****

****

毕业设计（论文）外文资料翻译

**学 院： 计算机科学与工程**

**专 业 班 级： 软件工程**

**指 导 教 师： 谷 钰**

**姓 名 学 号： 刘育超 20141450**

**外 文 出 处：**[https://scholar.google.com.hk/scholar?hl=en&as\_sdt=0%2C5&q=Infrastructure+for+Automatic+Dynamic+Deployment+Of+J2EE+Application+in+Distributed+Environments&btnG=0](https://www.sciencedirect.com/science/article/pii/S1877050917329800)

**2020年3月**

指导教师签字：

年 月 日

评阅人签字：

年 月 日

**作者：**Anatoly Akkerman, Alexander Totok, and Vijay Karamcheti

**英文原文：**

**Infrastructure for Automatic Dynamic Deployment**

**Of J2EE Application in Distributed Environments**

**Abstract:** in order to achieve such dynamic adaptation, we need an infrastructure for automating J2EE application deployment in such an environment. This need is quite evident to anyone who has ever tried deploying a J2EE application even on a single application server, which is a task that involves a great deal of configuration of both the system services and application components.

**Key words:** j2ee; component; Distributed; Dynamic Deployment;

**1 Introduction**

In recent years, we have seen a significant growth in component-based enterprise application development. These applications are typically deployed on company Intranets or on the Internet and are characterized by high transaction volume, large numbers of users and wide area access. Traditionally they are deployed in a central location, using server clustering with load balancing (horizontal partitioning) to sustain user load. However, horizontal partitioning has been shown very efficient only in reducing application-related overheads of user-perceived response times, without having much effect on network-induced latencies. Vertical partitioning (e.g., running web tier and business tier in separate VMs) has been used for fault isolation and load balancing but it is sometimes impractical due to significant run-time overheads (even if one would keep the tiers on a fast local-area network) related to heavy use of remote invocations. Recent work [14] in the context of J2EE component based applications has shown viability of vertical partitioning in wide-area networks without incurring the aforementioned overheads. The key conclusions from that study can be summarized as follows:

• Using properly designed applications, vertical distribution across wide-area networks improves user-perceived latencies.

• Wide-area vertical layering requires replication of application components and maintaining consistency between replicas.

• Additional replicas may be deployed dynamically to handle new requests.

• Different replicas may, in fact, be different implementations of the same component based on usage (read-only, read-write).

• New request paths may reuse components from previously deployed paths.

Applying intelligent monitoring [6] and AI planning [2, 12] techniques in conjunction with the conclusions of that study, we see a potential for dynamic adaptation in industry-standard J2EE component-based applications in wide area networks

Through deployment of additional application components dynamically based on active monitoring. However, in order to achieve such dynamic adaptation, we need an infrastructure for automating J2EE application deployment in such an environment. This need is quite evident to anyone who has ever tried deploying a J2EE application even on a single application server, which is a task that involves a great deal of configuration of both the system services and application components. For example one has to set up JDBC data sources, messaging destinations and other resource adapters before application components can be configured and deployed. In a wide area deployment that spans multiple server nodes, this proves even more complex, since more system services that facilitate inter-node communications need to be configured and started and a variety of configuration data, like IP addresses, port numbers, JNDI names and others have to be consistently maintained in various configuration files on multiple nodes.

This distributed deployment infrastructure must be able to:

• address inter-component connectivity specification and define its effects on component configuration and deployment,

• address application component dependencies on application server services, their configuration and deployment,

• provide simple but expressive abstractions to control adaptation through dynamic deployment and undeployment of components,

• enable reuse of services and components to maintain efficient use of network nodes’ resources,

• provide these facilities without incurring significant additional design effort on behalf of application programmers.

