Towards a Deployment System for Cloud Applications

Ruici Luo[1,3], Wei Ye[2,3*], and Shikun Zhang[2,3]

[1]School of Electronics Engineering and Computer Science, Peking University, China
[2]National Engineering Research Center for Software Engineering, Peking University, China
[3]Key Laboratory of High Confidence Software Technologies, Ministry of Education
{*luoruici,wye,zhangsk*}*@pku.edu.cn*

Abstract

A sophisticated deployment system plays an important role in automating and improving the process of software delivery, especially for cloud applications. Since cloud applications usually consist of many components run on different virtual machines, i.e., EC2 instances, the deployment is time-consuming and error-prone, which may involves manual operations and complex scripts. We develop a deployment system aiming to accelerate cloud application delivery. First of all, we propose a component model and a connector model involving cloud feature. Then we present a component management system, in which component can be configured and instantiated rapidly based component inheritance and composition. Finally, we develop a novel deployment mechanism that can automate deployment process across multiple cloud instances. Experiment shows that our approach can reduce the build time and downtime so that it can speed up the delivery process of software application.

Keywords: software architecture; application deployment; cloud computing; continuous delivery

1. Introduction

For enterprise applications, continuous integration is increasingly seen as an effective tool for reducing the cycle time from product backlog to receiving actual user feedback. This can result in real increases in developer and team productivity when combined with cloud computing. One key trend that is growing in importance daily is Continuous Delivery[13]. More and more organizations are looking to embrace an agile model in which stringent, automated testing allows enhancements or "micro-releases" to go live without the traditional waterfall release cycles. We are seeing a major shift in enterprise software development to cloud-based, continuous delivery, with fully automated quality, coverage, functional and performance tests gating live deployments. Thus, incremental deployment become more critical since it is very expensive to rebuild and redeploy the whole application. Meanwhile, cloud applications usually consist of many components run on different virtual machines making deployment time-consuming and error-prone, which may involves manual operations and complex scripts. It has become a common issue to reduce the build time and downtime so that it can speed up the delivery process of cloud application.

In the past two decades, application servers have been designed to serve multiple applications, which means applications share identical runtime and deployment scenario. With the advent of Cloud Computing the IT resources can be rapidly and elastically delivered via Internet. Examples of compute Clouds are Amazons Elastic Compute Cloud (EC2) and Google App Engine (GAE). Meanwhile, the complexity of applications and servers grows rapidly. Nowadays, the benefits of making the server immutable[14] becomes more obvious and clear. If anything of an application has been changed, a new immutable server instance will be made next to the existing one, which will be destroyed soon. One can request and release resources in a few seconds with low costs, e.g. creating instances in EC2 and running applications in GAE. In this context, back-end infrastructure including middleware container has become one integral part of one specific application. LXC (Linux Containers) is an operating system level virtualization method for running multiple isolated Linux systems (containers) on a single control host, instead of creating a full-fledged virtual

*Corresponding Author

(DOI reference number: 10.18293/SEKE2015-192)

machine. We see that it has become feasible in cloud environment to make application deployment and runtime immutable with virtualization and lightweight container technology like Linux Container. If each component has its own infrastructure and isolated runtime, the cost of rebuild and redeploy a single component will be relatively low. By imposing a well-organized connectivity and lightweight communication mechanism between application components, breaking down partial components will not terminate the whole application. We could leverage the power of cloud infrastructure to support individual evolution, ensuring good replaceability and upgradeability of components in software system to achieve the goal of continuous delivery.

Application management is a key issue for successful continuous delivery. Developer need to take care of evolution of component configuration. Current solution for application management dedicated to IaaS(Infrastructure as a Service) and PaaS(Platform as a Service). Many configuration management systems such as Puppet[5], Chef[2], which provide a DSL to model a virtual machine instance, including files to present and application stack that should be running. These configuration management systems manage the configuration of applications in a centralized server and the work of deployment is assigned to operation teams, not developers. Although virtual machine[9] based on IaaS platform is a solution to deploy application, we also need a simple and lightweight way to manage the deployment of applications.

