Vietnam J Comput Sci (2017) 4:39–49
DOI 10.1007/s40595-016-0074-0

REGULAR PAPER

Autonomic fine-grained replication and migration at component level on multicloud

Linh Manh Pham[1] · **Tuan-Minh Pham**[2]

Received: 30 November 2015 / Accepted: 3 July 2016 / Published online: 12 July 2016

Abstract Although migration and replication of applications in a distributed environment have been discussed by many researchers, the implementations of these features are rarely focused when deployed in the cloud. The cloud enterprises usually have to migrate or replicate partly or fully their services because of economical or disaster preventing reasons. Because the cost of copying the whole virtual machines is too high due to their big size, the replication at application level is a possible approach. This work proposes an autonomic replication and migration mechanism integrated in an implementation of a fine-grained deployment framework which enables ability to migrate and replicate-service components on the clouds. We formulate the deployment problem of replicated components to optimize the system performance as a quadratic program. Our proposed framework ensures the high availability and scalability of services, and complies with the service-oriented architecture. Our experiments conducted in real scenarios of elastic demands demonstrate that the proposed fine-grained migration and replication is more efficient than the coarse-grained ones when an autonomic system responds to fluctuation of webapp's workload. We also show the influence of adding servers and upgrading server connections on the system performance.

Keywords Autonomic computing · Cloud computing · Application migration · Application replication · Network virtualization

1 Introduction

Cloud computing is a recent trend of information technology, with its application distributed in every field. With minimal effort or service provider interaction, the configurable computing resources can be rapidly provisioned and released on demand with a pay-as-you-go style [1]. Ideally, all things a cloud user needs are a machine with an enabled web browser. In terms of service model, there are three well-discussed layers known as IaaS for Infrastructure as a Service, PaaS for Platform as a Service, and SaaS for Software as a Service. Many other XaaS terms are used nowadays to name different provided services in the cloud.

Cloud environments can be used to host service-based applications following a service-oriented architecture (SOA). SOA is a collection of self-contained services which communicate with each other using provided interfaces [2]. Management of service-based applications in cloud environments is a challenging task in the aspects of fault tolerant, performance and security. Cloud service management plays an important role to respect the service-level agreement (SLA) between the service consumers and providers. In this context, one of the solutions to ensure the SLA is the ability to support mobility services which allows the migration and replication of services between virtual machines (VM) or among different containments. Many attempts to provide migration and replication of service-based applications in the cloud exist. They can be classified into three categories: application-centric migration, image-based migration and migration to a virtualized container. The application-centric

✉ Tuan-Minh Pham
minhpt@hnue.edu.vn

Linh Manh Pham
linh-manh.pham@imag.fr

1 Laboratoire d'Informatique de Grenoble, University of Grenoble Alpes, Grenoble, France

2 Faculty of Information Technology, Hanoi National University of Education, Hanoi, Vietnam

migration such as [3,4] extracts and migrates application artifact, resources and configuration from the source to a new provisioned application deployment environment on the target. The image-based migration such as [5,6] converts source into VM images and imports them into target cloud after some adjustment. The migration to virtualized container technique, such as [7] migrates the source VM to run in a virtualized container inside the target without any modifications. All of the mentioned approaches do not offer an autonomically fine-grained solution at service component level. Such a solution helps cloud users mitigate manual effort which is tedious and error-prone. Furthermore, much research so far pay attention to migration of legacy applications to the cloud, but cloud-to-cloud (i.e. C2C) migration/replication is rarely focused [8].

As mentioned, migration/replication can help cloud services to keep functioning in case of failure and respond to environmental changes on time. This increases the system reliability and scalability, two of non-functional requirements frequently described in the SLA in terms of service-level objectives (SLO). Migration and replication, migrates or replicates the critical software components, so that if one of components fails, the others can be used instead, or if one is not enough, others can be created to share workload. With a high level of component abstraction, these issues can be optimized by providing fine-grained granularity. Granularity refers to the unit of sharing in the cloud that can be an entire VM or a tiny file [9]. Granularity needs to be considered if multiple choices in moving or replicating a service in the specific conditions exist. We would not need to migrate or replicate an entire VM if partly migration or replication can solve the problems better.

The major contributions of this paper are as follows. First, we propose an autonomically fine-grained service migration and replication mechanism at component level, which is implemented on a multicloud distributed deployment framework. We also make a contribution in the development of a hierarchical DSL (domain-specific language) of the framework. The DSL helps not only to describe structure of component-based application naturally but also to demonstrate replicating/migrating rules in an intuitive and friendly way. Our first result was presented at the 2nd Nafosted Conference on Information and Computer Science [10]. Second, we complement our framework with the optimal deployment problem of replicated components. More specifically, we tackle the online optimization problem of component placement on multicloud regarding to the communication cost under constraints on system resources and a hierarchical structure of component-based applications. We formulate the placement problem as a quadratic program. Third, we validate our proposed mechanism by an experiment conducted in the context of an elastic scenario, and provide useful insights for cloud providers to decide if they should add servers or

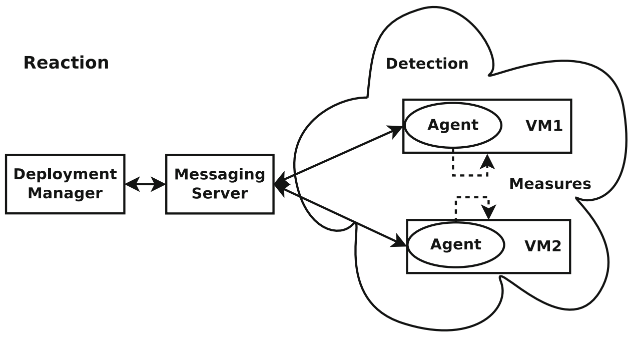


Fig. 1 Autonomic framework for migration and replication of cloud applications

upgrade server connections to improve the average responsive time of the system.

