An Adaptable Framework to Deploy Complex Applications onto Multi-cloud Platforms
Linh Manh Pham, Alain Tchana, Didier Donsez, Vincent Zurczak, Pierre-Yves Gibello, Noel de Palma

▶ To cite this version:

HAL Id: hal-01232615
https://hal.archives-ouvertes.fr/hal-01232615
Submitted on 23 Nov 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
An Adaptable Framework to Deploy Complex Applications onto Multi-cloud Platforms

Linh Manh Pham¹, Alain Tchana², Didier Donsez³, Vincent Zurczak³, Pierre-Yves Gibello⁴, Noel de Palma¹
¹University of Joseph Fourier, Grenoble, France. E-mail: first.last@imag.fr
²University of Toulouse, Toulouse, France. E-mail: first.last@enseeiht.fr
³Linagora, Grenoble, France. E-mail: (vincent.zurczak, pygibello)@linagora.com

Abstract—Cloud computing is nowadays a popular technology for hosting IT services. However, deploying and reconfiguring complex applications involving multiple software components, which are distributed on many virtual machines running on single or multi-cloud platforms, is error-prone and time-consuming for human administrators. Existing deployment frameworks are most of the time either dedicated to a unique type of application (e.g. JEE applications) or address a single cloud platform (e.g. Amazon EC2). This paper presents a novel distributed application management framework for multi-cloud platforms. It provides a Domain Specific Language (DSL) which allows to describe applications and their execution environments (cloud platforms) in a hierarchical way in order to provide a fine-grained management. This framework implements an asynchronous and parallel deployment protocol which accelerates and makes resilient the deployment process. A prototype has been developed to serve conducting intensive experiments with different type of applications (e.g. OSGi application and ubiquitous big data analytics for IoT) over disparate cloud models (e.g. private, hybrid, and multi-cloud), which validate the genericity of the framework. These experiments also demonstrate its efficiency comparing to existing frameworks such as Cloudify.

Keywords—Middleware/business logie, Software Engineering/Management

I. INTRODUCTION

For many decades now, we are seeing the continuous growth of the complexity of applications, due to the development of new technologies on the one hand, and the emergence of new needs on the other hand. An application does not address a single problem but several. This growth of their complexity implies the same phenomenon regarding their execution environment on organization of physical machines or devices. For instance, this has brought forward a change from centralized to distributed and heterogeneous place of execution. All of this makes human administration very difficult because they are errors prone, slow to respond (e.g. fault solving), and highly costly (e.g. wages). In the early 2000s, IBM [1] proposed to automate their administration throughout the use of what we called Autonomic Computing Systems (ACS for short). This practice consists in transferring human administration knowledge and behaviors to computing systems. This can be done in two ways, either by introducing autonomic behaviors into applications components at its implementation time (e.g. [9]) or by building a computing system (different to the application we want to administrate) which will make the application autonomous [4]. In this paper we consider the second way. As summarized by [1], administration tasks can be divided into the following categories: (un)installation and reconfiguration. The former includes the initialized provisioning of the execution environment, the installation of artifacts from repositories, the configuration of the application, its start-up as well as stopping and releasing allocated resources when needed. About reconfiguration tasks, they are performed at runtime in order to reconfigure the application when a particular situation is detected (e.g. fault). ACSs have proved their usefulness and now the major part of research in this topic focuses on reconfigurations tasks [2].

However, recent years have seen the development of a new technology called Cloud computing which is a challenging domain for existing ACSs, as it introduces an intermediate level of administration for virtual machines (VMs). Moreover, it sometimes requires the utilization of several clouds at once (hybrid and multi-cloud). To make matter worse, clouds API are not standardized, which results to non interoperable clouds. For example, running an enterprise financial/bank application within the Cloud generally requires two clouds: a private cloud (e.g. vSphere or OpenStack) located in the company to run business-critical part and a public cloud (e.g. EC2 or Azure) to run non-critical part. Note that in some situations, the latter part can move from one cloud to another for price and competitiveness reasons.

