Xen requires significantly longer time than KVM despite running the same internal logic for recontextualization, using the same calls in the libvirt API. This indicates performance flaws either in the link between libvirt and Xen or in the core of Xen itself.

Contextualization versus recontextualization

Although contextualization and recontextualization share many similarities in that they help adapt services to running in the cloud, they do however differ in one fundamental way. Contextualization can only be used during deployment time to configure an application to a specific IP's environment. Recontextualization, however, enables an application to be migrated freely between dissimilar environments during runtime through the use of an additional level of abstraction. One could consider recontextualization to be an extension of contextualization, and base contextualization of services during deployment (or development) is also needed to enable runtime recontextualization.

Context-aware lifecycle management

As discussed in Section 'Service lifecycle', a cloud service is prepared with configuration data built in during the definition phase. Complementing contextualization data,

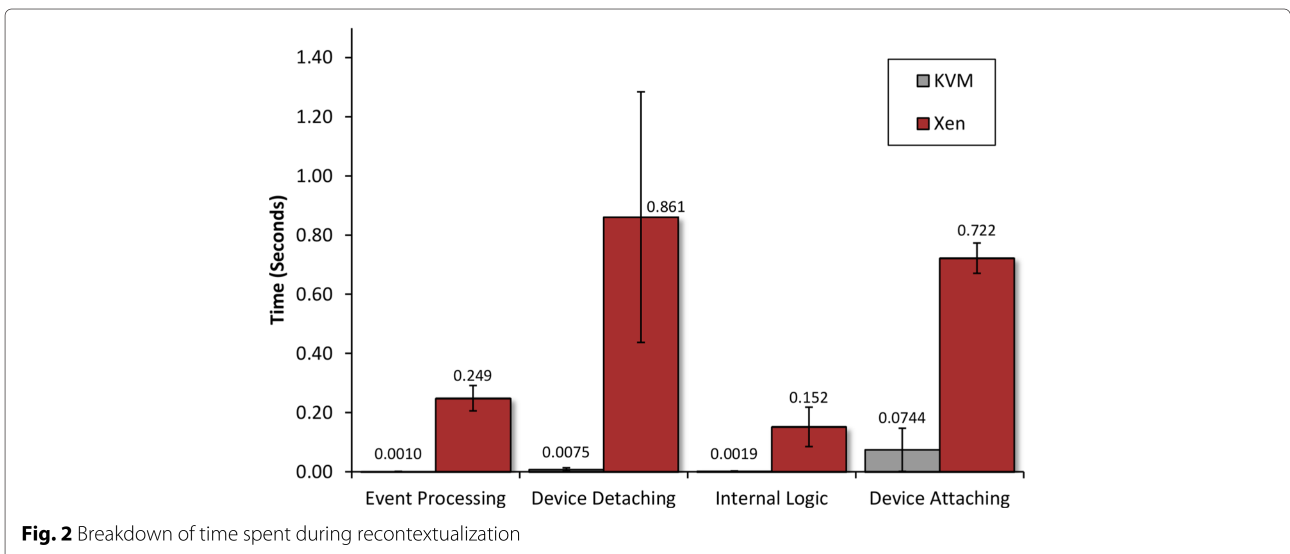

Fig. 2 Breakdown of time spent during recontextualization

either generated or read from a repository, is injected by the infrastructure during deployment. Parts of the contextualization data may also be recontextualized during runtime. In this work, we strive to create a model and mechanism capable of managing any kind of configuration and contextualization data in a fail-safe manner that appears seamless to the software running inside as part of the service.

To make the approach generally applicable we consider several *classes* of data used for configuration/contextualization, where the classification is related to the entity from where it originates. For example:

- Web server configuration data is normally provided by the service developer, so this data would belong to the *service* class.
- WAN network data is by necessity provided by the infrastructure, and would belong to the *infrastructure* class.

Other classes of data may include, e.g., *legislative* data that depends on the current legislation in which the service is hosted, or data injected by a *broker* during an advanced service deployment.

Defining a complete ontology of service classes and their relations is out of scope at this point, but is an important part of future work. The model presented in this paper supports an arbitrary number of service classes, and relies on an internal prioritization of the classes to determine which class should take precedence if the same data is available from multiple sources. The feasibility of the approach is illustrated using a use-case featuring a distributed file system. The use-case employs data from *service* and *infrastructure* classes, and infrastructure-class data is configured to have precedence in case of overlapping datasets.

Practical applications

A generic approach with support for several classes of data enables a service to be designed for generic cloud deployment with a set of default settings. These default settings may be complemented and partly replaced with settings provided by, e.g., the infrastructure to which the service is deployed. Through future formalization of the type of data offered by different actors, dynamic services with inherit support for self-* operations, e.g. self-configuration, can be constructed and deployed to the cloud.