The rest of this paper is organized as follow: Sect. 2 gives an overview of architecture of the framework in the aspects of deployment manager, fine-grained hierarchical DSL and autonomic management. We introduce the migration and replication mechanism in Sect. 3. We present the formulation of the optimal deployment problem in Sect. 4. The validating experiments are presented in Sect. 5. Section 6 reviews the related work. Conclusions are stated in Sect. 7.

2 The autonomic framework

To support the autonomic migration and replication of cloud applications, we need a framework which satisfies the following requirements: (1) supports for describing components of cloud applications naturally at design time; (2) provides auto model parsing as well as dependencies and constraints resolving at runtime; (3) implements IaaS coordination which distributes service components into multiclouds, thereby avoiding the vendor lock-in problem, provides runtime execution as well; (4) advocates autonomic management to detect and respond to changes in runtime environments. Such a framework is depicted in Fig. 1, which is made up of several modules:

– The Deployment Manager (DM) is an application in charge of managing VMs and the agents (see below). It acts as a coordination interface to the set of VMs or devices on premises and on clouds. It is also in charge of instantiating VMs in the IaaS and physical machines such as embedded boards. It includes core modules of a MAPE-K autonomic model [11] such as Monitoring, Analysis, Planning, Execution and Knowledge exchange, which help the DM response to changes from surrounding environment.
– The Agent is a software component that must be pre-installed and bootstrapped on every VM and device which

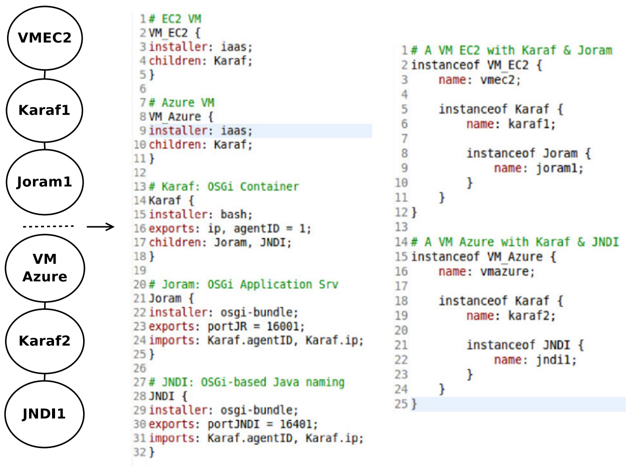
are managed. Agent probes the status of both hardware and software components and send these data to the DM periodically. These agents communicate with the DM and each other through an asynchronous messaging server.

– The Messaging Server is the key component acting as a distributed registry of import/export variables that enable communications between the DM and the agents and among agents themselves.

More details in these modules can be found in [12]. We have proposed and developed a deployment platform, which satisfies the requirements (1), (2) and (3) in [13]. In that paper, an open-source platform is designed to deploy complex distributed applications on multiclouds, which fosters deployment automation a step further by distributing virtual resources with pre-installed software (i.e. virtual appliances). It allows to describe distributed applications and handle deployment of the entire or a part of them. The platform is improvable and adaptable with a lightweight kernel which implements all necessary mechanisms to plug new behaviors for addressing new applications and a new execution environment. Moreover, the platform supports scaling and dynamic (re)configuration natively. This provides flexibility and allows elastic deployments.

With regards to SOA principle, the platform sees a complex application as a combination of "components" and "instances". While each component is a self-contained service that is homologous with the "object" definition, instances are obviously embodiment of these objects. In addition, the platform is designed to see an application also as a hierarchy of components. It means some components may be a containment to host or provide execution environment for other ones such as VM or container. All components derive from an abstract "root" component. Instances inherit all properties derived from its corresponding component. The main motivation of hierarchy is to keep track of exactly where instances were implemented in the system. It helps the system to make right decisions in autonomic deployment. The parent/children relationship of components is depicted naturally in an example about OSGi-based [14] application in Fig. 2. It is a cloud application providing Java message service (JMS) through its software components as Joram [15] and JNDI [16]. These components are containerized as bundles complying to the OSGi specification. A Karaf [17] component is also needed serving as OSGi container for the Joram and JNDI bundles.

In Fig. 2, there is an important field: children which lists other components that can be instantiated and deployed over this component. In the OSGi example, it means we can deploy Karaf over a VM instance. In turn, Joram and JNDI can be deployed over instances of Karaf. The hierarchical model resolves the containment relations (i.e. vertical relationship)



```
# EC2 VM
VM_EC2 {
installer: iaas;
children: Karaf;
}

# Azure VM
VM_Azure {
installer: iaas;
children: Karaf;
}

# Karaf: OSGi Container
Karaf {
installer: bash;
exports: ip, agentID = 1;
children: Joram, JNDI;
}

# Joram: OSGi Application Srv
Joram {
installer: osgi-bundle;
exports: portJR = 16001;
imports: Karaf.agentID, Karaf.ip;
}

# JNDI: OSGi-based Java naming
JNDI {
installer: osgi-bundle;
exports: portJNDI = 16401;
imports: Karaf.agentID, Karaf.ip;
}
```

```
# A VM EC2 with Karaf & Joram
instanceof VM_EC2 {
    name: vmec2;

    instanceof Karaf {
        name: karaf1;

        instanceof Joram {
            name: joram1;
        }
    }
}

# A VM Azure with Karaf & JNDI
instanceof VM_Azure {
    name: vmazure;

    instanceof Karaf {
        name: karaf2;

        instanceof JNDI {
            name: jndi1;
        }
    }
}
```

Fig. 2 An OSGi-based JMS service described by the framework's DSL (the graph in the middle and the initial instances description in the right)

amidst components at disparate layers. At runtime, the Graph and the Initial Instances Description are used to determine which components can be instantiated, and how they can be deployed. Abstracted components include the deployment roots (e.g. VMs, containers, devices, remote hosts), databases, application servers and application modules (e.g. WAR, ZIP, etc.). They list what the deployers want to deploy or possibly deploy. What is modeled in the graph is really a user choice. Various granularity can be described. It can either dig very deeply into the hierarchical description or bundle things together such as associating a given bundle with an OSGi container. Multi-IaaS is supported by defining several root components. Each one will be associated with various properties (e.g. IaaS provider, VM type, etc.). It is worth noting that an instance in a hierarchy can be located using an absolute path in the framework's DSL. For example, the "Joram1" instance can be referred to by the path "/vmec2/karaf1/joram1".

