An Approach for Virtual Appliance Distribution for Service Deployment

Gabor Kecskemeti[a,b,*], Gabor Terstyanszky[b], Peter Kacsuk[a,b], Zsolt Nemeth[a]

[a] *Laboratory of Parallel and Distributed Systems, MTA-SZTAKI*
13-17 Kende u., Budapest 1111, Hungary
[b] *Centre of Parallel Computing, University of Westminster*
115 New Cavendish Street, London W1W 6UW, United Kingdom

Abstract

Fulfilling a service request in highly dynamic service environments may require deploying a service. Therefore, the effectiveness of service deployment systems affects initial service response times. On Infrastructure as a Service (IaaS) cloud systems deployable services are encapsulated in virtual appliances. Services are deployed by instantiating virtual machines with their virtual appliances. The virtual machine instantiation process is highly dependent on the size and availability of the virtual appliance that is maintained by service developers. This article proposes an automated virtual appliance creation service that aids the service developers to create efficiently deployable virtual appliances – in former systems this task was carried out manually by the developer. We present an algorithm that decomposes these appliances in order to replicate the common virtual appliance parts in IaaS systems. These parts are used to reduce the deployment time of the service by rebuilding the virtual appliance of the service on the deployment target site. With the prototype implementation of the proposed algorithms we demonstrate the decomposition and appliance rebuilding algorithms on a complex web service.

Keywords: cloud computing, grid computing, virtualization, web service, deployment

1. Introduction

Services abstract system functionalities from the applied technology and they enable the access of these functionalities through predefined interfaces. In service based systems [11] these interfaces, defined by service descriptions, allow the users to access the services transparently without knowing the exact details of the used service instance. The vision of these service based systems incorporates highly dynamic environments [10] where service instances are deployed, utilized, and decommissioned on demand. Service deployment [31] prepares the service code on the infrastructure of the service provider for later usage. Dynamism in the service based systems require the automation of the service deployment process.

Cloud computing [2, 6] promises simple and cost effective outsourcing of applications (Software as a Service – SaaS), platforms (Platform as a Service – PaaS) and hardware resources (Infrastructure as a Service – IaaS). PaaS and IaaS systems can be used to extend Service Based Systems by deploying services on their resources. On PaaS cloud systems the services are developed specially for the given platform. In contrast, IaaS systems use hardware virtualization to support a wider variety of applications. These systems require the encapsulation of the services in virtual appliances. Therefore, services are deployed by instantiating a virtual machine in the IaaS system.

Virtual appliances combine services and their support environment in a form executable by virtual machines. In current IaaS systems the users either use virtual appliances already published in a virtual appliance marketplace [21, 22] or they have to *create the required virtual appliance* on their own. However, these newly created virtual appliances are not specifically designed with their frequent deployments in mind and this seemingly minor issue can seriously hinder exploiting the dynamic features of the system. For example some IaaS systems charge for the network usage during deployment, therefore improperly created virtual appliances entail hidden deployment costs for their users. Hence, one of the main aspects of the work is providing the designers an automated mechanism and support for creating virtual appliances.

In highly dynamic service environments a service request might impose a service deployment before the request can be served. As a result, the inclusion of the deployment increases the apparent execution time of the request. Current deployment systems (e.g. [28]) try to *reduce the effects of pre-execution deployment* by replicating the virtual appliances within the IaaS system. However, IaaS systems regularly charge for the extra storage needs of the owner of the replicated virtual appliance. The other main aspect of our work is a means to reduce turnaround time and keep deployments transparent to the users.

In this article we propose the automated virtual appliance creation service (AVS) that supports the service de-

*Corresponding author

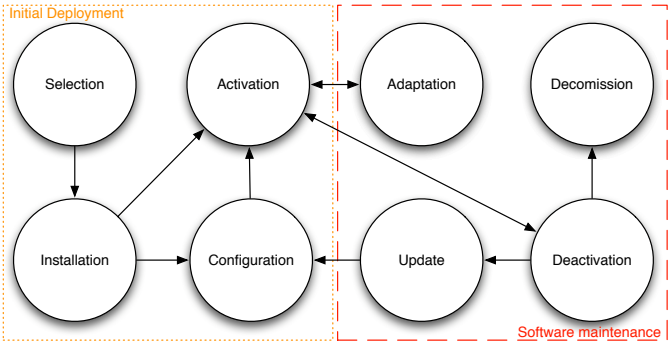

Figure 1: Relations of deployment tasks

velopers in the *virtual appliance creation* and publication process. The AVS creates the service's virtual appliance based on an already operable service installation on the developer's system. After the initial virtual appliance is created the developer can request the service to prepare and publish the appliance for execution in various IaaS systems.

We also present active repositories as an approach for *reducing the effects of pre-execution deployment*. These repositories automatically decompose the stored virtual appliances to smaller parts, thus they allow the partial replication of the appliances. Our approach only replicates virtual appliance parts that are common in the stored virtual appliances. As a consequence to decomposition, our solution rebuilds the original virtual appliances on the target site of the deployment.

This article is organized as follows: Section 2 gives an overall look on the state of the art of the service deployment. Then Section 3 discusses the Automatic Service Deployment architecture that incorporates the automated virtual appliance creation and active repository functionalities. Later sections are focusing only on publication and distribution of virtual appliances however, Section 3 offers a general overview and an outlook of the architecture. Next, Section 4 proposes and details the AVS, active repository and virtual appliance rebuilding algorithms and their relations. The implementation and experiments on the proposed solutions are discussed in Section 5 that is followed by the conclusion in Section 6.

2. Service Deployment Overview

Service deployment is the process of making a service instance available for the users. We define deployment as a complex process that is composed of following deployment tasks (see Figure 1): selection, installation, configuration, activation, adaptation, deactivation, update, and decommission. In the next paragraphs we detail the purpose of these tasks.

First, the *Selection* task chooses the appropriate (hardware and software) target system to deploy the service on. Then the *Installation* task manages the addition of the new software components to the target system. These components include the service itself and its dependencies that are described in [5, 26]. The installed components are adjusted to fit the target system's specific needs with the *Configuration* task [18, 31]. During the *Activation* task the deployment system makes the service available for the users, and usually executes it.

The remaining deployment tasks are related to software maintenance. While the target system is still running and serving requests its previously installed and configured software components are fitted by the *Adaptation* task to its current needs. Then the *Deactivation* task enables those maintenance related operations that cannot operate on an active service. The *Update* task replaces previously installed software components and initiates a reconfiguration on them. Finally, the *Decommission* task removes the service's software components from the target system, and then issues a reconfiguration task for the remaining software systems.

We have identified three types of service deployment systems that can accomplish all deployment tasks: (*i*) *manual* service deployment systems, (*ii*) *container based* deployment systems and (*iii*) *appliance based* deployment systems. Manual deployment systems require continuous user interaction during the deployment process, therefore they are not suitable for automated deployment systems. Container based systems (e.g. [7, 24]) predefine an execution platform for the services that includes the deployment system. Therefore services have to be developed for this specific platform and they are not portable between the different container based deployment systems. Finally, appliance based deployment systems (e.g. [15, 19]) encapsulate services and their specific support environments in virtual appliances, thus service deployment is achieved by instantiating a virtual machine that executes the virtual appliance with the service. The next section elaborates the concept of virtual appliances.

