Towards an Architecture for Deploying Elastic Services in the Cloud

Johannes Kirschnick[*1], Jose M. Alcaraz Calero[*1,2], Patrick Goldsack[*1], Andrew Farrell[*1]
Julio Guijarro[*1], Steve Loughran[*1], Nigel Edwards[*1], Lawrence Wilcock[*1]

[*1]Cloud and Security Laboratories
Hewlett Packard Laboratories
BS34 8QZ Bristol
United Kingdom
Email: {johannes.kirschnick, jose.alcaraz-calero, patrick.goldsack, andrew.farrell, julio_guijarro, steve.loughran, nigel.edwards, lawrence_wilcock}@hp.com

[*2]Communications and Information Engineering Department
University of Murcia
30100 Murcia Spain
Email: jmalcaraz@um.es

Abstract

Cloud computing infrastructure services enable the flexible creation of virtual infrastructures on-demand. However, the creation of infrastructures is only a part of the process for provisioning services. Other steps such as installation, deployment, configuration, monitoring and management of software components are needed to fully provide services to end-users in the cloud. This paper describes a peer-to-peer architecture to automatically deploy services on cloud infrastructures. The architecture uses a component repository to manage the deployment of these software components, enabling elasticity by using the underlying cloud infrastructure provider. The life-cycle of these components is described in this paper, as well as the language for defining them. We also describe the open source proof-of-concept implementation. Some technical information about this implementation together with some statistical results are also provided.

Keywords: cloud computing, automated service deployment, service provisioning, elastic services

1. Introduction

Cloud computing infrastructures enable the flexible creation of virtual infrastructures on-demand in a pay-for-use model. Organizations such as government, universities and businesses can use these resources in a very flexible manner, by changing the quantity of the rented compute resources to match the actual needs. This new business model promises a reduction of IT infrastructure costs: given that today most IT service infrastructures are provisioned to cope with peak demand, if the excess capacity is rarely needed, the investment is wasted. Renting virtual resources from a cloud provider allows organizations to treat infrastructure as a commodity, leaving them to focus on how to provision and manage the services running on top of this infrastructure. However, the creation of virtual infrastructures is only a part of the process for provisioning services. Other steps such as installation, deployment, configuration, monitoring and management of software components are needed to provide an end-user with a consumable service in the cloud.

Most cloud providers today offer the ability to start a variety of virtual machines, differentiated by selecting the virtual machine image to boot, but usually no further software deployment services. The process of customizing an image to provide a specific service can be split into two approaches, a static and a dynamic approach. The static approach requires an image which contains the appropriate software is pre-configured such that it starts automatically after booting. The dynamic approach relaxes these constraints and allows the image to be customized after it has been started.

The static approach to customization of volume images depends on operations staff to create a volume image prior to deployment, in which the operating system and all the services to be provided are already installed and properly configured according to the requirements of the organization. This approach creates standardized images that can be reused to deploy similar variants of the service, but has many limitations. Firstly, the operation staff needs to perform the installation and configuration of all the required services manually. Secondly, images must be maintained on a frequent basis, applying patches for security and other reasons. Thirdly, each time the image is changed, the new version has to be uploaded to the cloud provider. This might be very time consuming since a typical volume can easily be many GBs and must be transferred over the Internet. Finally, it is difficult to offer flexibility in configuration, since every configuration option leads to a possible new image that needs to be created, uploaded and maintained by the operations staff.

The dynamic nature of cloud computing means that properties of the infrastructure, such as IP addresses or hostnames of machines, are unknown prior to deployment. So, the complete automation of the provisioning process needs to take into account of late binding of such properties.

Our approach is the dynamic installation and configuration of software components after the machines have started. This approach has several advantages. Firstly, everything is done automatically, reducing the time to provision the services. Secondly, only the required software packages rather than the whole volume image needs to be sent though the network, making the deployment process much more agile. Finally, the maintenance of the base volume image is done by the cloud provider themselves.

We use the *SmartFrog* framework (Goldsack, et al. 2009). In previous work it has been used to enable the automatic installation, configuration and deployment of services in distributed environments. In this paper, we describe how we extended and adapted the framework for cloud environments.

The main contribution of this paper is to describe an architecture that enables the automatic provisioning of services in the cloud. In this architecture, the creation of virtual infrastructure is part of the deployment of services. The architecture provides a new life-cycle model that fits well with cloud environments and deals with new features such as elastic services. The proposed architecture has been designed considering service provisioning in cloud infrastructure as a critical service. Thus, it is a fully distributed, extensible and scalable architecture with no central point of failures.

The remainder of this paper is structured as follows: Section 2 describes some related work on automated provisioning of services. Section 3 describes the architecture to carry out the automated provisioning of services in the cloud. Section 4 describes the language used to describe the services to be deployed.Section 5 describes how a new service is managed in the architecture. Section 6 explains how the provisioning of services is carried out. Some implementation aspects are described in section 7. Section 8 provides some experimental performance and scalability results. Finally, section 9 gives some conclusions and discusses future work.

2. Related Work

In recent years there have been several tools developed for automated provisioning of services in distributed environments. For example, *PUPPET* (Turnbull 2007) and *CHEF* (Jacob 2009) are software solutions to automate the installation and configuration of software in distributed environments. They are client-server architectures in which an orchestrator or server is in charge of controlling the provisioning of the services deployed into a set of computers or clients. Even more recently, *Control-tier* (DTO Solutions 2010) and *Capistrano* (Frost 2009) have been released as additions to both *PUPPET* and *CHEF* respectively, providing orchestration capabilities over the installation and configuration processes. Another solution is *CFEngine3* (Burgess 2009), a client-server architecture for installing software components in a distributed environment.