In this paper we propose the infrastructure for automatic dynamic deployment of J2EE applications, which addresses all of the aforementioned issues. The infrastructure defines architecture description languages (ADL) for component and link description and assembly. The Component Description Language is used to describe application components and links. It provides clear separation of application components from system components. A flexible type system is used to define compatibility of component ports and links. A declaration and expression language for configurable component properties allows for specification of inter-component dependencies and propagation of properties between components. The Component (Replica) Assembly Language allows for assembly of replicas of previously defined components into application paths by

Connecting appropriate ports via link replicas and specifying the mapping of these component replicas onto target application server nodes. The Component Configuration Process evaluates an application path’s correctness, identifies the dependencies

of application components on system components, and configures component replicas for deployment. An attempt is made to match and reuse any previously deployed replicas in the new path based on their configurations. We implement the infrastructure as a part of the JBoss open source Java application server [11] and test it on several

Sample J2EE applications – Java Pets tore [23], Rubies [20] and TPC-W-NYU [32]. The infrastructure implementation utilizes the JBoss’s extendable *micro-kernel* architecture, based on the JMX [27] specification. Componentized architecture of JBoss allows incremental service deployments depending on the needs of deployed applications. We believe that dynamic reconfiguration of application servers through dynamic deployment and undeployment of system services is essential to building a resource-efficient framework for dynamic distributed deployment of J2EE applications. The rest of the paper is organized as follows. Section 2 provides necessary background for understanding the specifics of the J2EE component technology which are relevant to this study. Section 3 gives a general description of the infrastructure architecture, while section 4 goes deeper in describing particularly important and interesting internal mechanisms of the infrastructure. Section 5 describes the implementation of the framework, and related work is discussed in section 6.

**2 J2EE Background**

**2.1 Introduction**

**Component frameworks.** A component framework is a middleware system that supports applications consisting of components conforming to certain standards. Application components are “plugged” into the component framework, which establishes their environmental conditions and regulates the interactions between them. This is usually done through containers, component holders, which also provide commonly required support for naming, security, transactions, and persistence. Component frameworks provide an integrated environment for component execution, as a result significantly reduce the effort .it takes to design, implement, deploy, and maintain applications. Current day industry component framework standards are represented by Object Management Group’s CORBA Component Model [18], Sun Microsystems’ Java 2 Platform Enterprise Edition (J2EE) [25] and Microsoft’s .NET [17], with J2EE being currently the most popular and widely used component framework in the enterprise arena.

**J2EE.** Java 2 Platform Enterprise Edition (J2EE) [25] is a comprehensive standard for developing multi-tier enterprise Java applications. The J2EE specification among other things defines the following:

• Component programming model,

• Component contracts with the hosting server,

• Services that the platform provides to these components,

• Various human roles,

• Compatibility test suites and compliance testing procedures.

Among the list of services that a compliant application server must provide are messaging, transactions, naming and others that can be used by the application components. Application developed using J2EE adhere to the classical 3-Tier architectures – Presentation Tier, Business Tier, and Enterprise Information System (EIS) Tier (see Fig. 1). J2EE components belonging to each tier are developed adhering to the

Specific J2EE standards.

1. **Presentation or Web tier.**

This tier is actually subdivided into client and server sides. The client side hosts a web browser, applets and Java applications that communicate with the server side of presentation tier or the business tier. The server side hosts Java Servlet components [30], Java Server Pages (JSPs) [29] and static web content. These components are responsible for presenting business data to the end users. The data itself is typically acquired from the business tier and sometimes directly from the Enterprise Information System tier. The server side of the presentation tier is typically accessed through HTTP(S) protocol.

2. **Business or EJB tier.**

This tier consists of Enterprise Java Beans (EJBs) [24] that model the business logic of the enterprise application. These components provide persistence mechanisms and transactional support. The components in the EJB tier are invoked through remote invocations (RMI), in-JVM invocations or asynchronous message delivery, depending on the type of EJB component. The EJB specification defines several types of components. They differ in invocation style (synchronous vs. asynchronous, local vs. remote) and statefulness: completely stateless (e.g., Message-Driven Bean), stateful non-persistent

(e.g., Stateful Session Bean), stateful persistent (e.g., Entity Bean). Synchronously invocable EJB components expose themselves through a special factory proxy object (an EJB Home object, which is specific to a given EJB), which is typically bound in JNDI by the deployer of the EJB. The EJB Home object allows creation or location of an EJB

Object, which is a proxy to a particular instance of an EJB 1.

