SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **00**:1–19
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Reliable Self-deployment of Distributed Cloud Applications

Xavier Etchevers[1,*], Gwen Salaün[2], Fabienne Boyer[2], Thierry Coupaye[1], and Noel De Palma[2]

[1] *Orange Labs, Meylan, France and* [2] *University of Grenoble Alpes, France*

SUMMARY

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 conform to 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. Copyright © 2016 John Wiley & Sons, Ltd.

Received October 2015.

KEY WORDS: Distributed applications, software components, automatic deployment, robustness and reliability.

1. INTRODUCTION

Cloud computing emerged a few years ago as a new approach based on virtualized hardware resources and software applications distributed 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 combines various 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, IaaS-based applications (or cloud application for short) are distributed applications composed of a set of interconnected execution units called software elements, running on separate virtual machines. 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

*Correspondence to: Xavier Etchevers, 28 Chemin du Vieux Chêne, 38240, Meylan, France.

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 that the application reaches 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 have focused on this issue, *e.g.*, [1, 2, 3, 4] (see Section 5 for a detailed presentation of existing results). The contribution presented in this paper goes one step further than these works, by designing a 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 application architecture, that is, the interconnected components and their distribution over the virtual machines. 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 [5]. LNT is one of the input languages of the CADP verification toolbox [6], 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. 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 [7].

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 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 reliable self-deployment protocol. Section 3 presents the formal specification and verification tasks. Section 4 details implementation and evaluation aspects. In section 5, related works are discussed before concluding in section 6.

DOI: 10.1002/spe

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 application 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 application architecture level, which is based on the Fractal component model [8]. 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.

It is worth noting that these models can be manually achieved or generated automatically using complementary algorithms, such as those existing in the Vulcan planification system [9] or the BtrPlace manager [10]. In the latter case, SLA requirements or other placement constraints can be taken into account. The deployment protocol we present in the rest of this paper can be viewed as the (low-level) mechanics to configure distributed cloud applications, independently of the technique used for computing the target application model.

From a user perspective, the application model is the main part of our approach, which needs to be provided by the user in order to deploy an application. Our tool support also requires a *wrapper* for each component handled in the application. A wrapper implements the behavior of the component and exposes a set of interfaces describing the primitives for manipulating the corresponding applicative software element (*e.g.*, start, stop, export). In other words, a wrapper consists in the implementation class of a component. Thus, according to the effort to develop reusable code, wrappers can be specific to one application or can be reused for deploying different applications requiring similar type of components. In the rest of Section 2, we present the deployment protocol, which takes as input an application model and automatically deploys the corresponding VMs and configurators in charge of actually instantiating and connecting the components specified in the application model.

In the remainder of this paper, we will use as running example a three-tier Web application with a cluster of application servers. Figure 1 gives the application model, which consists of four (or more) VMs. The first one (VM1) hosts a front-end HTTP server (Apache). Then we have a cluster of JEE application servers (two VMs, VM2 and VM3, in Figure 1 for illustration) where each VM hosts a JEE application server (JOnAS). The fourth VM (VM4) corresponds to the database management system (MySQL). These components are connected through remote bindings (*e.g.*, Apache bound to JOnAS) on mandatory (m) and optional (o) imports.

DOI: 10.1002/spe

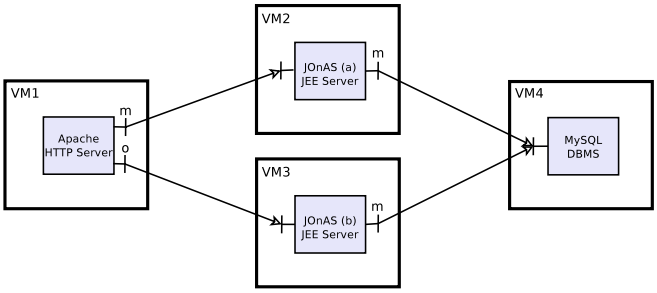

Figure 1. A Three-tier Web Application Model

2.2. *Participants*

The self-deployment protocol involves two kinds of participants (Fig. 2): a deployment manager and a set of VM configurators (shortened as configurators). The deployment manager (DM) guides the application's configuration by instantiating VMs, as described in the target application model, and creating a new instance of a VM when a failure occurs. Each VM in the distributed application is equipped with a configurator. Each configurator takes as input the target application and is responsible for connecting bindings and starting components once the VM instance has been created by the deployment manager. The protocol is therefore generic in the sense that the deployment manager and the configurators do not depend on the application model, and can thus deploy any application that can be described using the model presented in Section 2.1.

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 infinitely messages to buffers.

We assume that the asynchronous communication model we rely on is reliable in the sense that no messages are lost, even in the presence of a finite number of transitory failures. Moreover, the message ordering is preserved, *i.e.*, messages are received in the same order by a target configurator, as they were sent by a source configurator. Last, there is no guarantee of the time at which the messages will be handled by the target configurator. We will see in Section 4 how these properties are guaranteed in practice.

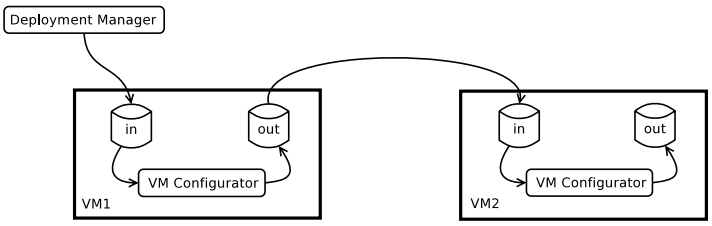

Figure 2. Participants: Deployment Manager and VM Configurators

2.3. *Deployment*

Protocol execution is driven by the configurators embedded on each VM. All configurators evolve in parallel, and each of them is aware of the application to be deployed. Each manager executes 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,

DOI: 10.1002/spe

etc.). Diamonds stand for choices, and each arrow outgoing from a choice either enters another box or is annotated with 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, the DM creates an image of this VM (❶) and then initiates the VM execution. Each VM is equipped with a configurator, which starts when the VM initiates its execution. A configurator is responsible for binding ports as described in the application model and for starting components in a specific order: a component can be started only if all its mandatory imports are satisfied. The configurator does not depend on any application model and is able to deploy any concrete application.

