Hot Deployment with Dependency Reconstruction

Haicheng Li, Chun Cao, Ping Yu
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, yuping}@nju.edu.cn

***Abstract*—The hot deployment mechanism is a typical feature of mainstream application servers. But current application servers only support hot deployment of standalone applications, which cannot satisfy the requirement of modular applications. Failures will occur when some depended modules are updated online, which will result in failure of the whole application platform. To solve the problem, all the dependent modules must be redeployed in a manual or semi-automatic manner at the cost of increasing overhead after deployment of depended modules. So a new technology of hot deployment with dependency reconstruction is introduced in order to avoid the unnecessary redeployment. Dependency management and maintenance is placed in module class loaders, so that the cost of dependency reconstruction will be reduced. Experiments show that our technology of hot deployment can ensure the correctness of modular applications and operating efficiency of application servers will be highly improved.**

Keywords*—*Hot deployment; Dependency reconstruction; Class loading.

I. INTRODUCTION

Nowadays application servers[1] provide platforms for enterprise applications to be deployed, operated and maintained. They usually contain comprehensive services, such as clustering, security and transaction management, et. al, 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[2] and WebLogic[3], 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 spans 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 are already packed independently. They still works 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 of the original dependencies between the portion to be hot deployed and the rest of the application are broken and applications are actually down[4].

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 the affection scope of the redeployment. Then they can redeploy the dependent ones by hand. However, this requires human intervention can be strenuous and even error-prone, especially for large-scale application. Alternatively, they can also redeploy all the modules for that application. But it is not only unnecessary but also cause the state of unaffected part to be destructed, such as data storage module which is hardly updated.

In this paper, we introduce a general approach to enable the correct hot deployment respecting the dependencies among modules. The dependencies are constructed with class loaders for the modules and upon the dependent modules can be calculated and their hot deployment can be triggered. In order to update flexibly and efficiently, dependency reconstruction can be done in the class loaders. We can get and manage dependencies from the static module profiles or by dynamic way during hot deployment, so that dependent modules only update their class loaders and they don't need to do anything else except class loading in the process of redeployment. Experiment shows the problems of calling failure can be solved through this technology and the efficiency of application servers is improved.

The rest of this paper is organized as follow. Section II reviews the hot deployment mechanism and analyzes reasons of failures based on hot deployment mechanism. Section III describes dependency reconstruction in application servers, which is a common approach to solve dependency lost. And Section IV presents a more effective approach of dependency reconstruction in 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 future work.

II. BACKGROUND

In this section, the hot deployment mechanism of current application servers are reviewed. We also analyze the cause of the failure in hot deploying the modular applications.

A. Hot Deployment

Hot deployment built in mainstream application servers enables applications on the servers to be are published, updated or withdrawn without restarting the servers. This feature is based on the class loading mechanism beyond the standard class loaders[5] in Java system. Conceptually, each module, as an standard alone application, to be deployed has its own class loader instead of sharing one with other modules so that standalone applications 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 some module are 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[6].

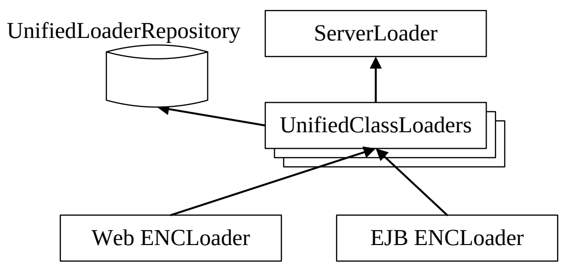


Fig. 1: class loader architecture in JBoss

Fig. 1 shows an illustrative architecture of class loading in JBoss and other application servers adopt the similar design. As for JBoss, *ServerLoader* is responsible to load JBoss server itself. For applications of different types, including EARs, JARs, WARs, etc., seen by the deployment scanner, the server applies various loaders, including *EJB ENCLoader* and *Web ENCLoader*, to load classes respectively. In advance, each class loader delegates the *UnifiedClassLoader* to manage the classes loaded in a repository. Fig. 2 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. If these two modules are deployed in JBoss, two *EJB ENCLoader* instances are responsible for loading the classes and they delegate two *UnifiedClassLoader* instances to load classes. So each *UnifiedClassLoader* instance serves the corresponding module as its module class loader.

