Hot Deployment with Dependency Reconstruction

Haicheng Li, Chun Cao, Xiaoxing Ma, Jian Lu
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@gmail.com, {xxm, lj}@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*—Modular applications; 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, et. al, so that developers can focus on the business logic. As one 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 JBossAS[8] and xxxx, 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 modulars) 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. (we should explain how the dependencies are constructed during the first time deployment here)

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 existents of potential relationships among the modulars. 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.

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 modulars and be aware the affection scope of the redeployment. Then they can redeploy the dependent ones by hand. However, this requires human interaction can be strenuous and even error-prone, especially for large-scale application. Alternatively, they can also redeploy all the modulars for that application. But it is not only unnecessary but also cause the state of unaffected part to be destructed (give one example here ....).

In this paper, we introduce a general approach to enable the correct hot deployment respecting the dependencies among modulars. The dependencies are constructed with class loaders for the modulars and upon the dependent modulars can be calculated and their hot deployment can be triggered. ...

In order to update flexibly and efficiently, we put dependency management and maintenance in module class loaders. Dependency reconstruction will be done in the class loaders, 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.

The rest of this paper is organized as follow. Section II analyzes reasons of failures, introduces a technology of hot deployment with dependency reconstruction. Section III discuss disadvantages of dependency reconstruction in application servers and describes our approach of dependency reconstruction in module class loaders. We introduce the design and implementation of class loaders in Section IV. Section V gives the evaluations. Section VI is the related work. Lastly in Section VII, we conclude this paper and discuss future work.

II. Background

In this section, some technical details are described, including dependency and hot deployment. Moreover, we give reasons why dependency may be lost and failures may happen when running modular applications in a mainstream application server.

A. Technical Details

1) Dependencies in Modular Applications: A modular application may contain plenty of modules and a module may contain plenty of classes. Dependencies in modular applications can be treated as dependencies of its different modules. The module is defined as $M =< C, R >$, which means a set of classes $C = \{c_1, c_2, c_3 \cdots\}$ and a set of related modules $R = \{M_1, M_2, M_3 \cdots\}$. We use it to describe dependencies of modules. And dependency R of module M can be created in the following way.

$$R = \{M_1 =< C_1, R_1 >| \ \exists c_1 \in C, c_2 \in C_1, c_1 \to c_2\}$$

The "$\to$" means import, that is to say, class c_1 in class set of module M may inherit, create or invoke class c_2 in class set of module M_1. So if $R = \emptyset$, M doesn't depends any modules. Otherwise, M depends all of modules in R.

2) Hot Deployment and Class Loading Mechanism: In the J2EE[10] field, mainstream application servers, such as JBossAS, have abilities to hot deploy and hot undeploy. When applications running in an application server need to update, hot deployment technology makes server not restart to guarantee the operation of other applications. In order to pursue dynamics of programs and hot deployment, standard class loader[11] architecture must be destroyed. That is because it has a main (share) class loader to load all of the application classes. Updating online some applications leads to the update of the main class loader, so that other applications will be affected. The new architecture using module class loader chain can support hot deployment of applications and it is used in many mainstream application servers. Fig.1 gives an overview of class loader architecture that supports hot deployment. Delegator class loader delegates classes to a class loader chain. In this chain, each module has its own class loader instances and cache. Under this mechanism, each module class loader is independent. When hot deploying, the corresponding module class loader in the chain will be removed and a new version of the module class loader will be created. In this way, application servers ensure consistency of the application versions and complete hot deployment of applications[12].

B. Reasons of Failures

The current application servers have hot deployment capabilities using class loader architecture in Fig.1. If a single module as an application deploys and updates online, module class loader will also update with it and application will be running regularly. However, when there are dependencies between the modular applications which need update depended modules, call failure will occur. Module class loaders in the class loader chain will load classes from each module separately. When only a depended module updating online, its original module class loader will be replaced by a new module class loader. But the module class loader of dependent module still exists, and the module it depends is still an old version which has been removed. In this case, dependency loss occurs and calling the depended module again will result in call failure.