For example, in previous work on recontextualization [6], a monitoring use case was used to illustrate how the infrastructure can update a setting used for internal processes in the VM during runtime. In this work, we illustrate how the procedure can be used at the service level to reconfigure the internal connections of a distributed file system, adapting its operation to a new geographic layout resulting from migration of VMs (see Section 'Functional evaluation').

Further use cases related to other aspects of self-* management are part of future work. For self-protection, the ability to change the security level of the software executing inside the VM depending on, e.g., the geographical location of the current IP is an interesting area of research. This could include adjusting the level of encryption of data streams passing in and out of the service component to manage the balance between security and performance. If the hosted service contains data only allowed to be hosted within certain legislations, recontextualization could be used to ensure that the operations of the service component is temporary suspended if the VM is (erroneously) migrated to a disallowed environment.

Design and architecture

As described in Section 'A recontextualization mechanism', earlier work explored several approaches to passing data between the hypervisor and the processes running inside the VM. The work in this paper retains the use of virtual devices as a transport mechanism, due to its native support in major hypervisors and operating systems, and does not depend on network access. This is a key feature, since network access is one of the primary things that contextualized settings can be used to enable. As previously mentioned, the core idea of our suggested approach is to dynamically generate customized ISO images which are made available to the running VM instance. As the ISO images can be freely changed (and re-mounted) during run-time, the context data can be updated at any point in the VM lifecycle.

Our proposed solution contains two distinct parts; tools and services for contextualization running as part of the cloud controlling software on the infrastructure side, and a minimalistic process running inside each VM. The code running inside the VM runs a small virtual file system (described in Section 'Flexible file system') that encapsulates the complexity of dealing with potentially overlapping and conflicting sets of context data of different classes. The software required for the file system is made available during contextualization, and has minimal OS dependencies. An overview of the components designed to run on the infrastructure side is shown in Fig. 3, and a description of the roles of each component follows.

- **Contextualizer** offers API-level integration with the surrounding cloud control system to integrate the contextualization process in the workflow executed prior to deployment. The result of the contextualization call is an ISO image file customized to suit the running hypervisor with the necessary context data. Once completed the file is uploaded to a

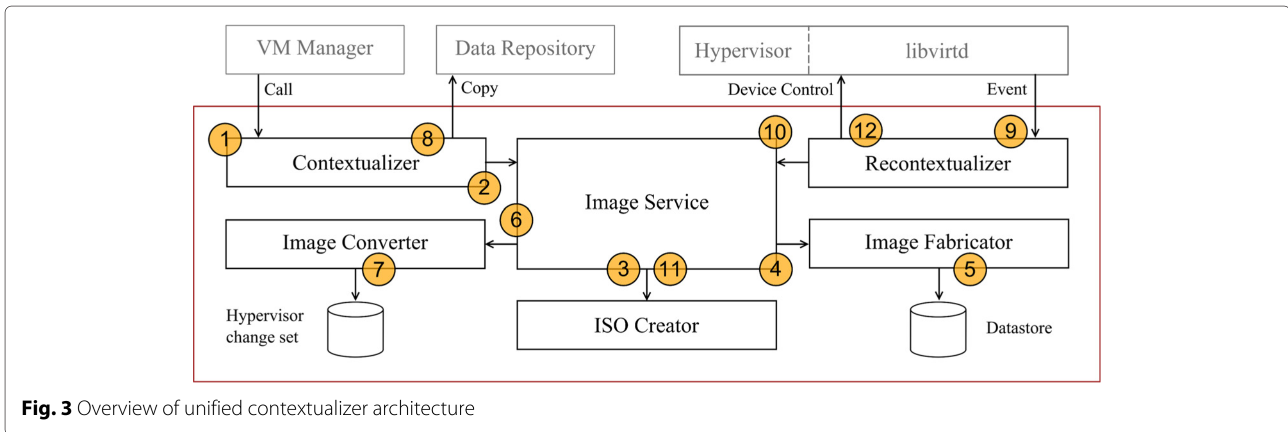

Fig. 3 Overview of unified contextualizer architecture

data repository from where it can be fetched later during the service deployment procedure.

