Formal Design of Dynamic Reconfiguration Protocol for Cloud Applications

Rim Abid, Gwen Salaün, Noel De Palma

University of Grenoble Alpes, France

Abstract

Cloud applications are complex applications composed of a set of interconnected software components running on different virtual machines, hosted on remote physical servers. Deploying and reconfiguring this kind of applications are very complicated tasks especially when one or multiple virtual machines fail when achieving these tasks. Hence, there is a need for protocols that can dynamically reconfigure and manage running distributed applications. In this article, we present a novel protocol, which aims at reconfiguring cloud applications. This protocol is able to ensure communication between virtual machines and resolve dependencies by exchanging messages, (dis)connecting, and starting/stopping components in a specific order. The interaction between machines is assured via a publish-subscribe messaging system. Each machine reconfigures itself in a decentralized way. The protocol supports virtual machine failures, and the reconfiguration always terminates successfully even in the presence of a finite number of failures. Due to the high degree of parallelism inherent to these applications, the protocol was specified using the LNT value-passing process algebra and verified using the model checking tools available in the CADP toolbox. The use of formal specification languages and tools helped to detect several bugs and to improve the protocol.

Key words: Cloud Computing, Dynamic Reconfiguration, Distributed Applications, Fault-Tolerance, Verification

Email addresses: Rim.Abid@inria.fr (Rim Abid), Gwen.Salaun@inria.fr (Gwen Salaün), Noel.Depalma@imag.fr (Noel De Palma)

1. Introduction

Cloud computing leverages hosting platforms based on virtualization and provides resources and software applications as service over the network (such as the Internet). It allows users to benefit from these services without requiring expertise in each of them. For service providers, this gives the opportunity to develop, deploy, and sell cloud applications worldwide without having to invest upfront in expensive IT infrastructure.

Cloud applications are intricate distributed applications composed of a set of virtual machines (VMs) running a set of interconnected software components. Cloud users need to (re)configure and monitor applications during their time life for elasticity or maintenance purposes. Therefore, after deployment of these applications, some reconfiguration operations are required for setting up new virtual machines, replicating some of them, destroying or adding virtual machines, handling VM failures, and adding or removing components hosted on a VM. Some of these tasks are executed in parallel, which complicates their correct execution.

Existing protocols [15, 17, 31] focus mainly on self-deployment issues where the model of application (the number of virtual machines, components, ports, and connections between components) is known before the application execution. These approaches manage static applications which do not require to be changed after the deployment phase. Existing deployment solutions barely take into account configuration parameters. Unlike these static applications, cloud applications need to be reconfigured in order to include new requirements, to fulfill the users expectations, or to perform failure recovery. More specifically, cloud users need protocols that are not only limited to deploy specific applications, but that are also able to modify applications during their execution and take into account the changes that can occur such as the failure of some virtual machine.

In this article, we introduce a novel protocol which aims at automatically deploying and (re)configuring applications in the cloud. These applications are composed of multiple and interconnected software components hosted on separate virtual machines. A reconfiguration manager guides the reconfiguration tasks by instantiating new VMs or destroying/repairing existing VMs. The reconfiguration manager may also apply a number of architectural changes to the application by adding new components or removing existing components hosted on a specific VM. After the creation of a component due to the instantiation of a VM or to a component addition request, the proto-

col is responsible for starting all components in the correct order according to the architectural dependencies. Each VM embeds a local reconfiguration agent that interacts with the other remote agents. For each component, the VM agent tries to satisfy its required services by connecting them to their providers in order to start the component. The component cannot be started before the components it depends on. The provider of the service can be hosted on the same VM or on another VM. When a VM receives a VM destruction or a component removal request from the reconfiguration manager, it tries to stop and unbind each component. A component cannot stop before all partner components connected to it have unbound themselves. In order to exchange messages and bind/start/unbind/stop components, VMs communicate together through a publish-subscribe messaging system. The protocol is also able to detect VM failures that occur to a running application. When a VM failure occurs, the protocol notifies the VMs that are impacted. The protocol supports multiple failures. It always succeeds in finally reconfiguring the application at hand and stopping/starting all components.

Our management protocol implies a high degree of parallelism. Hence, we decided to use formal techniques and tools to specify, verify the protocol, and ensure that it preserves important architectural invariants (*e.g., a started component cannot be connected to a stopped component)* and satisfies certain properties (*e.g., each VM failure is followed by a new creation of that VM*). The protocol was specified using the specification language LOTOS NT (LNT for short) [12], which is an improved version of LOTOS [20], and verified with the CADP verification toolbox [18]. For verification purposes, we used 600 hand-crafted examples (application models and reconfiguration scenarios). For each example, we generated the Labelled Transition System (LTS) from the LNT specification and we checked on them about 40 properties that must be respected by the protocol during its application. To do so, we used the model checking tools available in the CADP toolbox. These formal techniques and tools helped us improve the protocol by (i) detecting several issues and bugs, and by (ii) correcting them systematically in the corresponding Java implementation.

Our main contributions with respect to related approaches are the following:

- We propose a novel protocol, which reconfigures cloud applications consisting of a set of interconnected software components distributed over remote virtual machines.

- The protocol is fault-tolerant in the sense that it is able to detect and repair VM failures.
- The protocol was verified using model checking techniques and is implemented in Java.

The outline of this article is as follows. Section 2 introduces the reconfiguration protocol and presents how it works on some concrete applications. We present in Section 3 the LNT specification of the protocol and its verification using CADP. Section 4 reviews related work and Section 5 concludes the article.

2. Reconfiguration Protocol

This section successively presents the application model, the protocol participants, the protocol itself including the different possible reconfiguration operations, and an overview of the Java implementation.

2.1. Application Model

In this section, we present an abstraction of the model, which is sufficient for explaining the protocol principles. The real model exhibits more details such as port numbers, URLs, and other implementation details. The model we use here is used for verifying the soundness of the protocol but its primary role is to keep track of the VMs and components currently deployed in a cloud application.

An application is composed of a set of interconnected software components hosted on different virtual machines (VMs). Each component may provide services via exports and require services via imports. Ports are typed and match when they share the same type. One export can provide its service to several components and can thus be connected to several imports. An import must be connected to one export only provided by a component hosted on the same machine (local binding) or a component hosted on another machine (remote binding). When many exports provide the same service, the import is bound to one of them which is randomly chosen by the publish-subscribe system. The import can be mandatory or optional. A mandatory import represents a service required by the component to be functional. Therefore, if the component needs mandatory imports, it cannot be started before all its imports are satisfied (*i.e.*, all mandatory imports are connected to started components). On the other side, an optional import is a service needed by

the component but not necessary for starting it. When a component has an optional service, it can be started before the component that provides it, which makes our model supports circular dependencies that involve at least one optional import. Hence, the connection of an optional import may be achieved after the start-up of the service requester and provider. Both kinds of ports are usual in component-based applications. Our protocol does not distinguish port types during the port resolution and (un)binding phases, but does when starting / stopping components.
A component used in this model has three states, namely, stopped, started, and failed. Initially, when the VM is instantiated, all its hosted components are stopped. For each component, once all its mandatory imports (if any mandatory import exists) are bound to started components, the component can be started. Reversely the component must stop when at least one of the components to which it is connected through mandatory import requires to stop. When the VM fails, all its components are failed.
This model is classic for designing autonomous protocols and has been used for describing, deploying, and reconfiguring real-world applications (*see e.g.*, [21, 9, 28, 15, 17, 31]). It is expressive enough for reconfiguration purposes, but should be enhanced for other tasks. For instance, if one's interest is quality of service, we would need more information in the model.