Fig.2 is an example of an easy modular application and shows details of loading classes. Developers can write A.java and B.java to implement Class A and Class B easily and use

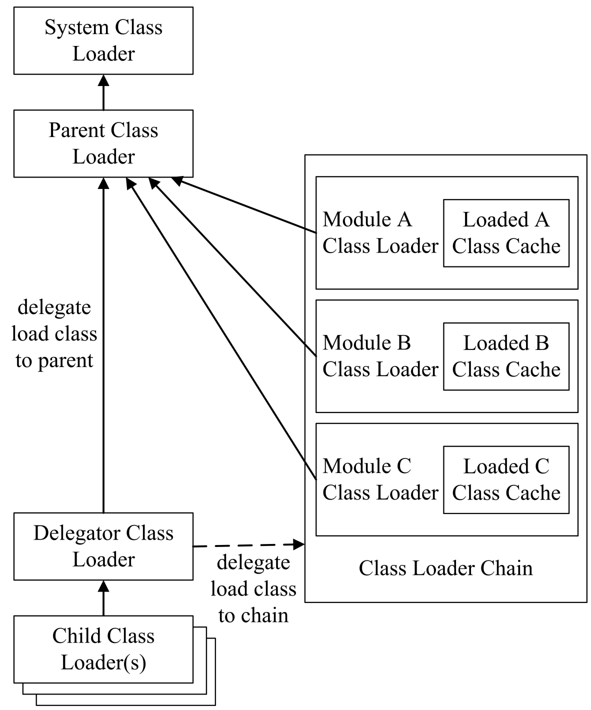

Fig. 1. class loader architecture with hot deployment

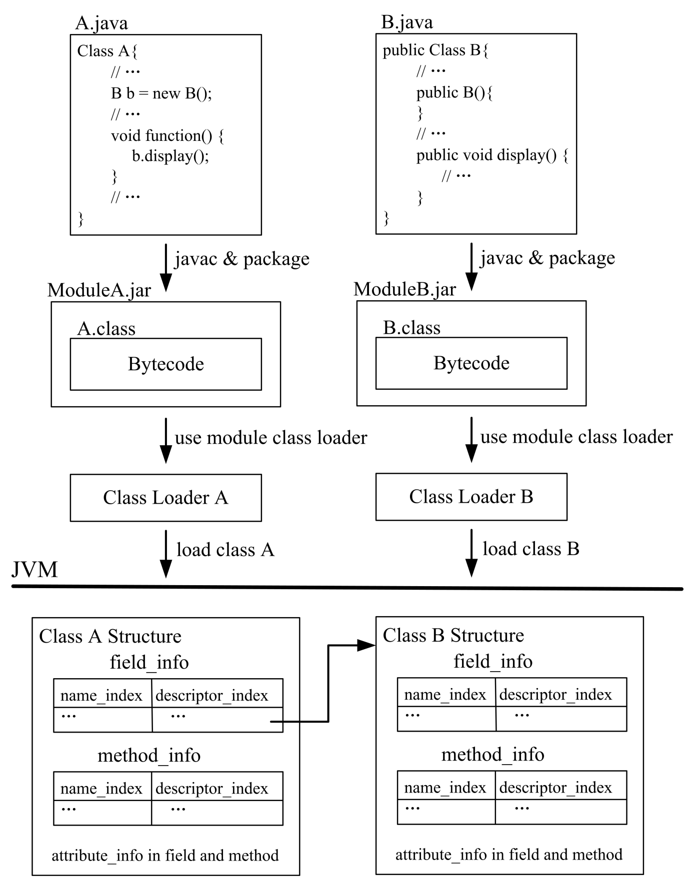

Fig. 2. details of loading classes

javac command to interpret them into bytecode. Usually, two class files are packaged into one jar package as a standalone application. As shown in Fig.2, we put class file A.class into ModuleA.jar and put class file B.class into ModuleB.jar in order to construct a modular application. When two modules are deployed, the classes in the modules are loaded by module class loaders. Hot deployment required that every module must has its own class loader instance, so that Class Loader A loads Class A in ModuleA.jar while Class Loader B loads Class B in ModuleB.jar. From the source code A.java, Class B is created and referenced by Class A. This is a dependency which we defined before. In this modular application, ModuleA.jar is a dependent module and ModuleB.jar is a depended one.