- **Image service** contains the code used for image generation that is shared between contextualization and recontextualization. This includes the coordination of calls to image creation, fabrication, and conversion.
- **ISO image creator** manipulates a set of template ISO images. In addition, this sub-component is responsible for the creation of ISO images and the embedding of context data. The ISO Image Creator has access to a single set of context data processing scripts that are agnostic of a VM's operating system, embedded in the ISO image and responsible for reading context data from the ISO image at run time. An internal interface provides access to contextualization scripts embedded in the VM image.
- **Image fabricator** manipulates VM images, passed to it by the VM Contextualizer, using underlying system tools. This sub-component installs context data originating from a manifest. A set of prefabricated operating system specific contextualization scripts, from a data store, are embedded for each operating system with foreseen use at the IP level. The main function of these scripts are to set the context of operating system level components such as configuring the network for DHCP access, or connecting to a service-level VPN.
- **Image converter** transforms images from one format to another for the purpose of supporting interoperability between IPs using different hypervisor technology. Depending on the hypervisor and operating system, conversion can require the changes made by the Image Fabricator to be reverted and reapplied. The Image Converter supports Xen, KVM, VMware, and VirtualBox hypervisors and has built-in support for a range of different image formats.
- **Recontextualizer** is a stand alone component that monitors events related to VM domains during runtime. The recontextualizer is responsible for triggering the creation and association of new infrastructure class context data when applicable domains are migrated to the infrastructure. The recontextualizer is integrated with libvirt, and relies on libvirt's unified approach to domain and device management to support runtime reactive virtual device management across a multitude of different hypervisors.

Figure 3 also includes the order in which components are called. From the figure it can be seen that the Contextualizer component is invoked by the VM Manger during application deployment (step 1) to create ISO images (steps 2, 3), create VM images (steps 4, 5) and/or manipulate VM images (steps 6, 7). After images have been created and/or manipulated, they are stored in a local data repository (step 8) for deployment by the VM Manager. During operation, if an event from the underlying hypervisor indicates that a VM has been stopped, started or migrated (step 9), alterations to the existing ISO images are made (steps 10, 11) and reinserted into the VM's virtual device (step 12).

The flow of logic through the contextualizer component is as follows: context data is gathered from a range of sources, it is processed into a usable fashion for use within a VM, stored within an ISO image and finally associated with the virtual hardware of a domain for use during booting. The recontextualization mechanism (i.e. the internal logic), outlined in Procedure 1, registers interest with libvirt in a VM's lifecycle by means of a callback function before beginning its operation. When an event is registered in the callback function it is classified and appropriate action taken. The exact events used to represent a VM stopping or starting as a result of migration may differ depending on the underlying hypervisor. In the example shown we use vmStop and vmStart events to also cover migration, as these are the only events available with the XEN hypervisor and libvirt.

If a stop event is detected the previous device containing context data is detached and the device change is committed via libvirt, which proceeds to propagate the virtual hardware change at the hypervisor level. In the case of a start event being detected, infrastructure class context data is generated then packaged into an ISO image. The recontextualization mechanism interacts with libvirt to create a new virtual device with the contents of the dynamically generated ISO image. A failed VM migration will not affect recontextualization as the corresponding events are only sent after a migration has been successful.

Procedure 1 Recontextualization Mechanism Internal Logic

```
connection = libvirt.open('URL')
libvirt.EventRegister(Callback())
while true do
    if connection.event then
        function CALLBACK(event, domain)
            if event.vmStop then
                device = getDeviceDef(domain)
                detachDevice(device)
                updateDevices()
            end if
            if event.vmStart then
                contextData=createContextData(domain)
                makeIso(contextData)
                attachDevice(device)
                updateDevices()
            end if
        end function
    end if
    sleep()
end while
```

Although not shown in the illustration, the infrastructure can trigger recontextualization in response to any event, not just migration. This enables the infrastructure to update the contents of the virtual device by generating new context data and detaching and re-attaching the virtual device. Using this construct, infrastructure specific settings can be changed during runtime at the convenience of the provider.

Flexible file system

Flexible File System (FFS) is a lean, read-only file system implementation that allows separate directory trees to be mounted into a single location. The resulting file system looks and acts like a single unified directory structure while retaining all file contents and properties in the underlying directory trees.

FFS can be seen as a layered stack of file-sets, each set comprised of different files and directories. When a user attempts to read a file in the unified structure, FFS will attempt to fetch a file from the topmost (highest prioritized) layer. If the file cannot be found at this layer, the call will be delegated to lower layer(s) until it is found, or return an error if the file cannot be found in any layer. This process is illustrated in Fig. 4. In this example, there are two set of files (Layer 1 and Layer 2), and files in Layer 2 have precedence. FFS will ensure that the two file sets are displayed and presented to the user in a coherent way, and the user will not be able to distinguish which files are from which file-set.