VM and component start-up. We now explain how a newly instantiated VM binds its ports and starts the components to be deployed on this VM. 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). As for remote bindings, the configurator performs 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, the VM in charge of that component 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 if there is no cycles of bindings over mandatory imports. A cycle of bindings is possible if at least one optional import is involved in those bindings. Failure handling (❾ in Figure 3) will be detailed in the next subsection.

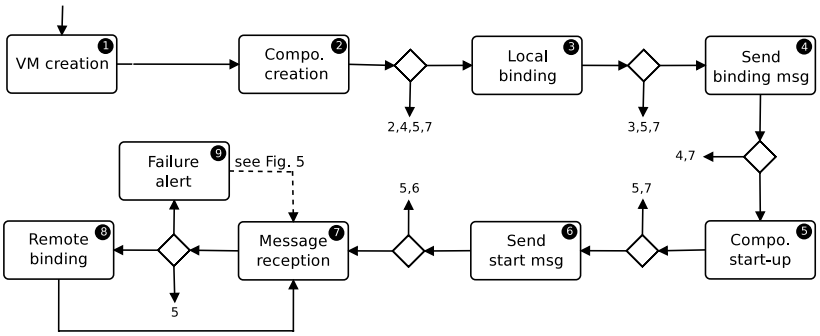

Figure 3. VM Configurator Lifecycle

Example. Figure 4 shows a Message Sequence Chart (MSC) illustrating a specific scenario for the start-up of the Web application introduced in Figure 1. First, all four 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, the VM4

DOI: 10.1002/spe

configurator knows that VM3 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 VM3. Upon reception of this binding message, the configurator binds both components. Note that the VM4 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 VM4 configurator indicates to VM2 and VM3 that its MySQL component has started. Upon reception of this *start* message, the VM3 configurator for instance starts its JOnAS component, and sends a similar message to VM1. The VM1 configurator starts the Apache component when it has received a *start* message for the JOnAS component from VM2, because VM1 is connected to that component on a mandatory import, whereas the import is optional for the connection to the JOnAS component of VM3. The application is fully operational.

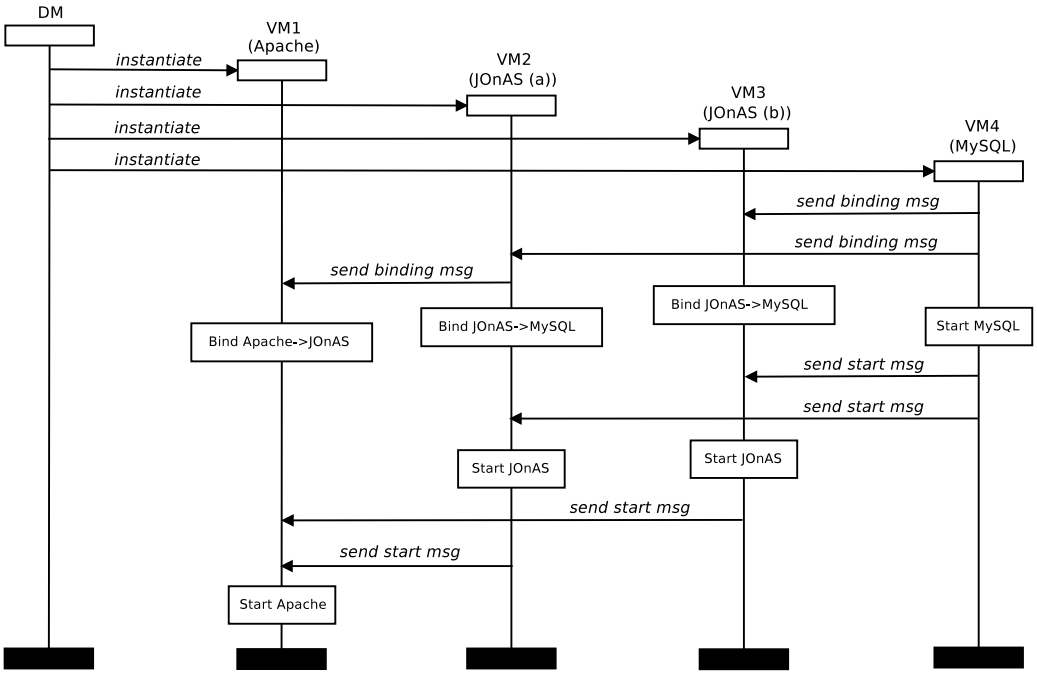

Figure 4. Web Application Start-up Scenario

2.4. Failures

The protocol presented in this paper is generic in the sense that it 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:

- crash-stop failure of an applicative VM: this interrupts the normal execution of the virtual machine, which becomes unusable without any possible recovery;
- crash-stop 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;
- byzantine 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 transitory (not definitive).

DOI: 10.1002/spe

Self-repairing the rest of the bootstrapping participants involved in the deployment system (*e.g.*, the deployment manager) is beyond the scope of this paper: it could be addressed thanks to well-known replication techniques, see Section 5 for more details. Each case of failure introduced above always involves a VM, thus we will use failure or VM failure indifferently in this paper.