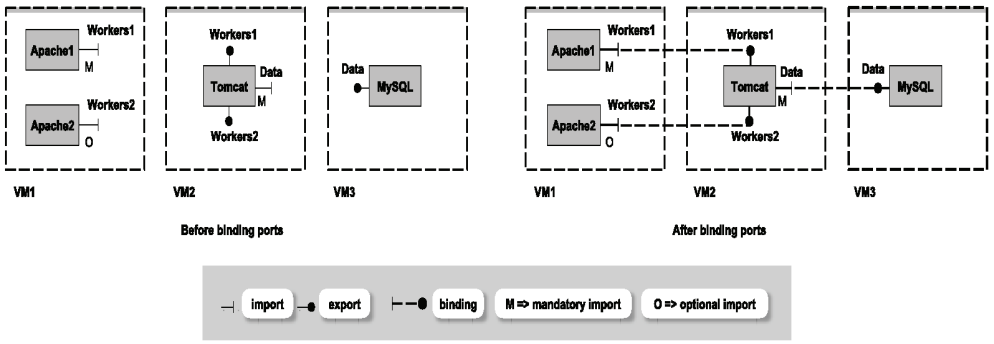

Figure 1: Example of an application model

Figure 1 gives an example of an application model, which consists of three VMs. The first one (VM1) hosts two components Apache1 and Apache2.

Apache1 has a mandatory import that is Workers1. Therefore, Apache1 cannot be started before the start-up of the Workers1 provider. Apache2 has an optional import that is Workers2. Thus, it can be started before the start-up of Workers2 provider. Both VM2 and VM3 are hosting one component (Tomcat and MySQL respectively). Tomcat provides two services that are Workers1 and Workers2 and has a mandatory import that is Data. Therefore, Tomcat cannot be started before the Data provider. MySQL does not require any service. Therefore, it can be started immediately.

2.2. Protocol Participants

The management protocol includes three participants as showed in Figure 2. The reconfiguration manager (RM) guides the application reconfiguration by posting reconfiguration operations (*e.g.,* instantiating/destroying VMs, adding/removing components) that are described in a reconfiguration scenario. The RM is also in charge of repairing a VM failure by creating a new instance of the failed VM. Each VM is equipped with a reconfiguration agent (agent for short in the rest of this article) that is in charge of (dis)connecting bindings and starting/stopping components upon reception of VM instantiation/destruction and adding/removing reconfiguration operations from the RM. The publish-subscribe messaging system (PS) supports the exchange of communications among all VMs. The PS transfers messages from a machine to others machines. To do so, it contains a list of buffers (a buffer for each VM). Each buffer is used to store the messages exchanged between the agent of its VM and the other agents. When a new VM is instantiated, a buffer for that VM is added to the PS. When an existing VM is destroyed, its buffer is removed from the PS. We assume that the communication model is reliable and messages are never lost during the communication between all participants. The PS also contains two topics: [1] (i) the export topic where a component publishes its exports and subscribes its imports, and (ii) an import topic where a component publishes its imports and subscribes its exports.

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

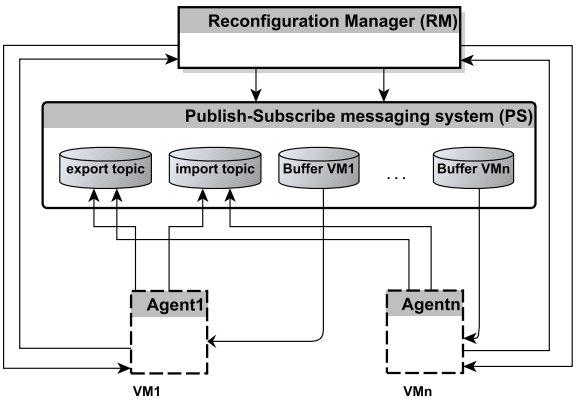

Figure 2: Protocol participants

2.3. Protocol Description

The protocol allows to instantiate new VMs/destroy existing VMs, to add a new component to an existing VM or to remove a component from its hosting VM. The protocol also supports VM failures. We explain in this section how the protocol works in these different situations.

2.3.1. Start-up

The RM guides the application reconfiguration by instantiating new VMs and also adding components to an existing VM. In both cases, the goal of the VM agent is to start components.

We explain first how the protocol works in order to add a component to a running application and start it. We present in Figure 3 the agent behavior when the component has mandatory imports only. First the agent receives an add component request (❶) from the RM. When the component provides services, for each service, it subscribes to the import topic and then publishes the service to the export topic (❷). Then, if the component does not have any import, it starts immediately. Otherwise, each mandatory import requires an export with the same type provided by another component hosted on the same VM or on another VM. To be functional, the component expects that all its mandatory imports will be connected to started components. Therefore, for each mandatory import, the component subscribes to the export topic and then publishes the import to the import topic (❸). The PS receives that message and checks the import topic. If it does not find a provider for the

service, the publication message is deleted. If it finds more than one service provider, the PS picks one randomly and sends a message with binding details and component state. Upon reception of that message (❹), the import binds to this compatible export (❺). Then, the VM agent sends a binding message (❻) to the PS. When all the mandatory imports are connected and all the partner components have been started, the VM agent starts the component (❼) and sends a started message to the PS (❽).
When the component has also optional imports, it can start even if these imports are not satisfied. For each optional import, it subscribes to the export topic and publishes the import to the import topic. Upon reception of a message with binding details and component state, the import binds to this compatible export only if the component state is started.

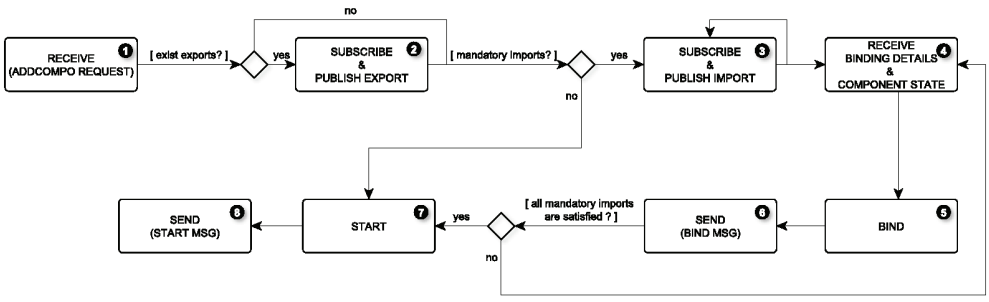

Figure 3: The Agent behavior when adding a new component with mandatory imports only

Second, we present how the protocol works to add a new VM to a running application. Once the RM instantiates a new VM, a buffer for this VM is added to the list of the PS buffers. After the creation of the new VM, its agent is in charge of starting all the components hosted on this VM. For this, it considers each component hosted on the new VM like a new component that will be added to an existing VM (see explanations above).

2.3.2. Shutdown

The second goal of the protocol is to remove existing components from a VM or to destroy an existing VM from a running application.