In essence, the FFS file system solves many of the complications of dealing with file-sets from different sources, as files from one set can *overshadow* a file from another set, without making permanent changes to either of the files. In the example shown in Fig. 4, the file 'F' from Layer 1 will not be visible to the user as it has the same name and path as a file from another file-set (Layer 2), which has precedence. If the Layer 2 file-set is removed (as can happen due to recontextualization, see Section 'Recontextualization using FFS'), file 'F' from Layer 1 would be visible to the user, and the running applications are able to reconfigure to use the available settings.

FFS is implemented using Filesystem in Userspace (FUSE) [35], a kernel module that allows custom filesystem implementations to run in userspace without the need for any changes in kernel space. The kernel module is available for most Unix-like system, including OpenSolaris and OS X, and was merged into the main Linux kernel tree in 2011. Work on FUSE compatible technologies for Windows is ongoing [36].

Creating a read-only file system requires four method calls to be implemented:

- **readdir(path)** List all items in a path. In this case, FFS returns the union of listings from all layers.
- **getattr(path)** List attributes for the file or folder indicated by the path. FFS will return the actual file attributes, with the attributes at the highest layer taking precedence if the path exists in more than one layer.
- **open(path, flags)** Similarly to getattr(), the access flags for the file with the highest precedence will apply.
- **read(path, size, offset)** Reads *size* bytes starting from *offset* in the file indicated by *path*, again being resolved to the file located at the layer with the highest precedence if more than one layer exists.

The required logic for managing several classes of data is implemented within these method calls, creating a virtual file structure based on the set of files currently available.

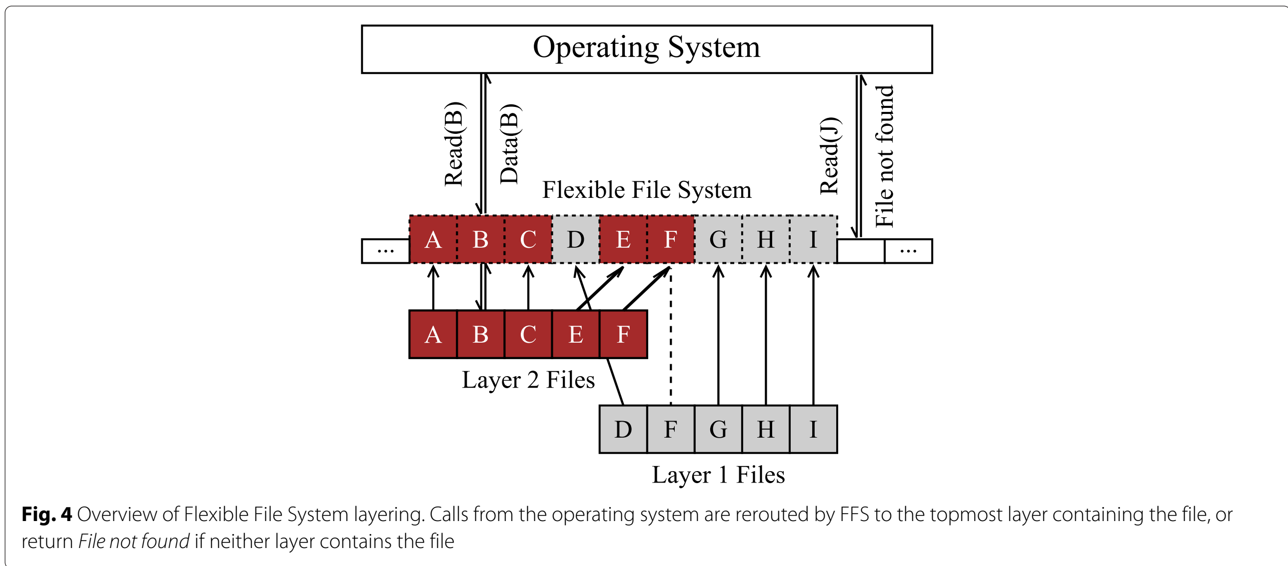

Fig. 4 Overview of Flexible File System layering. Calls from the operating system are rerouted by FFS to the topmost layer containing the file, or return *File not found* if neither layer contains the file

Recontextualization using FFS

The file system is initialized and started during the deployment phase of the cloud service (see Table 1), with the required files being part of the context data class originating from the service manifest. If multi-layered contextualization is not required for a particular service, the associated files can simply be omitted from the set of files specified in the manifest. When other layers of context data (such as infrastructure specific settings) are available they will be made available as a separate layer in the FFS stack. The order of the layers (configurable as part of the initial context files) determine the priority of files in case of overlapping datasets.