1.1. Contributions

This paper makes the following contributions:

- We present an component model from deployment perspective for accelerating continuous delivery process. In our approach, components consist of business function code, configuration options and runtime software stack(OS, middleware, dependencies library) definition which is called image. A image can be instantiated to a instance.
- We also provide a system to manage the images and instances. An inheritance mechanism ensures that each component can evolve independently and reused in many situations. Unlike all previous techniques and systems of which we are aware, our approach makes components immutable. If any changes occurs, a new image of the specific component is generated and instantiated to replace the old instance. Also the evolution process is recorded, i.e. the history of images is maintained by our deployment system. It is easy to rollback to any state of application.
- We present an system that automates the deployment of distributed application based on this model in the cloud. The deployment system processes the instantiation and resolve the interconnections of components.

The rest of this paper is organized as follows. Section 2 presents a motivating example that illustrates how our approach works. Section 3 describes a component-based model and an application management system for cloud application. Section 4 describes an automated deployment system. Section 5 gives an example in practice to evaluate our approach. Section 6 discusses related works. We conclude in Section 7.

2. Component Model

To meet the challenges that applications in cloud should be highly scalable and flexible, we propose a component model as an abstraction of deployment elements. From the business function perspective, following the object-oriented principles of "Single Responsibility" and "Concerns Separation", a component should focus on a small, single and independent business function that it is responsible for. So parallel development can be organized straightforwardly and incremental evolution could be performed smoothly. From the infrastructure perspective, application changes are not only about application itself. Changes that is about environment should also be considered as part of evolution. Dynamic components interchangeability should be supported. So adding or updating components will not require redeployment of the entire application.Using the component model to abstract the target deploying application, we are aiming to following features:

- Self-contained infrastructure: Each Component contains the infrastructure required at the runtime. The infrastructure includes but is not limited to middleware and OS environments. This ensures the isolation of components and also improves the reliability since they do not affect each other.
- Individual Evolution: For the reason that each component has its own infrastructure and isolated to each other, they can be developed and maintained independently at the time. This can greatly improve the productivity of software and reduce the maintenance cost.
- Disposable Components: In cloud environment, virtualized instance can be acquired cheaply and pirated. The past method that deploying components into middleware or operating system is out of time.

We propose a component model, which is an extension of BU(Business Unit) model from BuOA[15]. BU contains presentation layer, business logic layer and data accessing layer inside. BU also provides attributes, operations and

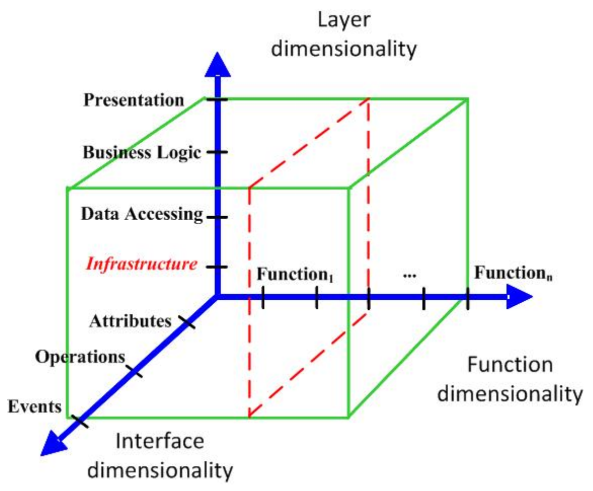

Figure 1. Extending infrastructure layer of BU

events as its external interface. Basically, attributes can be treated as representations of BUs internal state; operations provide ways to query and change its internal state; and events will indicate the changing of its internal state. However, BU only concerns abstraction in application logic level, lacking mechanism to support cloud features. Thus, we add infrastructure layer in Business Unit model, which includes middleware and OS environment. The extended model is illustrated in Figure 1.

The component model has two levels, business level and infrastructure level. Business level is not always necessary. A image with a MySql database can be seen as having only infrastructure level. Componentonnectivity between images can be categorized into three categories base on the Two-Level perspective.

Business dependency This kind of relationship can be further divided into four categories: observing, injecting, weaving and binding as well as BuOA. We will not cover them in detail here.

Infrastructure dependency: This kind of relationship defines the dependencies of infrastructure, e.g. application server has dependencies of cache and database server.

Data sharing Another way to connect two components is to share data between them. An example is that multiple business components use a same database server.

3. Component Management

To support maintaince and evolution of components, we present a system to manage components of cloud applications.