2.1. The Appliance Model

During their life cycle software systems may have several versions and subversions. The diversity of their different installations means their vendors have to support an unforeseen number of software environments. These software environments are built up from several software components interfaced with each other. However, the fact that they are installed on the same system means they could also interfere with each other. In several cases the software vendor cannot identify the cause of an irregular situation that only occurs in exceptional conditions.

As a solution to this problem the vendor maintains a specialized environment, and the software system is executed in this controlled environment only. There are two distinct ways to offer this environment: (*i*) with the *Software as a Service* [8] approach the vendor deploys and maintains the software within the administrative domain of the vendor, and the functionality is accessed through

well defined interfaces over the network; (*ii*) with the *Software Appliance* [1] model the vendor packages the software system and its specialized support environment (including the OS and the necessary libraries) and offers it with specific hardware requirements.

Virtual appliances (or VAs - [27]) are software appliances prepared to run in virtualized environments. A virtual appliance defines a VM state that contains the software system and its support environment. Virtualized environments range from software to full hardware virtualization. Software virtualization encapsulates the software appliance and isolates its execution from the OS by concealing its interfaces. As an opposite hardware virtualization offers virtual machines (VM) limiting the resources utilized (e.g. maximum processing power, network bandwidth) by the software installed on them. Hardware virtualization is provided by virtual machine monitors (VMM) that offer virtual machine management functionalities, including their creation, start up and shutdown procedures.

2.2. Requirements for Automating Service Deployment

In this section we identify the requirements an automated deployment system should comply with: (*i*) externally controllable deployment tasks, (*ii*) scales with the size of the service based system, (*iii*) reduced data storage, (*iv*) minimal deployment time, (*v*) minimal disruption of the other services in the service based system. The next paragraphs discuss these requirements in detail.

First of all automation of service deployment requires that deployment tasks (see Section 2) should be exposed with their interfaces to the outside world. As a result, the *control*ling components (e.g. service brokers, orchestrators) of the service based system can influence the entire deployment process.

Service based systems are massively distributed environments, thus the deployment system should be able to *scale* to the size of the service based system. For example, larger sized service based systems involve more frequent deployments. However the increase in the number of deployments should not affect the normal operation of the system.

Frequent deployments intensify the amount of data transferred to the target sites that could affect the network connections of already deployed services. Large-scale service based system involve a large number of deployable services that are stored in repositories. The deployment system should *reduce* the size of the deployable service components in order to achieve their effective storage and transfer.

Highly dynamic service environments require the deployment system to frequently perform service deployments and decommissions. Service deployment often precedes a service call to the newly deployed service instance, because the controlling components of the service based system regularly instantiate deployments when the timely execution of a service call requires it. The deployment system should *minimize the deployment time* in order to reduce the total time of the service call on the newly deployed service instance.

Finally, new service deployments should not *disrupt* the service based system's overall behavior. The newly deployed service should not be able to obstruct the ongoing tasks of the previously deployed services.

2.2.1. Taxonomy of Related Deployment Systems

To further detail our requirements we establish the following categorization for service deployment systems: (*i*) *isolation level* defines the separation between services installed on the same host; (*ii*) *repository support* to store the code of the available services for deployment; (*iii*) *universal* service support increases the number of deployable services in the system; (*iv*) *non invasiveness* requires no modifications on the service code in order to support deployment; finally, (*v*) *state transfer* enables the newly deployed system to resume from the state of a remote service.

Service deployment solutions are categorized depending on their *isolation level* that defines the level of service separation during the deployment on a host already offering services. The lowest isolation level means all the other services are stopped and their states are lost. The highest isolation level does not decrease the turnaround time of the other service invocations during and after deployment or even when malicious code gets activated with a newly deployed service. There are two main approaches to tackle the isolation problem. The first built on the fact that *service containers* offer basic isolation, however service containers do not separate the services entirely (e.g. newly deployed services can exhaust system resources, thus degrading the previously deployed services). The second approach provides isolation with *virtualization* that offers the highest isolation levels. Isolation ensures that improper deployment decisions do not influence the overall system performance therefore, *virtualization based isolation is a key requirement* for an automated deployment system.

Service deployment solutions are also categorized depending on their support of *repositories*. Repositories could act as the primary source of trust if they enclose security information (e.g. a signature of the service's developer) or enforce different registration policies – for example they require the validation of services. Without repositories service code has to be collected before every deployment operation. This gets even worse when the same service has to be collected and deployed several times, instead of downloading it from an already prepared location – the repository. The continuous repetition of the expensive service code collection tasks reduce the overall performance of the service based system even though deployments were initiated to avoid performance drop. Therefore *repositories are required* for the automation of deployment by acting as the sources of deployable services.

The next categorization is the *universality* of the deployment solution that defines the generality of the deployment solution with regards to the deployable services. Specialized service deployment solutions are optimized for

a specific service. They can support all the deployment tasks from installation towards adaptation and decommission, however to simplify the deployment problem they specialize these tasks for a given service. Container based isolation techniques jeopardize universality by supporting services only compatible with the chosen container. In service based systems every service is represented with its interface that cannot be used to differentiate between deployable and non-deployable services. With specialized deployment solutions only few services are deployable, therefore highly dynamic service environments *require universal deployment* solutions that are not differentiating the services in the service based system.

Another categorization is based on the level of *invasiveness* the deployment solution enforces on developers. Invasive systems require the service code to be modified to support the service's deployment. E.g. these systems require the service to implement an additional interface or they require the use of special libraries and solutions that enable further deployments. As an opposite, non-invasive systems are more compatible with the existing service based systems and therefore a *non-invasive deployment solution is required.*

Finally, there are deployment solutions with *state transfer* capabilities. State transfer in service based systems requires that a service suspended on a site is resumable on a different one. In this case the whole process depends on the state representation of the service [23]. The deployment system might introduce new interfaces for state transfer, it might require the developer to implement the state transfer mechanisms for its own system, or finally it can also use a standardized state representation mechanism.

2.3. Related Works

Table 1 and 2 compares the existing deployment solutions according to the previously defined taxonomy.

Nimbus virtual workspace service [15] was not developed for service deployment however, it has been demonstrated that it is also capable of deploying services packaged as appliances [25]. It supports hardware virtualization with Xen [4], and its generic framework lets the developers support other virtualization solutions. Virtual appliances are not stored with metadata in repositories but on a single file-system or a remote http server. In its latest versions the virtual workspace service also supports the cluster on demand concept that enables single services to reside in multiple virtual appliances.

