HAL
open science

Verification of a Dynamic Management Protocol for Cloud Applications

Rim Abid, Gwen Salaün, Francesco Bongiovanni, Noël de Palma

To cite this version:

Rim Abid, Gwen Salaün, Francesco Bongiovanni, Noël de Palma. Verification of a Dynamic Management Protocol for Cloud Applications. 11th International Symposium, ATVA 2013, Dang Van Hung and Mizuhito Ogawa, Oct 2013, Hanoi, Vietnam. pp.178-192, 10.1007/978-3-319-02444-8_14 . hal-00863262

HAL Id: hal-00863262

https://inria.hal.science/hal-00863262v1

Submitted on 18 Sep 2013

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.

Verification of a Dynamic Management Protocol for Cloud Applications

Rim Abid[1,3], Gwen Salaün[2,3], Francesco Bongiovanni[1], and Noel De Palma[1]

[1] UJF, Grenoble, France
{Francesco.Bongiovanni, Noel.Depalma}@imag.fr
[2] Grenoble INP, France
[3] Inria Rhône-Alpes, Grenoble, France
{Rim.Abid, Gwen.Salaun}@inria.fr

Abstract. Cloud applications are composed of a set of interconnected software components distributed over several virtual machines. There is a need for protocols that can dynamically reconfigure such distributed applications. In this paper, we present a novel protocol, which is able to resolve dependencies in these applications, by (dis)connecting and starting/stopping components in a specific order. These virtual machines interact through a publish-subscribe communication media and reconfigure themselves upon demand in a decentralised fashion. Designing such protocols is an error-prone task. Therefore, we decided to specify the protocol with the LNT value-passing process algebra and to verify it using the model checking tools available in the CADP toolbox. As a result, the introduction of formal techniques and tools help to deeply revise the protocol, and these improvements have been taken into account in the corresponding Java implementation.

1 Introduction

Cloud computing is a new programming paradigm that emerged a few years ago, which aims at delivering resources and software applications over a network (such as the Internet). Cloud computing leverages hosting platforms based on virtualization and promotes 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 sell cloud applications worldwide without having to invest upfront in expensive IT infrastructure. Cloud applications are distributed applications that run on different virtual machines (*a.k.a.*, Infrastructure as a Service, IaaS). Therefore, to deploy their applications, cloud users need first to instantiate several virtual machines. Moreover, during the application time life, some management operations may be required, such as instantiating new virtual machines, replicating some of them for load balancing purposes, destroying or replacing virtual machines, etc.

Existing protocols [6, 8, 19] mainly focus on self-deployment issues where a model of the application (virtual machines, components, ports, and bindings) to be deployed exists and guides the configuration process. This approach works fine

only with specific applications where the application does not need to be changed after deployment. Unfortunately, this is not the case in the cloud, where most applications need to be reconfigured for integrating new requirements, scaling on-demand, or performing failure recovery. Therefore, cloud users need protocols that are not limited to deploying applications but can also work, as automatically as possible, in all the situations where changes have to be applied on to a running application. Such reconfiguration tasks are far from trivial, particularly when some architectural invariants (*e.g.*, a started component cannot be connected to a stopped component) must be preserved at each step of the protocol application.

In this paper, we first present a novel protocol which aims at (re)configuring distributed applications in cloud environments. These applications consist of interconnected software components hosted on several virtual machines (VMs). A deployment manager guides the reconfiguration tasks by instantiating new VMs or destroying existing ones. After instantiation, each VM tries to satisfy its required services (ports) by binding its components to other components providing these services. When a VM receives a destruction request from the deployment manager, that VM unbinds and stops its components. In order to (un)bind/start/stop components, VMs communicate together through a publish-subscribe communication media. As an example, for connecting one component hosted on a VM to another component hosted on another VM, the second VM must send its IP address to the first one for binding purposes.

Designing such protocols is a complicated task because they involve a high degree of parallelism and it is very difficult to anticipate all execution scenarios, which is necessary to avoid unexpected erroneous behaviours in the protocol. Hence, we decided to use formal techniques and tools for ensuring that the protocol satisfies certain key properties. More precisely, we specified the protocol in LOTOS NT (LNT) [4], which is an improved version of LOTOS [11]. The main difference between LOTOS and LNT is that LNT relies on an imperative-like specification language that makes its writing and understanding much simpler. For verification purposes, we used more than 600 hand-crafted examples (application model and reconfiguration scenario) and checked on them 35 identified temporal properties that the protocol must respect during its application. For each example, we generated the Labelled Transition System (LTS) model from the LNT specification and verified all the properties on it using model checking tools available in the CADP toolbox [9].