All these tools solve the problem of deploying software in distributed environments, but they do not take into account several requirements associated with cloud infrastructures. Firstly, they do not cover the creation of the virtual infrastructure as part of the deployment process. Secondly, they do not deal with the ephemeral nature and scope for rapid change to cloud infrastructures; instead they assume pre-existing fixed sets of resources. Thirdly, deployment of services is mission-critical and needs to be highly scalable in cloud scenarios, so this service cannot have a single central point of failure and bottleneck ,as is the case in traditional client-server architectures.

In addition to the above, some work has been done by cloud vendors themselves and third parties to assist in provisioning of services. For example Amazon offers pre-configured base volumes which correspond to specific services such as *Amazon Elastic Map-Reduce* (Amazon 2010). Additionally, third parties like *CohesiveFT* (CohesiveFT 2010) enable images to be created and configured dynamically based on preferences which can afterwards be uploaded to cloud infrastructure providers. *Scalr* (Scalr, Inc 2010) is a recent web based solution which offers a pre-defined catalog of services which can be automatically deployed, installed and configured into a virtual on-demand infrastructure using the *Amazon EC2* (Amazon Inc. 2009) cloud provider. This is the closest approach to the solution provided in this paper. However, *Scalr* is closed since it does not provide any way to extend the catalog of services to be deployed. It does not offer fine-grain control over the configuration of the deployed services. Moreover, to the best of our knowledge the *Scalr* architecture has not been published, hampering evaluation by the scientific community.

The architecture described in the paper has been designed to be suitable for cloud environments, overcoming the limitations described above. Thus, our architecture is a totally distributed peer-to-peer network providing a highly reliable scalable architecture with no single central point of failure. This makes our solution better for providing a critical cloud provider service such as the automated provisioning of services.

Moreover, it deals with infrastructure provisioning as part of the service deployment, as well as elasticity over deployed services. It is open source and extensible with new components.

3. System Architecture

This section describes our architecture for automatic provisioning of services in the cloud. Figure 1 gives an overview of the stack of logical layers and tiers available in the architecture.

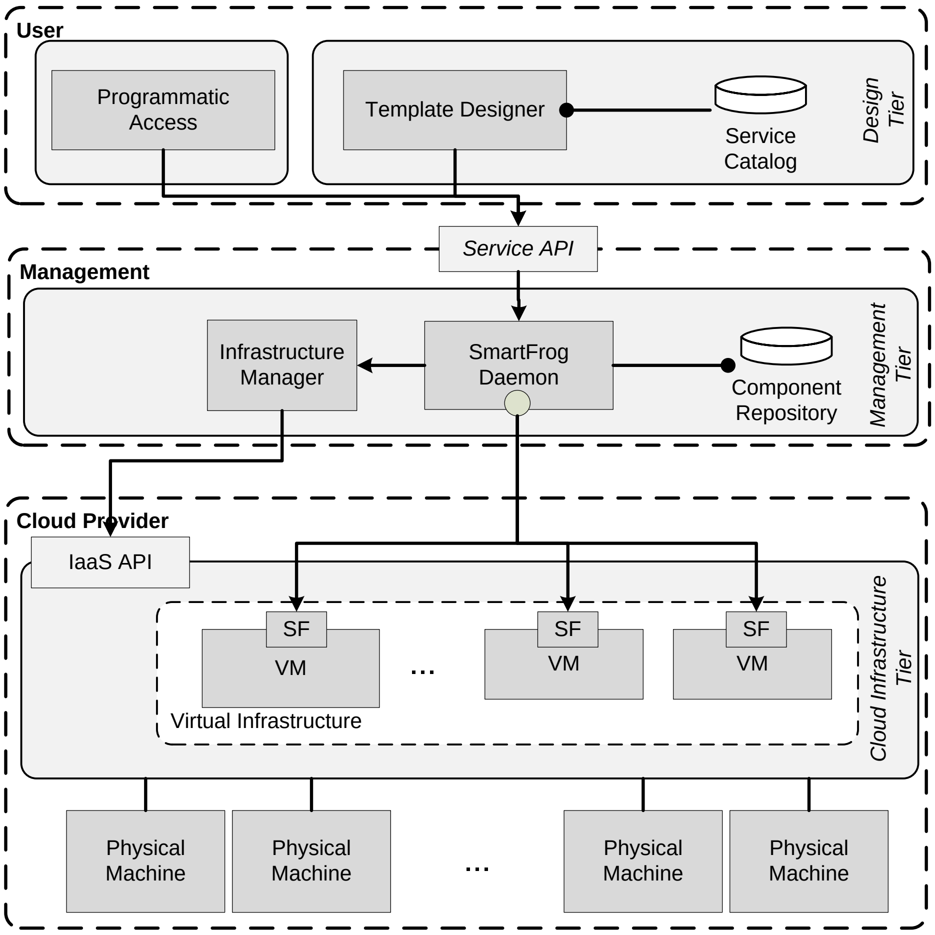


Figure 1. Architecture Overview

The architecture is divided into three tiers. The *Cloud Infrastructure Tier* provides the entry point to the cloud provider, enabling the creation of virtual on-demand infrastructures. Cloud infrastructure providers utilize many physical resources to deliver virtual infrastructures. Currently, there are a number of different commercial vendors with comparable infrastructure offerings, such as *Amazon EC2* (Amazon Inc. 2009), *Rackspace, Hexagrid Computing, GoGrid, Flexiscale, ElasticHosts, etc*. Even though the offerings differ, all of them have at least in common the ability to dynamically create and destroy virtual machines. This is known as *Infrastructure as a Service (IaaS)*. The functionality for managing these virtual infrastructures is provided by an API (*IaaS API*). Usually, this API is provided directly by the cloud vendor provider being exposed at a well-known URL address. Due to its critical nature, the *IaaS API* should be published as highly reliable service with load balancing, fault tolerance and scalability features. These features are usually implemented and controlled by smart management of DNS.

