Cross-layer Management of Distributed Applications on Multi-Clouds

Antonis Papaioannou, Damianos Metallidis and Kostas Magoutis
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Heraklion 70013, Greece
Email: {papaioan,metal,magoutis}@ics.forth.gr

***Abstract*—Existing cloud provisioning and deployment frameworks do not yet provide sufficient support for multi-cloud setups. In this paper we improve on the state of the art by adapting the SmartFrog framework to handle multi-cloud setups during the lifecycle (provisioning, deployment, change management, termination) of distributed applications. An administrator of a private cloud additionally needs to be aware of cross-layer dependencies between application components and the physical infrastructure to efficiently carry out a range of administration tasks. In this paper we propose an information repository and system to discover and store such cross-layer dependencies and use it to answer questions such as "which physical machines are hosting particular application components (through hypervisors)" and vice versa. We demonstrate the use of our system in two cases of service-instance (SI) migration and replication on federated multi-cloud setups using the SPEC jEnterprise2010 application benchmark: a) migration of a SI from a private to a public cloud to support the need for maintaining service availability during downtime of physical hardware at the private cloud; b) replication of a SI from a large cloud provider to a regional (or "near") cloud provider to improve response time of clients that are geographically closer to the latter.**

I. Introduction

The provisioning and deployment of distributed applications in increasingly decentralized global infrastructures is an important challenge today. A typical web-based multi-tier application serving Internet clients operates off of one or more (private or public) cloud computing infrastructures. During the course of its lifetime, a large-scale distributed application can be re-deployed several times to adapt to conditions such as downtime of physical infrastructure or to improve availability and performance (e.g., moving the service closer to the physical location of the user). While several application deployment tools have emerged in recent years [1], [2], [3], the advent of cloud computing has introduced new sources of complexity: the need to take into account cross-layer relationships between infrastructure (physical, virtual) and application components and the need to effectively deploy applications on heterogeneous multiple cloud (or *multi-cloud*) platforms. Existing deployment tools are limited in their support for both features.

In this paper we advance the state of the art in both directions by extending HP SmartFrog [4], a declarative approach to distributed application deployment, to take into account cross-layer relationships and to handle multi-clouds. Prior to our work, SmartFrog (Figure 1) was able to configure and deploy distributed application components on already provisioned resources, typically drawn from a homogeneous resource pool. We extended SmartFrog to dynamically provision virtual machines (VMs) from different types of private of public cloud infrastructures, based on resource specifications listed in the application description (top of Figure 1). We further extended SmartFrog to be able to discover dependencies between VMs and the underlying infrastructure and use them in performing re-deployments based on conditions involving the physical infrastructure (such as planned downtime). While the focus of our implementation in this paper is on SmartFrog, our proposed framework is more general and can be adapted to use other deployment and orchestration tools such as those based on the TOSCA standard [5]. To ensure that one can provision VMs of comparable grade across cloud providers we incorporate into our system the ability to rate and compare VM types based on a benchmarking methodology [6].

Our contributions are particularly applicable in the case of cloud federations where private and public cloud providers partner up and enable *cloud bursting* (i.e., the use of additional external resources) across each other. Bursting is especially useful in private cloud providers who may occasionally face resource shortage due to limited in-house infrastructure. In this paper we address the following challenges:

- Lack of support for seamlesss multi-cloud provisioning and deployment in existing frameworks
- Discovery of application components that are affected (may need to be replicated/migrated) when a private cloud provider faces issues with physical infrastructure
- Dynamic provisioning of resources on another cloud provider (resulting in multi-cloud setups) and re-deployment of affected application components there

The paper addresses research challenges in lifecycle management of multi-cloud deployments combining private and public clouds. While there are existing solutions in the areas of (I) deployment of multi-cloud applications (e.g., using APIs such as jClouds [7]); (II) lifecycle management of distributed applications [4], [5]; (III) configuration management of multi-cloud applications [6]; (IV) cross-layer dependencies management [8], [9], currently there is no solution that addresses all four of these challenges together. This paper addresses the research question of how to provide lifecycle management (beyond deployment) of multi-cloud applications achieving user goals on properly characterized public and private clouds. Addressing this research question requires the combination of state-of-the-art solutions to (I)-(IV) integrated into a novel

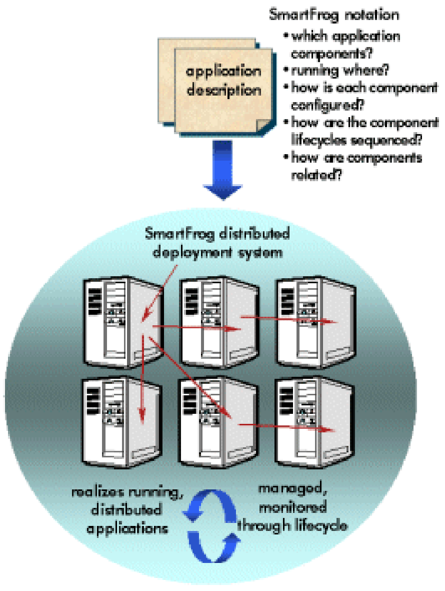
SmartFrog notation
• which application components?
• running where?
• how is each component configured?
• how are the component lifecycles sequenced?
• how are components related?
application description
SmartFrog distributed deployment system
realizes running, distributed applications
managed, monitored through lifecycle

Fig. 1: SmartFrog Overview [10]