These verification techniques helped us to improve the protocol. For instance, in an initial version of the protocol, the component start-up/shutdown was guided by a centralised deployment manager. We observed an explosion in terms of states/transitions in the corresponding LTSs, even for simple examples involving few VMs. This was due to an overhead of messages transmitted to and from the deployment manager, which was supposed to keep track of all modifications in each VM to possibly start/stop components. We proposed a decentralised version of the protocol for avoiding this problem, where each VM is in charge of starting and stopping its own components. We also detected a major bug in the VM destruction process. Originally, when it was required to stop a component,

it was stopped before the components bound to it. This typically violates some architectural invariants (*e.g.*, a started component cannot be connected to a stopped component) and impedes the robustness level expected from the protocol. We corrected this issue by stopping properly components, which required a deep revision of the protocol. Thus, in the current version of the protocol, when a component must be stopped, it requests to all the components connected to it to unbind and once it is done, it can finally stop.

The rest of this paper is structured as follows. In Section 2, we present the reconfiguration protocol and show how it works on some concrete applications. In Section 3, we present the LNT specification of the protocol and its verification using CADP. We also comment on some experimental results and problems found. We discuss related work in Section 4 and we conclude in Section 5.

2 Dynamic Management Protocol

2.1 Application Model

Distributed applications in the cloud are composed of interconnected software components hosted on virtual machines. A component exports services that it is willing to provide and imports required services. Ports are typed and match when they share the same type, *i.e.*, an import for being satisfied requires an export with the same type. For effectively using a service, a component has to bind its import to an export with the same type. A component can import a service from a component hosted on the same machine (local binding) or hosted on another machine (remote binding). An import can be either mandatory or optional. Unlike optional imports, mandatory imports represent the services required by the component to be functional. A component has two states, started and stopped. Initially a component is in a stopped state. A component can be started when all its mandatory imports are bound to started components. Reversely, a started component must stop when at least one partner component connected to a mandatory import is required to stop.

An example of application model is given in Figure 1. This application consists of two VMs, both hosting two components. We can also see on this figure how imports and exports as well as their optional/mandatory parameter are described, and how bindings can be achieved on ports with the same type.

2.2 Protocol Participants

The management protocol involves three kinds of participants as presented in Figure 2. The deployment manager (DM) guides the application reconfiguration by successively instantiating and destroying VMs. Each VM in the distributed application is equipped with a configuration agent (agent for short in the rest of this paper) that is in charge of (dis)connecting bindings and starting/stopping

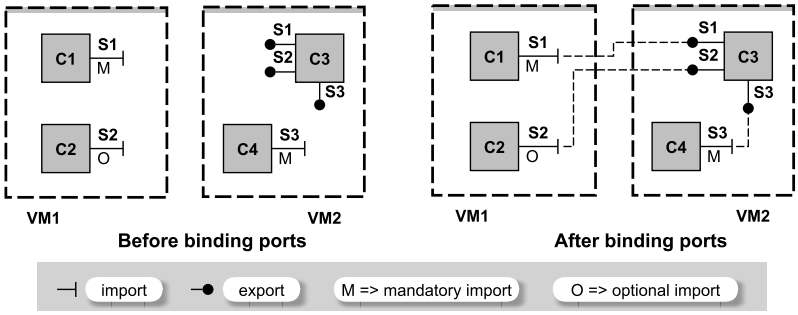

Fig. 1. Example of application model

components upon reception of VM instantiation/destruction reconfiguration operations from the DM. Communications between VMs are carried out thanks to a publish-subscribe communication media (PS). The PS is equipped with two topics[4]: (i) an export topic where a component subscribes its imports and publishes its exports, and (ii) an import topic where a component subscribes its exports and publishes its imports (we show in Section 2.3 why this double subscription/publication is required). The PS also contains a list of buffers used to store messages exchanged between agents. When a new VM is instantiated, a buffer for that VM is added to the PS. When an existing machine is destroyed, its buffer is removed from the PS.

