HAL
open science

Reliable Self-Deployment of Cloud Applications

Xavier Etchevers, Gwen Salaün, Fabienne Boyer, Thierry Coupaye, Noël de Palma

To cite this version:

Xavier Etchevers, Gwen Salaün, Fabienne Boyer, Thierry Coupaye, Noël de Palma. Reliable Self-Deployment of Cloud Applications. SAC 2014 - 29th ACM Symposium on Applied Computing, Mar 2014, Gyeongju, South Korea. hal-00934042

HAL Id: hal-00934042

https://inria.hal.science/hal-00934042v1

Submitted on 21 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Reliable Self-Deployment of Cloud Applications

Xavier Etchevers
Orange Labs, France
xavier.etchevers@orange.com

Gwen Salaün
Grenoble INP, Inria, France
gwen.salaun@imag.fr

Fabienne Boyer
UJF-Grenoble 1, LIG, France
fabienne.boyer@imag.fr

Thierry Coupaye
Orange Labs, France
thierry.coupaye@orange.com

Noël De Palma
UJF-Grenoble 1, LIG, France
noel.depalma@imag.fr

ABSTRACT

Cloud applications consist of a set of interconnected software elements distributed over several virtual machines, themselves hosted on remote physical servers. Most existing solutions for deploying such applications require human intervention to configure parts of the system, do not respect functional dependencies among elements that must be respected when starting them, and do not handle virtual machine failures that can occur when deploying an application. This paper presents a self-deployment protocol that was designed to automatically configure a set of software elements to be deployed on different virtual machines. This protocol works in a decentralized way, *i.e.*, there is no need for a centralized server. It also starts the software elements in a certain order, respecting important architectural invariants. This protocol supports virtual machine and network failures, and always succeeds in deploying an application when faced with a finite number of failures. Designing such highly parallel management protocols is difficult, therefore formal modeling techniques and verification tools were used for validation purposes. The protocol was implemented in Java and was used to deploy industrial applications.

1. INTRODUCTION

Cloud computing emerged a few years ago as a new approach based on virtualization for an efficient delivery of hardware resources and software applications over a network (typically the Internet). One of the main reasons organizations adopt cloud computing is to reduce IT costs by outsourcing hardware and software maintenance and support. Cloud computing is at the junction of several recent computing paradigms such as grid computing, virtualization, autonomic computing, peer-to-peer architectures, utility computing, etc. It allows users to benefit from all these technologies without requiring extensive expertise in each of them. Autonomic computing is particularly convenient for automating specific tasks such as on-demand resources provisioning or facing peak-load capacity surge (*a.k.a.*, elasticity management). Automation reduces user involvement, which speeds up the process and minimizes human errors.

In this paper, cloud applications are distributed applications composed of several virtual machines running a set of interconnected execution units called software elements. This type of cloud application can benefit from several services provided in the cloud such as database storage, virtual machine cloning, or memory ballooning. To deploy their applications, cloud users need first to build virtual images corresponding to the applicative software stacks (*i.e.*, including operating system, middleware, binaries and data), to provision and instantiate them as virtual machines (VMs), and to indicate the software elements to be run on them. Then, they have to configure these software elements. This involves setting up the configuration parameters that depend on the runtime environment (*e.g.*, IP address, port number). Finally, cloud users have to start the software elements. Both configuration and activation tasks are complex and error-prone if handled manually due to functional interdependencies between software elements. These dependencies between the software elements of a given application define the order which must be respected during the configuration and activation process. This order avoids the application reaching undesired inconsistent states where, for instance, a started software element is connected and sends requests to another element that is not started yet. Therefore, there is a need for management protocols to automate these deployment tasks. A few recent works, *e.g.*, [17, 7, 13, 15], have focused on this issue. The contribution presented in this paper goes one step further than these works, by designing a fault-tolerant deployment protocol capable of supporting VM and network failures.

This paper introduces a novel self-deployment protocol able to automatically deploy an application on a cloud. Beyond instantiating each VM, the protocol is also responsible for starting each element in a precise order according to the functional dependencies of the applicative architecture. This start-up process works in a decentralized manner, without requiring any centralized manager. Thus, each VM embeds a local configuration agent, named *configurator*, which interacts with other remote configurators (*i.e.*, on other applicative VMs) to (i) solve dependencies by exchanging configuration information and (ii) determine when a software element can be started, *i.e.*, when all the elements it depends on are started. This protocol is also able to detect VM and network failures occurring during the configuration and activation process. When such a failure occurs, the protocol informs the remaining VMs of what has happened to make

.

the system restore a consistent state, and instantiates a new instance of the failed VM. The proposed protocol supports multiple failures and always succeeds in finally deploying the application and starting the corresponding components (assuming that the number of failures is finite).