Failure detection. This detection relies on a synchronous *heartbeat* mechanism. As soon as a configurator embedded on an applicative VM is activated, it launches a thread which periodically and synchronously sends a signal or beat to the deployment manager. The continuous reception of these beats by the deployment manager indicates that the execution of the VM and network are correct. The heartbeat mechanism is unidirectional (no *ack* messages). 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 watchdog 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[†]. We use sampling techniques and simulation for choosing timer values. A limit of this solution is that some failure may be detected due to an additional network delay whereas there is no real failure, but this case does not show up often in practice. However, the repair mechanism can deal with such false positive cases (see below).

Failure repair. When a failure occurs and is detected, the DM tries to recover by replacing the VM whose heartbeat was interrupted. Due to the lack of accuracy for determining the type of failure, the DM needs to deal with false positive cases that result from potential byzantine network failures. To do so, it first tries to delete the VM suspected to be faulty. Then it 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 that 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.

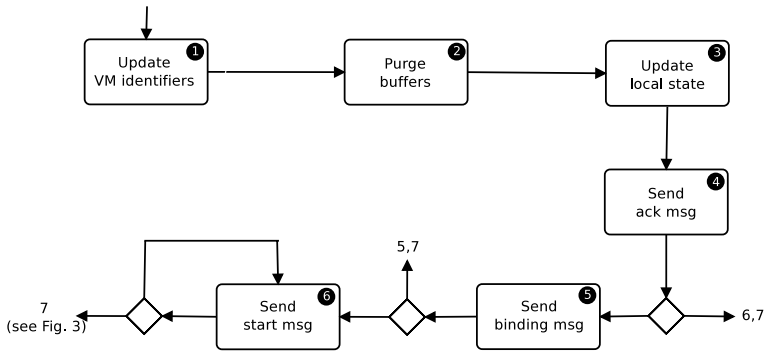

Figure 5. VM Configurator Lifecycle Handling VM Failures

[†]The principle of this mechanism is to detect failures, not to determine their type.

DOI: 10.1002/spe

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 eventually terminates successfully: all VMs are instantiated and components will be started.

Example. In Figure 6, we show an example of 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, all remaining VMs (VM1, VM3, and VM4) 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 and VM4, each configurator changes VM2's identifier, purges its two buffers, and sends an acknowledgement message to VM2'. The VM4 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. As a result, its JOnAS component can be started and, as a side effect, the VM1 configurator can also start the Apache component.

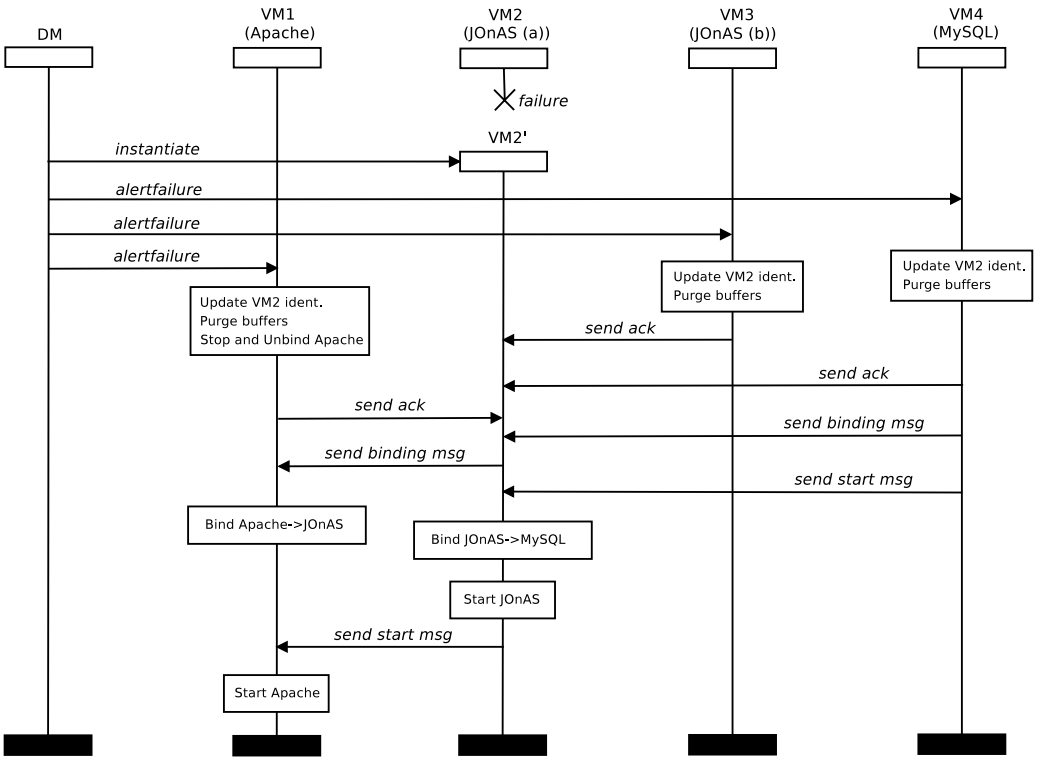

Figure 6. Web Application Failure Scenario

3. SPECIFICATION AND VERIFICATION

In this section, we present the formal specification and verification of the protocol.

DOI: 10.1002/spe

3.1. Specification

For validation purposes, we specified the self-deployment protocol with the LNT value-passing process algebra [5]. We chose LNT because it is expressive enough to describe data types, functions, and concurrent behaviors. In addition, LNT is one of the input languages of the CADP toolbox [6], 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.