2.3 Protocol Description

We now explain how the protocol works and we illustrate with several simple scenarios. Once a VM is instantiated, the agent is in charge of starting all the local components. When a component does not have any import or only optional ones, it can start immediately. Otherwise, each mandatory import requires an export (local or remote) with the same type. The PS is used to resolve compatible dependencies. When an import is bound to an available compatible export, it can be started only after the partner component has been started. The PS is also used to exchange this start-up information between the two VMs involved in a same binding.

Let us focus on two concrete scenarios (Fig. 3) for deploying an application composed of two VMs: in the first scenario we instantiate VM1 and then VM2, whereas they are instantiated in the other way round in the second scenario. These scenarios help to understand how the PS is used for resolving port dependencies and start/stop components.

[4] A topic is a logical channel where messages are published and subscribers to a topic receive messages published on that topic.

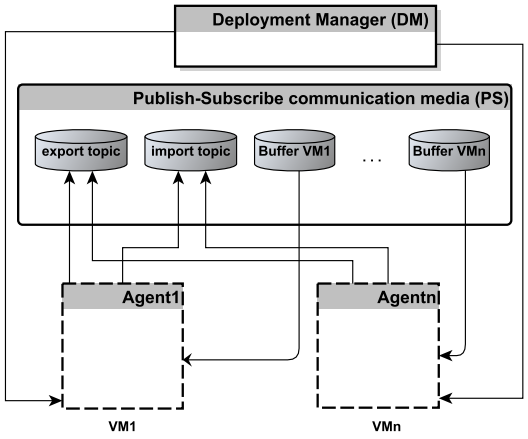

Fig. 2. Protocol participants

In the first scenario, when VM1 is instantiated, the Apache component requires a mandatory service whose type is Workers. Therefore, it subscribes to the export topic (1) and then publishes its import to the import topic (2). The PS receives that message from the VM1 agent, checks the import topic, and does not find a provider for the Workers service: the publication message is deleted. VM2 is then instantiated. The Tomcat component does not have any import and can therefore be started immediately (3). It provides an export with type Workers, so it subscribes this export to the import topic (4) and publishes it to the export topic (5). The start-up information is also sent to the PS. The PS receives that message from the VM2 agent, checks and finds that the Apache component hosted on VM1 has required this service (it has subscribed to the export topic). Hence, a message with binding details and Tomcat's state is added to VM1 buffer (6). Upon reception of this message, the Apache component is bound to the Tomcat component (7) and the VM1 agent starts the Apache component (8). The application is fully operational.

In the second scenario, when VM2 is instantiated, the Tomcat component does not have any import and is therefore started immediately (1). It provides an export with type Workers, so it subscribes this export to the import topic (2) and publishes it to the export topic (3). The PS receives that message from the VM2 agent, checks and does not find any component that requires Workers: the publication message is deleted. When VM1 is instantiated, the Apache component requires a mandatory service whose type is Workers. Therefore, it subscribes to the export topic (4) and publishes its import to the import topic (5). The PS receives that message from the VM1 agent, checks the import topic, and finds that Tomcat has provided the Workers service (it has subscribed to the import topic). The PS notifies VM2 that there is an Apache hosted on VM1 that needs Workers (6). VM2 receives the notification message, so it publishes

Tomcat's export and state, that is started (7). The PS forwards this information to the VM1 agent (8), and the Apache component can be bound to the Tomcat component (9) and started (10).

