Flexible deployment of component-based distributed applications on the Cloud and beyond

Linh Manh Pham[1*] and Truong-Thang Nguyen[2]
[1] Center of Multidisciplinary Integrated Technologies for Field Monitoring, University of Engineering and Technology – Vietnam National University, Hanoi, Vietnam
[e-mail: linhmp@vnu.edu.vn]
[2] Institute of Information Technology - Vietnam Academy of Science and Technology
[e-mail: ntthang@ioit.ac.vn]
*Corresponding author: Linh Manh Pham

Received July 14, 2018; revised September 6, 2018; accepted September 21, 2018; published March 31 2019

Abstract

In an effort to minimize operational expenses and supply users with more scalable services, distributed applications are actually going towards the Cloud. These applications, sent out over multiple environments and machines, are composed by inter-connecting independently developed services and components. The implementation of such programs on the Cloud is difficult and generally carried out either by hand or perhaps by composing personalized scripts. This is extremely error prone plus it has been found that misconfiguration may be the root of huge mistakes. We introduce AutoBot, a flexible platform for modeling, installing and (re)configuring complex distributed cloud-based applications which evolve dynamically in time. AutoBot includes three modules: A simple and new model describing the configuration properties and interdependencies of components; a dynamic protocol for the deployment and configuration ensuring appropriate resolution of these interdependencies; a runtime system that guarantee the proper configuration of the program on many virtual machines and, if necessary, the reconfiguration of the deployed system. This reduces the manual application deployment process that is monotonous and prone to errors. Some validation experiments were conducted on AutoBot in order to ensure that the proposed system works as expected. We also discuss the opportunity of reusing the platform in the transition of applications from Cloud to Fog computing.

Keywords: Application deployment, dynamic reconfiguration, distributed application, cloud computing, fog computing

A preliminary version of this paper appeared in IEEE SoICT 2016, December 08-09, Ho Chi Minh City, Vietnam. This version extends the D&C protocol significantly to cover deployment and dynamic reconfiguration for distributed applications not only in Cloud computing but also beyond the Cloud such as Fog computing. Current related work is also updated to cover the state-of-the-art. New experiments were conducted to validate the flexibility and portability of our approach in both Cloud and Fog environment. This work is supported by FIRST Central Project Management Unit, Ministry of Science and Technology, Vietnam under Grant Agreement number 12/FIRST/2a/IoIT and Vietnam National University, Hanoi (VNU) (Project No. VNU QMT.17.03). This research has also been financially supported in part by Grants VAST01.06/15-16 and PTNTĐ17.01 from Vietnam Academy of Science and Technology, the Vietnam government.

http://doi.org/10.3837/tiis.2019.03.002 **ISSN : 1976-7277**

1. Introduction

1.1 Motivation

Configuration errors account for a significant proportion of the causes of problems reported by users in both open source and commercial systems [1]. In addition, these errors are classified as highly serious and therefore require really quick reactions. Dynamic (re)configuration of distributed components is not an ordinary problem [2], and the growth of cloud computing in the recent years as well as its evolution such as Fog computing have created the issue more obvious. Cloud computing has resulted in the advancement of new highly added value services which aim at enhancing the time and cost effectiveness of enterprises' IT departments. At its heart, virtualization has allowed service providers to use easily and transparently commodity hardware in an effort to offer scalable services with a reduced price. By merging loosely - coupled components, new applications can be deployed and updated if needed using the Cloud or Fog. Unluckily, cloud providers just supply a couple of practical mechanisms that only permit service providers to access the resources of theirs. In addition, tying to vendor-specific solutions might help make your life easier temporarily, but it can have a negative impact on the road in case the provider side suffers from disasters including a major outage [3]. Few vendor-neutral tools offer end-to-end features, from the application modeling to the addition of additional components on the fly. However, these tools often aim to provide holistic application management solutions requiring expert knowledge as well as consuming study time.

Put simply, what is certainly lacking is really a simple, cloud-portable and flexible management design allowing service providers to automatically set up and configure their applications as well as allow future modifications on the application while always guaranteeing that these modifications do not interfere with the current configuration. We are first working on minimizing configuration mistakes that users make when implementing complex legacy systems in this article. Secondly, we wish to deliver consistent configurations during consecutive updates. The issue in the context of Cloud computing in particular can be consulted in [4].

If done manually, the activation and configuration are extremely complex and error-prone due to the variety of software stacks bundled in the VMs. This becomes even worse in Fog environment where software components are distributed across its three strata: Edge, Fog and Cloud [5]. Moreover, the models and engines of current related work proposed to design and deploy distributed applications are not quite natural and very complicated, which prohibit extension of these approaches beyond Cloud. Inserting and deleting software components and/or full VMs flexibly and rapidly require us to rethink the way we develop dynamic deployment and configuration (D&C) platforms resulting in a lightweight model to design and deploy complex distributed applications.

1.2 Our Contributions

To deal with the mentioned D&C issue, we introduce AutoBot platform consisting of three main modules. The first module includes a model that describes hierarchically any distributed stacks of applications and interdependencies of components. The second one is a dynamic D&C engine which installs and configures automatically all the earlier described component instances on cloud environment. Finally, a runtime system ensuring the proper global

configuration of the running application when new VMs and/or component instances are added and removed.

Such a dynamic behavior requires a flexible platform catching the necessary configuration aspects to provide a fully automated environment for deploying any distributed applications in the Cloud. AutoBot boldly promotes the features following:

Flexibility: its model automates the deployment of all distributed applications regardless of their programming language and model, runtime environment, and business domain. It offers (i) a consistent abstraction of the application's software architecture and (ii) a consistent configuration interface staying away from specific configuration scripts managed by hand. Each component of this model includes a part of the management functionality to be deployed by the distributed software. This model is also entirely independent of any Infrastructure-as-a-Services (IaaS), and the AutoBot runtime is actually developed as independently of the infrastructures as possible. In case of incorporating another cloud infrastructure, not many changes in the AutoBot runtime are actually required because it just relies on 2 IaaS primitives: create and delete VMs, and a little information relating to their status.

Portability: A couple of plug-in types are defined and developed in AutoBot, which allow developers to customize and extend the platform as they wish to be independent of the application model, runtime system, and hardware infrastructure. The *DSL (Domain Specific Language) plugin* enables the developers to select or add languages describing the application model such as the built-in AutoBot DSL, TOSCA [6], or CAMEL [7]. The *Runtime plugin* (i.e. connectors) is for the development and integration of any scripting languages (e.g. Bash, Python) or configuration management tools (Chef [8], Puppet [9]). Finally, the developers can integrate new infrastructure from different cloud providers using the *IaaS plugin*. This portable mechanism helps future developers to extend the platform beyond Cloud computing and cover the Fog computing's concepts, for instance.

Scalability: The proposed configuration protocol is actually decentralized to be able to cope with many customer applications. As soon as the VMs are actually instantiated, the configuration protocol is enabled and the entire application is configured without any centralized servers (as opposed to methods such as Puppet configuration server). Each VM incorporates the necessary understanding of the application model along with a configuration agent which manages the configuration software components to operate in the VM.

Loose coupling: The dynamic configuration protocol was made with asynchrony in mind to implement the solution's agility. Configuration agents talk in a reliable and asynchronous manner using a message queuing system,. Hence, no globalwide synchronization is needed between the configuration agents to post configure and activate the distributed application. This process evolves progressively every time a VM is readily available and ultimately the distributed application is properly configured, installed and running.

The rest of this article is organized as follows. In Section 2, a common use case of distributed application is described that we use throughout this article as an example. Section 3 shortly introduces the main modules of the proposed platform. Section 4 presents, through the use case in Section 2, a designed model for the distributed application, which consists of the meta data, the variables (especially exported/imported structures) along with the interdependencies between components/software types. Both Sections 5 and 6 discuss our dynamic D&C protocol. Some performance evaluations conducted on Amazon EC2 are reported in Section 7. At last, related work is presented in Section 8 and conclusion is made in Section 9.

2. Use case

Nowadays many enterprise applications are complex multi-tier ones. It means that the application is composed of many parts, and these parts can be located on a same computer, or can be distributed among multiple computers connected through a network. A part is basically a software installation, providing or using services of another part, through interface. The advantage of this design is that when the parts are communicating through interfaces, implementation can be changed. The second advantage is that under high load (either an increase of users or user requests, or an increase of the work to be done), some parts can be replicated in order to distribute the workload. It is one of the most basic ways to scale a multi-tier application. One of the most known and used multi-tier applications is the three-tier one: a frontend server, a backend server, and a storage server. A well-known implementation of this pattern is Apache-Tomcat-MySQL. That is the example we will use in the next sections of this article.