3. **Enterprise Information System (EIS) or Data tier.**

This tier refers to the enterprise information systems, like relational databases, ERP systems, messaging systems and the like. Business and presentation tier component communicate with this tier with the help of resource adapters as defined by the Java Connector Architecture [26].The J2EE programming model has been conceived as a distributed programming model where application components would run in J2EE servers and communicate with each other. After the initial introduction and first server implementations, the technology, most notably, the EJB technology has seen some a significant shift away from purely distributed computing model towards local interactions 2. There were very legitimate performance-related reasons behind this shift, however the

Distributed features are still available. The J2EE specification has seen several revisions, the latest stable being version 1.3, while version 1.4 is going through last review phases 3. We shall focus our attention on the former, while actually learning from the latter. Compliant commercial J2EE implementations are widely available from BEA Systems [4], IBM [9], Oracle [21] and other vendors. Several open source implementations, including JBoss [11] and JOnAS [19] claim compatibility as well. A

Recent addition to the list is a new Apache project Geronimo [1].

**2.2 J2EE Component Programming Model**

Before we describe basic J2EE components, let’s first address the issue of defining what a component is a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [31].According to this definition the following entities which make up a typical J2EE application would be considered application components (some exceptions given below):

• EJBs (session, entity, message-driven),

• Web components (servlets, JSPs),

• messaging destinations,

• Data sources,

EJB and Web components are deployed into their corresponding containers provided by the application server vendor. They have well-defined contracts with their containers that govern lifecycle, threading, persistence and other concerns. Both Web and EJB components use JNDI lookups to locate resources or other EJB components they want to communicate with. The JNDI context in which these lookups are performed is maintained separately for each component by its container. Bindings messaging destinations, such as topics and queues, are resources provided by a messaging service implementation. Data sources are resources provided by the application server for data access by business components into the enterprise information services (data) tier, and most commonly are exemplified by JDBC connection pools managed by the application

Server. A J2EE programmer explicitly programs only EJBs and Web components. These custom-written components interact with each other and system services both implicitly and explicitly. For example, an EJB developer may choose explicit transaction demarcation (i.e., Bean-Managed Transactions) which means that the developer assumes the burden of writing explicit programmatic interaction with the platform’s *Transaction Manager* Service through well-defined interfaces. Alternatively, the developer may choose Container-Managed transaction demarcation, where transactional behavior of a component is defined through its descriptors and handled completely by the EJB container, thus acting as an implicit dependency of the EJB on the underlying *Transaction Manager* service.

**2.3 Links Between Components**

**2.3.1 Remote Interactions**

J2EE defines only three basic inter-component connection types that can cross application server boundaries, in all three cases; communication is accomplished through special Java objects.

*•* Remote EJB invocation: synchronous EJB invocations through EJB Home and EJB Object interfaces.

• Java Connector outbound connection: synchronous message receipt, synchronous and asynchronous message sending,

Database query using Connection Factory and Connection interfaces.

• Java Connector inbound connection: asynchronous message delivery into Message-Driven Beans (MDBs) only, utilizing Activation Spec objects. In the first two cases, an application component developer writes the code that performs lookup of these objects in the component’s run-time JNDI context as well as code that issues method invocations or sends and receives messages to and from the remote component. The component’s run-time JNDI context is created for each deployment of the component.

Bindings in the context are initialized at component deployment time by the deployed (usually by means of component’s deployment descriptors). These bindings are assumed to be static, since the specification does not provide any contract between the container and the component to inform of any binding changes In the case of Java Connector inbound communication, Activation Spec object lookup and all subsequent interactions with it are done implicitly by the MDB container. The protocol for lookup has not been standardized, though it is reasonable to assume a JMX- or JNDI-based lookup assuming the underlying application server provides facilities to control each step of deployment process, establishment of a link between J2EE components would involve:

• Deployment of target component classes (optional for some components, like destinations),

• Creation of a special Java object to be used as a target component’s proxy,

• Binding of this object with component’s host naming service (JNDI or JMX),

• Start of the target component,

• Deployment of referencing component classes,

• Creation and population of referencing component’s run-time context in its host naming service,

• start of the referencing component.