The *Management Tier* is the core layer of the architecture. It carries out the automatic provisioning of services in the virtual on-demand infrastructures. The tier is composed of the *SmartFrog Daemon* and the *Infrastructure Manager* component. The *Infrastructure Manager* manages, creates and destroys virtual infrastructures, providing a common interface across different existing cloud providers. The *SmartFrog Daemon* is the entry point to the peer-to-peer architecture that enables the distribution of the software services among all the peers, automatically provisioning services in such peers. Each *SmartFrog Daemon* uses a *Component Repository* which contains the set of components that can be automatically deployed; it can be extended with new service components, thus increasing the number of deployable services.

The *Management Tier* is distributed across all the virtual machines provided by the IaaS. Each virtual machine in the cloud infrastructure has installed the *SmartFrog Daemon* and optionally can provide the *Management and Design Tiers* shown in *Figure 1*.

The communication model used between *SmartFrog Daemons* is based on unstructured peer-to-peer communication without a discovery process. This requires that each daemon has to know the IP address of its peers. These IP addresses are provided by the SmartFrog daemon in the *Management Tier*: since it creates each new VM it has access to the metadata including the IP addresses for those VMs. When a daemon receives a system description from a user, it is distributed among all the peers according to the deployment information available in the description. This communication model is described in section 6 which explains how the *Management Tier* carries out automatic provisioning of services.

The *Management Tier* offers a *Service API* to clients. This *API* provides clients with the ability to define the services to be deployed automatically in the architecture. This *Service API* uses the *System Description Language*, explained in section 4, to describe such services.

Note that the *Management Tier* has to connect to the *SmartFrog Deamon* running on the created *Virtual Machines (VMs)*. The existence of a *SmartFrog* daemon running on a VM is the only precondition for a VM to work within the architecture, being a common endpoint for accessing inside each of the VMs. To achieve this precondition, the volume image used for these VMs need to have the *SmartFrog Daemon* installed and configured therein, being automatically run during the boot process.

The user submits a *System Description* of the services to be deployed using the *Service API* exposed in the architecture, which in turn, triggers automatic deployment. Although the *SmartFrog Daemon* has to be necessarily running in all the VMs, the number of VMs which exposed the *Service API* of the *Management Tier* can be directly determined by the user according to its requirements (at least one is required to enable a user entry point). Note that having the *Management Tier* running in multiple VMs together with a smart DNS management provides load balancing, scalability and fault tolerance.

The *System Description* can be generated either programmatically or by using any text editor. The *Design Tier* provides a *Template Designer* to aid creation of *System Descriptions*. This *Template Designer* is a smart code editor, with code completion and syntax highlighting features, among others. The *Template Designer* also manages different preconfigured *System Descriptions* in a *Service Catalog*.

4. Service Description Language

This section describes the *Service Description Language (SDL)*. This language is used to describe the different services to be deployed in the cloud infrastructure. The syntax and semantics of this language was described in detail in our previous work (Goldsack, et al. 2009). So this section focuses on the description of the new language features to control the development of services in the cloud.

To illustrate the use of our system we use an example consisting of a system description composed of two *Virtual Machines* provided by the *Amazon EC2* cloud provider. The first virtual machine contains a *Mysql* database configured according to some predefined parameters. The second contains an *Apache Tomcat* application server with the *MediaWiki* application installed. The *MediaWiki* uses the *MySQL* database to provide persistence. Figure 2 shows the *System Description* in *SDL* of this example.

```
#include "softwarecomponents/definitions.sf";
#include "cloudprovider/definitions.sf";

AmazonVM extends VM {
   cloudProvider  "AmazonEC2";
   vol-image "ubuntu-sf";
 }

sfConfig extends {

 vmA extends AmazonVM {
   software extends VirtualSoftwareComponent {
              sfProcessHost LAZY PARENT:ip;
              mysql extends Mysql {
                 user "joey";
                 port  30388:
              }
   }
 }

 vmB extends AmazonVM {
   software extends VirtualSoftwareComponent {
              sfProcessHost LAZY PARENT:ip;
              tomcat extends Tomcat {
                 port 80;
                 https true;
                 https-port 443;
                 apps extends applications {
                    mediaWiki extends MediaWiki{
                     db LAZY PARENT:PARENT:vmA:software:mysql;

                    }
                 }
              }
   }
 }
}
```

Figure 2. Running Example described using the Service Description Language

A *System Description* is composed of *Component Descriptions*. A *Component Description* is defined as a software service managed by the system, e.g. *Mysql*. A *Component Description* is a set of key/value pairs of attributes which represent the configuration parameters of the component that represents this description. A *Component Description* is always associated with a *Component*. A Component's manager is implemented in *SmartFrog* by a Java class. The association of description to component-manager is done using a well-known attribute called *sfClass*. A *Component Description* can inherit the attributes from other *Component Descriptions* using the reserved word *extends*. For example, Figure 3 shows a fragment of the *System Description* from Figure 2 in which an instance of a *Mysql* component (definition from one of the include files, not shown) is (re-)used with some parameters configured. Note that the *mysql* instance not only contains the specified parameters but also contains all the parameters inherited from the *Mysql* component description, which in turn, contains the *sfClass* attribute that associates the *mysql* instance with the *Java* class to manage the *Mysql* component. This way of defining services enables a high-level description of components hiding low-level details.

```
mysql extends Mysql {
            user "joey";
            port  30388:
        }
```

Figure 3. Fragment of the running example shown in Figure 2

A *Component Description* can contain other *Component Descriptions* as part of its definition, creating a parent/child relationship. This is a key concept in the architecture. The language requires the root parent called *sfConfig* which acts as the entry point to the *System Description.* As a result, the *System Description* is always a hierarchical description of components to be deployed.

