Architectural Model and Planification Algorithm for the Self-Management of Elastic Cloud Applications

Loic Letondeur, Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, Noël de Palma

Architectural Model and Planification Algorithm for the Self-Management of Elastic Cloud Applications

Loïc Letondeur,
Xavier Etchevers,
Thierry Coupaye
Orange Labs
28, chemin du vieux Chêne
F-38243 Meylan - France
Email: {*firstname.lastname*}@orange.com

Fabienne Boyer,
Noël De Palma
Université Joseph Fourier
BP 53
F-38041 Grenoble Cedex 9 - France
Email: fabienne.boyer@imag.fr, noel.depalma@ujf-grenoble.fr

***Abstract*—This paper introduces a generic approach for managing automatically applications elasticity. The proposed solution addresses a noticeably wider scope of use-cases and does not depend on the underlying execution environment. It consists of: (i) a model and a formalism used for specifying valid applicative architectures (or elasticity scenarios) according to the resources allocated to the application. The originality of this first contribution lies in the association of a component model with a set oriented query language; (ii) a defeasible reasoning-based planning algorithm that computes the target applicative architecture from the model and elasticity requests; (iii) a first qualitative and quantitative evaluation that highlights the relevance and the viability of the proposed approach.**

I. Introduction

Cloud computing [1] constitutes an execution environment for applications (i.e. a cooperative set of software and hardware elements) that obtain the resources they need by consuming cloud services. These services can be either hardware entities (e.g. virtual machines or VMs), or software platforms pieces that can make up applications, or also full end-user applications. Because a cloud client is charged according to the consumed services, this consumption shall fit the best with the application charge. At any time, an application must consumes enough services to have enough resources but must also minimizes its services consumption to optimize costs. This optimization is operated dynamically, during the application run and without service interruption. This is a complex process called *elasticity*. It consists in modifying the application by adding, removing or else reconfiguring application elements.

One cloud promise is to automate the elasticity. This is a process that mainly must answer to two issues:

1) *When* and *which operation* must be decided?
2) *How* this decision must be concretely realised to adapt the application. This issue is related to a *Planification* of modifications to operate on the application.

Current solutions aiming at offering an automated elasticity do not explicitely address the Planification. Such a situation results in strong limitations around covered elasticity scenarios and all applications can not be addressed. This is the result of a tightly coupling of each elastic solution with a particular context of elasticity [2].

This paper proposes a general solution that pushes the limits of the current automated elasticity. It proposes a framework for a generic elasticity that operates regardless of the application, its architecture and the cloud provider. More precesely, contributions of this paper are:

- a model for the description of an elastic application in the cloud.
- an algorithm that makes use of the model for the planification of elastic applications.
- a first quantitative and qualitative evaluation of the proposed approach.

This article is organized as follows. Section 2 describes the motivations of the proposed solution. Section 3 presents the principles of the solution. Section 4 presents the implementation of a prototype and its evaluation. Section 5 provides the positioning of the work presented in this article. Section 6 concludes and presents the perspectives of this work.

II. Motivations

Making an application elastic in the cloud implies to be able to dynamically modify it. For instance, more worker parts can be added to manage heavy loads (*scale out* operation), or else removed during lighter loads to save resources, hence money (*scale in* operation). In the case of a web three tiers application (i.e. one presentation tier, one business tier, one database tier), a typical scale out consists in adding one more application server to the business tiers. Such a *scale out* is in fact more complex than just booting one VM and running an executable inside: this operation requires not only a functional configuration of the added server but also a reconfiguration of the running application to integrate the new server. So, the management of elasticity is an adaptive process that consists in continuously scheduling a set of operations ranging from VM creation to the configuration of all applicative functional entities. Moreover, **elasticity issues do not solely consist in deciding when a *scale in/out* is needed, they also include the way *how* to effectively realise it**. All solutions that aim at offering an automated elasticity must deal with the continuous determination of the full realisation of elasticity scales.

The automation of elasticity is a continuous and an adaptive process for which the autonomic computing's approach is well-suited. Autonomic computing models such a permanent adaptation over resources by the so-called MAPE-K [3] loop (or autonomic control loop).

Nevertheless, the consideration of the state of the art of the automated elasticity shows that all stages of MAPE-K are not equally addressed. Whereas Monitoring, Analysis and Execution have a vast variety of solutions, **Planification is either implicit and proper to a particular context or not addressed at all**. In facts, all approaches get around the problem of the planification by simplifying elasticity. This simplification consists in shrinking the elasticity parameters as much as possible to obtain a solution able to manage the elasticity automation in a particular context. For example, for one application, Amazon and Microsoft use the same virtual image, with the same VM hardware profile, only in their cloud, and in the same geographic area. The way how all application parts work together are achieved by proprietary intermediate load balancers. These solutions have the big advantage to offer an automated elasticity that can be simply used for all multi-tiers applications. However, these solutions have two major drawbacks: **a restricted elasticity** (only n-tiers architectures with explicit load balancer components)**and a strong vendor lock-in**. So, Amazon and Microsoft leverage planification by removing many degrees of liberty. The provided elasticity can be simply used but is also limited and a vendor lock-in occurs. In these solutions, Analysis and Planification are mixed together: the decision of an elasticity operation is implicitly known how to be fulfilled (one virtual image, one hardware profile, one geographic region, in one cloud, with known load balancers).