B. Hot Deployment Failures Caused by Module Dependency

The modules in the above example actually have an internal dependency between them as class *ComputeBean* uses *iszero* method of class *ValidatorBean* to verify whether the divisor is zero or not by referring the latter class. Such a reference takes effect because while the classes are loaded into the JVM[10], 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*, as shown in

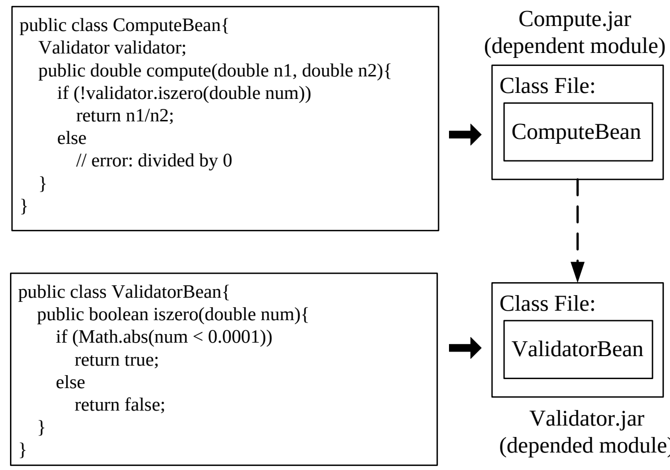


Fig. 2: a modular application for dividing calculation

Fig. 3. 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[11].

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. By a waiting queue of deploying modules, 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* then will throw an exception.

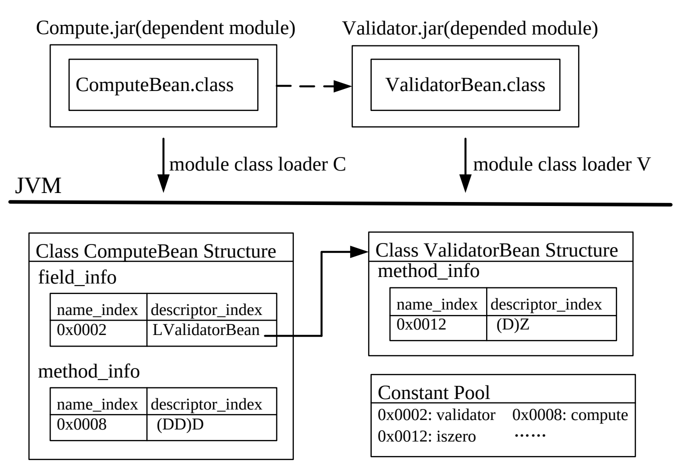


Fig. 3: a modular application and loading details

The reason is that, according to the JVM specification[12], a class loader cannot load a class twice, which means we cannot use the original class loader 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. Dependency between the two modules are broken and it consequently causes a failure.

To support hot deployment of modular applications, the key is reconstructing the lost 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 dependency. The details of dependency reconstruction will be presented in the following sections.

III. Dependency Reconstruction in Application Servers

Dependencies among modules can be explicitly stated by the application developers. Developers can configure the profiles of modules to describe their dependencies. The dependencies will be collected as soon as the modules are deployed in application servers. Fig. 4 is a modular application with three modules A, B and C. Module A depends on module B and C, which is described as profile of module A. 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 and module A must be redeployed after the hot deployment of module B. In this way, lost dependencies are reconstructed simply and calling failure problems are solved.

