Int J Software Informatics, Vol.2, No.1, July 2008, pp. 17–31 E-mail: ijsi@iscas.ac.cn
International Journal of Software and Informatics ISSN 1673-7288 http://www.ijsi.org
 Tel: +86-10-62661040

Achieving Flexibility in Off-the-Shelf Middleware Services Integration*

Yan Li, Minghui Zhou, Donggang Cao, Lu Zhang, Hong Mei

(Software Institute, School of Electronics Engineering and Computer Science, Peking University, China)

(Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education, Beijing 100871, China)

E-mail: {liyan05, zhmh, caodg, zhanglu}@sei.pku.edu.cn, meih@pku.edu.cn

Abstract The development of component-based software engineering enables the construction of application servers by integrating reliable OTS middleware services. However it is difficult to achieve flexibility in conventional hard coding way. In this paper, we propose a flexible OTS middleware services integration framework to address this problem. In this framework, we define two kinds of modules: the *middleware service contract module* to represent the stable contract which specifies the abstract interaction logic between the application server and a kind of middleware services, and the *middleware service implementation module* to encapsulate the mutable implementation details of different OTS middleware services in a unified way. Additionally, we propose a *module management mechanism* to enable the application server to replace the OTS products at runtime via configuration. We implement the framework in a J2EE application server, and the evaluations show that our framework effectively reduces the cost and the time of maintaining and customizing the OTS middleware services-based application server.

Key words: flexibility; middleware service; integration; off-the-shelf

Li Y, Zhou MH, Cao DG, Zhang L, Mei H. Achieving flexibility in off-the-shelf middleware services integration. *Int J Software Informatics*, **2008, 2(1): 17–31. http://www.ijsi.org/1673-7288/2/17.pdf**

1 Introduction

Until now, J2EE application servers are becoming gigantic and complex like never before. Take RedHat Jboss[1], a widely used Java application server, for example, the size of it has increased three times (from less than 30MB at version 3.0.0 to more than 100MB at version 5.0.0), and the number of middleware services it provides has reached to 20, most of them are complex, such as EJB/Web container and transaction service. Component-based software engineering[2] is a good solution to optimize

* The research was sponsored by the National Grand Fundamental Research 973 Program of China under Grant No.2002CB312003, the National Nature Science Foundation of China under Grant Nos.60603038, 60503029, and the National High-Tech Research and Development Plan of China under Grant Nos.2007AA01Z133, 2006AA01Z156, 2006AA01Z189.
Corresponding author: Minghui Zhou, zhmh@sei.pku.edu.cn
Manuscript received 28 Sept. 2007; revised 4 Jul. 2008; accepted 28 Jul. 2008; published online 1 Aug. 2008.

the time and the cost of the application server construction, therefore rather than enabling all the features themselves, more and more application server vendors are inclined to selectively implement some middleware services on the one side, such as the EJB container, and to integrate some reliable Off-the-Shelf (OTS) middleware services on the other side, such as ObjectWeb JOTM[3] for transaction service, Apache Tomcat[4] for Web container, which is called *OTS middleware service integration*. (In the context of this paper, we use the terms "OTS middleware service", "OTS product" interchangeably.)

However, conventional integration approach usually hard codes the concrete OTS middleware services. Because the OTS products are continually evolving by third-parties, hard-coding approach will burden the application server developers with onerous maintenance work. Take Apache Tomcat for example, it has released 22 different versions containing 4 milestones from late 2004 to 2007.A minor change of the integrated OTS product demands the developers to review and modify all the related codes scattered in the application server, and either the large scale of related codes or the deficiency of development document make the maintenance harder. Moreover, the application servers are bound with fixed OTS middleware services, which can not meet the various needs of different applications. For example, sometimes the speed of transaction service is their first concern, while sometimes it is better to choose a slower but less memory consumed one. For these reasons, it is very difficult to achieve flexibility in the construction of application servers through assembling together different OTS products.

Flexibility relates to the range of possible changes that can be supported by a platform[5]. In the OTS middleware service integration, flexibility can in turn be refined into the requirements supported by the application server: (1) ease of modification for the evolvement of OTS products; (2) ease of OTS products substitution.

To address the flexibility, we propose an OTS middleware services integration framework. According to the information hiding principle[6], we define two kinds of modules: the *middleware service contract module* to represent the stable contract which specifies the abstract interaction logic between the application server and a kind of middleware services, and the *middleware service implementation module* to encapsulate the mutable implementation details of different OTS middleware services in a unified way. Additionally, we propose a *module management mechanism* to enable the application server to replace the OTS products at runtime via configuration.

