Hot Deployment with Dependency Reconstruction

Haicheng Li, Chun Cao, Xianping Tao
State Key Lab for Novel Software Technology, Nanjing University, Nanjing, China
Dept. of Computer Science and Technology, Nanjing University, Nanjing, China
lhc_happy@sina.cn, {caochun, txp}@nju.edu.cn

***Abstract*—Hot deployment is a typical feature in mainstream application servers. But current application servers treat each module as a standalone application and may fail if a module with dependencies against other ones is partially updated with hot deploying. The reason lies in that those module dependencies are not respected in current application servers. Direct countermeasures that manage dependencies in application servers are actually inefficient or even infeasible. So in this paper, we propose an approach that automatically constructs the module dependencies with class loading mechanism which further helps to reconstruct the modular application respecting the dependencies upon hot deploying. Experiments show that our technology of hot deployment can ensure partial update of the modular applications correctly and efficiently.**

Keywords*—*Hot deployment, Dependency reconstruction, Class loading

I. Introduction

Nowadays application servers provide platforms for enterprise applications to be deployed, operated and maintained. They usually contain comprehensive services, such as clustering, security and transaction management, so that developers can focus on the business logic. As one of those sophisticated services, hot deployment enables the server to put applications into production without restarting the server itself. By hot deployment, the existing application can be upgraded in this fashion as well.

The technology of hot deployment highly improves the efficiency and flexibility of the application servers. It becomes one of the typical features of mainstream application server products, such as JBoss [1] and WebLogic [2], which play a significant role in the development of enterprise applications. By the philosophy of design of the application servers, the units to be deployed, in packages (or modules) of ear/war/jar, are standalone applications, which means they are closed and self-contained. This implicit assumption does not prohibit the developers to build the applications that span multiple packages with internal dependencies. Applications in this case can still work as long as the referencing across different deployment units are supported. As a matter of fact, the unit deploying mechanisms built in the application servers like JBoss do make this happen with appropriate class loader designed. Application servers ensure that modules are deployed after the deployment of modules which it depends on. The dependencies are constructed in the context of deployment unit during the first time deployment.

Applications in multiple packages, which are called as *modular applications*, are common when they are developed by different individuals collaboratively or assembled with third-party libraries that have been already packed independently. They still work fine on application servers until some portion of the application is about to be hot deployed with the new versions. As the application server treats the packages as isolated ones, it ignores the existence of potential relationships among the modules. Failures may happen because the original dependencies between the portion to be hot deployed and the rest of the application are broken, which makes the whole application down.

To solve the problem, the dependencies need to be reconstructed. Intuitively, this can be done manually. The developer should have the knowledge of dependent configuration among the application modules and be aware of the affection scope of the redeployment. Then they can redeploy the dependent ones by hand. However, this requires human intervention which can be strenuous and even error-prone, especially for large-scale applications. In this paper, we introduce a general approach to enable the correct hot deployment with module class loaders that respects the dependencies among modules. The dependencies are constructed automatically during module loading and dependency reconstruction is triggered when any modules being depended by others are replaced with new versions at runtime. We further propose a lazy reconstruction approach for better performance. Experiments show that the partial update of the modular applications can be carried out through this technology correctly and efficiently.

The rest of this paper is organized as follow. Section II reviews the hot deployment mechanisms and analyzes the reasons of failures based on the mechanisms. Section III describes dependency reconstruction in application servers. And Section IV presents a more effective approach of dependency reconstruction with class loaders. We give the evaluations in Section V and Section VI is the related work. Lastly in Section VII, we conclude this paper and discuss the future work.

II. Background

Hot deployment built in mainstream application servers enables applications on the servers to be published, updated or withdrawn without restarting the servers. This feature is based on the class loading mechanism beyond the standard class loaders [3] in Java system. Conceptually, each module to be deployed, has its own class loader as a standalone application instead of sharing one with other modules so that the standalone application in that module can be independently operated. All the effective module class loaders are organized together (in a chain or pool) so that they can find each other and delegate each other to load classes and resources which enables the mutual referring and sharing among the modules. When a module is hot deployed, the corresponding module

class loader will be removed and a new one will be created to load the new version of the module. In this way, application servers complete hot deployment of applications [4].