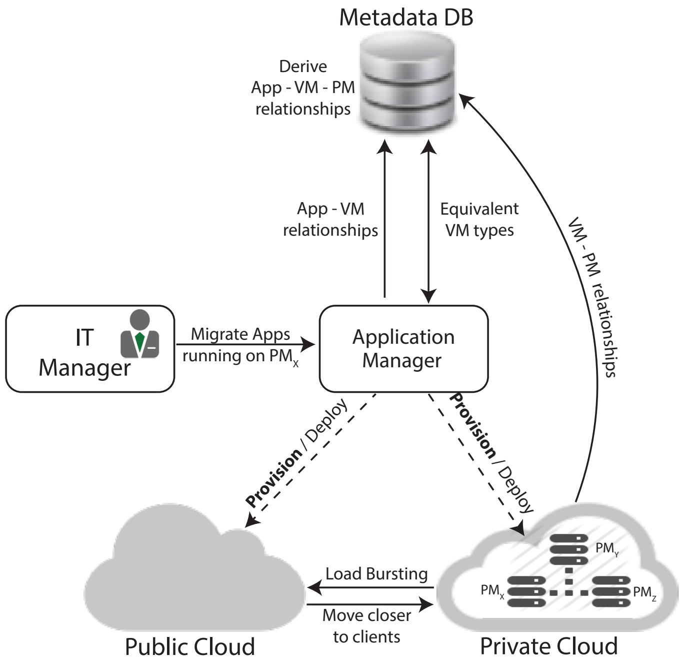

Fig. 2: System architecture

system architecture. Our architecture addresses an important open problem, offers functionality not currently achievable by any single system, and is supported by an evaluation featuring real-world use-cases combining private and public clouds.

We evaluate our system in two bursting scenaria under the well-known SPEC jEnterprise2010 application benchmark: (1) migration of application components from a private cloud to a public cloud to maintain service capacity when the former is facing infrastructure shortages; (2) replication of application components from a large public cloud provider to a smaller regional cloud provider to improve response time for specific clients whose geographic position is close to the regional cloud provider. In both cases, the distributed application deployed over the two clouds is using virtual private network (VPN) technology to isolate and protect its cross-cloud traffic.

II. ARCHITECTURE

Figure 2 depicts the architecture of our system. At the core of the architecture is a deployment and management framework, such as SmartFrog [4] or OpenTOSCA [11], [5], which is used to deploy and manage multi-cloud application resources. We base the implementation described in this paper on SmartFrog and contribute a novel use of it to dynamically provision heterogenous resources over the course of the execution of SmartFrog scripts rather than requiring the provisioning of resources beforehand. A *metadata database* (or MDDB, Section II-B), an extended form of configuration and management repository, collects information from different sources, including SmartFrog (application components and VMs they are deployed on) and private cloud management systems (VMs and physical machines (PMs) they are deployed on). The MDDB builds on this information to derive cross-layer relationships between application components and PMs (as well as other such relationships) where possible (e.g., in a private cloud environment).

An IT manager can utilize this system to adapt the placement of application components to the evolving status of the physical infrastructure. Such a task requires combining the automation provided by frameworks such as SmartFrog with knowledge of cross-layer relationships. A specific use-case we focus on in this paper is the migration of certain application components away from physical servers slated for maintenance soon (e.g., PM$_X$ of the private cloud in Figure 2) and into public-cloud resources (a scenario commonly referred to as *bursting*). In this case, the MDDB relationships are used to determine which application components should be moved and a specific SmartFrog workflow executed to realize the plan. To maintain service capacity, a public-cloud VM type equivalent or better than the private-cloud VM(s) being decommissioned is needed – this information is also provided by the MDDB [6]. Another use case we focus on in this paper is the migration (or replication) of application components from a public cloud to a private cloud to take advantage of proximity to clients (thus better data locality and lower response time).

In the remainder of this section we provide more details on the SmartFrog framework (II-A), the MDDB (II-B), and the role of the IT manager (II-C) within the scope of our work.

A. *SmartFrog*

The SmartFrog framework (Figure 1) comprises an object-oriented language, runtime, and toolset supporting the deployment of distributed applications and the handling of their lifecycle events. In SmartFrog, each software component is modeled as an object whose state represents a set of configuration parameters and whose lifecycle events correspond to specific actions (expressed in Java methods *sfDeployWith, sfDeploy, sfStart, sfTerminate*). SmartFrog programs (descriptions of component configuration, deployment, and workflows) are included in *.sf* files and accompanied by Java implementations of the lifecycle action methods. The latter have access

