OpenEJB 1.1 Specification

Draft 0.7

Richard Monson-Haefel

David Blevins

Section 1: Introduction

1.0 Overview

1.1 Monolithic vs. Modular Design

OpenEJB represents a revolution in application server design, a view that application servers should be modular, not monolithic. A modular application server is built from subsystems rather than constructed as one huge, tightly coupled platform. Modularization of application server software allows vendors to focus on their core competencies instead of reinventing every subsystem from scratch to create a complete platform. Not only is modularization possible, OpenEJB makes it a reality.

OpenEJB is an EJB container system - not a monolithic EJB server - that can be plugged into any application server to make it a fully compliant EJB server.

The Enterprise JavaBeans API itself does not specify a separation of responsibilities among the application server, the container, and the primary services (transaction, security, and connectors). As a result, EJB vendors must build proprietary monolithic application servers to support all the subsystems needed for a complete EJB platform.

OpenEJB clearly defines the separation of its responsibilities as a container system from those of the application server that hosts it, and from the primary services that support it. This decoupling enables vendors of application servers, transaction managers, and providers of security services and connectors to focus on their own specialties, while OpenEJB focuses on delivering a high-speed container system that combines the services into a single EJB platform.

This Introduction Section examines the need for a separation of responsibilities among application server, container, and primary services, and how OpenEJB provides a powerful container system and a set of programming interfaces that make this modularization possible.

1.1.2 The container-server contract

Enterprise JavaBeans defines a portable server-side component model for enterprise computing. EJB clearly specifies a bean-container contract and a client-server contract that allow developers to switch EJB server products in an enterprise system without significant redevelopment costs. While the EJB specification defines portability in terms of the enterprise-bean and EJB-client programming models, the EJB servers themselves - all the subsystems that lie between the enterprise beans and the client applications - remain proprietary.

1.1.3 No formal contract in the EJB specification

The Enterprise JavaBeans specification does not define a server-container contract. This omission is intentional; it was done to facilitate maximum flexibility for vendors defining EJB server technologies. Beyond isolating the beans from the application server, the container’s responsibility in the EJB system is vague. At the application server level, the EJB specification defines only a bean-container contract and does not define the server-container contract.

It is difficult to determine exactly, for example, which is responsible for resource management and other services, the container or the application server. Without a clear separation of responsibilities between the container and the application server, EJB vendors must bear the burden of implementing the entire platform, including the distributed-object service, naming, transaction management, security, and EJB container. As a

result, commercial EJB servers tend to be complex monolithic platforms with proprietary and hidden implementations.

1.1.4 The need for a contract

The advantage to defining a server-container contract is that it allows third-party vendors to produce containers that can plug into any application server. If the responsibilities of the container and application server are clearly defined, then vendors who specialize in the technologies that support these different responsibilities can focus on developing the container or application server that best matches their core competencies. Web vendors focus on managing web requests; CORBA vendors focus on distributed-object requests; TP monitors focus on transaction management. Meanwhile, the EJB container vendor focus on managing the enterprise beans within the container. Until now this separation of responsibilities did not exist.

1.2 The OpenEJB server-container contract

OpenEJB is a pre-built, self-contained, portable EJB 1.1 container system that can be plugged into any application server environment. OpenEJB provides a clear separation of responsibilities between the EJB container and the EJB server. The application server and OpenEJB container system interact through an elegant and powerful programming interface, which forms the server-container contact. This contract is defined by the Server-Container Interface (SCI), a small, simple, and refined set of classes and interfaces.

1.2.1 The application server's role

Application servers that use OpenEJB are responsible for providing client applications with naming and remote access to the application server. Its services may include providing JNDI and proxy implementations that fulfill the client-server contract of the EJB programming model. In application servers that will use OpenEJB locally, such as servlet engines, OpenEJB already provides the necessary JNDI and proxy implementations. When a client makes a request on a remote bean reference, the application server delivers the request to the OpenEJB container system, which delegates the request to the appropriate enterprise bean and applies transaction, connectors and security services appropriately.

The Apache Tomcat server is a good example of an application server that could easily be extended using OpenEJB to provide its servlets with a complete EJB container system. Enydra and OpenORB are other examples of application servers that would benefit from OpenEJB integration.