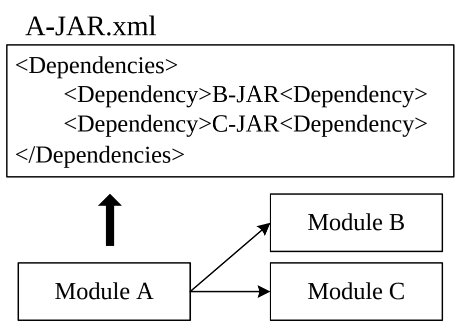


Fig. 4: a modular application with three modules

To provide 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 as component assembling. If the module has a description for this kind of dependency, it can be used to find out the dependencies between two modules. Then the dependencies are recorded in the dependency mapping table, which provides dependency reconstruction with dependency information in the hot deployment. However, the injections are not complete. For example, some modules can include 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 appropriate for this. That means the dependency mapping table cannot record all the dependencies we want, so that dependency reconstruction is not complete propably.

For this case, application server can obtain the dependencies according to the waiting queue of modules. The correct deploying order should make each module meet dependencies. The order of the modular application in Fig. 4 may be B-C-A or C-B-A, because the dependent module A must be deployed after the depended modules B and C. From the deploying order, we cannot record all the dependencies preciously, but we can redeploy all the possible dependent modules for dependency reconstruction. If the order is C-B-A and module C is hot deployed, redeploying module B and module A is a reliable way to reconstruct dependencies, even though there is no need to update module B.

Although reconstruction in application servers can solve the problems of dependency loss and 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 profiles. Dependencies of inheritance or general reference cannot be detected by servers. So application servers cannot reconstruct these kinds of dependencies and the problem of dependency loss still exists in most of modular applications. In addition, this dependency reconstruction is a coarse granularity way to reconstruct and hot deploy modules. All the dependent modules need to be redeployed, so they will be handled through plenty of steps from the deployment process. Actually, depended modules are the updated modules and updating dependent modules is just for reconstructing dependencies. This causes application servers inefficient and also causes hot deployment inflexibility.

IV. Dependency Reconstruction with Class Loaders

Building the dependency reconstruction process into the deployment process suffers from inefficiency or even infeasibility. Because each module has its own class loader, the dependencies of modules are just the dependencies of class loaders. It's 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 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 to complete the loading of fundamental classes, such as those in java.lang.*, with system class loaders respecting the parent delegation policy. By this means, the safety and uniqueness of system classes are guaranteed. This loading strategy gives priority to load class by parent class loader. Bootstrap class loader has the highest priority and the next is extension class loader and system class loader.

If the class to load 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 the class in the same module 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.

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 dependency against other modules. Take the division modular application shown in Fig. 2 as an example, Compute.jar is a dependent module and $R = \{Validator.jar\}$, because class *ComputeBean* calls the method in class *ValidatorBean* and these two classes are in different modules. Validator.jar is a dependent module used to verify the divisor and $R = \emptyset$, which means Validator.jar doesn't depends on any modules.

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
$c \leftarrow$*findLoadedClass*($name$)
if $c = null$ && $visitTime < firstTime$ **then**
  $visitTime \leftarrow currentTime$
  for each class loader dep in $depArrayList$ **do**
    $c \leftarrow$ *dep.loadClass*($name$, $firstTime$)
    if $c \neq null$ **then**
      return c
  $c \leftarrow$*findClass*($name$)
else
  return c

Algorithm 1 shows the module class loader how to load classes. Under the definition of module dependency, class set C, class loader L and dependencies R of each module M 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 *depArrayList* in Algorithm 1.

The dependency graph probably has a circle so that there must be a circle detection in the algorithm of loading classes. In Algorithm 1, $firstTime$ is a timestamp which records the time for the beginning of the class loading and $visistTime$ is a timestamp which is used to represent the last visit time for using this class loader. If $visitTime < firstTime$, there is no need to delagate dependent class loaders to load because they were delagated some time ago. In this way, the class loaders will not delegate each other, causing endless loop, even if there exists interdependence among modules.

