Microkernel Architecture: Making Application Servers Open to Change
Donggang Cao, Hong Mei, Qianxiang Wang, Gang Huang
Software Institute, Peking University
100871 P.R. China
{caodg, meih, wqx, huanggang}@sei.pku.edu.cn

Abstract
Application server software is required to be highly adaptive and reconfigurable so as to satisfy the changing requirements of various component-based applications in enterprise computing environment. To meet this goal, an open-to-change architecture is a must, which challenges almost all distributed system software designers. This paper describes our work on designing an adaptive J2EE application server named PKUAS. PKUAS has a microkernel based, service oriented architecture, which allows different services to be plugged into it and get managed conveniently. The PKUAS microkernel has well-defined structure that strictly separates management concern from business concern, which brings excellent modularity and extensibility to PKUAS without causing much performance degradation. The practices show that this approach can effectively make application servers open to change.

Key words: Adaptation, Microkernel, Application Server, Middleware, Architecture

1. Introduction

Application server is a kind of system software that resides between applications and underlying resources and services in enterprise computing environments. It provides runtime hosting environment for component-based applications. There are already some widely used application servers (e.g., commercial WebSphere[1] and WebLogic[2], and open-source JBoss[3] and JonAs[4]). And many other products offering similar functions are also emerging continuously.

However, enterprise computing environment is often highly complex. A typical enterprise computing environment usually consists of heterogeneous IT resources and multiple organizational domains thus posing many hard issues (e.g., distribution, concurrency, asynchrony, heterogeneity, evolution, autonomy, etc.) that the application server systems must address[5]. A successful application server will possibly possess the following characteristics:

- **Openness**: Enabling heterogeneous components to interact; incorporating various systems and resources into a whole without costly ad-hoc developments; integrating new features conveniently.
- **Flexibility**: Capable of evolving the existing legacy systems to meet the changing requirements; providing a flexible environment for developers to create their own services and integrate them into the system without changing the core structure of the system.

1. Introduction
systems; capable of facing run-time changes, being dynamically reconfigured to accommodate changing circumstances.

- **Modularity**: Separating a holistic complex system into several relatively smaller parts; allowing these parts to be autonomous; enabling “separation of concerns” and “divide and conquer”.

- **Manageability**: Allowing the components of a system to be monitored, controlled and managed in order to support dynamic reconfiguration, online QoS tuning, fault tolerance provision, etc.

- **Transparency**: Masking from applications the details of low level infrastructure mechanisms, such as security, transaction, communication; helping applications to focus on high level business logics.

There are many challenges to successfully constructing such a system. The key point is how to design a flexible architecture to enable application servers withstanding fast-breaking technology and business changes.

This paper discusses the design and implementation of an adaptive application server system named PKUAS[6] using microkernel-based approach. Microkernel first appears in the field of modern operating systems[7]. This approach brings to operating system implementation several advantages, such as extensibility, modularity, flexibility, portability, and reliability. There are several similarities between operating systems and application servers: both are system software, both need to remain operational for a long time, both are large and complex. Perhaps the most important common feature is that both provide an operating environment -- OS for normal applications, and application server for component-based applications. So an interesting question arises: is it possible to apply those successful ideas in OS architecture design to that of application server?

We thus apply the idea of microkernel from OS design to the construction of PKUAS. PKUAS is a J2EE application server with a microkernel based, service oriented architecture, which allows different services to be plugged into it and get managed conveniently.

The PKUAS microkernel has well-defined structure that strictly separates management concern from business concern, which brings excellent modularity and extensibility without causing much performance degradation. This feature distinguishes PKUAS from other microkernel based application server systems such as JBoss. Our practices show that this approach can effectively make application servers open to change.

The rest of this paper is organized as follows: Section 2 introduces the background of this paper; Section 3 explains the detailed PKUAS architecture design; Section 4 gives two examples and discusses the effects of this approach; Section 5 outlines the related work; and Section 6 concludes the paper.

2. Background
2.1. Microkernel in operating systems

The idea of microkernel originates from the design of modern operating systems.

Traditional layered operating systems are found hard to evolve, because changes in one layer can have numerous effects on code in adjacent layers. To solve this problem, some modern operating systems adopt the microkernel approach.