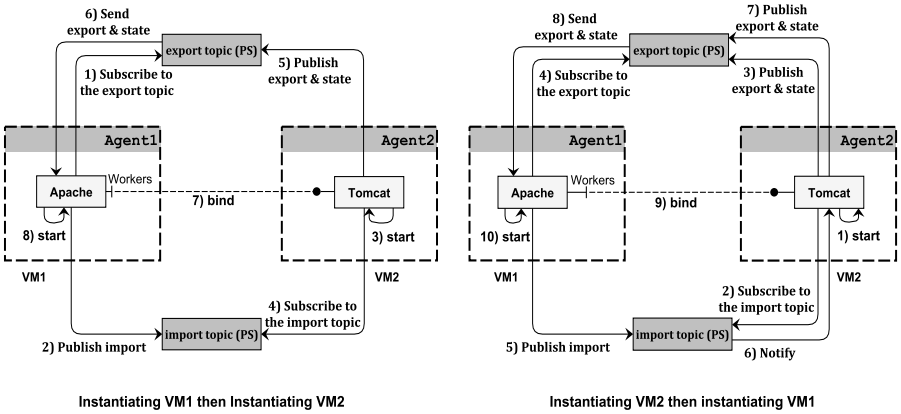

Fig. 3. Examples of VM instantiation scenario

Another goal of this protocol is to properly stop components when a VM is destroyed. In that case, all the components hosted on that VM need to be stopped as well as all components bound to them on mandatory imports (components bound on optional imports just need to unbind themselves). If a component does not provide any service (there is no component connected to it), it can immediately stop. Otherwise, it cannot stop before all partner components connected to it have unbound themselves. To do so, the component is unsubscribed from the import topic and then for each export, messages are sent to all components subscribed to that export requiring them to unbind (hence stop if they are bound on mandatory imports). Then the component waits until all components bound to it disconnect and inform the component through the PS. When the component is notified that all components connected to it have effectively unbound, it can stop itself. The component shutdown implies a backward propagation of "ask to unbind" messages and, when this first propagation ends (on components with no exports or only optional imports), a second forward propagation "unbind confirmed" starts to let the components know that the disconnection has been actually achieved.

We present in Figure 4 an example of application containing three VMs where VM3 receives a destruction request from the DM. VM3 hosts the MySQL

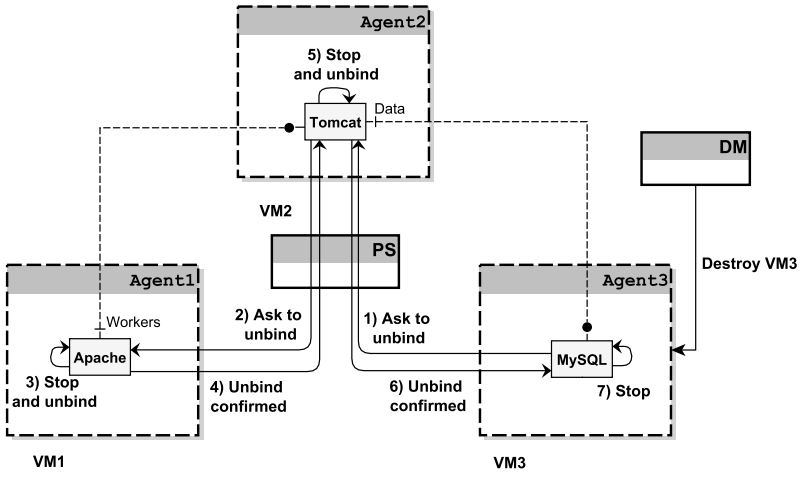

Fig. 4. Example of VM destruction scenario

component that provides a service imported by the Tomcat component, and thus cannot be stopped before Tomcat. Therefore, it unsubscribes from the import topic and sends a message to the PS asking to unbind the Tomcat component. The PS receives this message, transmits it to VM2 hosting Tomcat, that is subscribed to the import topic (1). Once VM2 receives this message, it cannot stop Tomcat because Apache is bound to it. VM2 sends a message to the PS asking to unbind Apache (2). Once VM1 receives the message, Apache does not provide any service so it is immediately stopped and unbound (3). VM2 then receives a message from the PS informing it that Apache has been unbound (4). Tomcat has no component bound to it now, so it is stopped and unbound from MySQL (5). VM3 receives a message from the PS informing it that Tomcat is no longer bound to it (6) and MySQL is finally stopped (7).

3 Specification and Verification

We specified the protocol in LNT [4], one of the input languages of CADP [9]. LNT is an improved version of LOTOS. We chose this language because it has the adequate expressiveness for the problem at hand and its user-friendly notation simplifies the specification writing. Moreover, we rely on the state-of-the-art verification tools provided by CADP to check that the protocol works correctly and as expected. CADP is a verification toolbox dedicated to the design, analysis, and verification of asynchronous systems consisting of concurrent processes interacting via message passing. The toolbox contains many tools that can be used to make different analysis such as simulation, model checking, equivalence checking, compositional verification, test case generation, or performance evaluation.

In the rest of this section, we present the specification of the protocol in LNT, its verification using the CADP model checker (Evaluator), some experimental results, and problems detected and corrected during the verification process. It is worth noting that since these techniques and tools work on finite state spaces only, although dynamic reconfiguration may apply infinitely, we use only finite models and scenarios for verification purposes in this section.