With both of these approaches, the Planification is coarse. It results in many limitations:

- Some applications without a multi-tiers architecture are not addressed. For example, Content Delivery Networks (CDN), and Map-Reduce frameworks can not be made elastic directly by those platforms. Such applications require either to use a similar service of the cloud provider (vendor lock-in) or to rewrite a new tool to manage such an application.
- Management of HA across more than one cloud is made more complex because of the use of vendor specific services (load balancers of Amazon and Microsoft).
- The automation of elasticity across different parts of clouds is not possible.
- The coverage of elasticity scenarios are restricted to a defined set. The vertical elasticity is not always offered and this is a big problem to respond to sudden loads.
- The automation of elasticity across different clouds is not managed by some existing solutions.

Thus, we believe that it is primordial to have a real Planification separated form the Analysis, so as to separate the specification of elasticity objectives from the realization of these objectives. The proposed Planification in this paper is separated from the the rest of other MAPE-K steps. It addresses the issue of **how an application must be reconfigured to apply a decision for an elasticity operation**. There is no restriction around elasticity parameters described at the beginning of this section.

III. Principles of the solution

The main goal of this paper is to provide a Planification which is integrated into a MAPE-K loop to offer an automated elasticity. The Analysis decision is its input whereas its output is the Execution model.

Besides the integration into a MAPE-K loop, the Planification must do some computations proper to its role. To this, our solution makes use of a model that centralizes all "live" parameters and parts of the application: it represents the current state of an application. This model is called *Extensional Model* (EM) and is continuously updated to always represent the current state of an application. The EM is also at the heart of the computation of the further application states: it is the support for all computations of the planification.

The proposed Planification has also a template for all possible EMs for an application. This is a second model different of the EM called *Intensional Model* (IM) [4].

A third feature is an algorithm that makes use of both the EM and the IM. When an elasticity operation is decided by the Analysis, the algorithm computes the current EM into a next one with consideration of the IM.

A. Extensional Model

The Extensional Model (EM) represents all relevant data about the application at a given time. In MAPE-K, this model centralizes all information the Monitoring lets know about an application. During the Planification, the EM is used to determine how the application will be modified by the Execution. The EM relies on a component-based architecture like Fractal [5]. Thanks to the use of a component based architecture, the EM is generic . In the EM, **each part of an application is represented by a component**: note that this is just a representation of underlying applicative resources and absolutely not an implementation requirement for applications. Consequently, **there is no need to recode the applications**.

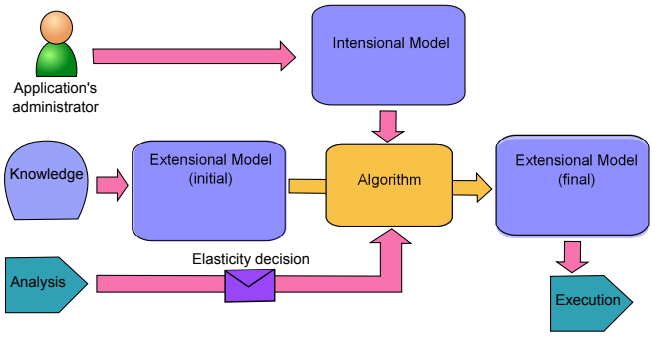

Fig. 1. Planification's inner functioning. The algorithm computes a target application state (i.e. a new Extensional Model (EM)) according to the Intensional Model, the current EM and the Analysis elasticity decision.

A component can be either a *primitive component* either a *composite component*. A primitive component contains no other components whereas a composite component may contain zero or more components (primitive and/or composite). In the rest of this article, primitive components are just called *components* and composite components *containers*. Both components and containers may have attributes that can represent their internal parameters like an IP address, a network port or a path to an executable. Apart from components and containers, a component based architecture use two other concepts: bindings and placements.

- Components may have *bindings* with others. A binding is in fact a communication channel between two components. Bindings are oriented from the client component to the server component: it means that the client depends on the server to work.
- Components are located into a hierarchy of containers in which one container can be contained by another one. The location of a given component or a given container inside a container defines a placement: this contained component or this contained container is said to be placed inside the outer container.

The Figure 2 depicts a Content Delivery Network (CDN) application. This application aims at delivering data from a central server to geographically spread users through a distributed system constituted of several *nodes*. This service has two goals:

- Reducing load on the central server.
- Improving the user's experience by reducing latencies and/or maximizing the throughput.