From these descriptions, the platform then takes the burden of launching the VMs, deploying software on them, resolving dependencies dynamically among software components, updating their configuration and starting the whole stuff when ready. The monitoring of each component after launching is also taken into consideration. Our continuous works to bring forward autonomic features to this platform, which satisfies the requirement (4), are discussed as our contributions in the next sections.

3 The autonomic replication and migration mechanism

We apply the autonomic management as an integrated feature of the platform in [13], which consists of two parts. On one side, agents retrieve measures on their local node. These measures are compared against some values given in the agent's configuration. If they exceed, equal or are lower than the given values, depending on the configuration rules, agents send a notification to the DM. On the other side, when the DM receives such a notification, it checks its configuration to determine which actions to undertake. These actions can range from a single log entry, to e-mail notification or even replicating a service on another machine. Figure 1 also sums up the way autonomic management works. While detection is delegated to the agents, reactions are managed by the DM.

The autonomic configuration is in fact defined in application projects. It means every project has its own rules and reactions. In this perspective, the project structure is enriched with a new directory, called autonomic. The autonomic directory expects two kinds of files. Measures files include a set of measures to perform by the agent. Such a file is associated with a given component in the graph. Hence, we can consider the autonomic rules as an annotation on a component in the graph. Rules files define the actions to undertake by the DM when a measure has reached a given limit or a particular condition has met. Such a file is associated with the whole application. Both types of files consume a part of the DSL language of the platform. This DSL part is specific to fine-grained migrating and replicating actions detailed as follows.

Measures files indicate measures an agent will have to perform regularly on its machine. An agent can use several options to measure something. The option or extension used to perform the measure is declared explicitly along with the measure name. Each measure is performed independently of the others. It means every measure matching the rule results in a message sent to the DM. The agent measures and notifies when needed and it has not to interpret these measures. This is responsibility of the autonomic modules of the DM. Here is the syntax for the declaration of a measure:

```
[EVENT measure-extension measure-name]
```

The measure-extension includes LiveStatus, REST and File. The LiveStatus [18], which is the protocol used by Nagios [19] and Shinken [20], allows to query a local Nagios or Shinken agent. We simply write a LiveStatus request:

```
# A simple query for Live Status.
[EVENT nagios myRuleName-80]
GET hosts
Columns: host_name accept_passive_checks
         acknowledged
Filter: accept_passive_checks = 1
```

An agent can query a REST service. The result can be interpreted as an integer or as a string.

```
# Check the result returned by a
# REST HTTP service.
[EVENT rest myRuleName-1]
Check http://google.fr THAT value > 0
```

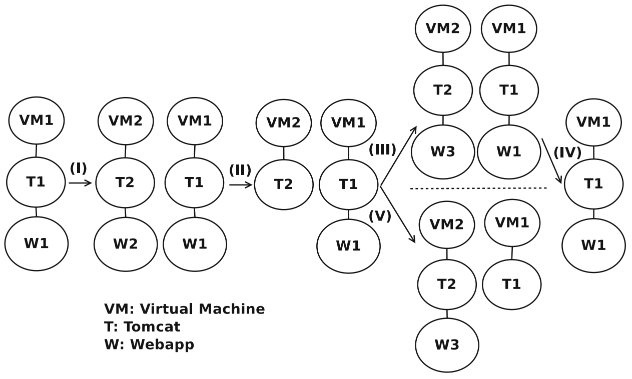


Fig. 3 Illustration of using five replicating/migrating reactions

An agent can also check the local file system. Depending on the existence of a file or a directory, or based on the absence of a given file, a notification will be sent to the DM.

```
# Notify the DM if a file exists
# and delete it.
[EVENT file myRuleName-1]
Delete if exists /opt/tmp
```

Rules files contain the reactions to undertake by the DM when a measure verified by a given rule on the agent side. These files use a custom syntax as following one.

```
[REACTION measure-name reaction-handler]
Optional parameters for the handler
```

There are four available handlers. Log is to log an entry without any parameters. Mail is to send an e-mail. It accepts only one parameter which is an e-mail address. Replicate-Service is to replicate a component on a new machine. It takes a chain of component names as parameters. Delete-Service is to undeploy and remove a component that was replicated. It takes a component name as parameter. To demonstrate for utility of the mentioned rules, we use an example about a J2EE application with three tiers including web (Apache), application (Tomcat) and database (MySQL) servers. The Apache uses "mod_jk" to provide load-balancing mechanism for Tomcat servers which host instances of a Webapp. The autonomic events which can affect to such system are going to be discussed carefully in Sect. 5. In this section, we describe five basic reactions usually used in autonomic replication/migration as follows. The illustration of these five reactions are shown in Fig. 3.

I To replicate the entire stack of "/VM1/Tomcat1/Webapp1" (all three instances):

```
[REACTION high-RT-1 Replicate-Service]
/VM1/Tomcat1/Webapp1 /VM2/Tomcat2/Webapp2
```

It is worth noting that if an empty VM2 already exists, it will be reused and "filled" with a new Tomcat2 containing a new Webapp2. Otherwise, a totally new entire stack will be created.