After class loading, the information of Class A and Class B is resolved and is stored in their own class structure in the Java virtual machine(JVM). JVM is the cornerstone of the Java platforms. It is the component of the technology responsible for its hardware and operating system independence, the small size of its compiled code, and its ability to protect users from malicious programs[13]. Here we focus on several class information stored in the class structure. Field table which labeled field_info is used to describe the variables declared in the class. Method table which labeled method_info is used to describe the methods of the class. These two tables have many attributes. Some attributes are described in the attribute table labeled attribute_info. But our concern is the index marked name_index and descriptor_index. In short, we can find the field name b and field type B from the index in the field_info of class structure A. The name_index points to the constant pool and the descriptor_index points to the referenced class structure if this variable is a reference type. So, the classes in different modules are associated in the JVM and the JVM can ensure the normal operation of the modular application.

When only the depended module ModuleB.jar needs to be updated, ModuleB.jar will be redeployed with the new version of Class B. The new version of Class B may modify some implementation of methods and the modification doesn't affect implementation of Class A. In this case, Class A is still the original version and ModuleA.jar won't be updated. From the JVM specification[14], a class loader cannot load a class twice, which means we cannot use the original Class Loader B to load the new version of Class B. We should update Class Loader B and use this new class loader to load the new version of Class B. Obviously, the class structure of Class B in JVM is also updated. At this moment, the dependency that JVM creates at the first deployment is lost. The index of original class structure of Class A still points to the original class structure of Class B which is old version and will be removed after redeployment of ModulueB.jar with the new version of Class B. That is why call failure occurs when executing the method that calls the method of Class B.

III. APPROACH

According to the reasons of failures, reconstructing the lost dependencies is the key to support hot deployment of modular applications. The core idea of dependency reconstruction is to ensure consistency of modules and their dependencies. The consistency is not only for the updating modules, but also for the original modules if they depend on the updating modules. To find out the dependent modules when hot deployment happens, we need to manage and maintain all the dependencies when modular applications are deployed at the first time. This section gives our approach to dependency reconstruction.

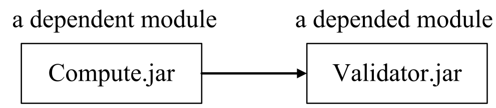

Fig. 3. an example of an EJB modular application

A. Reconstruction in Application Servers

In the process of application development, it is very common that modular applications split into separate components and they are deployed in the application server at the same time. Fig.3 is such an example of an easy modular application with dependency injection[15], which has two EJB[16] components deployed in JBossAS 5.1. The dependent module Compute.jar is a session bean for computing division. The depended module Validator.jar is another session bean to verify the divisor is zero or not. The dependency reflects in the divisor must be check whether it is zero when calculating the division of two numbers. Suppose that Validator.jar needs to be redeploy by a new version and Compute.jar makes no change. In this case, hot deployment in JBossAS 5.1 will result in the exception of the EJB application as the reasons that we discussed before.

To implement dependency reconstruction, we need to acquire deployment process of JBossAS. Briefly, from the beginning to the end of the deployment process, an EJB module will be handled by scanner, main deployer, sub deployer, dependency injector and unified class loader[17]. The dependency injector is used to manage dependency injection like the example application that we give. The unified class loader is actually classified by module class loader, so each module corresponds to a module class loader instance. From the source code of the EJB application in Fig.4, ComputeBean in Compute.jar injects ValidatorBean in Validator.jar by the dependency injection annotation @EJB and the mapped name ValidatorBean/remote. Through the dependency injector, the deployment unit context of Compute.jar can record this dependency injection and manage the dependencies all the time. Because of dependencies, the depended module must be deployed and loaded before the dependent module. Finally, all of them finish deployment.