The main contributions of this framework are as follow. First, we implement the framework in a J2EE application server called PeKing University Application Server (PKUAS)[7] to demonstrate its feasibility and effectiveness in constructing the complex application server. Second, it effectively reduces the cost and the time of maintaining the OTS middleware services-based application server. Through comparing two versions of PKUAS (with and without the framework), we found the numbers of classes needed changing during the modification or substitution of OTS products have been greatly reduced (the best result is from 53 classes to 5 classes). Third, it allows the application server vendor rapidly customizing the application server to better meet the diverse application requirements through configuration.

The rest of the paper is organized as follows: Section 2 introduces the integration framework. Section 3 describes the implementation of the framework in PKUAS.

Section 4 gives the evaluation of this framework. Section 5 summarizes the related work. Finally, section 6 concludes the paper and discusses the further work.

2 The Middleware Service Integration Framework

The goal of the integration framework is to achieve flexibility in the middleware service integration. David Parnas has discussed in[6,8] that modularization is a mechanism for improving the flexibility in complex system construction, and the only assumptions that should appear in the interfaces between modules are those that are considered unlikely to change. So to achieve this goal, the diverse middleware service implementations should be modularized in a unified form, which not only to hide the implementation details, but also to enable the application sever to manage (e.g. load, configure, and so on) them through unified operations. Besides, explicit and stable contracts for invoking the middleware services should be defined to decouple any product implementation-specific logic from the logic of the application server.

The framework overview is shown in Fig.1. Each middleware service implementation is wrapped in a unified form: middleware service implementation (abbr. MSI) module (Section 2.1.1). To avoid tangling the product implementation-specific logic with the logic of the application server, the framework defines a contract for each kind of middleware service, and encapsulates it into the middleware service contract (abbr. MSC) module (Section 2.1.2). The dependencies among middleware services should comply with the contracts. The module management mechanism (Section 2.2) is responsible for loading, instantiating, registering the MSC modules, and configuring the appointed middleware service implementation for each kind of middleware services (MS Manager). The runtime behaviors of the framework are described in Section 2.3.

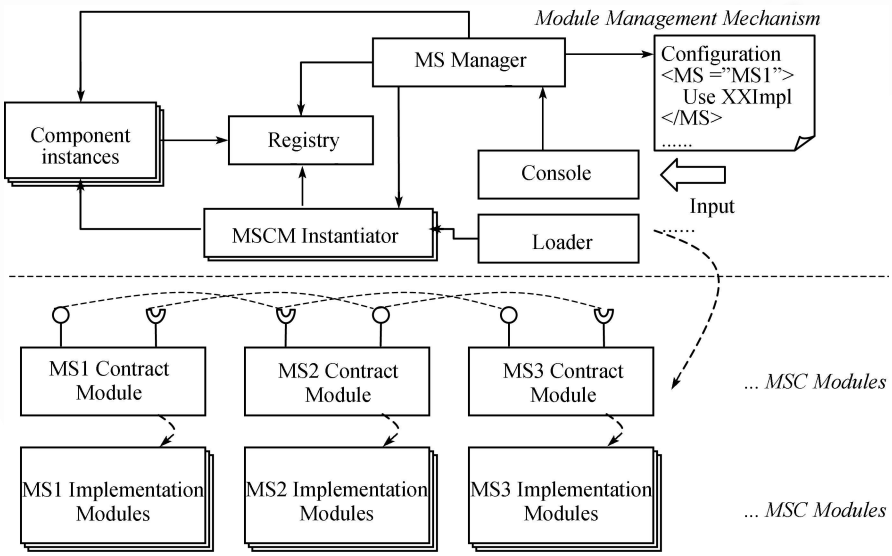

Figure 1. Framework overview

2.1 Modules

2.1.1 Middleware Service Implementation Module

Each middleware service implementation module (abbr. MSI module) corresponds to a middleware service implementation. It is composed of a set of adapters and one middleware service implementation, shown in Fig.2.

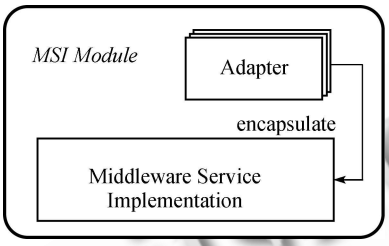

Figure 2. The structure of middleware service implementation module

As the middleware service contract (Section 2.1.2) may differ from the APIs of a middleware service implementation, adapters are employed to adapt the APIs to the contract. Because the APIs (of an OTS product) only describe the functionality of the component and provide no insights for adapting the component[9], the application server developers need first to do an operation mapping between the contract interfaces and the API. We classify the issues occurring during the mapping practices into three kinds of mismatch named *name mismatch*, *function mismatch*, and *function deficiency*. The definitions are listed in Table 1. According to the mapping results, different adaptation strategies are employed: As for name mismatch, the adapter directly forwards the invocation to the corresponding OTS product's API. As for function mismatch, the adapter either wraps the API (e.g., add the pre/post processing parts) or invokes the related APIs in a sequence. As for function deficiency, the adapter must implement the function itself.