However, none of modern application servers allow detailed control of the deployment process for all component types beyond what is possible by limited options in their deployment descriptors 4. Therefore our infrastructure will use a simplified approach that relies on features currently available on most application servers:

• Ability to deploy messaging destinations and data sources dynamically,

• Ability to create and bind into JNDI special objects to access messaging destinations and data sources,

• Ability to specify initial binding of EJB Home objects upon EJB component deployment,

• Ability to specify a JNDI reference 5 in the referencing component’s run-time context to point to the EJB Home binding of the referenced EJB component. In our infrastructure which is limited to homogeneous application servers, these options are sufficient to control intercomponent links through simple deployment descriptor manipulation. However, in context of heterogeneous application servers, simple JNDI references and thus simple descriptor manipulation are insufficient due to cross-application-server

Classloading issues.

**2.3.2 Local Interactions**

Some interactions between components can occur only between components co-located in the same application server JVM and sometimes only in the same container. In the Web tier, examples of such interactions are servlet-to-servlet request forwarding. In the EJB tier, such interactions are CMP Entity relations and invocations via EJB local interfaces. Such local deployment concerns need not be exposed at the level of a distributed deployment infrastructure other than to ensure collocation. Therefore, the infrastructure treats all components requiring collocation as a single component.

**2.4 Deployment of J2EE Applications and System Services**

**2.4.1 Deployment of Application Components**

Deployment and undeployment of standard J2EE components has not yet been standardized (see JSR 88 [10] for standardization effort 6). Therefore, each application server vendor provides proprietary facilities for component deployment and undeployment. And while the J2EE specification does define packaging of standard components which includes format and location of XML-based deployment descriptors within the package, this package is not required to be deployable by an application server without proprietary transformation. Examples of such transformation are

• Generation of additional proprietary descriptors that supplement or replace the standard ones,

• Code generation of application server-specific classes.

In order to proceed with building a dynamic distributed deployment infrastructure capable of deploying in heterogeneous networks, we propose a universal unit of deployment to be a single XML-based deployment descriptor or a set of such,

Bundled into an archive. The archive may optionally include Java classes that implement the component and any other resources that the component may need. Alternatively, the deployment descriptors may simply have URL references to codebases. We assume presence of a dynamic deployment/undeployment service on all compliant J2EE servers and a robust application server classloading architecture capable of repeated deployment cycles without undesired classloading-related issues. Most modern application servers (e.g., JBoss [11] and Geronimo [1]) do provide such facilities.

**2.4.2 Deployment of System Components (Services)**

While lacking only in the area of defining a clear specification of deployment and undeployment when it comes to application components, the J2EE standard falls much shorter with respect to system services. Not only a standardized deployment facility for system services is not specified, the specification, in fact, places no requirements even on life cycle properties of these services, nor does it address the issue of explicit specification of application component dependencies on the underlying system services. Instead it defines a role of human deploy who is responsible for ensuring that the required services are running based on his/her understanding of dependencies of application components on system services as implied by the nature of components and their deployment descriptors.

**References**

[1] Matt Bishop. Computer Security: Art and Science. New York, 2002

[2] Matt Bishop. Vulnerabilities Analysis. Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection. Los Angeles 2006

[3] Balasubra maniyan. An Architecture for Intrusion Detection using Autonomous Agents[M]. Department of Computer Sciences, Purdue University, 1998.