VMPlants [19] goes a step further with appliance based deployment and offers faster delivery of virtual appliances by constructing them on site with the help of directed acyclic deployment graphs. These deployment graphs let the deployment system to build the service from smaller parts coming from a distributed repository called VM Warehouse. Analogous to the virtual workspace service it has two main interfaces, one for managing the virtual machine itself (VMPlant) and one for creating a new one (VMShop). VMPlants defines a framework that includes techniques for representing virtual machine configurations in a flexible manner, for efficient instantiating of virtual machine clones, and for composition of services to support large number of VM "plants".

Hot Deployment Service (HDS) [29] uses an entirely different approach for deploying services by introducing a service for Open Grid Services Architecture [12] based service containers called ServiceFactory. With this service the system is capable to deploy and decommission services while the service container is still running. With the HDS, the service container needs to be changed (at configuration level) to support the operations for new service registration and class loading.

HAND (Highly Available Dynamic Deployment Infrastructure) [24] supports several approaches for deployment and provides two deployment solutions called HAND-C (container level solution that requires the container to be restarted after a service injection), and HAND-S (a service level solution that leaves all other services unaffected during deployment). Between HAND and HDS the only difference that HAND supports the latest Globus Web Services Resource Framework (WSRF - [3]) container and offers container and service level deployment.

WSPeer [14] defines a message-oriented interface for registration, deployment, discovery, and invocation. The authors introduce two implementations; one for regular web services based software, and another for P2P Simplified protocol. The deployment in the WSPeer system is accomplished by passing a class file to the WSPeer service that registers this class as a web service, or in a more complicated case the WSPeer automatically generates a proxy for a given service used by the clients. This second case is used to interface the WSPeer environment with a workflow environment.

Dynagrid [7] offers an all in one solution that covers the aspects of WSRF service [3] deployment and even reaches out of the boundaries of deployment. It offers a solution for a unified invocation interface (called ServiceDoor), and with the help of the ServiceDoor the system even offers service state transfers. Deployments are done with the help of the dynamic service launcher (DSL) components that need to be deployed in all the WSRF containers. This approach implies the most changes in the current service based systems but offers scheduling, migration, and invocation support over the deployment capabilities.

Finally, [33] summarizes the development and standardization activities about Configuration Description Deployment and Lifecycle Management (CDDLM) that is a collection of standards from Open Grid Forum (OGF) focusing on deployment and management. The CDDLM API provides management and deployment interfaces meanwhile the CDL language [32] offers a generic way to configure the deployed application. According to [33] there are four available implementations. For instance the implementation called SmartFrog from HP Labs provides a framework to create deployment solutions for specific soft-

	Isolation	Repository support	Universality	Non-Invasiveness	State Transfer
Nimbus [15]	Virtualised	Partial	√	√	—
VMPlants [19]	Virtualised	√	√	√	—
Hot Depl. srv. [29]	Container	—	√	√	—
HAND [24]	Container	—	√	√	—
WSPeers [14]	Container	—	√	√	—
Dynagrid [7]	Container	—	√	—	√
CDDLM impls. [33]	N/A	—	—	—	—

Table 1: Taxonomy based classification of the different deployment solutions

	Select	Install	Configure	Activate	Adapt	Deactivate	Update	Decommission
Nimbus	—	√	—	√	—	√	—	√
VMPlants	—	√	—	√	—	√	√	√
Hot Depl. serv.	—	√	Partial	√	—	√	—	√
HAND	—	√	Partial	√	—	√	—	√
WSPeers	√	√	Partial	√	—	√	—	√
Dynagrid	√	√	Partial	√	—	√	—	√
CDDLM impls.	—	√	√	√	√	√	—	√

Table 2: Classification of the different deployment solutions depending on their deployment task support

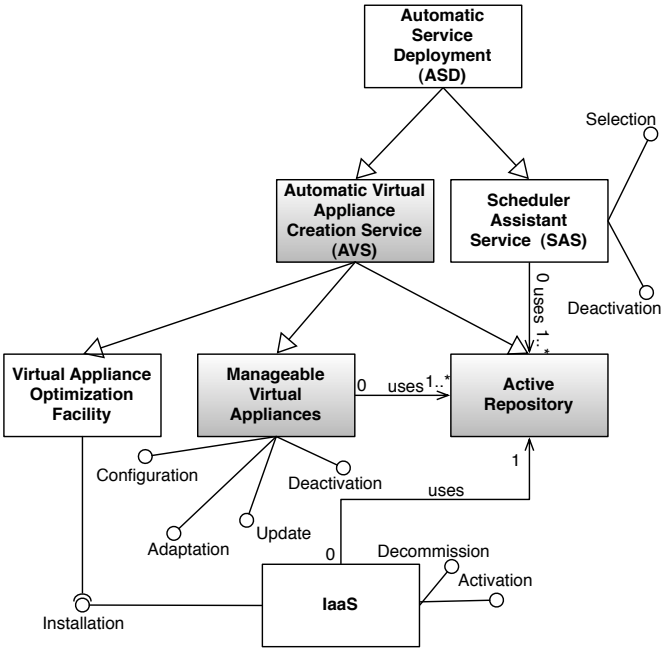

Figure 2: Architectural connections of the Automatic Service Deployment system

ware components. The other implementations only implement selected standards of the CDDLM.

3. Architecture

This section discusses the Automatic Service Deployment (ASD) architecture that we proposed in [16]. This architecture aims to provide an appliance based, universal and non-invasive deployment system that supports all deployment tasks (outlined in Figure 1) on Infrastructure as a Service cloud systems. The proposed components of the architecture are presented in Figure 2. This article only covers the Automated Virtual appliance creation Service (AVS), the Manageable Virtual Appliances and Active repository that are grey colored in Figure 2. However, this section gives a general overview and also discusses the relationships between the different components.

The deployment tasks supported by the ASD architecture are represented with interface lollipops in Figure 2. Beyond supporting all deployment tasks there are two unsolved issues of the current solutions that are targeted by this architecture. First, it offers a solution for acquiring and optimizing virtual appliances. Second, it also offers support for schedulers, service compositors and other controlling components making deployment related decisions in the dynamic service ecosystem. Nevertheless, this article focuses only on the solution for acquiring and optimizing the delivery of virtual appliances.

Virtual appliances are *acquired* and managed with the help of the *Automated Virtual appliance creation Service (AVS)*. The service's main functionality *acquires* virtual appliances from donor systems maintained by the service developers. The service also supports the following virtual appliance *management* tasks: (*i*) transformation of the appliance between various virtual machine formats, (*ii*) size optimization of virtual appliances (called the *optimization facility*) and (*iii*) uploading the virtual appliance to a repository. The general properties and behavior of the AVS are discussed in Section 4.1. In this paper we assume that all parts of the appliances are required for deployment. In contrast, the optimization facility reduces virtual appliances by altering them prior storage. The optimization facility is discussed in detail in a separate paper [17].

In the ASD architecture, virtual appliances are stored in *active repositories* that are defined in Section 4.2. These

repositories are active because they optimize the delivery of their contents by (*i*) decomposing the virtual appliances to smaller parts and (*ii*) replicating the commonly used portions of the stored virtual appliances to other repositories.