Since layers can be added and removed during runtime, recontextualization is natively supported by this approach. In the following section, we show results of testing using this setup to replace infrastructure specific data after migration to a new environment.

Functional evaluation

Earlier in the paper we discussed how virtualization invalidates some basic assumptions of server hosting, such as the physical hardware constitution of the executing environment (vertical elasticity), or the physical location of the system (migration). A similar behavior can also be observed at the application level inside the VMs. For example, distributed file systems such as Hadoop FS [37] perform optimization during initialization to ensure that the system exchanges data with a nearby server node to maximize throughput and reduce latency. This approach assumes that the geographical location where the system is being hosted remains unchanged throughout the uptime of the service. This is normally a reasonable assumption when using physical hosting but one easily invalidated in virtualized hosting as a result of VM migration. As a consequence, the performance of applications such as Distributed File Systems (DFS) may be degraded post-migration until the system is rebooted and a new initialization phase optimization can take place. This scenario is illustrated in Fig. 5 where a client receives a considerable performance penalty for accessing a DFS's cache at a geographically remote location after migration has occurred. In this scenario, the contextualization and recontextualization data is the IP address that points to the local cache of the DFS and is modified in the configuration file of the DFS client running within the VM.

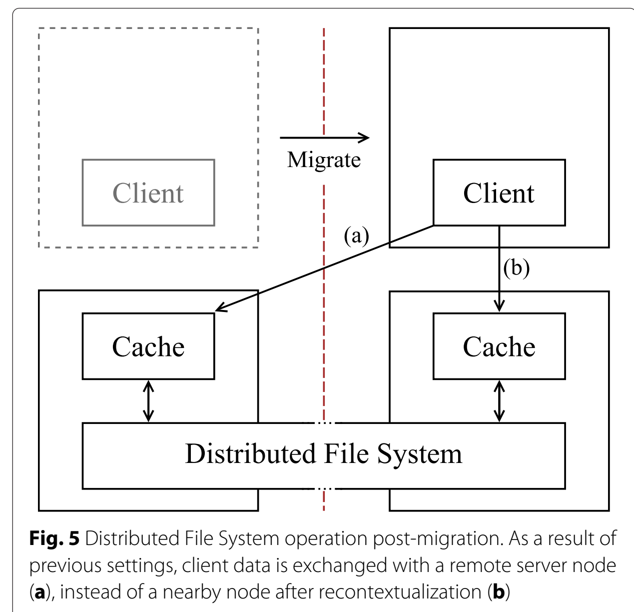

Fig. 5 Distributed File System operation post-migration. As a result of previous settings, client data is exchanged with a remote server node (**a**), instead of a nearby node after recontextualization (**b**)

For example, before recontextualization this IP could be 192.168.1.1 and after 192.168.2.1.

As the use of virtualized hosting becomes more widespread, application-level systems need to be updated to better respond to previously unexpected changes in the hosting environment. For example, the degraded performance could in theory be detected by internal self-optimization techniques during runtime. However, from the perspective of the individual VM a performance loss as a result of migration would be hard to distinguish from a loss of performance due to heavy load on the network. This is one factor that makes application-level self-optimization approaches more challenging to design.

By supporting recontextualization, the proposed contextualization framework offers unambiguous means of triggering internal system events as a result of external events such as VM migration or a change in the surrounding hosting infrastructure. In the distributed file system case, recontextualization can be used to trigger a new optimization phase post-migration, either by calling designated functions in the file system client (if supported), or simply by restarting the client process.

The above scenario has been used to illustrate the feasibility of the suggested approach through a series of tests. The tests were performed on host machines with quad core Intel i7-920 CPU @ 2.66GHz, 24GB DDR3 @ 1333MHz, 1GBit NIC and two 1500GB 7200RPM HDDs in RAID1. The following software inclusive of version numbers were used: Debian 6 (Kernel 2.6.32-5-xen-amd64), XEN 4.0.1, Libvirt 1.0.5 and GlusterFS 3.3.2. Three paravirtualized VMs were created, two acting as Gluster servers serving a "brick" each as a replicated volume and a third as a Gluster Client where the tests were performed on the exported file system stored within the volume. In these tests, traffic shaping techniques (managed via the Linux Kernel Traffic Control command `tc`) have been used to simulate network delays and latencies between different hosts. The tests show how network delay and latencies affect the I/O performance of the Gluster FS distributed file system. The remote host in the test scenario is limited to 25Mbit of bandwidth, with an average 10ms latency. The latency has a 2 % variance following a normal distribution and 25 % correlation with the previous traffic to simulate network fluctuations. The local host is connected with a 500Mbit connection and a measured average latency of 1.02ms (0.5 % variance). These settings correspond to typical network conditions of WAN and LAN networks, respectively. The I/O testing is performed by running a series of tests using the Bonnie++ I/O testing framework [38]. Results from the tests are shown in Figs. 6 and 7.