Figure 2 shows a *System Description* in which two virtual machines *(VM components*) are instantiated. The VM component creates a *VM* on *Amazon EC2* using a volume image in which *SmartFrog* is run during system boot. The VM descriptions contain *VirtualSoftwareComponents*. The *VirtualSoftwareComponents* represent the different software components that will be deployed in the VMs. Note the usage of the attribute *sfProcessHost*. This special attribute is used by *SmartFrog* framework to distribute the *Component Descriptions* across the different peers in the architecture and it determines on which host, in this case a VM, the *Component Description* is processed. This attribute is defined as defined as *LAZY PARENT.ip* in the running example. A *LAZY* reference is a reference which is resolve during run-time (late binding). This value is resolved with the IP address of the VM once it has been created by the cloud infrastructure provider. This resolution method avoids the need for a discovery protocol for *SmartFrog* peers since all the involved IPs are resolved by the component-manager responsible for creating virtual machines. In our running example, a *Tomcat* application server with the *Wikimedia* application is deployed in *vmB* and the *Mysql* database is deployed in *vmA*. The *LAZY* reference to *vmA* defined in the *Wikimedia* application so it can use the *Mysql* component as persistence layer. This reference is resolved after the deployment of the *MySQL* component. A *Java* class is associated with each component, implementing the component-manager to control the component life-cycle including installation, configuration, execution and termination. Details are explained in section 5.

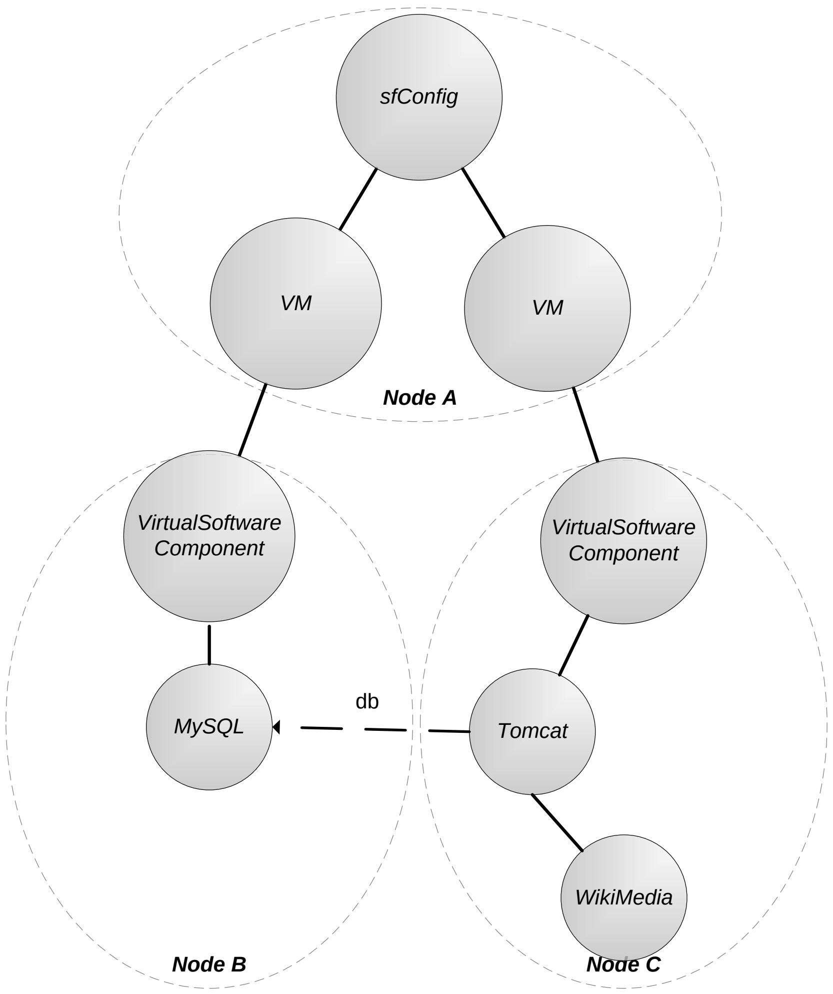


Figure 4. Hierarchical Representation of the Running Example

Figure 4 is the hierarchical tree of the *Component Descriptions* in Figure 2. There are three different nodes in this tree. *Node A* is the VM running the *Management Tier* (see Figure 1) to which the *System Description* has been submitted and *Nodes B* and *C* are the virtual machines created on-demand into the cloud provider. Note that *NodeA* sends the *VirtualSoftwareComponent* descriptions to *Nodes B* and *C* to be processed once they have been created. Further details of these mechanisms are provided section 6. The hierarchical tree of the components is important in the architecture as it affects the life-cycle of the different components, for example if a parent is terminated, all children are terminated.

With respect to our previously published work the new features reported here enable us to define *VMs*, *VirtualSoftwareComponents* and *Cloud Providers* in the *System Description* to drive the creation of a virtual infrastructure and the deployment of services within that infrastructure.

5. Component Management

Management Tier has a component repository with all the components managed in the system. A component-manager is defined by a *Java* class. A *Component Description* usually contains some predefined attributes including the *sfClass* attribute to associate the component description to the *Java* class of the component-manager which controls the life-cycle for a component in the framework. The architecture is extensible, adding a new component requires a new component description and component-manager.

Figure 5 shows the state diagram of the life-cycle for a component in the framework and the methods associated with each state transition. Initially, a component is *deployed* in the target system by means of the *deploy* method. The deployment usually includes the installation and configuration of the component. After that, the component is *started* in order to provide the service. While the component is running, it could be *reconfigured*. And finally, the component is *terminated*. The state diagram defines the *Java* interface to be implemented by a new component manager: deploy, start, terminate and reconfigure actions.