The last component is the *manageable virtual appliance* that is a special virtual appliance designed to be embedded into the service's appliance in order to allow the management of the service and its software environment. In this paper we focus on how the manageable virtual appliance is used for rebuilding decomposed virtual appliances. The rebuilding algorithm is detailed in Section 4.2.2.

The *Scheduler Assistance Service (SAS)* is designed to support deployment decisions made by various components of the service based system and the target site selection task. SAS offers (*i*) site ranking to support the selection of the candidate sites for deployment, (*ii*) deployment time and cost estimation on the different sites. This article is not aimed at discussing the SAS.

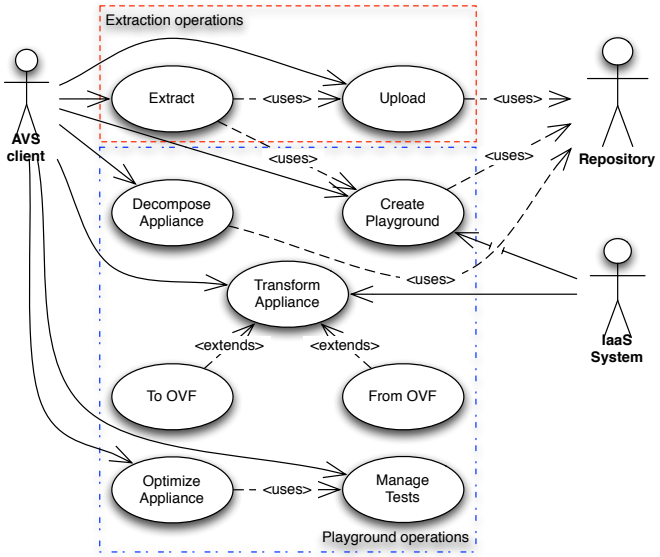

Figure 3: Use cases and relations of the AVS subsystem

4. Virtual Appliance Management

Our research identified four basic operations for virtual appliance management: (*i*) *extraction* of the appliances from preinstalled donor systems, (*ii*) *publication* of the acquired appliances in order to allow their deployments, (*iii*) *decomposition* of the published appliances to optimize their storage and replicate their highly demanded parts, (*iv*) *rebuilding* of the decomposed appliances to allow their faster delivery to the target site before deployment.

Extracting the virtual appliance is the first task in every appliance based deployment system. In the ASD architecture, the automated virtual appliance creation service (AVS) is designed to support the process of extraction and publication. Compared to the frequent deployment requests virtual appliance creation is a rare task, therefore earlier systems leave it as a manual task. With the help of the AVS the deployable virtual appliance is extracted from the developer's system. This operation is further discussed in Section 4.1.

The AVS also supports the *publication* of the acquired appliances in repositories as it is discussed in Section 4.1.3. After the virtual appliance is acquired it is stored in a repository to enable further deployments. Repositories accept virtual appliances using different policies, for example they require third party validation of the uploaded content (e.g. user rating) or repository owners manually select the publicly available appliances. If automated deployments occur target sites decide on allowing deployments by including the acceptance policies of the repositories in their decision, e.g. they only allow the deployment of highly rated appliances.

Virtual appliances are large by nature. Thus, our approach prefers to download them from a high bandwidth and low latency party. Higher bandwidth is achieved by *decomposing* the virtual appliance to smaller portions and the commonly used parts are spread widely allowing their parallel download. This decomposition algorithm is detailed in Section 4.2.

Finally, initiating a virtual machine is straightforward only if its state – the virtual appliance – is available entirely. However, due to decomposition virtual appliances are no longer available in a single package. Therefore, they have to be *rebuilt* before they are used to instantiate a virtual machine. A repository either does this rebuilding process internally, or alternatively with a virtual appliance that is capable of reconstructing itself on site. Both both options are detailed in Section 4.2.2.

4.1. Automated Virtual Appliance Creation Service

The task of the AVS service is to extract a virtual appliance from a virtualized host and package it for later deployments. This package is stored in a virtual appliance (VA) repository. Therefore the AVS has to interact with the following three main actors from the outside world (see Figure 3).

For the *AVS client actor* the AVS offers an interface that exposes the VA extraction functionality. The AVS has to be installed on the same host as the virtual machine monitor (VMM) to ensure the AVS accesses the virtual machine control functionality of the VMM (see Section 2.1). Therefore on client request the AVS collects the state of a virtual machine to form a virtual appliance.

For the *IaaS system actor* the AVS offers two actions. First the AVS enables the *transformation* of a VMM specific virtual appliance format (that can be used to create a virtual machine on a specific VMM) to a platform independent one. This is used when the IaaS system receives a request to deploy a virtual appliance that is not supported by the current VMM. The second, *playground*, operation for the IaaS system is initiated by the AVS itself when

it optimizes the virtual appliances with the optimization facility.

Finally, the AVS uses a package *repository actor*. The AVS stores the packaged VA in the user specified repository and it also replicates it to several other repositories on user request. It is capable to search through multiple repositories to find similarities between a VA package and the appliances stored in the given repositories.

4.1.1. The AVS Client Interface

The AVS has two basic functionalities (see Figure 3). First, it provides operations to extract and publish an initial virtual appliance from a running system (*Extraction operations*). Second, it provides operations on an existing virtual appliance (*Playground operations*). The playground operations either use an existing virtual appliance or one that the extraction operation has just created.

The *extract* operation collects the appliance by creating a snapshot of the user specified machine (either virtual or physical). If the user specifies a virtual machine as the source of the virtual appliance then the extraction operation can create a snapshot of both running and stopped virtual machines. The operation creates the virtual appliances with the file-systems of the specified virtual machine and optionally with the memory state of the running virtual machines.

In dynamic service environments [10] the caller of the service is not aware if a deployment precedes the service call. Therefore the startup time of the virtual appliance is critical because it is added to the time of the first service invocation on the newly deployed service instance. Virtual machines are started up differently depending on the contents of the previously created virtual appliance: (*i*) appliances of running VMs are resuming their previous state by loading their system memory from the appliance, (*ii*) in contrast, appliances of stopped VMs execute the entire boot procedure before the activation of the embedded service in the appliance. The last step of the extraction algorithm temporarily creates two virtual appliances (one with memory state and one without it). Then it measures the startup time of both virtual appliances and only publishes the appliance with faster startup time.

At the end of the extraction process the AVS directly publishes the extracted appliance in a repository or alternatively it creates a *playground* for further optimization of the extracted virtual appliance. There are three main operations the AVS offers on the playgrounds: (*i*) optimization, (*ii*) decomposition and (*iii*) transformation. The *optimize* operation minimizes the size of the virtual appliance that is functionally validated by the tests added with the *Test* operation. The *decompose* operation splits the appliance to smaller parts to allow the delivery optimization of the virtual appliance (this operation is further detailed in Section 4.2). Finally, the two *transformation* operations (*fromOVF* and *toOVF*) enable the AVS service to operate on VMM specific appliance representations by applying the Open Virtualization Format (OVF - [20]) as an intermediary.