To achieve these goals, the nodes need to store data and to be located near the users whereas the users must be routed to the best node. CDN services are typically used to deliver static data content of a web server like files or images. They are also used to provide streaming of video contents.

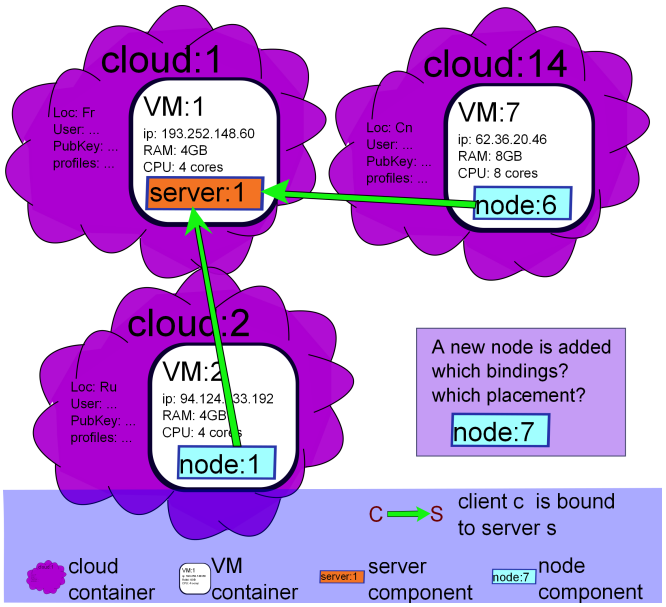

Fig. 2. Graphical representation of an Extensional Model (EM) for a CDN deployed in the cloud.

All components have one binding to the central server. They are also placed into a VM container, itself placed into a cloud container. All components and all containers have a unique id. For instance, the component named *node:1* is placed into the VM container named *VM:2*, itself placed into the cloud container named *cloud:2*. Some attributes are also mentioned as their IP addresses.

B. Intensional Model

The Intensional Model is a template for all possible EMs. It constitutes a mean to express *How* an application must be modified during elasticity. All possible modifications are coming from different types of constraints:

- Inheritance: an application is composed of one or more components. All of these components inherit their global characteristics from a type of component similarly to an object in the object oriented programming paradigm. In the EM, a component can not exist if it is not the instance of a component type. By the same way, a container must inherit from container type.
- Bindings: each component has bindings from or to other components. These bindings must satisfy some constraints. For example, in Figure 2, each component representing a node must have a unique binding to the component *server:1*.
- Placements: each component must be placed into a complete hierarchy of containers. For instance, in the Figure 2, each component must be placed into its proper VM container. Each VM container must be placed into one cloud container.
- Configurations: each component and each container must have a complete configuration. In the case of the Figure 2's example, each node component must expose its service trough the port 80.
- Mix of the previous constraints: in facts some constraints may result of the combination of inheritance, existing bindings, existing placements, or the current configurations. Such a mixed constraint can be seen in Figure 2: each VM container that contains a node component must be configured to have at least 2GB of RAM and 2CPUs.

In the approach proposed in this article, the IM makes use of an original formalism that allows for the introspection of the current EM, the determination of some modifications on the current EM and the verification of the constraints in the new EM. The IM specifies what are the component types used by an application. Each component present in an IM is an instance of one type. By this way is addressed the constraints dealing with the inheritance. Constraints about bindings, placements and configurations are addressed by means of a Set-Oriented Query Language (SOQL) as SQL [6]. It is the heart of the originality of the proposed formalism for the IM. A SOQL permits to extract subsets of entities from the EM, to filter some elements of such subsets and then

to order these filtered elements to obtain a result. In facts, a SOQL language constitutes a programming way to achieve the satisfaction of constraints. For example, in a pseudo-language, the placement constraint that requires 2GB RAM and 2 CPUs for all VM container in which a node component is placed, can be expressed very simply as shown by the Algorithm 1.

Algorithm 1 Example of a query for the configuration of all VM containers hardware profiles.

1: **for each** $Container\ c$ **in** $EM/[all\ VM\ containers]$
2: **where** $c\ contains\ a\ component\ inheriting from\ "node"$
3: **return** *Configure(c, RAM=2GB, CPU=2)*
4: ▷ Only VM containers hosting a node component are configured. The result *Configure(c, RAM=2GB, CPU=2)* will be interpreted by the Planification's engine.

In this example, the returned result is a configuration that is then interpreted by the engine of the Planification.

Using a programming SOQL to manage all constraints in the IM has several advantages. Firstly it is not specific to a kind of constraint. It can cover a wide range of concerns such as the little set of examples presented in the Table I.