For illustration purposes, we show below an excerpt of the LNT specification corresponding to the deployment manager process, generated for an application model involving four VMs (`VM1`, `VM2`, `VM3`, and `VM4`). The `deployer` process defines first the list of actions used in its behavior (`CREATEVM`, `FINISH`, `FAILURE`, etc.). Actions can be typed (with the types of their parameters), but this is optional and we use the keyword **any** in that case. After the declaration of a few variables, we associate using function `assign_id_vm` an identifier to each VM image (*e.g.*, identifier `1` for `VM1`) and stores in the `counter` variable the next identifier to be used in case of failure. Then, we use the LNT parallel composition **par** (pure interleaving here) for instantiating the four VMs. Then, the process engages an active looping which either stops when the whole deployment terminates (`FINISH`) or when a failure occurs (`FAILURE`). In the latter case, a new identifier is associated to the failed VM (`newid`), a new instance of that VM is created (`CREATEVM`), and the partner VMs alerted (`ALERTFAILURE`i) along with the identifier of the newly created instance of the failed VM.

```
process deployer [CREATEVM:any, FINISH:any, FAILURE:any,
    ALERTFAILUREVM1:any, ALERTFAILUREVM2:any, ... ] (lid: STID) is
var counter:Nat, vmidlist: TVMIDSet, newid: Nat in
  vmidlist:=assign_id_vm(lid,1); counter:=len_stid(lid)+1;
  par
    CREATEVM (!VM1, !get_id_vm(VM1,vmidlist), !true, !vmidlist)
  ||
    CREATEVM (!VM2, !get_id_vm(VM2,vmidlist), !true, !vmidlist)
  ||
    CREATEVM (!VM3, !get_id_vm(VM3,vmidlist), !true, !vmidlist)
  ||
    CREATEVM (!VM4, !get_id_vm(VM4,vmidlist), !true, !vmidlist)
  end par;
  loop theloop in
    select
      FAILURE (!VM1 of TID);
      vmidlist:=update_id_vm(VM1,vmidlist,counter);
      newid:=counter; counter:=counter+1;
      CREATEVM (!VM1, !get_id_vm(VM1,vmidlist), !false, !vmidlist);
      ALERTFAILUREVM2 (!VM1, !newid); ALERTFAILUREVM3 (!VM1, !newid);
      ALERTFAILUREVM4 (!VM1, !newid)
    [] ... (* similar code for VM2 *)
    [] ... (* similar code for VM3 *)
    [] ... (* similar code for VM4 *)
    [] FINISH; break theloop
    end select
  end loop
end var
end process
```


DOI: 10.1002/spe

3.2. Properties

We identified 15 key properties that must be respected by the protocol. These properties help to verify that architectural invariants are satisfied during the 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 formulation in the MCL language [11], the temporal logic used in CADP. MCL is an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators.

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.

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

   In the running example, the JOnAS (a) 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 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 [12].
6. Two failures of a same VM are always separated by a 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.

3.3. Experiments

To verify this specification, we used a database of 210 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 I summarizes the results obtained for three versions of our running example (with 1, 2, and 3 JOnAS servers, resp.), with an increasing number of possible failures ($|F|$). We give the size of the LTS generated (before and after strong reduction [13]) 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 an Intel Xeon X5560 (2.80GHz, 148GB 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

DOI: 10.1002/spe

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. Large applications are not necessary to detect issues in the protocol, most problems were found on small examples exposing corner cases.

Id	Application model			|F|	LTS (states/transitions)		Time (m:s)	
	|VM|	|Comp|	|Bd|		raw	minimized	Gen.	Verif.
1	3	3	2	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	>3h
2	4	4	4	0	5,308 / 18,750	5,287 / 18,723	0:13	1:44
				1	1,101,598 / 4,818,992	355,016 / 1,326,368	12:58	104:38
				2	30,828,377 / 139,116,259	6,160,018 / 23,786,583	>3h	>3h
3	5	5	6	0	149,721 / 676,715	148,105 / 672,259	14:28	71:22
				1	1,707,075 / 7,223,859	956,467 / 5,166,328	121:11	>3h

Table I. Experimental Results

3.4. 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. We particularly present two of them in the rest of this section.

First, there was a problem in the way local components are started during the protocol execution. After reading a message from the input buffer, the configurator must check all its local components, and start those with mandatory imports bound to started components. However, one traversal of the local components is not enough. Indeed, launching a local component can make other local components startable. Consequently, starting local components must be done in successive iterations, and the algorithm stops only when no other components can be started. If this is not implemented as a fix point, the protocol does not ensure that all components involved in the architectue are eventually started. This bug was detected with property 2 checking that "*all components are eventually started*".

Second, 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.

4. IMPLEMENTATION AND EVALUATION

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

4.1. VAMP Principles

We developed a reference implementation of the reliable self-deployment protocol using the Virtual Applications Management Platform (VAMP) system [3]. VAMP is a generic solution dedicated to the self-deployment of arbitrary distributed applications in the cloud. In its first version, VAMP relied on an unreliable deployment process. The current version of VAMP implements the reliable

DOI: 10.1002/spe

deployment protocol presented in Section 2. This implementation enables VAMP to deploy an application in finite time, even if a finite number of VM or network failures occurs during the deployment phase. VAMP takes as input a model of the application (see Section 2.1) and assumes that each component of the application is equipped with a wrapper, which is an interface exhibiting the primitives for manipulating the component. This wrapper either exists or must be provided by the developer.

When receiving a deployment request from a user, VAMP creates a new VM in which a deployment manager (DM) is instantiated. The DM is in charge of deploying the application. Therefore, it 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. It also interacts with the other configurators to exchange information of interest with respect to the deployment status of the other VMs.

