Autonomic Provisioning, Configuration, and Management of Inter-Cloud Environments based on a Software Product Line Engineering Method

Alessandro Ferreira Leite
University of Brasilia
E-mail: alessandro.leite@acm.org

Vander Alves
University of Brasilia
E-mail: valves@unb.br

Genaína Nunes Rodrigues
University of Brasilia
E-mail: genaina@cic.unb.br

Claude Tadonki
MINES ParisTech / CRI
E-mail: claude.tadonki@mines-paristech.fr

Christine Eisenbeis
Inria Saclay / Université Paris-Sud
E-mail: christine.eisenbeis@inria.fr

Alba Cristina Magalhaes Alves de Melo
University of Brasilia
E-mail: albamm@cic.unb.br

***Abstract*—Configuring and executing application across multiple clouds is a challenging task due to the various terminologies used by cloud providers to describe their services and features. Likewise, the services are regularly offered at different levels of abstraction, such as infrastructure-as-a-service (IaaS) and platform-as-a-service (PaaS). While IaaS services provide low-level access to the infrastructure, PaaS services enable the users to delegate the management of the computing environment to the cloud providers. Consequently, at the IaaS level, the users are responsible for managing the computing resources, whereas, at the PaaS level, the users must develop native cloud applications following the constraints defined by the PaaS provider. These two options exist, mostly because the clouds target web applications, whereas users' applications are commonly batch-oriented. Considering how difficult is the task of configuring and executing applications across various clouds, we advocate the use of autonomic systems to do this work automatically. For this purpose, in this paper, we propose and evaluate an autonomic and goal-oriented system. Our system implements self-configuration, self-healing, and context-awareness properties. In addition, it relies on a hierarchical P2P overlay (a) to manage the virtual machines running on the clouds and (b) to deal with inter-cloud communication. Likewise, it depends on a software product line engineering (SPLE) method to enable applications' deployment and reconfiguration at runtime, without requiring pre-configured virtual machine images. Experimental results show that our system frees the users from the duty of configuring and managing the execution of a non-native application on single clouds and over many clouds. In particular, our system tackles the lack of middleware prototypes that can support different scenarios when using simultaneous services from multiple clouds.**

I. Introduction

Deploying and executing applications on infrastructure-as-a-service (IaaS) clouds demand cloud and system administration skills. Likewise, it represents an error-prone and difficult task. The difficulties are mostly associated to the kind of applications considered by these clouds. While the clouds focus on web applications, the users' applications are many times batch-oriented, performing parameter sweep operations. Furthermore, users applications may require specific configurations and some of them may take days or even weeks to complete their executions.

While the former issue can be dealt with pre-configured virtual machine image (VMI), the latter requires monitoring and fault-tolerance strategies in order to reduce the chances of losing the work just before it completes. Nonetheless, creating virtual machine images is often time-consuming and demands advanced technical skills. Monitoring, on the other hand, is usually realized by employing heartbeats [1]. Heartbeats are presence notification messages sent by each virtual machine to a monitor [2]. In this case, when a heartbeat message fails to reach the monitor within a given threshold, the virtual machine (VM) is considered unavailable and a process to recovery it or to create a new one starts.

Since clouds are susceptible to failures, it is interesting to consider an inter-cloud scenario. In an inter-cloud, a service or application runs on resources distributed across multiple clouds [3], and these clouds are unaware of each other [4]. Consequently, the users are responsible for aggregating the resources distributed across different clouds to execute their services or applications. Figure 1 illustrates an inter-cloud scenario, where two services run across three clouds.

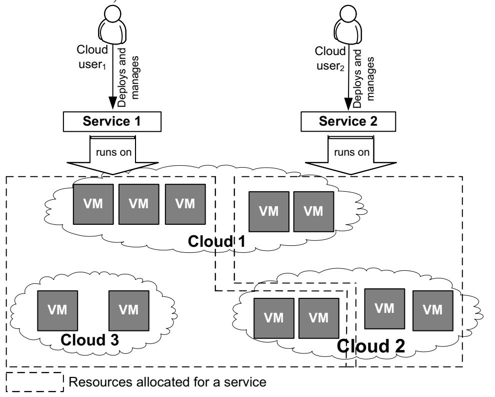


Fig. 1: Example of an inter-cloud scenario, where two different services runon resources distributed across three clouds

In this case, the users have to: (a) identify the clouds and the VM types that fit the applications needs; (b) select the VMI to launch the VMs on each cloud; and (c) deploy the applications considering functional and temporal dependencies, as well as clouds' constraints. Additionally, the users should be aware of the clouds' internals and the communication pattern of their applications in order to reduce the financial cost, for instance. For example, in some cloud providers, the cost of data transfers between virtual machines in the same data center and using their external IP addresses is the same as the cost of transferring data between different clouds or region sites. Moreover, the network throughput is often lower when using external IP addresses. Hence, changing the applications to consider the location of the nodes or handling this work manually may be very difficult for the users, especially when there are various cloud providers and the environment can change at runtime.

As can be seen, automatic inter-cloud configuration adds a new dimension that currently lacks a successful technology/product. As a result, pre-configured virtual machine images is the common approach used to configure such cloud environments. Thus, an inter-cloud scenario highly benefits from automatic resource discovery mechanisms, automatic matching process, and homogeneous management interface. In practice, this means automated coordination of multiple cloud environments. In other words, inter-cloud environments require autonomic computing systems that can manage and adjust themselves according to environmental changes in order to achieve users' objectives [5]. Thus, such systems can free the users to concentrate on what they want to accomplish with the systems rather than spending their time overseeing the systems to get them there [5].

Although configuration management tools have evolved to enable a declarative specification of the desired state of the resources, these tools require both computer programming and system administration skills from the users, and they do not deal with automatic resource selection in a single or in an inter-cloud scenario. Examples of these tools include Chef (*chef.io*), Ansible (*ansible.com*), Docker (*docker.com*), Juju (*juju.ubuntu.com*), and Vagrant (*vagrantup.com*).

Contribution: in this paper we propose and evaluate an autonomic and goal-oriented system for inter-cloud environments. Our system, namely *Dohko*, relies on a software product line engineering (SPLE) [6], [7] method to (a) implement automatic resource discovery and selection; and (b) the autonomic properties [5]: self-configuration, self-healing, and context-awareness. Software product line engineering (SPLE) is a strategy to design a *family of related products* with variations in features, and with a common architecture [6], [7]. A feature means a user requirement or a visible system's function [8]. Hence, software product line engineering (SPLE) helps on developing a platform and to use mass customization to create a group of similar products that differ from each other in some specific characteristics. These characteristics are called *variation points* and their possible values are known as *variants* [7]. Thus, following a declarative strategy, in *Dohko*, the users specify their applications and requirements, and it automatically: (a) selects the resources that meet the users' needs; (b) configures the whole computing environments on the clouds; (c) handles resource failures; and (d) executes the applications on the clouds. Additionally, our feature based method enables *Dohko* to work with the configuration management tools, including Ansible (*ansible.com*) and Docker (*docker.com*). In this case, *Dohko* generates the deployment scripts and use the corresponding tool to execute them on the computing environment. We executed a genomics application to compare up to 24 biological sequences with the UniProtKB/Swiss-Prot database in two different cloud providers — Amazon EC2 and Google Compute Engine (GCE) —, considering a single and an inter-cloud scenario. Experimental results show that our system could transparently manage various clouds and configure the computing environments, requiring minimal human intervention. Moreover, by employing a hierarchical peer-to-peer (P2P) overlay to organize the nodes, our system could also handle failures and organize the nodes in a way that reduced inter-cloud communication. In particular, *Dohko* tackles the lack of middleware prototypes that can support various scenarios when executing services across multiple IaaS clouds. Moreover, it met some functional requirements identified for inter-cloud unaware systems [9] such as: (a) it can aggregate services from different clouds; (b) it provides a homogeneous interface to access the services from the clouds; (c) it implements automatic resource selection on the clouds; (d) it can deploy its components on multiple clouds; (e) it provides automatic procedures for deployments; (f) it utilizes an overlay network to connect and to organize the resources distributed across the clouds; and (g) it does not impose constraint for the selected clouds.

Structure: Section II presents our SPLE method to enable resource provision and configuration in inter-cloud scenarios; i.e., to implement the self-configuration property. Section III presents the architecture of our systems and its main components, followed by a description of its autonomic properties. Then, experimental results are discussed in Section IV. Next, a comparative view of some important features of cloud systems is discussed in Section V. Finally, Section VI concludes this papers and presents envisioned future works.

II. Software Product Line Engineering Method

For a reference example, consider two different users' profiles: one profile comprises non-technical users or users who only have high-level knowledge about either system administration or cloud computing, whereas the second ones comprises specialized users (e.g., system administrators). While users belonging to the former group might be interested in creating their computing environment based on higher-level descriptions such as CPU, memory size, and operating system, the latter may want to create the environment (or at least to have the option to create it) based on fine-grained options such as hypervisor, virtualization type, storage technologies, among others.