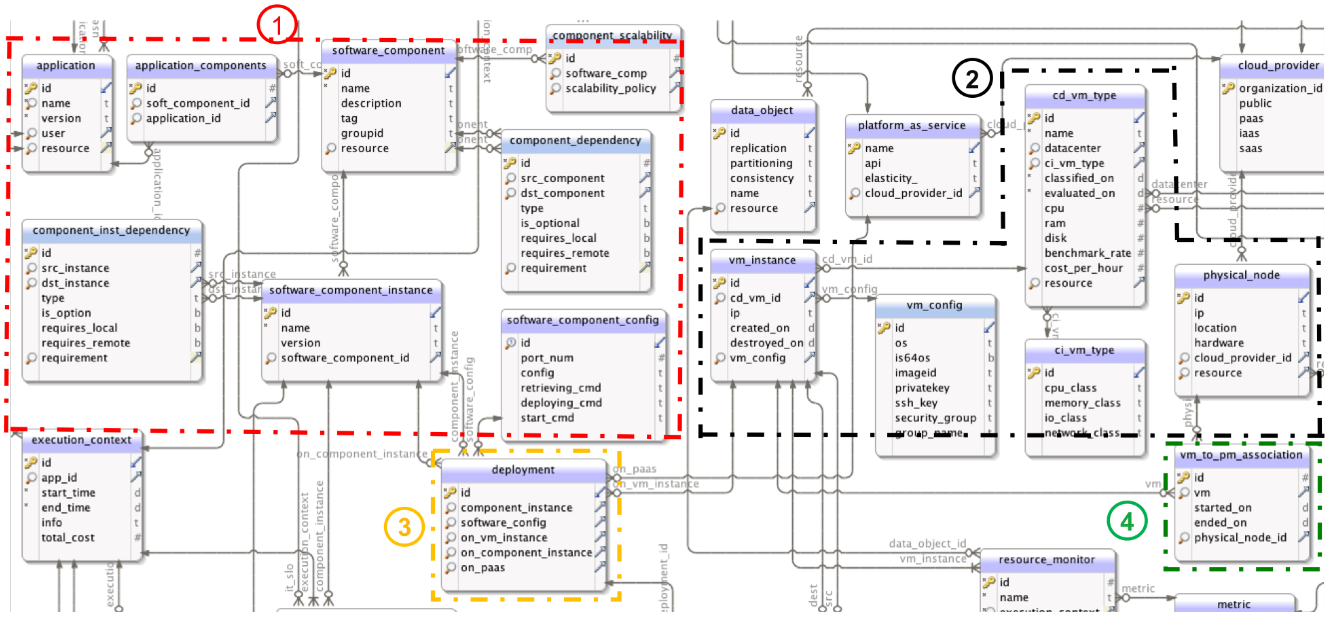

Fig. 3: Subset of the MDDB schema relating to modeling of applications and their underlying resources.

to component configuration attributes via calls to a special *sfResolve()* method. The location where a specific component is deployed and started is specified in the *.sf* file as the address of a server in the *sfProcessHost* attribute of the component.

SmartFrog links the lifecycle of different components by grouping them together in constructs such as *compound* (start and terminate components together), *parallel* (start components concurrently but let them evolve independently), *sequence* (a component starts after its previous component in the sequence terminates), etc. For compound components, *sfDeploy* is called for each component in the order that the components are listed in the .sf file and prior to any other component's (within the compound construct) *sfStart* being called. The SmartFrog runtime consists of a collection of daemons (called *sfDaemons*) present in each managed system and responsible for components deployed there. Application components may cross-reference each other's attributes. Some attributes may be lazily bound at deployment or execution time (they must be declared as such using the LAZY keyword).

To date we are only aware of uses of SmartFrog to deploy distributed applications in homogeneous environments where resources (VMs) are provisioned and known in advance. In this paper we extend the applicability of SmartFrog to on-demand provisioning of cloud VMs and dynamic deployment of software components on them. We model VMs as SmartFrog component objects, similar to any other application component. Each of these VM objects has its own lifecycle, state, configuration parameters, and management events.

The execution of SmartFrog descriptions of applications originates at a management machine and may involve other SmartFrog-enabled machines. The IPs of dynamically-provisioned VMs are typically not known until they are up and running. A specially created PublicCloudVM or PrivateCloudVM SmartFrog component encapsulates the requirements of provisioned VMs (type of VM, name of VM image, etc.) and (late-bound) attributes such as the allocated IP address of the VM. VM requirements must be set by component methods through invocation of *sfResolve()* prior to provisioning of the VM. Actual provisioning of the VMs takes place within the *sfDeployWith* method of the PublicCloudVM or PrivateCloudVM component. This ensures that a VM is provisioned prior to the sfDeploy method of depending components (which require access to the VM's attributes) being executed. The *sfProcessHost* attribute is late bound to the allocated IP address ensuring the object is deployed onto the provisioned VM.

In this work we have leveraged SmartFrog support for a workflow system [12] and event framework. The framework provides a distribution mechanism for events hiding the details of where events are generated or consumed. Our main use of the SmartFrog workflow framework in this paper concerns communicating requests for adaptation between the IT Manager and components responsible for reconfiguration actions.

B. *Metadata Database*

The MDDB [6] stores a variety of information. In the remainder we focus on the subset of information that is relevant to our work in this paper (Figure 3). For reference, the full schema is avalable online [13]. Applications are modeled using the SOFTWARE_COMPONENT class (abstract descriptions of software components, e.g., a generic servlet component) and APPLICATION_COMPONENT class (used to model applications and their corresponding components) depicted in the upper left corner of Figure 3 (rectangle 1). A software component is characterized by its ID, name, description, and tag (relating it to a taxonomy of software component types). An application has an ID, name, version, and belongs to a certain user.

A SOFTWARE_COMPONENT_INSTANCE provides a more specific description (including configuration attributes) compared to the generic SOFTWARE_COMPONENT class. A software component can be deployed either on another software component object or on a VM_INSTANCE object, which represents a provisioned VM resource. The deployment relationship is modelled with the DEPLOYMENT class (rectangle 3 in Figure 3).