All the management entities participating in the deployment of a given application (*i.e.*, the dedicated DM and the configurators) communicate through an asynchronous distributed message oriented middleware (MOM), the AAA bus [14]. This bus is reliable in the sense that no messages are lost. This middleware interconnects agents (configurators and deployment manager here). 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.*, the reaction is entirely executed or not at all. 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 occurs. 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 (reliable communication), *i.e.*, messages are received in the same order by a given target agent, as they were sent by a given source agent.

4.2. Assessment

The evaluation process aims at measuring the impact on the time to deploy a three-tier Web application (Fig. 1) while randomly injecting a number of failures. Although the protocol adopts a decentralized design, its capacity to deal with large scale architectures was previously discussed in [15] and will not be tackled in these experiments, which focus on the reliability of the deployment. We will not compare the centralized version of the protocol with its current distributed version, which is more efficient in terms of message exchange and overall performance, see [16] for details.

In the rest of this section, we present two kinds of experiments. In the first case, we measure the benefit, in terms of time to deploy, introduced by the reliable self-deployment protocol compared to its previous non-reliable version. In a second case, we measure the time to deploy an application with multiple replicas of the applicative tier using the reliable self-deployment protocol.

For both experiments, each measure results from the average value of 10 iterations of the same experiment. The application deployment was considered completed when all the software components were configured and then, simultaneously started. We simulated the failure of an applicative VM by shutting it down through the infrastructure (*i.e.*, IaaS) API.

The underlying platform used to carry out these tests is an OpenStack Havana IaaS solution 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 computing node runs a KVM hypervisor to instantiate the virtual machines.

Time efficiency of the reliable self-deployment protocol compared to its non-reliable version. In this first experiment, the Web application consists of three instances, *i.e.*, one for each tier (Apache

DOI: 10.1002/spe

HTTP server, JOnAS JEE server, MySQL DBMS). Each instance of each tier runs on a dedicated VM. We randomly inject failures on any of the applicative VMs while proceeding to the initial deployment of the application. The number of injected failures varies from 0 to 39.

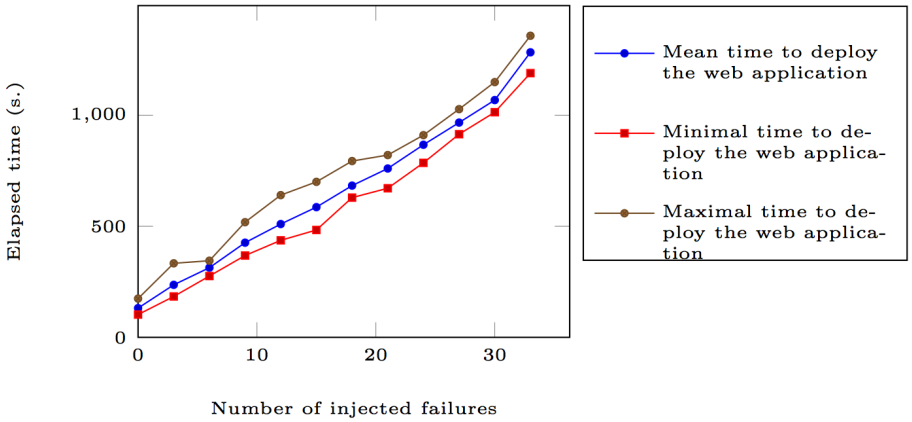

Figure 7. Time to Deploy a Three-tier Web Application with VAMP while Injecting Failures

Figure 7 shows that the time to deploy the Web application increases linearly in accordance with the number of failures encountered. $f(x) = ax + b$ is the trend line associated with the mean time measured to deploy an application in the presence of x failures, where $a = 35{,}245$ and $b = 84{,}106$[‡]. In comparison, the use of the previous non-reliable version of the protocol that requires to redeploy the entire application when a failure occurs in an applicative VM, follows the trend function $g(x) = (b+1)x$ where x stands again for the number of failures encountered and $b = 84{,}106$. Therefore, both equations highlight the gain in deployment time introduced by the reliable self-deployment protocol compared to its unreliable version. This gain linearly depends on the number of failures and can be estimated as $G(x) = 1 - \frac{f(x)}{g(x)}$. Its value tends to $\lim_{x \to +\infty} G(x) = 1 - \frac{a}{b} = 58\%$. In other words, the 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.

Time to deploy an application with several replicas of the applicative tier. In the second experiment, the Web application is built up with:

- one instance of the presentation tier (Apache HTTP server);
- one instance of the database tier (MySQL DBMS);
- a farm of 5 instances of the business tier (JOnAS JEE server).

Each instance of each tier runs on a dedicated VM. We measure the benefits of the replication, while randomly injecting failures on a subset of the farm of VMs member of the business tier (JOnAS servers). The number of injected failures varies from 0 to 10 whereas the rate of affected VMs in the farm varies from 20% to 100% of the farm (*i.e.*, from 1 to all 5 VMs).

It is worth noting that Figure 8 exhibits several noticeable behaviors. First, similarly to the first experiment, the deployment duration linearly depends on the number x of injected failures. However, the presence of steps whose size varies according to the number of potentially affected VMs in the farm, demonstrates that the deployment duration also depends linearly, in a non continuous way, on the number v of potentially affected VMs in the farm. Such relationships highlights the parallelism capacity for the reliable self-deployment protocol to manage many failures occurring on different applicative VMs. The equation that models the deployment duration in such case is the following: $f'(x, v) = a' \lfloor \frac{x+v-1}{v} \rfloor + b'$, where a' and b' are two constants. Experimentally,