When updating is detected, some affected class loaders and dependencies must be modified. However, modifying the changing class loaders and their dependencies is not enough because dependencies may be lost while hot deploying depended modules. For instance, if Validator.jar is updated, in order to reconstruct dependencies, all of modules which depend on Validator.jar also require to be updated even though they have no new versions and users don't want to update them. So we reconstruct the dependencies for Validator.jar by updating these class loader of affected modules. But reconstruction is not finished because these dependent modules may be depended on other modules. We need to reconstruct their dependencies too in the same way, until there is no updated modules are depend on any of the updated modules. All the affected modules should own their new class loader instance instead of the old one, because the old class loader has the cache of old classes and it cannot load the new classes with the same name. Replacing the old class loaders is the most convenient way to update modules and reconstruct dependencies. Meanwhile, updating the old class loader instances and dependencies is necessary.

B. Constructing the Dependency Graph

The dependency graph can be created by the module profiles and it can be managed in the static way. But editing module profiles is a quite messy work 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 priority order of loading rules 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. 5 shows the loading classes process of the division application that we used before. There are two classes *ComputeBean* and *ValidatorBean* in the application. We put them 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. 5a. Then it is turn to load Class *ValidatorBean* in Fig. 5b. Module class loader V loads it successfully, so we add the class loader into repository. With more classes loaded successfully, more class loaders will 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. 5c. If a class loader in the repository loads the other class successfully, that means it is the dependent class loader of the other 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. Without static profiles, module class loaders create dependencies dynamically while loading classes, so we call this type of module class loader *Adpative Class Loader*.

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 and the process is just like deployment process. Dependency reconstruction in dynamic dependency management is similar with it in static dependency management. We can find out all the dependent modules group by group. No matter whether these modules have the new version, they must be removed in the repository as the old ones and then put them in the loading queue to be reloaded. So, all the lost dependences will be reconstructed by taking the place of the original class loaders and their dependent class loaders.

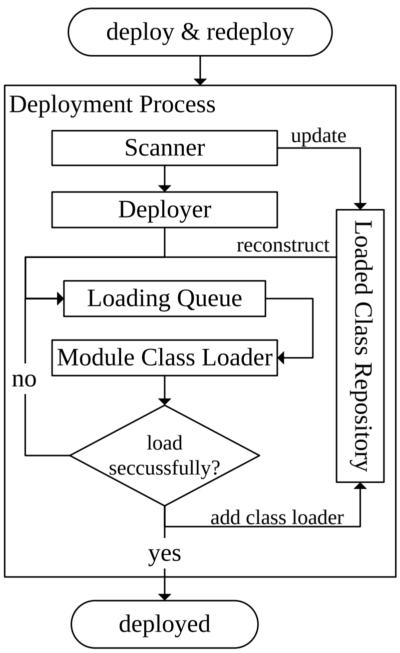


Fig. 6: dependency reconstruction in class loaders

Fig.6 shows the whole process of hot deployment with dependency reconstruction in 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 part 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 profiles of modules is a time saving thing for developers of modular applications or administrators of application servers.

C. Lazy Reconstruction

Delay Class Loader is a type of module class loader with lazy reconstruction. Comparing with *Adaptive Class Loader*, *Delay Class Loader* delays dependency reconstruction when hot deployment happens. *Delay Class Loader* doesn't reconstruct dependencies at once. So all the affected class loaders only mark as invalid class loaders and none of class loaders