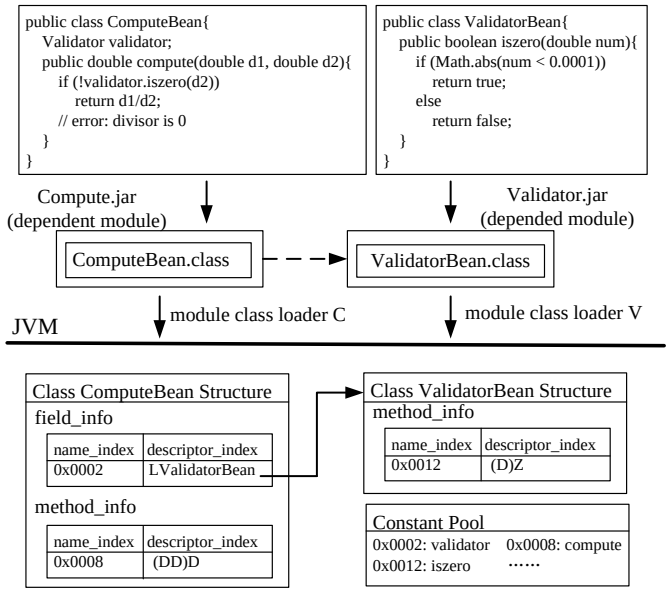


Fig. 1: A modular application and class model

Fig. 1 shows two EJBs packed in JARs. Each EJB contains only one class here for simplicity. Class *ComputeBean* in *Computer.jar* is a session bean for dividing calculation. Class *ValidatorBean* is another session bean that verifies whether a number is zero or not. The modules actually have an internal dependency between them as class *ComputeBean* uses *iszero* method of class *ValidatorBean*. Such a reference takes effect because while the classes are loaded into the JVM [5], the information of the classes are resolved by their module class loaders and stored in their own class structure tables in the virtual machine. As for the reference against *ValidatorBean*, the *field_info* table of *ComputeBean* stores the address of the class structure of *ValidatorBean*. In short, we can find the field name *validator* and field type *ValidatorBean* from the index in the *field_info* of class structure of *ComputeBean*. The *name_index* points to an entry in the constant pool and the *descriptor_index* points to the referenced class structure if this variable is a reference type [6].

At the first time of deployment, the dependency can be captured by the server because each module must be check whether it meets dependencies before deployment. Depended modules must be deployed before dependent ones, so that applications can be operated normally after module deployment and class loading. However, if the depended module is hot deployed again later, calling failures may occur after running the whole application. For example, if the class *ValidatorBean* raises its floating point precision from 0.0001 to 0.00001 and the new packed JAR file *Validator.jar* is dropped into the server, calling class *ComputeBean* will then throw an exception. It is due to the fact that according to the JVM specification [7], a class loader cannot load a class more than once, which means the original class loader cannot be used to load the new version of class *ValidatorBean*. So the server will simply destroy the existing class loader for *Validator.jar* and create a new one for the lately deployed one while *Compute.jar* remains untouched. The class structure of class *ValidatorBean* in JVM is recreated and maintained in a new place in the memory. In this case, the original reference becomes invalid. The dependency between the two modules is broken and it consequently causes a failure.

III. Dependency Reconstruction in Application Servers

To support hot deployment of modular applications, the key is to reconstruct the broken dependencies. An intuitive approach is to enable the application server to find the mutual dependencies among the modules and refresh the dependent ones to build the broken dependencies. The dependencies among modules can be explicitly stated by the application developers. Developers can provide the profiles of modules to describe their dependencies. The dependencies will be collected as soon as the modules are deployed in application servers. Fig. 2 is a modular application with three modules A, B and C. Module A depends on module B and module C, which is described as a profile *A-JAR.xml*. When deploying them at the first time, servers can create dependency mapping table by these profiles. If module B is hot deployed later, servers will realize module A depends on module B. Then module A must be redeployed and its module class loader must be updated after the hot deployment of module B. In this way, the broken dependencies are reconstructed simply and the calling failure problems are solved.