Table 1 Three kinds of mismatch

Kind	Definition
Name mismatch	The API rightly corresponds to the operation M in the contract, but only the names of the operations do not match.
Function mismatch	The API is incomplete to the operations in the contract, e.g., application server has to do pre or post processing for the operations in the contract. Or operation M in the contract should be implemented by a sequence of the operations in the APIs.
Function deficiency	No operation in the API supports operation M in the contract.

The MSI module brings two advantages: first, it hides the complex details of various middleware service implementations from other parts of the application server, by wrapping it to a separate module. Second, it reduces the maintenance workload of the OTS middleware service integration, by confining the possible changes to the adapters whenever the third parties upgrade their products.

2.1.2 Middleware Service Contract Module

Each middleware service contract module (abbr. MSC module) corresponds to a kind of middleware service. It is composed of an entry component, a metadata file, and a set of contract interfaces for this kind of middleware service, shown in Fig.3.

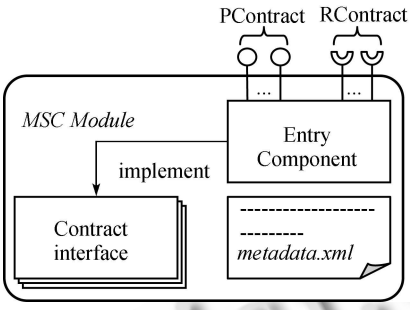

Figure 3. The structure of middleware service contract module

The contract interfaces are defined by the application server developer from the perspective of the application server rather than a concrete OTS product, that is, the contract interfaces exactly cover the functionalities required by the application server to invoke and control this kind of middleware service, while reveal little about the concrete implementation. When defining the contract interfaces, application server developers had better first investigate the specifications related to this kind of middleware service. If the Service Provider Interface (SPI) has already existed, the contract interfaces should be consistent with it. Then, the developers define the application server specific operations for this kind of middleware service. Until now, the majority of SPIs have not been thoroughly prescribed: they may either scatter in several specifications or even not be defined. It is recommended that the developer should be careful when defining server-specific operations, because it may not be supported by all the OTS products, which eventually brings much burden to the adapter implementations. The evolutions of the specifications and the application server itself may result in the modification of the contract interfaces, but they are considered to be stable most of the time.

As the entrance of the MSC module, the entry component is responsible for not only instantiating the configured MSI module via the Java reflection mechanism, but also routing the invocations between the MSI module and other parts of the application server. On the one hand, the entry component implements some of the contract interfaces (PContract), and processes the requests from the external modules through invoking the configured MSI module. On the other hand, the entry component expresses the required contracts (RContract) of this kind of middleware service, and forwards the requests of the MSI module to the external modules.

The metadata file contains the name of the MSC module and the implementation class of the entry component, the contract interfaces it provides and requires, and the component properties (e.g., the properties about the OTS product).

The MSC module brings two advantages: first, well defined contract interfaces release the application server from headachy integration maintenance whenever the OTS products update. In the conventional hard-coding integration, the developer had to review and modify each part depending on the OTS product in the application server, which is not only time-consuming but also error-prone with the increasing

number of OTS middleware services (think of 5 OTS middleware services and each with average 20 related parts in the application server), and the constantly growing size of the application server (think of 1000+ classes which is ordinary for some popular application servers, e.g. Redhat JBoss). In contrast, now the application server interacts with the OTS products through the relative stable contract interfaces. The adapters, which implement the contract interfaces in an MSI module, instead of the whole application server need to be modified along with the OTS products update. That is, the scale of modification is effectively reduced. Second, through organizing the contract interfaces into a separate module rather than combining with the middleware service implementations, it alleviates the work of OTS middleware service substitution to a large extent. The application server no longer binds to a specific one at its code level but to the MSC module, and the Java reflection mechanism employed in the entry component makes it possible to substitute the OTS product via configuration at runtime, rather than modify a great deal of classes and restart the application server.

2.2 Module Management Mechanism

The framework contains several components to execute the management tasks: the loader loads the classes of the MSI and MSC modules; the MSCM (Middleware Service Contract Module) Instantiators instantiate the entry components in the corresponding MSC modules; the registry processes the instances registration and lookup issues; the MS (Middleware Service) manager is in charge of configuring the middleware services; the console is used to receive the commands from the application server administrator, such as replacing an existing OTS product with another one at runtime, and then the console forwards the commands to the MS manager.