The MDDB captures the identities of the VMs on which an application is deployed along with the cloud providers they are sourced from. Rectangle 2 in Figure 3 depicts the classes that represent this type of information. The CLOUD_PROVIDER class stores information about private or public cloud providers. Information about a virtual machine instance, such as its name, IP address and lifetime is stored in the VM_INSTANCE class, while the VM instance configuration properties are modelled in the table VM_CONFIG. Each VM instance is of a particular CD (cloud dependent) VM type and CI (cloud independent) VM type. A CD VM type describes a real-world VM type offered by a cloud provider whereas CI VM types are the result of (periodic) classifications into cloud-agnostic categories with similar VM capabilities [6]. The MDDB additionally models the underlying physical nodes and their properties as well as the association between them and the VMs they host (classes PHYSICAL_NODE and VM_TO_PM_ASSOCIATION in rectangle 4).

The MDDB receives information about application components and the VMs they are deployed on from deployment management systems (e.g., the SmartFrog runtime) or infrastructure discovery probes [8], [9]. It receives information about VMs and the PMs they are hosted on from private cloud managers or other infrastructure management systems. In case of public clouds, the topology can be inferred from measurements or exported to interested parties by the cloud provider via special management APIs [14]. The MDDB builds on this information by producing derivative cross-layer relationships between application components and physical servers. For example, if the MDDB contains the associations $p_i \rightarrow v_j$ and $v_j \rightarrow a_k$, where p_i, v_j, and a_k are a PM, a VM, and an application component respectively, then we infer that $p_i \rightarrow a_k$ (see Section III, Listing 5 for an example query).

C. IT Manager

The IT Manager is concerned with traditional IT management tasks (which are often human driven or supervised) aiming to best align applications with their underlying infrastructure. It is thought of as independent of the cloud platforms used. However when application deployments involve private cloud platforms more information is available to the MDDB and thus the IT Manager has access to a wider range of tasks. In this paper we specifically focus on management tasks involving the use of resources drawn from multiple clouds.

Resources in private clouds are typically limited. In certain cases, demand for extra resources may require outsourcing from public providers (a process also known as *load bursting*). In other cases a scheduled maintenance on the physical resources of the private Cloud platform may disturb the normal operation of the application. The IT Manager component can take as input this kind of information (e.g. the physical nodes that will stop operating) and query the MDDB to identify the affected VMs and the corresponding applications they host. Then it can schedule the migration of the applications software components to an alternative public Cloud provider without any manual effort.

III. Implementation

SmartFrog code and runtime support. In this paper we have SmartFrog-enabled the software stack of a representative enterprise application, the SPEC jEnterprise2010 benchmark. The software stack includes the JBoss application server, MySQL database server, application logic packaged in an enterprise archive file (specj.ear), and a load balancer interposed between load-generating clients and application servers (the inclusion of a load balancer is an enhancement over the standard version of SPEC jEnterprise, which supports a single application server instance). The SmartFrog version of SPEC jEnterprise2010 operates as follows: Each component (JBoss, MySQL, load balancer) includes a sfStart() method that triggers the installation and deployment of the respective component. Components terminate via the sfStop() function.

The SmartFrog management code for SPEC jEnterprise2010 comprises the following four main components

- Virtual machine, dynamically provisioned in a public, private or hybrid cloud
- Database server
- Application server
- Load balancer, responsible for spreading client load over multiple application servers

Listing 1 shows SmartFrog declarations of abstract components that will be extended and referenced later in *.sf* files. A key abstract component is *Public_Cloud_VM*, which represents the provision of VMs of a specific type. The attributes of *Public_Cloud_VM* include the cloud provider the VM is going be deployed on, the type of the VM (e.g small, medium, etc), an indication of which software component will be deployed on it (taking values in LB (load balancer), AS (application server), DB (database), etc). An important attribute in *Public_Cloud_VM* is *VM_IP* whose value is late bound to the IP address of the VM when the latter is provisioned and started. Late bound variables in Listing 1 are declared using SmartFrog's TBD keyword. The binding is performed in the *sfDeployWith()* function of any concrete VM class that extends *Public_Cloud_VM* (see Listing 2). *LB* and *ReconfigureLB* are the SmartFrog components responsible for deploying and reconfiguring the load balancer during a migration. The *LB* component consists of two attributes, the *IP* address of the load balancer, and the IP of the provisioned application server (*Worker_IP*). The SmartFrog declarations for the initial deployment of a full SPEC jEnterprise2010 software stack are shown in Listing 2.

Listing 1: Abstract SmartFrog code

```
Public_Cloud_VM extends Prim{
        sfClass          "...";
        CloudProvider    TBD;
        VM_IP            TBD;
        typeOf           TBD;
        SoftwareTypeVM   TBD;
}
LB extends Prim{
        sfClass          "...";
        LB_IP            TBD;
        Worker_IP        TBD;
}
ReconfigureLB TBD;
```

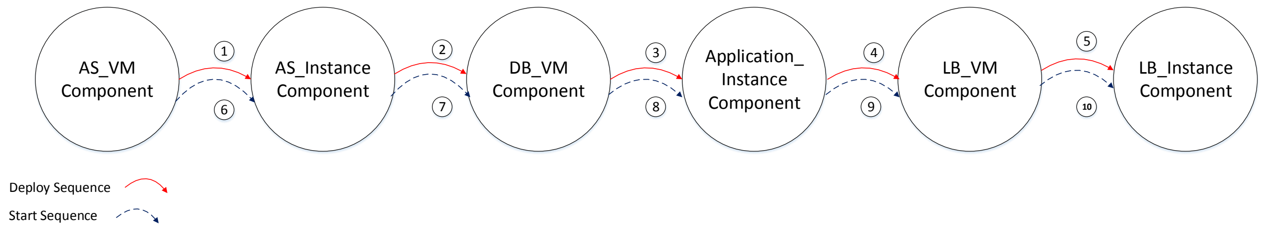

Fig. 4: Solid (red) arrows denote deployment sequence ("happens before"); Dashed (blue) arrows denote start action sequence.