A microkernel is a highly modular system core running in kernel mode. It composes of only absolutely essential functions such as primitive memory management, inter-process communication, I/O and interrupt management[7]. Other operating system functions are implemented to execute in user mode and to use the microkernel for critical services.

This approach helps to control the growing complexity of modern operating systems. New features can be added easily with only minimum impact. Existing services can be subtracted or modified so as to produce a customized implementation. A microkernel operating system may have advanced features such as extensibility, modularity, flexibility, portability, and reliability.

2.2. Microkernel pattern

Since the microkernel approach in operating system has been proved to be effective, some researchers try to abstract this approach to a kind of architecture pattern to guide the design of adaptive systems [8, 9]. The pattern structure is illustrated in Fig.1.

The microkernel serves as a socket for plugging in new features and has the responsibility of coordinating service collaboration. Internal server extends the functionality of microkernel, which makes microkernel small. External server is a layer of abstraction built over atomic services and provides programming interfaces for user applications.

User application calls adaptor for service. The adaptor invokes methods of external servers on behalf of clients. The introducing of adaptor avoids tight coupling between user applications and underlying servers.

![Fig.1 Microkernel Pattern Structure](image-url)

The microkernel architectural pattern is expected to be applied to software systems that are under the pressure of constant evolution so as to adapt to changing
requirements. However, distributed computing systems are complex in nature. There is no hard and fast rule that can be stated about what constitutes a specific microkernel and what is the corresponding structure.

### 2.3. J2EE Application server

J2EE[10] application server hosts JAVA enterprise applications on the server side and provides deployment, management, and execution support to the hosted J2EE applications. There are two major parts in the J2EE platform architecture: application components, which hold presentation and business logic; and containers, which provide J2EE runtime environments for the application components.

The J2EE family is comprised of a large set of coordinated specifications and technologies. These specifications are not stable and keep on evolving, which forces the J2EE product vendor to follow. This often means making evolution on existing products.

Because the J2EE application server functionality is always subject to change and adaptation, the architecture should not be structured as monolithic but rather be decomposed. The challenge is how to design a lightweight container that meets functional requirements while at the same time being flexible enough to adapt to changing requirements easily.

### 3. PKUAS: An adaptive application server

In this section we will illustrate the microkernel-based, service-oriented architecture of PKUAS, and explain how this design helps make the whole system adaptive and open to change.

The PKUAS project has been ongoing for three years since 2001. The system has evolved from the initial 0.1 version to current version 2004 (1.0), which fully conforms to EJB 2.0 and J2EE 1.3 standards. The system now has been used in several application domains.

#### 3.1. Design rationale

The following explains the rationales that underpin the design of PKUAS:

**Separation of concern is the principal rule governing PKUAS design.** The tangling of different concerns is the root cause of high system complexity. So the system designers make a systematic use of well-known software engineering principles, e.g., using abstraction to separate the behavior of software components from their implementation; using modular approach to localize design issues; minimizing module coupling and maximizing module cohesion.

**PKUAS classifies adaptation into three levels.** The first level is online adaptation, which means the system
can automatically adapt to changing environments dynamically. The second level is static configuration of system settings. The last level is source code changing, which is not preferred but inevitable. The server architecture should facilitate all these adaptations.

**Services are the main ingredients of PKUAS.** There are the standard J2EE services provided to J2EE applications, platform services provided to server system (e.g., logging service, fault-tolerant service, etc), and user-defined domain/application level services. The server should allow old services to be easily modified and new services easily added.

**Container design is vital.** Containers provide the runtime support for J2EE applications and are the key role of J2EE server. Applications can’t directly access the underlying services but only through containers. It is containers that face the radical changes of services and must adapt to these changes.

**Management mechanism is necessary.** Management functions are not relevant to business concerns, but are often essential to system running. Example management functions include monitoring, online tuning, configuring, online adaptation, etc. So the server architecture should naturally integrate management mechanism.

### 3.2. Microkernel based architecture

The microkernel-based, service-oriented architecture of PKUAS is depicted in Fig.2. There are functionally four major parts in Fig.2: a microkernel, a service framework, a container framework and a server manager.

Everything plugged into the PKUAS microkernel is in fact a kind of service. The service framework deals with the heavy work related to J2EE services and user-defined services. The platform services mainly deal with platform related management issues. In fact, the ServiceManager, ContainerSystem, and ServerManager are all platform services.