3.1 Specification in LNT

The specification can be divided into three parts: data types (200 lines), functions (800 lines), and processes (1,200 lines). Most processes are generated for each input application model[5], because a part of the LNT code depends on the number of VMs and on their identifiers. Therefore, the number of lines for processes grows with the number of VMs in the application model. We have given above the number of lines for an example with three VMs.

Data types are used to describe the application model (VMs, components, ports) and the communication model (messages, buffers, and topics). We show below a few examples of data types. An application model (`TModel`) consists of a set of virtual machines (`TVM`). Each VM has an identifier (`TID`) and a set of components (`TSoftware`).

```
type TModel is set of TVM end type
type TVM is tvm (idvm: TID, cs: TSoftware) end type
type TSoftware is set of TComponent end type
```

Functions apply on to data expressions and define all the computations necessary for reconfiguration purposes (*e.g.*, changing the state of a component, extracting/checking information in import/export topics, adding/retrieving messages from buffers, etc.). Let us show an example of function that aims at removing the oldest message from a FIFO buffer. This function takes as input a buffer (`TBuffer`) that is composed of an identifier (`TID`) and a list of messages (`TMessage`). If the buffer is empty, nothing happens. When the buffer is not empty, the first message is removed.

```
function remove (q: TBUFFER): TBUFFER is
  case q in
     var name:TID, hd: TMessage, tl: TQueue in
        | tbuffer(name,nil)         -> return tbuffer(name,nil)
        | tbuffer(name,cons(hd,tl)) -> return tbuffer(name,tl)
  end case
end function
```

Processes are used to specify the different participants of the protocol (a deployment manager, a publish-subscribe communication media, and an agent per

[5] We developed an LNT code generator in Python for automating this task.

VM). Each participant is specified as an LNT process and involves two kinds of actions, that are either interactions with other processes or actions to tag specific moments of the protocol execution such as the VM instantiation, the effective binding/unbinding of an import to an export, the component start-up/shutdown, the destruction of a VM, etc.

For illustration purposes, we give an example of main process involving three VMs. This process describes the parallel composition (`par` in LNT followed by a set of synchronization messages) of the protocol participants. We can see that all the agents do not interact directly together and evolve independently from one another. VM agents interact together through the PS. The DM is aware of the VMs existing in the system (parameter `appli`). Each agent is identified using the VM name, and the PS is initialised with a buffer per VM and two topics for imports/exports (`ListBuffers`). Each process also comes with an alphabet corresponding to the actions belonging to its behaviour. For instance, the DM defines actions for VM creation and destruction (`INSTANTIATEVMi` and `DESTROYVM`, resp.). Each agent defines actions for port binding (`BINDCOMPO`), for starting a component (`STARTCOMPO`), for stopping a component (`STOPCOMPO`), etc., as well as interactions with the PS (`AGENTtoPSi` when sending a message to the PS and `PStoAGENTi` when receiving a message from it). All these actions are used for analysing the protocol as we will see in the next subsection.

```
process MAIN [INSTANTIATEVM1:any, DESTROYVM:any, STARTCOMPO:any,..] is
   par INSTANTIATEVM1, ..., INSTANTIATEVM3, DESTROYVM in
       DM [INSTANTIATEVM1, ..., INSTANTIATEVM3, DESTROYVM] (appli)
   ||
       par AGENTtoPS1, PStoAGENT3, ... in
           par
               Agent[INSTANTIATEVM1, AGENTtoPS1, PStoAGENT1,
                     DESTROYVM, STARTCOMPO, BINDCOMPO, STOPCOMPO,
                     UNBINDCOMPO] (vm1)
           ||
               Agent[...] (vm2)
           ||
               Agent[...] (vm3)
           end par
       ||
           PS[AGENTtoPS1, ..., PStoAGENT3] (!?ListBuffers)
       end par
   end par
end process
```

3.2 Verification using CADP

To verify the protocol, we have first identified and specified 35 properties in MCL [15], the temporal logic used in CADP. MCL is an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators. We distinguish properties dedicated to start-up scenarios (Prop. 1

and 2 below for instance), destruction scenarios (Prop. 4), and mixed scenarios (Prop. 3). All these properties aim at verifying different parts of the protocol. Some of them focus on the protocol behaviour for checking for example that final objectives are fulfilled (Prop. 1 below) or progress/ordering constraints respected (Prop. 3 and 4). Other properties guarantee that architectural invariants for the application being reconfigured are always satisfied (Prop. 2).