3. AutoBot architecture

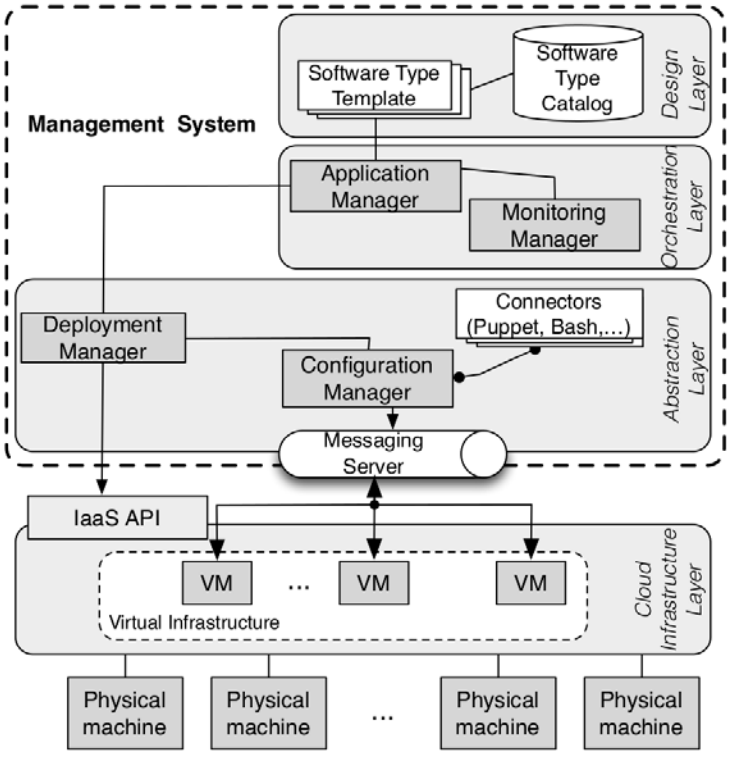

Fig. 1. Overall system architecture and interactions

A unit of software is called a *software type* in AutoBot. A number of these *software types* will compose a distributed application. Running version of these types, *i.e. software instances*, are instantiated at runtime. In **Fig. 1** about AutoBot global architecture, the roles and relative interactions between these abstractions are explain briefly. During the article, operations of AutoBot is demonstrated through RUBiS [10], a typical multi-tier web application. RUBiS, similar to ebay, is an open source auction site which can be implemented with three tiers

including Web-Apache, Application-Tomcat and Database-MySQL. AutoBot modules and abstractions are detailed as follows:

Software Types. As mentioned it is an object-oriented (O-O) abstraction enabling users to describe meta-data needed for installing, configuring or upgrading a component.

Software Instances. Also deriving from O-O model, it is an occurrence of a particular instantiated software type. In a VM, identification of instance is unique, but software instances of the same software type may co-locate on the same instance of VM (i.e. software consolidation).

Deployment Manager (DM). It is a module coordinating the entire deployment process of the application on multiple VMs and keeping a global view of the current model of managed applications. By a dedicated *admin* channel resided in the messaging server, DM controls the instantiation of VMs using IaaS API, installs and configures the software instances and types inside the VMs, and maintain essential information of other components in the system. Therefore, DM is important not only for the initial application deployment but also for the dynamic application updates (insert/delete components/VMs).

Configuration Manager (CM). CM is responsible for publishing exported and imported variables (i.e. the inter-dependencies) which are needed to complete the installation of different software instances. Exported variables and imported variables of an instance is the ones that other components require and the ones the instance needs to function correctly. This module also handles configuring the software component(s) thanks to the configuration connectors like Puppet and Bash scripts.

Application Manager. It is a module that helps manage the life cycle of the managed application and its components, regularly check their current status (started, stopped, error, etc.). By the application manager the running application can be updated by inserting/removing software instances.

Monitoring Manager. It consists of 3 sub-modules: *(1). Auto-monitor:* this module uses heartbeats to monitor status of VMs and informs the user when VMs come down. No particular action is conducted that allows service providers freely implement their own failure notification solutions; *(2). Auto-protection:* this module allows service providers apply their own firewall rules to make interconnections possible between software instances deployed on different VMs; *(3). Auto-repair:* this module reacts to changes captured by the monitor such as removing the crashed instances running on the VM, cleaning the application, killing the VM if needed or asking for other VMs with similar characteristics and redeploy the instances. This module is by default not activated and turned on by users if required. These submodules enable distributed applications to react to changes of surrounding environment, which helps latency-sensitive IoT (i.e. Internet-of-Things) applications operate effectively in Fog-computing environment.

It is worth noting that AutoBot uses methods to install and configure software independent of the technologies dedicated to installation of software components, e.g. Puppet or Bash scripts. Moreover, a configuration protocol based on exchanging of asynchronous message enables starting of components in correct order as long as the local constraints of these components are resolved. The combination of our proposed model and the asynchronous message exchange help the system track the status and relationships of the application at runtime. Next sections will discuss AutoBot model in details through RUBiS examples and see how AutoBot handle its dynamic D&C protocol.

4. Application model of AutoBot

To facilitate the description of application model, AutoBot DSL using ML-style notation have been developed enabling users to define various software types. Under AutoBot language, an application (lines 6 – 14, **Fig. 2**) contains definitions of software types/instances and VM types/instances. While VM type is characterized by specifying the identification of the IaaS image (line 73, **Fig. 2**), VM instance is referred to its VM type along with an IP and an IaaS flavor such as “small” or “micro” (lines 65 – 71, **Fig. 2**).

A software type (lines 57 – 63, **Fig. 2**) consists of parameters assigned to a value in some cases. These parameters are local configuration of the software and exported/imported structures (i.e. variables). The imported variable has a special parameter indicating if receiving this variable is obligatory for booting. In these structures the parameter “channel” defining the topic on the message server where the corresponding software instance publishes or subscribes. Using this parameter prohibit software instance from broadcasting to all other instances requiring or sending this type of structure. A parameter indicating the configuration technology is also needed to be point out. Some scripts defining operations on how to install and configure the software on the VM must be provided along with the model. Some variables in these scripts are parameterized because they only can get practical value at runtime. A VM instance’s name can be specified in the software type, otherwise a VM type’s name must be included instead. All the parameters of a software type are inherited by its software instance (lines 46 – 55, **Fig. 2**). Furthermore, a software instance contains a reference to the VM instance on which it is running, and some data about its present status.

VM type is a subtype of Container type which includes any hardware or virtual "box" hosting any of software types. AutoBot’s developers can add other subtypes of Container type as the IaaS plugins. Some subtypes can be physical machine, virtual container or even Fog nodes such as set-top boxes, access points. All the parameters of VM/Container type are inherited by VM instance in particular or Container instance in general. This reduces difficulties of dealing with the heterogeneity of both the Cloud and Fog environments.

5. Dynamic Deployment and Configuration protocol

The AutoBot dynamic D&C protocol is described in this section through the installation and configuration of RUBiS. It is known that RUBiS can be run as a Java servlet embedded in a servlet container such as JOnAS application server [11] or Apache Tomcat.

5.1 The system overview

Four parts are implemented in the system to deploy an application like RUBiS: (i) a web application hosting the DM receiving calls from the clients to handle instantiation and removal of software instances, VMs and to keep an eye on the managed application(s); (ii) an IaaS provisioning or releasing VMs when the DM asks for; (iii) a *configuration agent* located in each VM, which is activated after the VM has booted to handle communication on behalf of the VM; (iv) a message server to exchange messages among the agents themselves and between the agents and DM.