Fig.5 is a logical block diagram of hot deployment with dependency reconstruction. To reconstruct dependencies, if depended modules are redeployed, the dependent modules must be redeployed too. This is our core idea of solving the problem. We modify some source codes of JBossAS 5.1 to improve its hot deployment capabilities and extend the deployment process by two steps. Firstly, a reverse dependency mapping table is created when an EJB module is deployed at the first time. Secondly, when an EJB module is redeployed, we look for path of the dependent EJB modules according to the reverse dependency mapping table and redeploy them at once. In this semi-automatic way, we reconstruct dependencies simply and ensure the correctness of applications after hot deployment.

Although reconstruction in application servers can solve the problem of dependency loss and call failure, there are

```
// source code ComputeBean.java in the module Compute.jar
@Stateless
public class ComputeBean implements Compute {
    @EJB(mappedName="ValidatorBean/remote")
    Validator validator;
    @Override
    public double compute(double n1, double n2)throws EJBException {
        double result;
        if(!validator.isZero(n2)){
            result = n1/n2;
            return result;
        }else
            throw new EJBException("ZERO_EXCEPTION!");
    }
}

// source code ValidatorBean.java in the module Validator.jar
@Stateless
public class ValidatorBean implements Validator {
    @Override
    public boolean isZero(double num) {
    if(Math.abs(num) < 0.000001)
        return true;
    else
        return false;
    }
}
```

Fig. 4. the main source code of two session beans

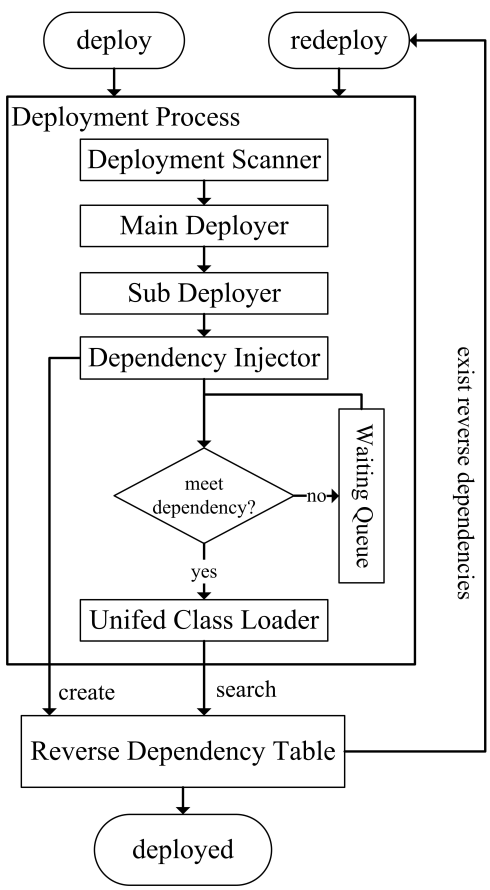

Fig. 5. a lgical block diagram of JBossAS 5.1 extension

some disadvantages in the extension. According to the define of dependency, not all the dependencies can be obtained from deployment process of the JBossAS 5.1 extension and be recorded in the reverse dependency mapping table. Dependencies of inheritance or reference cannot be detected by dependency injector. So JBossAS 5.1 extension 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. Every dependent module needs to be redeployed, that means it 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 reconstruct dependencies. This causes application servers inefficient and also causes hot deployment inflexibility.

B. Reconstruction in Class Loaders

Application servers can manage dependencies of modules via the reverse dependency table. when hot deployment occurs, the reverse dependency table helps application servers to search the dependent modules whose dependencies are lost. However, these modules are still original versions and they don't make any changes. Most of the work in redeploying dependent modules makes no sense except class loading in the modules. Dependency management and reconstruction should be handled in the class loader. Then dependent modules just need to reload the classes to reconstruct dependencies and most of the deployment process is omitted. Updating efficiency is greatly improved.