Our management protocol involves a high degree of parallelism, which makes its design very complicated. Since correctness of the protocol was of prime importance, it was decided to specify the protocol using formal concurrent specification languages, namely the LNT value-passing process algebra [6]. LNT is one of the input languages of the CADP verification toolbox [16], which was used to verify that the protocol satisfies certain key properties, *e.g.*, *"when a VM fails, all the remaining VMs are notified of that failure"* or *"each VM failure is followed by the creation of a new instance of that VM"*. At the implementation level, we have proposed an XML-based formalism to describe the cloud applications to be deployed, and we have developed a Java tool chain, named Virtual Application Management Platform (VAMP), which includes the reference implementation of the self-deployment protocol [13]. For evaluation purposes, this implementation has been used to deploy real-world applications, *e.g.*, multitier Web application architectures or the Clif load-injection framework [10].

Our main contributions with respect to existing results on this topic are the following:

- We propose and design an innovative, decentralized protocol to automatically deploy cloud applications consisting of interconnected software elements hosted on several VMs.
- The deployment process is also able to detect and handle VM and network failures, and always succeeds in configuring the application at hand.
- We verified that the protocol respects some key properties using formal specification languages and model checking techniques.
- We implemented the protocol in Java and applied it to industrial applications for evaluation purposes.

The outline of this paper is as follows. Section 2 introduces the protocol, with a specific focus on failure detection and handling. Section 3 first presents the formal specification and verification tasks, and then the implementation aspects. In section 4, related works are discussed before concluding in section 5.

2. SELF-DEPLOYMENT PROTOCOL

This section first introduces the model used to describe the application to be deployed. Then it presents the protocol participants, the protocol itself, and our solution to handle VM and network failures.

2.1 Application Model

An *application model* represents an abstraction of the target application to be deployed. This model consists of two levels: the runtime environment and the applicative architecture. At the runtime environment level, an application is modeled using a set of VMs. Each VM is characterized by its hardware characteristics (*e.g.*, number of CPUs, size of memory) and a virtual image to be instantiated (*e.g.*, the associated software stack including an operating system, middleware, applicative binaries and data). These VMs do not play any role *per se*, from a functional point of view, but each of them hosts a set of applicative software elements, where the functional part of the application resides. The description of the software elements involved in an application is modeled using the applicative architecture level, which is based on the Fractal component model [5]. Each software element is abstracted as a component. A component can either provide or require services. Services are modeled using ports: an *import* port (shortened as import) represents a service required by a component, whereas an *export* port (shortened as export) represents a service provided by a component. An import on one component will be connected to an export on another component. This type of connection is called a *binding*. A component can import a service exported by a component hosted on the same VM (local binding) or hosted on another VM (remote binding). An import can be optional or mandatory. A component has three states: *started*, *stopped*, or *failed*. An import is *satisfied* when it is bound to a matching export and the component offering that export is started. A component can be started when all its mandatory imports are satisfied. Therefore, a component can be started even if its optional imports are not satisfied. A component moves to the failed state when its VM fails.

In the remainder of this paper, a three-tier Web application will be used as a running example. Figure 1 gives the application model, which consists of three VMs. The first one (VM1) hosts a front-end HTTP server (Apache). The second one (VM2) hosts a JEE application server (JOnAS). The third one (VM3) corresponds to the database management system (MySQL). These components are connected through remote bindings (*e.g.*, Apache bound to JOnAS) on mandatory imports (m).

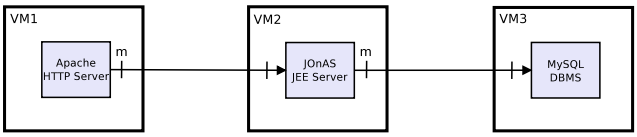

Figure 1: A Three-tier Web Application Model

2.2 Protocol Participants

The self-deployment protocol involves a deployment manager and a set of VM configurators (shortened as configurators). The deployment manager (DM) guides the application's configuration by instantiating VMs and creating a new instance of a VM when a failure occurs. Each VM in the distributed application is equipped with a configurator responsible for connecting bindings and starting components once the VM instance has been created by the deployment manager. Communication between participants (DM and VM configurators) is asynchronous, involving FIFO buffers. Each VM is equipped with two buffers, an input buffer and an output buffer. When a VM configurator needs to post a message, it puts that message in its output buffer. When a configurator wants to read a message, it takes the oldest one in its input buffer. Messages can be transferred at any time from an output buffer to its addressee's input buffer. It is worth noting that buffers are not explicitly bounded. They

are implicitly bounded by the communication system memory size, but the protocol does not involve looping tasks that would make the system send infinite messages to buffers.

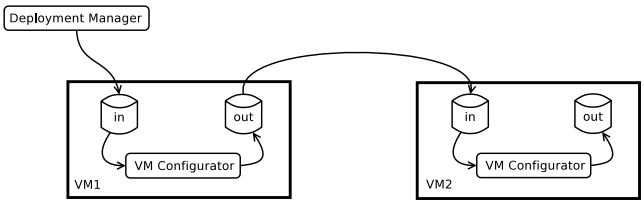

Figure 2: Participants

2.3 Component Start-up

Protocol execution is driven by the configurators embedded on each VM. All configurators evolve in parallel, and each of them carries out various tasks according to a precise workflow, as summarized in Figure 3. In this figure, boxes identified using natural numbers (❶, ❷, etc.) correspond to specific actions (VM creation, component creation, etc.). Diamonds stand for choices, and each choice is accompanied by a list of box identifiers that can be reached from this point.