5.2 The dynamic Deployment and Configuration protocol

Fig. 3 sums up various steps required to deploy and configure RUBiS using AutoBot on the Cloud:

```
datatype applicationState = RUNNING| STOPPED |PROBLEM ;
datatype configuratorType = PUPPET| BASH |NOTHING ;
datatype softwareInstanceState = RUNNING | UPDATING | INSTALLING | VM_HAS_BEEN_TERMINATED | ↩
      INTO_THE_PIPE | STOPPED | PROBLEM ;
datatype vmState = INSTANTIATING | RUNNING | PROBLEM | TERMINATED ;

type application = {
  name : string,
  softwareTypeList : (software) list option,
  softwareInstanceList : (softwareInstance) list option,
  vmTypeList : (vm) list option,
  vmInstanceList : (vmInstance) list option,
  state : applicationState option,
  autoProtection : autoProtection option,
  autoRepair : bool option };

type autoProtection = {
  firewallExceptionList : (firewallException) list,
  defaultRules: (ruleType) list };

type entry = {key : string, value : string};

type exportedStruct = {
  name : string,
  channel : string option,
  varRef : (string) list };

type firewallException = {
  softwareTypeName : string,
  port : int };

type importedStruct = {
  name : string,
  channel : string option,
  varName : (string) list,
  requiredToBoot : bool,
  importList : (import) list option };

type import = {
  softwareInstanceNameExportingVars : string,
  importedVars : (entry) list };

type myMapEntry = {name : string, value : string };

type ruleType = {rule : string};

type softwareInstance = {
  name : string,
  softwareTypeName : string,
  varList : (myMapEntry) list option,
  exportedStructList : (exportedStruct) list option,
  importedStructList : (importedStruct) list option,
  configurator : configuratorType,
  vmTypeName : string option,
  vmInstanceName : string,
  state: softwareInstanceState option };

type software = {
  name : string,
  varList : (myMapEntry) list option,
  exportedStructList: (exportedStruct) list option,
  importedStructList: (importedStruct) list option,
  configurator : configuratorType,
  vmTypeName : string option };

type vmInstance = {
  name : string,
  vmTypeName : string,
  instanceId : string,
  capacity : string,
  ip : string,
  state: vmState };

type vm = {name : string, idIaas : string};
```

Fig. 2. AutoBot type model

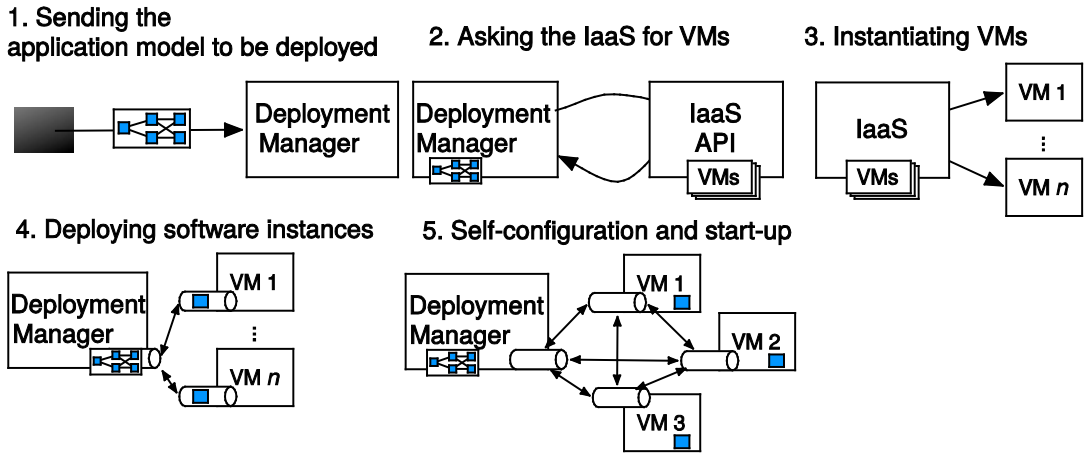

Fig. 3. Steps of the dynamic D&C protocol

1. The model of the distributed application to be deployed is sent to the DM. **Fig. 4** and **Fig. 5** represents the RUBiS application model under AutoBot DSL language. **Fig. 4** represents the various components forming RUBiS. As shown in the lines 17 – 30 and lines 33 – 44, **Fig. 4**, two similar types are provided (Tomcat and JOnAS, respectively). The RUBiS servlet can be hosted by both these type of servers and integrated with Apache thanks to the Apache JServ Protocol connector (AJP). The DM receives this model and converts it into a Cloud-Independent Constraint Problem (CICP) model which the description of software components is completely independent of the cloud service providers. Afterward, the DM combines information from the CICP model and the supported cloud providers (described in the *IaaS plugins*) to generate a Cloud-Specific Constraint Problem (CSCP) model. The CSCP model is a proposed solution for optimal deployment of the distributed application on existing cloud resources.
2. Based on the CSCP model, the DM prepares primitive requests which are sent to the corresponding IaaS API for the instantiation of required VMs. With RUBiS, first we deploy one instance of Apache (lines 4 – 15, **Fig. 5**), one instance of Tomcat (lines 18 – 29, **Fig. 5**) and one instance of MySQL (lines 32 – 41, **Fig. 5**). These instances are put on VM instances "VM1", "VM2", and "VM3" respectively.
3. The three VMs are instantiated by the IaaS. The present status of the VMs are kept track of by the DM. The DM waits for the VMs to be launched and keep running.
4. The DM puts configuration files and software instance definition on the message queue of each VM. When a VM is up, it first connects to the message server to retrieve the material of the instances for which it is responsible and begin installing these instances.
5. After being installed, all software instances of the application start to exchange their imported and exported configurations. In other words, they participate in the global application configuration thanks to the message server. Each time the instances get new data, they update themselves. The correctness of dynamic reconfiguration is ensured by this mechanism.

5.3 Start of the application

The deployed application is by default stopped when deployed. The user has to explicitly ask for his application to start. When a user asks for the startup, the DM checks the status of the dependencies of software instances. If all dependencies are resolved, i.e. it has received

"stopped" messages from all the components, it can send a "start" message to start the application. It asks for the start of software instances that have no mandatory dependencies to start (in RUBiS, the MySQL database can start up and be already up and running before Tomcat and/or Apache), or those whose dependencies are running. If some dependencies cannot be resolved, it outputs a message and waits for the configuration protocol to finish. It keeps on doing that while there are still software instances not running. In this process, when the DM wants a software instance to start, it sends a message to the agent on the VM through the message server. The agent then starts the software instance. This way to start application is a step towards a fully decentralized which would require each software instance not only to have the up-to-date model of the application locally but also to have the same consistent view as the other instances. Although several protocols exist, which provide such guarantees (View Synchrony), we argue that our "centralized" version is not an impediment since the DM "only" tracks the running model of the application and no other information is stored on the DM. The message server on the other hand may represent a bottleneck in case of a growing number of application and thus software instances. That is why in our system, (i) we allow the message server to be on a separate machine and (ii) we can have one message server per DM as well as shared message servers between multiple DMs in case many concurrent large applications are running.

Varying architectures using channels. By default, each software instance exchanging variables communicates to other software instances exchanging this same structure. We can see an example of that behavior in **Fig. 6(a)**, where one Apache, two Tomcat, and two MySQL have been deployed. It may lead to broadcast of multiple variables which are unnecessary in some specific cases. To avoid this, we can specify dedicated communicating channels for both Tomcat and MySQL by changing the channel parameter. For example, we can define a channel named "channel1" for the imported structure of *Tomcat1* and for the exported structure of *MySQL1*. We do the same for *Tomcat2* and *MySQL2* with a channel named "channel2". We can see in **Fig. 6(b)**, this results in a change of the application architecture: each Tomcat is now linked to only one MySQL. This feature is practical when service providers want to use dedicated VMs to support premium services, for instance.

6. Dynamic Reconfiguration

In the previous section, we presented a basic overview of the system and D&C protocol. We now detail the deployment process of a new software instance to become a running application. To present the deployment and configuration steps of new software instance, we illustrate it with our RUBiS. We start with an Apache, a Tomcat and a MySQL, as previously mentioned, each on a different VM. In **Fig. 5** the model transmitted by the user to the DM, which represents this occurrence of the application, is described.