Finally, the AVS offers the *upload* operation after all requested operations were executed on the current virtual appliance in the playground. This operation allows the user to upload the appliance to single or multiple repositories. After uploading to a single repository the AVS requests the replication of the uploaded content if the user targeted multiple repositories.

4.1.2. The IaaS System Interface

The result of the *extract operation* (see Section 4.1.1) is a virtual appliance in a format specific to a virtual machine monitor. The AVS service uses the Open Virtualization Format (OVF - [20]) as a generic intermediary format between the different VMM specific appliance formats. During deployment the currently stored appliances might not be supported by the target IaaS system, therefore the active repository converts the generic form of the appliance to a supported format with the *fromOVF* operation of the AVS service.

The AVS also implements the *transformation* functions of the opposite direction (from VMM Specific format to OVF - via the *toOVF* operation). Thus the IaaS system supports the recognition of the appliance in the repository and it acts according to the appliance format. If the appliance format is supported by the VMMs controlled by the IaaS system, then the appliance is used in its original form. Otherwise the active repository initiates a VA playground and requests an OVF transformation: the unsupported VMM specific format is converted to the generic OVF format by the *toOVF* operation, then the supported VMM specific format is achieved by converting the generic format with the *fromOVF* operation.

The optimization *playground* operation uses the IaaS interface to check the service's functionality of the partially optimized virtual appliances. In order to confirm the functionality of the appliances they are deployed on the IaaS system. The playground of the AVS acts as a source of the virtual appliances by either behaving as a repository or temporarily uploading them to the repository of the target IaaS system.

4.1.3. The Repository Interface

Repositories store arbitrary artifacts with their metadata description. The AVS service stores virtual appliances in the repository as the primary artifacts. Therefore a *virtual appliance* is a storage artifact that encapsulates the disk and the optional memory image of a virtual machine in a VMM specific format. The appliance is stored as several file-systems and a swap-like area containing the memory state of the virtual machine just before the image was created. This subsection defines the metadata the AVS service assigns to the appliances to enable the deployment tasks using the stored appliances.

The *repository package* is defined as an artifact encapsulated with the corresponding metadata. Different pack-

age formats use different metadata descriptions, however they always include the following items: (*i*) *Dependency description* to specify other packages required for successfully configure and run the virtual appliance packaged together with the description, (*ii*) *Configuration description*, used by the configuration task, specifies the series of configuration and decision steps that result an executable service, (*iii*) *Human readable description* to support searching and indexing of the software packages within repositories, (*iv*) *Version information* to support exchangeability tests between different repositories. The user of the AVS service defines the metadata required for the repository package manually. However, the AVS automates the metadata collection of dependencies and installation task related information.

Based on the encapsulated dependency description we define two categories of repository packages: (*i*) the self-contained packages and (*ii*) the delta packages. A *self-contained package* encapsulates deployment related metadata with an entire virtual appliance that is usable for immediate VM creation without further modifications. The *delta package* holds parts of a virtual appliance and is dependent on either on other delta packages, or on a self-contained package. Our decomposition algorithm automatically creates delta packages and identifies their dependencies. Virtual appliances are reconstructed from delta packages with our VA rebuilding algorithm. Both algorithms are discussed in Section 4.2.

During the virtual appliance extraction operation the AVS service collects metadata supporting the *installation* task. The collected metadata defines the VMM, the virtualized hardware and network requirements for the virtual machine hosting the virtual appliance. Later, the installation task is performed by the IaaS system using the previously collected requirements available in the repository.

4.2. Extending Repositories with Virtual Appliance Delivery Optimization

In related works repositories are represented as local file-systems or file servers (e.g. FTP, HTTP). The only task of today's repositories is to safely store and provide access to their entries for authorized parties. Therefore they act passively and only user actions change their contents. To minimize the download time of repository entries during the *installation task* we propose the extension of these repositories with automated entry management algorithms.

With our approach download time minimization is accomplished through the virtual size reduction of the appliances. This technique is called virtual because it decreases only the apparent appliance size during delivery. This approach includes a virtual appliance decomposition algorithm (detailed in Section 4.2.1) that separates and replicates commonly used parts of the appliances. These replicated parts can be efficiently delivered to the *target site*, thus only the custom appliance parts are downloaded from dedicated and maybe low bandwidth repositories.

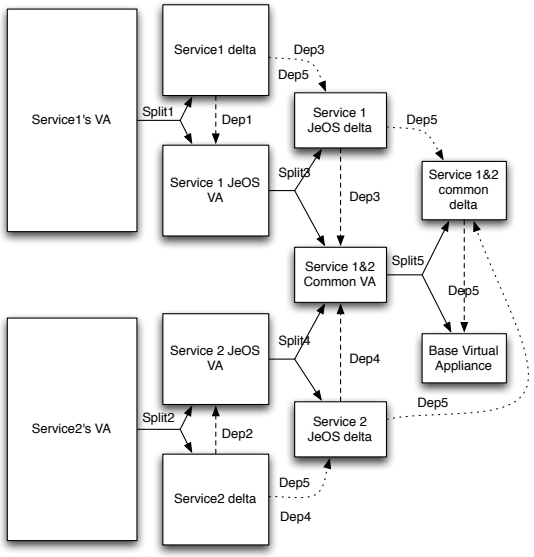

Figure 4: Ideal decomposition graph of two virtual appliances

The separately delivered parts of the appliances are rebuilt with the algorithm proposed in Section 4.2.2.

4.2.1. Active Repository Functionality

Repositories capable of automatic entry management are active entities in the service based system because they automatically (*i*) create, (*ii*) merge, (*iii*) destruct and (*iv*) replicate their entries (the self-contained or delta repository packages). New entries are *created* with our decomposition algorithm. Entries downloaded by the same appliance rebuilding process are *merged* to reduce the repository connections required during the rebuilding process. Low usage of the automatically created or merged entities initiates their *destruction*. Finally, highly used entities are *replicated* to other repositories. All four active repository functionalities are executed as low priority background processes by repositories during low demand periods. The following paragraphs discuss the automation of these management functions.

First, the *decomposition algorithm* is initiated when a new virtual appliance is added to the repository. A virtual appliance (stored in the repository entry) is composed of two parts: (*i*) the service that provides its main functionality and (*ii*) its support system called the Just enough Operating System (JeOS - [13]). The decomposition algorithm identifies these two parts and publishes them as a delta and a self-contained package (see Figure 4). The newly created delta package is registered with a dependency (marked as *Dep{1-5}* in Figure 4) on the self-contained package that holds the JeOS.

The decomposition algorithm identifies the two parts of the virtual appliance in two steps. First it calculates and stores the hash values for all the files in the newly added virtual appliance. Identical hash values allow the identification of those files that are already stored in other virtual appliances. These common files (e.g. see "*Service*

1&2 common VA" in Figure 4) are used to form a self-contained package, while the rest results two delta packages (one from the newly added virtual appliance and one from the appliance with the common files - e.g. "*Service 1 JeOS delta*" and "*Service 2 JeOS delta*" in Figure 4).