The configuration file prescribes the specific OTS middleware services to be used. For each kind of middleware service, the configuration should specify the name of the middleware service's entry component, the property values of the entry component, and so on (refer to List.3 for more details). Generally, there is a default configuration file.

2.3 Framework Runtime Behaviors

To make clear how the framework works, we illustrate two types of runtime behaviors of the framework in this section: the behaviors when bootstrapping and the behaviors when substituting an OTS product.

At bootstrap, there are seven steps, as shown in Fig. 4:

1. The loader loads every MSC module (step 1.1), creates an MSCM Instantiator for it (step 1.2). The MSCM Instantiator will register itself with the name of the MSC module to the registry in order to make the MS manager find them (step 1.3).

2. The MS manager parses the configuration file and stores the configuration (step 2).

3. To instantiate the entry component in each MSC module, the MS manager first looks up the module's MSCM Instantiator from the registry (step 3.1), and asks factory to create an instance of the entry component (step 3.2∼3.3).

4. The MS manager configures the MSC module with the configuration loaded in step 2, that is, the instance of the entry component is notified with the appointed OTS product at that time. This is achieved by setting the properties value to the instance (step 4).

5. After configured, the MSC module requests the loader to load the appointed MSI module (step 5). If the MSI module is successful loaded, the MSC module is valid, that is, it can be used by other MSC modules.

6. As the instance of the entry component registers itself with its provided contract to the registry, so that the registry could inject it to the other entry components that require this contract (step 6).

7. At last, the registry binds the MSC modules according their provided and required contract (step 7). Dependency injection[27] technique is employed here. At runtime, the middleware service invokes each other via the predefined contract interfaces, and the entry component forwards the invocation to the appointed OTS product.

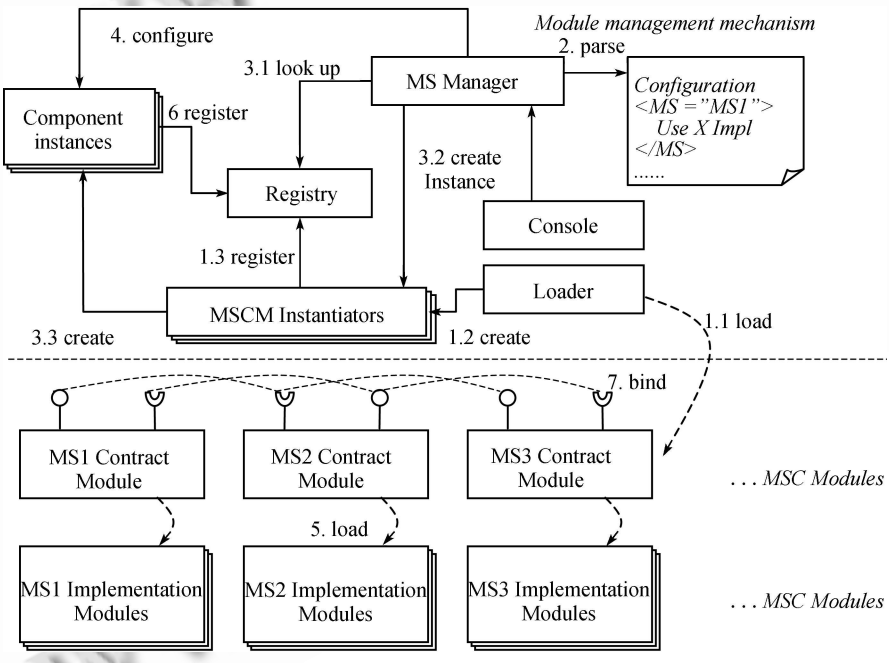

Figure 4. Runtime behaviors at bootstrap

When the application server administrator needs to substitute an OTS product due to the variation of the application requirements, the following steps are executed, shown in Fig. 5:

1. After the administrator configures a new OTS product from the console, the console calls the MS manager (step 1).

2. The MS manager looks up the component instance of this middleware service (step 2.1), and configures the component instance with the new configuration (step 2.2).

3. The MSC module requests the loader to first load the new MSI module (step 3.1) and then to unload the old MSI module (step 3.2).