Because each module has its own class loader, the dependencies of modules are just the dependencies of class loaders. Each class loader that represents each module should record which class loaders it depends on. At the same time, the class loader of the depended module should record what depends on itself as the reverse dependencies. So, each class loader may have one or more depended class loaders and dependent class loaders. All the information is collected when classes are loaded at the first time and their class loaders maintenance these important dependency information. From an overall perspective, these module class loaders constitute a dependency graph. Every vertex of this graph can be regard as a class loader or a related module and every edge represents a dependency from one class loader to another. Due to the safety of system classes, such as String, Object and so on, they require system class loader or even bootstrap class loader to load them. 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. The last is our module class loader which is defined by us. This is parent delegation mechanism[18] to ensure the safety and uniqueness of system classes.

Generally, there should be a parent class loader called system class loader in each module class loader. This parent class loader can be treated as a depended class loader because the original class really depends on the system class which loaded by its parent class loader. Based on parent delegation and our dependency graph for module class loaders, our class loader architecture model is organized in both a class loader hierarchy and a class loader network[19].

If 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 A.jar is updated, in order to reconstruct dependencies, all of modules which depends on

```
<?xml version="1.0" encoding="GB2312"?>
<Result>
        <MainClasses>
                <MainClass>NodeAMain.class</MainClass>
        </MainClasses>
        <Dependencies>
                <Dependency>B-JAR</Dependency>
                <Dependency>C-JAR</Dependency>
        </Dependencies>
        <InvokeMainNodes>
                <InvokeMainNode>A-JAR</InvokeMainNode>
        </InvokeMainNodes>
</Result>
```

Fig. 6. A-JAR.xml file

A.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 A.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.

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 in order to 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 of the hot deployment. The hot deployment becomes more flexible.

IV. Implementation

As we know, class loader is an important concept in Java projects and application servers based on J2EE. We implement to three safe and reliable module class loaders to support hot deployment of dependency reconstruction. They are Circle Dependency Reconstruction Class Loader(CDR Class Loader) with static dependency management, Adaptive Dependency Reconstruction Class Loader(ADR Class Loader) and Adaptive Dependency Reconstruction Delay Class Loader(ADRD Class Loader) with dynamic dependency management. Our customized class loaders not also complete basic class loading, but also maintain and reconstruct dependencies. Thus, hot deployment and dependency reconstruction becomes more flexible and efficient.

A. CDR Class Loader

Static dependency management is based on profiles of deployment nodes. The profile need to describe dependencies of the deployment node. So dependency graph is created by CDR Class Loader according all of profiles in the deployment nodes. Profiles in deployment nodes are used to describe the module information including dependencies and main classes. To manage the deployment node with a unified identification, the profile is named after the deployment node where it is placed. For example, a profile named A-JAR.xml in the deployment node A-JAR is shown in Fig.6. Algorithm 1 shows CDR Class Loader how to load classes. The class loader will try the dependent class loaders to load classes at first. Because the system class loader is the dependent class loader of every module class loader, it will load the system classes at last. The dependency graph probably has a circle so that there must be circle detection in the algorithm of loading classes.

Algorithm 1 function loadClass of CDR 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

Reconstruction process is similar to the process of creating dependency graph. The graph and CDR Class Loaders are updated by the new version of deployment nodes profiles before the loading of classes. For instance, if A.jar is updated, in order to reconstruct dependencies, all of modules which depends on A.jar also require to be updated even though they have no new versions and users don't want to update them. In this way, we reconstruct the dependencies for A.jar. But reconstruction is not finished because these dependent modules may be depended on other modules. Their dependencies need to be reconstructed too in the same way, until there is no updated modules are depend on any of the updated modules. We maintenance and reconstruct dependencies in the CDR Class Loaders by the static way.