For each application model and reconfiguration scenario taken from our dataset of examples, we generate an LTS by applying the LNT specification to this example and generating all the possible executions using CADP exploration tools. Finally, we use the Evaluator model checker that automatically says whether these properties are verified or not on that LTS. When a bug is detected by model checking tools, it is identified with a counterexample (a sequence of actions violating the property). Let us present some concrete properties verified on the application model presented in Figure 4:

1. All components are eventually started.

   (μX . (**<** **true** **>** **true** **and** [**not** `"STARTCOMPO !Apache !VM1"`] X))
   and
   ...
   and
   (μX . (**<** **true** **>** **true** **and** [**not** `"STARTCOMPO !MySQL !VM3"`] X))

   This property is automatically generated from the application model because it depends on the name of all VMs and components hosted on each VM.

2. A component cannot be started before the component it depends on for mandatory imports.

   [
   true* . `"STARTCOMPO !Apache !VM1"` .
   true* . `"STARTCOMPO !Tomcat !VM2"`
   ]

   The Apache component is connected to the Tomcat component on a mandatory import, therefore we will never find a sequence where Apache is started before Tomcat. This property is automatically generated from the application model because it depends on the component and VM names in the application model.

3. There is no sequence where an import (mandatory or optional) is bound twice without an unbind in between.

   [**true*** .
   `"BINDCOMPO !Apache !WORKERS"` .
   (**not** `"UNBINDCOMPO !Apache !VM1"`)* .
   `"BINDCOMPO !Apache !WORKERS"`
   ] **false**

   When a component is connected to another component through an import, it cannot be bound again except if it is stopped and unbound before.

4. A component hosted on a VM eventually stops after that VM receives a destruction request from the DM.

   (< **true*** . {`DESTROYVM ?vm:String`} .
   true* . {`STOPCOMPO ?cid:String !vm`} > **true**)

   This property does not depend on the application. Parameters can be related in MCL by using variables in action parameters (*e.g.*, `vm` for the virtual machine identifier). This property shows the data-based features that are available in MCL.

3.3 Experiments

Experiments were conducted on more than 600 hand-crafted examples on a Pentium 4 (2.5GHz, 8GB RAM) running Linux. For each example, the reconfiguration protocol takes as input the application and a specific scenario (a sequence of instantiate/destroy VM operations). The corresponding LTS is generated using CADP exploration tools by enumerating all the possible executions of the system. Finally, the verification tools of the CADP toolbox are called, providing as result a set of diagnostics (true or false) as well as counterexamples if some verifications fail. Let us note that for validating the protocol we used a large variety of examples, ranging from simple ones to pathological models and scenarios in order to check boundary cases.

Table 1 summarizes some of the numbers obtained on illustrative examples of our dataset. The application model used as input to our protocol is characterised using the number of virtual machines (vm), components (co), imports (imp), exports (exp), and reconfiguration operations (op). Then we give the size of the LTS before and after minimization (*wrt.* a strong bisimulation relation). The last column gives the time to execute the whole process (LTS generation and minimization on the one hand, and properties checking on the other).

It is worth observing that the size of LTSs and the time required for generating those LTSs increase with the size of the application, particularly with the number of VMs and the number of ports that can be connected: the more VMs and ports, the more parallelism in the system. Increasing the number of reconfiguration operations yields more complicated scenarios, and this also increases the LTS size and generation time. Let us look at examples 0219, 0222, and 0227 in Table 1 for instance. When we slightly increase the number of components and ports in the application, we see how LTS sizes and analysis time (generation and verification) gradually grow. We can make a similar statement when comparing examples 0227 and 0228. These two examples are exactly the same, but one more reconfiguration is achieved in 0228, resulting in a noticeable grow in the corresponding LTS size and analysis time. Example 0453 shows how this time can take up to several hours. Fortunately, analysing huge systems (with potentially many VMs) is not the most important criterion during the verification of the protocol. Indeed, most issues are usually found on small applications describing pathological reconfiguration cases.