To meet the objectives of these two different users' profiles, we propose a software product line engineering (SPLE) method. Our method (Figure 2) comprises two phases named *domain engineering* and *product engineering*. The former comprises the activities that are performed at design time to develop reusable artifacts. In other words, the *domain engineering* phase describes the commonality and variability in the cloud domain, whereas the *product engineering* phase comprises the activities that bind the variabilities according to the users goals. The overall process fulfills with the domain and application engineering refinement proposed by [6] and [7].

In this case, the proposed method targets infrastructure-as-a-service (IaaS) inter-clouds, and it relies on *abstract* and *concrete* feature models and on a *configuration knowledge (CK)* [10]. The *abstract feature model* provides a homogeneous and provider-independent resources' description, thus, enabling inter-cloud resource discovery.

A *feature model (FM)* [8], [10] consists of a tree and constraints. In the tree, each node represents a feature of a solution. Relationships between a parent (or compound) feature and its child features (i.e., sub-features) are categorized as: *mandatory*, *optional*, *or* (at least one-child feature must be selected when its parent feature is), and *alternative* (exactly one-child feature must be selected) [10]. Besides these relationships, constraints can also be specified using propositional logic to express dependencies among the features. Features can be classified as *abstract* or *concrete* [11]. On the one hand, *abstract features* are used to structure future models, and they do not have any instance as they represent domain decisions [11]. On the other hand, *concrete features* represent configuration options.

The *concrete feature models*, on the other hand, are partial instances [12] of the *abstract model* representing clouds' configuration options. In this case, they enable resource selection at each cloud, which tackles the challenge of resource selecting on heterogeneous clouds. The configuration knowledge [10] comprises the data specifying how the products are instantiated, which helps on implementing automatic configuration. Hence, *in our method, a product is a computing environment running on cloud infrastructures that meets users' and cloud configurations' constraints, including software package dependencies*.

Furthermore, the proposed method employs an off-the-shelf constraint satisfaction problem (CSP) solver to create a computing environment using resources distributed across various clouds. Thus, based on the *abstract feature model*, the solver selects: (i) the concrete models (i.e., the clouds) matching an environment description; (ii) the resources (e.g., VMs) at each model to instantiate, taking into account their dependencies; and (iii) the assets on the configuration knowledge.

Additionally, the method considers three distinct roles: (a) *domain engineers*, (b) *system engineers*, and (c) *cloud users*. *Domain engineers* are cloud experts responsible for creating the *abstract model*. *System engineers* are *domain engineers* with *system administration* and *software architecture* skills. They are responsible for creating the *concrete models* and for defining the *configuration knowledge*. In other words, *system engineers* use the *abstract feature model* to bind the variation points of the clouds. Finally, *cloud users* are all people interested in using the cloud infrastructure.

In this context, when the users need to use the clouds, they submit their requirements, employing the terms defined in the *abstract model*. Then, our system looks for the *concrete models* that meet the users needs. Next, having the *concrete models*, the system creates the products, executing the actions described in the *configuration knowledge*. In this case, a computing environment is always a product of a model.

As a feature may have many constraints, handling it manually is usually time-consuming and error-prone. To tackle these issues, the *configuration knowledge (CK)* contains the steps defining how each feature is instantiated, its requirements, and its post-conditions. In practice, this means that the *CK* makes a mapping between the features and the artifacts implementing them. For instance, when a VM is started, it must be contextualized, as depicted in Figure 3, in order to: (i) communicate with the other virtual machines running on the same cloud and on the other clouds; (ii) have the required software packages and security rules up-to-date; and (iii) allow the users to connect to any VM, without having to manually importing their public keys (e.g., SSH keys) in each cloud and VM.

III. Design of the Proposed Solution

In our proposal, we assume an inter-cloud environment, where each cloud has a public application programming interface (API) that implements the operations to: (a) manage the virtual machines (e.g., to create, to start, to shutdown) and their disks; (b) get the list of the available VMs and their state (e.g., running, stopped, terminated, among others); (c) complement the VMs' descriptions through the usage of metadata. We also consider that the virtual machines fail following the fail-stop model. In other words, when a VM fails, its state changes to stopped and the system can either restart it or replace it by a new one.

As *Dohko* relies on a P2P overlay to organize the nodes, we assume that there is a set of nodes responsible for executing some specialized functions such as system bootstrapping and communication management. It is important to notice that the presence of such nodes does not mean a centralized control [13]. In addition, the number of these specialized nodes may vary according to some objective, e.g., the number of clouds.

Dohko uses a message queue system (MQS) to implement a publish/subscribe policy. The publish/subscribe interaction pattern is employed due to its properties such as [14]: space decoupling, time decoupling, and synchronization decoupling. These properties help the system to increase scalability and to make its communication infrastructure ready to work on distributed environments. Moreover, message queue systems normally do not impose any constraint for the infrastructure. In this case, all messages are persisted to support both node's and services' failures; and a message continues in the queue until a subscriber consumes it or until a defined timeout is achieved.

Dohko's architecture comprises three layers: *client*, *core*, and *infrastructure*, as depicted in Figure 4. There is also a cross-layer called *monitoring*. In the context of this work, the rationale for using a layered architecture is that it enables a loosely coupled design and a service-oriented architecture (SOA) strategy.

A. Client Layer

The *client* layer provides the *job submission* module. This module takes an *application descriptor* as input, and submits it to a node in the cloud. This node is called *application manager*, and it will coordinate the execution of the applications across the clouds.

An *application descriptor* has five parts (Listing 1): user, requirements, clouds, applications, and on-finished action. The user section contains the user's information such as user name and his/her access keys. If the user does not have the access keys, they will be generated by the system. These keys are used to create and to connect to the virtual machines. The requirements section, on the other hand, includes: (a) the maximal cost to pay for a VM per hour; (b) the minimal number of CPU cores and RAM memory size; (c) the operating system; and (d) the number of virtual machines to be created in each cloud. These requirements are used to select the instance types based on the software product line engineering (SPLE) method (Section II). The clouds section comprises the data of the user in each cloud provider. These data consist of the access and the secret key required to invoke the clouds' operations, and they are given to users by the cloud providers. This section also contains informations such as region and instance types. However, these parameters target advanced users (e.g., system administrators) who may already have the regions and/or the instance types to be used. The applications section describes the tasks to be executed including their inputs and outputs. Finally, the on-finished (Line 42 in Listing 1) part instructs the system about what to do after the applications have finished. The options are: *NONE*, *FINISH*, and *TERMINATE*. The *FINISH* option shuts down the virtual machines, which means that their persistent disks continue in the clouds; whereas the *TERMINATE* one shuts down and deletes the virtual machines.

```
 1 ---
 2 name:

 4 user:
 5   username:
 6   keys:
 7     key: []

 9 requirements:
10   cpu:
11   memory:
12   platform:
13   cost:
14   number-of-instances-per-cloud:

16 clouds:
17   cloud:
18   - name:
19     provider:
20       name:
21     access-key:
22       access-key:
23       secret-key:
24     region:
25     - name:
26       zone:
27       - name:
28     instance-types:
29       instance-type:
30       - name:
31     number-of-instances:

33 applications:
34   application:
35     name:
36     command-line:
37     file:
38     - name:
39       path:
40       generated:

42 on-finished:
```