We present first how the protocol works when the RM requires the removal of a component from a VM as depicted in Figure 4. When the agent receives the remove component request from the RM (❶), it tries to stop the component without violating any architectural invariants (*e.g., a started component cannot be connected to a stopped component*). To do so, if the component does not provide any export, it stops immediately (❺). Then it unbinds (❻) all its connections and sends an *"unbind confirmed"* message for each import (❼). When the component provides services, before stopping it, all components bound to it through mandatory imports must stop and then unbind in order to restore consistency of the whole application. Components bound to it through optional imports, need only to unbind without stopping. Therefore, for each export, it unsubscribes from the import topic (❷) and then sends *"an ask to unbind"* message (❸) to the PS. Then , it waits until all components bound to it through mandatory imports stop and then disconnect. Components bound to it through optional imports need only to disconnect. When the PS receives an *"ask to unbind"* message, it transmits it to all components subscribed to that export. It notifies the component that need to stop when another component connected to it has unbound by sending an *"unbind confirmed"* message (❹). Once all components connected to it have effectively unbound, it can stop itself (❺), unbind all its imports (❻), and send an *"unbind confirmed"* message to the PS (❼).

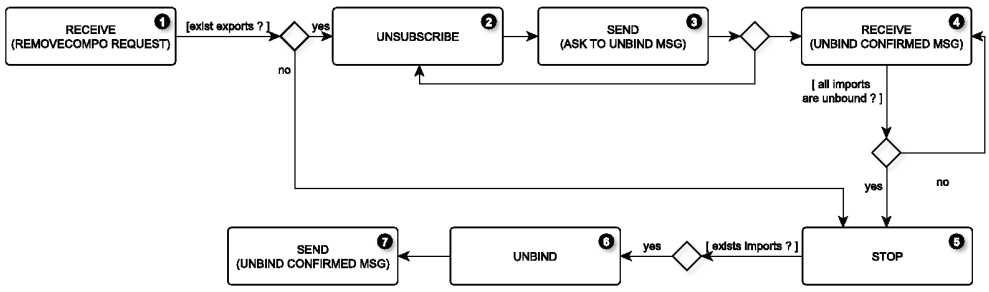

Figure 4: The agent behavior when removing a component

When a component receives an *"ask to unbind"* message (Figure 5), it can remain in the same state and unbinds directly if it is bound through

an optional import. If it is bound through a mandatory import, the protocol handles it as done for the component that needs to be removed. The only differences between stopping a component that will be removed and a component that receives an *"ask to unbind"* message are that, the second one should not unsubscribe from the import topic if it provides exports and after unbinding, it publishes its import in order to connect to another export proposed by the publish-subscribe system (❼).

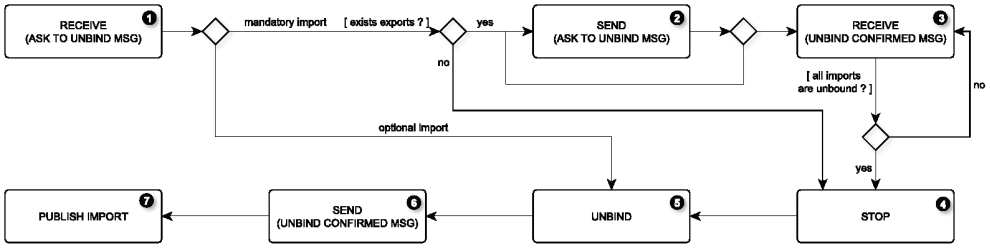

Figure 5: The agent behavior when stopping a component

The component shutdown implies a backward propagation of *"ask to unbind"* messages and, when this first propagation ends (on components with no export or optional imports only), a second forward propagation of *"unbind confirmed"* messages starts to let the components know that the disconnection has been actually achieved.

When the RM sends a destruction VM request, the VM agent must properly stop all the components hosted on that VM as well as all components bound to them. Therefore, for each component hosted on the VM that will be destroyed, it behaves as if it has received a remove component request. Once all components are removed, the VM is finally destroyed (see explanations above).

2.3.3. VM Failures

VM failures and component failures can occur at any time to a running application (*e.g.*, during a reconfiguration). We present in this section only the VM failure. A component failure is a specific case of a VM failure, so this is not a fundamental addition to the current protocol. When a VM fails,

all its hosted components stop unexpectedly without alerting any partner components. Therefore, we can be in a situation where started components are connected to failed components. In order to restore the application consistency (*i.e.*, architectural invariants), VM failures must be handled immediately. We explain in this section how the protocol takes into account VM failures and how it works to manage them (Fig. 6). The protocol supports also the case of multiple failures.

When a VM is instantiated, its agent interacts with the RM by periodically sending an *"alive"* message. The RM is configured to receive this message periodically from each agent. As long as the RM receives these *"alive"* messages from each agent within a specific delay, it considers that its VM is working correctly and still alive [6] (❶). Once the RM does not receive one of these *"alive"* messages from an agent before the delay elapses, it assumes that its VM has failed (❷). In that case, it alerts the PS (❸) and then starts the repair phase by creating a new instance of the failed VM (❹). The new VM behaves as if the machine is instantiated for the first time (see Section 2.3.1).

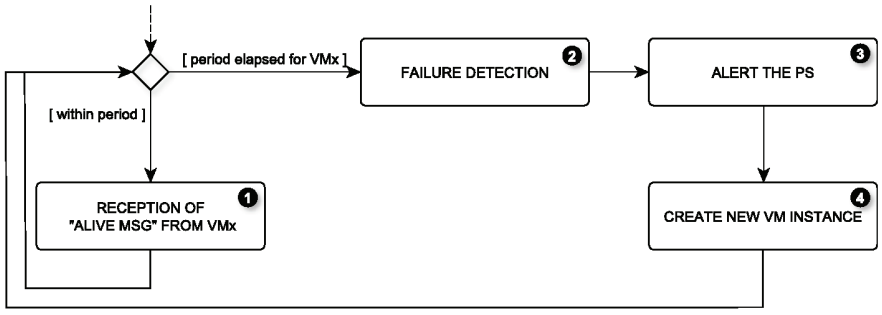

Figure 6: The RM behavior when detecting a VM failure

When the PS receives an *"alert failure"* message from the RM (❺), it unsubscribes each failed component exporting services from the import topic to avoid that other components try to connect to it (❻). Then, it warns the other VMs of this failure by:

- sending messages notifying all components bound to the failed components about the VM failure and asking them to unbind (❼). When

an agent receives this kind of message, it tries to stop the impacted component if it is bound through a mandatory import or remains in the same state if it is bound through optional import. Then it unbinds it and tries to connect the disconnected import in order to restart the component as presented in Section 2.3.2;

- sending an *"unbind confirmed"* message to each component to which a failed component was connected via mandatory or optional import (❽). The reception of this message by a component that needs to stop indicates the disconnection has been actually achieved.

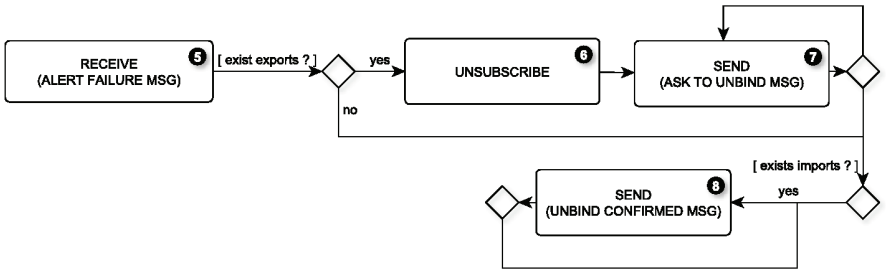

Figure 7: The PS behavior when receiving an alert failure message

In the case of a component failure, the agent of the VM on which it is hosted informs the PS and then creates a new instance of the failed component. The new component behaves as a component added to the application for the first time. When the PS receives the component failure alert, it reacts in the same way as for each component hosted on a failed VM.