When these three new packages are added to the repository the decomposition algorithm is initiated on them. The algorithm stops creating new packages if there are no more common files between the repository entries or if the common files cannot form a self-contained package (they cannot be used to initiate a virtual machine). We call the finally created self-contained package the *base virtual appliance*.

Second, we discuss the *package merging and destruction* management algorithms. The repository monitors the download times and frequencies of the different packages. The algorithm uses this data (*i*) to project expected download frequencies and (*ii*) to identify correlated downloads. Downloads are correlated when packages are downloaded in a specific order during a time period by the same user. The algorithm assigns weight values to correlations inversely to the length of the download time period.

Package merging shortens the virtual appliance rebuilding time by offering correlated delta packages in a bundle. The merging operation is applied to the packages with the highest correlation weight values. The algorithm creates the merged package by bundling together the correlated package contents and updates the dependencies of delta packages that were dependent on all the correlated packages. For example in Figure 4 "*Service1 delta*" and "*Service 1 JeOS delta*" can be merged if other packages are not dependent on them. The last tasks of the merge operation enable the evaluation of the merging decision by (*i*) holding the correlated packages to ensure they are still available as individual downloads and by (*ii*) marking the merged packages.

After a predefined amount of downloads the merging decision is evaluated by checking the download frequencies of the merged and correlated packages. To conserve disk space on the repository hardly downloaded merged packages are *destructed*. As an opposite if the merged package received all the download requests during the evaluation phase then the correlated packages are destructed.

Finally, we explicate the *package replication algorithm* among active repositories. In the algorithm we assume that repository users download from the repository that offers their required packages with the highest bandwidth. Therefore, the algorithm estimates the location of the users and remote repositories by the network latency between the current repository and the remote hosts. These network latency values are used to group (latency groups) the different users and repositories (similar latency values are used as a measure of distance). The algorithm determines the replication target repository by selecting a repository from the latency group with the highest cardinality (this group holds the most hosts with the same distance). Then the current repository creates the replicated entry in the target repository in order to place the required packages closer to the users.

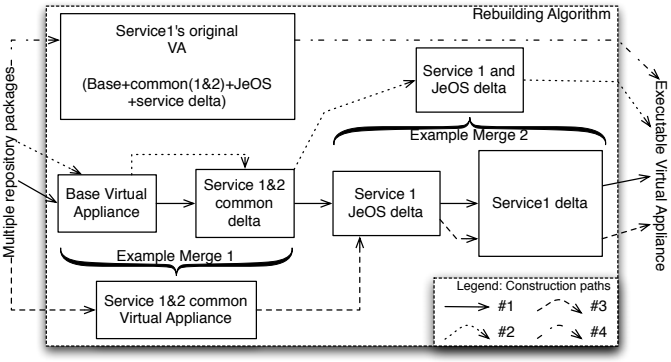

Figure 5: Virtual appliance rebuilding options

4.2.2. Virtual Appliance Rebuilding

After executing the decomposition algorithm presented in the previous subsection virtual appliances are stored in the repositories in multiple packages. The individual decomposed parts of a virtual appliance are not suitable for initiating a virtual machine. The algorithm rebuilds the original appliance before deployment if the appliance is offered as multiple packages. The rebuilding algorithm constructs the original virtual appliance on the execution site using a self-contained package and several delta packages. Therefore the algorithm first *selects the repositories* where the different packages are downloaded from. Then it *reconstructs* the original appliance from the downloaded packages. Finally, it *configures* the rebuilt virtual appliance according to the user requirements.

During the *repository selection* phase the rebuilding algorithm first identifies all required packages and the repositories offering them. The algorithm identifies the repository offering the package with maximum bandwidth by estimating the download time for each package. Figure 5 reveals that after several merges the repositories could contain: (*i*) packages representing the ideal decomposition (discussed in Figure 4), (*ii*) packages resulted from previous merge operations (*example merge 1-2*) and (*iii*) the *original* virtual appliance itself. Based on these packages and their dependencies (represented as arrows in Figure 5) our repository selection algorithm identifies the possible construction paths starting from the decomposed packages and finishing at the rebuilt virtual appliance. Finally, as a result of package merging and destruction there are several ways to construct a decomposed virtual appliance, therefore the selection algorithm chooses the construction path with the lowest cumulative download time.

The rebuilding algorithm includes two *virtual appliance reconstruction* strategies that are applied either before (offline) or after (online) the appliance's hosting virtual machine is initiated. The algorithm decides on the strategy based on the capabilities of the base virtual appliance. If the base appliance offers management interfaces

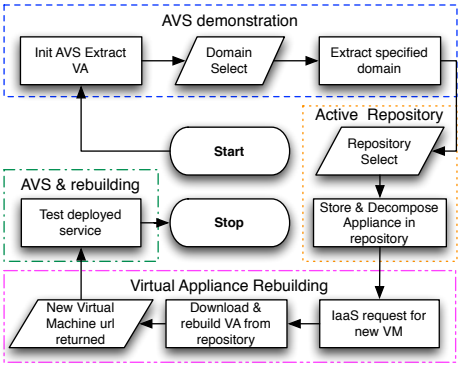

Figure 6: Our demonstration scenario

to install, configure and update components of a virtual machine then our algorithm chooses the *online reconstruction strategy*, otherwise it uses the *offline reconstruction strategy*. Both of these strategies are executed on either the target site or on a nearby repository (if the policy of the target site does not allow the on site appliance construction).

The offline reconstruction strategy downloads, constructs and finally configures the original virtual appliance prior it is used for initiating a virtual machine. The strategy creates a disk image with the base virtual appliance and adds the content of the delta packages to the image. Then the metadata of the original virtual appliance is added to the disk image in order to allow its local deployment. Before the deployment takes place the configuration task is executed by the reconstruction strategy on the disk image. The algorithm executes the *configuration* task just as it would be executed by a virtual machine of the appliance.

The *online reconstruction strategy* follows an entirely different approach that requires the base virtual appliance to comply with the requirement of exposing management interfaces. Therefore the base virtual appliance cannot be decomposed to arbitrarily small packages. This base virtual appliance is used to initiate a virtual machine that is transformed by our reconstruction strategy to host the original virtual appliance. The rebuilding algorithm uses the management interfaces to install the required delta packages on the running virtual machine, therefore most of the installation task is executed within the virtual machine that is going to host the deployed service by the end of this procedure. After the installation task, configuration is accomplished by using the management interfaces to adjust the service in order to conform the configuration description provided by the user.

5. Implementation

In this section we aim at providing a proof for the concepts and algorithms proposed in the previous sections. To demonstrate the capabilities of the ASD architecture, we present a test scenario (see Figure 6) that covers all the highlighted components in Figure 2.

For the demonstration of the architecture we have selected a complex web service (GEMCLA) designed and developed by the University of Westminster. The GEMLCA [9] service has several internal (e.g. a Globus Toolkit 4 (GT4) service container) and external dependencies (e.g. grid sites supporting legacy code execution) in order to support the management and submission of legacy code applications to various grid systems. Therefore in this demonstration the AVS *creates* a virtual appliance for the GEMLCA service and *publishes* it in a repository.

