HAL
open science

Robust Reconfiguration of Cloud Applications

Francisco Durán, Gwen Salaün

To cite this version:

Francisco Durán, Gwen Salaün. Robust Reconfiguration of Cloud Applications. The 17th International ACM Sigsoft Symposium on Component-Based Software Engineering (CBSE 2014), Jun 2014, Lille, France. hal-01016401

HAL Id: hal-01016401

https://inria.hal.science/hal-01016401v1

Submitted on 30 Jun 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.

Robust Reconfiguration of Cloud Applications

Francisco Durán
University of Málaga, Spain
duran@uma.es

Gwen Salaün
University of Grenoble Alpes,
Inria, LIG, CNRS, France
gwen.salaun@imag.fr

ABSTRACT

Cloud applications involve a set of interconnected software components running on remote virtual machines. Once cloud applications are deployed, one may need to reconfigure them by adding/removing virtual machines or components hosted on these machines. These tasks are error-prone since they must preserve the application consistency and respect important architectural invariants related to software dependencies. We present in this paper a protocol for automating these reconfiguration tasks.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*Languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and Interfaces*

General Terms

Algorithms, Reliability

Keywords

Cloud Computing, Component-based Systems, Dynamic Reconfiguration

1. INTRODUCTION

Cloud computing aims at delivering resources and software applications on demand over a network, leveraging hosting platforms based on virtualization, and promoting a new software licensing and billing model based on the *pay-per-use* concept. For service providers, this means the opportunity to develop, deploy, and possibly sell cloud applications everywhere on earth without investing in expensive IT infrastructure. Cloud computing is at the crossroads of several recent computing paradigms such as grid computing, peer-to-peer architectures, autonomic computing, or utility computing. It allows users to benefit from all these technologies without requiring a deep expertise in each of them. In particular, autonomic computing is convenient for automating specific tasks such as the provisioning of resources on-demand or facing peak-load capacity surge. Automation reduces user involvement, which speeds up the process and minimizes the possibility of human errors.

Cloud applications are distributed applications composed of a set of virtual machines running a set of interconnected software components. Such applications 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 provision and instantiate some virtual machines (VMs) and indicate the software components to be run on them. Once these applications are deployed, some reconfiguration operations may be required, such as instantiating new VMs, dynamically replicating some of them for load balancing purposes (elasticity), destroying or replacing VMs, etc. However, setting up, monitoring, and reconfiguring distributed applications in the cloud are complicated tasks because software involves many dependencies that oblige any change to be made in a certain order for preserving application consistency. Moreover, some of these tasks can be executed in parallel for execution time and performance optimization, but this cannot easily be achieved manually. Thus, there is a need for robust protocols that fully automate reconfiguration tasks on running applications distributed across several VMs. The design of these reconfiguration mechanisms is complicated not only due to the high level of parallelism inherent to such applications, but also because they must preserve important architectural invariants at each step of the protocol application, *e.g.*, a started component cannot be connected to (and then possibly using) a stopped component.

In this paper, we present a new protocol, which aims at reconfiguring at runtime cloud applications consisting of a set of interconnected components hosted on remote VMs. We consider several kinds of reconfiguration operations, namely addition and suppression of bindings, components, and VMs. When configuring an application (up phase), the protocol is able to instantiate VMs, effectively connect the components as required, and start these components respecting their functional dependencies. When removing parts of an application (down phase), the protocol needs to stop and disconnect components in a certain order for preserving the architecture consistency. For instance, since we never want a started component to be connected to a stopped component, in order to stop a component, we must previously ask their client components (components bound to that component) to unbind, and we can stop the

component only when they have all done so. This supposes a backward-then-forward propagation of messages across VMs composing the application, along bindings connecting components on mandatory required services.

The paper is structured as follows. We present the reconfiguration mechanisms in Section 2. We review related work in Section 3 and we conclude in Section 4.