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

DOI: 10.1002/spe

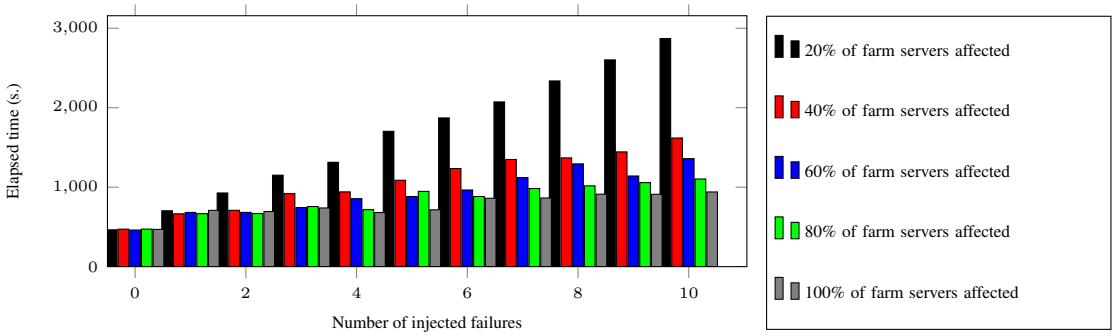

Figure 8. Time to Deploy a Three-tier Web Application (Farm of Servers) with VAMP while Injecting Failures

we obtained the following rounded values: $a' = 239$ and $b' = 444$ with a mean correlation factor of 0,992.

The previous equation models two trends that can be observed in Figure 8:

1. for a given number x of injected failures, all application instances whose number of potentially affected VMs are greater or equal to x presents an equivalent deployment duration (*e.g.*, the behavior of the 80% and of the 100% affected VMs farms are equivalent from 0 to 4 injected failures);
2. for a given application architecture, the average duration to deploy an applicative instance is inversely proportional to the number of potentially affected VMs part of the instance, *e.g.*, the deployment duration in presence of 3 failures in the 20% affected VMs case is equivalent to the value obtained for 9 failures with 60% affected VMs. In other words, the wider the occuring failures can be spread over a large set of VMs that can be affected –hence can be recovered by the protocol simultaneouly, in parallel, independently one from each other–, the less time it takes to deploy the application.

Finally it is noticeable that both functions $f(x)$ and $f'(x)$ obtained through these evaluations highly differ according to the values of their coefficients (*i.e.*, (a, b) and (a', b'), respectively). This is due to:

- the variation of the application architecture used. In the first experiment, the application includes one instance of each applicative tier (*i.e.*, 3 applicative VMs) whereas in the second it consists of one instance of the presentation tier, one instance of the database tier, and five instances of the business tier (*i.e.*, 7 applicative VMs).
- the underlying cloud infrastructure whose behavior can vary according to the application architecture to deploy. In our case, it depends among other things on the size of the cluster of physical machines used to instantiate simultaneously the applicative VMs (3 physical machines), on the computation capabilities of each of these physical machines, and on the policy to balance the VMs on these physical machines (round robin algorithm).

5. RELATED WORK

We first focus on existing approaches relying on ADL-based approaches for deploying software applications. [1, 2] propose languages and configuration protocols for distributed applications in the cloud. SmartFrog [1] is a framework for creating configuration-driven systems. It has been designed with the purpose of making the design, deployment and management of distributed component-based systems simpler and more robust. [2] adopts a model driven approach with extensions of the *Essential Meta-Object Facility* (*EMOF*) abstract syntax to describe a distributed application,

DOI: 10.1002/spe

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, [2] does not work in a decentralized fashion, and this harms the scalability of applications that can be deployed.

[17] 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 [18] 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.

[19] 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. This limit regarding deployment abilities is also present in most current IaaS frameworks, such as OpenStack, VMWare vCD, or Amazon WS. [20] 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).

[3, 21, 22] 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 related work [4] 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 [4] does not focus on composition consistency, architectural invariants preservation, or robustness of the reconfiguration protocol. This is one of the main forces of our approach, that is to work in a decentralized and automated fashion, and to support at the same time correctness features via invariants preservation and failure handling.

[23] presents a protocol for reconfiguring applications running in the cloud. The protocol supports the addition of new components and VMs as well as the removal of components and VMs. All these reconfiguration operations are posted through a cloud manager, which is in charge of guiding the reconfiguration of the whole application. This protocol also detects the occurrence of failures and in those situations makes the application restore a global consistent state. The main difference is that in this article, we focus on the deployment of applications whereas, in [23], the protocol applies on running applications and proposes operations to modify them. Our protocol does not only detect failures but also correct them by re-instantiating failed VMs. Last but not least, [23] presents the formal design of a reconfiguration protocol whereas the deployment protocol was also implemented and applied on real-world applications for evaluation purposes.

In [24], the authors present a technique for writing distributed applications which manage themselves over one or more utility computing infrastructures. This technique particularly allows to dynamically acquire new computational resources, deploy themselves on these resources, and release others when no longer required. However, this approach does not propose any failure

DOI: 10.1002/spe

recovery mechanism. [25] 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 of the closest solution to ours. Yet the centralized architecture and the absence of fault-tolerance mechanisms are two important differences. On a wider scale, and as a summary, these two characteristics are also the two main differences between all approaches presented before in this section and our deployment solution.

Aeolus [26] is a rich component model for designing distributed, scalable cloud applications. This model particularly consider component dependencies, non-functional requirements, and stateful applications. Each component is equipped with a state machine describing how required and provided functionalities are enacted. Another contribution of this work is to check the existence of a deployment plan. In comparison, in our approach, the plan is given by the user, but we provide automated mechanisms for executing it. [27] presents an approach to enable the seamless integration of script-centric and service-centric configuration technologies in order to customize and automate provisioning of composite cloud applications. In [28], the authors present an extension of the TOSCA standard in order to specify the behavior of a cloud application's management operations. They also propose several kinds of analysis for checking, *e.g.*, the validity of a management plan or the possibility to reach certain configuration given a plan. Our approach is different because there is no explicit plan, and the protocol itself executes implicitly the plan to deploy the application at hand. In that sense, our approach is more generic, but we only tackle the deployment phase so far.