4. Right after the MSC module becomes valid again, the registry binds it with other MSC modules.

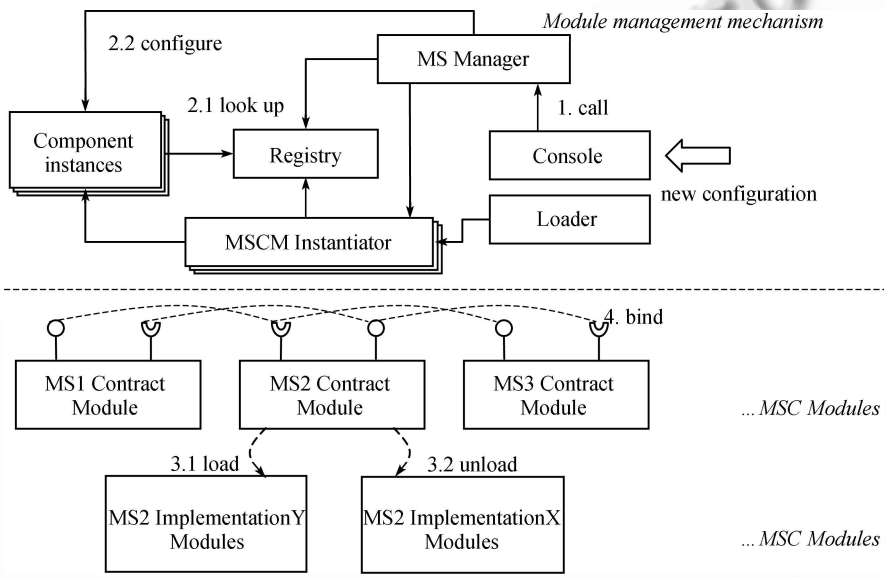

Figure 5. Runtime behaviors when substituting an OTS product

3 Implementation

PKUAS, developed by Peking University, is an open source J2EE-compliant application server[10]. It develops some middleware services itself, such as EJB container, data source and transaction service, while at the same time also integrates some OTS middleware services, such as ObjectWeb JOTM for transaction service. However, in its old version, the integration was done in hard coding way, so the logic of PKUAS itself was interweaved with that of OTS middleware services, which led to two annoying problems: first, the high maintenance cost of the integration which has been discussed in Section 2.3. Second, the customization deficiency: nowadays applications often want to customize the kinds and the implementations of middleware services to exactly fit their needs, rather than accept all the features that the vendor provides. However, as specific OTS middleware services were hard coded in PKUAS, implementations substitution is not supported.

Due to the problems, PKUAS has implemented the integration framework, shown in Fig. 6. The module management mechanism is implemented in its kernel. The middleware service implementations, ObjectWeb JORAM[11], Codehaus ActiveMQ[12] and SONIC OpenJMS[13] for message service, PKUAS Transaction, ObjectWeb JOTM for transaction service, and Apache Tomcat and Mort Bay Consulting Jetty[14] for

Web container, and PKUAS EJB container, are encapsulated in MSI modules respectively. Besides the classes of the OTS product, one MSI module also contains the adapters developed by PKUAS for this OTS product. These adapters implement the contract interfaces to allow the MSC module invoking this OTS product when receiving requests coming from other middleware services, such as EJB container. For each kind of middleware service, PKUAS develops an MSC module. Each MSC module contains three parts: the contract interfaces, the entry component, and the metadata file. To give a concrete view of the MSC module, we take the transaction service for example to describe each part of MSC module, which is similar to other MSC module.