6.1 Dynamic insertion of a new software instance

We assume that the application described in the previous section (Apache, Tomcat, and MySQL) is deployed, configured and started. We now present the steps to add a new JOnAS application server to this application. For example, the new software instance JOnAS is deployed by transmitting the software instance model depicted in **Fig. 7**.

```
val rubis: application = { name="RUBiS",

  softwareTypeList= SOME
   (* Apache *)
    [{name="Apache",
      varList= SOME [{name="ip",value=""},
                     {name="email",value="myMail@abc.com"}],
      exportedStructList= NONE,
      importedStructList= SOME [{name="workers",
                                 channel=NONE,
                                 varName=["ip","portAJP"],
                                 requiredToBoot=true,
                                 importList=NONE}],
      configurator=PUPPET, vmTypeName=NONE},

    (* Tomcat *)
     {name="Tomcat-With-Rubis",
      varList= SOME [{name="ip",value=""},
                     {name="portAJP",value="8009"}],
      exportedStructList= SOME [{name="workers",
                                 channel=NONE,
                                 varName=["ip","portAJP"],
                                 requiredToBoot=true,
                                 importList=NONE}],
      importedStructList= SOME [{name="database",
                                 channel=NONE,
                                 varName=["ip","port"],
                                 requiredToBoot=true,
                                 importList=NONE}],
      configurator=PUPPET, vmTypeName=NONE},

    (* Jonas *)
     {name="Jonas-With-Rubis",
      varList= SOME [{name="ip",value=""},
                     {name="portAJP",value="8009"}],
      exportedStructList= SOME [{name="workers",
                                 channel=NONE,
                                 varRef=["ip","portAJP"]}],
      importedStructList= SOME [{name="database",
                                 channel=NONE,
                                 varName=["ip","port"],
                                 requiredToBoot=true,
                                 importList=NONE}] ,
      configurator=BASH, vmTypeName=NONE},

    (* MySQL *)
     {name="MySQL",
      varList= SOME [{name="ip",value=""},
                     {name="port",value="3306"}],
      exportedStructList= SOME [{name="database",
                                 channel=NONE,
                                 varRef=["ip","port"]}],
      importedStructList= NONE,
      configurator=PUPPET, vmTypeName=NONE}]
```

Fig. 4. RUBiS software types described in AutoBot model

```
 softwareInstanceList= SOME
  [
  (* Apache instance *)
    {name="apache1", softwareTypeName="Apache",
    varList = SOME [{name="ip", value=""},
                    {name="ip", value=""}],
    exportedStructList= NONE,
    importedStructList= SOME [{name="workers",
                               channel=NONE,
                               varName=["ip","portAJP"],
                               requiredToBoot=true,
                               importList=NONE}],
    configurator=PUPPET,
    vmTypeName=NONE, vmInstanceName="VM1",
    state=NONE},

   (* Tomcat instance *)
    {name="rubis1", softwareTypeName="Tomcat-With-Rubis",
    varList = SOME [{name="ip", value=""},
                    {name="portAJP", value="8009"}],
    exportedStructList= SOME [{name="workers",
                               channel=NONE,
                               varName=["ip","portAJP"],
                               requiredToBoot=true,
                               importList=NONE}],
    importedStructList= NONE,
    configurator=PUPPET,
    vmTypeName=NONE, vmInstanceName="VM2",
    state=NONE},

  (* MySQL instance *)
    {name="db1", softwareTypeName="MySQL",
    varList = SOME [{name="port", value="3306"},
                    {name="ip", value=""}],
    exportedStructList= SOME [{name="database",
                               channel=NONE,
                               varRef=["ip","port"]}],
    importedStructList= NONE,
    configurator=PUPPET,
    vmTypeName=NONE, vmInstanceName="VM3",
    state=NONE}],

vmTypeList=NONE, vmInstanceList=NONE,
state=NONE,
autoProtection=SOME [
  {softwareTypeName="Apache", port="80"},
  {defaultRules= {
    rule="iptables -A INPUT -i lo -j ACCEPT",
    rule="iptables -A INPUT -m conntrack --ctstate ESTABLISHED,RELATED -j ACCEPT",
    rule="iptables -A INPUT -p icmp -j ACCEPT",
    rule="iptables -A INPUT -p tcp --dport ssh -j ACCEPT",
    rule="iptables -P FORWARD DROP",
    rule="iptables -P INPUT DROP"}}],
autoRepair=NONE };
```

Fig. 5. RUBiS software instances described in AutoBot model

6.1.1 Provisioning of a VM instance if needed

The VM instance defined in the software instance model is first checked by the DM when the deployment of a new software instance is requested by the user.

- If the request contains information to deploy the software application on an existing VM instance, the model is sent immediately to the VM by the DM (part described in the subsection 6.1.2).
- Otherwise, the DM checks whether a VM type (e.g. “t1.micro”, “m1.small”, etc.) has been defined in the software type of the software instance or not. If yes, one VM

instance of this kind will be asked the IaaS for instantiation. Otherwise, a VM instance will be instantiated from the default VM template.

In any cases, a message queue on the message server is created for each new VM.

6.1.2 Sending the software instance model to the VM instance

The DM serializes and sends the object and the configuration scripts of software instance to the message queue of the VM. In the case the VM is not running, this operation still can be done thanks to the asynchronous message server. When the VM is on and the agent is started, it connects to the message queue and gets all the materials. Note that the agent knows who to contact and on which message queue name through the user data specified by the VM when asking the IaaS for the instantiation of the VM.

6.1.3 Setup of the software instance on the VM and local configuration

When the software instance model and its configuration files are at the side of VM agent, it checks which "connector" used to perform the installation and configuration for the software instance. Four basic operations setup, update, start, and stop are implemented by a connector which is a Java class. The operations relative to each particular software type are described in its configuration files. A basic operation called by each connector will transmit the configuration of the model to the configuration and packaging system. This mechanism allows user to separate actual operations on VM from the configuration exchange. It means a wide range of software types can be covered: from distributed software to the ones available in Linux repositories.

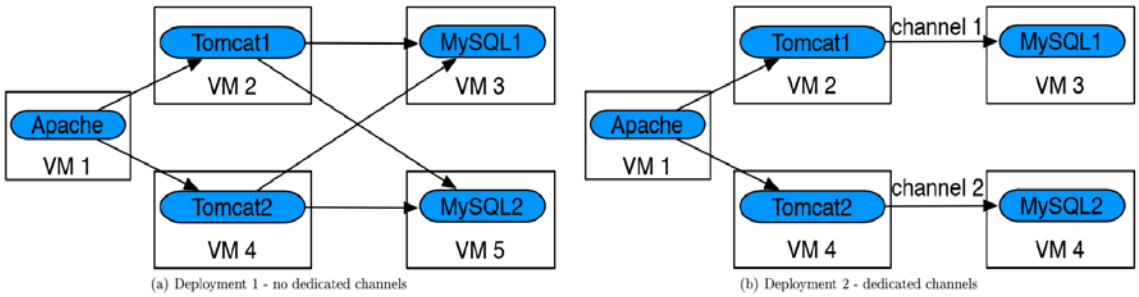

Fig. 6. Different strategies of communicating channels

```
val newJOnAS: softwareInstance = {
  name="jonas1",
  softwareTypeName = "Jonas-With-Rubis",
  varList = SOME [{name="portAJP", value="8009"},
            {name="IP", value=""}],
  exportedStructList= SOME [{name="workers",
              channel=NONE,
              varRef=["ip","portAJP"]}],
  importedStructList= SOME  [{name="database",
              channel=NONE,
              varName=["ip","port"],
              requiredToBoot=true,
              importList=NONE}],
  configurator=BASH,
  vmTypeName=NONE, vmInstanceName= "VM4",
  state=NONE };
```

Fig. 7. Adding a JOnAS application server at runtime

6.1.4 Distributed configuration

When the required software instances are all installed successfully on VMs, they now need to exchange configurations. After the installation, the agent on VM inspects the local model of the software instance, and applies the steps depicted in the following high-level Algorithm 6.1. The agent then listens for messages on the topics it subscribed to, and:

- If it receives a notification on a topic, it publishes the structure on the topic.
- If it receives a structure on a topic, it checks if it has instances that need that structure, and if so, it adds the structure to instances.

Algorithm 6.1 Distributed Configuration Exchange algorithm

```
1: for each software instance on the VM do
2:
3:     for each export of the instance do
4:         subscribe to the topic of the structure
5:         publish the structure on the topic of the structure
6:     end for each
7:
8:     for each import of the instance do
9:         subscribe to the topic of the structure
10:        publish a notification on the topic of the structure
11:    end for each
12:
13: end for each
```

This behavior enables the system to configure in any order, and also works if an instance is added after the initial deployment. We will illustrate the algorithm with the use case: adding a JOnAS server to an already deployed RUBiS application with Apache-Tomcat-MySQL:

- JOnAS exports a structure named “workers”. The agent subscribes to the channel named “import.workers”, and publishes the structure filled with values in the topic “export.workers”. The JOnAS had already subscribed to this “export.workers” topic, therefore it receives this new “workers” structure and can update itself with the IP address and the AJP port of the JOnAS server as well.
- JOnAS, in the same way, will receive the information regarding the database and update its configuration accordingly.

The entire dynamic configuration protocol of adding the JOnAS instance is depicted in **Fig. 8**.

6.1.5 Start of the software instance

In the same way as described in **Fig. 3**, after the DM puts the software instance model in the message server queue, the DM checks if the application is running or stopped. If the application is running, a message is sent to the same message queue asking the software instance to start. The agent on the VM, receiving messages in the same order, first installs and configures the software instance and then starts it.