Listing 1: Structure of an application descriptor

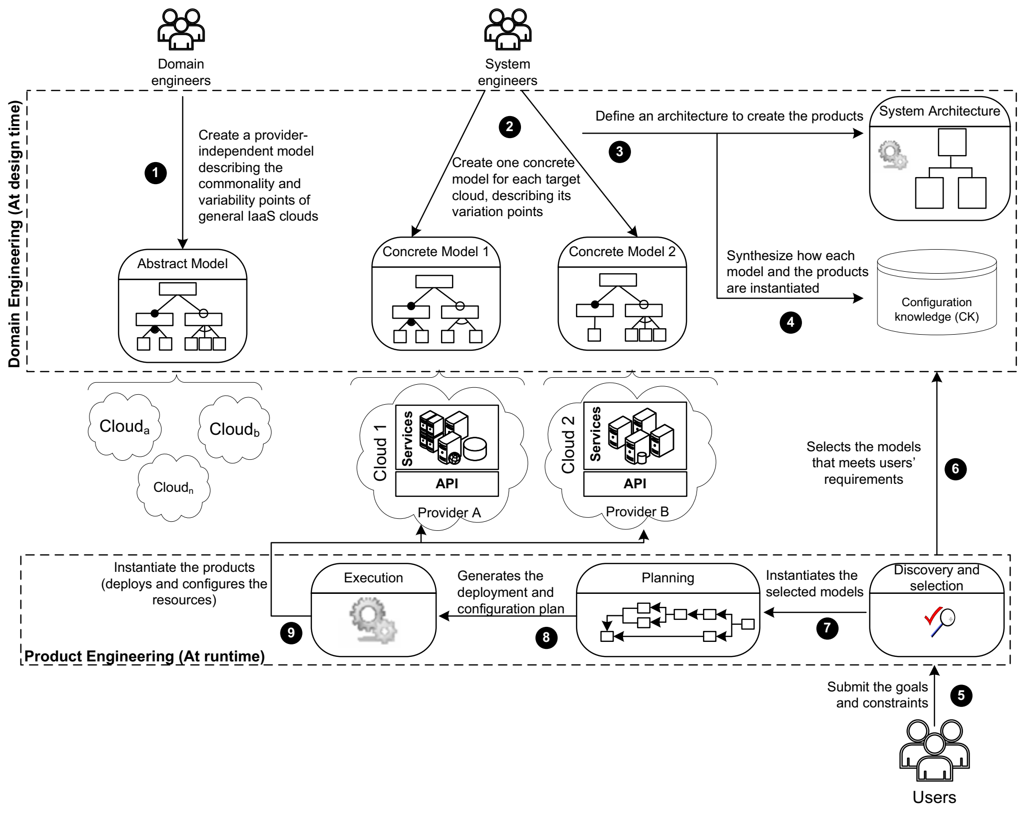

Domain engineers
System engineers
Domain Engineering (At design time)
1
Create a provider-independent model describing the commonality and variability points of general IaaS clouds
2
Create one concrete model for each target cloud, describing its variation points
3
Define an architecture to create the products
System Architecture
Abstract Model
Concrete Model 1
Concrete Model 2
4
Synthesize how each model and the products are instantiated
Configuration knowledge (CK)
Clouda
Cloudb
Cloudn
Cloud 1
Services
API
Provider A
Cloud 2
Services
API
Provider B
6
Selects the models that meets users' requirements
Instantiate the products (deploys and configures the resources)
Execution
9
Generates the deployment and configuration plan
8
Planning
Instantiates the selected models
7
Discovery and selection
Product Engineering (At runtime)
Submit the goals and constraints
5
Users

Fig. 2: A feature-based engineering method to deal with automatic resource selection and configuration in inter-cloud environments

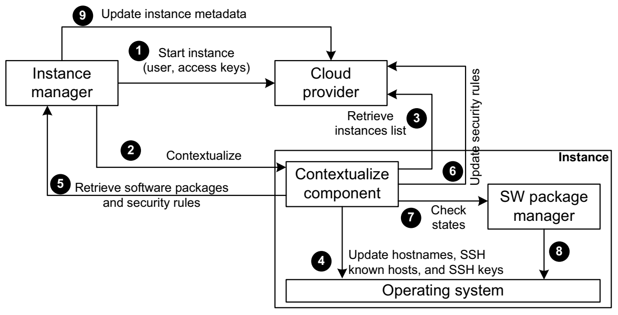

9 Update instance metadata
Instance manager
1 Start instance (user, access keys)
Cloud provider
Retrieve instances list 3
Update security rules
2 Contextualize
Instance
Contextualize component
6
5 Retrieve software packages and security rules
SW package manager
7 Check states
4 Update hostnames, SSH known hosts, and SSH keys
8
Operating system

Fig. 3: The contextualize process executed after a virtual machine is provisioned

B. Core Layer

The *core* layer of the architecture (Figure 4) comprises the modules to provision, to create, to configure, to implement group communication, and to manage data distribution, as well as to execute the applications.

The *provisioning* module is responsible for receiving an *application descriptor* and for generating a *deployment descriptor*. It utilizes the *feature engine* to obtain the instance types that meet the users' requirements. The *feature engine* implements the feature models [15]. The *deployment descriptor* comprises all data required to create a virtual machine. These data are: (a) the cloud provider; (b) the instance type; (c) the zone (data center); (d) the virtual machine image; (e) the network security group and its inbound and outbound traffic rules; (f) the disks; and (g) the metadata. Listing 2 shows an example of a *deployment descriptor*. In this example, one virtual machine should

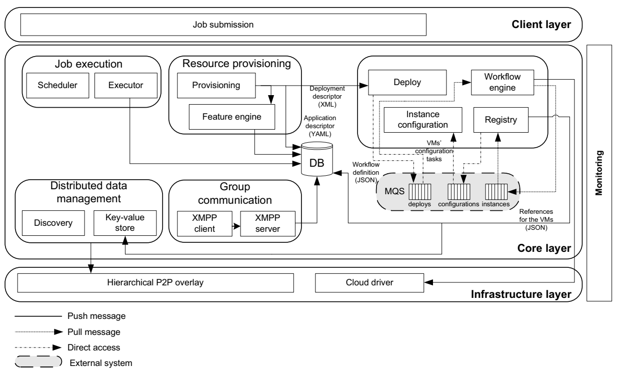


Fig. 4: Design and main modules of the autonomic system to deal with inter-cloud environments

be created with the name *e8e4b9711d36f4fb9814fa49b74f1b724-1* in the zone *us-east-1a* of region *us-east-1*. The cloud provider should be Amazon, using the instance type *t2.micro* and the virtual machine image *ami-864d84ee*. Furthermore, two metadata (i.e., tags) are added to the virtual machine.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<deployment user="username">
  <uuid>f4070433-a551-4a60-ba57-3e771ceb145f</uuid>
  <node name="e8e4b9711d36f4fb9814fa49b74f1b724-1"
    count="1" region="us-east-1"
      zone="us-east-1a">
    <provider name="amazon">
      <image>ami-864d84ee</image>
      <instance-type>t2.micro</instance-type>
    </provider>
    <tags>
```

```
        <tag>
           <name>app-deployment-id</name>
           <value>e8e4b9711d36f4fb9814fa49b74f1b724</value>
        </tag>
        <tag>
           <name>manager</name>
           <value>i-a1f8798c</value>
        </tag>
     </tags>
  </node>
</deployment>
```

Listing 2: An example of a deployment descriptor generated by the provisioning module

The *deployment descriptor* is sent to the *deploy* module. Then, the *deploy* module creates a workflow (*deployment workflow*) with the tasks to instantiate the VMs. A workflow is used since there are some precedent steps that must be performed in order to guarantee the creation of the VMs. For example, (a) the SSH keys must be generated and imported into the clouds; (b) if the instance types support the *group* feature, the system must check if one exists in the given zone and if not, create it. Furthermore, using a workflow enables the system to support partial failures of the deployment process. In addition, it decouples the architecture from the clouds' drivers. In this case, based on the provider's name, the *deploy* module selects in the database its correspondent driver to be instantiated at runtime by the *workflow engine*. This *deployment workflow* is enqueued by the *deploy* module and dequeued by the *workflow engine*. The *workflow engine* executes the workflow and enqueues the references to the created virtual machines in another queue.

The *registry* module is responsible for: (a) storing the instances informations in a local database and in the distributed key-value store; and (b) selecting the configurations (i.e., software packages) to be applied in each instance. These configurations are usually defined by the *system engineers*, and they represent a set of scripts. Each script may have some binding points. The binding points are informations only known at runtime such as the location, the name, the addresses (e.g., internal and external), among others, that must be set by the system. Listing 3 illustrates one script with three binding points. A binding point is defined between ${[and]}. In this example, the script is exporting the region's name of a node, its endpoint, and its zone's name.