II To remove a specific instance Webapp2 of the stack "/VM2/Tomcat2/Webapp2", we need to provide the absolute path of this instance:

```
[REACTION low–RT–1 Delete–Service]
/VM2/Tomcat2/Webapp2
```

III To replicate a specific instance Webapp1 of the stack "/VM1/Tomcat1/Webapp1" to under the Tomcat2 (/VM2/Tomcat2/) and name it Webapp3:

```
[REACTION high–RT–2 Replicate–Service]
/VM1/Tomcat1/Webapp1 /VM2/Tomcat2/Webapp3
```

IV To remove the entire stack "/VM2/Tomcat2/Webapp3", we only need to provide the absolute path of the root instance which is VM2 in this case:

```
[REACTION low–RT–2 Delete–Service]
/VM2
```

It is also worth noting that the VM2 and its children (Tomcat2, Webapp3) will be gracefully stopped, undeployed and removed from the system orderly and automatically. In the case of migration, we combine both "Replicate-Service" and "Delete-Service" rules. For instance, after [III]:

V To migrate a specific instance Webapp1 of the stack "/VM1/Tomcat1/Webapp1" to under the Tomcat2 (/VM2/Tomcat2/) and name it Webapp3:

```
# Replicate the Webapp1 first
[REACTION low–RT Replicate–Service]
/VM1/Tomcat1/Webapp1 /VM2/Tomcat2/Webapp3
# Then remove the Webapp1
[REACTION low–RT Delete–Service]
/VM1/Tomcat1/Webapp1
```

These five basic reactions are reusable by any adaptive engines of any cloud platforms as long as they own modules supporting the description and distribution of fine-grained components.

4 Optimal deployment of components

The previous section introduces the mechanism of the autonomic replication and migration in which the agents notify the DM of measures collected on their local nodes, and the DM interprets these measures into actions. In this section, we study the optimization of DM's actions. More specifically, we determine an application server to which a component should be deployed to optimize the application performance under constrains on computing resources and the structure of components.

The DM decides the replication or undeployment of a component, depending on metrics on local nodes and the autonomic configuration defined in application projects. These changes in the deployment of components are represented by the current network of components of all applications and the new network of components. Let $P_1 = (C, G)$ be the current network of components of all applications where C is a set of components, and G is a set of links among components, called c-links. $g_{ij} \in G$ $(i, j \in C)$ is the c-link between component i and component j, where i and j are components of the same application and they exchange data if $g_{ij} = 1$, or not if $g_{ij} = 0$. Similarly, we denote the new network of components by $P_1' = (C', G')$.

We consider a network of server instances $P_2 = (S, E)$ where S is a set of servers and E is a set of links among server instances, called s-links. Several server instances can be deployed in the same physical server. $w_{ij} \in E$ $(i, j \in S)$ is the communication cost between server instances i and j. We denote by u_{cs} $(c \in C, s \in S)$ the current deployment state of P_1, where component c is deployed on server s if $u_{cs} = 1$, or not if $u_{cs} = 0$. Let $\tilde{C} = C' \backslash C$ be new components that the DM decides to replicate. Let d_i $(i \in S)$ be the total computing resource of server i. We define r to be an amount of computing resource required to deploy component i to a server. Let $x = (x_{cs})$ $(c \in \tilde{C}, s \in S)$ be a candidate solution of deployment of new components where component c is deployed on server s if $x_{cs} = 1$, or not if $x_{cs} = 0$.

The system performance of a deployment solution is measured by the total cost of the communication among new components $c_1 \in \tilde{C}$, the communication among components $c_2 \in C \cap C'$ that are components appearing in both the current deployment and the new one, and the communication between c_1 and c_2. Since the cost of the communication among c_2 in different solutions is similar, the cost of a deployment solution is given by

$$\varphi(x) = \sum_{c_1 \in \tilde{C}} \sum_{s_1 \in S} \sum_{s_2 \in S} x_{c_1 s_1} w_{s_1 s_2} \times \left(\sum_{c_2 \in \tilde{C}} x_{c_2 s_2} g'_{c_1 c_2} + \sum_{c_2 \in C \cap C'} u_{c_2 s_2} g_{c_1 c_2} \right) \quad (1)$$

The low cost of a deployment solution results in the improvement of average responsive time (ART). Therefore, the objective of the DM is to optimize the deployment so that the communication cost added by new components are minimized. Specifically, given the current network of components $P_1 = (C, G)$, the new network of components

$P_1' = (C', G')$, the network of server instances $P_2 = (S, E)$, the current deployment state (u_{cs}) where $c \in C$ and $s \in S$, find a deployment solution $x = (x_{cs})$ where $c \in C' \backslash C$ and $s \in S$ to minimize the communication cost subject to constraints on computing resources. This can be formulated as the following quadratic programming problem:

$$\begin{aligned} \text{Min} \quad & \varphi(x) \\ \text{s.t.} \quad & \sum_{c \in \tilde{C}} x_{cs} r_c + \sum_{c \in C \cap C'} u_{cs} r_c \le d_s \quad \forall s \in S && (2) \\ & \sum_{s \in S} x_{cs} = 1 \quad \forall c \in \tilde{C} && (3) \\ & x_{cs} \in \{0, 1\} \quad \forall c \in \tilde{C}, s \in S && (4) \end{aligned}$$

The optimization problem of service placement has been studied in several areas of future Internet, and the integer programming has been considered as a potential approach for solving the problem [21,22]. The complexity of the service placement problem is NP-hard [23]. A quadratic program can be solved by several optimization tools such as the CPLEX Optimizer [24]. In the following section, we are going to study the system performance of a deployment solution using the CPLEX Optimizer to solve the problem in various system configurations.

5 Evaluation

We first conduct an experiment to validate the proposed framework in context of an elasticity scenario. Then, we answer the question of whether we should add more server or upgrade server connection to improve the system performance.

5.1 Experiment setup

The elasticity context applies to the aforementioned J2EE application. With application tier, we use in initials two Tomcat servers dedicating to serve two different webapps: Webapp1 and Webapp2. We use the “mod_proxy” to build a cluster of Apache servers to avoid yet another bottleneck. Each of Apache server implements the “mod_jk” serving as a load balancer in front of these Tomcat ones. This experiment focuses on elasticity of application tier, thus without loss of generality, the database one is shared among webapps and hosted on a single MySQL server. All the VM used in this system are Microsoft Azure Standard_A2 instances with 2 cores and 3.5 GB memory. Each Tomcat created in the elastic reactions is a Amazon EC2 m3.medium with 1 core and 3.75 GB memory. The managed system is called System Under Test (SUT) that we use CLIF server [25], an distributed load injector, to create load profile and generate workload for the SUT to observe how the system reacts to changes of average response time (ART). These reactions are empowered by autonomic mechanisms aforementioned in Sect. 3. The topology of this scenario is depicted in Fig. 4.