To respect provisioning and deployment dependencies we use SmartFrog's Compound statement to link the lifecycles of grouped components. Within a Compound (Listing 2) components are first deployed in the stated sequence prior to being started in the same sequence, as shown graphically in Figure 4.

Listing 2 shows a Compound deployment structure where the VMs that the application server and database server are going to be deployed on are provisioned first. The code that performs the actual provisioning is pointed at by the *sfClass* attribute (details omitted for clarity). Then component *Application_Instance* deploys the application logic (expressed in *specj.ear*) and connects to the IP addresses of the provisioned database and application servers. Finally a load balancer instance is provisioned and deployed. The *LoadBalancer* component expresses the load balancer reconfiguration logic. Note that components expressing middleware to be deployed on previously provisioned VMs in Listing 2 (such as *Application_Instance*) are late-bound and cross reference the *VM_IP* attribute of their underlying VM. Similarly components at higher levels (such as *LB_Instance*) cross reference attributes of their underlying middleware components.

Listing 2: Initial Deployment SmartFrog code

```
InitialDeployment extends Compound{
    AS_VM extends Public_Cloud_VM{
        CloudProvider     "Amazon";
        VM_IP             "NotKnownYet";
        typeOf            "m1.small";
        SoftwareTypeVM    "AS";
    }
    AS_Instance extends Prim{
        sfClass           "...";
        AS_Name           "JBoss-1";
        AS_IP    LAZY     AS_VM:VM_IP;
    }
    DB_VM extends Public_Cloud_VM{
        CloudProvider     "Amazon";
        VM_IP             "NotKnownYet";
        typeOf            "m1.small";
        SoftwareTypeVM    "DB";
    }
    Application_Instance extends Prim{
        sfClass           "...";
        DB_Name           "MySQL-1";
        AS_IP    LAZY     AS_vm:VM_IP;
        DB_IP    LAZY     DB_VM:VM_IP;
    }
    LB_VM extends extends Public_Cloud_VM{
        CloudProvider     "Amazon";
        VM_IP             "NotKnownYet";
        typeOf            "m1.small";
        SoftwareTypeVM    "LB";
    }
```

```
    LB_Instance extends LB {
        LB_Name            "mod_jk-1";
        LB_IP              LAZY LB_VM:VM_IP;
        Worker_IP          LAZY AS_VM:VM_IP;
    }
}
```

Listing 3 shows the onEvent component (part of SmartFrog's workflow [12] architecture) that handles the adaptation event. The *Load_Balancer* component handles the arrival of one or more events describing conditions that require adaptation. The *singleEvent* attribute declares whether the onEvent component should respond once or multiple times to the arrival of a potential stream of events. *sfProcessComponentName* declares the name of the component for resolution purposes.

Listing 3: onEvent SmartFrog code

```
LoadBalancer extends OnEvent{
singleEvent false;
sfProcessComponentName "LoadBalancer";
    ReconfigureLB extends LAZY LB{
        sfClass    "...";
        WorkerIP   LAZY HOST Migration_Provisioner_IP
            :AS_IP:VM_IP;
        LB_IP      LAZY HOST Initial_Provisioner_IP:
            VM_LB:VM_IP;
       }
}
```

Listing 4 shows a sendEvent component (also part of SmartFrog's workflow architecture) that triggers the event called *ReconfigureLB* on a specific host and software component via specific references, in this case the *LoadBalancer* component (Listing 3).

Listing 4: sendEvent SmartFrog code

```
LoadBalancerEvent extends EventSend{
      sendTo:a  LAZY HOST Initial_Provisioner_IP:
          LoadBalancer;
      event       "ReconfigureLB";
    }
```

MDDB and queries to discover cross-layer relationships. The MDDB [6] is used as a central repository to collect information about the Cloud infrastructure topology (VM-to-PM relationships) in case of private cloud platforms and information regarding the deployment of an application (application components-to-VM relationships). The MDDB is also a provider of information regarding cross-layer dependencies, such as which application components are hosted on a specific PM. The MDDB exposes an API through which clients such as

SmartFrog can obtain the necessary relationships. The MDDB executes queries like the one described in Listing 5 to derive the cross-layer information on demand. We decided not to introduce all possible cross-layer relationships explicitly in its schema by explicit new classes that model this information as these relationships may change over time and this could result in unnecessarily high complexity. Instead we support the modeling and discovery of specific relationships when needed and let the client decide on when or how frequently to execute queries against the MDDB to store the results.

Listing 5: Query identifing cross-layer dependencies

```
SELECT sci.*
FROM software_component_instance sci
WHERE sci.id IN
    (SELECT component_instance
     FROM deployment
     WHERE on_vm_instance IN
        (SELECT inst.id
         FROM vm_to_pm_association assoc, vm_instance
             inst
         WHERE assoc.vm=inst.id AND assoc.
             physical_node_id IN
            (SELECT id
             FROM physical_node
             WHERE ip='xx.xx.xx.xx')
        )
    )
```

Low-level support for cross-cloud provisioning. The multi-cloud provisioning logic we embedded in SmartFrog uses Cloudify [2] and the *dasein* [15] and *boto* [16] libraries to operate across different cloud architectures. An obstacle we faced early on was that there was no support available for the version of our Eucalyptus private cloud or for the Flexiant FCO public cloud platform. We thus decided to create a custom Cloud Driver to support Eucalyptus using the *boto* [16] library, and Flexiant FCO using our own extended version of the *dasein* [15] library. The custom cloud driver along with the use of the *Amazon AWS SDK* [17] allows us to provision resources across providers such as Eucalyptus, Flexiant FCO, Amazon EC2, Microsoft Azure, etc. An alternative way would be to use the API of a portable cloud platform such as Jclouds [7]. However no such portable cloud API supported our full needs and therefore we opted for our own custom cloud driver.