2. RECONFIGURATION PROTOCOL

2.1 Application Model

For the sake of comprehension, we abstract away from several implementation details such as IP addresses or configuration parameters. Thus, an application model consists of a set of VMs. These VMs do not play any role *per se*, from a functional point of view, but each of them hosts a set of components, where resides the functional part of the application. A component can be in one of these two states: *started* and *stopped*. A component can either provide or require services. This is symbolized using ports: an *import* represents a service required by a component and an *export* represents a service provided by a component. An import can be *optional* or *mandatory*. An import is *satisfied* when it is connected to a matching export and the component offering that export is started. Such a connection is called a *binding*. A component can import a service from a component hosted on the same VM (local binding) or hosted on another VM (remote binding). A component can be started, and then be fully operational, when all its mandatory imports are satisfied. A component can be fully operational even if its optional imports are not satisfied.

We will use as running example a three-tier Web application (Fig. 1). Although this is a simple example, it shows several kinds of dependencies and allows us to illustrate our algorithm on interesting cases. VM1, hosts two components: a front-end Web server (Apache) and a profiling component. VM2 hosts an application server (Tomcat) and an object cache component. VM3 corresponds to the database management system (MySQL). These components are connected using local or remote bindings. These bindings can involve optional imports (o in the figure) or mandatory imports (m).

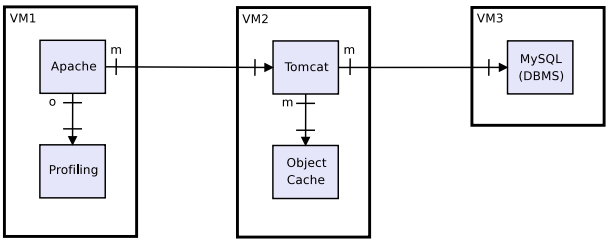

Figure 1: Example: A Web Application Model

2.2 Protocol Features

Our reconfiguration protocol exhibits three main important design features, namely, it is fully automated, decentralized, and robust.

Each VM is equipped with a VM manager in charge of automating the reconfiguration tasks at the VM level.[1] All VM managers work without any human intervention. The cloud manager posts reconfiguration operations that can be given by a cloud user or encoded into a scripting language. Thus, the cloud manager does not necessarily require the presence of a human being for interacting with the running system and application.

VM managers are in charge of starting/stopping their own components and no centralized manager is used for that purpose. The protocol is also loosely-coupled because each VM manager does not have a global view of the current state of the application and particularly of the other VMs. Yet the VM managers need to exchange information in order to connect bindings on remote components or to let certain components know that other (partner) components have started or stopped. The only way to exchange necessary information for the component start-up/shutdown is to interact via asynchronous message passing. Each VM is equipped with two FIFO buffers, one for incoming messages and one for outgoing messages. VMs interact together in a point-to-point fashion (no broadcast or multi-way communication). This solution is standard in distributed systems, and avoids the use of bottleneck centralized servers or communication media (*e.g.*, a publish-subscribe messaging system [2]), which limit the parallelism induced in the distributed system by transforming it somehow into a centralized one.

The protocol is robust in the sense that, during its application, some important architectural invariants are preserved, *e.g.*, all mandatory imports of a started component are satisfied (*i.e.*, bound to started components). These invariants are crucial because they ensure that component assemblies are well-formed. Therefore, they must be preserved during the whole lifetime of the application and at any step of the reconfiguration protocol execution.

2.3 Participants

The reconfiguration protocol involves a cloud manager and a set of VM managers. The cloud manager (CM) guides the application reconfiguration by instantiating/destroying VMs and adding/removing components/bindings. Each VM in the distributed application is equipped with a VM manager that is in charge of (dis)connecting bindings and starting/stopping components upon VM instantiation/destruction operations posted by the CM. Communications between participants (CMs and VM managers) are achieved asynchronously via FIFO buffers. When a participant needs to post a message, it puts that message in its output buffer. When it wants to read a message, it takes the oldest one in its input buffer. Messages are transferred at any time from an output buffer to its addressee's input buffer. Buffers are unbounded, but the protocol does not involve looping tasks that would make the system infinitely send messages to buffers.

Fig. 2 depicts a sample system with a CM and two VMs, and shows how they exchange messages through their buffers (dashed lines). More precisely, when the CM, for instance, needs to send an output message to VM1, it first adds it to its output buffer. The message is then transferred from CM's output buffer to VM1's input buffer. The VM1 manager can finally consume this message from its input buffer.

[1]We distinguish in the rest of this paper a VM, which is a software implementation of a physical machine, and a VM manager, which is the piece of software embedded on a VM in charge of applying the reconfiguration tasks on that VM.