```
...
export NODE_REGION_NAME=${[location.region.name]}
export NODE_REGION_ENDPOINT=${[location.region.endpoint]}
export NODE_ZONE_NAME=${[location.name]}
...
```

Listing 3: Example of one script with three binding points

The configuration tasks are put in a queue to be executed by the *instance configuration* module. The *instance configuration* module connects to the virtual machine via SSH and executes all the scripts, installing the software packages. This process guarantees that all instances have the required software packages.

The *group communication* module uses the Extensible Messaging and Presence Protocol (XMPP) (*xmpp.org*) for instant communication among the nodes in a cloud. It has an *XMPP server* and an *XMPP client*. The architecture uses the XMPP since some other group communication techniques such as broadcast or multicast are often disabled by the cloud provider. In addition, it supports the *monitoring* layer through its presence states (e.g., available, off-line, busy, among others).

The *job execution* module comprises a *scheduler* and an *executor*. They are responsible for implementing a task allocation policy and for executing the tasks. The task allocation policy determines in which node a task will be executed, considering that there are no temporal precedence relations among the tasks [16]. The goal of a task allocation/scheduling problem may be: (a) minimize the execution time of each task; (b) minimize the execution time of the whole application; (c) maximize the throughput (number of tasks completed per period of time). On its generic formulation, this problem has been proved NP-complete [17]. For this reason, several heuristics have been proposed to solve it. By default, this architecture provides an implementation of a simple task allocation policy (i.e., self-scheduling (SS) [18]). However, other task allocation policies can be implemented and added to our system, i.e., this architecture does not assume any specific task allocation policy.

The self-scheduling (SS) policy assumes a master/slave organization, where there is a master node responsible for allocating the tasks, and several slaves nodes that execute the tasks. Moreover, it considers that very few information is available about the execution time of the tasks and the computing power of the nodes. In this case, it distributes the tasks, one by one, as they are required by the slave nodes. Thus, each node always receives one task, executes it, and, when the execution finishes, asks for more task [18].

In our system, the node that receives the *application descriptor* becomes the application master node, and it is the responsible for creating and coordinating the configuration of the other nodes.

The *distributed data management* module provides a *resource discovery* and a distributed *key-value store* service. These services are based on a *hierarchical P2P* [19], [20] overlay available at the *infrastructure* layer (Section III-C). Table I shows the operations implemented by the *key-value store*. Basically, it providers three operations: (i) insert, (ii) retrieve, and (iii) remove. The insert operation takes a key and a value, and stores the value under the given key. In case there exists a value with the same key, all of them they are stored. In other words, the value of a key may be a set of objects. The retrieve method receives a key and returns its values. Finally, the remove method can remove all values of a given key or only one specific value.

C. Infrastructure Layer

The *infrastructure* layer consists of the *hierarchical P2P* overlay and the *cloud drivers*. The *hierarchical P2P* overlay is used to connect the clouds and their nodes. In this case, in a cloud with n nodes, where $n > 0$, $n - 1$ nodes (i.e., leaf-nodes) join an internal overlay network and one node (i.e., super-peer) joins the external overlay network. In other words, there is one overlay network connecting the clouds and another overlay network in each cloud connecting its nodes. The super-peer and its leaf-nodes communicate through a HTTP service, and the leaf-nodes monitor the super-peer via XMPP. Both overlays are implemented using the Chord [21] protocol. Figure 5 illustrates the hierarchical P2P overlays connecting two clouds, each one with four nodes.

When a node n running in cloud c starts, it asks the bootstrapping node through a HTTP service, the super-peers of cloud c. If node n is the first peer of cloud c, it joins the super-peers overlay by the bootstrapping node. Otherwise, it demands the super-peer, its leaf-nodes and joins the leaf-nodes overlay network or creates a overlay network. After has joined one overlay network, the node stores in the key-value store its information under the keys: $/c/n$, if it is the super-peer or $/c/ <$ super-peer's id $>$ /members, otherwise.

TABLE I: Main operations implemented by the key-value store

Operation	Description
void **insert** (key, value)	inserts the value into the network with the given key. If two or more values exist with the same key, all of them are stored
Set<Value> **retrieve** (key)	returns all the values with the given key
remove (key)	removes all the values stored under the given key
void **remove** (key, value)	removes the value stored under the given key

Leaf-nodes of different clouds communicate through their super-peers. In this case, when a leaf-node wants to communicate with a node outside its cloud, it sends a message for its super-peer that first connects to the super-peer of such node, and next forwards the message for it.

When a node leaves one cloud, it notifies its super-peer that removes the information about the node from the system.

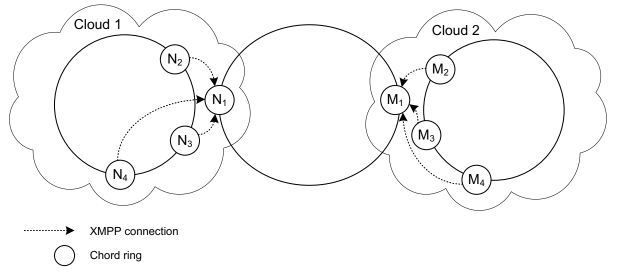


Fig. 5: Structure of the hierarchical P2P overlay connecting two clouds

D. Monitoring Cross-Layer

The *monitoring* layer is responsible for (a) checking if there are virtual machines that were not configured; (b) detecting and restarting failure nodes; (c) keeping up-to-date the information about the super-peers and the leaf-nodes in the system.

E. Autonomic Properties

This section presents how our system implements the following autonomic properties: self-configuration, self-healing, and context-awareness. Although there exists other autonomic properties, as described in [5], [22], our system implements only these properties since our focus is to help the users on tackling the difficulties of deploying and executing applications across multiple clouds, taking into account different objectives, and without requiring cloud and system administration skills from the users. Figure 6 shows the autonomic control loop implemented by this architecture.

The process follows a declarative strategy. A declarative strategy allows the users to concentrate on their objectives rather than on dealing with cloud or system administration issues. In this case, the process starts with the users describing their applications and constraints. Then, using a self-configuration process, the system (a) creates and configures the whole computing environment taking into account the characteristics and the state of the environment, i.e., the availability of other virtual machines; (b) monitors the availability and state of the nodes through the self-healing; (c) connects the nodes taking into account their location. In other words, using a hierarchical organization, nodes in the same cloud joins an internal overlay network and one of them joins an external overlay network, connecting the clouds (Figure 5). Nodes in the internal overlay network use internal IP addresses for communication, which often has zero cost and a network throughput higher than if they were using external IP addresses; finally (d) executes the applications.

In the next sections, we will explain each one of these properties in detail.

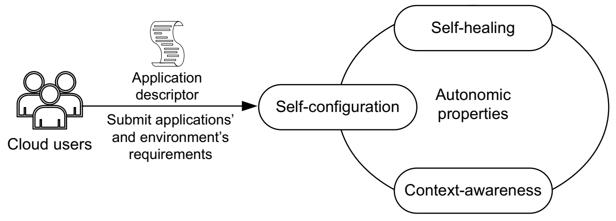
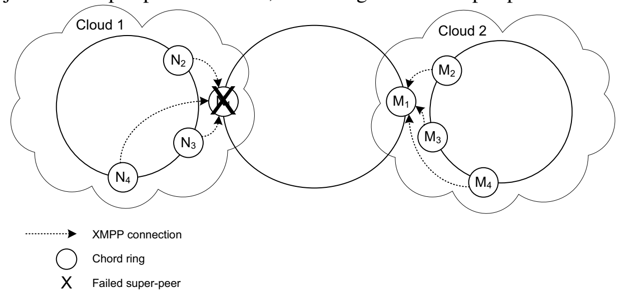


Fig. 6: The autonomic properties implemented by our architecture

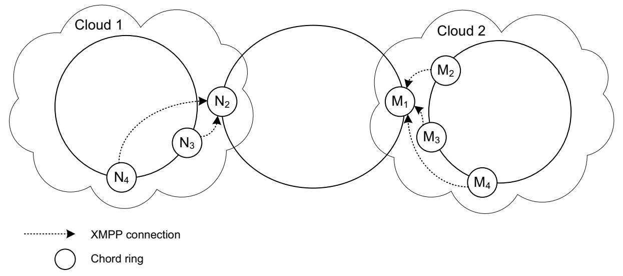
1) Self-Configuration: Our system automatically creates and configures the virtual machines in the clouds. The configuration process is based on the *concrete feature* model [15] defined by the *system engineers* (Figure 2) and on the users requirements. When the system receives an *application descriptor*, it (a) creates the access keys (i.e., the private and the public keys) to be used by the virtual machines, and imports them into the clouds; (b) creates a security group with the inbound and outbound rules; (c) uses the clouds' metadata support to describe the VMs, allowing the users to trace the origin of each resource, and to access them without using the architecture, if necessary. In addition, these metadata provide support for the self-healing process; (d) selects one availability zone to deploy the virtual machines according to the availability of other VMs running in the same zone; (e) selects the instance types; (f) configures the instances with all the software packages; (g) starts all the required services considering the instance state. For example, if a service requires one information (e.g., its region's name, up-to-date IP addresses) about the environment where it is running, it is automatically assigned by the architecture before it starts. For instance, when an instance fails or needs to be restarted, the architecture detects the new values of its ephemeral properties such as the internal and external addresses, and starts its services to use up-to-date informations.

The metadata of an instance include: (i) the user name to access the VM, (ii) the value of the keys, (iii) the name of its virtual machine image (VMI), (iv) the owner (i.e., the user), (v) the name of its application manager, and (vi) the name of the feature model used to configure it. These data help the system to support failures of both the manager and the architecture. For example, suppose that just after having created the virtual machines, the application manager node fails. Without these metadata, another node could not access the instances to configure them, leaving for the system only the option to terminate the instances, which implies a financial cost as the users will pay for the instances without having used them.

2) Self-Healing: The cross-layer *monitor* module uses the XMPP to monitor the availability of the virtual machines. In this case, every virtual machine runs an XMPP server, which periodically, requests the *discovery* service to list the VMs running in the cloud. The returned VMs are included as the monitor contact. In this context, each VM utilizes the XMPP's status to report to other one its status and also to know their status. When a VM's status changes to off-line, its application manager waits a determined time and requests the cloud provider to restart it. The restarting avoid unresponsive node due to its workload state (i.e., overloaded VM) or some network issues. If the node's status does not change to on-line, the manager terminates it, and creates a new one. In case of the super-peer's failures (Figure 7(a)), the leaf-node with the higher uptime tries to recover the failed super-peer. If it is impossible, this leaf-node leaves its network (Figure 7(b)) and joins the super-peers network, becoming the new super-peer.



(a) The cloud's 1 super-peer (N_1) failed disconnecting the clouds 1 and 2



(b) The leaf-node N_2 leaves its overlay network and joins the new super-peer overlay network, connecting the clouds 1 and 2

Fig. 7: Example of super-peer failure and definition of a new super-peer

Since the application manager may completely fail during the configuration of the virtual machines, the *monitor* checks if there are instances created by the architecture without have been completely configured. If such instances do exist, it contacts their application manager or sends them for other node to continue its configuration.

3) Context-Awareness: Similar to [23] and [24], by context we refer to the topology of the overlay network or peers association, which may impact the overall system's performance (e.g., throughput and latency), as well as in the financial cost (e.g., network communication cost). In other words, in the context of this work, context-awareness means the capacity of the P2P communication protocol be aware of the peers' locations and of adapting the system's behavior based on the situation changes [23], [25].

As described in Section III, the nodes are organized into two overlay networks. This avoids unnecessary data transfers between the clouds and often increases the network throughput between the nodes in the internal overlay. Furthermore, it helps to decrease the cost of the application execution, avoiding inter-cloud data transferring due to the overlay stabilization activities [21]. If all nodes were organized in the same P2P overlay network, they would have to communicate using external IP addresses (i.e., public IP), which implies in Internet traffic cost, even if the nodes are located in the same data center. In addition, in a high churn rate scenario [26], it decreases the cost of maintaining the distributed hash tables (DHTs) up-to-date since it does not require inter-cloud communication.

F. Using Dohko*to Execute a Cloud-Unaware Application*

In order to illustrate the usage of our system, consider that one user is interested in executing his/her application in the cloud (Figure 8). In this example, the application manager is the node that receives the user's demand (*application descriptor*), and the worker is a node that receives a task to execute.

The process starts when the user defines an *application descriptor* depicted in Listing 4, with the requirements and applications. In this example, one virtual machine with at least 1 CPU core and 1 GB of RAM memory is requested, with the Linux operating system, and a cost of at most 0.02 USD/hour. Moreover, this instance should be created on Amazon using the given access and secret key. Finally, the task consists of getting the information about the CPU (Line 25 in Listing 4).

The *application descriptor* is submitted to the system through the *job submission* module (Figure 8 (1)). Then, the *job submission* module looks for an appropriate node through the *discovery* service (Figure 8 (2)), and sends the *application descriptor* to it (Figure 8 (3)).

After, the *provisioning* module in the application manager takes the *application descriptor* and persists into its database (Figure 8 (4)). Next, the *provisioning* module demands the *feature engine* module: (a) an instance type that meets the user's constraints, and (b) a virtual machine image (VMI). The *feature engine* returns the *t2.micro* instance type in the *us-east-1* region, and the virtual machine image *ami-864d84ee*. With these data, the *provisioning* module: (a) selects a zone to host the virtual machine, (b) generates the *deployment descriptor* (Listing 2), and (c) submits it to the *deployment* module (Figure 8 (5)).

The *deployment* module creates a workflow (Figure 9) with the steps to instantiate the VM, and enqueues it through the MQS (Figure 8 (6), queue: deploys). The *workflow engine* executes the deployment workflow, i.e., it connects to the cloud and creates the virtual machine (Figure 8 (7 and 8)). The data about the VM are enqueued in *instances* queue (Figure 8 (9)). Next, the *registry* module dequeues the instance from the queue *instances* and inserts its information into the database and into the *key-value store* (Figure 8 (10)). After, it creates the configuration tasks to be executed in the virtual machine (Figure 8 (11)). Each task comprises a host, a user name, the SSH keys, and the scripts to be executed. After, the *instance configuration* module: (a) connects to the node via SSH; (b) executes the scripts; and (c) starts an instance of this architecture (Figure 8 (12)). Then, the virtual machine, which is now executing the architecture, notifies the application manager using the *group communication* module (Figure 8 (13)). Finally, the manager uses the *scheduler* (Figure 8 (14)) to distribute the tasks, according to the self-scheduling (SS) task allocation policy (i.e., self-scheduling (SS) [18]). In other words, the *scheduler* sends a task to the worker, that should execute it and return its result to the application manager. The whole process is monitored by the *monitoring* module.