It is important to note that the performance results are strongly correlated with the traffic shaping settings, and the numerical results shown in the figures should not be emphasized. The purpose of these tests are to illustrate the functional capabilities of the suggested approach, and demonstrate that contextualization and recontextualization can be used to adapt a running VM to changing execution environments.

As illustrated using these tests, considerable performance gains can be achieved by optimizing the setup of application-level systems such as distributed file systems during runtime. This is achieved by using recontextualization to update the application with new settings containing details of what server node to use for its interactions. This demonstrates how the suggested approach can be used as a corner stone in application-level self-* operations.

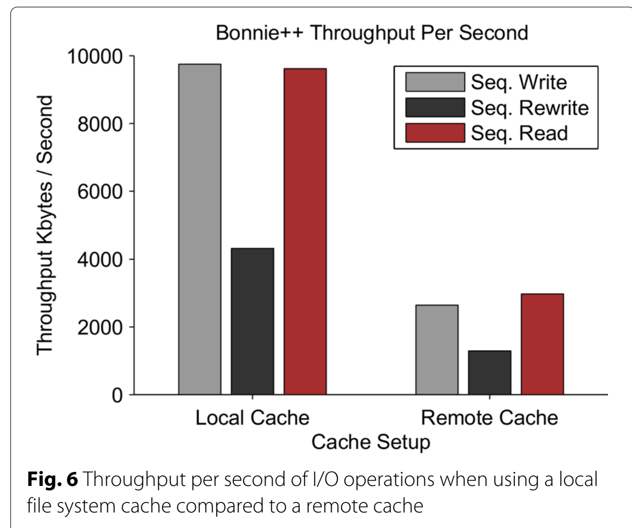

Fig. 6 Throughput per second of I/O operations when using a local file system cache compared to a remote cache

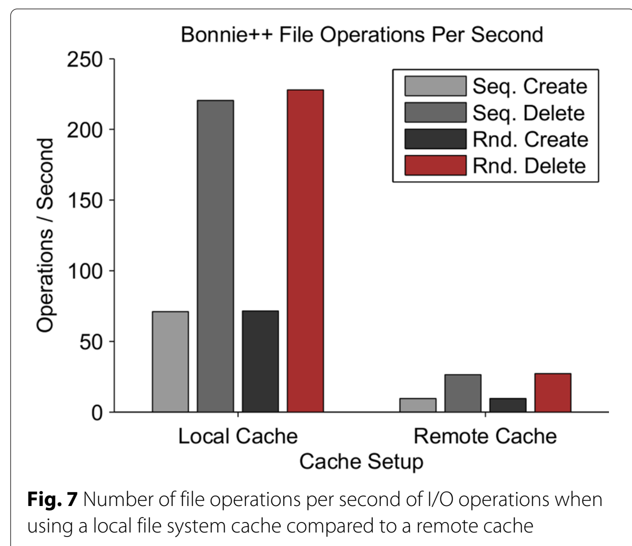

Fig. 7 Number of file operations per second of I/O operations when using a local file system cache compared to a remote cache

Conclusions and future work

In this work we have suggested a contextualization approach that supports several classes of data. The data can originate from different sources, e.g. data related to the service the VM instance is part of or data related to the platform on which the instance is running. Data of different classes are layered on top of each other using a lean virtual file system implementation, making it possible to add or remove data layers during runtime. For example, the platform specific class of data can be replaced by the infrastructure as a response to, e.g., a migration of the VM instance to a new executing environment. This way, the suggested contextualization solution is the first known approach to natively support recontextualization. Recontextualization offer means to reconfigure the VM instance as a result of external events or changes in the execution environment.

Recontextualization can be used as a trigger for self-* operations that cannot misinterpret events. This is in contrast to a VM trying to analyse when an event had occurred via its own internal procedures, for example based on when its external IP address changes. A distributed file system use case demonstrated the feasibility of our approach. In the use case recontextualization is used to reconfigure the VM to recover from performance degradation experienced as a result of VM migration. Although the degradation is partly synthetic and correlated to the network settings used, we demonstrate that performance gains can be achieved by runtime optimization triggered by recontextualization.

The limitations of our research are constrained to the self-configuration property of autonomic computing. Future work includes evaluating further use cases related to other self-* aspects such as self-protection. This includes the ability to adjust the security level and functionality of the service component depending on the current geographical location.