6.2 Removal of a software instance

When the user wants to remove a software instance, it performs a “graceful leave”. The DM sends a message to the VM, through the message server, asking for the removal of the software instance. The agent receiving this message unsubscribes to all the topic(s) it has previously subscribed and notifies other software instances using its exported structure that it is leaving. For example, in the use case used in the previous part, the agent would unsubscribe to the topic

"import.workers" and "export.database", and it would then publish to the topic "export.workers" to notify it is leaving. The Apache would update itself to correct its configuration. The same kind of process is triggered when the user asks to shut down a complete VM. A removal can also occur if a VM crashes; in this case, if the *Auto-repair* submodule is enabled, it will take care of the correct removal of the VM according to the following procedure:

- It cleans the software instances handled by the dead VM from the application as these instances cannot notify other running instances that they left, this module does it on their behalf.
- It removes these software instances.
- It asks the IaaS to kill the VM with the problem. It does that to ensure that the VM will not reappear and mess up the application.
- It asks the IaaS to hire a new VM with the same characteristics as the crashed one.
- It deploys the software instances on the new VM.

7. Evaluation

Some results of deployment time of a running application on the Cloud and beyond performed in order to validate AutoBot. A small command line front-end has been developed to facilitate the basic management operations with AutoBot. RabbitMQ is chosen as the system's message server as well as Puppet and Bash are chosen to install software components locally. The verification of the proposed D&C protocol has been conducted in our original work [4].

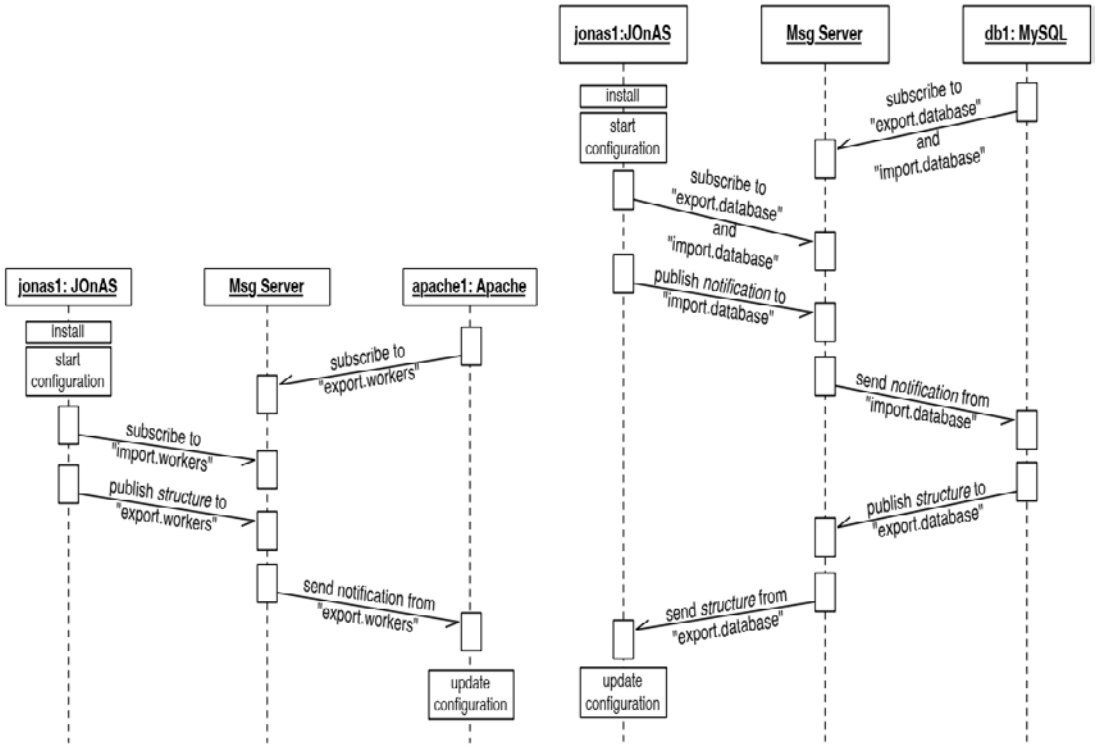

Fig. 8. Different strategies of communicating channels

We use AutoBot with Puppet connector to deploy the RUBiS application with three following components: 1) Apache Web server, 2) Tomcat included RUBiS servlet, and 3) MySQL database. The JOnAS application server with Bash script can be used in place of the original Tomcat application server in some experiments.

Settings. All the experiments use "t1.micro" instances from Amazon EC2. Without loss of generality, we combine the Message Server and the DM in one VM. Each software instance is deployed on a single VM instance. These entire RUBiS deployment system with AutoBot in the Cloud is shown in **Fig. 9**. First the original 3-tier application is deployed and then we switch to the following ones:

1. Insert a new instance of Tomcat type hosting the RUBiS servlet, called *tomcat2* as depicted in **Fig. 10**.
2. Insert an instance of JOnAS including the RUBiS servlet, too.
3. Remove the *tomcat2* instance.

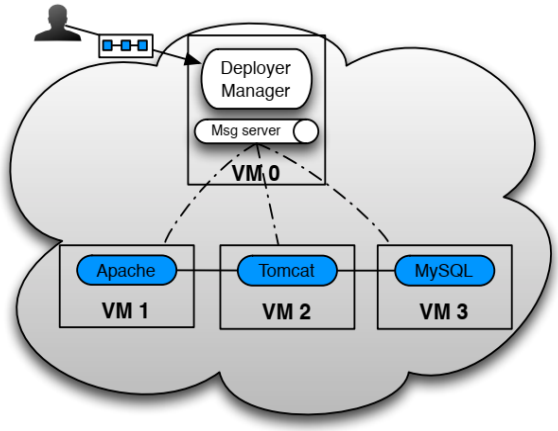

Fig. 9. RUBiS deployment with AutoBot

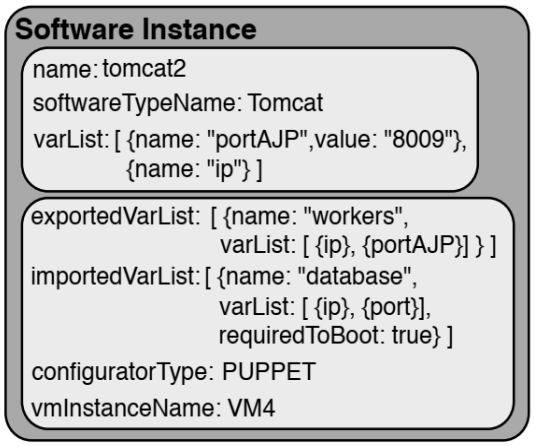

Fig. 10. Adding a new Tomcat software instance

The connections between the components are described step by step as follows in terms of variable exchange. The MySQL instance exports its port number and IP address. To serve as inputs of Apache instance for the AJP connector, the Tomcat instance exports its IP address, port number and the number of worker threads. It also imports the variables exported from the database instance. Like Tomcat application server, The JOnAS instance has the same set of imports/exports variables. Finally, the variables exported from Tomcat/JOnAS instances are imported by Apache web server. The web server then forwards its requests to the RUBiS servlet located in the application ones. In the original scenario, the components of RUBiS application are deployed on 3 VMs. Then two more VMs are inserted to host the newly deployed JOnAS instances. These two VMs will be removed afterward.

Time measurement. All measurements are the nmean and average results of ten different runs. Before deploying the application, we give some insights about the time it takes to deploy and install each software instance alone The deployment time of each component is shown in **Table 1**. The duration of an instance in the table represents the time for VM provisioning plus software deployment until the moment this instance starts. For example, we see that Apache takes 105 seconds which is the shortest time for deployment. In general, all the measurements are conducted between the time DM receives the request and the time it gets the up-to-date status of the instance being tracked.

Table 1. Mean deployment time of the standalone software instances

Mean deployment time (s)		
	Mean	**99th percentile**
MySQL	153.811 ± 17.6	180.044 ± 18.5
JOnAS	147.429 ± 5.3	154.214 ± 4.7
Tomcat	114.031 ± 26.5	165.440 ± 36.3
Apache	105.034 ± 8.6	121.130 ± 11.3

7.1 The 3-tier RUBiS deployment

In this experiment, the original RUBiS web application with Apache, Tomcat and MySQL instances is deployed and performed repeatedly. The average time and its standard deviations needed to have running instances of RUBiS are presented in **Fig. 11**. This duration is a combination of the VM provisioning time, the software installation time, and the dynamic configuration of these components. As mentioned, the asynchronous configuration mechanism of AutoBot enables the deployment of components in any order. Take advantage of this time-decoupling mechanism, the components are deployed and installed in parallel. We can see that a fully working instance of RUBiS hosted in modest EC2 nodes can be deployed within 152 seconds on average.