3.1. Image

Image is subject to describe the state of a component, including hardware characteristics, software stack(OS, middleware, etc...) and applicative binaries. Each image has a unique identifier(usually generated by system via hash algorithms). There are many images generated in the evolution process of a component. So we aggregate these images to a *repository*. In analogy with Git version control system, a *image* is a commit and a *repository* is correspond to the same name. A *repository* potentially holds multiple variants of an image. In the case of our ubuntu image we can see multiple variants covering Ubuntu 10.04, 12.04, 12.10, 13.04, 13.10 and 14.04. Each variant is identified by a tag and you can refer to a tagged image like ”ubuntu:14.04”.

When developers decide to publish components for test or release, the management system just generate a new image with a unique identifier via the configuration about OS, middleware and applicative binaries and it is pushed to the central registry. The deployment system and other developers can pull this image and get an instance of it after pushing.

3.2. Instance

An *instance* is an instantiation of a *image* by assigning concrete values to configuration, and it is deployed to the IaaS platform as the runtime of a component. An *instance* consists of software stacks and applicative binaries. An instance has a global unique identifier as well as images. The identifier could distinguish two instances instantiated by one image.

3.3. Image Inheritance

To extend and reuse images for productivity, the image could be inherited. For example, each portion of a web application(web server, application server, cache, database, etc...) is running in a Ubuntu linux operating system. So we can make an ”abstract”(which could also be instantiated) image for inheritance.

To extend base image, developers can add or override dependencies and configurations to generate new images which means that the configuration and the dependencies are all inherited from a base image. On the other hand, images with different versions of a same name should be co-located in one repository.

3.4. Image Composition

Another way to extend and reuse existing images is composition. For example, if we need a image that consists of

Java runtime environment and mysql database and there exist independent images of JRE and mysql. We can compose them and get a new image that has the java and mysql features.

However, it is not appropriate to merge images in some cases. If an image is based on ”ubuntu” and another image is based on ”windows”, they can not be merged apparently. So before merge, we would check if the images has the same ancestor in the image tree, and then check if there are any conflicts between them. After composition the images tree becomes a Directed Acyclic Graph(DAG).

4. Application Deployment

4.1. Overview

To deploy the application to IaaS platform, we present a deployment system which takes a deploy plan as following configuration written by YAML:

```
web:
  image: onboard-core:1.2
  ports:
    - 8080
  volumes:
    - .:/code
  links:
    - redis
redis:
  image: redis:latest
  command: redis-server --appendonly yes
```

This defines two components:

- **web**, which is built from an image called onboard-core with a version number 1.2. It also says to expose 8080 port, connect up the redis component and mount volumes for data sharing.
- **redis** which uses the redis image with latest version directly.

Each element on the top of the YAML file describes a component. It specifies the name and version of image and the dependencies to other component.

4.2. Disposable Distribution

As we mentioned above, to avoid the issue that infrastructure has been patched again and again, we make the infrastructure as part of application distribution. This means that any changes to the infrastructure is equivalent to the application. In our new situation, we absolutely know a system has been created via automation and never changed since the moment of creation. A distribution of application is never modified after deployed, and merely thrown away after being replaced with a new distribution.

Another consideration is that the data related to an application is not immutable and cannot be thrown away. A practical way is shipping the data storage off of the BU distribution. Technically, sending log files to a central system log server, using shared file system like NFS, choosing mountable cloud service as storage devices are all feasible practice to guarantee data integrity.

4.3. Individual Evolution and Development

Incremental deployment is critical in the software evolution since it is very expensive to rebuild and redeploy the whole application. As we separate application into components each of which has its own infrastructure and isolated runtime, the cost of rebuild and redeploy a single component is relatively low. Due to the lightweight communication mechanism between components, breaking down partial BUs will not terminate the application. The individual evolution can also ensure good replaceability and upgradeability of components in software system.

To keep things simple, we consider two inter-related components in an application. One component requires services provide by another component and they are developed in parallel. As the developer(s) of each component, they do changes everyday with building and releasing SNAPSHOT version of distribution. So the developer(s) of the component that requires services of another does not need to get the source code and build another component, he/she/they only have to pull the SNAPSHOT of distribution and run it locally. This greatly reduce the time cost of dependencies building, testing and configuration. In summary, the collaboration mechanism between components varies from source code level to component with infrastructure level and will give a huge boost to improve the quality, reliability and productivity of software application.