In the cloud computing area, there are several configuration management tools, such as Puppet or Chef, which allow one to automatically configure new machines as described in dedicated files called manifests or recipes. Industrial technologies, such as Bosh, Cloudify, and Heat, help to create, deploy, and orchestrate applications in multiple cloud infrastructures. As far as the configuration steps are concerned, these technologies rely on the aforementioned configuration management tools (*e.g.*, Puppet or Chef). These industrial tools are very convenient for configuring many kinds of applications and systems. Our focus is slightly different here because we propose an approach working in a decentralized way and supporting failure detection and repair for complex application architectures.

Fault tolerance and system reliability have been the subject of many works and studies in the last decades [29]. A system may fail first because of incorrect specification, incorrect design, design flaws, poor testing, or undetected software faults. In such cases, a possible approach aims at improving system design and development using static analysis or specific programming models [30]. Other reasons of failures are external events [31], caused either by human errors or computing failures (both network failures and hardware failures). In all cases, failures are said to be fail-stop and their occurrence leads the system to abruptly stop itself. To manage fail-stop failures, a classic approach consists in using redundancy techniques, either active or passive ones.

Active redundancy techniques [32, 33, 34, 35, 36] intend to replicate the execution units in a system, typically the processes or the threads that are involved. This means that any request sent to a process or to a thread is in fact sent to the complete group of process replicas. Such redundancy techniques rely on dedicated communication protocols providing atomic and ordered group communication [37], for ensuring that all the members of a group receive a request or none at all receive it. With this approach, the system can tolerate $n - 1$ failures when the replication group is composed of n replicated processes.

Using passive redundancy techniques is an alternative approach [38, 39], which consists in replicating the state and/or the data of a system. The objective is to ensure that such data or state will stay available in case of failures, which can be achieved by saving it on a persistent storage such as a disk. The state or data are saved at particular execution points (checkpoints) such that the system can restart from the saved information (rollback) [40]. This approach is complex in a distributed setting, because in the general case, only local checkpoints can be performed for efficiency reasons and these checkpoints should allow for a global restart of the system in case of failure [41].

DOI: 10.1002/spe

In our approach, we adopt a recovery technique that relies on an asynchronous distributed message oriented middleware (MOM). The MOM provides a certain level of reliability, tolerating, among other things, transitory byzantine network failures using checkpoints and rollback techniques. We extended the set of tolerated faults to consider crash-stop failures of physical and virtual machines, by introducing a distributed replay mechanism (based on event logging) for state recovery. Overall, we leveraged a lightweight passive approach for failure detection and recovery, which was simple to develop and sufficient for ensuring fault-tolerance of our deployment solution.

A preliminary version of this work was published in [42] and was extended here as follows:

- We illustrate our contributions on a more complex case study.
- We describe more precisely the part of the protocol dedicated to the failure detection and management.
- We present the formal specification and verification of the deployment protocol with more details. In addition, we show a larger variety of experimental results and comment on issues detected using these analysis techniques.
- We present with more details the implementation of the protocol and show evaluation results in terms of performance and scalability on several real-world applications.
- We present an extended discussion comparing our approach with related work, particularly those handling failures.

6. 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 does not only detect 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 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 to 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.

REFERENCES

1. Goldsack P, Guijarro J, Loughran S, Coles A, Farrell A, Lain A, Murray P, Toft P. The SmartFrog Configuration Management Framework. *SIGOPS Oper. Syst. Rev.* 2009; **43**(1):16–25.
2. Chapman C, Emmerich W, Márquez FG, Clayman S, Galis A. Software Architecture Definition for On-demand Cloud Provisioning. *Proc. of HPDC'10*, ACM Press, 2010; 61–72.
3. Etchevers X, Coupaye T, Boyer F, de Palma N. Self-Configuration of Distributed Applications in the Cloud. *Proc. of CLOUD'11*, IEEE Computer Society, 2011; 668–675.
4. Fischer J, Majumdar R, Esmaeilsabzali S. Engage: A Deployment Management System. *Proc. of PLDI'12*, ACM, 2012; 263–274.