Algorithm 2 function loadClass with lazy reconstruction

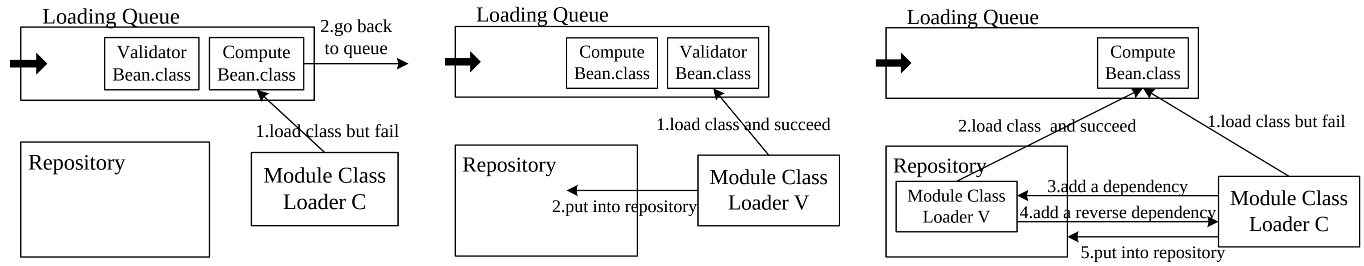
Input:
Fully qualified name of the class: $name$
Output:
The loaded class instance: c

```
c ←findLoadedClass(name)
if c = null then
    if name.startWith("java.") then
        c ← systemClassLoader.loadClass(name)
        if c ≠ null then
            return c
    else
        c ←findClass(name)
        if c = null then
            for each class loader dep in depArrayList do
                c ← dep.loadClass(name)
                if c ≠ null then
                    return c
            for each class loader cl in repository do
                if cl.valid = false then
                    cl ←reconstruct(cl)
                    cl.valid←true
                c ← cl.loadClass(name)
                if c ≠ null then
                    depArrayList.add(cl)
                    return c
        else
            return c
else
    return c
```

are updated after hot deployment. Dependency reconstruction will happen when applications are executed. Because in this time, *Delay Class Loaders* have 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 of *Adaptive Class Loader*. Similarly, dependency reconstruction process is also as the same as the reconstruction process of *Adaptive Class Loader*. Their main difference is their reconstruction time. *Delay Class Loader* reconstructs dependencies when applications are executed, so that it may save the time of hot deployment and avoid reconstruction if users never execute some modules after updating them.

D. Contribution

Our contribution is to propose a technology of hot deployment with dependency reconstruction. From an engineering point of view, hot deployment with dependency reconstruction solves the problem of hot deploying modular applications in practical engineering projects. Particularly, we customize the class loader which can manage dependencies of modules. Then we can reconstruct dependencies in the class loader, so that we improve the efficiency of hot deployment. In the future, we can split a module into several sub modules or assemble some sub modules of different modules according their original dependencies. Each class loader only the loads classes of corresponding sub modules. So reconstruction in class loaders reduces the granularity of the update and narrows the scope



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

Fig. 5: details of loading classes via Adaptive Class Loader

of the hot deployment. The hot deployment becomes more flexible.

V. Evaluation

In this section, we did some evaluations on efficiency, dynamic feature and flexibility for our technology of hot deployment with dependency reconstruction.

A. Experiment

Mainstream application servers only support hot deployment of standalone applications, which cannot satisfy the requirement of modular applications. Hot deployment with dependency reconstruction can ensure the correctness of the modular applications. We use a simple division modular application in Fig. 2 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 in AS)

We start the application server and deploy two session beans called Compute.jar and Validator.jar and we can compute division correctly through this modular application. Then we only update Validator.jar. After hot deployment, the application in JBoss AS 5.1.0 will throw the EJB exception when we use Compute.jar to do division of two numbers. However, JBoss AS 5.1.0 extension reconstructs lost dependencies and redeploys Compute.jar, so that we can make the calculation without exception. Dependency reconstruction guarantees the correctness of hot deploying dependent modules in modular applications.

B. Efficiency

Comparing with JBoss AS 5.1.0, the application server extension, using dependency reconstruction in application servers, needs more time to hot deploy modules because the server searches dependencies in the reverse dependency table and deploys more dependent modules. But this type of dependency reconstruction can guarantee the correctness of hot deployment without restarting application servers. Obviously, the hot deployment with dependency reconstruction which can avoid restarting servers is more efficient.

Reconstruction in application servers is a native way to solve calling failures because many dependent modules of original versions are redeployed after the depended modules. However, most of the work in redeploying dependent modules makes no sense except class loading in the modules. Reconstruction in class loaders improves updating efficiency because dependency management and reconstruction is handled in the class loader most of the deployment process is omitted.