The next major task occurs before the repository offers the content to the public. As part of the *decomposition* process – see Section 4.2.1 – the repository looks for parts of the virtual appliance already published. During the experiment the repository contains a standalone Globus toolkit 4 installation published as a virtual appliance. Thus there are portions in the newly collected virtual appliance that are identical (e.g. the service container of Globus) to the contents of the repository. As a result the virtual appliance of the GEMLCA is decomposed by the repository to a *self-contained package* including the Globus toolkit and a *delta package* including the GEMLCA service itself.

Finally, we initiate deployment on a manually selected IaaS system. As a result the IaaS system initiates a virtual machine using the self-contained package referred by the delta package of GEMLCA. Then the *online reconstruction strategy* (see Section 4.2.2) downloads and configures the delta package inside the newly created virtual machine to install the GEMLCA service. At the end of this procedure we receive the URL of the newly created virtual machine that we use to prove the success of the deployment by requesting a legacy code submission from the newly deployed GEMLCA service.

The successful execution of the demonstration scenario ensures the correctness of the proposed architecture by using all the AVS related functionality: (*i*) the extracted Virtual appliance is functional, (*ii*) the decomposition finds the common parts of the different appliances and (*iii*) the decomposed virtual appliances are still functional after rebuilding.

5.1. Testbed

We have evaluated several available open source IaaS systems (Nimbus, OpenNebula, Eucalyptus) that could use the appliance rebuilding functionality with the smallest modifications. We have chosen Virtual Workspace Service (VWS) offered by the Nimbus project, because it was not bound to a specific repository implementation. Therefore we have extended the VWS service to support accessing and rebuilding appliances from the active repository implementation. For the active repository functionality we have extended the Virtual Appliance ACS – [30] – that implements the Application Content Service (ACS) a proposed recommendation of the Open Grid Forum. This recommendation offers extensible metadata definition (requirements defined in Section 4.1.3), various

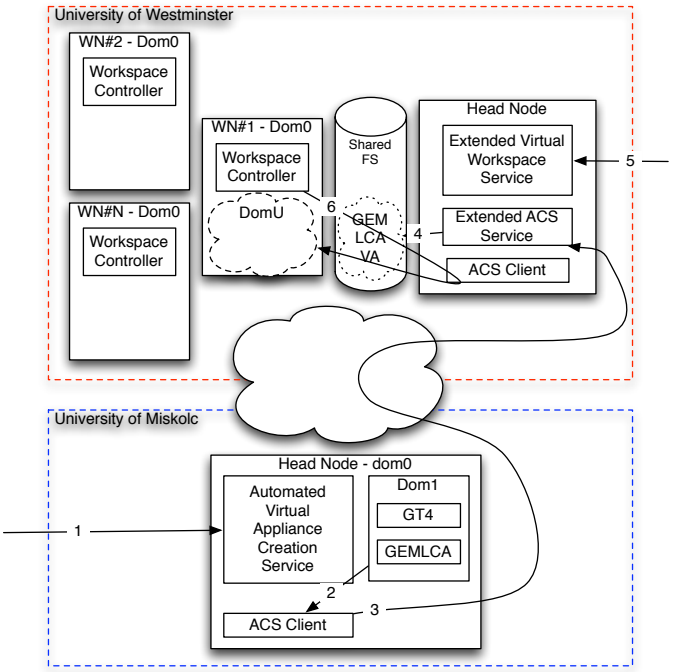

Figure 7: The testbed

metadata search operations (required for the appliance rebuilding and decomposition algorithms detailed in Section 4.2) and seamless integration to web service based systems (simple Nimbus integration).

Figure 7 shows the testbed that was set up for the execution of the demonstration scenario. The testbed currently incorporates two sites one at University of Miskolc (UoM), and another one at the University of Westminster (UoW). UoM runs the original GEMLCA service that is going to be extracted and then deployed at UoW.

The AVS is installed at UoM has access to the local VMMs to support the demonstration scenario. The current AVS implementation supports Xen [4] virtual machines for virtual appliance extraction. AVS installs its extractor components on the Xen host (marked "*dom 0*" in Figure 7) to directly access the state of the executed virtual machines (see Section 2.1 for details). One of the hosted virtual machines (marked "*Dom1*") already runs the GEMLCA service. Finally, at the UoM site there is a repository client installed to enable uploading the extracted content to a repository.

At *UoW* we run a more complex stack. First, the repository and the extended virtual workspace service are installed on the site's head node accessible from UoM. Second, the workspace service also requires a workspace controller component (shipped with the original VWS) on all the virtualization-enabled nodes of the cluster (marked "*WN#X*" in Figure 7). This component instantiates other virtual machines on request. Finally, the extension of the virtual workspace service requires a repository client (marked "*ACS Client*") installed next to the VWS service.

Test of the currently available ASD components. In Figure 7 arrows represent the execution of the demonstration scenario on the testbed. The *first step* starts with the user request for the extraction of the GEMLCA service. Then the virtual machine (called "*Dom1*"), which holds the GEMLCA service, is shut down. After the shutdown, the AVS creates a GEMLCA VA (*step 2*) and uploads (*step 3*) it to the repository located on the *UoW* head-node. In *step 4* the repository scans the uploaded VA for possible decomposition then stores the VA.

We store the configuration file of the virtual machine (which held the GEMLCA service) in the metadata of the created appliance. On user request the extended virtual workspace service uses the stored configuration and the VA files to install the GEMLCA at *UoW* on one of the virtualization enabled nodes (*step 5*). Then in *step 6* the VWS activates the virtual machine that acquires an IP address and the GEMLCA service starts up automatically during its boot process. Finally, after the virtual workspace service reported success we have successfully executed a GEMLCA job on the newly deployed service.

6. Conclusions

We have elaborated the core components of the Automatic Service Deployment architecture that is based on the concept that services can be deployed with virtual appliances. Our research provides a technique for acquiring existing service instances as virtual appliances; furthermore we introduce a new method for virtual appliance storage in repositories. We also provide solutions to decrease virtual appliance distribution cost. The solutions are based on the partial replication of virtual appliance contents where the replicated parts are defined automatically by a decomposition algorithm. The presented decomposition algorithm determines which parts of the virtual appliance have to be distributed and on which sites. Finally, we provide a mechanism to rebuild the decomposed virtual appliances on the target site.

Our approach supports service developers during the creation and publication of their services in the form of virtual appliances. This task is usually performed manually that hinders dynamic service deployment and makes impossible to create dynamic adaptive systems. The primary contribution in our work is a mechanism that automates this task. Consequently, it enables faster adaptation of new services by the various infrastructure as a service cloud systems. As more services become deployable in cloud systems they can increase the dynamism of the service based system. Our virtual appliance rebuilding algorithm reduces deployment time by employing the decomposed and replicated parts of the appliances. Therefore, this solution reduces the apparent service execution time of service calls preceded by deployments.