In this context, existing ACSs [3], [6] are inappropriate for several reasons. (1) Existing ACSs only consider one level of deployment/execution: an application runs within a physical machine, whereas in the subject of the Cloud, the application runs within a VM or a container, which in turn runs on a physical machine. (2) The target execution environment is not static in the context of the Cloud, an application does not stay within the same cloud during its overall lifetime. (3) Existing ACSs are built to administrate both application and execution environment while in the context of cloud, administration is ensured by two actors: the deployer administrates its application while the cloud provider administrates VMs and physical machines. Although generic ACSs [4] for grids and clusters of machines exist, their enhancement for clouds requires a high expertise for the deployer. Concerning cloud solutions, they [14], [15], [16] are either proprietary, devote to a specific application, or target a static cloud.

This paper addresses these problems by proposing a generic (administrate any kind of applications), extensible, multi-cloud (target several clouds at once), scalable, and fine-grained reconfigurable deployment framework. The key ideas behind the framework are the following: (1) It is the composition of a ACS
light-weight kernel which implements basic administration mechanisms; (2) a set of reusable components which are easily improvable by any deployer; and (3) a hierarchical DSL (Domain Specific Language) for a fine-grained expression of applications and execution environments. To keep the thing simple, we focus on a subset of administration stacks: initial installation (including provisioning, configuration, start-up, software deployment, stopping and uninstallation), dynamic installation at runtime, and incrementally partial or full application installation. In summary, we make the following contributions in this paper: (1) introduce a generic, improvable, and scalable deployment framework for multi-cloud applications. The prototype of this system is open source1 and is actually in use by several enterprises [32]; (2) perform several experiments which validate all the properties of the proposed framework. We deploy a web application onto a hybrid cloud (a private VMware vSphere hosting center combined with Amazon EC2 [33] and Azure [34] clouds). We also deploy an application utilizing for Home Automation (OpenHAB [35]) on embedded board, EC2 and Azure clouds; (3) propose a hierarchical DSL which allows fine-grained administration.

The rest of the paper is organized as follows. Section II presents the related work. Section III presents architecture of our deployment system. The evaluation and results are presented in Section IV. Lastly, Section V concludes the paper.

II. RELATED WORK

A lot of research works [7] have been devoted to the automation of the administration of distributed applications in cluster or grid environments. Some of them [3], [6] have focused on the installation phase. [5] presents a survey of software installation until 2007. Since the introduction of the Cloud computing technology, most research in the domain of autonomic computing for grid systems have been refocused. This section presents what is done for Cloud platforms.

As for grid systems, a number of research in the context of Cloud computing are dedicated either to a single cloud platform or a single type of applications. [8], [12] present a set of deployment frameworks in order to provide scalable testing solutions based on the Cloud. [10] focuses on fault tolerance on machine, VM and software component at runtime. Like our work, it relies on a publish/subscribe mechanism to coordinate the state of deployed components. This approach also allows to paralyze the deployment and facilitates fault detection. [9] is comparable to [4] which provides a very low level API to the deployer for its improvement. [25] presents Engage, a deployment framework which is very close to [26], it allows the user to only express a partial installation specification and it generates the full one. In contrast, our system provides a comparable feature since it is able to deploy a stack of software from a partial specification. [11] motivates the use of Model Driven Engineering as we do in our framework to build a useful multi-cloud platform. Therefore, it introduces CloudML which can be seen as a sub-part of our DSL. [13] focus on the migration of enterprise applications to hybrid cloud while considering enterprises constraints such as price, location, etc.