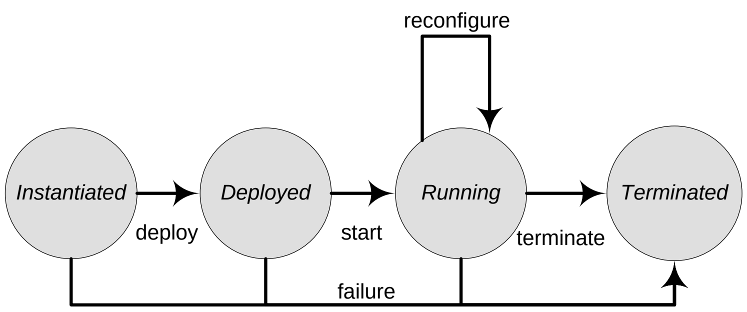


Figure 5. Component Life-cycle

As an example, Figure 6 shows a simplified version of such *Java* class which implements the *VM* component manager used for creating virtual machines in a cloud provider. The *Deploy* method of a VM, instantiates a *CloudAPI Factory* corresponding to the cloud provider specified in the *Component Description* and a new *VM* is created and booted. Note that the VM is booted during the *Deploy* transition rather than during the *Start* transition. This is because the VMs have to be booted to allow the deployment of other components on these VMs. The *IP* address of the new *VM* is added to the attributes of the *VM* component. Finally, the *terminate* method destroys or shutdown the VM in the cloud provider.

```
import cloudapi.*;

public class VM extends Compound{

 CloudAPI facade;

 public void sfDeploy(){
  String cloud =   sfResolve("cloudprovider");
  facade =  CloudAPIFactory.getFactory(cloud);
  String ip =  facade.createAndBootVM(sfName);
  sfAddAttrbute("ip",ip);
 }

 public void sfStart(){
 }

 public void sfReconfigure(){
 }


 public void sfTerminate(){
  if (facade != null){
   facade.destroyVM(sfResolve("ip"));
  }
 }
}
```

Figure 6. Simplified Java class of the VM component-manager

Software component-managers usually implements the *deploy* method calling external *APIs* to packages installation software such as *apt-get*, *yum*, *synaptics* as well as generate configuration files, *XML templates* and *INI templates*. These component-managers usually implements the *start* method calling batch scripts, .exe files, bash scripts, etc. whereas they implements the *terminate* method calling uninstall facilities, stop scripts, etc.

6. Provisioning Services in the Cloud

To explain how the *Management Tier* carries out the automatic provisioning of services, let's consider a use case in which the user submits the *System Description* example depicted in Figure 2.

The user can submit the *System Description* in any *VM* in the system which has the *Management Tier* installed therein. There is no single, central point to which they have to send the *System Description*. Let suppose *NodeA* is this VM. The *SmartFrog* daemon in *NodeA* receives the *System Description* and parses it generating the hierarchical tree of component shown in Figure 4. Then, this tree is processed in *NodeA* using a *breadth-first search algorithm* for processing the deployment of each component.

This algorithm establishes the communication model between the peers. For each tree node (component), the *SfProcessHost* value is retrieved and both Java *class* and

Component Description associated with the component are sent to the peer specified in the *SfProcessHost (localhost* if it is not defined*)*. Regardless of whether the component description is processed locally or remotely in another node, the parent has to wait for the successful deployment of such child nodes. To this end, it uses a synchronous remote method invocation (RMI) programming model matching.

The algorithm shown in Figure 7 is the pseudo-code of this distribution algorithm. In essence, *breadth-first* is a recursive function which processes an *action* on the parent node. After that, it retrieves all its children and for each child node determines where it should be processed using the *sfHost* attribute. As a result, it is recursively processed at the correct destination. Note that *host.breadth-first* is a RMI invocation which causes that parent component to wait until all the child components have executed the same action.

```
function breadth-first(root, action){
  // First perform action on the parent component (deploy and start)
  root.execute(action);

  //Then, perform action on the children components
  childrenComponents = getChildren(root)

  for each childComponent in childrenComponents{
        host = getSfProcessHost(childComponent);
        if ( action = “DEPLOY”){
           send(childComponent, host);
        }
        host.breadth-first(host,action);
  }
}
```

Figure 7. Breadth-first search algorithm used to deploy and execute components

The pseudo code designed as entry point for performing the automated provisioning of services in cloud architectures is shown in Figure 8. Note that deployment and starting of services are always executed sequentially. Then, an event handler is registered processing *Stop* signals to do a clean termination applying a *post-order depth-first* search algorithm and *Reconfigure* signals doing an update of the component descriptions applying the *breadth-first search algorithm*.

```
function main(systemDescription){
  root = parseTree(systemDescription);
  breadth-first(root, “DEPLOY”);
  // at this point all the services has been deployed remotelly.
  breadth-first(root, “RUN”);
  // at this point all the services are runnin

  registerEventHandler(new EventHandler(){
   function terminate(){
     post-order depth-first(root,”END”);
    }
   function update(){
      breadth-first(root, “UPDATE”);
   }
  }
```

Figure 8. Pseudo code for performing the automated provisioning of services

Although *SmartFrog* allows different patterns for orchestrating the life-cycle components, a specific one is used in this work. Our lifecycle pattern creates a virtual cloud infrastructure as part of the automatic provisioning of services. The life-cycle uses the hierarchical tree of component and applies the following rules over it:

- The deployment process is defined as the transition between *Instantiated* and *Deployed* states. It is done following *breadth-first* processing. Firstly, the parent is deployed. Secondly, all the children are deployed. After that, all the children of these children are deployed and so on. In our running example, depicted in Figure 4, the *VMs* have to be deployed to start the deployment of the software components therein (and *Tomcat* before *Wikimedia).*
- All components have to be in state *deployed* before starting the running process and all the components have to be in *running* state before *termination* of any of such components.
- The running process *(*transition between *Deployed* and *Running* states*)* is also done following a *breadth-first pattern*. In our running example, the *Tomcat* has to be in state *running* before the *start* transition is initiated on the *MediaWiki* component.
- A *LAZY* reference can produce a reordering of the node processing search algorithm to resolve the reference
- The terminating process *(*transition between *running* and *terminated* states*)* is done following a *post-order depth*-first pattern. Firstly, all the leaves of the tree are terminated and removed from the tree. Then, all the tree leaves are terminated and removed from the tree and so on. In our running example, the *Wikimedia* has to terminate correctly in order to enable *Tomcat* and *MySQL* component to terminate be correctly and then terminate the *VM* components and so on.