2.4. Examples

We present in this section an example of removal and addition of a component to the application showed in Figure 1, as well as an example of VM failure. In this example, we begin with a running application where all components are connected and started.

2.4.1. A scenario of component removal

As it is depicted in Figure 8, the RM requests the VM2 agent to remove Tomcat (RM.1). Once VM2 receives the request, it aims to stop the Tomcat component before removing it. To do so, all components connected to the Tomcat must be stopped before. Tomcat provides two services Workers1 and Workers2. Thus, for each export, it unsubscribes from the import topic (A2.1 and A2.3) and sends messages to the PS asking it to unbind all components connected to it through Workers1 and Workers2 (A2.2 and A2.4). The PS receives these messages, checks the export topic and finds that Apache1 (Apache2 respectively) hosted on VM1 imports Workers1 (Workers2 respectively) from Tomcat. Therefore, it sends *"ask to unbind"* messages to Apache1 and Apache2 (PS.1 and PS.2). When VM1 receives these messages, Apache1 does not provide any service and it is bound to the Tomcat component through a mandatory import, so it is immediately stopped (A1.1). Then, it is unbound from Tomcat, sends an *"unbind confirmed"* message to the PS (A1.2), and publishes its import to the import topic (A1.3). VM2 receives that message from the PS (PS.3) but cannot stop the Tomcat component because Apache2 is still connected to it. Apache2 is connected to the Tomcat component through an optional import. Thus, it is only unbound from Tomcat without stopping, then sends an *"unbind confirmed"* message to the PS (A1.4), and publishes its import to the import topic (A1.5). The PS checks the import topic but there is no component that provides Workers1 or Workers2. VM2 receives the *"unbind confirmed"* message from the PS (PS.4). Tomcat has no component bound to it any more, so it is stopped (A2.5). Tomcat is unbound from MySQL and the VM2 agent sends an *"unbind confirmed"* message to the PS (A2.6). The PS finally sends that message to the MySQL component (PS.5).

2.4.2. A scenario of component addition

We present in this section an example of component addition to an existing VM. We show how to add a new instance of Tomcat to VM2 after removing it in Section 2.4.1.

VM2 receives the Tomcat addition request from the RM (1). The Tomcat component requires a mandatory service whose type is Data. Therefore, it subscribes to the export topic and then publishes its import to the import topic (A2.1). Tomcat provides two services Workers1 and Workers2. There-

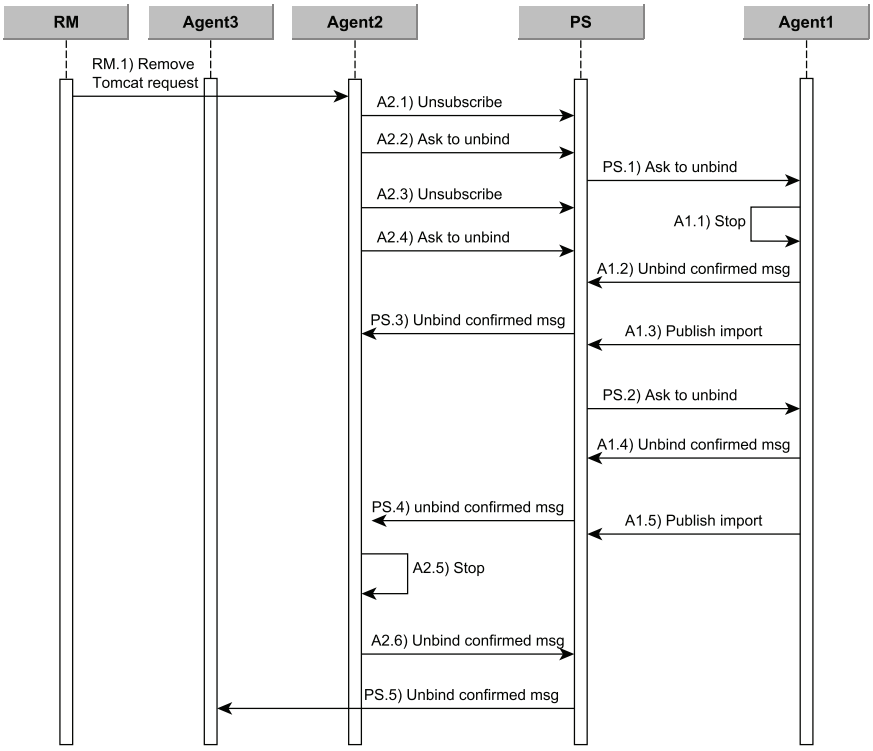

Figure 8: The participants behavior when removing the Tomcat component

fore, for each export it subscribes to the import topic and then publishes it to the export topic (A2.2 and A2.3). The PS receives the first message from the VM2 agent, checks the import topic and finds that only MySQL provides a Data service (it has subscribed to the import topic). The PS notifies VM3 that there is a Tomcat that needs Data (PS.1). When Tomcat publishes its exports, the PS forwards the binding details and Tomcat state to Apache1 because it has subscribed to the export topic for Workers1 (PS.2) (Apache2 respectively because it has subscribed to the export topic for Workers2 (PS.3)). When VM1 receives the binding details and Tomcat state, Apache1 connects to Tomcat (A1.1) but it cannot start because the Tomcat state is stopped. Apache2 cannot connect to Tomcat because it is started and Tomcat is still stopped. After receiving the notification message from the PS about the Tomcat component need (PS.1), VM3 sends the MySQL binding details and state that it is started (A3.1). The PS receives the start-up information from the VM3 agent, checks and finds that the Tomcat component has required this service (it has subscribed to the export topic). Hence, a message with binding details and MySQL's state is added to VM2 buffer (PS.4). Upon reception of this message, the Tomcat component is bound to the mySQL component (A2.4) and the VM2 agent starts the Tomcat component (A2.5). Then, the Tomcat component publishes a started message containing its new state (A2.6). The PS receives that message and forwards it to VM1 (PS.5). Upon reception of this message, the VM1 agent starts the Apache1 component (A1.2). Apache2 can finally connect to the Tomcat component (A1.3).

2.4.3. A scenario of VM Failure repair

We present in this section an example of failure/repair of VM2. When VM2 fails, Tomcat is suddenly stopped without alerting the components bound to it. Therefore, Apache1 and Apache2 that are started are bound to a stopped component. When the RM detects the failure of VM2 (RM.1), it immediately alerts the PS (RM.2). Then, the RM creates a new instance of VM2 (RM.3).