**英文翻译：**

基于J2EE在分布式环境下的底层结构的自动动态配置的应用

Anatoly Akkerman, Alexander Totok, and Vijay Karamcheti

摘要：为了实现广域网中符合工业标准基于组件的应用程序中动态的可适应性，我们需要一种框架来在这样的环境里自动化地配置J2EE 应用程序。这种需要对于哪怕在单一的应用程序服务器上尝试部署J2EE应用的人来说也显而易见，这种任务设计到大量的系统服务和应用组件的配置。

关键词：j2ee；动态配置；分布式；组件；

1 前言

近几年，我们已经看到基于组件的企业应用开发的显著增加。这种应用程序通常被部署在公司的内部网或者是因特网上，以高事务容量，大量的用户和覆盖范围广的访问为特征，它通常会被部署在中央区域，采用服务器集群来均衡负载从而支持用户下载。但是这种平均负荷的方法被证明只对减少应用转发的用户可以察觉的反应时间有效，而对于减少网络方面的延迟作用不大，垂直分割（例如…运行网络层和事务层在不同的虚拟机）被用于错误分离和均衡负荷，但是它是不符合实际的由于远程调运的大量使用显著地增加了运行时时间。最近的著作已经表明在广域网中利用垂直负荷而不引起前面所述的超时问题的可行性。那非研究的主要结论可以概括如下：

(1) 应用合适的应用程序，在广域网中的垂直负荷可以察觉的延迟。

(2) 广域垂直层需要复制应用层组件而且需要维持和原组件间的一致性。

(3) 新加的复制组件可以被动态配置以满足新的需要。

(4) 事实上，不同的复制组件可能会根据应用不同的方式实现相组件。

(5) 新的请求路径可以复用先前的组件配置路径。

应用智能监视和人工智能规划方法再结合那个研究得出的结论，我们看到通过动态布置基于动态监视的额外的应用组件，在广域网中符合工业标准基于组件的应用程序中动态的可适应性是可以实现的。然而，为了实现这种动态可适性，我们需要一种框架来在这样的环境里自动化地配置J2EE 应用程序。这种需要对于哪怕在单一的应用程序服务器上尝试布置J2EE应用的人来说也显而易见，这种任务设计到大量的系统服务和应用组件的配置。例如你必须在配置和部署应用组件前先建立JDBC数据源，设立消息目的地和资源适配器。在需要跨越多个节点服务器的广域网配置中，这将更加复杂，因为更多的便利内部节点通信的系统服务需要配置和启动，而且多种配置数据比如IP地址，端口号，JNDI名字和其他的数据在多个节点的配置文件中必须维持一致性。这种分布式配置框架必须满足：

(1) 声明内部组件一致性规范和定义它对组件配置部署的影响。

(2) 声明应用程序组件对应用服务器，以及它们的配置和部署的依赖性。

(3) 提供简单但可表达的抽象方法去控制通过部署和拆卸组件获得的适用性。

(4) 能够复用服务和组件从而高效的利用网路节点资源。

(5) 提供上述便利而不会增加应用程序员的设计负担。

在本论文中，我们提出自动动态部署J2EE应用程序的框架涉及了上面的所有问题，这种框架为组件定义了结构描述语言，链接说明和集合。这种组件说明语言用来描述应用程序组件和链接，它使得应用组件与系统组件中清晰的分开。一种灵活的系统类型用来定义组件接口和端口的兼容性。一种为配置组件属性而开发的定义和表述语言允许内部组件间独立的规范和组件间属性的继承。组件集合语言允许先前定义的复制的组件通过连接合适的端口集合到应用路径，连接时通过链接复制对象和具体把这些复制组件映射到目标应用服务器节点。组件配置过程评估了应用程序路径的正确性，确认在系统组件上的应用组件的独立性和完成复制组件的部署。根据这些配置使先前部署的复制组件在新的路径中得以匹配和复用的努力正在做出。我们把这种架构作为JBoss开源java应用服务器的一部分加以实现，在几个J2EE样本程序比如Java PetStore,，RUB和 TPC\_W\_NYU中进行测试。这种架构实现利用了JBoss的可扩展的微内核结构，基于JMX规范。JBoss的组件结构允许根据部署应用程序的需要增加服务配置。我们相信通过动态部署和拆卸系统服务来重构应用服务器对构建高效资源框架的动态分布部署的J2EE应用程序来说是非常必要的。本文如下部分是这样组织的。第2部分提供了必要的背景以理解和研究有关的J2EE组件技术规范。第3部分对这种架构给出了一般性的描述。第4部分更深入的描述了有关这种架构特别重要的和有趣的内部机制。第五部分描述了如何实现这种架构，相关联的工作将在第六部分介绍。