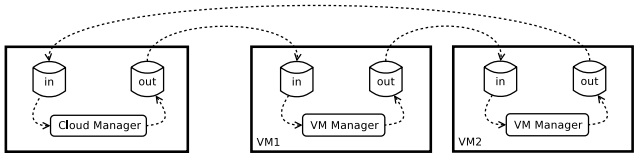

Figure 2: Participants and Communication Model

2.4 Cloud Manager

The CM submits reconfiguration operations to the running application and keeps track of the state of the deployed VMs. We consider the following reconfiguration operations: instantiation/destruction of a VM, addition/removal of a component on/from an existing VM, and addition/suppression of bindings.

In order to ensure a correct execution of the protocol, the CM validates the operations before applying them, *e.g.*, a VM is destroyed only if instantiated before, a new binding is added only if both ports exist in the application, or a binding is added only if it does not form a cycle along mandatory imports. Our reconfiguration mechanisms are triggered by the execution of a sequence of such operations posted by the CM for, *e.g.*, maintenance or elasticity purposes [20].

The protocol works applying *up* and *down* phases. A phase has a coarse-grained granularity compared to atomic reconfiguration operations introduced above. An *up* phase corresponds to a set of reconfiguration operations dedicated to start-up operations (*e.g.*, VM instantiation or binding addition). When the CM instantiates a VM, it creates an image of this VM and the VM starts executing itself. When a CM adds a set of required bindings to the running application, it submits messages to all VMs impacted by these changes, that is, all VMs hosting components involved in those bindings. These messages come with some configuration information necessary to the VM manager for binding purposes. In contrast, a *down* phase involves shutdown operations only (*e.g.*, VM destruction or binding removal). When the CM decides to destroy a VM, it sends a message to that VM. A VM destruction message implies the destruction of all bindings on components hosted on that VM. These two kinds of phases are applied alternatively in sequence and each phase is initiated by the CM. The CM also keeps track of the current state of all VMs running in the system (instantiated VMs and whether they are started or not). A VM is declared *started* when all components on that VM are started. A VM is declared *stopped* otherwise. Fig. 3 summarizes the CM lifecycle where we distinguish reconfiguration operations posted by the CM (solid lines) and messages received from the VM managers (dashed lines).

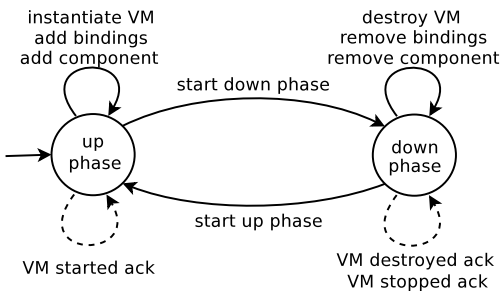

Figure 3: Cloud Manager Lifecyle

We give in Fig. 4 a scenario with successive up/down phases for our running example. In a first up phase, we instantiate all VMs and add a set of required bindings (Bds corresponds to the set of bindings in Fig. 1). Then, we decide to remove the MySQL component for replacing it by a new version (down phase). Finally, we add this new component (MySQL') on VM3 and add a binding (Bd') connecting the Tomcat component to the new MySQL component.

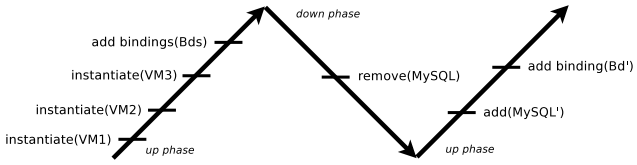

Figure 4: Running Example: Up/Down Scenario

2.5 VM Manager

Each VM is equipped with a VM manager, which starts its activity when the CM instantiates its VM. A VM manager is in charge of binding, unbinding, starting, and stopping components. In the rest of this section, we present the two most general reconfiguration operations, which are the instantiation and the destruction of a VM, resp.