IV. Evaluation

We evaluate our system using the distributed SPEC jEnterprise2010 benchmark [18] as a case study. SPEC jEnterprise2010 is a full system benchmark that allows performance measurement and characterization of Java EE 5.0 servers and supporting infrastructure. The benchmark models supply a chain consisting of an automobile manufacturer (referred to as the Manufacturing Domain) and automobile dealers (referred to as the Dealer Domain). The Web-based interface between the manufacturer and dealers supports browsing a catalog of automobiles, placing orders, and indicating when inventories have been sold. The SPEC jEnterprise2010 application requires a Java EE 5.0 application server and a relational database management system (RDBMS), which comprise the *system under test* (SUT). The primary metric of the SPECjEnterprise2010 benchmark is throughput measured as jEnterprise operations per second (EjOPS). A load generator (referred to as a Driver) produces a mix of browse, manage, and purchase business

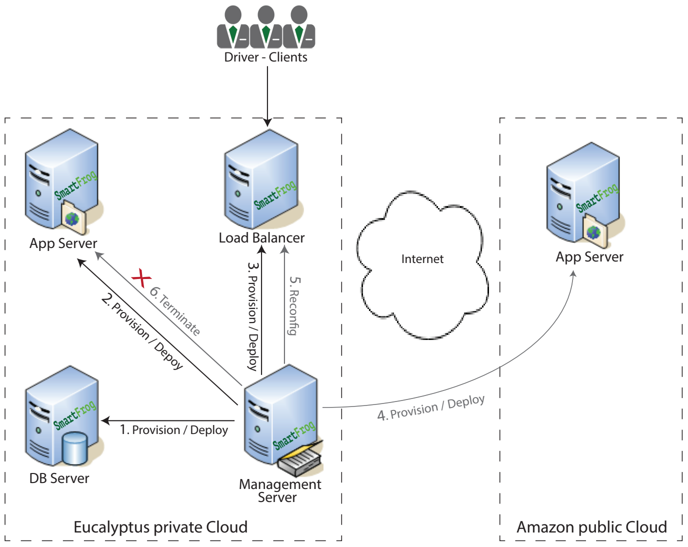

Fig. 5: Scenario 1 deployment procedure

transactions at the targeted injection rate (or txRate, equal to the number of simulated clients divided by 10), which aims to produce the targeted EjOPS. The Driver measures and records the response time (RT) of the different types of business transactions. Failed transactions in the measurement interval are not included in the reported results. At least 90% of the business transactions of each type must have a RT of less than the constraint threshold (set to 2 seconds for each transaction type). The average RT of each transaction type must not exceed the recorded 90^{th} percentile RT by more than 0.1 seconds. This requirement ensures that all users see reasonable response times. The Driver checks and reports whether the response time requirements are being met during a run.

We model the SPEC jEnterprise2010 application using three software components corresponding to the business logic of the application, the application server, and the RDBMS [6]. These components are instantiated as *specj.ear* and *emulator.war* files, a JBoss 6.0 application server, and a MySQL 5.5. We deploy SPEC jEnterprise2010 in our in-house Cloud platform running Eucalyptus 3.1.2 (the latest version that supported Xen and the AMD processors used in our servers).

A. Scenario 1

In this scenario we demonstrate our system handling the migration of a SPEC jEnterprise2010 service instance from our private in-house cloud platform to a public cloud provider (Amazon EC2) to guarantee service availability and capacity during scheduled downtime of the underlying physical infrastructure in the private cloud. In this scenario, the IT Manager wants to move all activities off of a specific physical machine so that it can perform maintenance to it. To achieve this it queries the MDDB (Listing 5) for the cross-layer relationships between the physical machine and software components that depend on it. Normally, a cloud provider will be able to satisfy the needs of migrating the software components off of the targeted physical machine from internal resources. However, in times of resource strain (to which a private cloud operator is more vulnerable) the IT Manager needs to use

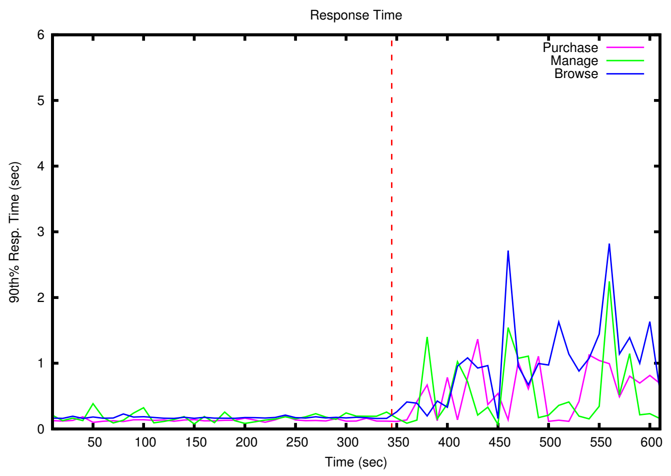

Fig. 6: Transaction response time in scenario 1.