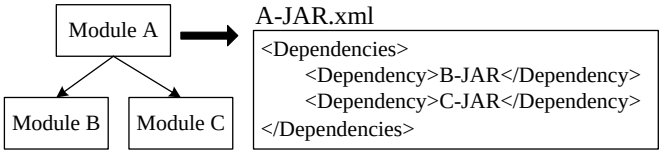


Fig. 2: A modular application with three modules

However, providing such a explicit model for dependencies is tedious, especially for the large-scale enterprise modular applications. Actually some component technologies have mechanisms to specify the inter-relationship between components. For example, EJB provides resource injection mechanism [8] as component assembling. If the module has a description for this kind of dependency, it can be used to find out the dependency between two modules. Then the dependency is recorded in the dependency mapping table, which provides dependency reconstruction with dependency information in the hot deployment. But the injections cannot reflect all the dependencies. Some modules can be SPI (service provider interfaces) and other modules are the implementation for those interfaces. In this case, they are still dependent but no component level injections are used to describe this. That means the dependency mapping table cannot record all the dependencies we want, so that dependency reconstruction probably cannot solve the calling failure problems that we discussed before.

In summary, although reconstruction in application servers can solve the problems of calling failure, there are some disadvantages in this approach. Not all the dependencies can

be obtained during deployment process and be recorded in the dependency mapping table without static module profiles. Application servers cannot reconstruct these kinds of dependencies and the broken dependencies still exist. In addition, this dependency reconstruction is a coarse-grained way to reconstruct and hot deploy modules. The dependent modules don't need updating and being redeployed is just for reconstructing dependencies. So all the dependent modules will be handled through plenty of steps from the deployment process. This causes application servers inefficient and also causes hot deployment inflexibility.

IV. Dependency Reconstruction with Class Loaders

Each module has its own class loader, the dependencies of modules are just the dependencies of class loaders. So, it is more direct to have the class loaders to maintain the dependencies among the modules that they are responsible to load.

A. Class loading respecting dependency

Designing a class loading mechanism for modular applications involves three basic considerations. Firstly, the loader should be able to load the classes in the modules correctly. Secondly, the loader should be able to resolve the dependencies among classes from different modules. Lastly, the loaders should be able to reconstruct the dependencies upon some of the modules are update later. The loading process for one module can be defined as three steps:

- Try use the system loaders to load
- Try use its dependent class loaders to load
- Try use itself to load

The first step is in charge of the loading of fundamental classes, such as those in java.*, with system class loaders respecting the parent delegation policy [9]. By this means, the safety and uniqueness of system classes will be guaranteed. This loading strategy gives priority to load classes by parent class loader. If the loading class is not a fundamental class, the class loader will try to load it by itself. The classes in the same module are visible to each other. But they may have references to the classes that come from different modules, the class loader should delegate the dependent class loaders to resolve the classes for it. We regard the system class loader as a dependent class loader of each module class loader, so that class loaders should use dependent class loaders to load classes before using itself.

Algorithm 1 shows the module class loader how to load classes in modular applications. In modular applications, a module can be defined as $M =< C, L, R >$. $C = \{c_1, c_2, c_3 \cdots\}$ is the set of classes in that module. L is a module class loader to load all the classes in the class set C. $R = \{M_1, M_2, M_3 \cdots\}$ is the set of its dependencies against other modules. They are determined from the dependency graph, which can be created by the static module profiles before class loading. Moreover, we can obtain the dependent class loader list of L from each class loader of each module in R, which represents *depList* in Algorithm 1. The module

Algorithm 1 function *loadClass* of module class loader

Input:
Fully qualified name of the class: $name$
Request time of class loading: $firstTime$
Output:
The loaded class instance: c

```
1: c ←findLoadedClass(name)
2: if c = null && visitTime < firstTime then
3:    visitTime ← currentTime
4:    for each class loader dep in depList do
5:       c ← dep.loadClass(name, firstTime)
6:       if c ≠ null then
7:          return c
8:    c ←findClass(name)
9: return c
```