Additionally, we have mostly limited our research to IaaS level applications of contextualization and recontextualization. We touch upon using these technologies also for accessing platform specific functionality, such as network based storage, offered as a service by IPs. There is great potential for employing these techniques within PaaS systems, for example enabling dynamic code replacement where software libraries (JAR files, DLLs, etc.) can be made available by the infrastructure for execution within the service. This would allow for arbitrary recontextualization also at the application level.

Future technical work includes formalizing and standardizing the use of different classes of context data and their semantics. A common approach to contextualization is key to enable wide-spread and compatible support across cloud boundaries. A unified representation and interpretation of contextualization meta-data in manifests and service definitions also needs to be further developed.

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
DA and DE conducted the design, analysis and experiments of contextualization and recontextualization system within the paper. All authors drafted, read and approved the final manuscript.

Acknowledgements
The research that led to these results is partially supported by the European Commission's Seventh Framework Programme (FP7/2001-2013) under grant agreement numbers 257115 (OPTIMIS) and 610874 (ASCETiC). The authors would also like to thank Tomas Forsman for technical assistance and expertise.

Author details
[1]School of Computing, Faculty of Engineering, University of Leeds, LS2 9JT Leeds, UK. [2]Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden.

Received: 31 March 2015 Accepted: 30 June 2015
Published online: 15 July 2015

References
1. Mell P, Grance T (2011) The NIST definition of cloud computing. NIST Spec Publ 800:145
2. Keahey K, Freeman T (2008) Contextualization: Providing One-Click Virtual Clusters. In: Proceedings of the 4th IEEE International Conference on EScience (ESCIENCE '08), Washington, DC, USA. pp 301–308
3. Armstrong D, Djemame K, Nair S, Tordsson J, Ziegler W (2011) Towards a Contextualization Solution for Cloud Platform Services. In: Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on. IEEE. pp 328–331
4. Ferrer A. J, Hernández F, Tordsson J, Elmroth E, Ali-Eldin A, Zsigri C, Sirvent R, Guitart J, Badia R. M, Djemame K, Ziegler W, Dimitrakos T, Nair S. K, Kousiouris G, Konstanteli K, Varvarigou T, Hudzia B, Kipp A, Wesner S, Corrales M, Forgó N, Sharif T, Sheridan C (2011) OPTIMIS: a Holistic Approach to Cloud Service Provisioning. Futur Gener Comput Syst 28:66–77
5. Rochwerger B, Caceres J, Montero RS, Breitgand D, Elmroth E, Galis A, Levy E, Llorente IM, Nagin K, Wolfstha Y (2009) The Reservoir model and architecture for open federated cloud computing. IBM J Res Dev 53(4)
6. Armstrong D, Espling D, Tordsson J, Djemame K, Elmroth E (2013) Runtime Virtual Machine Recontextualization for Clouds. In: Caragiannis I, Alexander M, Badia R, Cannataro M, Costan A, Danelutto M, Desprez F, Krammer B, Sahuquillo J, Scott S, Weidendorfer J (eds). Euro-Par 2012: Parallel Processing Workshops. Lecture Notes in Computer Science. Springer Berlin Heidelberg, Heidelberger Square 3, 14197 Berlin, Germany Vol. 7640. pp 567–76
7. Barham P, Dragovic B, Fraser K, Hand S, Harris T, Ho A, Neugebauer R, Pratt I, Warfield A (2003) Xen and the art of virtualization. SIGOPS Oper Syst Rev 37(5):164–177
8. Popek GJ, Goldberg RP (1974) Formal requirements for virtualizable third generation architectures. Commun ACM 17(7):412–421
9. Dawoud W, Takouna I, Meinel C (2012) Elastic virtual machine for fine-grained cloud resource provisioning. In: Global Trends in Computing and Communication Systems. Springer Berlin Heidelberg, Heidelberger Square 3, 14197 Berlin, Germany. pp 11–25
10. Kalyvianaki E, Charalambous T, Hand S (2009) Self-adaptive and self-configured cpu resource provisioning for virtualized servers using kalman filters. In: Proceedings of the 6th International Conference on Autonomic Computing. ACM. pp 117–126
11. Galán F, Sampaio A, Rodero-Merino L, Loy I, Gil V, Vaquero LM (2009) Service specification in cloud environments based on extensions to open standards. In: Proceedings of the Fourth International ICST Conference on Communication System Software and Middleware. ACM. p 19