7.2 Addition and removal of software instances

To have component colocation, AutoBot allows software instances to be added or removed on the fly on an existing or new VM. With stateless applications that their components do not maintain the present status, this mechanism is especially effective. In this second experiment, starting from a running instance of the 3-tier RUBiS, we will add a new VM hosting an instance of Tomcat-RUBiS named *tomcat2*. It takes approximately 270 seconds to have a fully working 4-instance RUBiS from the moment the request to add the software instance is issued. This duration is a combination of the VM provisioning time, the software installation time inside this VM, the auto configuration of these components plus the update(s) impacting other components (Apache in this case). We next insert a JOnAS-RUBiS instance working with the Tomcat-RUBiS instance to serve the requests originating from Apache. The average deployment time of JOnAS is more than Tomcat (299 secs vs 270 secs) since JOnAS provides more special features. Lastly, *tomcat2* is removed and it only takes eight seconds on average to do this process including the dependency reconfiguration and bring the entire system back to a stable status. Details of the time it takes for each step of reconfiguration are shown in **Table 2**. These experiments validate our proposed platform and also show that legacy application can be easily deployed, managed and updated on modern clouds using the AutoBot model and runtime.

7.3 Beyond the Cloud

With a commercial web application like RUBiS, workload often peaks at some period daily. This results in slower response time causing negative impressiveness and feedback from customer side. In this situation, creating new application server instances like Tomcat or JOnAS to cover this demand is a critical task. To reduce end-to-end response time (RT), service providers can take advantage of devices at the edge where is closer to customers to install Tomcat/JOnAS instances inside. This context fits the Fog computing model. In this

specific experiment, we aim to install extra JOnAS instances inside Beaglebone Black embedded boards [12] when workload of the RUBiS website peaks (RT over 800 milliseconds in this experiment) as well as to remove these instances when the workload gets down (RT under 200 milliseconds). An AutoBot's IaaS plugin, called *Fog_Node*, has been developed to cover Fog computing's concepts. This plugin acting as a proxy to send requests to AutoBot agents located inside the Beaglebone Black instances. We do the same method in Section 7.2 to measure the deployment time of adding/removing a new JOnAS instance to/from a Beaglebone Black board. This results in about 310 seconds with the addition and 10 seconds with the removal.

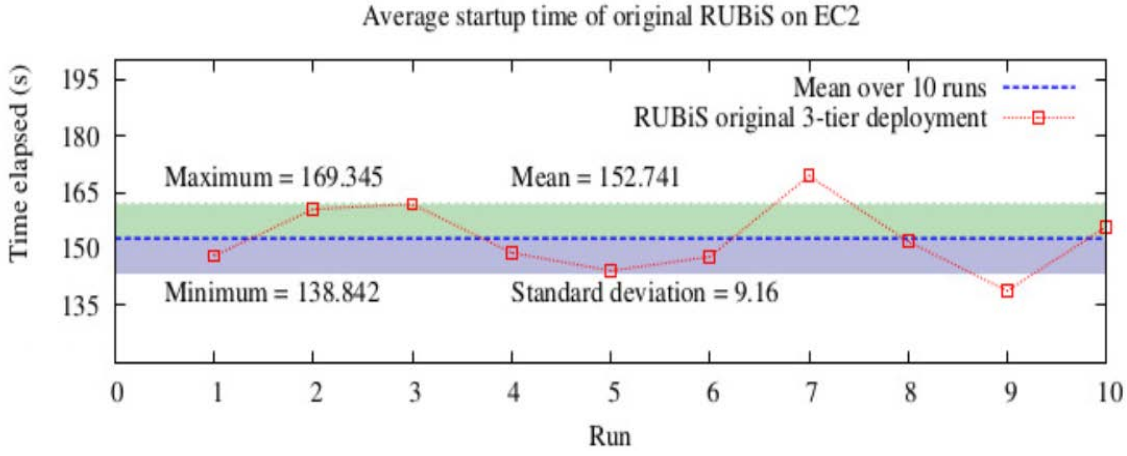

Fig. 11. Average startup time of original RUBiS on EC2

Table 2. Mean deployment time of an evolvable RUBiS application

Mean deployment time (s)		
	Mean	**99th percentile**
Original 3-tier RUBiS	152.7 ± 9.1	168.6 ± 11.2
Add a new Tomcat (*tomcat2*)	270.2 ± 37.7	289.3 ± 13.4
Add a JOnAS server	299.2 ± 43.1	394.2 ± 67.1
Remove *tomcat2*	8.6 ± 0.5	9.1 ± 0.3

Fig. 12 shows the response time fluctuates during the experiment time. First, the experiment starts with a RUBiS application consisting of one Apache instance, two JOnAS instances, and one MySQL instance all on Amazon EC2. Until around the 90th second, this system handles well the workload which is generated from the RUBiS client emulator. After the 90th second, the workload significantly increases resulting in a high RT. AutoBot whose *Auto_Repair* submodule was turned on triggers a command to add a new JOnAS instance to the system. As mentioned, the deployment time of JOnAS instance is about 310 seconds and during this period, the RT is still over 800 milliseconds. The RT gets a subtle peak around 420th second when the deployment time finished and the dynamic reconfiguration is happening. Afterward, the RT reduces under the max RT threshold thanks to the working together of the three JOnAS instances. Since the workload also decreases gradually, the RT goes under 200 milliseconds around 1220th second. At this point, AutoBot triggers another command to remove a redundant JOnAS instance. This process takes only 10 seconds and then

the RT goes over 200 milliseconds once more. Another removal could happen if the RT goes under the min RT threshold again.

This experiment validates the flexibility of AutoBot which can be extended to the environments beyond Cloud computing such as Fog computing. It also gives a proof that the submodules of the monitoring manager are functioning correctly.

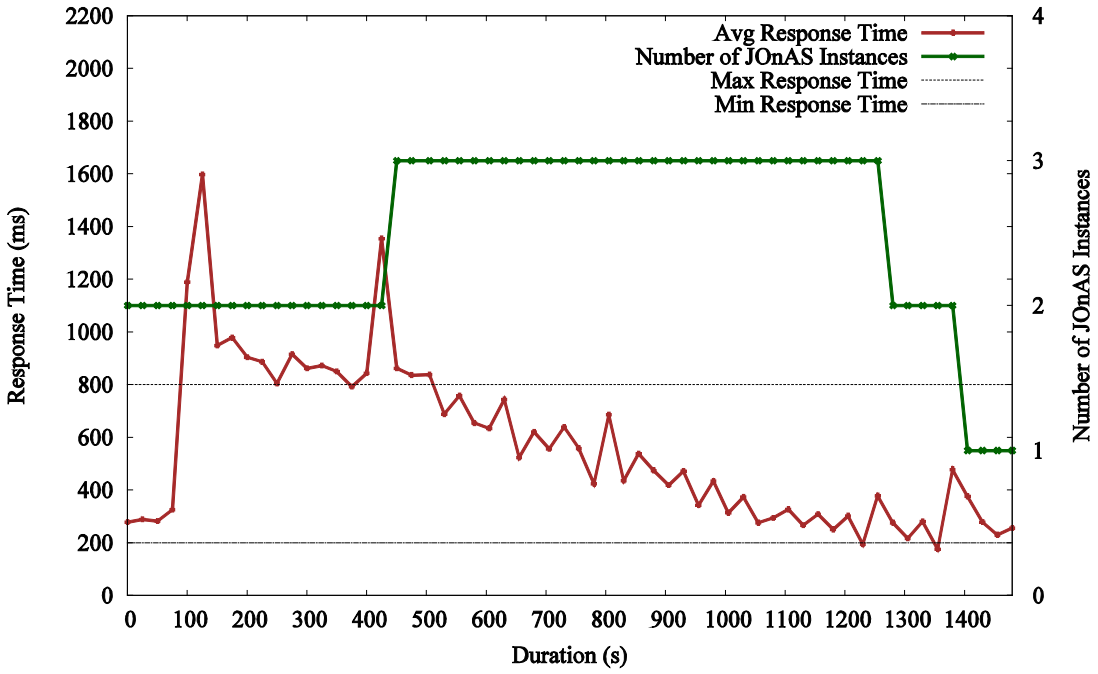

Fig. 12. Adding/Removing JOnAS instance in the context of Fog computing

7.4 The Overhead of AutoBot

Two thousand requests were generated and sent to the y-cruncher benchmark [13] to calculate the PI (π) to a specified number of digits after the decimal point (55 in this experiment) in order to evaluate the overhead introduced by AutoBot. Two cases were evaluated: i) the benchmark application deployed with AutoBot and ii) the benchmark application deployed without AutoBot. The experiments were conducted on VM c4.large instances (2 cores and 3.75 GB memory). In both scenarios, the requests were sent and executed 30 times. The results of the average execution time of each scenario as well as the additional costs brought by the AutoBot platform are presented in **Table 3**.