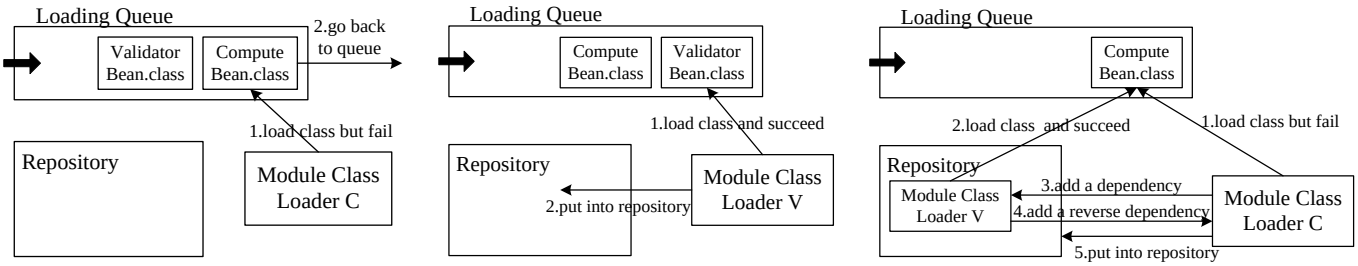
class loaders may delegate each other due to the *depList* with circles, which will cause endless loop. So we take a simple solution to detect circles in the algorithm of loading classes. In Algorithm 1, $firstTime$ is a timestamp for the beginning of the class loading and $visistTime$ is a timestamp used to represent the last visit time for using this class loader. If the condition $visitTime < firstTime$ is not *true*, there is no need to delegate dependent class loaders to load because they have already been delegated. In this way, the class loaders will not delegate each other, even if there is interdependence among modules.

B. Constructing the dependency graph

The dependency graph can be created by the module profiles like reconstruction in servers. But editing module profiles is a quite messy task when modular applications are composed of hundreds of modules. Dynamic dependency management avoids this work because it has a loading queue to make a try when loading classes. The loaded class will be placed in the repository and the unloaded class will enter the loading queue again to wait for the next loading. Due to continuous trying, we can obtain dependencies of all the classes (modules) and construct the dependency graph by an adaptive way. The loading rules extend the basic loading process and the priority order is described in the following:

1) Try to use the system class loader to load it, if the class name begins with java.*.
2) Try to use its own class loader to load it.
3) Try to use its dependent class loaders to load it.
4) Try to use the class loaders which have already loaded other classes in the repository.

Fig. 3 shows the loading classes process of the division application that we used before. We put two classes into the loading queue and try to load *ComputeBean* at first. Due to the dependencies, module class loader C which belongs to *Compute.jar* fails to load it. Class *ComputeBean* should go back to the loading queue again to wait the next loading, which is described as Fig. 3a. Then it is turn to load Class *ValidatorBean* in Fig. 3b. Module class loader V loads it successfully, so we add the class loader into the repository. With more classes loaded successfully, more class loaders will



(a) loading *ComputeBean* (b) loading *ValidatorBean* (c) loading *ComputeBean* again

Fig. 3: Details of loading class and constructing dependency

be put in the repository and they help to load other classes. Thus, *ComputeBean* can be loaded successfully at the second time by the class loader in the repository as presented in Fig. 3c. If a class loader in the repository loads the other class successfully, that means it becomes a dependent class loader and the dependency is created in the class loader incidentally. Finally, the loading queue becomes empty and all the classes are loaded with the dependencies.

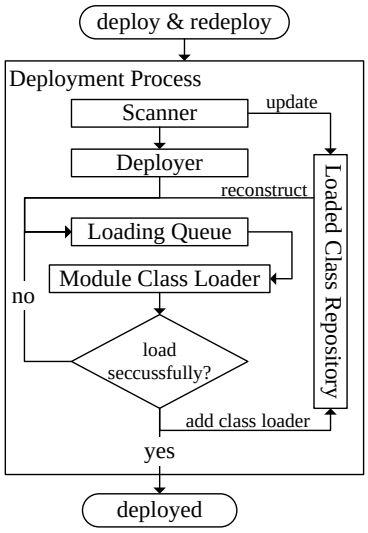


Fig. 4: Dependency reconstruction with class loaders

Fig.4 shows the whole process of hot deployment with dependency reconstruction with class loaders. The affected modules only update their module class loaders and the affected classes need to be reloaded through the loading queue, so that we save the partial cost of redeployment process for the dependent modules which are still the old versions. Even though adaptive way to load class is time consuming, avoiding editing dependencies in the module profiles is a time saving thing for developers of modular applications or administrators of application servers.