All J2EE application related work is put into the container framework, which mainly consists of the container systems, containers and container system manager.

![Fig.2 PKUAS microkernel based architecture](image)

Unlike that in operating system, the microkernel of PKUAS is relatively simple. It only needs to integrate services together and provide a mechanism to coordinate
and manage these services. The following are the basic functions the microkernel must provide:

- Providing registration/deregistration interface to allow services to be plugged in.
- Providing naming and query interface for finding the appropriate plugged-in services.
- Providing a bus for invocations on those plugged-in services through specific interfaces.
- Providing service lifecycle management mechanism.

The to-be-plugged-in service must separately define the interface of non-business concern and business concern in order to be dynamically loaded, unloaded, updated, reconfigured. It is thus possible to make online adaptation to services according to changing requirements and environment with minimal interference to business methods.

Note that the microkernel only provides the integration and coordination mechanism, the actual management functions are performed by specific management services. The ServerManager in Fig.2 plays this role. It receives management commands from the management console and performances subsequent actions.

This system architecture brings to PKUAS characteristics of easy extensibility and adaptation. All services related maintenance and evolution work is restricted to the service framework, while the corresponding application related work restricted to the container framework.

3.2.1. Microkernel implementation

The microkernel of PKUAS is not built from scratch, but a result of reusing an existing full-fledged product of JMX.

JMX (Java Management Extensions) is a specification that defines the architecture, design patterns, APIs and services for application and network management and monitoring in the Java programming language[11]. This technology has been commonly used for the integration and management of software.

In JMX, manageable resources are represented by managed beans (MBeans), which register themselves to the MBeanServer. The ServiceManager, ContainerSystem, ContainerSystemManager, ServerManager, and all other services in PKUAS are all MBeans. These MBeans expose their specific management interface through metadata and operations available at the MBeanServer interface.

The MBeanServer is the key component. Most of the microkernel functions described in the previous section can be provided by MBeanServer.

The reusing of existing JMX products lowers development cost and improves quality while fully
satisfies the needs of good extensibility, high adaptability, and ease of management.

3.2.2. Service framework

J2EE specification defines a set of general services that all J2EE application servers must support. The number of J2EE services is increasing as specification version changes. More often than not, domain specific applications may require domain/application specific services. So a central service management mechanism allowing for flexible service extension and customization is necessary.

PKUAS provides an extensible service framework based on its microkernel. There are two types of elements in this service framework: ServiceManager and various concrete services. Typical services include naming, security, transaction, mail, log, etc. All these services are pluggable MBeans.

Fig.3 shows the relationship between service interface and service instance. The design is somewhat like that of Java Applet. The abstract interface ServiceMBean defines the “hooked” management operations all service MBeans must support. An abstract class Service implementing the ServiceMBean interface is the base class of all concrete services. The specific concrete service should first define its own ConcreteMBean interface that derives from the ServiceMBean; then define its own business interface (e.g., ConcreteServiceI) to be used by the containers; and then implement the concrete service class (e.g., ConcreteService). This concrete service class should both implement all the mandatory management operations and its own business service operations. All services in PKUAS must follow this pattern.

The microkernel pattern in section 2.2 states that clients shouldn’t interact with internal-servers directly but through external servers to reduce module dependency. The serving pattern in PKUAS follows this principle, as depicted in Fig.4. In Fig.4, Client can only access ConcreteService through ServiceAdaptor, which implements the business interface of ConcreteService and provides programming interface for its clients. What is more, a separated independent ServiceAdaptor makes distributed service access possible.

The ServiceManager takes charge of the management of all services. It will parse service configuration files, load the service classes into JVM (Java Virtual Machine) and then create the service objects. After that it will invoke the init and start method of service object to start the service. This process is also much like that of Java Applet. And ServiceManager can perform online adjustments on services, e.g., tuning system performance parameters.
Containers are interposed between the application components and the underlying J2EE services. J2EE application components can never invoke the platform services or other applications directly but only through their containers. This allows the container to transparently inject the services defined by the application components’ deployment descriptors, such as declarative transaction management, security checks, etc. In PKUAS, this injection is accomplished with the help of interceptors.