5.2 Test scenario

The loads are injected into an entrance of the Apache cluster which is a virtual IP. Then this cluster distributes the loads to the corresponding webapps through the Tomcat servers. In this particular situation, the Webapp2 often gets low load, thus has a load profile as in Fig. 5 with 50 virtual users who try to send HTTP GET requests to the Webapp2 and then “think” a couple of time randomly. The virtual servers are threads created simultaneously by the CLIF server while the experiment was being performed according to the load profile of the Webapp2 (pre-defined also using CLIF server). Behaviors of the virtual users are captured from real-world operations using a capturing tool of the CLIF server.

The owner of the Webapp2 need not any elastic mechanisms provided by the cloud PaaS provider due to the low load of the Webapp2. On the other hand, the Webapp1 usually receives high load and thus has a load profile as in Fig. 5, which is also designed by the CLIF server. The Webapp1 usually takes the burden of about 450 virtual users who have similar behaviors as in the case of Webapp2. The owner of Webapp1 requires the cloud PaaS provider to ensure an acceptable performance for his webapp. Therefore, he demands an elastic load-balancing solution to guarantee an ART as low as possible as stated in an SLA established between him and the provider. When the ART varies, this solution includes provisioning a whole new Tomcat/Webapp1 server (reaction [I]) or replicating only the Webapp1 instances (reaction [III]) while scaling out as well as removing the servers (reaction [IV]) or migrating the webapps (reaction [V]) while scaling in with minimum side effects to overall system.

The polling periods is set to 10 s that means the ART of all requests from all users are collected each 10 s. These gathered data are sent to the framework’s analysis module to be aggregated and further analyzed. The analyzed information then is delivered to the planning module to generate new configuration for the system based on ECA (event-condition-action) rules.

The ECA rules decide whether the system should create an entire application server or only replicate a webapp instance using the set of reactions. One of the rules is to prevent multiple creating of new VMs or a new instance in a short period of time. At least the system needs to wait until it gets knowledge about the new one before another can be created automatically. It is called synchronization time which includes the VM provisioning, Tomcat installation, Webapp deployment and reconfiguration time for the existing Apache

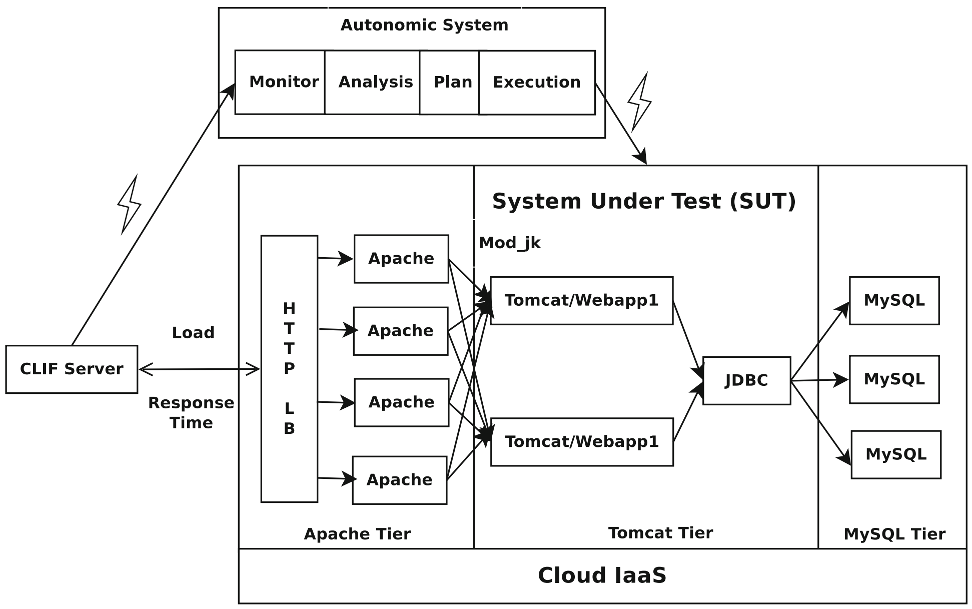


Fig. 4 Topology of J2EE test case using CLIF load injector

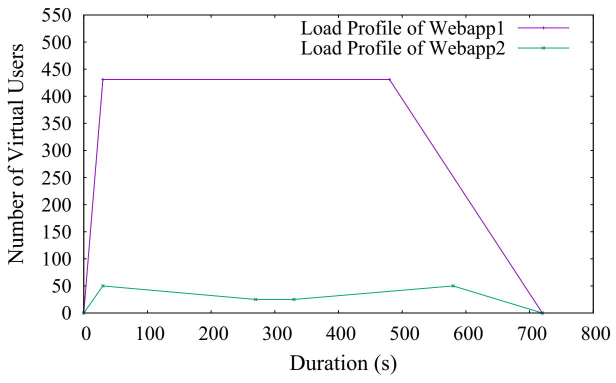


Fig. 5 CLIF load profiles of Webapp1 and Webapp2

(to know the attendance of the new Tomcat) in the case of creating a new Tomcat/Webapp server. With replicating or migrating a Webapp instance, the synchronization time only contains the two latter ones. Another rule is to prohibit the migration/replication of an instance to hosts where also are on-peak times. The very first 10-min snapshot of this experiment is shown in Fig. 6 and results are discussed deeply in the next section.

5.3 The efficiency of the fine-grained mechanism

Figure 6 shows the ART of both webapps and the corresponding reflections from the autonomic system to fluctuations of the response time. In addition, the figure also reports the changes in number of Tomcat servers while running the test case. The max response time of Webapp1 is set to 800 ms, it means if the ART goes over this limitation, creating new Tomcat server or replicating the Webapp request should be made. In contrast, if the ART goes under min response time (200 ms), a removing or migrating decision should be triggered.