```
1 ---
2 name: "example"
3 user:
4   username: "user"
5 requirements:
6   cpu: 1
7   memory: 1
8   platform: "LINUX"
9   cost: 0.02
10  number-of-instances-per-cloud: 1
```

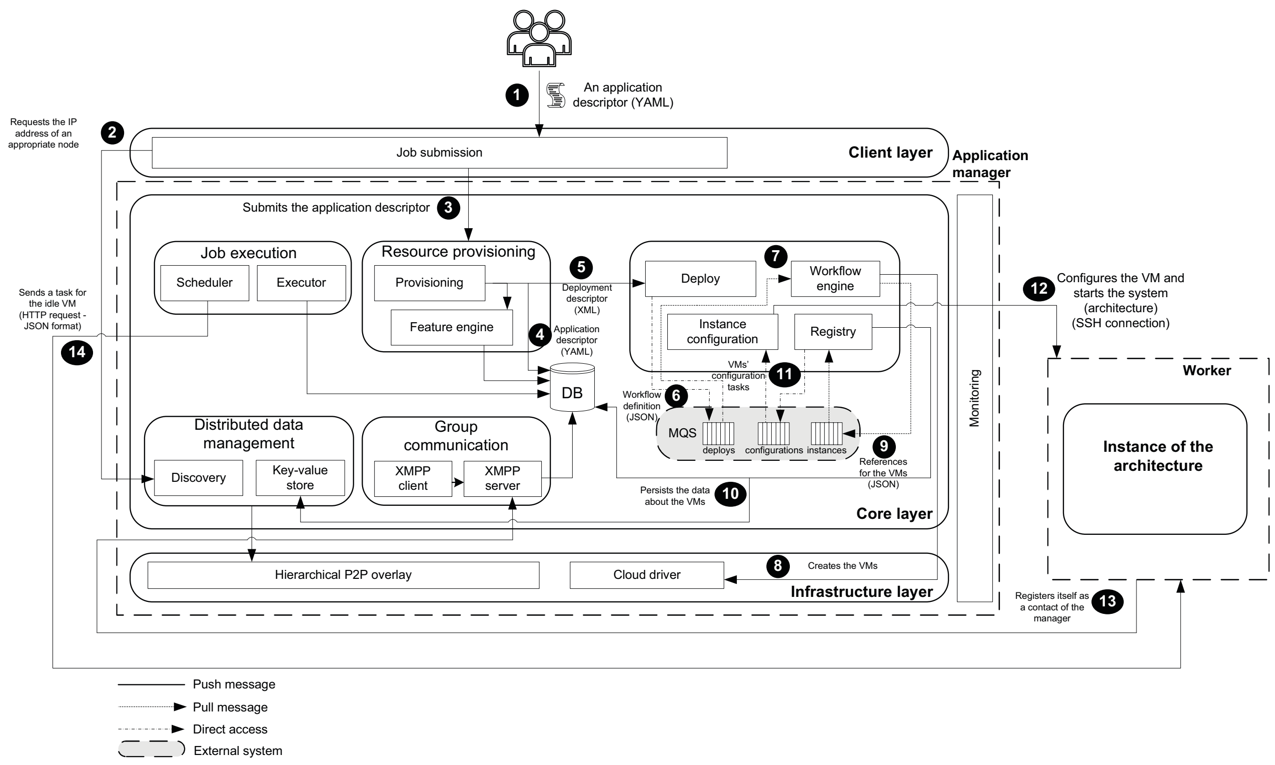

An application descriptor (YAML)
1
Requests the IP address of an appropriate node
2
Job submission
Client layer
Application manager
Submits the application descriptor
3
Job execution
Scheduler
Executor
Resource provisioning
Provisioning
Feature engine
5
Deployment descriptor (XML)
4
Application descriptor (YAML)
DB
Deploy
7
Workflow engine
Instance configuration
Registry
VMs' configuration tasks
11
6
Workflow definition (JSON)
MQS
deploys
configurations
instances
9
References for the VMs (JSON)
Persists the data about the VMs
10
Distributed data management
Discovery
Key-value store
Group communication
XMPP client
XMPP server
Core layer
Monitoring
12
Configures the VM and starts the system (architecture) (SSH connection)
Worker
Instance of the architecture
Sends a task for the idle VM (HTTP request - JSON format)
14
Hierarchical P2P overlay
Cloud driver
8
Creates the VMs
Infrastructure layer
Registers itself as a contact of the manager
13
Push message
Pull message
Direct access
External system

Fig. 8: Interaction between the architecture's module when submitted an application to execute