Fig. 5 shows the container framework of PKUAS. Because J2EE platform specification doesn’t specify the boundary between containers and the underlying application server, we place an intermediate layer named ContainerSystem to separate containers from the server. Logically, one ContainerSystem corresponds to one enterprise application which may contain several EJBs (Enterprise JavaBeans). The ContainerSystems are registered to the MBeanServer as MBeans and organize their contained containers into one manageable group.

3.2.3. Container framework
Such a layered structure effectively facilitates application management and allows container extension with limited impact on the underlying server.

The ContainerSystemManager manages all ContainerSystems. When it receives an application deployment request, it will create a ContainerSystem and perform some necessary initialization work. The ContainerSystem will parse the application deployment descriptor and create containers to host the to-be-deployed EJBs.

Because containers are interposed between applications and services, a big challenge is how to effectively integrate those diverse services dynamically while keeping the container implementation stable and clear. PKUAS uses interceptor mechanism to achieve this goal. Interceptors are loaded from configuration files when PKUAS initiate container.

Fig.6 shows the Interceptor interface in PKUAS. We can see that an interceptor has three methods to handle a request: beforeInvoke, afterInvoke and handleException. Each interceptor corresponds to one service. So there may be several interceptors installed in the container, and these interceptors are processed in a strict sequence that is determined when the container is initialized. As can be seen in Fig.5, before a request reaches the designated EJB, the container will first call beforeInvoke method of LoggingInterceptor, then SecurityInterceptor, then TransactionInterceptor. Typically the interceptors will ask their corresponding service instances in the service framework for business processing.

```java
public interface Interceptor {
    public void init(Container container) throws Exception;
    public void beforeInvoke(Object oid, Invocation iv) throws Exception;
    public void afterInvoke(Object oid, Invocation iv) throws Exception;
    public void handleException(Object oid, Invocation iv, Exception ex) throws Exception;
}
```

The idea behind the interceptor mechanism is the principle of “Separation of Concerns”. The interceptor functions as a bridge that links services to containers (applications). At the same time, it makes the container loosely coupled with these underlying platform services, which greatly facilitates services extension.

The introduction of interceptor mechanism makes the container implementation stable and lightweight, and makes containers immune to services changes.
3.3. Reflection

Reflection means that software is able to manipulate and reason about itself in the same way it does for its application domain.

PKUAS microkernel needs reflection mechanism to query and invoke those services unknown in compile time. Metadata facility is used for service description and manipulation. The PKUAS reflection mechanism is supported by JMX reflection facility and Java language reflection capabilities. A typical reflection use-case is loading configuration files in XML format to setup various services.

There are some high level reflection facilities in PKUAS that support manipulation of runtime software architecture (components, connectors, constraints)[12], which is outside the scope of this paper.

4. System evaluation

We give two examples here to see how this microkernel architecture helps PKUAS adapt to changes.

Example 1: Log service

Problem: Currently PKUAS supports two kind of logging service: logging to file and logging to console. Now we want to add another logging service: logging to database, so that a fault management program/module may analyze the logs and mine some fault patterns.

Solution: The existing ConsoleLogger and FileLogger diagrams are illustrated in Fig.8. The new DBLogger only needs to inherit from abstract parent class Logger and implement its own database logger function.

The starting/stopping/configuring of this new logger service is transparent to existing applications that use the log service. All the corresponding work are done by management facilities through LoggerMBean interface. And the new fault-mining service can be added to the system in the same way.

The configuration of DBLogger service is shown in Fig.9. The attribute “Filter” indicates that it will log the warn, serious, and error events. The attribute JNDIName indicate the data source name DBLogger uses.

![Fig.8 Adding new DBLogger service](image-url)

Some may argue that Fig.8 only shows a well-known design pattern. However, the focus here is not the design
pattern itself, but the adaptable underlying infrastructure mechanism supporting such pattern.

Example 2: Message compressing service

Problem: We once had one application interchanging large-sized messages with its clients. These large messages take up too much network bandwidth and the program spends much of its time on transmitting messages. So we want to compress all the messages before exchanging as to improve communication performance.

Solution: Obviously this needs supports from two sides: the server side and the client side. PKUAS provides the interceptor mechanism not only in the server side, but also in the client side.

So the solution is straightforward: plugging in the compressing/decompressing interceptor before messages are sent out at both server side and client side, as depicted in Fig.10.

In Fig.10, when messages are marshaled, they won’t be sent out directly, but will be compressed first by the client side CompressInterceptor. The peer server side CompressInterceptor then makes the reverse action – decompressing the message. For simplicity, the configuration of the CompressInterceptor is not shown here.