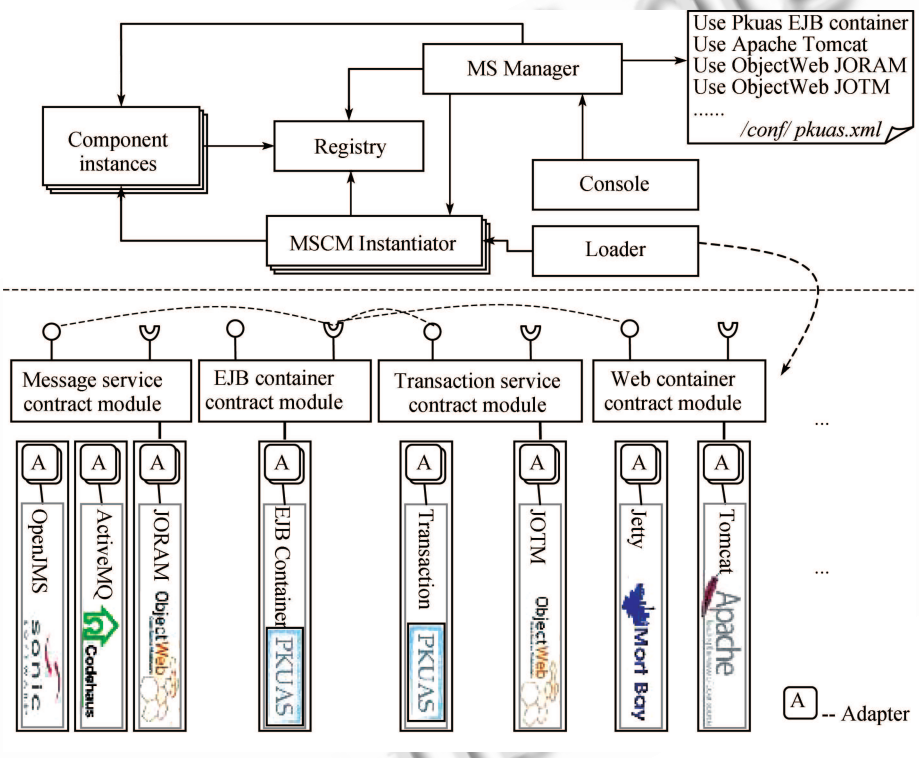

Figure 6. Framework overview in PKUAS*

As the contract interfaces should be consistent with the specifications, the transaction contract interfaces are derived from the Java EE 5 and the Java Transaction Service (JTS)[15] specification. The entry component implements the interfaces as its PContract (See Section 2.1.2) for other middleware services. List.1 is the contract interface for transaction service. However, other middleware services (e.g. message service) may have more than one contract interfaces.

* For the clarity and the space limited, only the dependencies between EJB container and other middleware service are shown in this figure, however, there are many dependencies between middleware services, such as data source service depends on transaction service.

```
package pku.as.contract.transaction
public interface TransactionService{
   public javax.transaction.TransactionManager getTransactionManager();
   public javax.transaction.Transaction
                        getTransactionByXid(javax.transaction.xa.Xid xid);
   public javax.transaction.xa.Xid getXid (javax.transaction.Transaction tx);
}
```

List. 1. Transaction service contract interface

List. 2 is the metadata file for the transaction service contract module. Most of the metadata files are comprised of three parts: the class of the entry component must be declared with the ¡component¿ tag; The PContract and the RContract are marked with ¡provides¿ and ¡requires¿ tags respectively, and the RContract is optimal; The properties used to configure the OTS product, if exist, could be announced with the ¡property¿ tag. Besides the property's name, the method used to set the property should be given.

```
<MSCM name = "TransactionService">
  <component className="pku.as.contract.transaction. TransactionService" >
    <provides
            interface="pku.as.contract.transaction.TransactionServiceInterface"/>
    <properties>
      <property name = "defaultTimeout" method="setDefaultTimeout"/>
      <property name = "implemtationProperties" method="setProperties"
                                                    type="java.lang.String"/>
    </properties>
  </component>
</MSCM>
```

List. 2. Metadata file in the MSC module

Additionally, to make the application server forward the invocations to the appointed OTS products, the administrator could configure the pkuas.xml configuration file. List. 3. is a snapshot of transaction service configuration in pkuas.xml. It employs the class name of the entry component to denote the middleware service, and the values of the properties for the appointed OTS product are given. MS manager parses this file, and then the boundary of the application server is determined at runtime.

```
...
<Service Class="pku.as.contract.transaction.TransactionService" LoaderDir="transaction" >
  <property Name="defaultTimeout" Value="6000"/>
  <Property Name="implementationProperties">
     <Set Name="pkuas.txmanager.class" Value="org.objectweb.jotm.Current"/>
     <Set Name="pkuas.tx.class" Value="org.objectweb.jotm.TransactionImpl"/>
     <Set Name="pkuas.xid2tx.method.param" Value="org.objectweb.jotm.Xid"/>
     <Set Name="pkuas.xid2tx.method" Value="getTxByXid"/>
     <Set Name="pkuas.tx2xid.method" Value="getXid"/>
  </Property>
</Service>
          ...                                                    conf/pkuas..xml
```

List. 3. PKUAS configuration file

4 Evaluation

In this section we first present an evaluation of the integration framework with respect to the flexibility discussed in Section 1. After that, we analyze the performance overhead of the framework in PKUAS.

4.1 Flexibility

As defined in the introduction section, the flexibility is the requirements of ease of modification for the evolvement of OTS products and ease of OTS products substitution. in the OTS middleware service integration. Both when modifying and substituting, the application server developers have to at least change all the classes, which refer to the OTS product's API and scatter in the application server. So we use this number of classes, named N_C, to measure the workloads of modification and substitution, that is, the flexibility.

In our evaluation, we compare the new PKUAS ($PKUAS^N$) with our original PKUAS ($PKUAS^O$) which hard codes a set of specific OTS products to comply with J2EE specification. Besides, to show the usage of the OTS products in original PKUAS is reasonable, we also count the corresponding N_C in a well-known open source application server ObjectWeb JOnAS v4.8[16], which also use these middleware services. The N_C of three OTS middleware services, which are ObjectWeb JOTM for transaction service, Object JORAM for message service and Apache Tomcat for web container, in the three application servers are shown in Table. 2.