When the PS receives the alert message from the RM announcing the VM2 failure, it unsubscribes Tomcat from the import topic (PS.1). Then, it checks the export topic in order to find the impacted components. Thus, it finds that Apache1 and Apache2 are connected to Tomcat. Therefore, it notifies them about the failure (PS.2 and PS.3). When VM1 receives the notification

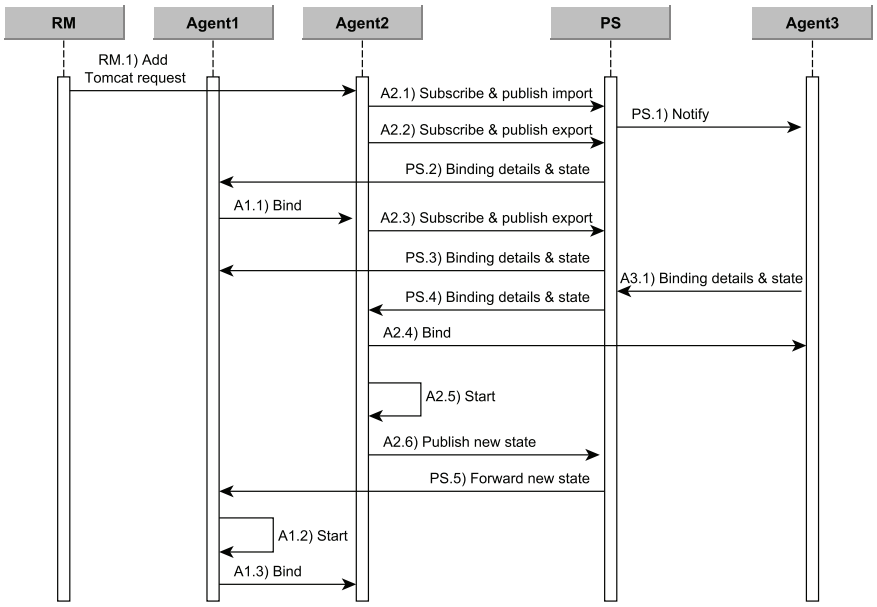

Figure 9: The participants behavior when adding the Tomcat component

messages, Apache1 does not provide any service and it is connect to Tomcat through a mandatory import so it is immediately stopped (A1.1). It is also unbound from the Tomcat component, sends an *"unbind confirmed"* message to the PS (A1.2), and then publishes its import to the import topic (A1.3). Apache2 does not provide any service and is connected to the failed component through an optional import so it is unbound from Tomcat without stopping, sends an *"unbind confirmed"* message to the PS (A1.4), and then publishes its import to the import topic (A1.5).
After the creation of a new instance of VM2, each agent starts the components impacted by the failure as presented in Section 2.3.1.

2.5. Implementation

We present in this section some implementation details concerning the protocol. The core of the system of our implementation is roughly 6K lines of Java code. It is based on a IaaS abstraction layer that allows to instantiate

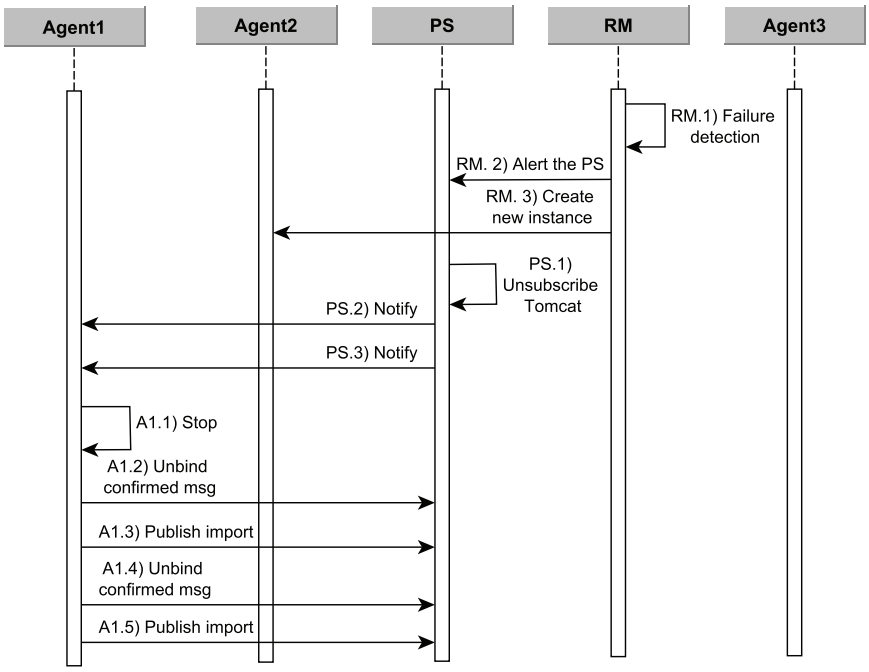

Figure 10: The participants behavior when detecting the VM2 failure

VMs on different IaaSs (such as Amazon EC2 and Microsoft Azure) and on RabbitMQ [3], which is an implementation of the AMQP standard [1]. It is a message server, implementing the publish-subscribe messaging system. It is mainly used for the communication between VMs and RM. We chose an AMQP communication system in order to have a reliable and asynchronous communication system.

When the user requests the deployment of a new software instance, the RM first checks about the VM instance defined in the software instance model. If the user asks to deploy the software application on an already existing VM instance, the RM immediately sends the model to the VM. Otherwise, the RM checks if the software type of the software instance has a VM type defined. If so, the RM asks the IaaS to instantiate one VM of this kind. Otherwise, the RM asks the IaaS to instantiate a VM from the default VM template. When the RM asks the IaaS to instantiate a VM, it creates a message buffer on the message server for that VM.

When the RM sends the software instance model to the VM instance, it serializes the software instance object and sends it to the message buffer of the VM along with the configuration files (Puppet recipes for example). This part can be done even when the VM is not running thanks to the messages stored in its message buffer. When the VM boots, the agent starts, connects to the message buffer and gets the message.

The agent on the VM gets the software instance model and the configuration files associated with it. It checks what connector it has to use to perform the operations for the software instance. A connector is a Java class that implements four simple operations: setup, update, start, and stop. A connector is independent from any software it will install, it only does basic operations. The operations related to the software are located in the configuration files. Each connector, when calling a basic operation, transmits the configuration of the model (such as variables and imported variables) to the packaging and configuration system. This mechanism enables the user to separate actual operations on VM from binding details. Multiple connectors enable users to keep the original way of installing and configuring the software. In addition, it also enables to cover almost any kind of software: from software available in Linux repositories to legacy software. We implemented a Puppet [2] connector in order to install and update software on VMs. Puppet is one of the most known configuration system. It provides a language and a program. The language enables to describe the state of a system by describing its packages, files and services.

When the installation is completed (software instances are installed on VMs), each agent publish its configuration (according to the exports) and subscribe to other remote configuration (according to the imports) by using the RabbitMQ API.

3. Specification and Verification

In this section, we present the specification of the protocol in LNT [12], an input language of CADP [18]. LNT is an improved and simplified version of LOTOS [20]. This language is expressive enough for specifying the reconfiguration protocol, in particular it supports the description of complex data types and concurrent processes. Its user-friendly notation simplifies the specification writing. LNT processes are built from actions, choices (**select**), parallel composition (**par**), looping behaviors (**loop**), conditions (**if**), and sequential composition (**;**). The communication between the protocol participants is carried out by rendezvous on a list of synchronized actions included in the parallel composition (**par**). The specification is analyzed using CADP [18]. 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. Therefore, we rely on the state-of-the-art verification tools provided by CADP to check that the protocol respects some important properties.