The start-up process begins when the DM instantiates the VMs (Figure 3). For each VM, it creates an image of this VM (❶) and the VM starts to execute. Each VM is equipped with a configurator, which starts when the DM instantiates the VM. A configurator is responsible for binding ports as described in the application model and starting components in a specific order: a component can be started only if all its mandatory imports are satisfied.

We will now explain how a newly instantiated VM binds its ports and starts its components. At instantiation time, the VM is aware of the binding information (for both local and remote bindings). Therefore, each configurator has explicit knowledge of how its components are supposed to be bound to local or remote components. First, local components are created (❷). Local bindings are handled by the configurator and do not require any interaction with other VMs (❸). For remote bindings, the configurator must perform two tasks. When an export of one of its components is involved in a binding, the configurator sends a message with its export connection information (*e.g.*, IP address, port number) to the VM hosting the client component (❹). When an import of one of its components is involved in a binding, this VM will receive the connection details from the server VM (❼) at some point and, upon reception of that message, the configurator makes the binding effective (❽).

In terms of component start-up, a configurator can immediately start a component without imports or with only optional imports (❺). If a component involves mandatory imports, that component can only be started when all its mandatory imports are satisfied, *i.e.*, when all its imports are bound to started components. When a component is started and that component is used by a remote component, the configurator of the first component must inform the configurator of the second component that the component has been started. To do this, the first VM sends a *start* message to the second VM (❻). Upon reception of this message (❼), the configurator updates an internal data structure storing the partner component states (export side) for each component. Every time a *start* message is received, the configurator checks if the corresponding component can be started, *i.e.*, if all its mandatory imports are satisfied (❺). Note that the start-up process involves propagation of *start* messages along bindings across several VMs. Local bindings are handled directly by the configurator, and there is no need to exchange messages with other VMs either for binding or start-up purposes. The start-up process always terminates successfully because binding cycles over mandatory imports are forbidden. Failure handling (❾ in Figure 3) will be detailed in the next subsection.

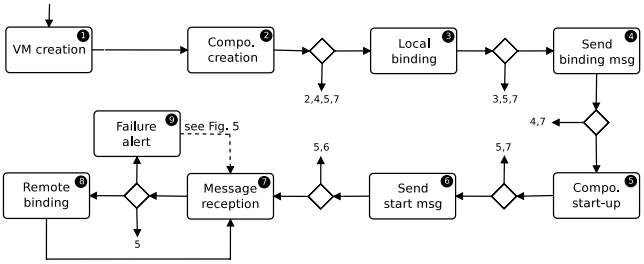

Figure 3: VM Configurator Lifecycle

Figure 4 shows a Message Sequence Chart (MSC) illustrating the start-up of the Web application introduced in Figure 1. First, all three VMs are instantiated by the DM. The corresponding configurators are launched and are aware of the whole application to be deployed (Figure 1). Then, each configurator creates its own components (not illustrated in Figure 4) and sends binding messages as required in the application model. For example, VM3 configurator knows that VM2 needs to connect its JOnAS component to the MySQL component, therefore the VM3 configurator posts a *binding* message with the information needed to connect to the database (*e.g.*, IP address, port number, login and password) to VM2. Upon reception of this binding message, the configurator binds both components. Note that the VM3 configurator can start the MySQL component quite early in this scenario because this component does not require any service from other components (no imports). The VM3 configurator indicates to VM2 that its MySQL component has started. Upon reception of this *start* message, the VM2 configurator starts its JOnAS component, and sends a similar message to VM1. The VM1 configurator finally starts the Apache component and the application becomes fully operational.

2.4 Failures

The protocol presented in this paper is generic in the sense that it only addresses failures that are not specific to the application to be deployed. Such failures are therefore external to the application and can affect both the execution environment (*i.e.*, the virtual and physical infrastructure) and the management system itself. Contrary to internal failures, which are specific to a given application, external failures can be detected and corrected without any knowledge of the application. This paper focuses on three kinds of failures that concern the execution environment:

- failure of an applicative VM: this interrupts the normal execution of the virtual machine, which becomes unusable;

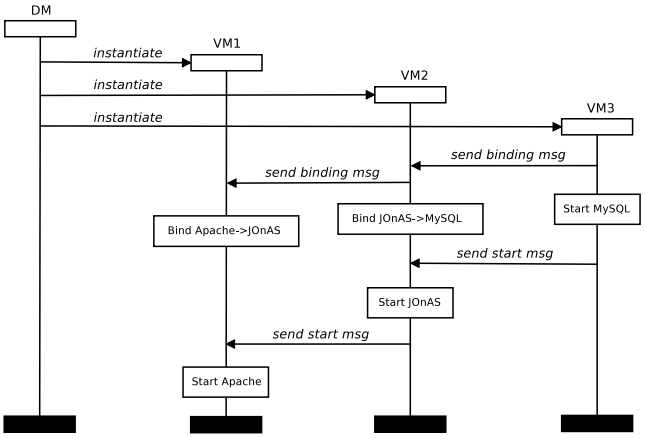

Figure 4: Web Application Start-up Scenario

- failure of a configurator running on a VM: this type of failure alters the configurator's behavior, disrupting all exchanges to and from the configurator;
- transitory network failure: this kind of failure interrupts part of the network services for a finite period. These failures can result in message loss or connection closing, but are not definitive.