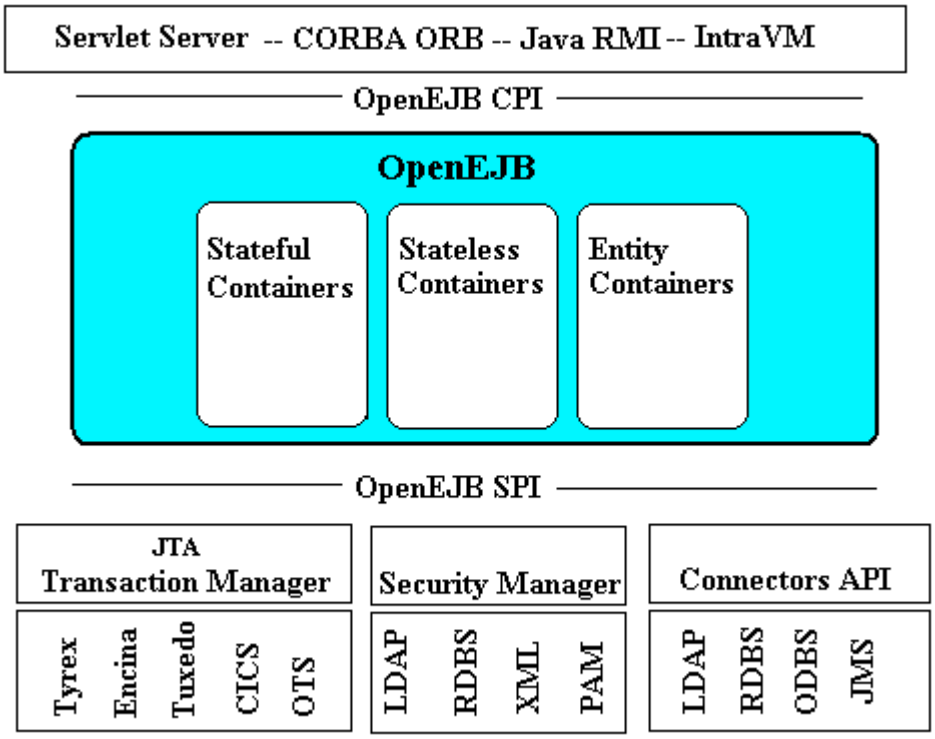
1.2.2 The container's role

OpenEJB manages the enterprise bean's lifecycle and coordinates the application of transactions (distributed or local), connectors, and security as defined by the EJB 1.1 specification. To manage these tasks in a way that is flexible and customizable, OpenEJB also enforces a separation of these responsibilities into separate services. OpenEJB provides a Service Provider Interface (SPI) for transaction, connectors and security services. These simple and flexible container-service contracts are based on simple adapters and industry standards like the Connector API and Java Transaction API (JTA), so it is easy for service providers to support the SPI and plug directly into OpenEJB. In addition, services are swappable and are easily configured by the application server vendor as well as the customer.

1.3 The Architecture

1.3.1 Overview

OpenEJB is the first EJB container system that allows developers of an EJB platform to assemble it from existing products rather than construct it from scratch. Vendors focus on what they do best while OpenEJB provides the container to host Enterprise JavaBeans. When plugged into any Java compatible application server, the result is a complete, yet modular Enterprise JavaBeans 1.1 container system. Through the server-container interface (SCI), an application server vendor can use the OpenEJB container system to create an instant and customizable EJB 1.1 platform. Through the service-provider interface (SPI), primary services may be interchanged to match any target environment's specific requirements. Figure 1 shows how OpenEJB separates responsibilities of the application server, container, and primary services.



1.3.2 The containers

The OpenEJB container system provides three robust container types, including stateless and stateful session-bean containers, and entity-bean containers for both bean- and container-managed entity beans. These containers are strictly compliant with the EJB 1.1 specification, and provide the full complement of security and transaction behaviors to beans. The OpenEJB containers are very lightweight because they multiplex requests concurrently, requiring less overhead to service more beans. The containers are also extremely fast because they introduce virtually no bottlenecks to service requests, allowing thousands of requests to execute within the container system simultaneously.

1.3.3 The primary services

OpenEJB defines three primary services: Transaction, security, and connectors. The containers use the transaction, security, and connector services while servicing beans and performing other responsibilities: the transaction service provides the container with transactional integrity; the security service provides authorization control; the connectors provides an API for managing resource connections like JDBC and Java Message Service (JMS).

1.3.4 The IntraVM server

OpenEJB includes an IntraVM server that allows for swift interaction among beans in the same virtual machine. While respecting the remote semantics required by EJB, optimizations in method calls from one bean to another provide for very fast throughput and little or no latency. The IntraVM provides an immediately available platform for application servers that do not need to support client access through distributed objects. For example, an existing servlet engine or web server can use the IntraVM server to integrate the OpenEJB container systems with very little effort.

1.4.5 Customizable

OpenEJB provides a core package that is a powerful default implementation of a container system. While this core is well engineered for performance and efficiency, it is possible to replace parts of the core system in favor of custom implementations. For example, the passivation strategy in the stateful container, which currently writes to a file, can be swapped out in favor of one that writes to a RDMBS or some other secondary storage.

Developers can also add custom containers to extend the scope of the container system. A container that uses JNI to interface with an ERP system, for example, could be integrated with other core OpenEJB containers. Even the configuration system in OpenEJB, which currently uses a flexible XML DTD, can be replaced. A proprietary container assembler could replace the current assembler to build containers from a database - or even read another vendor's configuration file to load a container system at runtime. Practically every feature of OpenEJB is replaceable, providing vendors and customers with unparalleled customization and flexibility. However, customization is not at all necessary. OpenEJB includes a powerful core package that makes it an extremely fast container system.