The implemented life-cycle algorithm is able to receive updates of the *System Description* once it has been deployed and running. This feature makes the architecture suitable for dealing with elastic services. The following rules are applies over changes between the old and the new *System Descriptions*:

- If there is a new software component description, it is deployed and executed following the previously described algorithm.
- If there is an update of a software component description, the "UPDATE" signal is distributed.
- If there is an old software component not available, then it is send a "TERMINATE" signal to such components following the *post-order depth-first search algorithm*.

This makes it easy to insert or remove a new *VM* component description to/from the *System Description*.

Figure 9 shows a sequence diagram of the deployment. The SmartFrog framework in the *Management Tier* runs on *NodeA* and is shown as *SmartFrogA* in the figure. When the *System Description* is parsed by *SmartFrogA*, the two *VMs* are firstly visited in the *breadth-first* algorithm. The host in which these components have to be executed is not specified (*sfProcessHost* attribute). By default, *localhost* is assumed. The *deploy* method of these VM components is invoked which creates two new VMs in the *Amazon EC2* cloud provider by means of the usage of the *IaaS API*. From this point, *NodeB* and *NodeC* are available since these VM run *SmartFrog* during the boot process. The *SmartFrog* daemons running in these VMs are shown as *SmartFrogB* and *SmartFrogC* in Figure 9. All the children nodes are retrieved for each *VM* component, i.e. a *VirtualSoftwareComponent* on each *VM*. These children have to be executed in *SmartFrogB* and *SmartFrogC*, respectively. Then, the whole tree branch of the *VirtualSoftwareComponent* is submitted to these hosts to be processed distributedly.

NodeB parses the hierarchical tree received and firstly visit the *Mysql* component using the breadth-first algorithm. The *deploy* method of this component uses the *apt-get* package repository to install *Mysql* and create a configuration file for the database.

Analogously, *NodeC* parses the hierarchical tree received and firstly visit the *Tomcat* component using the breadth-first algorithm. The *deploy* method of this component uses the *apt-get* package repository to install *Tomcat* and generate the configuration files according to the information available in the *Component Description*. Finally, the children of *Tomcat* node are retrieved. According to the *Component Description*, the *MediaWiki* has to be deployed in the same machine so its *deploy* method is invoked. This method uses the *SCP* command to remotely copy *MediaWiki* war file from a remote software catalog to the *Tomcat* application directory. Moreover, the configuration files are generated for use with the *Mysql* as database. The *LAZY* reference enables the *MediaWiki* component to inspect the attributes available in the *Mysql* component, so it can retrieve the information needed to connect to the database.