Self-repairing the rest of the bootstrapping configurators involved in the deployment system (*e.g.*, the deployment manager) is beyond the scope of this paper. Each case of failure introduced above always involves a VM, thus we will use failure or VM failure indifferently in the remainder of this paper. We also only consider permanent failures: if a transitory failure is detected, it will be treated as a permanent failure.

Failure detection. This detection relies on a *heartbeat* mechanism. As soon as a configurator embedded on an applicative VM is activated, it launches a thread which periodically sends a synchronous beat to the deployment manager. The continuous reception of these beats by the deployment manager indicates that execution of the VM and network are correct. The deployment manager is configured to accept a maximum delay between two beats from a given configurator. Each time it receives a heartbeat, the manager resets the timer associated with that configurator and waits for the next beat. If it does not receive the next beat before the timer expires, it considers that the configurator/VM or network has failed and initiates a repair phase.

Failure repair. When a failure occurs and is detected, the DM creates a new instance of the failed VM, and indicates the identity of the failed VM and the identity of the newly created VM to the other VMs. When a new instance of a VM is created as a result of a VM failure, the new VM must receive *acknowledgement* messages from all the other VMs indicating that they have been informed of its creation. This part of the protocol is crucial to avoid erroneous behavior, *e.g.*, the reception of messages by a VM from an unknown emitter.

We will now focus on the other VMs, that is, those that were instantiated before the VM failed and are still being deployed (Figure 5, where 7 stands for ❼ in Figure 3). Upon reception of a message indicating that a VM has failed and another instance of that VM has been created, the configurator first updates the list of known VM identifiers (❶). Then, it purges its buffers (❷), removing all messages coming from or destined for the failed VM, and updates its current state (❸), moving started components to a stopped state if they are connected to failed components and removing all bindings to failed components. When these updates have been completed, the configurator sends an *acknowledgement* message to the new VM indicating that it is aware of its presence in the application (❹). Finally, it re-sends the binding (❺) and start-up (❻) information details for all remote components (import side) connected to some of its components, and hosted on a re-instantiated VM.

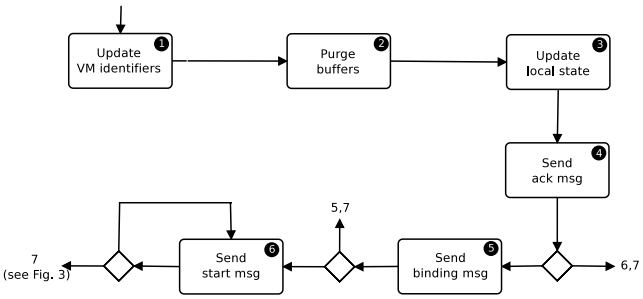

Figure 5: VM Configurator Lifecycle Handling VM Failures

It is worth noting that several VM failures may occur, this can be due to failures of different instances of a single VM or failures of different VMs. A failure can also take place when a VM is already handling a failure involving another VM (cascading failures). If the number of VMs is finite and if there is no cycle of bindings through mandatory imports, the self-deployment protocol terminates successfully: all VMs are instantiated and components will be started.

Example. In Figure 6, we show an example of a VM failure (VM2), occurring when all the VMs have been instantiated and all the components started. When the DM detects VM2's failure, it first creates a new instance of VM2 (VM2') and alerts the other VMs. Upon reception of these messages, both remaining VMs (VM1 and VM3) behave as shown in Figure 5. Thus, the configurator for VM1 changes VM2's identifier, purges its two buffers, stops its Apache component, and unbinds Apache from JOnAS. The VM1 configurator also sends an acknowledgement message to VM2' indicating that it knows it and can receive messages from it. Nothing else is required of VM1, and the VM1 configurator returns to its normal behavior, *i.e.*, ❼ (message reception), as illustrated in Figure 3. In the case of VM3, the configurator for VM3 changes VM2's identifier, purges its two buffers, and sends an acknowledgement message to VM2'. The VM3 configurator also needs to re-send binding information to VM2 and another message indicating that the MySQL component is started. Last but not least, after instantiation, when VM2' has received *ack* messages from all the other VMs, it behaves normally, as presented in Figure 3 and its JOnAS component can be started, as required in the application model.