According to deployment specification[16], 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 creatd 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 in these two stages for some modules and applications. Compute.jar and Validator.jar are two session beans which constitute an example application of division in Fig. 2. Java Pet Store[17] is a sample well-known application from the J2EE[18], so it is also one of our subjects. The rest testing modules are services of JBoss, which can be found in the examples directory of JBossAS 5.1.0 Release. 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 in class loaders, the time cost of the configuration stage will be saved when hot deployment happens. 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. The efficiency is highly improved according to the promotion rate of Table I.

C. Dynamic Feature

Comparing with the class loader based on static dependency management, *Adaptive Class Loader* and *Delay Class Loader* can obtain the dependencies of modules in the dynamic

TABLE I: time cost of redeployment

Module Name	*Configuration Stage*	*Distribution Stage*	*Total Time*	*Promotion Rate*
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%

way. By several trying, we can create the dependency graph on the basis of whether the class is loaded successfully. Developers don't configure the profile of modules to describe the dependencies of the modules. The dependencies are managed and reconstructed dynamically at any time.

Actually, static dependency management is more efficient than dynamic dependency management in terms of deployment and redeployment time. Distribution stage accounts for a large proportion of the total deployment process according to Table I. *Adaptive Class Loader* and *Delay Class Loader* may try to load a class for many times if this class depends on other classes which are packaged in different modules.

However, the advantage of dynamic feature is reducing management of modules for developers. As we know, enterprise applications probably contain hundreds of modules and complex dependencies. It is annoying for developers to create and manage configuration files of each module if there are no reliable automatic tools for generating them. Additionally, dynamic feature makes hot deployment more flexible, which we will talk about in the following part.

D. Flexibility

The class loader is an important part in application servers based on J2EE. Reconstruction in class loaders make hot deployment more flexible. On the one hand, dynamic dependency management allows class loaders to construct dependencies in the attempt to load classes without configuration files. On the other hand, reconstruction can be carried out at any time before execution of applications. The dependency reconstruction of *Adaptive Class Loader* is in the process of hot deployment, which means the class loaders finish reconstructing dependencies when the application servers complete hot deployment. In contrast, *Delay Class Loader* which should be reconstructed is 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 *Adaptive Class Loader*, *Delay Class Loader* delays reconstruction and increase the flexibility of reconstruction.

In a particular application scenario, *Delay Class Loader* Class Loader has a good performance. We compare performance of flexible reconstruction between *Adaptive Class Loader* and *Delay Class Loader* through experiments. We choose a simple modular application with three modules A, B and C, shown as Fig. 4. Then we copy this application for 20 times and build 20 modular applications with identification from 1 to 20. In the experiment, we deploy 20 applications with 60 modules in the application servers and execute all of them. Then we hot deploy module B of all the applications. To ensure the correctness of applications, module A and B need to be reconstructed. After redeployment, we execute all the applications so that all of class loaders finish reconstructing. If we execute only one of applications, only *Delay Class Loader* of this application modules will complete reconstruction. So, the time of hot deployment and execution decreases in Table II.

VI. Related work

The concept of modularity was proposed a long time ago by Divid Parnas[19]. He gave two main criteria[20] to be used in decomposing systems into modules in 1972. They were making each major step in the processing module and using information hiding as a criterion. After that, people gradually changed the way of developing software. And Modularity impacts on development, deployment, operation, maintenance in software engineering until now. So, many researchers focus on modularity and modular applications in all fields of software engineering, such as software metric[21] and software testing[22]. Even some new technologies and concepts(e.g. AOP[23] and cloud[24]) of software engineering also have related to modularity and modular applications.

Although many researchers dedicated to the study of hot deployment, most of the work focused on the hot deployment of distributed heterogeneous environments[25]–[28]. 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[29]–[31]. They involved a number of fields, such as distributed applications, design patterns, and software maintenance. In contrast, our work proposes a technology of hot deployment with dependency reconstruction, which makes hot deployment more flexible and efficient.

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[32] and they even proposed an approach to make an application server well-structured and dynamic[33]. And some research focused on refactoring application servers according to application requirements[34].