Very few solutions are close to our approach. Most of them are proprietaries and do not provide detailed documenta-

1http://roboconf.net/
solutions such as Vagrant [28] or Docker [27]. The agents communicate with each other through an asynchronous messaging server; (3) The SoftwareInstanceManager is developed as a plug-in to generate software life-cycle management on different software platform and monitor software instances themselves; (4) The Messaging Server is the key component acting as distributed registry of import/export variables that enable communications between the DM and the agents. It includes the message definitions, the interface to interact with a given messaging server (currently RabbitMQ [19]) and their implementations. The DM and the agents always communicate asynchronously through this server; (5) The Artifact and VM Image repositories are responsible for distribution software packages (i.e. artifact) and VM’s image, respectively. Artifact repositories can be managed locally or retrieved from public repositories such as Maven center or NPM. Image repository is a database to map each required VM image of each IaaS to corresponding infrastructure components. The required VM image can be an image available in the VM image marketplace provided by IaaS or a pre-built image created manually or automatically (e.g. using Docker or Vagrant); (6) Eventually, an admin console is required to control the DM. A shell-based console has been developed to interact with the DM through REST. It contains utilities to transform Java beans into JSON.

The framework is a distributed technology, based on AMQP [21] and REST/JSON. It is both IaaS and PaaS-agnostic. Many well-known IaaS are advocated including OpenStack, Amazon Web Services, Microsoft Windows Azure, VMware vSphere, as well as a “local” deployment plug-in for on-premise hosts. In the PaaS aspect, not only potential type of applications are densely brought up to the Cloud such as OSGi or Internet of Things (i.e. IoT) [29] but also state-of-the-art PaaS are purposefully encompassed such as Heroku, Google App Engine and CloudBees. Its architecture allows to satisfy most of state-of-the-art requirements of a modern multi-cloud PaaS such as dynamic dependency resolution, concurrent deployment, genericity, multi-cloud distributed deployment, middleware-orientation, modularity, extensibility, portability, scalability and reusable/configurable deployment plan.

B. A Hierarchical Architecture with DSL

The aforementioned framework is designed to see a distributed application as a set of “components”, and as a group of “instances” of these components. Let us take as an example the three-tier distributed application “Apache-Tomcat-MySQL”. "Apache" is a component, while an install of Apache on a particular machine is an instance. Another install of Apache on another machine is another instance. Besides, our system is built to see distributed application as a group of components that each one exchanges a group of simple data between each other. Data can be either strings or more complex objects. Components of a distributed application are composed of variables as for example the IP address or the port used. Parts of those variables may be needed by other components of the application, they are named "exported vars", while vars coming from other components of the application are named "imported vars". Moreover, definition of a component can be inherited by definition of another according to object-oriented design. It inherits all import/export vars and default values. For instance, Tomcat component can inherit properties of a generic "Application Server” component.

Now that we have a far view of an application, let us explain more precisely what are its components. In the above example, Apache in this case simply imports variables coming from Tomcat: IP and port of the application server. As we said earlier, we define elements (component and instance) as having a set of exported and imported variables. In this case of Apache there are only imported variables. A sample of Apache component under the framework’s language could be find in Fig. 2a left.

This portion is made up of several regions. alias: A component declaration. It is mandatory and provides a human-readable name. installer: A component property. It is mandatory and designates the plug-in that will handle the life cycle of component instances. In this example, we are using puppet implementation. imports: lists the variables this components needs to be resolved before starting. Variable names are separated by commas. They are also prefixed by the component that exports them. As an example, if Tomcat exports the ip variable, then a depending component will import Tomcat.ip. On the other hand, MySQL does not import data from other components, it is the only one exporting data which are its IP and port. The definition of MySQL is found at in Fig. 2a right.