5. Implementation and Evaluation

5.1. Component Implementation

We have implemented a prototype to verify our method and evaluate its performance. We use Docker[3] to implement the infrastructure level of components. The implementation is based on Spring Boot[6]. Spring Boot provides the ability to create stand-alone Spring based enterprise application that embeds an application as the middleware and can be run by itself. The prototype was tested on CentOS 6.4 and can also be applied or extended to the OS that supports Docker.

Application Logic Level of an image is the same as the architecture proposed in BuOA. In the example the component projectMg contains three bundles projectMgt.persistence, projectMgt.service, projectMgt.web, corresponding data accessing layer, business logic layer and presentation layer respectively. They communicate with each other based on contracted service interfaces, shielding implementation details completely. For example, data accessing bundles can choose different Object-Relation mapping frameworks to do the persistence work as long as it keeps the data accessing interface unchanged.

Infrastructure Level of an image is implemented as a Docker container which contains a Spring Boot based application. The Docker container is a virtualized and isolated operating system, and the Spring Boot based application is embedded application server like Tomcat or Jetty. It is a stand-alone application which means no external server is required. To describe the Docker container, we add a Dockerfile to each BU. The Docker container is created via Dockerfile with application build. The following text file is an example description of infrastructure level of a image:

```
#Inherit from a built container
#with Java environment.
From komljen/jdk6-oracle
#Get the latest version of Maven
Run apt-get update
Run apt-get install -y maven
Run mvn clean install
#Startup the application
Cmd java -jar target/sample-1.0.0.jar
```

5.2. Evaluation

Onboard[4] is an actual application which continuous runs for about 2 year. To begin with, we develop with the BuOA approach. All BUs are put into a virgo instance. At that time, each BU only contains application code and is not isolated to each other. Any little change of a BU will cause the application to restart. More seriously, bugs in a BU cause the JVM process down and the application halts. The 10 components run in their own Docker container and are isolated to each other. There is one JVM process running as the runtime of each component. For the evolution perspective, each BU has its own individual evolutionary process.

The evaluation is based on the build time and downtime of each release. In the old architecture, the calculation is very easy because each build is related to the entire application and the downtime is the restart seconds of the application server. We have an continuous integration server to do daily release of application. We collect the logs from servers and make a table to show the data below (The data is up to June 2014).

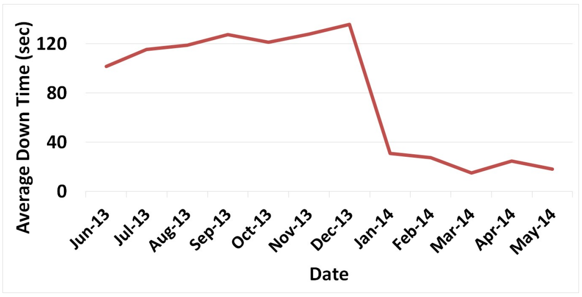

Figure 2. Downtime costs with the evolution of Onboard

Month	LOC	Change Times
July 2013	231137	4
Aug 2013	231251	5
Sep 2013	228879	3
Oct 2013	211297	4
Nov 2013	212132	6
Dec 2013	234853	4
The refactor separator		
Jan 2014	232268	16
Feb 2014	239247	19
Mar 2014	241317	20

Before June 2013, the application scale was relatively small, hence, newer data are shown in table above to keep our test in a consistent way. As we can see, the change times increased very fast when do the isolation of BUs. The reason is that each BU has its own evolution and the times is added by each of them. Developing with the new approach, the iteration has a much higher frequency.

The most important factor that affect the build time and downtime is the increment. With our new approach, increment of each release is quite small because upgrade a small part of BUs will not affect the status of other BUs. The test results are shown in Figure 2 and Figure 3. The data in these figures confirms that our new approach can reduce the build time and downtime so that it can speed up the delivery process of software application.

6. Related Works

The problem of deployment of application has attracted significant attention in the area of System Administration. Many tools exist: Puppet[5], Chef[2], CFEngine[1]. The goal of these systems is to simplify the management task of large scale machines. However, they only consider the configuration of systems or environments and not take the configuration and interconnections of components in application.