Binding and start-up. Fig. 5 shows how a newly instantiated VM proceeds in order to bind its ports and start its components. After instantiation (❶), the VM manager can immediately start a component without imports or with optional imports only (❷). If a component involves mandatory imports, that component can only be started when all its mandatory imports are satisfied, *i.e.*, when all these imports are bound to started components. When a component is started, its VM manager informs the VM managers of all remote components using it by sending *component started* messages (❸). If all components of a VM are started, its VM manager sends a message to inform the CM (❹), otherwise it starts reading messages from its input buffer (❺):

- If a VM receives from the CM some binding information (for both local and remote bindings), the manager first connects local bindings (❼). As for remote bindings, when an export of one of its components is involved in a binding, the VM manager sends a message (❽) with its export connection information (*e.g.*, IP address) to the VM hosting the other component (import side).
- If the VM receives a remote binding message, this means that an import of one of its components is involved in a binding. Upon reception of that message, the VM manager makes the binding effective (❻).
- Every time a *component started* message is received, the VM manager checks if the corresponding components can be started (❷). Each VM manager keeps the states of its partner components.

Note that the start-up process implies a propagation of *started* messages along bindings across several VMs. Local bindings are handled directly by VM managers and there is no need of exchanging messages with other VMs. The algorithm checks for cycles of bindings over mandatory ports, thus ensuring the termination of the start-up process.

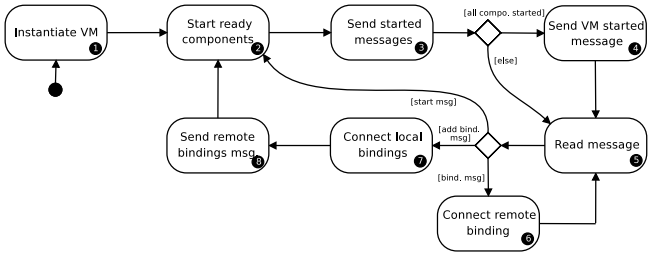

Figure 5: VM Manager Activity Diagram: Up Phase

Unbinding and shutdown. A VM manager is in charge of stopping some local components, or all its components when the VM is to be destroyed (Fig. 6, ❶), *i.e.*, removed from the running application. In this case, all the components hosted on that VM need to be stopped and all bindings on these components (connected to imports or exports) need to be removed. If a component involved in the shutdown process does not provide any service (there is no component connected to it), it can immediately stop, and all outgoing bindings can be removed for these components (❷). Otherwise, it cannot stop before all partner components connected to it on mandatory imports have unbound themselves. To do so, the VM manager of the VM under destruction first sends *unbind required* messages to all VMs hosting components connected to those VM's components (❸). The VM manager of the VM to be destroyed then collects *unbind confirmed* messages (❹) and stops the corresponding components when all components using that component on mandatory imports have stopped and unbound (❺). Whenever a component stops, an *unbind confirmed* message is sent (❻). The VM is destroyed and the CM informed when all components are stopped (❼).

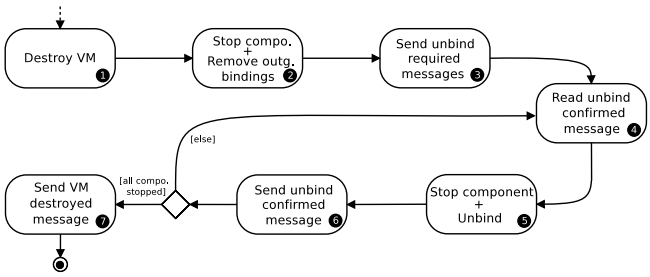

Figure 6: VM Manager Activity Diagram: Down Phase (Destruction)

As a side effect to a VM destruction, the other VM managers can receive messages (Fig. 7, ❶) from their partner VMs. Upon reception of an *unbind required* message, the VM manager either stops and unbinds some components (❷) if possible (no bindings on them or bindings on remote optional imports only), or sends similar messages for all remote components bound on mandatory imports to its components (❺). When a VM manager stops (and unbinds) a component (❷), it may send a message to the CM indicating that the VM is not fully operational (❸). It also sends messages to all remote partner components formerly providing a service to that component, to let them know that this component has been stopped/unbound (❹). Upon reception of an *unbind confirmed* message, the VM manager goes to step ❷.

Components bound on optional imports just need to unbind themselves, but do not need to stop. Local bindings are handled locally by the VM manager, but these changes can impact other remote components, and in that case additional *unbind required* messages may be emitted. The component shutdown implies a backward propagation of *unbind required* messages and, when this first propagation ends (on components without exports or with optional imports only), a second forward propagation of *unbind confirmed* messages starts to let the components know that the disconnection has been actually achieved. These propagations terminate because there is no cycle of bindings over mandatory imports.