Table 3. Execution Time and Overhead

Scenario	Average execution time	Overhead introduced by AutoBot
Application	14.65	-
Application + AutoBot	15.25	4.1%

From the results, the AutoBot platform introduces only 4.1% performance overhead. This additional cost is generated mainly by exchanging configurations of the dynamic reconfiguration mechanism. In summary, the AutoBot platform introduces a negligible overhead given the aforementioned advantages.

8. Related work

According to the N.I.S.T. definition of the cloud [14], the objective of Platform as a Service (PaaS) layer is to provide models and environments for the automatic management of all life-cycle stages of applications deployed on Cloud. The deployment one is the first stage. Carzaniga et al. [15] defines the stage of deployment with the following four steps: (i) the packaging/versioning at the end of the application development, (ii) static configuration of the application, (iii) its installation in the runtime environment, and (iv) post-configuration and activation of the application. Package managers at operating-system-level (OSLPM) like apt, dpkg, or RPM manage dependencies between packages on a single machine and therefore differ from AutoBot. They do not provide application or configuration management or deployment of multiple machines. However, AutoBot complements these systems because it can use an OSLPM to install a software instance locally on a VM (AutoBot supports Bash scripts so that nearly any OSLPM can be used). It supports for resolution of higher-level problems involving the management and maintenance of distributed applications on heterogeneous platforms.

Tools like Chef and Puppet are intended to (i) set the local configuration of running hosts and (ii) keep them in accordance with the desired application status. These solutions follow a client/server approach where a server has the desired machine status and the client ensures that the configuration is set up. AutoBot improves in several ways on these management systems. First, the specification of the software types and their private details for installation are clearly separated. This relieves configuration pain and resolution of dependencies. AutoBot allows a very compact specification of a distributed application, which is smaller than the order of magnitude required to deploy a complex distributed application. A particular application can be easily deployed on different platforms (Linux-based, Windows-based, etc.) without much more work than a single installation would require. Lastly, although these management systems support application management across multiple servers, each client works in isolation and therefore does not coordinate between servers. AutoBot provides an easy way to coordinate and configure services on multiple machines depending on their dependencies.

About the standards of Cloud computing, Organization for the Advancement of Structured Information Standards (OASIS) creates TOSCA (Topology and Orchestration Specification for Cloud Applications) providing a language for describing the components comprising the model of cloud applications and the processes for orchestrating these components. Similar to this approach, CAMEL is another language aiming at enabling interoperability among the components of multi-cloud applications. This language also proposes application models at runtime that enables the continuous management of multi-cloud applications from design time to runtime operations. AutoBot not only provides its own DSL but also supports taking advantage of other standards by customizing the *DSL plugins*.

The nature of AutoBot is to resolve an optimization problem of application deployment in distributed environment with the given resources. On the same topic of optimization problems in Cloud and Fog, each of the following works focuses on a specific issue. Canali et al. [16] also resolve deployment problem in Cloud data center but they only focus on virtual elements (VE). To minimize the energy consumption in a software-defined Cloud data center, the energy consumption is modeled by considering the computing costs of the VEs on the physical servers, the costs for migrating VEs across the servers, and the costs for transferring data between VEs. In [17], the same group of authors takes the maintenance cost into account and proposes the Maintenance and Electricity Costs Data Center (MECDC) algorithm to solve the optimization problem of balancing electricity consumption and maintenance costs in Cloud data centers. In another direction for software-defined Cloud data centers, Tajiki et al. [18]

propose CECT, a computationally efficient congestion avoidance scheme. Based on the flow requirements, the proposed algorithm not only reallocates the resources but also minimizes network congestion. In the scope of Fog computing, novel optimization algorithms are also proposed to resolve the selection problems of when 5G mobile users should offload computing workload to Wi-Fi [19] and when Fog servers should cluster the nearby cloudlets for offloading the task of a mobile [20]. In the same vein, energy-efficient and real-time management of the distributed resources available along the Fog strata is taken into consideration in [21]. The authors propose an energy-efficient adaptive scheduler for Vehicular Fog Computing (VFC), which optimizes the energy by taking advantage of the heterogeneity of Foglets (FIs). The FI provider shapes the system workload by maximizing the task admission rate over computation and data transfer. To facilitate mobile malware identification, Afifi et al. [22] propose a hybrid method to find the optimal parameters used for this purpose. In data preparation phase, a multi agent system architecture including sniffer, extraction, and selection agents is presented to capture and manage the pcap file. The principles used in these proposed algorithms can be applied by AutoBot to enhance its modules involving the optimization of application deployment.

Few of the heterogeneous industrial PaaS solutions offer fully automated systems to deploy an application regardless of its programming language, programming convention and model, business domain, infrastructure, and runtime environment. Engage [23], PaaSage [24], and Roboconf [25] are good representatives of these systems, which provide not only a method for describing distributed applications, but also a complete runtime system for managing these deployed applications. While PaaSage develops many efficient constraint-based optimization tools such as MILP and CP solvers, Engage predominates in enabling static checking of application configurations. Yet, Engage and PaaSage lacks some flexibility in inserting or releasing new software instances dynamically during runtime while still ensuring the proper resolution. Since they only depend on partial specifications rather than complete ones, there can be no guarantees that a software instance is solved even if an advanced constraint solver used. Although they provide elegant mechanisms to roll back to last state if an upgrade fails, the two do not detail in this feature further. Roboconf is the first research presenting a declarative language to describe the hierarchy of software components both horizontally and vertically. Nevertheless, Roboconf is built around its own describing language which is not a standard to describe the deployment of a distributed application. Furthermore, Roboconf does not have the means to describe the local configuration and, more generally, the content of the virtual appliances participating in the application to be deployed and instantiated in VMs. Since these PaaS solutions lack some precious features, we have chosen to design a simpler model for any distributed applications.

Similar to AutoBot, some works also aim to provide flexible and portable deployment solutions in Multicloud in terms of functional, data and service portability. Semantic approach of Cloud4SOA project [26] deals with the portability between PaaS. mOSAIC [27] provides service enhancement portability at both PaaS and IaaS levels. It includes a component-based application model along with asynchronous communication. Authors of soCloud solution [28] develop an API to offer service enhancement portability also running on IaaS and PaaS. Unlike mOSAIC, soCloud supports both synchronous and asynchronous communications. AutoBot not only aims to a flexible and portable application deployment solution on the Cloud but also provides the *DSL plugins* to abstractly describe concepts of heterogeneous environments beyond the Cloud such as Fog computing.

In recent years, several works in deployment and (re)configuration of the components in Fog computing have been introduced in the literature. Firstly, a service orchestration

architecture for Fog-enabled infrastructures is proposed by Brito et al. [29]. This orchestrator enables Fog nodes capability of running virtualized and containerized applications and offers these applications capability of accessing to connected/attached devices over different communication technologies. In another research, a Fog computing platform, which dynamically pushes programs to the devices, is implemented by Hong et al. [30]. These programs pre-process the data before transmitting them over the Internet in order to reduce the network traffic and the load of Cloud data centers. Moreover, this paper formulates a deployment problem of the programs and proposes an efficient heuristic deployment algorithm to solve it. The authors of [31] seek a solution to reduce the network costs and response time for the user by developing a resource-aware placement of data analytics platform in Fog computing architecture, which would adaptively deploy software components of the analytic platform to run either on the Cloud or the Fog. A novel integrated fog cloud IoT (IFCIoT) architectural model is introduced by the authors in [32]. The layers such as application layer, analytics layer, virtualization layer, reconfiguration layer, and hardware layer are included in the architecture. These layers facilitate abstraction and implementation for Fog computing paradigm where various vendors such as data, applications, services, and content providers are involved. Although these works contribute to the development of D&C solutions for Fog computing, they all do not reuse D&C modules developed previously for Cloud computing, which may increase the development time and cost. AutoBot aims to a platform where portable modules once developed and run well in Cloud will be extended and reused in other distributed environments such as Fog computing.