C. Lazy reconstruction

If hot deployment happens, the old version classes in the repository must be removed and their related dependencies must be dropped. Then, the new version classes are also placed in the loading queue to wait for loading. Dependency reconstruction with class loaders needs to find out all the dependent modules group by group. These modules must be removed in the repository and put their classes in the loading queue to be reloaded. So, all the broken dependencies will be reconstructed by taking the place of the original class loaders and their dependent class loaders.

Lazy reconstruction makes module class loaders more flexible. Comparing with immediate reconstruction, the class loader with lazy reconstruction delays dependency reconstruction when hot deployment happens. The class loader doesn't reconstruct dependencies at once. All the affected class loaders only mark as invalid class loaders and none of them are updated after hot deployment. Dependency reconstruction will happen when applications are executed. In this time, the class loader with lazy reconstruction has to reconstruct dependencies to ensure the correctness of applications. If the class loader is invalid, we should update and reconstruct it before using it to load classes. The algorithm of loading class is described in Algorithm 2, which uses the loading rules we discussed before.

V. Evaluation

Hot deployment with dependency reconstruction can ensure the correctness of the modular applications. We use a modular application example in Fig. 1 to evaluate the hot deployment with dependency reconstruction. And our experimental environment is following:

1) OS: Ubuntu 12.04
2) JDK: Java Version "1.6.0_24"
3) IDE: Eclipse IDE for Java EE Developers
4) AS: JBoss AS 5.1.0 & JBoss AS 5.1.0 extension (reconstruction with class loaders)

A. Efficiency

To ensure the correctness of applications, the broken dependencies must be reconstructed after partial update of the modular applications. The simple way is to redeploy the dependent modules with original versions manually after hot deployment of the depended ones. However, most of the work in redeploying dependent modules makes no sense except class loading in the modules. Reconstruction with class loaders improves updating efficiency because dependency management

Algorithm 2 function *loadClass* with lazy reconstruction

Input:
Fully qualified name of the class: $name$
Output:
The loaded class instance: c

```
1: c ←findLoadedClass(name)
2: if c = null then
3:    if name.startWith("java.") then
4:       c ← SystemClassLoader.loadClass(name)
5:    else
6:       c ←findClass(name)
7:       if c = null then
8:          for each class loader dep in depList do
9:             c ← dep.loadClass(name)
10:            if c ≠ null then
11:               return c
12:         for each class loader cl in repository do
13:            if cl.valid = false then
14:               cl ←reconstruct(cl)
15:               cl.valid←true
16:            c ← cl.loadClass(name)
17:            if c ≠ null then
18:               depList.add(cl)
19:               return c
20: return c
```

is handled in the class loaders, which means a lot of work in the redeployment process can be eliminated when reconstruction happens.

According to deployment specification [10], deployment is typically a three-stage process: configuration, distribution and start execution. We focus on the first two stages in our experiments. Deployment context of modules are created in configuration stage, but they are not changed when redeployment of dependent modules happens. We use the original context and only update its module class loader before distribution stage which is responsible for installing modules and loading classes. So we save the time used to create and set the deployment context in the redeployment process.

Table I shows the cost of redeployment stages in these two stages for some modules and applications. *Compute.jar* and *Validator.jar* are two session beans which constitute a modular application for dividing calculation in Fig. 1. Java Pet Store [11] is a sample well-known application from the J2EE, so it is also one of our subjects. The rest testing modules are services of JBoss, which can be found in the directory of JBossAS 5.1.0 Release named *examples*. We hot deploy them when the application server is running and calculate the time cost of the configuration stage and distribution stage.

In our approach of reconstruction with class loaders, the time cost of the configuration stage will be saved in the redeployment of dependent modules. The efficiency is highly improved according to the ratio of time saved in Table I. Moreover, dependent modules won't update themselves in the deployment directory and the scanning time, which is 5 seconds as default configuration, is saved and that means applications don't require to wait for the class loading delay of dependent modules.