Table 2 Numbers of classes referring to OTS middleware services

	ObjectWeb JOTM	ObjectWeb JORAM	Apache Tomcat
$PKUAS^O$	53	9	14
JOnAS v4.8	40	11	18
$PKUAS^N$	5	3	1

As shown in the table, the numbers of classes to be changed during modification and substitution have been greatly reduced. Because the MSC modules separate the application server from the concrete OTS products, only the adapters in the MSI module will use the specific OTS products' APIs, thus N_C in the new version is decreased to the number of adapter-related classes. Consequently, the application server developers just need to maintain the adapters whenever the OTS middleware services evolve, and can substitute an OTS middleware service by simply modifying the configuration.

4.2 Performance Overhead

As this integration framework inserts a set of intermediary entities (the entry components and the adapters) into the middleware service invocation process, it introduces a performance overhead into the application server. To analyze the overhead, we compared our original PKUAS application server with the new version by some performance tests. (Tests were performed on a Dell Precision 410 MT pc equipped with 1G Mb RAM and an Intel Pentium4 processor rated at 2.5G Mhz; the operating system was Microsoft's Windows XP).

As mentioned before, Apache Tomcat and ObjectWeb JOTM have been integrated in original and present PKUAS. In the experiments, we ran two test programs, each of which invoked one OTS product. Because our aim is to examine the performance overhead occurred when the OTS middleware services are invoked, we hope these test programs focus on their target middleware service and avoid interacting with other application server components. Therefore, for testing the integrated web container service (Tomcat), we chose Orientware XLinker[17]'s TestEcho web service as the server-side program, which only use the web container of the application server; and for testing the integrated transaction service (JOTM), we used the JOnAS Transaction Service conformance test suite[18]. Table.3 gives the obtained results.

Table 3 Experimental results

OTS Service	Avg. response time^N	Avg. response time^O	Performance overhead
Apache Tomcat	540 ms	532 ms	1.5%
ObjectWeb JOTM	133 s	129 s	3.1%

As shown in Table.3, about 3% performance is lost in the new version. Comparing with the benefits the integration framework brings, we believe the overhead is affordable.

5 Related Work

Considerable research work[19] has been done in component integration. Yau and Karim[20] focus on the distributed component integration by proposing a distributed component framework. Sauer et al.[21] establish a component-based approach to encapsulate and wrap the legacy system. Rather than flexibility, interoperability is their first concern. Instead of constructing an entire integration framework, Min et al.[22] focus on the connector which acts as an adapter to fill the gap between the candidate components and the specification of components required. Bastide et al.[23] present an approach to adapting component structures, but the first step of this approach is to decompose the component code, which is unfeasible for middleware service integration.

In current application servers' community, OTS middleware services integration is becoming popular. However, the integrated OTS product is generally wrapped into a component of the vendor-specific framework, while flexibility is not always mentioned. Java Management eXtentions (JMX)[24] is a popular framework to construct the application server: each OTS middleware service is wrapped as a managed bean component, named MBean, which ensures the application server to manage (e.g. life cycle management) different kinds of middleware services in the same way. However, the OTS middleware services are usually hard coded in the MBean which represents one kind of middleware service, that is, the application server is tightly bound to this OTS product. Recently, Open Service Gateway initiative (OSGi)[25], which bears some similarity with our work, emerges as a new framework to modularize the application server: each OTS middleware service is encapsulated into a bundle, but there is still lack of mechanism to separate the application server from the APIs of the OTS product. Apache Geronimo[26] proposes its own GBean framework which aims at integrating existing OTS middleware services: each OTS middleware service corresponds to one or more GBeans. Its deploy plan is similar with the configuration to

configure the OTS middleware service, but service-level contract is also not supported explicitly.

Compared to these work, our framework proposes a MSI module to insulate the other components of the application server from the implementation details of the OTS products, and based on the MSC module and module management mechanism, we realize the substitution of the OTS products through simple configuration. These make our application server flexible to the product evolution and substitution.

6 Conclusion

In this paper, we presented an OTS middleware service integration framework. The framework is composed of the middleware service implementation module, the middleware service contract module and the module management mechanism. The three parts together hide the complex details of OTS products, and make the OTS products plug into the application server with less amount of code modification, besides they enable the customization of the OTS middleware services via simple configuration. Through the flexibility the framework achieves, the maintenance and the customization cost for the application server developers is greatly reduced.