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

The number of lines of processes depends on the size of the application model (the number of VM, component and ports). Processes are generated

for each input application model[2], 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. The specification of the protocol consists of three parts: data types (200 lines), functions (800 lines), and processes (1,500 lines). The number given above corresponds to an example with three VMs.

3.1.1. Data types

They are used to describe the distributed application model (VMs, components, ports), the communication model (binding between components, messages, buffers, and topics), and the component states. We show below a few examples of data types. The 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
```

3.1.2. Functions

They apply on data expressions. Functions are used to define all the computations necessary for reconfiguration purposes (*e.g.*, extracting information from the application, describing buffers and basic operations on them like adding/retrieving messages, changing the state of a component, keeping track of the started/unbound components, verifying the satisfaction of the imports, etc.). Let us show, for illustration purposes, an example of function that aims at removing a message from a buffer by using the FIFO strategy. This strategy consists in removing the message from the beginning and adding a new message at the end of the buffer. The remove function takes as input a buffer (`q`) which type is (`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_MSG (q: TBUFFER): TBUFFER is
  case q in
```

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

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

3.1.3. Processes

They are used to specify the participants of the protocol (the reconfiguration manager, the publish-subscribe messaging server, and an agent per VM). The reconfiguration manager guides the application reconfiguration. Each agent drives the behavior of its VM and encodes most of the protocol functionality in order to start/stop all the components hosted on its VM. The publish-subscribe messaging system assures the communication between all VMs. Its is equipped by a set of FIFO buffers (a buffer for each VM in the application). Each participant is specified as an LNT process and involves two sort of actions: actions which correspond to interactions with the other participants such as `PStoAGENT` that presents the message transferred from the publish-subscribe messaging system processes to the agent processes, `AGENTtoPS` that presents the message transferred from the agent processes to the publish-subscribe messaging system. The second type of action taggs specific moments of the protocol execution such as the VM instantiation/destruction, the component start-up/shutdown, the component addition/removal, the effective binding/unbinding of an import to an export, the failure of a VM, etc.

For illustration purposes, we present the LNT process main (named `MAIN`) generated for an example of application model involving three VMs. The LNT parallel composition is described with `par` followed by a set of synchronization actions that must synchronize together. Two processes synchronize if they share the same action. We can see that the agents do not interact directly together and evolve independently from one another. The VM agents communicate together through the PS. Each agent is identified using the VM name and synchronizes with the PS on `AGENTtoPSi` action when sending a message to the PS and `PStoAGENTi` action (i = 1, 2, 3) when receiving a message from it. Each agent defines actions for port binding (`BINDCOMPO`), for starting a component (`STARTCOMPO`), for stopping a component (`STOPCOMPO`), etc. The PS is initialized with a buffer per VM and two topics for imports/exports (`ListBuffers`). The RM processes is composed

in parallel with the rest of system and synchronizes with the other processes on many actions. For example, the RM defines actions for VM creation and destruction (INSTANTIATEVMi (i = 1, 2, 3) and DESTROYVM, resp.). The RM guides also the application reconfiguration by adding and removing components from VMs (ADDCP and REMOVECP). When the RM detects a VM failure (FAILURE), it alerts the PS by an ALERTPS action.

All these actions are used for analyzing the protocol as we will see in the next subsection.

```
process MAIN [INSTANTIATEVM1:any, DESTROYVM:any, STARTCOMPO:any,
              ADDCP:any, REMOVECP:any, ..] is
   par INSTANTIATEVM1, ..., DESTROYVM, ADDCP, REMOVECP,
       FAILURE, ALERTPS in
       (* the reconfiguration manager *)
       RM [INSTANTIATEVM1, ..., DESTROYVM, ADDCP, REMOVECP,
           FAILURE, ALERTPS] (appli)
   ||
       par AGENTtoPS1, PStoAGENT3, FAILURE, ... in
           par
               (* first virtual machine, VM1 *)
               Agent[INSTANTIATEVM1, AGENTtoPS1, PStoAGENT1,
                     DESTROYVM, STARTCOMPO, BINDCOMPO, STOPCOMPO,
                     UNBINDCOMPO, ADDCP, REMOVECP, FAILURE] (vm1)
           ||
               (* second virtual machine, VM2 *)
               Agent[...] (vm2)
           ||
               (* third virtual machine, VM3 *)
               Agent[...] (vm3)
           end par
       ||
           (* publish-subscribe messaging system *)
           PS[AGENTtoPS1, ..., PStoAGENT3, ALERTPS] (?ListBuffers)
       end par
   end par
end process
```

3.2. Verification

We apply the LNT specification of the protocol to a set of 600 examples (application models and reconfiguration scenarios) in order to extensively

validate our protocol. From the specification and an example, CADP exploration tools generate an LTS that describes all the possible executions of the protocol for each example of application model and scenario. Moreover, we specified 40 properties in MCL [27], the temporal logic used in CADP. MCL is an extension of alternation-free μ-calculus with regular expressions, data-based constructs, and fairness operators. These properties helped us to chase bugs when finding them during the development phase. We use model checking to verify that they are respected during the protocol execution. There is an exception that when a VM failure appears, some properties are not respected (*e.g.*, a started component cannot be connected to a stopped one). The model checker automatically says whether these properties are not verified on the LTS. When a bug is detected by model checking tools, it is identified with a counterexample (a sequence of actions violating the property). We distinguish two kinds of properties: (i) those allowing to focus on the protocol behavior for verifying that the final objectives are executed (Prop. 1 below for instance) and guaranteeing that the architectural invariants for a reconfigurable application are always satisfied (Prop. 2), (ii) and those helping us to identify more precisely the source of error when one of the original properties was violated by verifying that the progress/ordering constraints are respected (Prop. 3, 4, 5, and 6). Following the scenarios that we want to check, these properties belong to different categories: properties dedicated to start-up scenarios (Prop. 1 and 2), destruction scenarios (Prop. 3), mixed scenarios (Prop. 4), and VM failure scenarios (Prop. 5, and 6). We present in this section some concrete properties verified on the application model showed in Figure 1:

1. All components are eventually started.

   (μX . (**< true > true and [not** `"STARTCOMPO !Apache1 !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 components on which it depends for mandatory imports.

```
[
 ( "nil" or
   true* . "STOPCOMPO !Apache1 !VM1" .
    (not "STARTCOMPO !Apache1 !VM1")* ."STOPCOMPO !Tomcat !VM2"
 ) . (not "STARTCOMPO !Tomcat !VM2")* ."STARTCOMPO !Apache1 !VM1"
] false
```

In the example of application showed in Figure 1, the Apache1 component is connected to the Tomcat component on a mandatory import. Therefore, we must never find a sequence where Apache1 is started before Tomcat, or a sequence where Apache1 is stopped, the Tomcat server is then stopped, and Apache1 is started again 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. All components hosted on a VM eventually stop after that VM receives a destruction request from the DM.

```
[ true* . { DESTROYVM ?vm:String } ]
          inev ( { STOPCOMPO ?cid:String !vm } )
```

This property uses the macro `inev (M)`, which indicates that a transition labeled with `M` eventually occurs. This macro is defined as follows:

```
macro inev (M) = mu x .( < true > true and [ not M ] X )
end macro
```

This property does not depend on the application. It can be verified for any application without knowing the name of VMs and components. Parameters can be related in MCL by using variables in action parameters (*e.g.*, `vm` for the virtual machine identifier and `cid` for the component identifier). This property shows the data-based features that are available in MCL.