Here has a minor different from the exports which lists the variables this component makes visible to other components. The “ip” is a special variable name whose value will be set by the agents at runtime. All the other variables should specify a default value. In terms of model and configuration files, the framework has the following concepts: (1) The application descriptor contains meta-information of the application such as name, version qualifier and description; (2) The graph is in fact a set of connected components. It defines Software components which go from the (virtual) machine, cloud platform to
the application package. The graph defines both containment and runtime relations. Two kinds of relations are defined as follows: (a) Containment means a component can be deployed over another one. As an example, a Tomcat server can be deployed over a VM. Or a web application (WAR) can be deployed over a Tomcat server (see Fig. 2b); (b) Runtime relations refer to components that work together. For instance, a web application needs a database. More specifically, it needs the IP address and the port of the database. Generally, this information is hard-coded whereas our system can instead resolve them at runtime and update components through the configuration or management APIs (e.g. JMX, REST). As an example, Apache, Tomcat and MySQL can be deployed simultaneously. Tomcat will be deployed but will not be able to start until it knows where is the database. Once the database is deployed and started, the system will update Tomcat configuration so that it knows where is MySQL. This is what dependency resolution at runtime makes possible; (3) If the graph defines relations between components, instances represent concrete components. Like a Java class, a component is only a definition. It needs to be instantiated to be used. Predefined instances aim at gaining some time when one wants to deploy application parts. As an example, the deployer could have defined a Tomcat component in the graph, and have two instances, one deployed on machine A, and another on machine B and other two on machine C. These would be four instances of the same component. The rules that apply to them are deduced from the graph, but they have their own configuration. An application is designed to see as hierarchy of components. The main motivation of hierarchy is to keep track of where instances are implemented in the system. It helps to make right decisions in dynamic deployment. A natural example of parent/children relationship of components of an OSGi application is depicted in Fig. 3.

There is a new important field: children which lists the components that can be instantiated and deployed over this component. In the example above, it means we can deploy Karaf over a VM instance. In turn, Joram and JNDI can be deployed over instances of Karaf.

While the hierarchical model resolves the containment relations (i.e. vertical relationship) among components at different layers and the export/import variables model is responsible for solving the runtime relations (i.e. horizontal relationship), the export/import variables model is responsible for solving the runtime relations (i.e. horizontal relationship) amongst components at the same tier, a bi-color Graph put everything together using the DSL. What is modelled in the graph is really a user choice. Various granularity can be described. It can goes very deeply in the structure of application or bundle components together such as associating a given WAR with an application server. Multi-IaaS is supported by defining several root components. Each one will be associated with various properties (e.g. IaaS provider, VM type). In fact, the deployer can define a single graph, or a collection of graphs. The interest of having several graphs is to define virtual appliances.

C. Deployment Process

1) Initial Deployment Process: We use three-tier example to understand how our approach works. As mentioned, dependencies between components is presented in Fig. 4. The goal is to deploy Apache, MySQL, Tomcat, Rubis instances on separate VMs that similar to Fig. 5b. The target software on Tomcat named RUBiS [36] which is a JEE application based on servlets, which implements an auction web site modelled on eBay. RUBiS defines interactions such as registering new users, browsing, buying or selling items. The updating the configuration files is performed as soon as dependencies can be resolved (e.g. when it is aware of the MySQL IP/port, this information will be sent to the Rubis nodes, so they can update its configuration and start). The application components (MySQL, Tomcat, Apache, Rubis) are defined as Fig. 5a. The VMs from disparate Cloud providers are supposed to support the deployment of either Apache, Tomcat, MySQL or Rubis components and each component is described in terms of imports/exports. With this description, the system knows when a deployed component can be started. It is when all its imports are resolved.

2) Redeployment Process: It often happens when everything is running, we need to create a new instance to adapt to changes from environment. It means the running system needs...
Fig. 5: (a) Example of a DSL Graph of components; (b) Example of a DSL Graph of instances.

IV. EXPERIMENTAL EVALUATION

To validate the soundness of the architecture, we conducted a number of experiments with scenarios selected from practical use cases.

A. Experiment 1

The first type of experiments demonstrates the advantage of hierarchical structure which is aforementioned. For this experiment, EC2 was the target cloud.

1) Scenario and Requirements: We performed this experiment with an OSGi application. Regularly, an OSGi application consists of one or several OSGi containers (e.g. Karaf, Felix, Equinox) providing runtime environment and management platform for OSGi bundles such as Joram, JNDI. We used two instances of EC2 m3.medium, each hosts 2 instances of Karaf. Each Karaf inside a VM is customized to choose either Felix or Equinox as underlying OSGi framework and hosts an instance of Joram (an OSGi JMS-supported server), or an instance OSGi JNDI or a JMS OSGI client (publisher/subscriber). Deployment of Joram, JNDI and OSGI JMS clients is handled by the "osgi-bundle" installer, specific to this type of application. We chose Cloudify as comparative objective because it also offers scripting language that can be used to express the structure of a distributed application. Our platform sees hierarchy of this application as depicted in Fig. 6 top, whereas Fig. 6 bottom shows flat structure of the same application in Cloudify.