B. Flexibility

The class loader is an important part in application servers based on J2EE. Reconstruction with class loaders makes hot deployment more flexible. On the one hand, it allows class loaders to construct dependencies in the attempt to load classes without profiles. By several attempts, we can create the dependency graph on the basis of whether the class is loaded successfully. Developers don't need to provide the module profiles to describe the dependencies of the modules. On the other hand, reconstruction can be carried out at any time before execution of applications. The class loader with lazy reconstruction, which should be reconstructed, is only marked as invalid when hot deployment happens. The class loaders and their dependencies won't be updated until the applications start to be executed. So, we handle invalid class loaders and reconstruct them at the time of application execution. Comparing with immediate reconstruction, lazy approach delays reconstruction and increase the flexibility of reconstruction.

In a particular application scenario, the class loader with lazy reconstruction has a good performance. We compare dependency reconstruction between the immediate reconstruction strategy and the lazy reconstruction strategy through the experiments on 20 modular applications. Each modular application contains three modules A, B and C while their dependencies among the modules are shown as Fig. 2. We deploy them and then hot deploy module B of all the applications. To ensure the correctness of applications, module A and B need to be reconstructed. With lazy reconstruction, we execute all the applications after redeployment so that all the class loaders of module B finish reconstructing. If we execute one application, only the class loaders in this application will complete reconstruction. So, the time of hot deployment and execution decreases in Table II from 52ms to 43ms.

VI. Related Work

Although many researchers dedicated to the study of hot deployment, most of the work focused on the hot deployment of distributed heterogeneous environments [12]–[14]. They solved problems about the dynamic service creation, service life cycle management and other issues based on OGSA (Open Grid Service Architecture). In terms of dependency injection, most of the work focused on the dependency injection mechanism and its performance in different environments [15], [16]. They involved a number of fields, such as distributed applications, design patterns, and software maintenance.

Improving abilities of application servers through component extension is not a new idea. For example, Li, et al. advocated an on demand approach of deploying services in application servers [17] and they even proposed an approach to make an application server well-structured and dynamic [18]. And some research focused on refactoring application servers according to application requirements [19].

Our work proposes a technology of hot deployment with dependency reconstruction, which makes hot deployment more flexible and efficient. By this technology, we solve the problems of calling failure when only depended modules are updated. But our main contribution is to help server administrators cut down the scope of the update and reconstruction in the engineering practice.

TABLE I: Time cost of deployment stages

Module Name	*Configuration Stage*	*Distribution Stage*	*Total Time*	*Ratio of Time Saved*
Compute.jar	13ms	33ms	46ms	28.3%
Validator.jar	12ms	29ms	41ms	29.3%
pestore.jar(1.1.2)	34ms	236ms	270ms	12.6%
pestore.jar(1.3.2)	24ms	113ms	137ms	17.5%
netboot.war	19ms	65ms	84ms	22.6%
persistent-service.sar	14ms	35ms	49ms	28.6%
ejb-management.jar	12ms	283ms	295ms	4.1%
derby-plugin.jar	16ms	16ms	32ms	50.0%
threaddump.war	10ms	52ms	62ms	16.1%
jbossts-tools.sar	21ms	87ms	108ms	19.4%
jbossxts.sar	52ms	860ms	912ms	5.7%

TABLE II: Performance of flexible reconstruction

Reconstruction Strategy	*Deploy and Execute All*	*Redeploy and Execute All*	*Redeploy and Execute One*
Immediate reconstruction	94ms	141ms	52ms
Lazy reconstruction	92ms	138ms	43ms

VII. Conclusion and Future Work

Hot deployment is one of the typical features of mainstream application servers. However, they cannot meet the demand with hot deployment of modular applications. This paper proposes a technology of hot deployment with dependency reconstruction which can solve the problem of hot deploying modular applications. Dependency reconstruction with class loaders is implemented, which makes hot deployment more flexible and efficient. Dependencies are managed in class loaders after deploying modules and reconstruction is done at any time before running applications. Experiments show that the problems of calling failure can be solved and the efficiency of application servers is improved.