```
clouds:
 cloud:
 - name: "ec2"
   provider:
     name: "amazon"
   access-key:
     access-key: "65AA31A0E92741A2"
     secret-key: "619770ECE1D5492886D80B44E3AA2970"
   region: []
   instance-types:
     instance-type: []
applications:
 application:
   name: "cpuinfo"
   command-line: "cat /proc/cpuinfo"
on-finished: "NONE"
```

Listing 4: Application descriptor with the requirements and one application to be executed in one cloud

IV. Evaluation

A. Experimental Setup

The architecture was implemented in Java 7 and it used the RabbitMQ as the MQS. The Linux distributions: Debian and Ubuntu were used by the nodes in the clouds. In this case, the Debian was used in the experiments executed in GCE and Ubuntu in the ones executed in Elastic Compute Cloud (EC2). Table II presents the setup of the application. The whole source code is available at dohko.io.

In order to evaluate our architecture, we used the self-scheduling (SS) task allocation policy [18] to execute parameter sweep applications. A parameter sweep application is defined as a set $T = \{t_1, t_2, \ldots, t_m\}$ of m independent tasks. In this context, independence

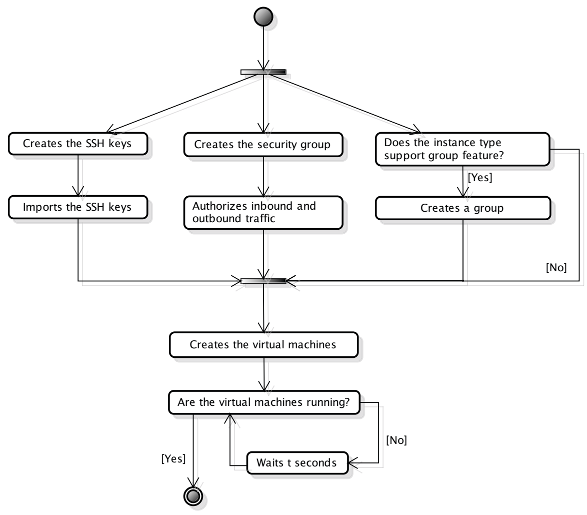

Creates the SSH keys
Creates the security group
Does the instance type support group feature?
[Yes]
Imports the SSH keys
Authorizes inbound and outbound traffic
Creates a group
[No]
Creates the virtual machines
Are the virtual machines running?
[No]
Waits t seconds
[Yes]

Fig. 9: Workflow to create one virtual machine in the cloud

means that there is neither communication nor temporal precedence relations among the tasks. Besides that, all the m tasks execute exactly the same program changing only the input of each task [27].

We used the SSEARCH application retrieved from *www.ebi.ac.uk/Tools/sss* as the parameter sweep application. In our tests, the SSEARCH application compared 24 sequences with the database UniProtKB/Swiss-Prot (September 2014), available

TABLE II: Setup of the experimental environment

Software	Version
GNU/Linux x86_64	3.2.0-4-amd64
GNU/Linux x86_64	3.13.0-29-generic
OpenJDK	1.7.0_65
RabbitMQ	3.3.5
SSEARCH	36.3.6

at *uniprot.org/downloads*, composed of $546,238$ sequences. The query sequences are the same presented in [28]. For each sequence, we defined an entry in the *application descriptor*, i.e., each sequence represents a task. Listing 5 illustrates the definition (i.e., command line) of one task.

In this execution, we requested at least 2 cores and 6 GB of RAM memory, the Linux operating system, and a cost of at most 0.2 USD/hour. We asked to execute the task *ssearch36* 24 times. Each *ssearch36* task receives as input one query sequence ($HOME/sequences/O60341.fasta in Listing 5) and one database (*$HOME/uniprot_sprot.fasta*). Its output should be stored in *$HOME/scores/O60341_scores.txt*.

We evaluated our architecture considering a multiple and a single cloud scenarios, and different users' requirements. The cloud providers were Google Compute Engine (GCE) and Elastic Compute Cloud (EC2). Table III presents the users' constraints and Table III the instance types that were selected based on these requirements.

Each experiment was repeated three times, and the mean was taken.