Discussion

The above two examples show that the microkernel architecture effectively makes PKUAS open to change. In our experiences the microkernel architecture has the following advantages:

First, it enables a better understanding of the design of the whole system. Experiments show that the first year postgraduates, with no prior knowledge about our project before, can master the designated modules and their inter-relationships after a relatively short time of study. The project continuity is guaranteed with a relatively low cost.
Second, management concerns and business concerns are clearly separated, which makes it easier to modify one concern without too much impact on others. At the same time, the code refactoring cost is reduced.

Third, new technology and services can be easily integrated into the existing system. The stable microkernel and extensible frameworks of PKUAS provides excellent extension mechanism.

Fourth, online reconfiguration and adaptation is possible[13].

Performance degradation always accompanies flexibility. The microkernel approach also has the performance problem. There is the cost of layered indirect message passing, reflection, etc. But as PKUAS microkernel only deals with management concern, the performance degradation of business concern is maintained in a reasonable level.

5. Related work

JBoss[3] is a well-known open source J2EE application server which is also built on JMX and claims to be microkernel-based. However, the JBoss microkernel doesn’t have a well-defined architecture – it simply regards the JMX MBeanServer as the microkernel. And JBoss doesn’t separate management concern from business concern -- both make heavy utilization of JMX reflection. In JBoss, the client request will first be accepted by one specific invoker, which converts the raw message to invocation object and passes the invocation object to MBeanServer by reflection. The MBeanServer then lookups the destined container and passes the message to the container. This can bring about severe performance degradation.

We run a test case to compare the performance of PKUAS to that of JBOSS. The test environment is: Win2000 Professional SP3, PIII800, 512M SDRAM, JBoss 3.2.1, JRMP. A standalone client sends a message with fixed length body (1, 32, …, 4096 bytes) to a stateless session bean, which does nothing but only acknowledges the receipt of this message. The client calculates the response time every 10,000 calls to minimize error.

![Fig.11 Performance metric of a test case on JBoss and PKUAS](image)

The result presented in Fig.11 shows that PKUAS response time is only 40% of JBoss in this test case. The reason is that in PKUAS the business invocations are not processed by microkernel as the case of JBoss, but are passed to containers directly. The PKUAS microkernel
only deals with management issues, which avoids performance degradation of business concern.

Some researchers introduce aspect-oriented approach [14] to middleware to help make middleware adaptive. DADO [15] is an approach to programming crosscutting concerns in distributed heterogeneous systems. This work uses the idea of aspect for modeling crosscutting changes to distributed systems at the IDL level. Component Virtual Machine[16] is a container model using aspect-oriented approach to implement custom services on the container side.

Reflective middleware is another hot topic. Some works exploit specially constructed reflective ORBs [17]. Chisel [18] is an open framework for dynamic adaptation of services using reflection in a policy-driven, context-aware manner. TAO is a dynamically configurable ORB applying pattern language to enhance the extensibility[19].

OSA+[20] is a scalable service-oriented microkernel middleware architecture that supports the distributed embedded real-time system design. This architecture adapts to heterogeneous environments.

6. Conclusion

This paper introduces our work on designing an adaptive J2EE application server named PKUAS using microkernel architecture approach. PKUAS has a microkernel based, service oriented architecture, which allows different services to be plugged into it and get managed conveniently. The PKUAS microkernel has well-defined structure that strictly separates management concern from business concern, which brings excellent modularity and extensibility to PKUAS without causing much performance degradation.

We plan in our future work to support autonomic adaptation. This requires two capabilities: first, the application server can sense out changes in its underlying infrastructures and environments; second, it should be able to dynamically adapt itself to the changed situation based on the information it sensed and gathered. An adaptive architecture can be of great help towards this goal.

Acknowledgement

This research is sponsored by the National Grand Fundamental Research 973 Program of China under Grant No. 2002CB312003; the National Natural Science Foundation of China under Grant No. 60233010, 60125206; the National High-Tech Research and Development Plan of China under Grant No. 2001AA113060; the Key Grant Project of Science and Technology Research of Ministry Of Education P.R.C. under Grant No. 0214.
Reference

[12] HUANG Gang, MEI Hong, YANG Fu-qing.