We see that the ART of Webapp1 peaked at the 40th second because of aggressive accesses of the 450 virtual users simultaneously. At point “A1”, a command to create a new Tomcat server was triggered instead of a replication due to a peak (≈400 ms) happening in Webapp2. The max and min response times of Webapp2, which are not shown in Fig. 6, were set to 400 and 100 ms, respectively. After this request, the framework observed the SUT silently without any further requests until it gets knowledge about the new server. This synchronization time finished at the 180th second (point “A2”), thus the Webapp1 users continued experiencing slow accesses during 2 min 20 s more. At point “B1”, once again the ART of Webapp1 was larger than the max limitation whereas the ART of Webapp2 was getting low. It is suitable to make a replication Webapp1 (/VM1/Tomcat1/Webapp1) to under Tomcat2 and name it Webapp1_2 (/VM2/Tomcat2/Webapp1_2). The synchronization time for creating the Webapp1_2 was about 20 s which offered about 2 min better than the case of creating a new Tomcat server. Moreover, the system avoided creating a totally new server resulting in saving resources for PaaS provider and money for both PaaS user and provider. In reverse, the system performed two times of the scaling in: a request to remove a Tomcat3 server at “C1” (which had been created at “A2”) and a request to remove the Tomcat1 server at “D1” (which had been there from the beginning). The synchronization in both cases were almost the same (more or less than 20 s, finish at “C2” and “D2”) because we do not care about the shutting down time of a VM. In spite of that, the result of this elasticity is the saving of two VMs (from 3 VMs at “C1” to 1 VM at “D2”) while the system were in low-load period.

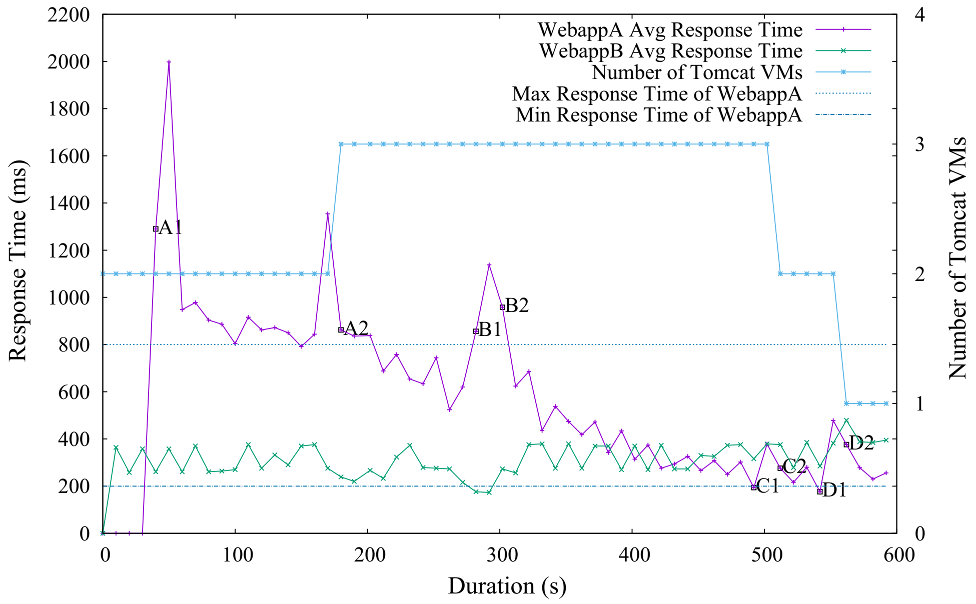


Fig. 6 Autonomic responses with fluctuation of average response time of Webapps

Because the changes in load of a website usually happen, applying the fine-grained migrating/replicating mechanism to an autonomic system brings significant performance improvement as well as cost saving.

5.4 The improvement on the system performance

We consider a baseline scenario in which a total of 100 components of 10 applications are deployed to 20 server instances. The computing resource of one server represented as a number of CPU cores, and the resource required to deploy a component to a server are uniformly distributed in [10, 100] and [1, 5], respectively. The communication cost among servers and the relation between components of an application are chosen uniformly in [1, 9] and [0, 1], respectively. A server is chosen uniformly in the set of servers whose resources are available for deploying a component.

We compare the performance cost of the system in three scenarios including the baseline scenario and two modified scenarios of the baseline scenario. In the first modified scenario, we add 4 servers to the baseline scenario. The resource of the new servers, the communication cost among new servers, and that between a new server and an old server are generated by the same rule described in the baseline scenario. In the second modified scenario, the communication cost between servers deceases by 1 in comparison with that in the baseline scenario. For the three scenarios, the number of new components replicated by the DM is varied between 5 and 45. We use the CPLEX Optimizer [24] to solve the optimal deployment of the new components in each setting, and compute the communication cost of the system. As one can see from Fig. 7, the communication cost of the system in a scenario of upgrading server connections is lower than that in a scenario of adding servers. This occurs because the

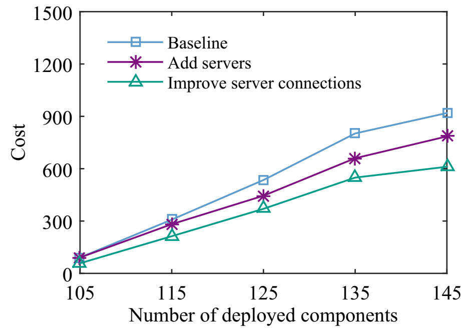


Fig. 7 The impact of adding servers and improving server connections

communication cost between two components is very small if they are deployed to the same server. The results suggest that it is better for a cloud provider to upgrade a server connection to improve the average responsive time of the system rather than add more servers.

5.5 The overhead of the framework

We evaluated the overhead introduced by the framework by doing experiments in two cases: (1) application deployed without the framework and (2) application deployed with the framework. The experiments are conducted on the VMAzure Standard A2 instances (2 cores and 3.5 GB memory). In the experiments, 1000 requests were generated. The requests sent were executed 20 times in both scenarios. Table 1 presents the results of the average execution time of each scenario as well as the additional costs introduced by the framework itself.