B. ADR Class Loader

Editing profiles of deployment nodes is a quite messy work when modular applications constitute 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 node 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) 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.

With more classes loaded successfully, more class loaders will be put in the repository and they help to load other classes. 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.

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. The idea of 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. It is obvious that adaptive way to load class is time consuming. But avoiding editing dependencies in the profiles of modules is a time saving thing for developers of modular applications or administrators of class loading platform.

C. ADRD Class Loader

Adaptive Dependency Reconstruction Delay Class Loader(ADRD Class Loader) inherits Adaptive Dependency Reconstruction Class Loader(ADR Class Loader). Comparing with ADR Class Loader, ADRD Class Loader delays dependency reconstruction when hot deployment happens. All the affected class loaders only mark as invalid class loaders and none of class loaders are updated. So, dependency reconstruction will happen when applications are executed. 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 ADR Class Loader. Similarly, dependency reconstruction process is also as the same as the reconstruction process of ADR Class Loader. Their main difference is their reconstruction time. ADRD 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.

V. Evaluation

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

Algorithm 2 function loadClass of ADRD Class Loader

Input:
  Fully qualified name of the class: $name$
Output:
  The loaded class instance: c
$c \leftarrow$*findLoadedClass*$(name)$
if $c = null$ **then**
  if $name.startWith$(”java.”) **then**
    $c \leftarrow systemClassLoader.loadClass(name)$
    if $c \neq null$ **then**
      return c
  else
    $c \leftarrow$*findClass*$(name)$
    if $c = null$ **then**
      for each class loader dep in $depArrayList$ **do**
        $c \leftarrow dep.loadClass(name)$
        if $c \neq null$ **then**
          return c
      for each class loader cl in $repository$ **do**
        if $cl.valid$ = false **then**
          $cl \leftarrow$*reconstruct*(cl)
          $cl.valid \leftarrow$true
        $c \leftarrow cl.loadClass(name)$
        if $c \neq null$ **then**
          $depArrayList.add(cl)$
          return c
    else
      return c
else
  return c

A. Correctness

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.3 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.

TABLE I. TIME COST OF REDEPLOYMENT

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

B. Efficiency

Comparing with JBoss AS 5.1.0, the application server extension needs more time to hot deploy modules because the server searches dependencies in the reverse dependency table and deploys more dependent modules. But this kind 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 the call 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[20], 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. Most testing modules are services of JBoss AS, which can be found in the examples directory of JBoss AS 5.1.0 Release. Compute.jar and Validator.jar are two session beans which constitute an example application of division in Fig. 3. Java Pet Store[21] is a sample well-known application from the J2EE, so it is also one of our subjects. 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 CDR Class Loader based on static dependency management, ADR Class Loader and ADRD Class Loader can obtain the dependencies of modules in the dynamic 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. ADR Class Loader and ADRD 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 ADR 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, ADRD 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 ADR Class Loader, ADRD Class Loader delays reconstruction and increase the flexibility of reconstruction.

In a particular application scenario, ADRD Class Loader has a good performance. We compare performance of flexible reconstruction between ADR Class Loader and ADRD Class

TABLE II. PERFORMANCE OF FLEXIBILE RECONSTRUCTION

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

Loader through experiments. We choose a simple 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 in Fig.6. 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 ADRD Class Loader of this application modules will complete reconstruction. So, the time of hot deployment and execution decreases in Table II.

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[22]–[25]. 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[26]–[28]. 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[29] and they even proposed an approach to make an application server well-structured and dynamic[30]. And some research focused on refactoring application servers according to application requirements[31].

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[32] or a module splits into several ones in order to reduce the cost of update[33]. 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. Experiment shows the problem 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[34], 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] Parnas, David Lorge. ”A technique for software module specification with examples.” Communications of the ACM 15.5 (1972): 330-336.

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

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

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

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

[6] 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.

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

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

[9] 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.

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

[11] Taylor, B. ”Java Class Loading: The Basics.” (2003).

[12] 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.

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

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

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

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

[17] 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.

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.