Id	Constraint type	Concern	Constraint example
1	Binding	Quality of Service	Bind only to components located at a distance below a given limit (topology or geography)
2	Binding	Cardinalities	Bind to components not already bound more than *n* times
3	Placement	Price	Use only infrastructures whose price per VM and per hour is below a limit
4	Placement	Location	Use only infrastructures located in Europe
5	Placement	Availability	Having replicas for services which are geographically distributed
6	Configuration	Scaling	Having a VM profile according to placed components
7	Configuration	Load balancing	Affecting weighted coefficients according to the load of each member of a service (dynamic load balancing)

TABLE I
EXAMPLES OF DIFFERENT CONSTRAINTS THAT CAN BE EXPRESSED IN THE INTENSIONAL MODEL.

As a comparison, the use of cardinalities (i.e. a *min* and a *max* on a binding between to component types) only addresses how many components have bindings to and from other components. Cardinalities are so not suitable to express the constraint 1 in the Table I. Moreover, the use of cardinalities does not explain how to modify the EM during elasticity: this is the role of an algorithm that needs additional semantics. As instance, the preference to obtain the min or else the max of a cardinality depends on the algorithm. A SOQL has an expressiveness very useful to address present and future concerns relative to the constraints required for a generic elasticity.

A second advantage of using a SOQL concerns its ability to manage a infinity of possible configurations. On contrary, the use of a list of all possible configurations quickly becomes a painful job.

A third advantage for a SOQL approach is its batch processing ability. As shown is the pseudo-language example in this section, one query treats entities by groups and return a set of results specific for one entity.

A fourth advantage is the ease for an administrator to use the proposed approach. SOQL is indeed well known thanks to a widespread usage of SQL [6] with relational databases or XQuery [7] with NoSQL databases. On contrary to a Domain-Specific Language (DSL), SOQL makes easier the learning for an application's administrator. There is also neither dependencies to solve nor compilation as with a compiled language.

The IM uses a SOQL formalism that permits the coverage of extensible constraints for the modification of the EM during the elasticity. This formalism needs to be interpreted and the returned result must be applied on the EM. This task is achieved by the second contribution of this article: an algorithm able to make changes in the EM according to IM and an operation of elasticity decided by the Analysis.

C. An algorithm to plan modifications

The algorithm is responsible for computing the transition from the current EM to a next one according to the constraints expressed in the IM. This transition is a set of modifications and must satisfy all types of constraints of the IM. The proposed algorithm relies on one fundamental property: each type of constraints results in a precise set of modifications over the EM. The Table II shows what are the possible modifications by type of constraints.

All modifications listed in the Table II are handled by the algorithm. However, the listed constraints may be mixed together. An example of such a mixed constraint is the creation of bindings according to the placement: only components placed into the same geographic area must be bound together (e.g. mix of constraints 5 and then 1 in the Table I). This constraint is really useful for the management of both HA and performances: by this way, an application is geographically distributed and latencies are also controlled. This mix of constraints constitutes a big issue for the ease of use. Indeed,

Constraint type	Possible modifications
Inheritance	add/remove a component/container
Bindings	bind/unbind components
Placements	place a component/container into a container
Configurations	set/unset a parameter of a component/container

TABLE II
EXAMPLES OF DIFFERENT CONSTRAINTS THAT CAN BE EXPRESSED IN THE INTENSIONAL MODEL.

the problem comes from the wide range of compositions possible: as binding constraints according to placements, as placement constraints according to bindings may occur. An example of the second statement is the placement of one probe component (e.g. for a monitoring tool) that is responsible of a new added component as a CDN node component (i.e. a scale-out operation): this probe must be placed into the same cloud container and the same VM container than the node component. This example underlines where resides a big challenge for each administrator: the creation, the understanding and the debug of complex mixed contraints expressed in the IM.

The proposed algorithm tackles this difficulty by composing constraints together. This algorithm automatically mixes simple constraints and offers in the same time a clear vision of how they are mixed. This algorithm executes each type of constraints in a given order (note this order is arbitrary). One after one, all types of constraints are computed. The type of executed constraints is verified by allowing only the modifications covered by each type of constraints as mentioned in the Table II, and possibly a *revision* requirement. As instance, in the step corresponding to the bindings constraints, the only modifications allowed are bind/unbind components or else a notification meaning *"a problem occurs because of the given reason XXX"* (where XXX is the problem description). Such an algorithm has two advantages:

- The creation of complex constraints are achieved by the composition of very simple constraints.
- Such queries are reusable: the simpler the queries, the more reusable. Queries are indeed less complex and so they are less specific to either one application, or one architecture, or one cloud... For example, the binding query that results in the binding of each node component to the unique server component *server:1* in the Figure 2 can be also used for a web application between the business tier and the database.