	Size					LTS (states/transitions)		Time (m:s)	
	vm	co	imp	exp	op	raw	minimized	LTS gen.	Verif.
0047	2	3	1	2	4	3,489/6,956	836/1,472	0:23	0:15
0219	3	3	2	2	5	28,237/68,255	2,775/6,948	0:35	0:48
0222	3	4	4	4	5	622,592 /1,416,167	10,855/32,901	12:15	2:40
0227	3	6	9	7	5	783,784/1,484,508	15,334/45,812	21:21	3:45
0228	3	6	9	7	6	802,816 / 1,629,118	17,923/54,143	29:25	4:10
0453	4	8	7	5	8	1,643,248 /2,498,564	68,468/227,142	153:12	28:22

Table 1. Experimental results

3.4 Problems Found

The specification and verification of the protocol using model checking techniques enabled us to revise and improve several parts of the protocol. Beyond correcting several very specific issues in the protocol (*e.g.*, adding some acknowledgement messages after effectively binding ports), we will comment in this section on two important issues we found out during the verification steps and that were corrected in the latest version of the protocol (the one presented in this paper), both in the specification and implementation.

In the initial version of the protocol, the component start-up/shutdown was guided by a centralised DM. More precisely, the DM kept track of the current state (bindings and component states) for each VM. To do so, each VM sends messages to the DM whenever a change is made in its VM, *e.g.*, a stopped component is started. As a consequence, the DM has an overall view of the current state of the system and can send messages to VMs in order to trigger a component start-up/shutdown (when dependencies and other component states permit that). An important drawback of this centralised version is that it induces an overhead of messages transmitted to and from the DM. This was observed during our experiments analysing the size of the corresponding state spaces: some quite simple examples resulted in huge LTSs. This issue was solved by proposing a decentralised version of the protocol, where the DM is not in charge of starting and stopping components any more. This task is delegated to the VM agents. This avoids additional, unnecessary messages exchanged between agents and the DM. The decentralised version of the protocol presents several advantages: more parallelism, better performance in the corresponding implementation of the protocol, and simplification in terms of number of communications (smaller LTSs).

We also detected a major bug in the way VMs are destroyed. Originally, when it was required to stop a component, it was stopped before the components bound to it. Stopping components in this order typically violates the consistency of the component composition and well-formedness architectural invariants. This may result for instance in started components connected to and therefore submitting requests to stopped components. This problem was detected thanks to a property stating that "*a component cannot be started and connected through an import*

(mandatory or optional) to another component, if that component is not started". In many cases, we observe that this property was not satisfied, particularly for application models and reconfiguration scenarios requiring to stop components in sequence across several VMs after reception of a VM destruction request. We corrected this issue by stopping properly components. This required a deep revision of the protocol. Thus, in the current version of the protocol, when a component must stop, it requests to all components connected to it to unbind and once it is done, it can finally stop. This implies first a backward propagation along components bound on mandatory imports. Once this first propagation stops, we start a forward propagation during which components are actually stopped and indicate to their partners that they have just stopped and unbound. This double propagation, as presented in Section 2.3, is necessary for preserving the component architecture consistency and for avoiding that started components can keep on using stopped components.

4 Related Work

First of all, let us mention some related papers [10, 5, 16] where are presented languages and configuration protocols for distributed applications in the cloud. [5] adopts a model driven approach with extensions of the Essential Meta-Object Facility (EMOF) abstract syntax to describe a distributed application, its requirements towards the underlying execution platforms, and its architectural constraints (*e.g.*, concerning placement and collocation). The deployment works in a centralised fashion. [16] suggests an extension of SmartFrog [10] that enables an automated and optimised allocation of cloud resources for application deployment. It is based on a declarative description of the available resources and of the components building up a distributed application. Descriptions of architectures and resources are defined using the Distributed Application Description Language. This paper does not give any details concerning the deployment process.

A recent related work [8] 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 [8] does not care about composition consistency issues, that is, their framework does not preserve architectural invariants ensuring for instance that a started component is never connected to a stopped component.

In [12–14, 1, 20, 3, 17], the authors proposed various formal models (Darwin, Wright, etc.) in order to specify dynamic reconfiguration of component-based systems whose architectures can evolve (adding or removing components and connections) at run-time. These techniques are adequate for formally designing dynamic applications. In [12, 14] for instance, the authors show how to formally analyse behavioural models of components using the Labeled Transition System Analyser. Our focus is quite different here, because we work on a protocol whose goal is to automatically achieve these reconfiguration tasks, and to assure that this protocol respects some key properties during its application.