3. EVALUATION

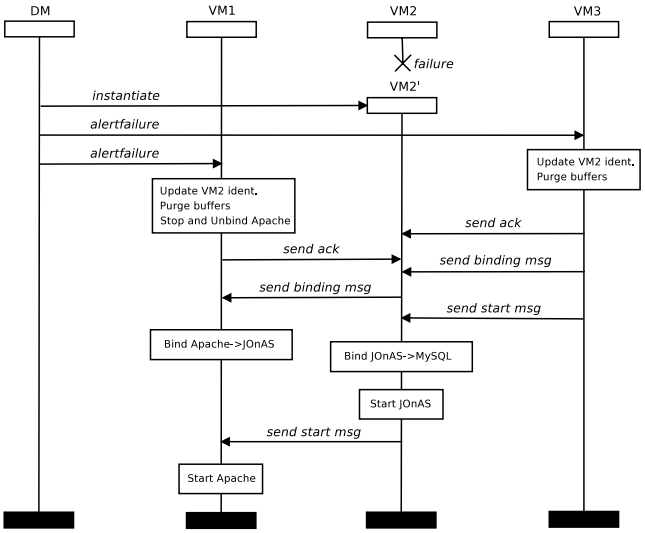

Figure 6: Web Application Failure Scenario

In this section, we first present the formal specification and verification of the protocol. Second, we introduce the Java implementation of the protocol.

3.1 Verification

Specification. For validation purposes, we decided to specify the self-deployment protocol with the LNT value-passing process algebra [6]. We chose LNT because it is adequately expressive to describe data types, functions, and concurrent behaviors. In addition, LNT is one of the input languages for the CADP toolbox [16], which provides a large variety of verification techniques and tools to automatically analyze LNT specifications.

The LNT specification for the self-deployment protocol consists of at least 2,500 lines of code. A part of the specification depends on the input application model, and is therefore automatically generated from a Python script we implemented. For instance, an application model with 6 VMs results in a 3,500-line LNT specification. Data types are used to describe the application model (VMs, components, ports, bindings), messages, buffers, etc. Functions apply to data expressions and are necessary for three kinds of computation: (i) extracting information from the application model, (ii) describing buffers and basic operations on them, (iii) keeping track of the started components to know when another component can be started (when all mandatory imports are satisfied). Processes are used to specify VMs (configurator, input and output buffer), failure injection, and the whole system consisting of interacting VMs possibly failing at some unpredictable point.

Properties. We identified 15 key properties for the protocol. These properties help to verify that architectural invariants are satisfied during protocol execution (prop. 1, 3 below), final objectives are fulfilled (prop. 2, 5, 7 below) or ordering constraints respected (prop. 4, 6 below). Let us give a few examples of such properties, with a particular focus on VM failure occurrences (prop. 3, 4, 5, 6, 7). For some of these properties, we also give their expression in the MCL language [20] used in CADP to formalize temporal properties:

1. There is no cycle of bindings in the component assembly through mandatory imports.

2. All components are eventually started.

3. No component can be started before the components it depends on through mandatory imports.

   ```
   [
     true* . {STARTCOMPO ?vm:String !"JOnAS"} .
     (¬'{FAILURE !.*}')* .
     {STARTCOMPO ?vm2:String !"MySQL"}
   ] false
   ```

   In the running example, the JOnAS component is connected to the MySQL component through a mandatory import, therefore we will never find a sequence where JOnAS is started before MySQL except in case of a failure. This property is automatically generated from the application model because it depends on the component names and bindings in the model.

4. After a VM fails, all other VMs are informed of that failure.

5. Each VM failure is followed by re-creation of that VM.

   ```
   library actl.mcl end library
   [ true* . '{FAILURE ?vm:String}' ]
     AU_A_A(true, not '{FAILURE !vm}',
                    '{CREATEVM !vm !.*}', true)
   ```

   This property is formalized making use of action CTL patterns [11].

6. Two failures (same VM) are always separated by VM creation.

7. A sequence exists resulting in protocol termination with no failure.

   ```
   <true* . (¬'{FAILURE ?vm:String}')* . 'FINISH'> true
   ```

   Termination is made explicit in the specification using the special `FINISH` action.

Experiments. To verify this specification, we used a database of 170 application models. For each input model, we used CADP exploration tools to generate the Labeled Transition System (LTS) corresponding to all possible executions of the protocol for this input. Then, we used the CADP model checker (Evaluator) to verify that this LTS satisfies the 15 properties of interest.

Table 1 summarizes the results obtained for our running example, with an increasing number of possible failures ($|F|$). We give the size of the LTS generated (before and after strong reduction) using CADP by enumerating all the possible executions of the system, as well as the time to obtain this LTS (generation and reduction) and verify all 15 properties. Experiments were carried out on a Xeon W3550 (3.07GHz, 12GB RAM) running Linux.

It is worth observing that increasing the number of failures induces a gradual growth in the LTS size and computation time. Since we use enumeration techniques, when there are, *e.g.*, four failures during the deployment process, it means

that all possible configurations are attempted (*e.g.*, cascading failures of different VMs, successive failures of the same VM, etc.) and all these executions appear in the corresponding LTS. The other parameter increasing these numbers is the number of VMs, which generates more parallelism in the system, and the number of remote bindings in the application model, which augments the number of messages exchanged between VMs. We were able to analyze applications with up to 6 VMs and 10 remote bindings. However, huge applications were not required to detect issues in the protocol: most problems are usually found on small examples exposing corner cases.