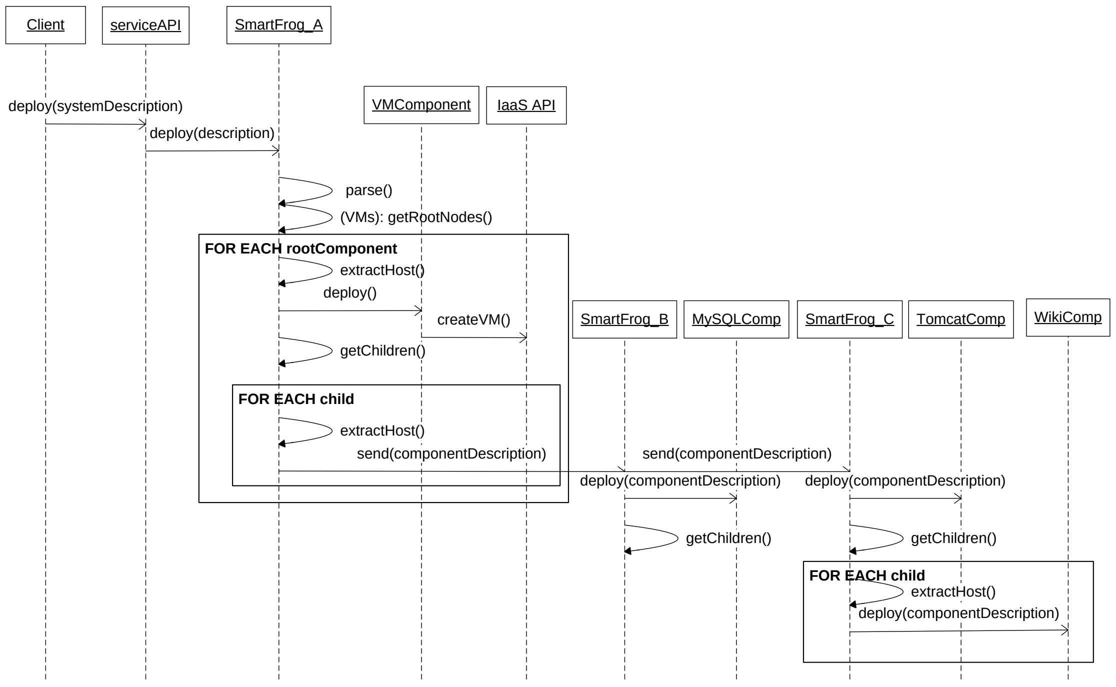


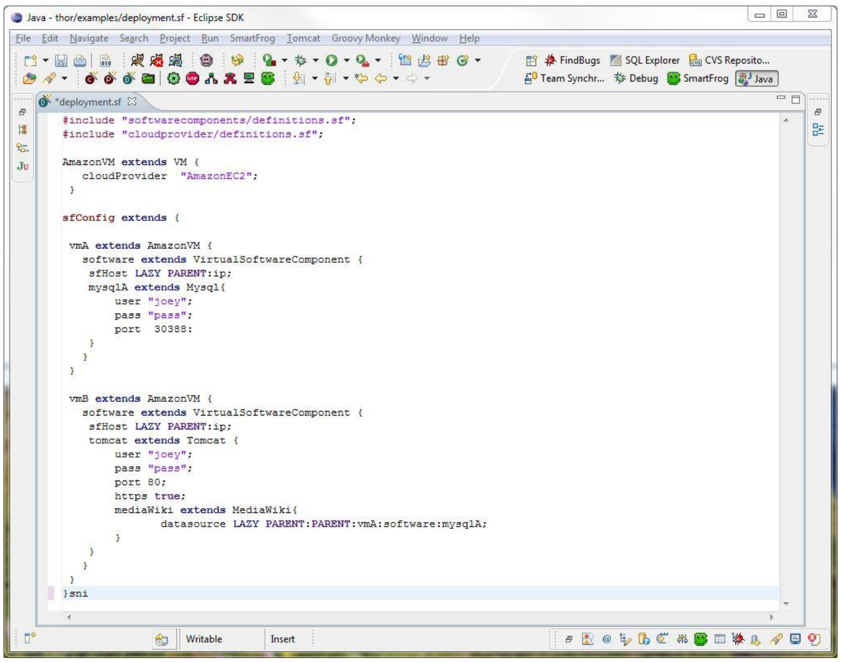
Figure 9. Sequence Diagram of the Running Example Deployment

When the deployment of all components is complete, they are started automatically using an analogous breadth-first algorithm. Finally, when eventually the user submits a terminate signal to the *service API* all the components are terminated using a post-order depth-first search.

7. Implementation

The peer-to-peer architecture described in this paper has been prototyped. SmartFrog has been released as *LGPL* license under the *SmartFrog*[1] *Sourceforge* project. The *SmartFrog* release has more than thirty different components. This framework provides scalability and avoids a central point of failure due to its peer-to-peer architecture. Moreover, it deals with security aspects like mutual authentication of all the peers and encrypted communications. The *VM component* described in this work has been prototypes to support EC2 and an internal-to-HP cloud platform, SUP (Service Utility Platform).

Regarding the *Design Tier* implementation, a *Template Designer* has been prototyped as a proof of concept for both integrated development environments (IDEs): *Eclipse* and *Netbeans*. This *Template Designer* has been designed as a plug-in to aid in the development of *System Descriptions*. These plug-ins provide code auto-completion and syntax highlighting, among other features. They enable the direct execution of the SmartFrog framework from the IDE making easy the testing and debugging processes. The plug-ins are also available in the *SmartFrog Sourceforge* project. Figure 10 shows a snapshot of the plug-in for *Eclipse IDE* showing an screenshot of the example used in this paper.


Java - thor/examples/deployment.sf - Eclipse SDK
File Edit Navigate Search Project Run SmartFrog Tomcat Groovy Monkey Window Help
FindBugs SQL Explorer CVS Reposito...
Team Synchr... Debug SmartFrog Java
*deployment.sf
Writable Insert

```
#include "softwarecomponents/definitions.sf";
#include "cloudprovider/definitions.sf";

AmazonVM extends VM {
   cloudProvider  "AmazonEC2";
 }

sfConfig extends {

 vmA extends AmazonVM {
   software extends VirtualSoftwareComponent {
    sfHost LAZY PARENT:ip;
    mysqlA extends Mysql{
        user "joey";
        pass "pass";
        port  30388:
    }
   }
 }

 vmB extends AmazonVM {
   software extends VirtualSoftwareComponent {
    sfHost LAZY PARENT:ip;
    tomcat extends Tomcat {
        user "joey";
        pass "pass";
        port 80;
        https true;
        mediaWiki extends MediaWiki{
               datasource LAZY PARENT:PARENT:vmA:software:mysqlA;
        }
    }
   }
 }
}sni
```

Figure 10. **Snapshot of the SmartFrog Eclipse Plug-in**

8. Performance

We implemented a test environment on SUP to investigate the performance of the architecture. We used the Mysql/Tomcat example described previously, varying the number of VMs in which the Wiki-based portal service is deployed. The intention is to investigate how efficiently the system reacts when the service has been scaled due to, for example, an explosion in the number of simultaneous requests. The number of VMs

[1] *SmartFrog* is available at *www.smartfrog.org*

occupied by *Mysql* and *Tomcat/WikiMedia* components was adjusted. The number of VMs deployed where kept in the ratio 1:2 with respect to *Mysql* and *Tomcat* services, so one *Mysql* VM provides support for two *Tomcat* VMs. Note that both *Mysql* and *Tomcat* components allows load balancing and this is how these service are deployed.

The Cloud architecture is composed of 6 physical dedicated servers which are capable of managing up to 30 virtual machines. Each physical server has the following setup: Intel Core 2 Quad (4 Cores + 4 Threads) 3.0Ghz, 12MB cache, 1333 FSB, 8Gb RAM, 2 TBytes SATA2 HDD in RAID 0 (2 HDD 1 TBytes). Each server runs *SUP*, the HP IaaS solution running over XEN as a virtualization layer.