2 J2EE背景知识

2.1 介绍

组件框架。组件框架是一种中间件系统，它支持遵守一定标准的有不同组件构成的应用程序。应用组件被塞入这种确立它们运行环境和规定它们交互的框架中。这通常是通过容器，组件持有者来实现的。这种容器也提供通常需要的功能以实现命名，安全性，事务，和持久性！组件框架为组件的执行提供了一个集成的环境，因此显著的减少了在设计，实现，部署和维护应用程序时工作。现在工业上的组件框架标准以对象管理组的CORBA组件模型， SUN 公司的JAVA 2 Platform J企业版[J2EE]和微软公司的.NET标准，其中在企业里应用最为广泛的组件框架是2EEE。J2EE. J2EE是开发多层企业应用JAVA程序的综合性的标准。J2EE规范定义如下：

(1) 组件编程模型。

(2) 组件和主服务器的链接。

(3) 服务器提供给组件的服务。

(4) 各种各样的人物角色。

(5) 兼容性检验装置和编译测试程序。

在众多的服务列表中，消息通信，事务处理，命名机制和其它应用组件用到的服务是应用服务器必须提供的。用J2EE进行应用开发必须遵守经典的3层结构—表现层，业务层和企业信息系统层。属于各层的J2EE组件在开发时遵守具体的J2EE标准。

1、表现层或者网络层

这一层实际上又被分为客户端和服务器端。客户端包括浏览器，applets，Java应用程序等和负责和服务器端的表现层或者业务层进行交互。服务器端包括servlet、jsp和静态网页内容。这些组件负责把业务数据传递给终端用户。数据本身通常从业务层获得有时也从企业信息系统层直接获得。表现层的服务器端通常通过Http协议来进行访问。

2、业务层或者EJB层

这一层包含EJB，即企业应用的事务逻辑模型。这些组件提供了持久化机制和事务支持。EJB中的组件通过RMI被调用。在Java虚拟机调用或者异步的消息传递，取决与EJB组件的类型。EJB规范定义了很多种组件。它们在调用风格（同步和异步，本地和远程）与状态（完全状态，不可持久状态，可持久）方面不同。同步调用的EJB组件通过特定的工厂代理对象来表现自己。这种工厂代理对象通常被EJB部署者绑定在JNDI中。EJB对象允许或者本地EJB对象是特定EJB实例的代理。

3、企业信息系统或者数据层

这一层指的就是企业信息系统，比如关系数据库，ERP系统，消息系统等。业务层和持久层在资源适配器的帮助下与该层进行通信。资源适配器在Java连结结构中被定义。J2EE编程模型一直被认为是分布式的编程模型，在该模型中应用组件在J2EE服务器上运行并且彼此可以相互交互。经过初始化说明和第一个服务实现后，该技术，更显著的说EJB技术，已经明显地从纯粹的分布式计算模型转向了本地交互。转变的背后有合理的性能有关的原因，然而分布式的特征现在还存在。J2EE规范已经经过了好几次修订，现在最稳定的版本是1.3，1.4版本正处于重审阶段。我们应该把注意力放在1.3版本上，而实际上是在学习后者。适用与商业的J2EE实现可以大量的从BEA系统，IBM，Oracle等赞助商得到。包括JBoss和JOnAS在内的开源实现据称兼容性也不错。最近名单上有多出了新的Apache project Geronimo。

2.2 J2EE组件编程模型

在我们基本的J2EE组件前，先让我们强调一下什么是组件。软件组件是有一系列的具体的接口和明确的上下文环境构成。它可以被独立的部署而且易于被第三方重构。根据以上的定义，如下的组成J2EE应用程序的实体可以看作是软件组件：

(1) EJBS（会话，实体，消息驱动）。

(2) Web组件（Servlet、JSP）。.