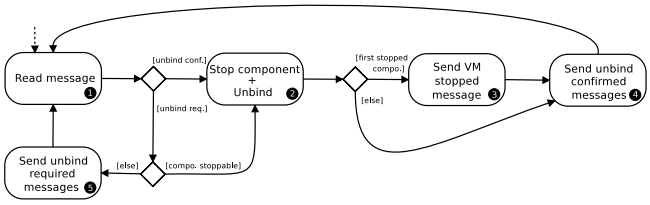

Figure 7: VM Manager Activity Diagram: Down Phase (Side Effect)

2.6 Examples of Reconfiguration Scenarios

We show in this section how the protocol works on simple reconfiguration scenarios for the Web application presented in Fig. 1. Let us assume that the application is fully operational and all components on all VMs are started (end of the first *up* phase in Fig. 4). A new version of the MySQL database management system is available and we decide to upgrade that component to this new version. Accordingly, the whole system initiates a *down* phase (middle part of Fig. 4) characterized by an emission of a *remove* message to VM3. We show in Fig. 8 a Message Sequence Chart (MSC) overviewing the interactions and behaviors of all participants (CM and VM managers) for this specific scenario.

Upon reception of the *remove* message, VM3 sends an *unbind required* message to VM2 requesting to unbind the Tomcat component from the MySQL component. When VM2 receives this message, it cannot unbind immediately because Tomcat is used by a remote component (Apache), therefore it sends too an *unbind required* message to VM1. Upon reception of that message, the VM1 manager stops the Apache component, because no other component is connected to it, and then unbinds the Apache component from the Tomcat component. VM1 sends a confirmation message to VM2 indicating that the disconnection has been achieved. VM1 also sends a *VM stopped* message to the CM indicating that all its components are not started anymore. When VM2 receives the *unbind confirmed* message, its manager stops Tomcat and unbinds it from MySQL. A confirmation is sent from VM2 to VM3 and a *VM stopped* message is sent to the CM. Once VM3 receives the confirmation message, its manager stops the MySQL component, and sends an acknowledgement message to the CM indicating that the VM is stopped too. Stopping Tomcat and Apache is required to preserve architectural invariants, here a started component cannot be connected to a stop component.

After the removal of MySQL, the application is in a situation where components Apache and Tomcat are off and components Profiling and Object Cache are on.