9. Conclusion and Perspectives

Dynamic configuration and deployment of complex distributed applications are broadly preferred but management systems are often complicated and expensive in practice. This is the reason we decided to rethink how to model, deploy and manage dynamic distributed application stacks. We introduce an explicit and concise domain specific language which offers the needed syntax to describe any sort of distributed applications, their configuration variables and interdependencies. The paper also presents a platform, AutoBot, that takes advantage of this language, which ensures (i) the successful parallel deployment of the components on the cloud infrastructure using minimal IaaS primitives, (ii) the coordination of the global configuration via a decentralized configuration protocol across multiple VMs and (iii) the capability to add/remove new software components during runtime to a deployed application. An important perspective of this work is the extension of the domain specific language to cover extremely heterogeneous environment of Internet of Things applications which can be deployed and managed automatically by a mechanism evolved from the D&C protocol proposed in this paper. Another perspective is that the platform is designed as flexibly as possible, thus its components can be added, reused or improved to serve for not only Cloud computing applications but also Fog computing ones.

References

[1] Yin, Z., Ma, X., Zheng, J., Zhou, Y., Bairavasundaram, L.N., Pasupathy, S., “An empirical study on configuration errors in commercial and open source systems,” in *Proc. of the 23th ACM Symposium on Operating Systems Principles*, ACM, pp. 159–172, New York, NY, USA, 2011. Article (CrossRef Link).

[2] Kramer, J., Magee, J., "The evolving philosophers problem: Dynamic change management," *Software Engineering, IEEE Transactions on*, 16(11), 1293–1306, 1990. Article (CrossRef Link).
[3] Adler, B., "AWS Outage Lessons Learned: If Netflix Can Suffer," *So Can You*, 2013. https://www.rightscale.com/blog/cloud-management-best-practices/aws-outage-lessons-learned-if-netflix-can-suffer-so-can-you
[4] Linh Manh Pham and Truong-Thang Nguyen, "AutoBot: a versatile platform for management of legacy applications in the cloud," in *Proc. of the Seventh Symposium on Information and Communication Technology (SoICT '16)*, ACM, 403-410, New York, NY, USA, 2016. Article (CrossRef Link).
[5] Mouradian, C., Naboulsi, D., Yangui, S., Glitho, R.H., Morrow, M.J., Polakos, P.A., "A comprehensive survey on fog computing: State-of-the-art and research challenges," *IEEE Communications Surveys Tutorials*, 20(1), 416–464, 2018. Article (CrossRef Link).
[6] O. T. TC., Topology and orchestration specification for cloud applications version 1.0, Nov. 2012. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/csprd01/TOSCA-v1.0-csprd01.pdf
[7] CAMEL Documentation v2015.9. Available: https://gitlab.ow2.org/paasage/camel/raw/master/documents/CAMELDocumentation.pdf
[8] Opscode Chef website. URL http://wiki.opscode.com/display/chef/Home
[9] Puppet Labs website. URL http://docs.puppetlabs.com/
[10] RUBiS OW2 website. URL http://rubis.ow2.org/
[11] JOnAS OW2 website. URL https://jonas.ow2.org/
[12] Beaglebone Black System Reference Manual. Available: https://cdn-shop.adafruit.com/datasheets/BBB_SRM.pdf
[13] Y-cruncher benchmark. URL http://www.numberworld.org/y-cruncher/
[14] Peter M. Mell and Timothy Grance, SP 800-145, the NIST Definition of Cloud Computing. Technical Report. NIST, Gaithersburg, MD, United States, 2011.
[15] Carzaniga, A., Fuggetta, A., Hall, R.S., Heimbigner, D., van der Hoek, A., Wolf, A.L., "A characterization framework for software deployment technologies," *Tech. rep.*, 1998.
[16] C. Canali, L. Chiaraviglio, R. Lancellotti and M. Shojafar, "Joint Minimization of the Energy Costs from Computing, Data Transmission, and Migrations in Cloud Data Centers," *IEEE Transactions on Green Communications and Networking*, vol. 2, no. 2, pp. 580-595, June 2018. Article (CrossRef Link).
[17] L. Chiaraviglio, F. D'Andreagiovanni, R. Lancellotti, M. Shojafar, N. Blefari Melazzi and C. Canali, "An Approach to Balance Maintenance Costs and Electricity Consumption in Cloud Data Centers," *IEEE Transactions on Sustainable Computing*, vol. 3, no. 4, pp. 274-2088, May 2018. Article (CrossRef Link).
[18] Tajiki, M.M., Akbari, B., Shojafar, M. et al., "CECT: computationally efficient congestion-avoidance and traffic engineering in software-defined cloud data centers," *Cluster Comput*, 2018. Article (CrossRef Link).
[19] J. Ho, J. Zhang and M. Jo, "Selective offloading to WiFi devices for 5G mobile users," in *Proc. of 2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, Valencia, pp. 1047-1054, 2017. Article (CrossRef Link).
[20] Y. Li, N. T. Anh, A. S. Nooh, K. Ra and M. Jo, "Dynamic mobile cloudlet clustering for fog computing," in *Proc. of 2018 International Conference on Electronics, Information, and Communication (ICEIC)*, Honolulu, HI, pp. 1-4, 2018. Article (CrossRef Link).
[21] Naranjo, P.G.V.; Pooranian, Z.; Shamshirband, S.; Abawajy, J.H., "Conti, M. Fog over Virtualized IoT: New Opportunity for Context-Aware Networked Applications and a Case Study," *Appl. Sci.*, 7(12), 1325, 2017. Article (CrossRef Link).
[22] Afifi F, Anuar NB, Shamshirband S, Choo K-KR, "DyHAP: Dynamic Hybrid ANFIS-PSO Approach for Predicting Mobile Malware," *PLoS ONE*, 11(9): e0162627, 2016. Article (CrossRef Link).
[23] Fischer, J., Majumdar, R., Esmaeilsabzali, S., "Engage: a deployment management system," *PLDI'12*, ACM, pp. 263–274, 2012. Article (CrossRef Link).
[24] PaaSage Cloud Platform. URL https://www.paasage.eu/

[25] L. M. Pham and T. M. Pham, "Autonomic fine-grained migration and replication of component-based applications across multi-clouds," in *Proc. of 2015 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, Ho Chi Minh City, pp. 5-10, 2015. Article (CrossRef Link).
[26] Kamateri E. et al., "Cloud4SOA: A Semantic-Interoperability PaaS Solution for Multi-cloud Platform Management and Portability," *Lau KK., Lamersdorf W., Pimentel E. (eds) Service-Oriented and Cloud Computing, ESOCC 2013. Lecture Notes in Computer Science*, vol 8135, Springer, Berlin, Heidelberg, 2013. Article (CrossRef Link).
[27] Petcu, D., Macariu, G., Panica, S., Crăciun, C., "Portable Cloud applications from theory to practice," *Future Generation Computer Systems*, 2012. Article (CrossRef Link).
[28] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier, "soCloud: a service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds," *Computing 98*, 5, 539-565, May 2016. Article (CrossRef Link).
[29] M. S. de Brito et al., "A service orchestration architecture for Fog-enabled infrastructures," in *Proc. of* 2017 Second International Conference on Fog and Mobile Edge Computing (FMEC), Valencia, pp. 127-132, 2017. Article (CrossRef Link).
[30] H. Hong, P. Tsai and C. Hsu, "Dynamic module deployment in a fog computing platform," in *Proc. of* 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS), Kanazawa, pp. 1-6, 2016. Article (CrossRef Link).
[31] Mohit Taneja, Alan Davy, "Resource Aware Placement of Data Analytics Platform in Fog Computing," *Procedia Computer Science*, Volume 97, Pages 1 3-156, ISSN 1877-0509, 2016. Article (CrossRef Link).
[32] A. Munir, P. Kansakar and S. U. Khan, "IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things," *IEEE Consumer Electronics Magazine*, vol. 6, no. 3, pp. 74-82, July 2017. Article (CrossRef Link).

Dr. Linh Manh Pham is a lecturer at University of Engineering and Technology, Vietnam National University, Hanoi (VNU). He was a postdoctoral researcher at Inria, France. He earned an MSc. in Computer Science in the USA and a Ph.D. in Cloud Computing in France. His area of research is Cloud/Fog Computing, and he intends to highlight the benefits of this relatively novel field of research.

Dr. Truong-Thang Nguyen is the Director of Institute of Information Technology, Vietnam Academy of Science and Technology since 2015. He has an MSc. and a Ph.D. both in Software Engineering in JAIST, Japan. His key areas of research include formal methods, SMAC, and network security. Moreover, Dr. Nguyen is the lead consultant for important government IT projects.