The algorithm proposed is written in the Algorithm 2. From the line 1 to the line 17, the algorithm begins with applying the elasticity decision on a copy of the current EM and then completes this first modification by additional modifications according to the constraints. From the line 5 to the line 17, all constraints types are computed on the target EM (i.e. the next application state during its creation). Inheritance constraints are managed when components or containers are added into the computing EM like in lines 5 and 30. Other constraints types (i.e. Bindings, Placements, Configurations) are managed by the function *manageConstraints* (lines 20 to 35). Each of the former constraints types are computed successively in lines 7, 9 and 11 by calling the function *manageConstraints*. This function computes each constraint of a given type, one after one (line 24). For each constraint, if no revision is required (line 25), the computed modifications can be applied on the computing EM (line 26). Only allowed modifications of the Table II are effectively applied. On the contrary, if a revision is required (line 27), the algorithm is rerouted to an additional computation step responsible for managing a revision. This step is denoted by the function *handleRevision* (lines 38 to 46). It mainly consists in performing a matching between the code that identifies the revision and additional modifications. For instance, if a revision meaning "an empty VM lacks" is run, this revision will match the pattern line 41 and a modification of the EM doing the addition of a new VM container is returned (line 42). In line 30, all additional modifications are applied onto the EM and the main sequence is started again (line 6).

Algorithm 2 Planification's algorithm.

INPUT: An elasticity operation from the Analysis or else an update request

ALGORITHM:

```
1:  M ← input.getCorrespondingBaseModifications()   ▷ First: get base modifications corresponding to the input
2:  em' ← em.copy()                                 ▷ Copy the current EM
3:  endFlag ← false                                 ▷ Set a flag meaning the algorithm's end
4:                                                  ▷ Inheritance is treated during the following line
5:  em' ← em'.doAllModifications(M)                 ▷ Apply base modifications on the EM's copy: add/remove/set components/container
6:  while not(endFlag) do                           ▷ Start the main sequence
7:      revisionFlag ← MANAGECONSTRAINTS(em',Bindings,im)
8:      if not(revisionFlag) then
9:          revisionFlag ← MANAGECONSTRAINTS(em',Placements,im)
10:         if not(revisionFlag) then
11:             revisionFlag ← MANAGECONSTRAINTS(em',Configurations,im)
12:             if not(revisionFlag) then
13:                 endFlag ← true
14:             end if
15:         end if
16:     end if
17: end while
18:
19:
20: function MANAGECONSTRAINTS(em',t, im)
21:     typedConstraints ← im.getConstraintsOfType(t)   ▷ get constraints by type in the IM
22:     revisionFlag ← false
23:     for c in typedConstraints do
24:         (M, r) ← c.executeOn(em')               ▷ Get new modifications in M and revision in r
25:         if r is null then                       ▷ No revision required: apply modifications
26:             em' ← em'.doAllowedModificationsForType(M, t)
27:         else                                    ▷ An revision is required
28:             revisionFlag ← true
29:             HANDLEREVISIONS(em',c,M,r)
30:             em'.doAllModifications(M)           ▷ Apply revision modifications
31:             break   ▷ Return to begin a new main sequence: Inheritance was checked in the previous line
32:         end if
33:     end for
34:     return revisionFlag
35: end function
36:
37:                 ▷ According to a set of revisions declarations, find a corrective set of modifications
38: function HANDLEREVISIONS(em',c,M,r)
39:     if r.getCode().equals("Lacks one component of type") then
40:         M ← {new Modification(add, r.getConcernedType()}
41:     else if r.getCode().equals("Lacks one container of type") then
42:         M ← {new Modification(add, r.getConcernedType()}
43:     else if others then
44:         ...
45:     end if
46: end function
47:
```

OUTPUT: Transform EM' (i.e. the new EM) into the Execution model formalism and then give it to the Execution

IV. Evaluation

The main objective of the following evaluation consists in showing the coverage of new elasticity scenarios, the ease of use and the efficiency of our solution. Such efficiency does

not concern the performances of an elastified application but rather the overhead introduced by an explicit Planification.

A. Implementation and examples

The formalism of the intensional model and the extensional model is based on XML. Constraints are expressed in the intensional model thanks to XQuery [7]. The code of the prototype engine (Vulcan) is based on the Java language. XQuery is a set-oriented query language particularly suitable for handling XML trees.

The Figure 3 and the Figure 4 show two IM examples.

XQuery is used to describe constraints in both of these examples. All requests used in the two examples are provided by a library. Even if the two modelled applications are very different, their IM are really similar. Some constraints are even used in the two IMs like a constraint requiring that each component must be placed into its own VM container (denoted by *local:placeOneOne()*). In addition to this similarity, one other can be found in the bindings contraints: *local:bindAllExactlyOne(X,Y)* denotes a constraint resulting in binding all components of type *X* to one component of type *Y*. The implementation presented uses XQuery to express constraints. This expression allows the engine (Vulcan) to make use of the intensional model to determine extensional models. The planification algorithm permits to express very dissimilar applications with reusable descriptions using libraries. Complex constraints are addressed by some simple ones. Thanks to these characteristics, making elasticity for an application is really simple.

```
<intensional-model>
<inheritance>
  <component name="Apache" init="1"/>
  <component name="Jonas" init="2" max="10"/>
  <component name="MySQL" init="1"/>
</inheritance>
<bindings>
  <query>local:bindExactlyOneAll("Apache","Jonas")</query>
  <query>local:bindAllExactlyOne("Jonas","MySQL") </query>
</bindings>
<placements>
  <query>local:placeOneOne()</query>
</placements>
<configurations>
        <query>local:configureVMs("2cpus", "2GB")</query>
</configurations>
[...]
```