2) Results: With its hierarchical DSL and its extensibility, users only have to write one deployment plan for EC2, one for Karaf OSGi container and reuse one plan for multiple OSGI bundles (Joram, JNDI, subscriber, publisher). With Cloudify, 6 deployment plans are needed, each one for each component (EC2, Karaf, Joram, JNDI, Subscriber, Publisher). Thus Cloudify users have to write twice more deployment plans.

B. Experiment 2

The second type of experiments gives some evidence for the correctness of the multi-cloud deployment feature. We compare deployment time of a Storm [30] cluster (an Event Stream Processing (ESP) application) on a multi-cloud platforms using our framework on the one hand and a manual configuration following installation guide from original owner on the other hand. Storm is part of a global solution for Ubilutecs [31] (ubiquitous big data analytics for IoT). Storm consists of Zookeeper cluster, Nimbus server, Storm supervisors and requires installation of IZMQ, ZeroMQ and Python. The experiment was conducted in a multi-cloud environment combining two public clouds (EC2 and Azure) and a private cloud (VMware vSphere). Three IaaS plug-ins for these clouds have been developed to provide coordination among the three IaaS providers. Each plug-in implementation always implements an interface of the framework’s plug-in API. The LOCs (lines-of-code) for the EC2 plug-in is 202, 393 for Azure, and 157 for VMware vSphere. Zookeeper cluster was installed on EC2, Nimbus server on Azure and Storm supervisors on our VMware vSphere data-center to take advantage of our computing strength. In this experiment, the time for installing Storm manually is compared with the time to automate its installation using our solution. The online installation guide of Storm is 8 pages long containing many external links to resource document of relevant dependencies. One of the
authors who had no knowledge about and attempted to install Storm previously tried to do manual installations. It took him about 6 hours the first time, 3 hours and 30 minutes the second time, and up to 1 hour from the third one. Actions eating effort time were reading imprecise instructions, resolving environment issues, seeking/downloading the required dependencies and debugging problems. On the novel framework side, time mainly devotes for writing deployment plan of Zookeeper, Nimbus, Supervisors, JZMQ, ZeroMQ and Python. About 120 LOC have been written for scripts of all Storm’s components. After installation, Storm can be managed (deploy, start, stop, undeploy, update) by our platform and automatically connect to other applications. The total development time for automation of Storm was about 2 hours 55 minutes. This time was divided into 30 minutes for component type’s design, 70 minutes for writing scripts and 75 minutes for debugging and testing. If the required packages are downloaded from the Internet, install of Storm needs 20 minutes and around 7 minutes if the packages are retrieved from a local repository. The automation of the Storm installation empowers Storm developers deploy their existing applications on multi-cloud with slight changes and no need to understand details of the middleware. It warrants a repeatable procedure and can be used as a part of larger deployments (e.g. Ubilitycs).

V. CONCLUSION

The novel distributed framework we propose in this paper is a generic (administrate any kind of application), extensible, multi-cloud, scalable, and fine-grained reconfigurable deployment framework. It is based on a lightweight kernel which implements basic administration mechanisms. Its simplicity (regarding its implementation) combined with its component based approach ease its improvement by any deployer. More important, it provides a hierarchical DSL (Domain Specific Language) for a fine-grained expression of applications and execution environments. This allows it to achieve fine-grained level of administration (physical machine, VM, software within VM, other stack inside software). Our experiments about OSGi, Storm cluster validate all the framework’s features on hybrid cloud. In addition, it outperforms most popular deployment frameworks offering DSL like Cloudify in terms of reusability. A further enhancement would be the implementation of a way to facilitate the integration of more sophisticated reconfiguration policies (e.g. for scalability, fault tolerance).

ACKNOWLEDGMENT

The work reported in this paper benefited from the advocate of the French National Research Agency through projects CtrlGreen (ANR-11-INFR-0012).

REFERENCES