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

   [**true*** .
   `"BINDCOMPO !Apache1 !Workers1"` .
   (not `"UNBINDCOMPO !Apache1 !VM1"`)* .
   `"BINDCOMPO !Apache1 !Workers1"`
   ] **false**

   When a component is connected to another component through an import (mandatory or optional), it cannot be bound again except if it is unbound before.

5. There is no sequence with two VM instantiations without a failure or a destroy in between.

   [**true*** .
   `"INSTANTIATEVM1"` .
   (not `"FAILURE !VM1 or DESTROY !VM1"`)* .
   `"INSTANTIATEVM1"`
   ] **false**

   When a VM is instantiated, it cannot be instantiated again except if this VM is destroyed or failed.

6. A failure action in the VM is eventually followed by an alert of this failure

   [**true*** . { `FAILURE ?vm:String` }] **inev ({ `ALERTPS !vm` })**

   This property does not depend on the application. The variable `vm` (the virtual machine identifier) used as first parameter of `FAILURE` must be the first parameter in `ALERTPS`. This property is verified for all applications and for all VM names and also uses the macro `inev`.

3.3. Experiments

We conducted our experiments on more than 600 hand-crafted examples on a Pentium 4 (2.5GHz, 8GB RAM) running Linux. Each example consists of an application model and a specific scenario (a sequence of instantiate/destroy VM operations and add/remove components to/from VMs). From this input, the CADP exploration tools generate the corresponding LTS by enumerating all the possible executions of the system. Finally, the CADP model checker is called, providing as result a set of diagnostics (true or false). The model checker returns true if a property is verified. When a property is not satisfied, it returns false as well as a counterexample. We present in this section experiments that summarize some of the numbers obtained when varying the number of VMs, the number of reconfiguration operations, and the number of failures.

We present in Figure 11 (left) the size of the LTS (transitions) as well as the time to execute the whole process (LTS generation and properties checking) when we modify only the number of VMs (no VM destruction or failure). The number of reconfiguration operations is the same as the VM number (we only instantiate VMs). Increasing the VM number leads to an increase of the number of components and ports. Then, the more VMs and ports, the more parallelism in the system and therefore the more messages exchanged among VMs. Figure 11 (left) shows how the LTS size grows exponentially when we slightly increase the number of VMs in the application. The computation time scales from a few minutes for applications with 1 VM and few ports to a few hours for an application with 4 or more VMs.

Figure 11 (right) summarizes the results obtained for the same application used to obtain results showed in Figure 11 (left) but with a destruction and re-instantiation operations at the end of the scenario. The LTSs size and analysis time increase in a remarkable way even when just adding one destruction operation to the same application. This operation triggers a double propagation, hence more exchanged messages between the PS and the agents.

Table 1 shows the size of the LTS (states and transitions) before and after minimization (*wrt.* a strong bisimulation relation) as well as the time to execute the whole process in the last column of the tables (LTS generation and minimization on the one hand, and properties checking on the other). These results are obtained for the application described in Figure 1, with an

increasing number of failures (F). The more VMs fails, the more the LTS size and analysis time increase.

Fortunately, our goal here was not to analyze huge systems (with potentially many VMs) during the verification of the protocol, but to find bugs in the protocol. Indeed, most issues were found on small applications describing pathological reconfiguration cases.

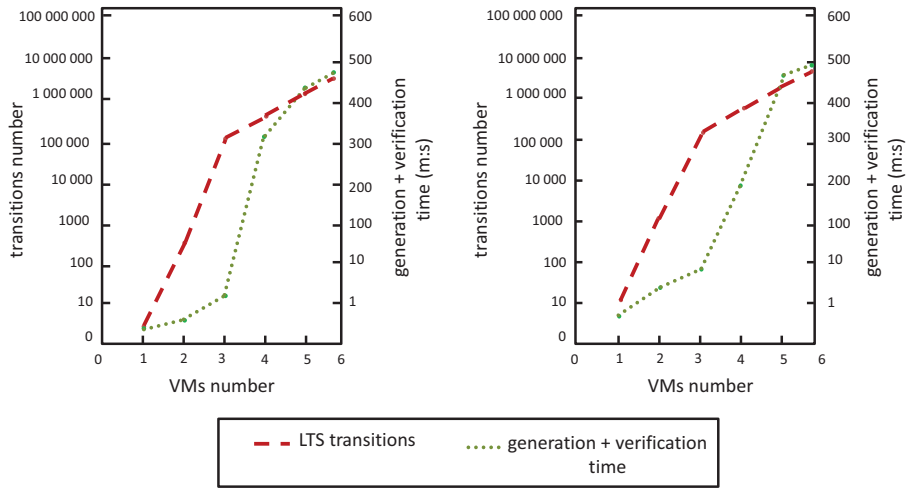

Figure 11: Experimental results when increasing the number of VMs (left) start-up scenario without failure, (right) start-up and destruction scenario without failure

3.4. Problems Found

We have presented in this section the specification and verification of our protocol. Model checking tools permitted to find bugs that were identified by counterexample analysis. This allowed us to revise several parts and correct specific issues (*e.g.*, adding some acknowledgement messages after binding/unbinding ports, starting/stopping components, etc.) in both the

F	LTS (states/transitions)		Time (m:s)
	raw	minimized	LTS gen. / Verif.
0	452,378/983,963	8,531/29,394	9:21 / 1:84
1	612,293/1,262,732	9,568/31,472	10:37 / 1:79
2	682,459/1,420,543	12,390/38,971	16:59 / 2:83
3	793,813/1,584,238	16,673/48,562	24:73 / 3:69
4	993,527/1,763,227	19,586/63,254	31:91 / 4:56

Table 1: Experimental results (with failures)

specification and implementation that were written at the same time. During the verification steps, we detected two important issues that we corrected in the latest version of the protocol. The implementation was systematically corrected. In the rest of this section we will focus on these two problems.

In the first version of the protocol, the RM was centralized and in charge of creating VMs, starting and stopping components. Therefore it kept track of the states of components for each VM. To do this, the RM was informed every time there was a change in the application. It communicated with all the agents by exchanging messages in order to update the application after each change in the component architecture (*e.g.*, a component binds to another component, a component changes its state). There was an overhead of messages transmitted to and from the RM. We noticed during our experiments that even with simple applications, CADP generated huge LTSs. We solved this drawback by proposing a decentralized version of the protocol. The new version of the protocol consists of an RM that is not in charge of starting or stopping components. The RM guides only the application reconfiguration by instantiating, destroying, and repairing VMs, or by adding and removing components from an existing VM. It is part of the agent behavior to drive the component start-up/shutdown. The decentralized version of the protocol avoids additional, unnecessary messages exchanged between agents and the RM. This version guarantees more parallelism and better performance in the corresponding implementation of the protocol.

The second issue that we detected during the verification of the protocol is in the way VMs are destroyed. Originally, when a component required to stop, it was stopped and then all components bound to it were stopped. Stopping components in this order induced started components connected to stopped components. This violated the consistency of the component com-

position and well-formedness architectural invariants. This bug was detected thanks to a property stating that *" a component cannot be started and connected through an import (mandatory or optional) to a stopped component "*. Thus, we corrected this bug by proposing another way to stop components. When a component needs to stop, it requests to all components bound to it to unbind and once it is done, it can immediately stop. This implies first a backward propagation along components bound on mandatory imports. Once this first propagation stops (a component does not provide service or is connected only through optional imports), 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.