In the future, we are planning to strengthen the framework to allow application deployers to specify a set of configuration rules to support self-configuration. These rules will be evaluated at runtime to decide which OTS middleware service should be bound (e.g. "when the memory is below 512M, switch to PKUAS-Transaction"). Furthermore, we are combining this framework with OSGi technologies to make the J2EE application server gain flexibility as well as dynamism.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments on the earlier draft of this paper. We would also like to Xi Sun, and other members of the PKUAS group from the Institute of Software, Peking University. The research was sponsored by the National Grand Fundamental Research 973 Program of China under Grant No. 2002CB312003, the National Nature Science Foundation of China under Grant No. 60603038, 60503029, and the National High-Tech Research and Development Plan of China under Grant No.2007AA01Z133, No.2006AA01Z156, No. 2006AA01Z189.

References

[1] JBoss Application Server. http://www.jboss.com/products/jbossas

[2] Szyperski C. Component software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (2002).

[3] ObjectWeb JOTM. http://jotm.objectweb.org/

[4] Apache Tomcat. http://tomcat.apache.org/1. Java Platform, Enterprise Edition, http://java.sun.com/javaee

[5] Parlavantzas N, Coulson G. Designing and constructing modifiable middleware using component frameworks. The Institution of Engineering and Technology 2007 (IET Softw.2007), pp. 113–126.

[6] Parnas DL. On the criteria to be used in decomposing systems into modules. Commun. ACM, Dec. 1972, 15: 1053–1058.

[7] Mei H, Huang G. PKUAS: An architecture-based reflective component operating platform. Invited Paper, 10^{th} IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS), Suzhou, China, 26-28 May 2004, pp. 163–169.

[8] Parnas DL, Clements PC, Weiss DM. The modular structure of complex systems. IEEE Transactions on Software Engineering, March 1985, SE-11(3).

[9] Heineman GT, Ohlenbusch HM. An evaluation of component adaptation techniques. Proc. of the 2^{th} ICSE Workshop on Component-Based Software Engineering, 1999.

[10] PeKing University Application Server, http://forge.objectweb.org/projects/pkuas/

[11] ObjectWeb JORAM. http://joram.objectweb.org/

[12] Codehaus ActiveMQ. http://activemq.apache.org/

[13] SONIC OpenJMS. http://openjms.sourceforge.net/

[14] Mort Bay Consulting Jetty. http://www.mortbay.org/

[15] Java Transaction Service (JTS). http://java.sun.com/products/jts/

[16] JOnAS. http://wiki.jonas.objectweb.org/xwiki/bin/view/Main/WebHome

[17] Ge S, Hu CM, Du ZX, Wang Y, Lin XL, Huai JP. A Web service-based application supporting environment. Proc. of the National Software and Application. 2002. 97-102 (in Chinese with English abstract).

[18] JOnAS team. Easybeans testsuit, http://www.easybeans.org/doc/testguide/en/integrated/testguide.html

[19] Land R, Crnkovic I. Existing approaches to software integration–and a challenge for the future. Proc. of the Software Engineering Research and Practice in Sweden (SERPS2004). Linkoping University, 2004.

[20] Yau SS, Karin F. Integration of object-oriented software components for distributed application software development. Proc. of the 7^{th} IEEE Workshop on Future Trends of Distributed Computing Systems (TDCS'99). IEEE, 1999, pp. 111–116.

[21] Sauer LD, Clay RL. Rob armstrong, meta-component architecture for software interoperability. Proc. of the Int'l Conference on Software Methods and Tools (SMT2000). IEEE, 2000, pp 75-84.

[22] Min HG, Choi SW, Kim SD. Using smart connectors to resolve partial matching problems in COTS component acquisition. Proc. of the Int'l Symposium on Component-Based Software engineering (CBSE 2004). LNCS 3054, Springer-Verlag, Berlin Heidelberg, 2004, pp. 40–47.

[23] Bastide G, Seriai A, Oussalah M. Adaptation of monolithic software components by their trans-

formation into composite configurations based on refactoring. Proc. of the Int'l Symposium on Component-Based Software engineering (CBSE2006). LNCS 4063, Springer-Verlag, Berlin Heidelberg, 2006, pp. 368–375.

[24] Java Management eXtensions (JMX). http://java.sun.com/javase/technologies/core/mntr-mgmt /javamanagement/

[25] Open Services Gateway Initiative (OSGi). OSGi Service Platform Specification. Version 4, 2004.

[26] Apache Geronimo. http://geronimo.apache.org/

[27] Martin Fowler, Inversion of Control Containers and the Dependency Injection pattern. http://martinfowler.com/articles/injection.html, Jan. 2004