```
---
name: "ssearch-app"
user:
  key: []
  username: "user"
requirements:
  cpu: 2
  memory: 6
  platform: "LINUX"
  cost: 0.2
clouds:
  cloud: []
applications:
  application:
    name: "ssearch36"
    command-line: "ssearch36 -d 0 ${query} ${database} >>
        ${score_table}"
    file:
    - name: "query"
      path: "$HOME/sequences/O60341.fasta"
      generated: "N"
    - name: "database"
      path: "$HOME/uniprot_sprot.fasta"
      generated: "N"
    - name: "score_table"
      path: "$HOME/scores/O60341_scores.txt"
      generated: "Y"
  ...
on-finished: "TERMINATE"
```

Listing 5: An application descriptor with one SSEARCH description to be executed in the cloud

B. Scenario 1: application deployment

This experiment aims to measure the deployment time, i.e., the time to create and to configure the virtual machines. The wallclock time was measured including: (a) the time to instantiate the VMs in the cloud provider; (b) the time to download and to configure all the software packages (e.g., Java, RabbitMQ, SSEARCH, among others); and (c) the time to start the architecture. By default, the architecture performs the configurations in parallel, with the number of parallel processes defined in a parameter of the architecture. In this case, the maximum of 10 configurations were done in parallel.

TABLE III: Users' requirements to deploy the SSEARCH application on the clouds

# Req.	# vCPU	RAM (GB)	Cost (USD/hour)	# VM	CP$^{\flat}$
1	2	6	0.2	5	EC2
	2	6	0.2	10	
2	2	6	0.2	5	GCE
	2	6	0.2	10	
3	4	6	1.0	5	EC2
	4	6	1.0	10	
4	4	6	1.0	5	GCE
	4	6	1.0	10	
5	4	6	1.0	5	EC2 and GCE
	4	6	1.0	10	

$^{\flat}$ cloud provider

TABLE IV: Instance types that met the users' requirements to execute the SSEARCH application

Req. #	Instance type	# vCPU	RAM (GB)	Cost (USD/hour)	Family type	CP$^{\flat}$
1	m3.large	2	7.5	0.14	General	EC2
2	n1-standard-2	2	7.5	0.14	Memory	GCE
3	c3.xlarge	4	7.5	0.21	Compute	EC2
4	n1-standard-4	4	15	0.28	General	GCE
5	c3.xlarge	4	15	0.21*	General	EC2
	n1-standard-4	4	15	0.28	General	GCE

$^{\flat}$ cloud provider
* 3 c3.xlarge and 2 n1-standard-4 virtual machines

Figure 10 presents the deployment time for the instance types listed in Table IV. As can be seen, in Amazon, increasing the number of virtual machines to deploy from 5 to 10 decreased the deployment time. This occurs because the virtual machines of each experiment are homogeneous, which enables us to request multiple instances at the same time. Similar behavior has already been observed by other works in the literature [29], [30]. On the other hand, in Google, the deployment time is proportional to the number of virtual machines.

In our experiment, the deployment time of 5/10 VMs took at most 10 minutes. With this, the applications can start across multiple VMs and multiple clouds, without requiring from users cloud and system administration skills, and also without needing the use of virtual machine image (VMI).

The 10 virtual machines of the instance type: *n1-standard-4* were deployed in a multiple cloud scenario, since GCE imposes a limit of 6 virtual machines of this type in each region. In this case, the system allocated 5 VMs in the U.S. and 5 in Europe.

C. Scenario 2: application execution

Figure 11 presents the wallclock execution time for the four standalone experiments (i.e., requirements 1 to 4 of Table III), and Table V their total financial cost. We can see that the instances that belong to the same family type have almost the same performance. The lower execution time (89 seconds) was achieved by the instance type *c3.xlarge* (40 vCPU cores) with a cost of 2.10 USD. Considering that the application does not take a long time to finish neither it demands many computing resources, this represents a high cost. If the users wait 33% more (43 seconds), they can pay 50% less (1.05 USD).

In Figure 12, we present the execution time for a multi-cloud scenario. In this case, for the requirement of 5 instances (i.e.,

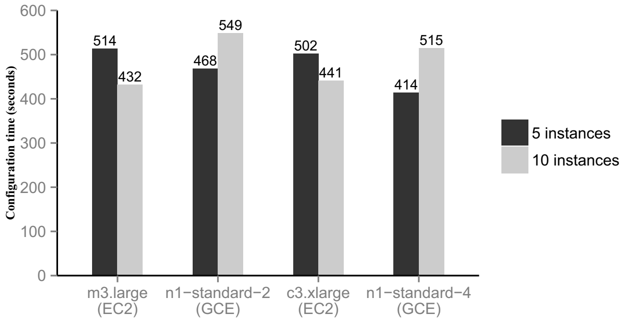


Fig. 10: Configuration time of the virtual machines on the clouds

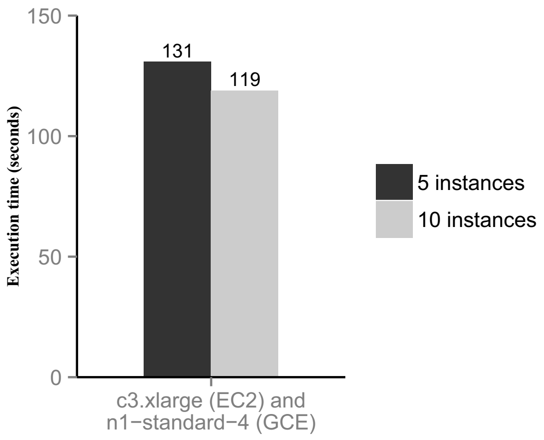


Fig. 12: Execution time of the SSEARCH application to compare 24 genomics query sequences with the UniProtKB/Swiss-Prot database in an inter-cloud scenario

requirement 5 of Table III), 3 virtual machines were used from EC2 and 2 instances from GCE. If we compare Figure 11 with Figure 12, we can see that the execution time increased almost 34%. One reason for this difference is probably the network throughput between the clouds of different providers, since we do not observe such overhead in the scenario with ten *n1-standard-4* virtual machines distributed across two GCE's clouds.

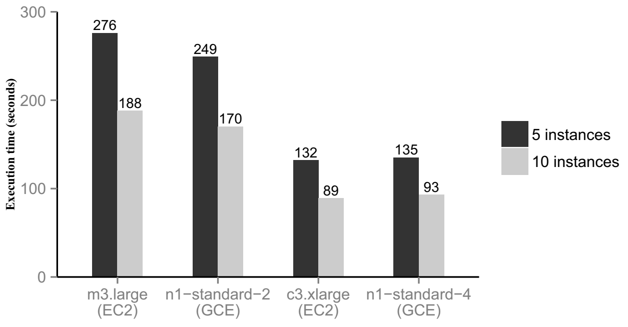


Fig. 11: SSEARCH's execution time on the clouds to compare 24 genomics query sequences with the UniProtKB/Swiss-Prot database

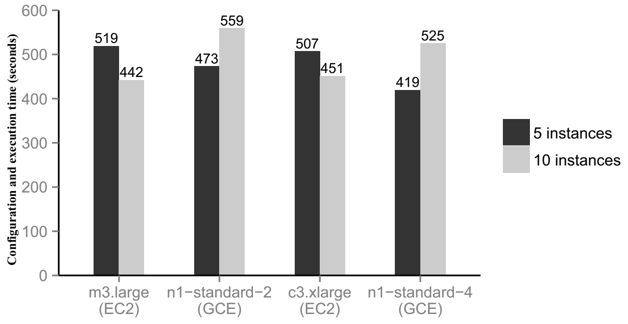


Fig. 13: Deployment and execution time of the experiments

TABLE V: Financial cost for executing the application in the cloud considering different requirements (Table III)

Instance type	# Instances	Wallclock time (seconds)	Total cost (USD)
m3.large	5	276	0.7
m3.large	10	188	1.4
n1-standard-2	5	249	0.7
n1-standard-2	10	170	1.4
c3.large	5	132	1.05
c3.large	10	89	2.10
n1-standard-4	5	135	1.4
n1-standard-4	10	93	2.94*
c3.large	3	131	0.63
n1-standard-4	2		0.53♯
c3.large	5	119	1.05
n1-standard-4	5		1.40♭

* *total cost: 2.96 (USD) (1.4 (U.S.) + 1.54 (Europe))*
♯ total cost: 1.16 USD
♭ total cost: 2.45 USD

Figure 13 presents the total time (i.e., deployment + execution time) for the experiments. In this figure, we can observe the impact of deploying 10 virtual machines has in the total execution time. In this case, the deploy time of 5 virtual machines was better than the deploy time of 10 virtual machines.

D. Scenario 3: application deployment and execution with failures

To evaluate the self-healing property of our architecture, we simulated two types of failures. In the first case, we stopped the application manager just after it had received the tasks to execute. In the second case, we executed a process that, at each minute, selected and stopped a worker. These failures were evaluated in a multi-cloud scenario (10 virtual machines – 5 *c3.xlarge* and 5 *n1-standard-4*). Figure 14 presents the execution time for each failure scenario.

The failure of the application manager has a high impact in the total execution time, however, it is low than of the workers. This is mostly occurs because worker's failures, demands a time from the application manager to detect it and to reassign the task of the failed worker to another one. Moreover, since multiple workers failed, this highly impacted the total execution time compared with both the failure-free scenario and the failure of the application manager. Moreover, when the application manager fails, the workers can still continue the execution of their tasks.

V. Related Work

Over the years, different cloud systems have been proposed in the literature, with different objectives. In Table VI, we present a comparative view of important features of these systems. The system is presented in the first column. The second column presents if the available system implements self-configuration, which means that it

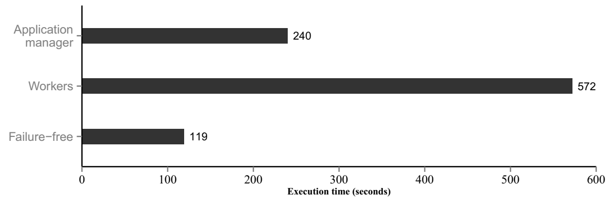


Fig. 14: Execution time of the application on the clouds with three different type of failures

can select and configure the resources automatically. The third column presents if it implements failure recovery policy. The fourth and the fifth columns show if the architectures are context-aware and if they implement self-optimization. Context-aware in this case, means if nodes are organized considering their location. The cloud model of the architecture is presented in the sixth column. Finally, the last column presents if the system can work in an inter-cloud environment.

As can be seen, half of the works implement self-configuration properties. In this case, the developers specify their needs in a service manifest, and the architecture automatically selects a cloud provider and deploys the applications. Moreover, self-healing property is implemented by some of the works. For example, Snooze [31] employs a hierarchical resource management, and uses multicast to locate the different nodes (e.g., group managers, local controllers). Similar to us, CometCloud [32] organizes the nodes using a P2P overlay to connect and to detect failures. However, it organizes the nodes unaware of their locations.

Only two architectures [32], [33] deploy the applications taking into account the location of the resources (i.e., nodes) where the application will run. For example, OPTMIS [33] deploys the applications taking into account the energy consumption of the cloud as well its carbon footprint. Another example is CometCloud [32]. CometCloud distributes the tasks considering three security domain level. In this case, there is a master that tries to send the tasks to trusted workers, i.e., workers that are running in a private cloud. Tasks are sent to untrusted workers (i.e., workers that are running in a public cloud) only when the service-level agreement (SLA) violation ratio exceeds a threshold.

Moreover, the majority of the architectures, implements self-optimization property to help the developers on meeting the quality of service (QoS) constraints of their native cloud applications [34]–[37] or to minimize power consumption [31].

Finally, only two architectures target IaaS cloud, and the other ones platform-as-a-service (PaaS) cloud.

CometCloud [32] is the closest work to ours. Our work differs from it in the following ways. First, we consider that self-configuration is an important feature to support the users on running their applications in the clouds, since most of the cloud potential users do not have cloud and/or system administration skills. Our self-configuration relies on feature models (FMs), to deal with heterogeneous cloud providers' resource descriptions, and also to enable a declarative strategy. This frees the users from the work of creating pre-configured virtual machine images on each cloud. Second, we use a hierarchical P2P overlay to organize the resources, which helps us to reduce the cost of communication between the clouds to keep the distributed hash table (DHT) up-to-date. In CometCloud, the resources are organized in the same P2P overlay, and it uses different security domains to distribute the tasks. However, this work does no implement self-optimization feature since this it requires advanced knowledge about the workload/application.