In future, we will focus on reconstruction with class loaders. Managing dependencies in class loaders make it possible to do fine-grained update and reconstruction. We can split a module into several sub modules [20] or assemble some modules [21] according their original dependencies. So reconstruction with class loaders will shrink the scope of the hot deployment. Moreover, we will implement dependency reconstruction in the code level [22], so that code-level dynamic update will be achieved and it will continue to decrease the granularity of update.

Acknowledgment

This work is partially supported by National Basic Research 973 Program(Grant No. 2015CB352202), and National Natural Science Foundation(Grant Nos. 61472177, 91318301, 61321491, 61361120097) of China.

References

[1] JBoss Application Server, http://www.jboss.org.

[2] Oracle WebLogic Server, http://www.oracle.com/technetwork/middleware/weblogic/overview/index.html.

[3] Taylor, B. "Java Class Loading: The Basics." (2003).

[4] Gangadharan, Binod Pankajakshy, Srikanth Padakandla, and Sivakumar Melapannai Thyagarajan. "Hot deployment of shared modules in an application server." U.S. Patent No. 7,721,277. 18 May 2010.

[5] Java Virtual Machine, http://java-virtual-machine.net/sun-java-virtual-machine.html.

[6] Zhiming, Zhou. "Understanding the JVM Advanced Features and Best Practices." (2011).

[7] The Java Virtual Machine Specification, http://docs.oracle.com/javase/specs/jvms/se7/html.

[8] Fowler, Martin. "Inversion of control containers and the dependency injection pattern." (2004).

[9] Rohit Chaudhri. "Understnding the Java Classloading Mechanism." Java Developer's Journal Vol.8, Issue 8, p.16.

[10] JSR-000088, Deployment API Specification, http://jcp.org/aboutJava/communityprocess/mrel/jsr088/index.html.

[11] Java Pet Store, http://www.oracle.com/technetwork/java/petstore1-3-1-02-139690.html.

[12] Dornemann, Kay, and Bernd Freisleben. "Discovering Grid Resources and Deploying Grid Services Using Peer-to-Peer Technologies."Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on. IEEE, 2009.

[13] Friese, Thomas, Matthew Smith, and Bernd Freisleben. "Hot service deployment in an ad hoc grid environment."Proceedings of the 2nd international conference on Service oriented computing. ACM, 2004.

[14] Abdellatif, Takoua, Jakub Kornas, and J-B. Stefani. "Reengineering J2EE servers for automated management in distributed environments."Distributed Systems Online, IEEE8.11 (2007): 1-1.

[15] Heinrich, Matthias, et al. "Enriching web applications with collaboration support using dependency injection."Web Engineering. Springer Berlin Heidelberg, 2012. 473-476.

[16] Rajam, Sidhant, et al. "Enterprise service bus dependency injection on mvc design patterns."TENCON 2010-2010 IEEE Region 10 Conference. IEEE, 2010.

[17] Li, Yan, et al. "Enabling on demand deployment of middleware services in componentized middleware."Component-Based Software Engineering. Springer Berlin Heidelberg, 2010. 113-129.

[18] You, Chao, et al. "Towards a well structured and dynamic application server."Computer Software and Applications Conference, 2009. COMPSAC'09. 33rd Annual IEEE International. Vol. 1. IEEE, 2009.

[19] Zhang, Charles, Dapeng Gao, and Hans-Arno Jacobsen. "Towards just-in-time middleware architectures."Proceedings of the 4th international conference on Aspect-oriented software development. ACM, 2005.

[20] Snchez, Ivn Bernab, Daniel Daz-Snchez, and Mario Muoz-Organero. "Optimizing OSGi Services on Gateways."Ambient Intelligence-Software and Applications. Springer International Publishing, 2013. 155-162.

[21] Brannen, Samuel Hugh, et al. "Computer system and a method of deploying an application in a computer system." U.S. Patent No. 8,359,590. 22 Jan. 2013.

[22] Gu, Tianxiao, et al. "Javelus: A Low Disruptive Approach to Dynamic Software Updates."Software Engineering Conference (APSEC), 2012 19th Asia-Pacific. Vol. 1. IEEE, 2012.