In [6, 7, 19], the authors present a protocol that automates the configuration of distributed applications in cloud environments. In these applications, all elements are known from the beginning (*e.g.,* numbers of VMs and components, bindings among components, etc.). Moreover, this protocol allows one to automate the application deployment, but not to modify the application at run-time. Another related work is [2], where the authors propose a robust reconfiguration protocol for an architectural assembly of software components. This work does not consider the distribution of components across several VMs, but assume they are located on a single VM.

5 Conclusion

We have presented in this paper a protocol for dynamically reconfiguring distributed cloud applications. This protocol enables one to instantiate new VMs and destroy existing VMs. Upon reception of these reconfiguration operations, VM agents connect/disconnect and start/stop components in a defined order for preserving the application consistency, which is quite complicated due to the high parallelism degree of the protocol. Therefore, we have specified and verified this protocol using the LNT specification language and the CADP toolbox, which turned out to be very convenient for modelling and analysing such protocols, see [18] for a discussion about this subject. Model checking techniques were used to verify 35 properties of interest on a large number of application models and reconfiguration scenarios. This helped to improve several parts of the protocol and to detect subtle bugs. In particular, we deeply revise the part of the protocol dedicated to the VM destruction and component shutdown. These issues have also been corrected in the corresponding Java implementation.

As far as future work is concerned, we first plan to add finer-grained reconfiguration operations in order to enable the deployment manager to not only add and remove virtual machines, but also to add and remove components on already deployed VMs. Another perspective aims at extending our protocol for handling VM failures.

Acknowledgements. This work has been 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. R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proc. of FASE'98*, volume 1382 of *LNCS*, pages 21–37. Springer, 1998.
2. F. Boyer, O. Gruber, and G. Salaün. Specifying and Verifying the Synergy Reconfiguration Protocol with LOTOS NT and CADP. In *Proc. of FM'11*, volume 6664 of *LNCS*, pages 103–117. Springer, 2011.
3. A. Cansado, C. Canal, G. Salaün, and J. Cubo. A Formal Framework for Structural Reconfiguration of Components under Behavioural Adaptation. *Electr. Notes Theor. Comput. Sci.*, 263:95–110, 2010.

4. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, V. Powazny, F. Lang, W. Serwe, and G. Smeding. Reference Manual of the LOTOS NT to LOTOS Translator (Version 5.4). INRIA/VASY, 2011.
5. C. Chapman, W. Emmerich, F. Galán Márquez, S. Clayman, and A. Galis. Software Architecture Definition for On-demand Cloud Provisioning. In *Proc. of HPDC'10*, pages 61–72. ACM Press, 2010.
6. X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-Configuration of Distributed Applications in the Cloud. In *Proc. of CLOUD'11*, pages 668–675. IEEE Computer Society, 2011.
7. 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.
8. J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A Deployment Management System. In *Proc. of PLDI'12*, pages 263–274. ACM, 2012.
9. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2010: A Toolbox for the Construction and Analysis of Distributed Processes. In *Proc. of TACAS'11*, volume 6605 of *LNCS*, pages 372–387. Springer, 2011.
10. P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The SmartFrog Configuration Management Framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, 2009.
11. ISO/IEC. LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, 1989.
12. J. Kramer and J. Magee. Analysing Dynamic Change in Distributed Software Architectures. *IEE Proceedings - Software*, 145(5):146–154, 1998.
13. J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In *Proc. of SIGSOFT FSE'96*, pages 3–14. ACM, 1996.
14. J. Magee, J. Kramer, and D. Giannakopoulou. Behaviour Analysis of Software Architectures. In *Proc. of WICSA'99*, volume 140 of *IFIP Conference Proceedings*, pages 35–50. Kluwer, 1999.
15. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*, pages 148–164. Springer, 2008.
16. J. Mirkovic, T. Faber, P. Hsieh, G. Malayandisamu, and R. Malavia. DADL: Distributed Application Description Language. USC/ISI Technical Report ISI-TR-664, 2010.
17. G. Salaün. Generation of Service Wrapper Protocols from Choreography Specifications. In *Proc. of SEFM'08*, pages 313–322. IEEE Computer Society, 2008.
18. G. Salaün, F. Boyer, T. Coupaye, N. De Palma, X. Etchevers, and O. Gruber. An Experience Report on the Verification of Autonomic Protocols in the Cloud. *ISSE*, 9(2):105–117, 2013.
19. G. Salaün, X. Etchevers, N. De 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.
20. 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, 2001.