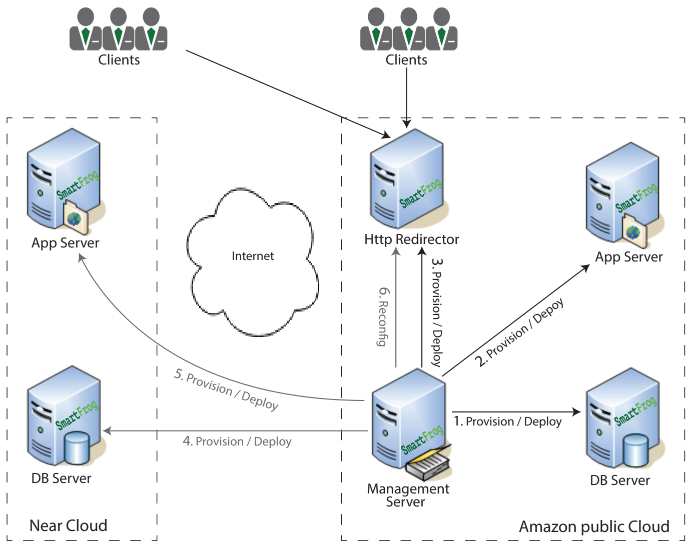

Fig. 7: Scenario 2 deployment procedure

external resources in order to maintain service capacity. In this scenario, the IT Manager uses SmartFrog to migrate the identified components (application servers) to Amazon EC2. The VM allocated on Amazon should have equivalent or better performance compared to the replaced VM in order to maintain service capacity. We use previous work on cloud-independent classification of VM types [6] (whose results are stored in the MDDB) to achieve this.

The application traffic between Clouds is routed through a VPN in order to federate the two Cloud platforms and isolate and protect the communication. From a deployment procedure viewpoint, Figure 5 shows the sequence of actions when deploying and migrating the SPEC jEnterprise2010 application. Steps 1-4 in Figure 5 show the provisioning of VMs and deployment of the application servers, the database and the load balancer in our private cloud. Steps 5-7 show the migration process. In step 5, SmartFrog creates a m1.small VM instance on the Amazon EC2 (EU) cloud platform and deploys the application server. In step 6 the load balancer is reconfigured to direct the load to the new application server instance. Finally in step 7 an application server VM instance in the private cloud is terminated as the migration procedure is completed and the traffic is directed to the new application server. Our system supports full automation in this scenario.

Figure 6 shows response time (RT) of the application before and after the migration. The load balancer reconfiguration (step 6 in Figure 5) is complete at 340 sec, at which point traffic is directed to the new application instance. RT is observed to increase as the new application server instance is now hosted at a public cloud provider, which is typically at a significant distance from the load balancer and database and the communication among them is routed over the Internet. The good news however is that the service is still available at acceptable levels (within the RT SLO of 2 sec (avg)). The impact on RT could be further reduced if we can opt for a public cloud provider with a data center closer to the private cloud or if a larger number of application servers were involved (reducing the fraction of accesses using the remote instance).

B. *Scenario 2*

The second scenario focuses on the case where a service deployed on a public cloud is replicated to a regional (typically private) cloud to achieve better response time by taking advantage of proximity to clients. The application deployed on two cloud operators, initially on an Amazon EC2 data center (eu-west-c1) and later replicated to a regional cloud provider (our local cloud) to improve response time. Figure 7 shows the SmartFrog procedure of achieving the initial deployment and replication tasks. The MDDB was used in this case to determine the type of VMs that should be allocated on the private cloud (of equal or better grade compared to those the service is already running on Amazon) to achieve a comparable level of service.

Figure 8 depicts SPEC jEnterprise2010 transaction response time when clients close to our geography are accessing the service instance deployed on the Amazon eu-west-c1 cloud (0-300sec) and the significant drop in response time in the 300-600sec time interval when a service instance (the full stack, including application server and database) are replicated to a cloud provider near the clients (such a regional cloud is often called a *near cloud*). This is an important scenario in the emerging field of federated clouds that combine major cloud providers and smaller regional players, demonstrating the ease with which our architecture can support such use cases.

V. Related Work

We discuss related work in two major areas: application deployment systems and cross-layer relationships in distributed systems.

Application deployment systems. Automating the provisioning of VMs and the deployment of distributed applications on them has been a longstanding goal of management systems such as Tivoli Provisioning Manager (TPM) [19]. Cloudify [2] is a recently introduced open source tool whose main goal is to provision resources from a single cloud and then deployment, monitor and management the lifecycle entity of

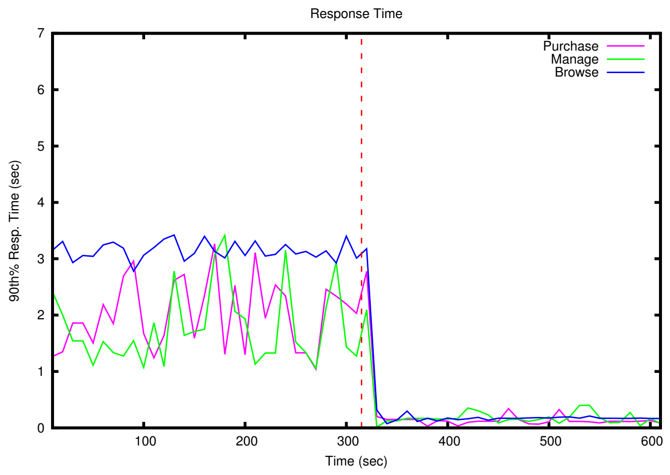

Fig. 8: Transaction response time in scenario 2

applications. Cloudify tasks are expressed in *recipes* that describe in detail the deployment and execution aspects of an application. Chef [3], another open-source framework, language, and toolset has recently become a popular platform for the deployment and management of distributed applications. A growing set of Chef *cookbooks* and *recipes* (akin to Cloudify recipes) are currently available in a publicly accessible repository [20]. SmartFrog [4] is a declarative open-source approach to application deployment and management that is used in this work. In SmartFrog, the configuration and deployment details of components are described using a Java-like domain-specific language and supports the execution of workflows [12]. Section II-A provides a brief introduction to SmartFrog.