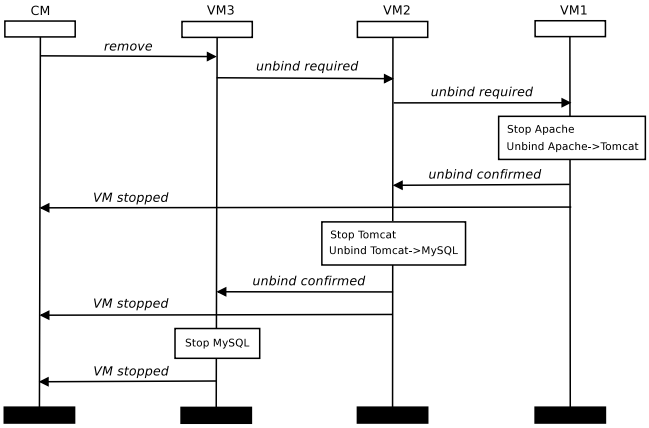

Figure 8: MySQL Removal Scenario

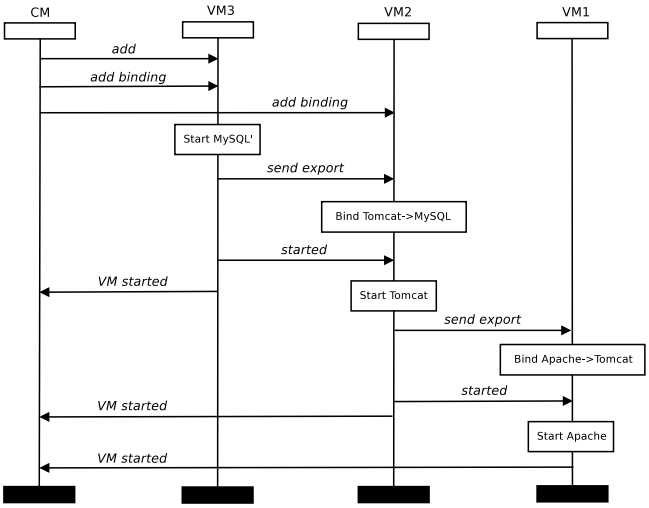

Figure 9: MySQL' Addition Scenario

In order to restore a fully operational application, let us now consider an *up* scenario (right-hand side of Fig. 4) where the CM manager adds a new version of the MySQL component on VM3 (*add* message) and a new binding between the Tomcat component and the new MySQL component. We show in Fig. 9 the interactions and actions involved in this scenario. VM3 can start the MySQL' component immediately because this component does not require any service from other components (no imports). VM3 knows that VM2 needs to connect its component to the MySQL' component, therefore the VM3 manager posts a *send export* message with the connection information to VM2. Upon reception, the VM2 manager can connect both components. The VM3 manager also indicates to VM2 that its MySQL' component has started and to the CM that VM3 is started. Upon reception of the *send export* message, the VM2 manager starts the Tomcat component. VM2 sends a *send export* message and a *started* message to VM1, because the VM2 manager knows the dependency between the Apache component and the Tomcat component. VM2 also informs the CM that VM2 is started. The VM1 manager finally binds Apache to Tomcat, starts the Apache component, and informs the CM that VM1 is started too. Note that acknowledgement messages are not systematically required. They are useful in some specific cases, *e.g.*, when a component (import side) expects its partner (export side) to start.

3. RELATED WORK

Dynamic reconfiguration has been extensively studied in the last 20 years in the context of, *e.g.*, software architectures [16, 19, 15, 4, 17], graph transformation [3, 24], software adaptation [22, 21, 8], metamodelling [14, 18], or reconfiguration patterns [7]. In software architectures, for example, the authors proposed various formal models, such as Darwin [16] or Wright [4], in order to specify dynamic reconfiguration of component-based systems whose architectures can evolve (adding or removing components and connections) at run-time. In the cloud computing area, some existing environments already provide some mechanisms to automatically scale deployed applications based on monitoring data (see, *e.g.*, the Elastic Beanstalk from Amazon Web Services). However, these approaches typically work at the application level (Platform-as-a-Service, PaaS). Moreover, changes are triggered with respect to the individual performance of each tier, although there are attempts to decide elasticity actions from entire application performance models, see, *e.g.*, the Reservoir [9] or ConPaaS projects [20]. In [11, 23, 12], the authors present a protocol that automates the deployment of distributed applications in cloud environments in a decentralized way. Each VM is in charge of starting its own components and to do so needs to interact with the other VMs in order to exchange binding information. Another related work [13] 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 [13] does not focus on composition consistency, architectural invariants preservation, or robustness of the reconfiguration protocol. [6, 5] present a reconfiguration protocol applying changes to a set of connected components for transforming a current assembly to a target one given as input. Reconfigurations steps aim at (dis)connecting ports and changing component states. The protocol is robust in the sense that all the steps of this protocol preserve a number of architectural invariants. This protocol does not easily scale to cloud applications because the authors assume that all components are hosted on a same VM and a unique centralized manager is in charge of the reconfiguration steps. In contrast, our protocol is fully parallel (all VMs evolve independently one from another, at different speeds). In [2], the authors present a management protocol for instantiating and removing VMs from a running cloud application, but the protocol is quite different because it relies on another communication model, namely a publish-subscribe messaging system.

4. CONCLUDING REMARKS

In this paper, we have presented a new protocol for automatically reconfiguring cloud applications consisting of interconnected components distributed over several VMs. The protocol does not only support VM instantiation and component start-up, but also VM destruction and component shutdown. These management tasks are guided by reconfiguration operations posted through a cloud manager. All

VMs work in a fully decentralized and loosely-coupled way in order to apply these reconfiguration tasks, exchanging messages when necessary via FIFO buffers. The protocol is robust in the sense that it preserves composition consistency and well-formedness architectural invariants at any step of its application.