(3) 消息目的。

(4) 数据源。

EJB和Web组件被部署在由应用服务赞助商提供的容器中.它们有定义良好的容器规则来管理生命周期，线程，持久化和其他问题。EJB和Web组件都利用JNDI目录机制去寻找资源和它们想要交互的其EJB组件。目录被执行的JNDI环境被独立的由容器的每个组件加以维护。该种环境下的绑定机制通常由组件部署的解释者加以配置。消息目的地，像对话和队列，是由消息服务执行所提供的资源。数据源是提供给应用服务器的为事务组件进入到企业信息服务层提供数据接口，通常由被应用服务器管理的JDBC连接池实例化。一个J2EE编程者明确编写的项目只有EJB和Web组件。这些用户编写的组件彼此交互而且系统服务可以是明显的也可以是隐含的。例如，EJB开发者可以选择明确的事务区分方式，这种方式意味着开发者假设通过定义良好接口的事务经理服务平台来书写明确的程序交互。或者，开发者也可选择容器管理事务区分的方式。这样由于组件的事务行为通过他的描述者来定义而且全部用EJB容器来处理，因此作为一个隐式独立的EJB提供潜在的事务管理服务。

2.3 组件间的链接

2.3.1 远程交互

J2EE仅定义了三种可以在不同应用服务器间传递的基本组件间连接类型。在这三种情况下，通信通过特定的Java对象来完成。

(1) 远程EJB调用：同步的EJB调用通过主EJB对象和EJB对象接口来实现。

(2) Java连结器的外部连接：同步消息接收，同步和异步消息发送，用连接工厂和连接接口进行数据库查询。

(3) Java连接器的内部连接：异步消息传递进入消息驱动Bean只能使用Activation Spec 对象。

在前两个实例中，应用组件的开发者不仅书写执行在组件的运行时JNDI环境中的对象目录代码，而且书写发布方法调用，与远程的组件相互发送和接受消息。组件的运行时JNDI环境为每一个组件部署所创建。环境中的绑定在组件部署时由部署者进行初始化。这些绑定被假设为是静态的，因为规格中没有提供任何的容器和组件间协议去提示绑定发生了变化。在 Java连接器的内部通信情景下，Activation Spec 对象查询以及所有的相应的M容器隐式的完成。虽然查询的协议还没有被标准化，但是假设一个基于JMX或者JNDI的查询是合理的。 假设潜在的应用服务器提供了所有的设备去控制部署过程的每一步，那么在两个J2EE组件间确立一个连接需要涉及：

(1) 部署目标组件类。

(2) 创建一个特定的Java对象用作目标组件代理。

(3) 用组件的命名服务去绑定目标。

(4) 启动目标组件。

(5) 部署指定的组件类。

(6) 在主机的命名服务中，创建和进行指定组件的运行环境。

(7) 启动指定的组件。

然而，没有一个现代的应用服务器允许详细的控制所有组件类型的部署过程除了在它们的部署解释器中的有限的选择。因此我们的架构将使用简化的途径，它所依赖的特征在现在的大多数的应用服务器上都可以得到。

(1) 动态部署消息目的和数据源的能力。

(2) 创建和绑定特定的JNDI目标去访问消息目的和数据源的能力。

(3) 把初始绑定的EJB对象到EJB部署组件的能力。

(4) 用在参考组件运行环境中的JNDI指引去指出绑定的参考EJB的能力。

在只有相同的应用服务器的架构中，上面的功能对通过简单的部署控制解释器方式来控件间的连接已经足够了。然而，在不同应用服务器的环境下，由于跨服务器的类下载问题，这种简单的控制解释器的方式是不够的。