Cross-layer relationships. The discovery of cross-layer relationships in distributed systems has been a topic of significant interest in the past. Several systems used online system monitoring and analysis of traces [21], [22], [23] while others used input from discovery sensors to build cross-layer relationships over multiple IT domains [9], [8]. The system described in this paper is closer in principle to the latter approach combining input from sensors from the domain of infrastructure clouds and application deployment systems. Our contributions in this paper focus on the use of cross-layer relationships in managing multi-cloud deployments, including private clouds.

VI. Conclusion

In this paper we highlight the value of cross-layer relationships in the domain of application deployment and adaptation in multi-cloud environments including private clouds. Our aim in this paper is to demonstrate that distributed application deployment engines can extend their scope and applicablity by taking into account relationships between software components, virtual infrastructure, and the underlying physical infrastructure. We support our aim by describing an application deployment system with multi-cloud capability and with the ability to handle adaptation events taking into account cross-layer relationships. We evaluate our system in two use-cases of practical interest involving service-instance migration between clouds. We show that (unlike existing systems that have no access to such information) they can support both with efficiency achieving the desired results.

Acknowledgment

We thankfully acknowledge the support of the PaaSage (FP7-317715) EU project.

References

[1] K. Bhargavan, A. Gordon, T. Harris, and P. Toft, "The Rise and Rise of the Declarative Datacentre," Microsoft Research, Tech. Rep. MSR-TR-2008-61, May 2008.

[2] "Cloudify." [Online]. Available: http://getcloudify.org/

[3] "DevOps Chef." [Online]. Available: http://www.getchef.com/

[4] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The smartfrog configuration management framework," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 1, pp. 16–25, Jan. 2009. [Online]. Available: http://doi.acm.org/10.1145/1496909.1496915

[5] "OASIS: OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0 Committee Specification 01 (2013)."

[6] A. Papaioannou and K. Magoutis, "An architecture for evaluating distributed application deployments in multi-clouds," in *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 01*, ser. CLOUDCOM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 547–554. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2013.79

[7] "jClouds library." [Online]. Available: http://jclouds.apache.org/

[8] K. Magoutis, M. Devarakonda, N. Joukov, and N. G. Vogl, "Galapagos: Model-driven discovery of end-to-end application-storage relationships in distributed systems," *IBM J. Res. Dev.*, vol. 52, no. 4, 2008.

[9] "IBM Tivoli Application Dependency Discovery Manager," Accessed 3/2014. [Online]. Available: http://www-03.ibm.com/software/products/en/tivoliapplicationdependencydiscoverymanager

[10] "SmartFrog," Accessed 4/2014. [Online]. Available: http://www.smartfrog.org/display/sf/SmartFrog+Home

[11] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "OpenTOSCA – a runtime for TOSCA-based cloud applications," in *11th International Conference on Service-Oriented Computing*, ser. LNCS. Springer, 2013.

[12] "SmartFrog Workflow," Accessed 3/2014. [Online]. Available: http://www.hpl.hp.com/research/smartfrog/releasedocs/smartfrogdoc/sfWorkflow.html

[13] "MDDB Design," http://users.ics.forth.gr/~papaioan/projects.html.

[14] I. Kitsos, A. Papaioannou, N. Tsikoudis, and K. Magoutis, "Adapting data-intensive workloads to generic allocation policies in cloud infrastructures," in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, 2012, pp. 25–33.

[15] "Dasein Cloud API," Accessed 5/2014. [Online]. Available: http://dasein-cloud.sourceforge.net/

[16] "Boto - Python interface to Amazon Services," Accessed 5/2014. [Online]. Available: http://boto.readthedocs.org/

[17] "AWS SDK." [Online]. Available: https://aws.amazon.com/sdkforjava/

[18] "SPEC jEnterprise2010 Benchmark," Accessed 5/2014. [Online]. Available: http://www.spec.org/jEnterprise2010/

[19] "IBM Tivoli Provisioning Manager," Accessed 3/2014. [Online]. Available: http://www-01.ibm.com/software/tivoli/products/prov-mgr/

[20] "OpsCode Chef Cookbooks," Accessed 5/2014. [Online]. Available: http://community.opscode.com/cookbooks

[21] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance Debugging for Distributed Systems of Black Boxes," in *19th ACM Symposium on Operating Systems Principles*, October 2003.

[22] A. Kind, D. Gantenbein, and H. Etoh, "Relationship Discovery with NetFlow to Enable Business-Driven IT Management," in *1st IEEE/IFIP International Workshop on Business-Driven IT Management*, 2006.

[23] H. Kashima, T. Tsumura, T. Ide, T. Nogayama, R. Hirade, H. Etoh, and T. Fukuda, "Network-Based Problem Detection for Distributed Systems," in *21st International Conference on Data Engineering*, 2005.