Fig. 3. Example of a intensional model for the application *Springoo* a web three tier application (representative of 80% of the Information System (IS) of Orange. In section *configurations*, a query sets a hardware profile for all VM containers (RAM: 2GB, CPU:2cores).

```
<intensional-model>
<inheritance>
  <component name="server" init="1" max="1"/>
  <component name="node" init="2"/>
</inheritance>
<bindings>
  <query>local:bindAllExactlyOne("node","server")</query>
</bindings>
<placements>
  <query>local:placeOneOne()</query>
  <query>local:placeOneVMInOneCloud()</query>
</placements>
<configurations>
        <query>local:configureVMsHosting("2cpus", "2GB",
                                server)</query>
        <query>local:configureVMsHosting("4cpus", "4GB",
                                node)</query>
</configurations>
[...]
```

Fig. 4. Example of an intensional model for a Content Delivery Network application. In section *configurations* two queries set VMs hardware profiles.

B. Elasticity scenarios

This section lists a **sample set** of elasticity scenarios. They demonstrate the expressiveness of Vulcan. Figure 5 depicts all of these scenarios.

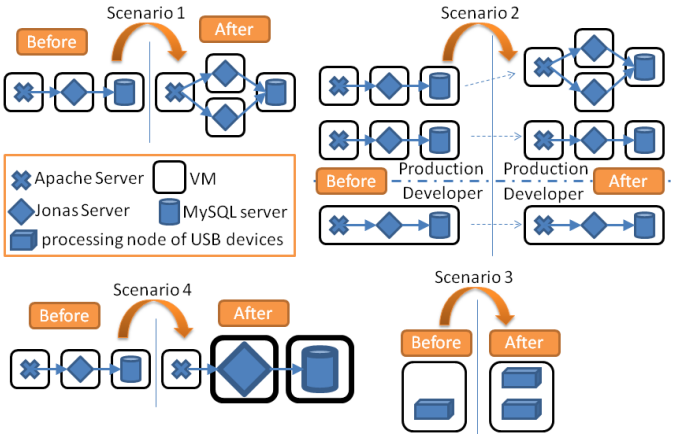

Fig. 5. Elasticity scenarios to evaluate the expressiveness of Vulcan.

1) Scenario 1: Multi-tier elasticity: This first scenario is elementary. It consists in operating elasticity on a multi-tier application on its business tier. To this, a Jonas application server is added into a new VM application. This scenario corresponds to the industrial state of the art. The proposed approach addresses this scenario and other which are more complex. Such more complex scenarios are exposed below.

2) Scenario 2: Profiled elasticity: An Orange-internal use case intends to modulate the deployment of an application according to a profile. When a profile named *developer* is used, the application is deployed in a single VM. However, the use of a profile *production* results in the deployment of the same application in different VMs and with a specific initial scale. In facts, the developer profile allows a development team to test functionalities. In this case, there is no need for good performances. On contrary, the production requires an application supporting the load. Each deployment must be elasticized independently with each others.

Vulcan addresses this scenario by using containers. Only one application is deployed but some subsets of components are isolated from each other according to their location in

a container. Thus it allows Vulcan to apply each elasticity request on precise subsets.

3) Scenario 3: Fine grained elasticity (intra-container): Another Orange-internal use case aims at managing multiple USB devices through over IP remote communications. The devices are managed by an application hosted in the cloud. This application is composed of many processing nodes. Such nodes have a limit inherent to the USB bus which prevents them from managing more than 128 devices simultaneously. Nevertheless, by hosting only one node a VM remains underutilized. One solution to this problem consists in grouping many processing nodes in one VM. Practically when a new device appears, it must be ensured that a processing node can manage it. If this is not the case, a new one must be added. Of course, one VM can not host too many nodes: the ability to add VMs must be also used according to the context.

Vulcan provides elasticity at the component level. Each component is executed within a VM. Vulcan can dynamically add/remove instances in VMs. It therefore addresses this scenario of elasticity thanks to its granularity finer than the VM.

4) Scenario 4: Vertical Elasticity: The vertical elasticity consists in operating changes on the processing capabilities of components and containers rather than their number. For example, during a load peak, it may be interesting to resize the profile of the VM by adding RAM and/or processor cores. The provision of a VM is effectively a long process. Because all load peaks are not predictable, it may be important to have a high reactivity.