TABLE VI: Comparison of the cloud architectures considering their autonomic properties

Architecture	SC	SH	CA	SO	Cloud Model	Inter-Clouds
Cloud-TM [34]	No	Yes	No	Yes	PaaS	Yes
JSTaaS [37]	Yes	No	No	Yes	PaaS	Yes
mOSAIC [36]	Yes	No	No	Yes	PaaS	Yes
Reservoir [35]	Yes	Yes	No	Yes	PaaS	Yes
OPTIMIS [33]	Yes	No	Yes	Yes	PaaS	Yes
FraSCaTi [38]	Yes	No	No	No	PaaS	Yes
TClouds [39]	No	Yes	No	No	PaaS	Yes
COS [40]	No	No	No	Yes	PaaS	No
Snooze [31]	No	Yes	No	Yes	IaaS	No
CometCloud [32]	No	Yes	Yes	Yes	IaaS	Yes
Dohko	**Yes**	**Yes**	**Yes**	**No**	**IaaS**	**Yes**

self-configuration (SC), self-healing (SH), context-awareness (CA), self-optimization (SO)

VI. Conclusion

Using multiple IaaS clouds is still a challenging and time-consuming activity, even for experienced system administrators. The difficulties mostly exist since IaaS clouds offer low-level access to the infrastructure resources. Thus, the users are responsible for selecting, configuring and managing the computing resources on the clouds. Likewise, the clouds target web applications, whereas the users' applications are often batch-oriented systems. Hence, this can be a barrier to use clouds' services.

Therefore, in this paper, we presented and evaluated *Dohko*, an autonomic and goal-oriented cloud system. *Dohko* enables a declarative approach. In this case, the users submit to the system a corresponding specification of the computing environment configuration they want. Then, the system automatically creates and configures the whole computing environments, taking into account users' requirements. Finally, it executes the applications in one or across multiple clouds. Our framework implements the autonomic properties [5]: self-configuration, self-healing, and context-awareness. Moreover, it relies on a hierarchical P2P overlay to organize the nodes distributed across various clouds.

In our experiments, we could execute the SSEARCH application to compare up to 24 query sequences with the database UniProtKB/Swiss-Prot in two distinct cloud providers — Amazon EC2 and Google Compute Engine (GCE) — considering five execution scenarios, without requiring from the users any cloud or system administration skills. This approach helps on tackling the lack of middleware prototypes that can support different scenarios when executing applications across multiple clouds. Furthermore, since *Dohko* is essentially a self-management system, it frees the users from the details of computing system operations, allowing them to concentrate on their objectives rather than spending their time managing the computing environments.

One issue of our system is the lack of a self-optimization strategy since it does not focus on task scheduling. Furthermore, our approach does not consider pre-packaged configuration descriptions, such as Docker Hub Images (*hub.docker.com*) or Ansible Galaxy (*galaxy.ansible.com*), and we leave these extensions as our future work. The software prototype is available at *dohko.io*.

Acknowledgment

The authors would like to thank FAPDF/Brazil, Inria Saclay (project POSTALE), and MINES ParisTech for their financial support.

References

[1] N. Hayashibara and A. Cherif, "Failure detectors for large-scale distributed systems," in *21st IEEE Symposium on Reliable Distributed Systems*, 2002, pp. 404–409.

[2] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*, 5th ed. Addison-Wesley, 2011.

[3] N. Grozev and R. Buyya, "Inter-Cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience*, vol. 44, no. 3, pp. 369–390, 2014.

[4] G. I.-C. T. Forum, "Use cases and functional requirements for inter-cloud computing," Global Inter-Cloud Technology Forum, Tech. Rep., August 2010, last accessed in January 2014. [Online]. Available: gictf.jp/doc/GICTF_Whitepaper_20100809.pdf

[5] P. Horn, "Autonomic computing: IBM's perspective on the state of information technology," www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf, 2001, last accessed in January 2014.

[6] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[7] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, 2005.

[8] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, November 1990, last accessed in June 2014. [Online]. Available: www.sei.cmu.edu/reports/90tr021.pdf

[9] D. Petcu, "Consuming resources and services from multiple clouds," *Journal of Grid Computing*, pp. 1–25, 2014.

[10] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.

[11] T. Thum, C. Kastner, S. Erdweg, and N. Siegmund, "Abstract features in feature modeling," in *15th International Software Product Line Conference*, 2011, pp. 191–200.

[12] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process: Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.

[13] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Computing Surveys*, vol. 36, no. 4, pp. 335–371, 2004.

[14] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, 2003.

[15] A. F. Leite, V. Alves, G. N. Rodrigues, C. Tadonki, C. Eisenbeis, and A. C. M. A. de Melo, "Automating resource selection and configuration in inter-clouds through a software product line method," in *8th IEEE International Conference on Cloud Computing*, 2015, pp. 726–733.

[16] T. L. Casavant and J. G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 2, pp. 141–154, 1988.

[17] R. Graham, E. Lawler, J. Lenstra, and A. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of discrete Mathematics*, vol. 5, pp. 287–326, 1979.

[18] P. Tang and P.-C. Yew, "Processor self-scheduling for multiple-nested parallel loops," in *International Conference on Parallel Processing*, 1986, pp. 528–535.

[19] X. Sun, Y. Tian, Y. Liu, and Y. He, "An unstructured P2P network model for efficient resource discovery," in *First International Conference on the Applications of Digital Information and Web Technologies*, 2008, pp. 156–161.

[20] S. Zoels, Z. Despotovic, and W. Kellerer, "On hierarchical DHT systems - an analytical approach for optimal designs," *Computer Communications*, vol. 31, no. 3, pp. 576–590, 2008.

[21] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2001, pp. 149–160.

[22] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[23] Q. Li, H. Li, P. R. Jr., Z. Chen, and C. Wang, "CA-P2P: Context-aware proximity-based peer-to-peer wireless communications," *IEEE Communications Magazine*, vol. 52, no. 6, pp. 32–41, 2014.

[24] K. Arabshian and H. Schulzrinne, "Distributed context-aware agent architecture for global service discovery," in *2nd International Workshop on Semantic Web Technology For Ubiquitous and Mobile Applications*, 2006.

[25] S. S. Yau, Y. Wang, and F. Karim, "Development of situation-aware application software for ubiquitous computing environments," in *26th Annual International Computer Software and Applications Conference*, 2002, pp. 233–238.

[26] A.-M. Kermarrec and P. Triantafillou, "XL peer-to-peer pub/sub systems," *ACM Computing Surveys*, vol. 46, no. 2, pp. 16:1–16:45, 2013.

[27] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand, "Heuristics for scheduling parameter sweep applications in grid environments," in *9th Heterogeneous Computing Workshop*, 2000, pp. 349–363.

[28] A. F. Leite and A. C. M. A. de Melo, "Executing a biological sequence comparison application on a federated cloud environment," in *19th International Conference on High Performance Computing*, 2012, pp. 1–9.

[29] S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of EC2 cloud computing services for scientific computing," in *Cloud Computing*. Springer, 2010, pp. 115–131.

[30] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for many-tasks scientific computing," *IEEE Transactions on Parallel and Distributed System*, vol. 22, no. 6, pp. 931–945, 2011.

[31] E. Feller, L. Rilling, and C. Morin, "Snooze: A scalable and autonomic virtual machine management framework for private clouds," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2012, pp. 482–489.

[32] H. Kim and M. Parashar, *CometCloud: An Autonomic Cloud Engine*. John Wiley & Sons, Inc., 2011, pp. 275–297.

[33] A. J. Ferrer, F. Hernndez, J. Tordsson, E. Elmroth, A. Ali-Eldin, C. Zsigri, R. Sirvent, J. Guitart, R. M. Badia, K. Djemame, W. Ziegler, T. Dimitrakos, S. K. Nair, G. Kousiouris, K. Konstanteli, T. Varvarigou, B. Hudzia, A. Kipp, S. Wesner, M. Corrales, N. Forg, T. Sharif, and C. Sheridan, "OPTIMIS: A holistic approach to cloud service provisioning," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 66–77, 2012.

[34] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo, "Cloud-TM: Harnessing the cloud with distributed transactional memories," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 1–6, 2010.

[35] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan, "The Reservoir model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, no. 4, pp. 4:1–4:11, 2009.

[36] D. Petcu, C. Craciun, and M. Rak, "Towards a cross platform cloud API," in *1st International Conference on Cloud Computing and Services Science*, 2011, pp. 166–169.

[37] P. Leitner, Z. Rostyslav, A. Gambi, and S. Dustdar, "A framework and middleware for application-level cloud bursting on top of infrastructure-as-a-service clouds," in *6th IEEE/ACM International Conference on Utility and Cloud Computing*, 2013, pp. 163–170.

[38] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani, "Reconfigurable SCA Applications with the FraSCAti Platform," in *IEEE International Conference on Services Computing*, 2009, pp. 268–275.

[39] P. Verissimo, A. Bessani, and M. Pasin, "The TClouds architecture: Open and resilient cloud-of-clouds computing," in *IEEE/IFIP 42nd International Conference on Dependable Systems and Networks Workshops*, 2012, pp. 1–6.

[40] S. Imai, T. Chestna, and C. A. Varela, "Elastic scalable cloud computing using application-level migration," in *5th IEEE/ACM International Conference on Utility and Cloud Computing*, 2012, pp. 91–98.