3.5. Discussion

In the final part of this section, we would like to comment on alternative verification techniques we could have used to complement our model checking based approach.

In order to formally analyse the reconfiguration protocol, we specified over 600 application models and reconfiguration scenarios in order to extensively validate our protocol. When achieving these tasks, we paid a lot of attention to cover very different applications and scenarios, in particular pathological and corner cases. Hence, this large investment in validation was helpful in order to detect very early in the development process subtle bugs and issues (see subsection 3.4 for more details). This also makes us highly confident in the correctness of the protocol. However, a limit of our model checking based analysis is that it works on concrete configurations and scenarios. A way to generalise these verification results would be to reason on *generic* applications and scenarios using for instance parameterised systems verification techniques [33]. The application of these techniques to our work is not straightforward and several difficulties have to be faced and solved. As an example, when we change a reconfiguration scenario by, *e.g.*, removing a reconfiguration operation or by changing the order of the reconfiguration operations, there is no formal relation connecting the LTS generated for the former scenario and the new LTS. Moreover, given an application model, we can infer an infinite number of possible reconfiguration scenarios. Nonetheless, this is a perspective of interest and we plan to study this idea in more

details in future work.

Another complementary approach is to prove the correctness of the protocol using theorem proving techniques as achieved with Coq in [7] for ensuring the correctness of the reconfiguration of component assemblies. This is a very interesting and ambitious objective, but this would complement the validation of the reconfiguration protocol using model checking techniques presented in this paper, by ensuring that the protocol would work for any application and any reconfiguration scenario.

4. Related Work

In [13, 19], the authors propose languages and configuration protocols for distributed applications in the cloud. [13] 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). Contrary to us, in [13], the deployment does not work in a decentralized fashion, and this harms the scalability of applications that can be deployed.

A recent related work [17] 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 [17] does not explain how they preserve composition consistency and architectural invariants when stopping resource drivers. The authors write that going from an active state to an inactive state has as precondition that dependencies are inactive without explaining the mechanism used to deactivate them.

There exist many approaches which focus on specifying and verifying distributed systems and component-based systems. In [22, 25, 26, 5, 32, 10, 29, 23], the authors proposed various formal models (Darwin, Wright, etc.) in order to specify dynamic reconfiguration of component-based systems whose architectures can evolve (addition/removal of components and connections) at run-time. These techniques are adequate for formally designing dynamic applications. In [22, 26] for instance, the authors show how to formally analyze behavioral models of components using the Labeled Transition System Analyzer. 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 [14, 15, 31], the authors describe 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 [8], 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 assumes that they are located on a same VM. In [16], the authors present a self-deployment protocol able to automatically deploy cloud applications in a decentralized fashion. This protocol supports VM failures by detecting and repairing them, but it does not support reconfiguration features.

In [24], the authors present a design and an implementation of a technique for the automatic synthesis of deployment plans. It provides a reconfiguration algorithm to deploy heterogeneous software components and compute the sequence of actions allowing the deployment of a desired configuration. The algorithm works even in the presence of circular dependencies among components. This work presents some similarities with ours, but [24] does not provide methods for verifying its techniques and algorithms.

In [11], the authors present the Aeolus component model that aims at mastering the complexity of cloud applications. It automates as much as possible the management of such applications. It uses an initial configuration, an universe of available components and a target component. A component is a grey-box presenting internal states and mechanisms to change its state during the deployment and reconfiguration process. In [11] the initial configuration, target components and actions are known from the beginning but the algorithm/protocol does not support modifications that can take place at run-time.

This article is an extended version of a conference paper published in [4]. The key additions of this journal version are as follows: *(i)* introduction of new reconfiguration actions (addition and removal of components), *(ii)* extension of the protocol in order to detect and repair VM failures, *(iii)* a detailed presentation of the properties to verify the protocol, especially properties which verify the failure detection and repair, *(iv)* more experiments especially with multiple VM failures, and *(v)* presentation of an updated review

of related work.

5. Concluding Remarks

In this article, we have presented a protocol that aims to dynamically reconfigure distributed cloud applications and support VM failures. It enables one to instantiate new VMs/destroy existing VMs and to add new components/remove existing components. 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. This protocol does not only detect VM failures but also repairs failures by creating a new instance for each failed VM and by warning the other VMs of this failure. The protocol is robust and fault-tolerant. It succeeds in stopping/starting components hosted on the different virtual machines even in case of multiple failures. The protocol was formally specified and verified using the LNT specification language and the CADP toolbox, which turned out to be very convenient for modeling and analyzing such protocols, see [30] for a discussion about this subject. Model checking techniques were used to verify about 40 properties of interest on a large number of application models and reconfiguration scenarios. More importantly, during this verification stage, we improved several parts of the protocol and found several bugs. In particular, we deeply revise the part of the protocol dedicated to the VM destruction and component shutdown. All these issues have been corrected in the corresponding Java implementation.

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. The authors would like to thank Radu Mateescu for his help during the formulation of MCL properties.

References

[1] AMQP.

[2] Puppet Labs Documentation.

[3] RabbitMQ.

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

[5] 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.

[6] M. Bertier, O. Marin, and P. Sens. Implementation and Performance Evaluation of an Adaptable Failure Detector. In Proc. of DSN'02, pages 354–363, Washington, DC, USA, 2002. IEEE Computer Society.

[7] F. Boyer, O. Gruber, and D. Pous. Robust Reconfigurations of Component Assemblies. In Proc. of ICSE'13.

[8] 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.

[9] A. Brogi, J. Carrasco, J. Cubo, F. D'Andria, A. Ibrahim, E. Pimentel, and J. Soldani. SeaClouds: Seamless Adaptive Multi-Cloud Management of Service-Based Applications. In Proc. of CIbSE' 14, pages 95–108. American Conference on software Engineering Steering Committee, 2014.

[10] 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.

[11] M. Catan, R. Di Cosmo, A. Eiche, T. A Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Aeolus: Mastering the Complexity of Cloud Application Deployment. In Proc. of ESOCC'13, volume 8135 of LNCS, pages 1–3. Springer, 2013.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

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

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

[23] C. Krause, Z. Maraikar, A. Lazovik, and F. Arbab. Modeling Dynamic Reconfigurations in Reo Using High-Level Replacement Systems. Sci. Comput. Program., 76(1):23–36, 2011.

[24] T. A. Lascu, J. Mauro, and G. Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In Proc. of FACS'13, volume 8348 of LNCS, pages 254–272, Nanchang, China, 2013.

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

[26] 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.

[27] 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.

[28] N. De Palma, S. Jean, S. Ben Atallah, and D. Hagimont. J2EE Applications Deployment: A First Experiment. In Proc. of PDPTA'04, pages 1440–1446, USA, 2004. CSREA Press.

[29] G. Salaün. Generation of Service Wrapper Protocols from Choreography Specifications. In Proc. of SEFM'08, pages 313–322. IEEE Computer Society, 2008.

[30] 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.

[31] 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.

[32] 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.

[33] L. D. Zuck and A. Pnueli. Model Checking and Abstraction to the Aid of Parameterized Systems (a Survey). Computer Languages, Systems & Structures, 30(3-4):139–169, 2004.