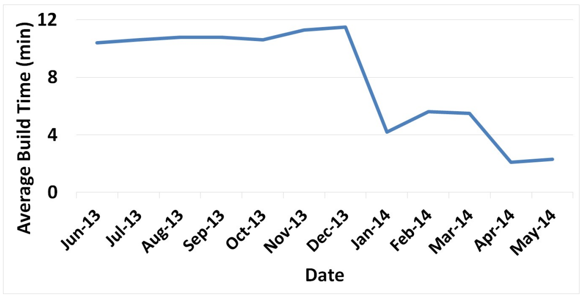

Figure 3. Build time costs with the evolution of Onboard

Aeolus[7] component model is specifically designed to capture realistic scenarios arising when configuring and deploying distributed applications in cloud environments. It is able to describe several component characteristics such as dependencies, conflicts, non-functional requirements. The Blender[10] toolchain extends [8] that automates the assembly and deployment of complex component-based software systems. By relying on a configuration optimizer and a deployment planner, the final deployment satisfies not only user requirements but also to be optimal with respect to the number of used virtual machines.

Engage[11] is a deployment management system. Throughout the paper the term *resource* is used as a synonym of component. *Resource* consists of *type* and *driver*. The former statically verifies deployment properties and generates the deployment plan, while the latter installs and manages the resource's lifecycle. Engage introduces three types of dependencies: **Inside** for nesting(e.g. application code runs into an application server); **Env** for local dependencies(Java programs need JRE); **Peer** for resources deployed anywhere else. The present paper has a similar idea to Engage: It separates the specification and runtime of components and automated generates the right order of deployment.

SmartFog[12] is a Java framework to manage deployment for distributed applications. It shares some concepts with the Engage that each component has a declarative description and a driver called lifecycle manager.

7. Conclusion

To address the critical challenge of deploying distributed application in the cloud, we present an component-based model that aims to automated configure and deploy over lightweight container. Base on the proposed model, we introduce an application management system as well as an inheritance-based mechanism that ensures each resource can be evolved independently and reused in different scenarios. We also present a deployment system that could process the dependencies and interconnected relationship of components automatically. To evaluate our approach, we implement a distributed application on an industrial IaaS platform. As a result, we can decompose a cloud application vertically into independent and cohesive modules which has dynamic interchangeability and evolvability.

Acknowledgments: This work was supported by the National Natural Science Foundation of China under Grant No.61202070.

References

[1] Cfengine. http://cfengine.com/. Accessed: 2015-05-10.
[2] Chef. https://www.chef.io/. Accessed: 2015-05-10.
[3] Docker. https://docker.com/. Accessed: 2015-05-10.
[4] Onboard. https://onboard.cn/. Accessed: 2015-05-10.
[5] Puppet. https://puppetlabs.com/. Accessed: 2015-05-10.
[6] Spring boot. http://projects.spring.io/spring-boot/. Accessed: 2015-05-10.
[7] M. Catan, R. D. Cosmo, A. Eiche, T. A. Lascu, M. Lienhardt, J. Mauro, R. Treinen, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Aeolus: Mastering the Complexity of Cloud Application Deployment. In K.-K. Lau, W. Lamersdorf, and E. Pimentel, editors, *ESOCC - European Conference on Service-Oriented and Cloud Computing - 2013*, volume 8135, pages 1–3, Malaga, Spain, 2013. Springer.
[8] R. Di Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, and A. Agahi. Automated synthesis and deployment of cloud applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 211–222, New York, NY, USA, 2014. ACM.
[9] X. Etchevers, T. Coupaye, F. Boyer, and N. de Palma. Self-configuration of distributed applications in the cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 668–675, July 2011.
[10] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. De Palma. Reliable self-deployment of cloud applications. In *SAC 2014 - 29th ACM Symposium on Applied Computing*, Gyeongju, South Korea, Mar. 2014.
[11] J. Fischer, R. Majumdar, and S. Esmaeilsabzali. Engage: A deployment management system. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 263–274, New York, NY, USA, 2012. ACM.
[12] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft. The smartfrog configuration management framework. *ACM SIGOPS Operating Systems Review*, 43(1):16–25, 2009.
[13] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
[14] K. Morris. Immutable server, June 2013.
[15] W. Ye, R. Luo, S. Zhang, X. Liu, and W. Hu. Buoa: An achitecture style for modular web applications. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 802–807. IEEE, 2012.