TABLE II: performance of flexibile reconstruction

Loading Strategy	*Deploy and Execute All*	*Redeploy and Execute All*	*Redeploy and Execute One*
Adative Class Loader	94ms	142ms	52ms
Delay Class Loader	92ms	137ms	43ms

On the other hand, application servers can preprocess modules before modular applications are deployed. It brings many benefits in performance. For instance, several modules merge into one module to reduce dependencies[35] or a module splits into several ones in order to reduce the cost of update[36]. It is also helpful to our future work on dependency reconstruction in class loaders. In this way, we will decrease the granularity of update and reconstruction.

VII. CONCLUSION AND FUTURE WORK

Hot deployment mechanism is one of the typical features of mainstream application servers. However, mainstream application servers cannot meet the demand with hot deployment of modular applications. This paper proposes a technology of hot deployment with dependency reconstruction. In order to make hot deployment becomes more flexible and efficient, dependency reconstruction is implemented by several ways, including static profiles way, adaptive loading way and dynamic management with lazy reconstruction. Experiment shows the problems of calling failure can be solved through this technology and the efficiency of application servers is improved.

In future, we will focus on reconstruction in class loaders. Managing dependencies in class loaders make it possible to decrease the granularity of update and reconstruction. Due to the more flexible hot deployment, the efficiency of application servers will be highly increased. Moreover, we will implement dependency reconstruction in the code level[37], so that code-level dynamic update will be achieved and it will continue to decrease the granularity of update.

ACKNOWLEDGMENT

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

REFERENCES

[1] What is an App Server, http://www.theserverside.com/news/1363671/What-is-an-App-Server.

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

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

[4] Huang, Gang, et al. "Simulation-based analysis of middleware service impact on system reliability: Experiment on Java application server." Journal of Systems and Software 84.7 (2011): 1160-1170.

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

[6] 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.

[7] The JBoss JMX Microkernel, http://docs.jboss.org/jbossas/jboss4guide/r5/html/ch2.chapter.html.

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

[9] JSR 220: Enterprise JavaBeans™ 3.0, https://www.jcp.org/en/jsr/detail?id=220.

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

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

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

[13] Olliges, Sascha. Runtime Reconfiguration in J2EE Systems. Diss. Master Thesis supervised by Jasminka Matevska and Wilhelm Hasselbring, University of Oldenburg, Germany, Department of Computing Science, Software Engineering Group, 2005.

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

[15] Meduri, Subbarao K., Thomas Edward Musta, and James Lee Van Oosten. "Collaborative classloader system and method." U.S. Patent No. 7,870,546. 11 Jan. 2011.

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

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

[18] Bodoff, Stephanie, ed. The J2EE tutorial. Addison-Wesley Professional, 2002.

[19] Parnas, David Lorge. "A technique for software module specification with examples." Communications of the ACM 15.5 (1972): 330-336.

[20] Parnas, David Lorge. "On the criteria to be used in decomposing systems into modules." Communications of the ACM 15.12 (1972): 1053-1058.

[21] Kazemi, Ali, et al. "A metric suite for measuring service modularity." Computer Science and Software Engineering (CSSE), 2011 CSI International Symposium on. IEEE, 2011.

[22] Judge, Lyndon, et al. "A modular testing environment for implementation attacks." BioMedical Computing (BioMedCom), 2012 ASE/IEEE International Conference on. IEEE, 2012.

[23] Przybyek, Adam. "Where the truth lies: AOP and its impact on software modularity." Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 2011. 447-461.

[24] Wettinger, Johannes, et al. "Enabling Dynamic Deployment of Cloud Applications Using a Modular and Extensible PaaS Environment." Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on. IEEE, 2013.

[25] Florian, Vladimir, Gabriel Neagu, and Stefan Preda. "An OGSA Compliant Environment for eScience Service Management."P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2010 International Conference on. IEEE, 2010.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] Razina, Ekaterina, and David S. Janzen. ”Effects of dependency injection on maintainability.”Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA. 2007.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.