DOI: 10.1002/spe

5. Champelovier D, Clerc X, Garavel H, Guerte Y, Powazny V, Lang F, Serwe W, Smeding G. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4) 2011. INRIA/VASY.
6. Garavel H, Lang F, Mateescu R, Serwe W. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. *Proc. of TACAS'11*, *LNCS*, vol. 6605, Springer, 2011; 372–387.
7. Dillenseger B. CLIF, a Framework based on Fractal for Flexible, Distributed Load Testing. *Annales des Télécommunications* 2009; **64**(1-2):101–120.
8. Bruneton E, Coupaye T, Leclercq M, Quéma V, Stefani JB. The FRACTAL Component Model and its Support in Java. *Softw., Pract. Exper.* 2006; **36**(11-12):1257–1284.
9. Letondeur L, Etchevers X, Coupaye T, Boyer F, de Palma N. Architectural Model and Planification Algorithm for the Self-Management of Elastic Cloud Applications. *Proc. of ICCAC'14*, 2014; 172–179.
10. Hermenier F, Lawall JL, Muller G. BtrPlace: A Flexible Consolidation Manager for Highly Available Applications. *IEEE Trans. Dependable Sec. Comput.* 2013; **10**(5):273–286.
11. Mateescu R, Thivolle D. A Model Checking Language for Concurrent Value-Passing Systems. *Proc. of FM'08*, *LNCS*, vol. 5014, Springer, 2008; 148–164.
12. Dwyer MB, Avrunin GS, Corbett JC. Patterns in Property Specifications for Finite-State Verification. *Proc. of ICSE'99*, ACM, 1999; 411–420.
13. Milner R. *Communication and Concurrency*. Prentice-Hall, 1989.
14. Bellissard L, Palma ND, Freyssinet A, Herrmann M, Lacourte S. An Agent Platform for Reliable Asynchronous Distributed Programming. *Proc. of SRDS'99*, IEEE Computer Society, 1999; 294–295.
15. Etchevers X, Coupaye T, Boyer F, Palma ND, Salaün G. Automated Configuration of Legacy Applications in the Cloud. *Proc. of UCC'11*, IEEE Computer Society, 2011; 170–177.
16. Kwiatkowski N. Comparative Study of Deployment Architectures in the Cloud 2013. Master's Degree Report.
17. Broto L, Hagimont D, Stolf P, Palma ND, Temate S. Autonomic Management Policy Specification in Tune. *Proc. of SAC'08*, ACM, 2008; 1658–1663.
18. Baduel L, Baude F, Caromel D, Contes A, Huet F, Morel M, Quilici R. Programming, Composing, Deploying for the Grid. *Grid Computing: Software Environments and Tools*. Springer, 2006; 205–229.
19. Nurmi D, Wolski R, Grzegorczyk C, Obertelli G, Soman S, Youseff L, Zagorodnov D. The Eucalyptus Open-Source Cloud-Computing System. *Proc. of CCGRID'09*, IEEE Computer Society, 2009; 124–131.
20. Chohan N, Bunch C, Pang S, Krintz C, Mostafa N, Soman S, Wolski R. AppScale: Scalable and Open AppEngine Application Development and Deployment. *Proc. of CloudComp'09*, *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, vol. 34, Springer, 2010; 57–70.
21. Salaün G, Etchevers X, Palma ND, Boyer F, Coupaye T. Verification of a Self-configuration Protocol for Distributed Applications in the Cloud. *Proc. of SAC'12*, ACM Press, 2012; 1278–1283.
22. Abid R, Salaün G, Bongiovanni F, De Palma N. Verification of a Dynamic Management Protocol for Cloud Applications. *Proc. of ATVA'13*, *LNCS*, vol. 8172, Springer, 2013; 178–192.
23. Durán F, Salaün G. Robust and Reliable Reconfiguration of Cloud Applications. *Journal of Systems and Software* 2016; To appear.
24. Yin Q, Cappos J, Baumann A, Roscoe T. Dependable Self-Hosting Distributed Systems Using Constraints. *Proc. of HotDep'08*, USENIX Association, 2008.
25. Kirschnick J, Calero JMA, Goldsack P, Farrell A, Guijarro J, Loughran S, Edwards N, Wilcock L. Towards an Architecture for Deploying Elastic Services in the Cloud. *Softw., Pract. Exper.* 2012; **42**(4):395–408.
26. Cosmo RD, Mauro J, Zacchiroli S, Zavattaro G. Aeolus: A Component Model for the Cloud. *Inf. Comput.* 2014; **239**:100–121.
27. Breitenbücher U, Binz T, Kopp O, Leymann F, Wettinger J. Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies. *Proc. of OTM'13*, *LNCS*, vol. 8185, Springer, 2013; 130–148.
28. Brogi A, Canciani A, Soldani J. Modelling and Analysing Cloud Application Management. *Proc. of ESOCC'15*, *LNCS*, vol. 9306, Springer, 2015; 19–33.
29. Zamojski W, Caban D. Introduction to the Dependability Modeling of Computer Systems. *Proc. of DepCoS-RELCOMEX'06*, IEEE Computer Society, 2006; 100–109.
30. Saha GK. Software Based Fault Tolerance: A Survey. *Ubiquity* 2006; .
31. Avizienis A, Laprie J, Randell B, Landwehr CE. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Trans. Dependable Sec. Comput.* 2004; **1**(1):11–33.
32. Lamport L. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks* 1978; **2**:95–114.
33. Schneider FB. Implementing Fault-tolerant Services using the State Machine Approach: A Tutorial. *ACM Comput. Surv.* 1990; **22**(4):299–319.
34. Chérèque M, Powell D, Reynier P, Richier JL, Voiron J. Active Replication in Delta-4. *Proc. of FTCS'92*, IEEE Computer Society, 1992; 28–37.
35. Becker T. Application-transparent Fault Tolerance in Distributed Systems. *Proc. of CDS'94*, IEEE, 1994; 36–45.
36. van Renesse R, Guerraoui R. Replication Techniques for Availability. *Replication: Theory and Practice*, *LNCS*, vol. 5959, Springer, 2010; 19–40.
37. Renesse RV, Birman KP, Maffeis S. Horus: A Flexible Group Communication System. *Comm. of the ACM* 1996; **39**(4):76–83.
38. Budhiraja N, Marzullo K. Highly-Available Services Using the Primary-Backup Approach. *Workshop on the Management of Replicated Data*, 1992; 47–50.
39. Ren Y, Rubel P, Seri M, Cukier M, Sanders WH, Courtney T. Passive Replication Schemes in Aqua. *Proc. of PRDC'02*, IEEE Computer Society, 2002; 125–130.
40. Elnozahy EN, Alvisi L, Wang YM, Johnson DB. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.* 2002; **34**(3):375–408.

41. Chandy KM, Lamport L. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.* 1985; **3**(1):63–75.
42. Etchevers X, Salaün G, Boyer F, Coupaye T, Palma ND. Reliable Self-Deployment of Cloud Applications. *Proc. of SAC'14*, ACM Press, 2014; 1331–1338.

DOI: 10.1002/spe