The current extraction algorithm assumes that the virtual appliances are extracted from virtual machines. Later, this algorithm should be extended to support extracting

virtual appliances from the physical machine of the service developer. In order to speed up the appliance rebuilding process the current implementation of the decomposition algorithm stores the same parts of a virtual appliance multiple times. We plan to investigate and extend the decomposition algorithm with simultaneous optimization of the repository contents for effective disk usage and achievable rebuilding speed-up. We also intend to extend the rebuilding algorithm to give feedback on possible merging of previously decomposed repository content based on the evaluation of different appliance rebuilding strategies.

[1] Appavoo, J., Uhlig, V., Waterland, A., 2008. Project kittyhawk: building a global-scale computer: Blue gene/p as a generic computing platform. SIGOPS Oper. Syst. Rev. 42 (1), 77–84.

[2] Armbrust, M., Fox, A., Griffith, R., Joseph, A., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M., February 2009. Above the clouds: A berkeley view of cloud computing. Tech. Rep. UCB/EECS-2009-28, University of California at Berkley.

[3] Banks, T., 2006. Web services resource framework (wsrf) – primer v1.2.
URL http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf

[4] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebar, R., Pratt, I., Warfield, A., October 2003. Xen and the art of virtualization. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles. ACM, New York, NY, USA, pp. 164–177.

[5] Belguidoum, M., Dagnat, F., 2007. Dependency management in software component deployment. Electr. Notes Theor. Comput. Sci. 182, 17–32.

[6] Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., Brandic, I., June 2009. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25 (6), 599–616.

[7] Byun, E.-K., Kim, J.-S., March 2009. Dynagrid: An adaptive, scalable, and reliable resource provisioning framework for wsrf-compliant applications. Journal of Grid Computing 7 (1), 73–89.

[8] Choudhary, V., jan. 2007. Software as a service: Implications for investment in software development. In: System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on. pp. 209a –209a.

[9] Delaittre, T., Kiss, T., Goyeneche, A., Terstyanszky, G., S.Winter, Kacsuk, P., June 2005. Gemlca: Running legacy code applications as grid services. Journal of Grid Computing 3 (1-2), 75–90.

[10] Di Nitto, E., Ghezzi, C., Metzger, A., Papazoglou, M., Pohl, K., 2008. A journey to highly dynamic, self-adaptive service-based applications. Automated Software Engineering 15 (3), 313–341.

[11] Erl, T., 2005. Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[12] Foster, I., Kishimoto, H., Savva, A., et al., October 2006. The open grid services architecture, version 1.5.
URL http://ogf.org/documents/GFD.80.pdf

[13] Geer, D., October 2009. The os faces a brave new world. Computer 42, 15–17.

[14] Harrison, A., Taylor, I. J., February 2005. Dynamic web service deployment using wspeer. In: Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies. Louisiana State University, pp. 11–16.

[15] Keahey, K., Tsugawa, M., Matsunaga, A., Fortes, J., 2009. Sky computing. IEEE Internet Computing 13 (5), 43–51.

[16] Kecskemeti, G., Kacsuk, P., Terstyanszky, G., Kiss, T., Delaitre, T., February 2008. Automatic service deployment using virtualisation. In: Proceedings of 16th Euromicro International Conference on Parallel, Distributed and network-based Processing (PDP 2008). IEEE Computer Society, Toulouse, France, pp. 628–635.

[17] Kecskemeti, G., Terstyanszky, G., Kacsuk, P., 2011. Virtual appliance size optimization with active fault injection. **Submitted to:** IEEE Transactions on Parallel and Distributed Systems.

[18] Kreger, H., Wilson, K., Sedukhin, I., August 2006. Web services distributed management: Management of web services (wsdm-mows) 1.1. web.
URL http://docs.oasis-open.org/wsdm/wsdm-mows-1.1-spec-os-01.pdf

[19] Krsul, I., Ganguly, A., Zhang, J., Fortes, J. A. B., Figueiredo, R. J., November 2004. Vmplants: Providing and managing virtual machine execution environments for grid computing. In: Proceedings of the ACM/IEEE SC2004 Conference on High Performance Networking and Computing. IEEE Computer Society, pp. 1–7.

[20] Distributed Management Task Force, January 2010. Open virtualization format specification, version 1.1.
URL http://dmtf.org/sites/default/files/standards/documents/DSP0243_1.1.0.pdf

[21] Public EC2 Amazon Machine Images, 2010.
URL http://developer.amazonwebservices.com/connect/kbcategory.jspa?categoryID=171

[22] VMWare public virtual appliances, 2010.
URL http://www.vmware.com/appliances/

[23] Pautasso, C., Zimmermann, O., Leymann, F., 2008. Restful web services vs. "big"' web services: making the right architectural decision. In: WWW '08: Proceeding of the 17th international conference on World Wide Web. ACM, New York, NY, USA, pp. 805–814.

[24] Qi, L., Jin, H., Foster, I. T., Gawor, J., February 2007. Hand: Highly available dynamic deployment infrastructure for globus toolkit 4. In: Proceedings of 15th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP 2007). IEEE Computer Society, Los Alamitos, CA, USA, pp. 155–162.

[25] Rana, A. S., Würthwein, F., Keahey, K., Freeman, T., et al., February 2006. An edge services framework (esf) for egee, lcg, and osg. In: Proceedings of Computing in High Energy Physics (CHEP06). T.I.F.R. Mumbai, India.

[26] Sangal, N., Jordan, E., Sinha, V., Jackson, D., October 2005. Using dependency models to manage complex software architecture. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. ACM, New York, NY, USA, pp. 167–176.

[27] Sapuntzakis, C., Brumley, D., Chandra, R., Zeldovich, N., Chow, J., Lam, M. S., Rosenblum, M., 2003. Virtual appliances for deploying and maintaining software. In: LISA '03: Proceedings of the 17th USENIX conference on System administration. USENIX Association, Berkeley, CA, USA, pp. 181–194.

[28] Schmidt, M., Fallenbeck, N., Smith, M., Freisleben, B., February 2010. Efficient distribution of virtual machines for cloud computing. In: Proceedings of 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2010). IEEE Computer Society, Pisa, Italy, pp. 567 –574.

[29] Smith, M., Friese, T., Freisleben, B., 2004. Hot service deployment in an ad hoc grid environment. In: ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing. ACM, New York, NY, USA, pp. 75–83.

[30] Szmetanko, G., 2008. Virtual appliance acs.
URL http://sourceforge.net/projects/vaacs/

[31] Talwar, V., Milojicic, D., Wu, Q., Pu, C., Yan, W., Jung, G., March-April 2005. Approaches for service deployment. Internet Computing 9, 70–80.

[32] Tatemura, J., January 2007. Cddlm configuration description language specification, version 1.0.
URL http://www.ogf.org/documents/GFD.85.pdf

[33] Toft, P., Loughran, S., 2008. Configuration description, deployment and lifecycle management working group (cddlm-wg) final report.

URL http://www.ogf.org/documents/GFD.127.pdf