	LTS (states/transitions)		Time (m:s)	
|F|	raw	minimized	Gen.	Verif.
0	233/565	233/565	0:5	0:10
1	6,196/16,272	3,125/8,205	0:13	0:21
2	61,548/175,796	21,980/62,042	0:14	0:31
3	349,364/1,045,883	100,008/293,555	0:48	7:19
4	1,489,515/4,601,552	366,269/1,097,990	3:34	26:29
5	5,381,794/17,035,375	1,206,934/3,654,952	40:13	hours

Table 1: Experimental Results

Issues detected. The formal verification of the protocol using model checking techniques helped to refine certain parts of the protocol and to detect subtle bugs. For instance, one important architectural invariant states that a started component cannot be connected to a stopped component. However, we encountered cases where optional imports violated this invariant, resulting in started components connected to and therefore submitting requests to stopped components. This problem was detected thanks to an extension of property 3 stating that "*a component cannot be started and connected to another component, if that component has not been started beforehand*". This invariant can only be preserved in the absence of failures: we cannot prevent a started component from being connected to a failed component.

3.2 Implementation

This section describes the principles and the assessment of the Java implementation of the protocol[1].

VAMP Principles. We developed a reference implementation of the reliable self-deployment protocol using the Virtual Applications Management Platform (VAMP) system [13]. VAMP is a generic solution dedicated to the self-deployment of distributed applications in the cloud. In its first version, VAMP relied on an unreliable deployment process [14]. Thus, the goal of this implementation is to replace the original deployment protocol by the reliable protocol presented in Section 2 so as to enable VAMP to deploy an application in the cloud, in finite time, even if a finite number of VM or network failures occurs[2].

When receiving a deployment request from a user, VAMP creates a new VM in which a deployment manager is instantiated. This DM is in charge of deploying the application and therefore bootstraps the deployment by generating virtual images participating in the application and instantiating them as VMs in one or several IaaS platforms. The configurators are included in the virtual images at the generation stage. Once an applicative VM has completely booted, the corresponding configurator starts applying the self-deployment protocol by instantiating, configuring, and activating the local applicative software components.

All the participants in the deployment of a given application (*i.e.*, a DM and the configurators) communicate through an asynchronous reliable distributed message oriented middleware (MOM), the AAA bus [3]. This middleware interconnects agents. An agent is a plain old Java object (POJO) that runs in a Java Virtual Machine. Each agent can send messages to other agents. When receiving a message, an agent behaves according to an event-action model. In the VAMP system, each management entity is an agent of the AAA middleware. AAA provides some noticeable properties. First, it is distributed within the agents, thus avoiding any centralized mechanism that might suffer from the bottleneck effect. Second, the reaction of an agent to an event is atomic, *i.e.*, it is entirely executed. This mechanism relies on the agent's state persisting before and after each reaction. Third, due to the combination of message persistence and the asynchronous programming model provided by AAA, any agent is assured of the delivery of the messages it sends, even when a finite number of transitory failures occur. However, there is no guarantee of the time at which the messages will be dealt with by the target agent. Finally, AAA preserves message ordering, *i.e.*, messages are received in the same order as they were sent.

Assessment. The evaluation process aims to measure the time to deploy a three-tier Web application (Fig. 1) while randomly injecting a number of failures that could affect any applicative VM. Each test was repeated 10 times. The application deployment was considered completed when all the software components were configured and started.

The underlying IaaS platform used to carry out these tests is an OpenStack Essex running on Linux Ubuntu 12.04 LTS 64 bits. It is deployed on a cluster of IBM HS22 blades (2 Intel Xeon E5504 3GHz quad-core, 32GB memory, 292 GB HDD) interconnected with a Gigabit ethernet network. Each computer node runs a KVM hypervisor to instantiate the virtual machines.

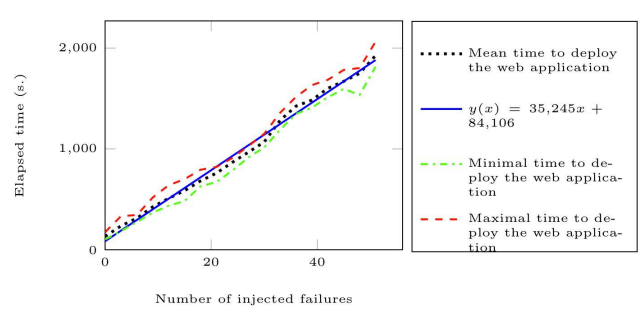

Figure 7: Time to Deploy a Three-tier Web Application with VAMP while Injecting Failures

Figure 7 shows that the time to deploy the Web application increases linearly in accordance with the number of

[1]Although the protocol adopts a decentralized design, its capacity to deal with large scale architecture was previously discussed in [14]. This paper only focuses on its reliability.
[2]Although the self-deployment protocol was implemented within a system dedicated to cloud applications (namely VAMP), it was also used within physical runtime environments providing on-demand machine allocation like grids.

failures encountered. $f(x) = 35{,}245x + 84{,}106$ is the trend line associated with the mean time measured to deploy an application in the presence of x failures[3]. This equation highlights the 58% gain in time to deployment introduced by the reliable self-deployment protocol compared to its unreliable version. Effectively, replacement of a faulty VM by the protocol only induces about 35 seconds' delay compared to the 84 seconds required to re-deploy a full instance of the Web application.