From the results presented in Table 1 , we can see that the overhead introduced by the framework is only 1.84 %. This

Table 1 Execution time and additional cost

Scenario	Average execution time	Overhead introduced by the framework
Application	10.85	–
Application with the framework	11.05	1.84 %

additional cost is generated mainly by the monitoring module which collects information for the elasticity autonomic mechanism. This module is integrated into the Deployment Manager. In summary, the overhead introduced by the framework is negligible given the discussed advantages.

6 Related work

Horizontal scaling action broadly supported by current cloud providers. They usually allow fixed-size VMs to be scaled out depending on current workload demand. On the contrary, vertical scaling, which obtains the elasticity by changing VM configuration (i.e. redimension), is offered scarcely by the providers. Even if redimension is supported, resizing VM resources on the fly is prohibited. GoGrid [26] allows its customers to increase RAM of VMs, but requires a VM reboot. Although Amazon EC2 introduces a wide range of VM instances with different sizes and configurations to simulate vertical scaling when needed (i.e. VM replacement or substitution), VM restart is still a must. The coarse-grained scaling with fixed-size VMs often leads to resource provisioning overhead resulting in the over-provisioning. Research on elastic VM of Rodero-Merino et al. [27] about VM substitution and Dawoud et al. [28] about fine-grained scaling to simulate the resizing have partly resolved this challenge. However, cloud providers is most likely prefer providing horizontal scaling with fatty VMs, thus research on combination of scaling actions on multiple levels of resource granularity as our work is really essential.

There have been several studies on the migration and replication of applications on cloud environments [29–33]. In [29], Ferrer et al. proposed the Optimis offering a chain of tools consisting of a Service Builder, an Administration Controller, a Deployment Engine, a Service Optimizer and a Cloud Optimizer. Service providers use these components to develop, deploy and execute applications on different clouds. Moreover, the SLA parameters are monitored and the services are migrated to another cloud if needed. Satzger et al. [30] developed the Meta-Cloud with existing standardizations that provides an API for web applications, recipes for migration/replication and deployment, resource templates for defining requirements and offerings, resource monitoring for checking QoS properties, etc. In [31], Reich et al. proposed a solution to migrate stateful Web Services on basis of SLA violations. The services are hosted by Web Service Resource Framework (WSRF) containers. These containers monitor the SLA parameters and detect violations. If an SLA condition is violated, a proper migration destination is searched over the P2P network for a service picked randomly by a particular container. The SLAs are defined for each service using WS-Agreement, an SLA description language. However, these solutions lack a DSL to abstract the complex services.

So far, there has been little discussion about the use of abstracted representations of component-based applications for the migration and replication of applications on cloud environments. In [32], Hao et al. introduced a General Computing Platform (GCP) which hosts different kinds of services, from infrastructure to application services. This system uses a workflow model for service composition and a cost model for making migration decision. The system is similar to our approach because services used in the workflow model are also the abstracted representations of real ones and then the abstracted ones are replaced with concrete instances. However, the hierarchical description of multi-tier applications is not feasible with this system. Our work is different as it proposes a DSL to describe a hierarchical structure of component-based application.

A model-driven approach for the design and execution of applications on multiple clouds is proposed by Ardagna et al. [33]. For the abstracted clouds, the code is semi-automatically translated and applications have to be implemented only one time. Then the best cloud for a service is selected regarding non-functional criteria such as the costs, the risks by a DSS (decision support system). Eventually, the dynamic migration/replication of services between clouds is provided by a runtime management API. However, with this approach, the autonomous factor is not taken into account. Our proposed framework provides an autonomic management mechanism in which the decision of migration and replication is controlled by the deployment manager responding to rapid fluctuations of demands. In addition, no research has been found that surveyed the online optimization problem of component placement on multicloud regarding the communication cost under constraints on system resources and a hierarchical structure of component-based applications. Most studies on the placement problem in cloud computing have been carried out in either the placement of virtual machines or an application component without considering the online solution with regard to both a structure of component-based applications and constraints on cloud infrastructure [34–37]. Our study was designed to fill the gap.

7 Conclusion

In the era of cloud computing, the trend which enterprises choose to deploy their software on hybrid and multicloud is indispensable. The flexible choice among cloud providers helps enterprises to save the cost due to which they can select the best services to install different software parts. This leads to demands of service migration and replication across the clouds. Our proposed framework supports service migration/replication mechanism to fulfill the need at component level. It allows deployers to describe software using a hierarchical DSL, deploy it using an implementation of a multicloud distributed deployment framework, and provide autonomic rules to respond to fluctuations of environment, thereby ensuring availability and scalability. In our framework, we also address the optimal deployment problem of components replicated by the DM. While the cost of copying the whole virtual machines is too high due to their big virtualization overhead, the fine-grained service replication/migration at component level is a possible solution. The experiments were conducted to prove the advantage of our partial approach in comparison to full migration and replication of an entire server stack. The numerical results also provide a suggestion of improving the system performance for a cloud provider.

Although componentized implementation improves scalability, our framework requires an application to follow a service-oriented architecture. It will be a future direction to study an application re-hosting pattern to integrate a monolithic legacy application with service-based applications in the framework for improving consistency and reducing cost through consolidation and sharing across cloud environments. Other possible future directions include the more detailed analysis of system performance in a large deployment scenario taking into account elastic demands and network fluctuations, or a deep analysis of comparison between different replication and migration approaches.

Acknowledgements This work was partially supported by Microsoft Research Europe, and project B2016-SPH-17 from the Vietnam Ministry of Education and Training.