2.3.2 本地交互

一些组件间的交互可以发生在同一地点的相同应用服务器虚拟机的组件间，有时候甚至可以发生在只有相同容器的组件间。在Web层，这种交互的例子是 servlet到 servlet的请求转发。在EJB层，这种交互的例子是CMP实体关系和通过EJB本地接口的调用。这种本地部署所关心的不是在分布式架构中去表现而是去增强一致性。因此，这种架构把所有的本地的组件请求当作一个单一的组件加以对待。

2.4 部署J2EE应用程序和系统服务

2.4.1 部署应用程序组件

部署和拆卸标准的J2EE组件还没有统一的标准，因此每个应用服务的提供商对组件的部署和拆卸提供了单独的功能于J2EE规范中没有定义标准组件的包，包的格式和包内的基于xml部署解释器的位置，因此这种包对于没有所属权变化的应用服务器不需要部署。具体变化的例子有：

(1) 支持或者取代标准所有者解释器的新的所有者解释器的产生。

(2) 具体服务应用程序类的代码的更替。

为了着手构建一个能够部署不可网络的动态的分布式的架构，我们提出了一种普遍的部署单元即一个简单的基于xml部署的解释器或者是一组类似的绑定到文档中的解释器。文档可能包含用于执行组件的Java类或者任何其它的所需组件。相应地，部署解释器也可以简单地用URL来索引代码。我们假设这种动态的部署和拆卸服务存在于所有的兼容的J2EE服务器上而且在不理解类重载相关问题时一个健壮的类重载结结构的应用服务器就能够重复的部署生命周期。大多数现代的应用服务器都提供这样的功能。

2.4.2 部署系统组件

对应用组件来说，J2EE规范只是少了在部署和拆卸时的明确定义，而对系统服务来说，在这方面做的更糟。对系统服务来说不仅没有具体的定义一个标准化的部署，实际上，这个规格甚至连没有强调在生命周期属性方面的要求，更不用手强调依赖也潜在的系统服务的应用组件的明确规范了取而代之的是它定义了部署者的角色，这个角色负责确保像组件的本性和系统的解释器所暗示的那样，所需的服务是基于应用组件对系统服务依赖性的基础之上。例如，假如有一个事务容器要至少用一种方法去开始一个新的事务，那么一个带有这样的事务容器的EJB就需要在应用层表示事务管理服务。与之相似的是，一个消息驱动的bean，也隐式需要一种运行在网络上消息服务实例。它为MDB管理消息目的以及基于查询的Java连接器通过它的管理服务层去提供这种消息服务。考虑到应用层可能通常只用到了应用服务器所提供的服务的一个子集，根据应用层的需要允许递增的配置服务的组件应用服务器允许更高效的利用多种资源。包括，开源的应用服务器，JBoss和OnAS在内，已经有多种J2EE应用服务器已经全部或者部分的实现了组件化。我们感觉到通过动态的部署和拆卸系统服务，动态的配置应用服务器对动态分布的部署J2EE应用程序是一种十分重要的构建资源有效型框架的方法。因此我们提倡并将把用JBoss应用服务器设计的微内核的应用服务器用作一个模型。在该模型中，一个微型的服务包括了系统调用总线，一个稳健的类下载子系统，一些命名子系统和一个动态配置子系统。所有其它的服务是热部署并且通过一个普通的调用总线来进行通信。例如，JBoss利用Java管理扩展服务器来实现基本的命名和调用功能。除此之外，JBoss实现了一个先进的类下载子系统和部署服务。所有其它的JBoss是动态配置的，并外在的表现为具有良好机制和生命周期的JMX MBeans.这样的一种应用服务器根据系统服务外在利用应用组件去设计相关功能，并且只有需要的系统服务才会得到合理配置和部署！

参考文献

[1] Matt Bishop. Computer Security: Art and Science. New York, 2002

[2] Matt Bishop. Vulnerabilities Analysis. Proceedings of the Second International Symposium on Recent Advances in Intrusion Detection. Los Angeles 2006

[3] Balasubra maniyan. Architecture for Intrusion Detection using Autonomous Agents[M]. Department of Computer Sciences, Purdue University, 1998.