Vulcan has the ability to configure resources and therefore VMs. Vulcan offers not only the possibility of establishing profiles of VMs but also of operating deployment only in infrastructures able to manage dynamic changes of VM profiles.

C. Performances

Tests were achieved on machine with a dual-core Intel Xeon W3503 processor and 2GB of RAM. The measured times correspond to the resolution of an elasticity request for adding one Jonas server to those already present. Figure 6 shows the measured time in function of the final number of Jonas servers. This graph shows that the time increases

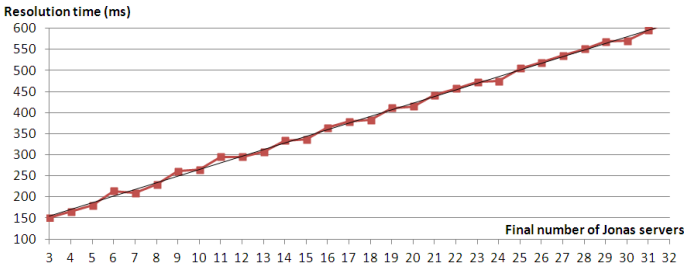

Fig. 6. Planification time in milliseconds depending on the final number of Jonas servers.

linearly with the number of servers. This increase is due to the placement constraint that requires to place each component in a separate VM. It requires to introspect each VM container to check if it is empty. Other experiments not reported here have shown comparable performances with other applications. The encountered time necessary for provisioning one VM [8] usually ranges from about 30s. to one minute. The ratio of the measured resolution time on the provisioning VM duration is in the worst case of 0.020. This rate shows that - for this scenario - the impact of processing elasticity planning is almost negligible. As a matter of fact, the approach is valid.

V. Related works

The consideration of both the literature and the industrial solutions shows that all steps of the MAPE-K loop applied to elasticity are not fairly addressed. The ecosystem of the monitoring step is rich. It presents both general solutions like Zabbix [9], Shinken [10] or Nagios [11] and proprietary solutions like Amazon CloudWatch [12]. Concerning the analysis, various research solutions exist. They generally focus on applications with multi-tier architectures [13], [14], [15]. Some of these solutions make possible the management of quality of service [16], [17]. The ecosystem of the analysis part also includes proprietary solutions like Amazon CloudWatch and RightScale [18] which trigger elastic operations of additions and removals of VMs when thresholds on different metrics are exceeded. Other solutions, which offer higher level, such as Microsoft Azure [19] and Redhat OpenShift [20] manage automatically the elasticity of some specific software components. Considering the fourth step, execution, its ecosystem is rich and consists of generic solutions such as VAMP, or more limited or specific ones as VEGA [21], Amazon AWS, Microsoft Azure and Redhat OpenShift.

For the planning step, to our knowledge, two generic solutions exist. The first is Application Deployment Toolkit(ADT) [22]. It provides a framework for the development of planification-like programs. To do this, ADT offers a set of features for the implementation enabling the execution and the monitoring of managed resources. This solution offers APIs for the development of both analysis and planification. ADT does not manage itself all aspects related to the planification. The second solution named ElaaS [23] allows to model the architecture of an elastic application by abstracting the specificities of a temporary architecture. It provides APIs allowing analysis to interface with it. Nevertheless, the aspect of planning is also let to the analysis.

Specific ”black boxes” planification solutions also exist but are used to address a limited number of scenarios. Amazon Auto-Scaling manages elasticity within Amazon infrastructure with VMs having necessarily the same profile, in the same area, with the use of load balancers between different groups of elasticity. RightScale is based on a similar approach with similar limits. In the case of Microsoft Azure and RedHat OpenShift, the planification is not accessible to the user. It applies only in a specific execution context and a finished software components available in a catalog together. ConPaaS [24] composes an application thanks to interconnected services. This solution has a VM granularity and can not finely manage

placement of components. The planification step has a lack of generic solutions able to manage elasticity independently of the application and the execution context. Table III summarizes the management by various industry and research solutions of elasticity scenarios presented in the evaluation.

	Scenario 1	Scenario 2	Scenario 3	Scenario 4
Amazon AWS	**Yes**	No	No	No
ConPaas	**Yes**	**Yes**	No	No
Microsoft Azure	**Yes**	No	No	No
Redhat OpenShift	**Yes**	No	No	No
RightScale	**Yes**	No	No	No
Vulcan	**Yes**	**Yes**	**Yes**	**Yes**
ADT	Partial	Partial	Partial	Partial
ElaaS	Partial	Partial	Partial	Partial

TABLE III
MANAGEMENT OF ELASTICITY SCENARIOS BY DIFFERENT SOLUTIONS. THESE SCENARIOS ARE PRESENTED IN THE EVALUATION. A *partial* MEANS THAT THE SOLUTION DOES NOT PROVIDE ITSELF THE MANAGEMENT OF THE SCENARIO BUT DOES NOT BLOCK IT.