4. RELATED WORK

We first focus on existing approaches relying on ADL-based approaches for deploying software applications. [17, 7] propose languages and configuration protocols for distributed applications in the cloud. SmartFrog [17] is a framework for creating configuration-driven systems. It has been designed with the express purpose of making the design, deployment and management of distributed component-based systems simpler and more robust. [7] adopts a model driven approach with extensions of the *Essential Meta-Object Facility* (*EMOF*) abstract syntax to describe a distributed application, its requirements towards the underlying execution platforms, and its architectural constraints (*e.g.*, concerning placement and collocation). Regarding the configuration protocol, particularly the distributed bindings configuration and the activation order of components, contrary to us, [7] does not work in a decentralized fashion, and this harms the scalability of applications that can be deployed.

[4] presents the design and implementation of an autonomic management system, TUNe. The main principle is to wrap legacy software pieces into components in order to administrate a software infrastructure as a component architecture. The authors also introduce high-level formalisms for the specification of deployment and management policies. This management interface is mainly based on UML profiles for the description of deployment schemas and the description of reconfiguration state diagrams. A tool for the description of wrapper is also introduced to hide the details of the underlying component model. ProActive [2] is a Java-based middleware (programming model and environment) for object and component oriented parallel, mobile and distributed computing. ProActive provides mechanisms in order to further help in the deployment and runtime phases on all possible kind of infrastructures, notably secure grid systems. ProActive is intended to be used for large scale grid applications. However, it does not handle fault occurrence and repair mechanisms.

[21] introduces Eucalyptus, an academic open source software framework for cloud computing that implements a IaaS solution, giving users the ability to run and control VM instances deployed across a variety of physical resources. Eucalyptus is a convenient solution for automated provisioning of virtualized hardware resources and for executing legacy applications. On the other hand, this platform has not been designed for monitoring and particularly deploying such applications. [9] presents AppScale, an open source extension of the Google AppEngine (GAE) PaaS cloud technology. These extensions facilitate distributed execution of GAE applications over virtualized cluster resources, including IaaS cloud systems such as Amazon's EC2 and Eucalyptus. AppScale implements a number of different components that facilitate deployment of GAE applications using local (non-proprietary) resources. This solution has a specific focus on the deployment of Web applications whose code conforms to very specific APIs (*e.g.*, no Java threads).

[13, 23, 1] present protocols that automate the configuration of distributed applications in cloud environments. These protocols work in a decentralized way as well, but do not support the possible occurrence of failures, nor the possibility to repair the application being deployed when a failure occurs. Another recent related work [15] presents a system that manages application stack configuration. It provides techniques to configure services across machines according to their dependencies, to deploy components, and to manage the life cycle of installed resources. This work presents some similarities with ours, but [15] does not focus on composition consistency, architectural invariants preservation, or robustness of the reconfiguration protocol.

[18] describes a peer-to-peer architecture to automatically deploy services on cloud infrastructures. The architecture uses a component repository to manage the deployment of these software components, enabling elasticity by using the underlying cloud infrastructure provider. The main added value of this framework is the incorporation of virtual resource allocation to the software deployment process. This framework is one the closest solution to ours. Yet the centralized architecture and the absence of fault-tolerance mechanisms (as all approaches presented in this section) are two important differences.

Last but not least fault tolerance and system reliability has been the subject of many works and studies in the last decades [26]. A system may fail first because of incorrect specification, incorrect design, design flaws, poor testing, or undetected fault. In such a case, a possible solution aims at improving system design and development using static analysis or specific programming models [22]. Another reason causing failures is the environment (*e.g.*, network, human intervention), aging component, operator errors, or hardware failure. A classic approach intends to tolerate faults through redundancy techniques, either active ones (applying redundancy on processes) or passive ones (applying redundancy on data), see *e.g.*, [19, 24, 8, 25, 12]. Here we preferred a lightweight approach for failure detection and recovery mechanisms, which was simple to develop and sufficient for ensuring fault-tolerance of our deployment solution.

5. CONCLUDING REMARKS

In this paper, we have presented a self-deployment protocol that aims to configure a set of software components distributed over a set of virtual machines. This protocol works in a fully automated way and in a decentralized fashion. To the best of our knowledge, this protocol is the first deployment protocol supporting VM failures, *i.e.*, the protocol not only detects failures, but also creates a new instance for each failed VM, and restores a stable state for all the other VMs. The protocol always succeeds in starting all the components hosted on the different VMs, even in case of multiple failures. The protocol was formally specified and validated using up-to-date verification tools. The protocol was implemented in Java and applied to industrial applications for evaluation purposes.