Due to the high degree of parallelism inherent to the applications to be reconfigured, the design of these reconfiguration mechanisms was very complicated and would have been impossible without the support of formal techniques and tools. Therefore, we specified the reconfiguration protocol using the rewriting-logic-based Maude language [10]. This results in a formal model of the protocol that we analyzed using Maude verification tools for chasing subtle bugs in boundary cases and therefore ensuring that our implementation satisfies some key properties and invariants. All Maude sources for our specification and its verification are available online [1]. It is worth noting that a Java implementation of the protocol is under development at Orange Labs in the context of the OpenCloudware funded project.[2]

A first perspective aims at improving the protocol to avoid using up/down phases. This is a non-trivial change since start and stop messages may be unwillingly mixed up. We also plan to extend the protocol to support VM failures.

Acknowledgements. This work has been partially supported by the Spanish project TIN2011-23795, the SeaClouds EU project, Universidad de Málaga (Campus de Excelencia Internacional Andalucía Tech), and the OpenCloudware project, which is funded by the French *Fonds national pour la Société Numérique* (FSN), by *Pôles* Minalogic, Systematic, and SCS.

5. REFERENCES

[1] `http://maude.lcc.uma.es/HRfCA`.

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

[3] N. Aguirre and T. Maibaum. A Logical Basis for the Specification of Reconfigurable Component-Based Systems. In *Proc. of FASE'03*, vol. 2621 of *LNCS*, pages 37–51. Springer, 2003.

[4] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of FASE'98*, vol. 1382 of *LNCS*, pages 21–37. Springer, 1998.

[5] F. Boyer, O. Gruber, and D. Pous. Robust Reconfigurations of Component Assemblies. In *Proc. of ICSE'13*, pages 13–22. IEEE/ACM, 2013.

[6] F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11*, vol. 6664 of *LNCS*, pages 103–117. Springer, 2011.

[7] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proc. of SERA'06*, pages 40–48. IEEE Computer Society, 2006.

[8] C. Canal, J. Cámara, and G. Salaün. Structural Reconfiguration of Systems under Behavioral Adaptation. *Science of Computer Programming*, 78(1):46–64, 2012.

[9] C. Chapman, W. Emmerich, F. G. Márquez, S. Clayman, and A. Galis. Software Architecture Definition for On-demand Cloud Provisioning. *Cluster Computing*, 15(2):79–100, 2012.

[10] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude - A High-Performance Logical Framework* , vol. 4350 of *LNCS*. Springer, 2007.

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

[12] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. D. Palma. Reliable Self-Deployment of Cloud Applications. In *Proc. of SAC'14*, pages 1331–1338. ACM Press, 2014.

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

[14] A. Ketfi and N. Belkhatir. A Metamodel-Based Approach for the Dynamic Reconfiguration of Component-Based Software. In *Proc. of ICSR'04*, vol. 3107 of *LNCS*, pages 264–273. Springer, 2004.

[15] J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.

[16] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proc. of SIGSOFT FSE'96*, pages 3–14, 1996.

[17] J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *Proc. of WICSA'99*, vol. 140 of *IFIP Conference Proceedings*, pages 35–50. Kluwer, 1999.

[18] J. Matevska-Meyer, W. Hasselbring, and R. Reussner. Software Architecture Description Supporting Component Deployment and System Runtime Reconfiguration. In *Proc. of WCOP'04*, 2004.

[19] N. Medvidovic. ADLs and Dynamic Architecture Changes. In *SIGSOFT 96 Workshop*, pages 24–27. ACM, 1996.

[20] G. Pierre and C. Stratan. ConPaaS: A Platform for Hosting Elastic Cloud Applications. *IEEE Internet Computing*, 16(5):88–92, 2012.

[21] P. Poizat and G. Salaün. Adaptation of Open Component-Based Systems. In *Proc. of FMOODS'07*, vol. 4468, pages 141–156. Springer, 2007.

[22] P. Poizat, G. Salaün, and M. Tivoli. On Dynamic Reconfiguration of Behavioural Adaptation. In *Proc. of WCAT'06*, pages 61–69, 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] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A Graph Based Architectural (Re)configuration Language. In *Proc. of ESEC / SIGSOFT FSE'01*, pages 21–32. ACM Press, 2001.

[2] `http://opencloudware.org`