VI. CONCLUSION AND PERSPECTIVES

This article proposes a solution for the planification aiming at offering an automated elasticity to applications deployed in the cloud. A first contribution is a model and its formalism for the description of elastic applications. The model used aims at describing how an application must be modified to apply an elasticity decision. This model makes use of a formalism based on a set-oriented query language. The formalism allows to express all constraints to be satisfied during elasticity. It is generic with regards to the concern its addresses (e.g. inheritance, bindings,...) and expressive enough to cover a wide range of scenarios not accessible by other solutions. A second contribution is an algorithm which computes new application architectures according to an elasticity decision. It allows an administrator for the expression of very complex constraints by specifying simple ones. This algorithm composes all constraints together. It eases the administrator's job by permitting to reuse library constraints. This algorithm is implemented inside an engine which uses the formalism introduced in this article. A third contribution is an evaluation of the proposed approach and its implementation. Through various examples it shows abilities for the re-usability of constraints and a larger coverage of elasticity in terms of managed scenarios. An evaluation shows the viability of both the formalism and the proposed implementation thanks to a negligible overhead. Vulcan currently manages elasticity requests one after one. A possible evolution is to manage several requests at the same time (group and/or competition). A more industrial projection is to achieve a complete autonomous loop of elasticity placing Vulcan as at the output of an analysis solution, as at the input of an execution one like VAMP [8].

ACKNOWLEDGEMENTS

This work is partially supported by the FSN OpenCloudware project, by the FSN Datalys project and by the CtrlGreen ANR project.

REFERENCES

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html
[2] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *Computer Communication Review*, vol. 41, no. 1, pp. 45–52, 2011.
[3] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.
[4] M. Hofmann, *Extensional constructs in intensional type theory*, ser. CPHC/BCS distinguished dissertations. Springer, 1997.
[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Softw., Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.
[6] T. Lacy-Thompson, *Informix-SQL - a tutorial and reference*. Prentice Hall, 1969.
[7] N. Wiegand, "Investigating xquery for querying across database object types," *SIGMOD Record*, vol. 31, no. 2, pp. 28–33, 2002.
[8] X. Etchevers, T. Coupaye, F. Boyer, and N. D. Palma, "Self-configuration of distributed applications in the cloud," in *IEEE CLOUD*, L. Liu and M. Parashar, Eds. IEEE, 2011, pp. 668–675.
[9] "Zabbix website," https://www.zabbix.org/.
[10] "Shinken website," http://www.shinken-monitoring.org/.
[11] "Nagios website," http://www.nagios.org/.
[12] "Amazon web services website," http://aws.amazon.com/fr/.
[13] K. Konstanteli, T. Cucinotta, K. Psychas, and T. A. Varvarigou, "Admission control for elastic cloud services," in *IEEE CLOUD*, R. Chang, Ed. IEEE, 2012, pp. 41–48.
[14] P. Marshall, H. M. Tufo, and K. Keahey, "High-performance computing and the cloud: a match made in heaven or hell?" *ACM Crossroads*, vol. 19, no. 3, pp. 52–57, 2013.
[15] H. Nishimura, N. Maruyama, and S. Matsuoka, "Virtual clusters on the fly - fast, scalable, and flexible installation," in *CCGRID*. IEEE Computer Society, 2007, pp. 549–556.
[16] A. Ali-Eldin, J. Tordsson, and E. Elmroth, "An adaptive hybrid elasticity controller for cloud infrastructures," in *NOMS*. IEEE, 2012, pp. 204–212.
[17] Y. Kouki and T. Ledoux, "Csla: A language for improving cloud sla management," in *CLOSER*, F. Leymann, I. Ivanov, M. van Sinderen, and T. Shan, Eds. SciTePress, 2012, pp. 586–591.
[18] "Rightscale website," http://www.rightscale.com.
[19] "Microsoft azure website," http://www.microsoft.com/windowsazure/.
[20] "Redhat openshift website," https://www.openshift.com/.
[21] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, "Dynamic scaling of web applications in a virtualized cloud computing environment," in *ICEBE*. IEEE Computer Society, 2009, pp. 281–286.
[22] M. Keller, C. Robbert, and M. Peuster, "An evaluation testbed for adaptive, topology-aware deployment of elastic applications," in *SIGCOMM*, D. M. Chiu, J. Wang, P. Barford, and S. Seshan, Eds. ACM, 2013, pp. 469–470.
[23] P. Kranas, V. Anagnostopoulos, A. Menychtas, and T. A. Varvarigou, "Elaas: An innovative elasticity as a service framework for dynamic management across the cloud stack layers," in *CISIS*, L. Barolli, F. Xhafa, S. Vitabile, and M. Uehara, Eds. IEEE, 2012, pp. 1042–1049.
[24] G. Pierre and C. Stratan, "Conpaas: A platform for hosting elastic cloud applications," *IEEE Internet Computing*, vol. 16, no. 5, pp. 88–92, 2012.