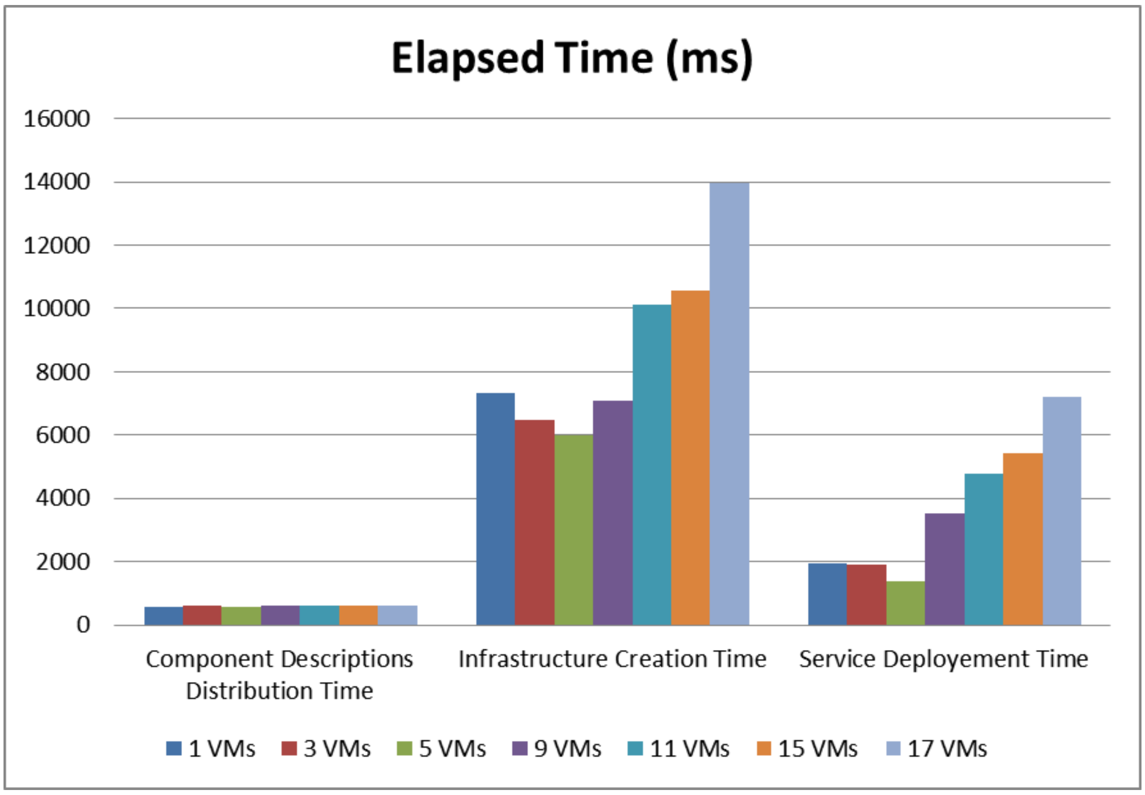


Figure 11. Performace Results for the automated deployment of a Wikimedia Service

Figure 11 shows a constant trend in the time required to distribute all the software components among all the peers in the network which validates the efficiency of the communication model established among all the peers in the network. Between 1 and 5 VMs the creation time varies only slightly, as only one VM is deployed in each physical machine (6 servers total available). For more than 5 VMs, the infrastructure creation time increases roughly proportional to the number of VMs created on each physical server. Service deployment time follows also the same trend. This result shows that the time for infrastructure creation and service deployment is driven by the number of VMs assigned to each physical machine, which is an artifact of the parallel deployment supported in the architecture. A sequential deployment strategy would produce a linear increase with the number of deployed VMs and components..

9. Conclusions and Future Work

A peer-to-peer framework for automated provisioning of services using virtual cloud infrastructure has been described. This main added value of this framework is the incorporation of virtual resource allocation to the software deployment process. A

Service Description Language has been presented which describes the individual software components needed to provision a service, and how they are deployed and configured. This language has been enhanced to capture topology information which describes the allocation of software components to virtual machines. The framework interprets this information to dynamically create the resources using a cloud provider. The framework relies on a peer-to-peer architecture for fault tolerance and scalability: there is no single point of failure and new nodes can be added dynamically.

A prototype has been implemented. The prototype has been designed to be extensible. It supports the incorporation of multiple different cloud vendors into which services can be deployed. Furthermore, new software components can be added to the central component repository, thus extending the number of services that can be dynamically managed by the framework. The use of the *Service Description Language* provides a high degree of flexibility for describing the configuration of services before the deployment. This eliminates the need for preconfigured environments. The component repository allows existing components to be reused in subsequent service definitions.

Additionally, *Eclipse* and *Netbeans* plug-ins have been developed to facilitate the creation of new *Service Descriptions*. They support amongst other features, syntax highlighting and code auto-completion. The plug-ins can communicate with the *SmartFrog Service API* and thus can be used to start the deployment of services from inside of the *Integrated Development Environments (IDE).*

In future work, we plan to look at the incorporation of self-management capabilities into the automatic provisioning of services. This includes finding methods for detecting and mitigating service level violations such as software failures, outages or overload situations.

10. Acknowledgement

Jose M. Alcaraz Calero is supported by the Fundacion Seneca under the post-doctoral grants and the project 04552/GERM/06 and the European Commission under the project FP7- CIP-ICT-PSP.2009.7.1-250453 SEMIRAMIS..

11. References

1. Amazon. *Amazon Elastic Mapreduce.* 2010. http://aws.amazon.com/elasticmapreduce/.
2. Amazon Inc. *Amazon EC2.* 2009. http://aws.amazon.com/ec2/.
3. Burgess, Mark. "Knowledge Management and Promises." *LNCS Scalability of Networks and Services* 5637 (2009): 95-107.
4. CloudKick. *LibCloud.* 2010. http://incubator.apache.org/libcloud/.
5. CohesiveFT. *Elastic Server©.* 2010. http://elasticserver.com/.
6. Cole, Adrian F., and Manik Surtani. *jCloud. Multi-cloud Framework.* 2010. http://code.google.com/p/jclouds/.
7. DTO Solutions. *Control Tier.* 2010. http://controltier.org/wiki/Main_Page.
8. enStratus Networks LLC. *The Dasein Cloud API.* 2010. http://dasein-cloud.sourceforge.net/.
9. Frost, Dan. "Using Capistrano." *Linux Journal* 177 (2009): 8.
10. Goldsack, Patrick, et al. "The SmartFrog configuration management framework." *ACM SIGOPS Operating Systems Review* 43 (2009): 16-25.

11. Jacob, Adam. "Infrastructure in the Cloud Era." *Proceedings at International O'Reilly Conference Velocity*. 2009.
12. Nurmi, Daniel, et al. "The Eucalyptus Open-source Cloud-computing System." *Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid.* 2009.
13. Scalr, Inc. *Self-Scaling Hosting Environment utilizing Amazon's EC2.* 2010. https://scalr.net/.
14. Turnbull, James. *Pulling Strings with Puppet.* FristPress, 2007.