References

1. Final version of NIST cloud computing definition published. http://www.nist.gov/itl/csd/cloud-102511.cfm. Visited on March 2015
2. Thanh, D.V., Jrstad, I.: A service-oriented architecture framework for mobile services. In: Proceedings of the advanced industrial conference on telecommunications/service assurance with partial and intermittent resources conference/e-learning on telecommunications workshop, pp. 65–70 (2005)
3. AppZero. https://www.appzero.com. Visited on March 2015
4. CliQr. http://www.cliqr.com/platform/. Visited on March 2015
5. Racemi. http://www.racemi.com. Visited on March 2015
6. CohesiveFT. https://cohesive.net. Visited on March 2015
7. Ravello. http://www.ravellosystems.com. Visited on March 2015
8. Lloyd, W., Pallickara, S., David, O., Lyon, J., Arabi, M., Rojas, K.: Migration of multi-tier applications to infrastructure-as-a-service clouds: An investigation using kernel-based virtual machines. In: Proceedings of the 12th IEEE/ACM international conference on grid computing (GRID), pp. 137–144 (2011)
9. Vardhan, M., Yadav, D., Kushwaha, D.: A transparent service replication mechanism for clouds. In: Proceedings of the sixth international conference on complex, intelligent and software intensive systems (CISIS), pp. 389–394 (2012)
10. Pham, L.M., Pham, T.M.: Autonomic fine-grained migration and replication of component-based applications across multi-clouds. In: Proceedings of the 2nd Nafosted conference on information and computer science (NICS), pp. 5–10 (2015)
11. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing - degrees, models, and applications. ACM Comput. Surv. **40**(3), 7:1–7:28 (2008)
12. Pham, L.M., Tchana, A., Donsez, D., Zurczak, V., Gibello, P.Y., de Palma, N.: An adaptable framework to deploy complex applications onto multi-cloud platforms. In: Proceedings of the IEEE RIVF international conference on computing communication technologies - Research, Innovation, and vision for the future (RIVF), pp. 169–174 (2015)
13. Pham, M.L.: Roboconf : an autonomic platform supporting multi-level fine-grained elasticity of complex applications on the cloud. Theses, Université Grenoble Alpes (2016). https://tel.archives-ouvertes.fr/tel-01312775
14. OSGI. http://www.osgi.org/Main/HomePage. Visited on March 2015
15. JORAM: Java (TM) Open reliable asynchronous messaging. http://joram.ow2.org. Visited on June 2016
16. Java Naming and Directory Interface (JNDI). http://docs.oracle.com/javase/8/docs/technotes/guides/jndi/index.html. Visited on June 2016
17. Apache Karaf. http://karaf.apache.org/. Visited on June 2016
18. Livestatus. https://mathias-kettner.de/checkmk_livestatus.html. Visited on March 2015
19. Nagios. http://www.nagios.org. Visited on March 2015
20. Shinken. http://shinken-monitoring.org. Visited on March 2015
21. Pham, T.M., Fdida, S.: DTN support for news dissemination in an urban area. Comput. Netw. **56**(9), 2276–2291 (2012)
22. Pham, T.M., Minoux, M., Fdida, S., Pilarski, M.: Optimization of content caching in content-centric networks. Tech. Rep. hal-01016470, UPMC Sorbonne Universités (2014). http://hal.upmc.fr/hal-01016470/en/
23. Baev, I., Rajaraman, R., Swamy, C.: Approximation algorithms for data placement problems. SIAM J. Comput. **38**(4), 1411–1429 (2008)
24. IBM ILOG CPLEX Optimizer. http://www.ibm.com/software/integration/optimization/cplex-optimizer/. Visited on November 2015
25. CLIF server. http://clif.ow2.org. Visited on March 2015
26. GoGrid. http://www.gogrid.com/. Visited on April 2016
27. Rodero-Merino, L., Vaquero, L.M., Gil, V., Galán, F., Fontán, J., Montero, R.S., Llorente, I.M.: From infrastructure delivery to service management in clouds. Futur. Gen. Comput. Syst. **26**(8), 1226–1240 (2010)

28. Dawoud, W., Takouna, I., Meinel, C.: Elastic vm for cloud resources provisioning optimization. In: Proceedings of the first international conference on advances in computing and communications, pp. 431–445 (2011)
29. Ferrer, A.J., Hernández, F., Tordsson, J., Elmroth, E., Ali-Eldin, A., Zsigri, C., Sirvent, R., Guitart, J., Badia, R.M., Djemame, K., Ziegler, W., Dimitrakos, T., Nair, S.K., Kousiouris, G., Konstanteli, K., Varvarigou, T., Hudzia, B., Kipp, A., Wesner, S., Corrales, M., Forgó, N., Sharif, T., Sheridan, C.: Optimis: A holistic approach to cloud service provisioning. Futur. Gener. Comput. Syst. **28**(1), 66–77 (2012)
30. Satzger, B., Hummer, W., Inzinger, C., Leitner, P., Dustdar, S.: Winds of change: from vendor lock-in to the meta cloud. IEEE Intern. Comput. **17**(1), 69–73 (2013)
31. Reich, C., Bubendorfer, K., Banholzer, M., Buyya, R.: A SLA-oriented management of containers for hosting stateful web services. In: Proceedings of the IEEE international conference on e-science and grid computing, pp. 85–92 (2007)
32. Hao, W., Yen, I.L., Thuraisingham, B.: Dynamic service and data migration in the clouds. In: Proceedings of the 33rd annual IEEE international computer software and applications conference (COMPSAC)., vol. 2, pp. 134–139 (2009)
33. Ardagna, D., di Nitto, E., Mohagheghi, P., Mosser, S., Ballagny, C., D'Andria, F., Casale, G., Matthews, P., Nechifor, C.S., Petcu, D., Gericke, A., Sheridan, C.: Modaclouds: A model-driven approach for the design and execution of applications on multiple clouds. In: Proceedings of the 4th international workshop on modeling in software engineering (MiSE), pp. 50–56 (2012)
34. Dong, J., Jin, X., Wang, H., Li, Y., Zhang, P., Cheng, S.: Energy-saving virtual machine placement in cloud data centers. In: Proceedings of the 13th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid), pp. 618–624 (2013)
35. Li, K., Zheng, H., Wu, J.: Migration-based virtual machine placement in cloud systems. In: Proceedings of the 2nd international conference on cloud networking (CloudNet), pp. 83–90. IEEE (2013)
36. Zhu, X., Santos, C., Beyer, D., Ward, J., Singhal, S.: Automated application component placement in data centers using mathematical programming. Int. J. Netw. Manag. **18**(6), 467–483 (2008)
37. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. J. Intern. Serv. Appl. **1**(1), 7–18 (2010)