A first perspective is to make use of co-simulation techniques to ensure that the specification and implementation conform to one another. Such techniques are not always

[3]The linear correlation ratio of this trend is 0.9933.

required: when a bug is found during the specification analysis, it is reported, and in many cases, this is a real bug, *i.e.*, a bug existing in the implementation. Co-simulation techniques would reduce the number of divergences between the specification and the implementation, and this would avoid reporting bugs that are in fact only specification errors. Second, we plan to consider failures of the deployment manager and propose a replication system to ensure reliability of this part of the deployment system. Our last perspective is to extend the protocol and widen the role of deployment managers, allowing them to dynamically reconfigure cloud applications, *e.g.*, by removing or replacing a deployed VM. As a side effect, the configurator's behavior would need to be extended to incorporate those new features.

Acknowledgements. This work was supported by the OpenCloudware project (2012-2015), which is funded by the French *Fonds national pour la Société Numérique* (FSN), and is supported by *Pôles* Minalogic, Systematic, and SCS.

6. REFERENCES

[1] R. Abid, G. Salaün, F. Bongiovanni, and N. De Palma. Verification of a Dynamic Management Protocol for Cloud Applications. In *Proc. of ATVA'13*, volume 8172 of *LNCS*, pages 178–192. Springer, 2013.

[2] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer, 2006.

[3] L. Bellissard, N. D. Palma, A. Freyssinet, M. Herrmann, and S. Lacourte. An Agent Platform for Reliable Asynchronous Distributed Programming. In *Proc. of SRDS'99*, pages 294–295. IEEE Computer Society, 1999.

[4] L. Broto, D. Hagimont, P. Stolf, N. D. Palma, and S. Temate. Autonomic Management Policy Specification in Tune. In *Proc. of SAC'08*, pages 1658–1663. ACM, 2008.

[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.*, 36(11-12):1257–1284, 2006.

[6] D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 2011.

[7] C. Chapman, W. Emmerich, F. G. Márquez, S. Clayman, and A. Galis. Software Architecture Definition for On-demand Cloud Provisioning. In *Proc. of HPDC'10*, pages 61–72. ACM Press, 2010.

[8] M. Chérèque, D. Powell, P. Reynier, J.-L. Richier, and J. Voiron. Active Replication in Delta-4. In *Proc. of FTCS'92*, pages 28–37. IEEE Computer Society, 1992.

[9] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. AppScale: Scalable and Open AppEngine Application Development and Deployment. In *Proc. of CloudComp'09*, volume 34 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 57–70. Springer, 2010.

[10] B. Dillenseger. CLIF, a Framework based on Fractal for Flexible, Distributed Load Testing. *Annales des Télécommunications*, 64(1-2):101–120, 2009.

[11] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in Property Specifications for Finite-State Verification. In *Proc. of ICSE'99*, pages 411–420. ACM, 1999.

[12] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[13] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-Configuration of Distributed Applications in the Cloud. In *Proc. of CLOUD'11*, pages 668–675. IEEE Computer Society, 2011.

[14] X. Etchevers, T. Coupaye, F. Boyer, N. D. Palma, and G. Salaün. Automated Configuration of Legacy Applications in the Cloud. In *Proc. of UCC'11*, pages 170–177. IEEE Computer Society, 2011.

[15] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A Deployment Management System. In *Proc. of PLDI'12*, pages 263–274. ACM, 2012.

[16] H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of TACAS'11*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011.

[17] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The SmartFrog Configuration Management Framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, 2009.

[18] J. Kirschnick, J. M. A. Calero, P. Goldsack, A. Farrell, J. Guijarro, S. Loughran, N. Edwards, and L. Wilcock. Towards an Architecture for Deploying Elastic Services in the Cloud. *Softw., Pract. Exper.*, 42(4):395–408, 2012.

[19] L. Lamport. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks*, 2:95–114, 1978.

[20] R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.

[21] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-Source Cloud-Computing System. In *Proc. of CCGRID'09*, pages 124–131. IEEE Computer Society, 2009.

[22] G. K. Saha. Software Based Fault Tolerance: A Survey. *Ubiquity*, 2006.

[23] G. Salaün, X. Etchevers, N. D. Palma, F. Boyer, and T. Coupaye. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. In *Proc. of SAC'12*, pages 1278–1283. ACM Press, 2012.

[24] F. B. Schneider. Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[25] R. van Renesse and R. Guerraoui. Replication Techniques for Availability. In *Replication: Theory and Practice*, volume 5959 of *LNCS*, pages 19–40. Springer, 2010.

[26] W. Zamojski and D. Caban. Introduction to the Dependability Modeling of Computer Systems. In *Proc. of DepCoS-RELCOMEX'06*, pages 100–109. IEEE Computer Society, 2006.