12. Aiello R, Sachs L (2010) Configuration Management Best Practices: Practical Methods that Work In the Real World. 1st edn. Addison-Wesley Professional, Boston, USA
13. CFEngine 3 - Configuration Management Software for Agile System Administrators. http://aws.amazon.com. Online (Accessed: March 2015)
14. Puppet - IT Automation for System Administrators. http://puppetlabs.com/. Online (Accessed: March 2015)
15. Chef - A Systems Integration Framework. https://www.chef.io/chef/. Online (Accessed: March 2015)
16. Kephart JO, Chess DM (2003) The Vision Of Autonomic Computing. Computer 36(1):41–50
17. Dautov R, Paraskakis I, Stannett M (2014) Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms. J Cloud Comput 3(1):13
18. Karakostas B (2014) Towards Autonomic Cloud Configuration and Deployment Environments. In: International Conference on Cloud and Autonomic Computing (ICCAC). pp 93–96
19. Fargo F, Tunc C, Al-Nashif Y, Akoglu A, Hariri S (2014) Autonomic Workload and Resources Management of Cloud Computing Services. In: International Conference on Cloud and Autonomic Computing (ICCAC). pp 101–110
20. Maurer M, Brandic I, Sakellariou R (2012) Adaptive resource configuration for Cloud infrastructure management. Futur Gener Comput Syst
21. Wood T, Shenoy P, Venkataramani A, Yousif M (2009) Sandpiper: Black-box and gray-box resource management for virtual machines. Comput Netw 53(17):2923–2938
22. Shrivastava V, Zerfos P, Lee K-w, Jamjoom H, Liu Y-H, Banerjee S (2011) Application-aware virtual machine migration in data centers. In: INFOCOM, 2011 Proceedings IEEE. IEEE. pp 66–70
23. Athreya AP, DeBruhl B, Tague P (2013) Designing for self-configuration and self-adaptation in the Internet of Things. In: Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on. IEEE. pp 585–592
24. Microsoft: Windows Azure. http://www.windowsazure.com. Online (Accessed: March 2015)
25. Microsoft: Windows Azure's Role Environment Class. http://msdn.microsoft.com/en-us/library/microsoft.windowsazure.serviceruntime.roleenvironment.aspx. Online (Accessed: March 2015)
26. Gong Z, Gu X, Wilkes J (2010) PRESS: PRedictive Elastic ReSource Scaling for cloud systems. In: Proceedings of the 6th International Conference on Network and Service Management (CNSM '10), Piscataway, NJ, USA. pp 9–16
27. Krishnan S, Counio JC (2010) Pepper: An Elastic Web Server Farm for Cloud Based on Hadoop. In: Proceedings of the 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom '10), Los Alamitos, CA, USA. pp 741–747
28. Vaquero LM, Rodero-Merino L, Caceres J, Lindner M (2009) A break in the clouds: towards a cloud definition. SIGCOMM Comput Commun Rev 39(1):50–55
29. Bradford R, Kotsovinos E, Feldmann A, Schiöberg H (2007) Live wide-area migration of virtual machines including local persistent state. In: Proceedings of the 3rd International Conference on Virtual Execution Environments. ACM. pp 169–179
30. Wood T, Ramakrishnan KK, Shenoy P, van der Merwe J (2011) CloudNet: dynamic pooling of cloud resources by live WAN migration of virtual machines. In: Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments. ACM. pp 121–132
31. Clark C, Fraser K, Hand S, Hansen JG, Jul E, Limpach C, Pratt I, Warfield A (2005) Live migration of virtual machines. In: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2. USENIX Association. pp 273–286
32. Liu P, Yang Z, Song X, Zhou Y, Chen H, Zang B (2008) Heterogeneous live migration of virtual machines. In: International Workshop on Virtualization Technology (IWVT'08)
33. Mohamed M, Belaid D, Tata S (2013) Adding Monitoring and Reconfiguration Facilities for Service-Based Applications in the Cloud. In: Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on. pp 756–763
34. Libvirt: The virtualization API. http://libvirt.org/. Online (Accessed: March 2015)
35. Szeredi M Filesystem in userspace. http://fuse.sourceforge.net/. Online (Accessed: March 2015)
36. Dokan. http://dokan-dev.net/en. Online (Accessed: March 2015)
37. Borthakur D (2007) The hadoop distributed file system: Architecture and design. Hadoop Project Website 11:21
38. Coker R Bonnie++ file-system benchmark. http://www.coker.com.au/bonnie++. Online (Accessed: March 2015)

Submit your manuscript to a SpringerOpen journal and benefit from:

- ► Convenient online submission
- ► Rigorous peer review
- ► Immediate publication on acceptance
- ► Open access: articles freely available online
- ► High visibility within the field
- ► Retaining the copyright to your article

Submit your next manuscript at ► springeropen.com