1.3.6 Flexible

The OpenEJB container system defines lightweight and flexible interfaces that allow vendors to integrate OpenEJB into their application server products easily. OpenEJB provides application server vendors without EJB support a fast track to EJB compliance. OpenEJB also opens up the integration of transaction, security, and connector services so that vendors and customers alike can choose the services that are the most appropriate for their EJB platforms. Service providers can quickly adapt their products to OpenEJB SPI, allowing them to compete with other vendors on the quality of their service.

1.4 OpenEJB is Open Source

1.4.1 The OpenEJB License

OpenEJB is an open source software project with a BSD license that is similar to the Apache license. Although it's a new open source project, OpenEJB already has several platforms targeted for integration including Apache Tomcat, OpenORB, and OpenJMS.

1.4.2 The open source advantage

Open source exposes all the source code of the software to the world, providing significant advantages to its creators, its customers, and other developers. Unlike proprietary software, open source products contain no mysteries, no hidden "features." The support costs are much lower with open source than for a proprietary product. Customers that encounter anomalous behavior can choose to locate the exact source of the problem themselves and report it. Most important, open source encourages customers and other developers to contribute enhancements to the software, so that an open source product becomes a joint development effort among its creators, its users, and other vendors.

1.4.3 The open source community

A community forms around every open source project, comprising developers and users that employ the software in commercial and non-commercial environments. If an open source project is well designed and useful, it can become wildly successful. Typical and striking examples include Linux, Apache, and Perl, each of which has enjoyed success in both open source and commercial communities.

1.4.4 The synergy of OpenEJB and open source

OpenEJB is especially well suited for open source because its modular design allows it to be integrated into many different platforms. This flexibility makes OpenEJB the universal solution for any application server that needs EJB functionality. OpenEJB represents a revolution in application server design, a view that application servers' subsystems should be built by organizations that specialize in those subsystems, then bolted together to create a single complete platform, turning the potential of modularization into a reality.

Open source code represents another revolution in software, one that has a special synergy with modular application server development. Open source exposes the code so that everyone understands how the software works; modular application server software like OpenEJB exposes public programming interfaces so that it can be assembled into an endless variety of platforms. Together, open source and modular software form an excellent union of two philosophies.

Section 2: The Server-Container Contract

2.1 Overview

The Server-Container Contract defines the separation of responsibilities between the application server and the OpenEJB container system. The server and container systems interact exclusively through the interfaces and classes in the `org.openejb` package[1], which are collectively refereed to as the Server-Container Interface (SCI). The sever-container contract is defined by the SCI as well as the policies enumerated in this section of the OpenEJB specification.

2.1.1 OpenEJB Responsibilities

OpenEJB instantiates a container system at runtime. The container system is responsible for managing enterprise beans at runtime according to the information provided in their XML deployment descriptors[2]. The container system interposes between requests made by the server and serviced by enterprise beans. The container system interposes to provide transaction, authorization security, and connector support for beans servicing requests from the server. The transaction, security, and connector services used during interposition are those assigned to the container system by the configuration.

The container system provides a fast minimally synchronized environment for enterprise beans that is conformant with the EJB 1.1 specification.

The container system provides the server with an API (the SCI) for locating the container that hosts a specific bean deployment and delegating requests to the bean.

2.1.2 Server Responsibilities

The server is any Java application that hosts the OpenEJB container system. The server is responsible for supporting the EJB Client API for distributed clients or some other similar API (CORBA, DCOM, etc.). The server must provide the stubs, network protocol, dispatching, and naming services used to locate and communicate requests with the container system.

2.1.3 The Server-Container Interface

The server and container interact through the server-container interface (SCI). The SCI is made up of the 10 classes and interfaces defined in the `org.openejb` package. Using the SCI the server will delegate requests to beans to the container system, which may reply with return values or exceptions depending on the outcome of the request and the type of container.

Several classes and interfaces in the SCI represent a hierarchy for managing and hosting enterprise beans at runtime. The participants in this hierarchy are described below.

---

[1] Its possible that the application server also implements one or more of the Service-Provider Interfaces, but that is outside the scope of the Server-Container contract discussed here.

[2] The deployment information is actually mapped by the deployer to the OpenEJB configuration information for the target plateform.

2.1.3.1 OpenEJB

`OpenEJB` is the root of the container system hierarchy. It is responsible for loading the configuration information and manufacturing the containers. The OpenEJB class is a static singleton that provides access to the deployments, containers, transaction manager, and security service.

```
Properties props = new Properties();
...
org.openejb.OpenEJB.init(props);
TransactionManager txMngr = OpenEJB.getTransactionManager();
SecurityService ss = OpenEJB.getSecurityService();
Container [] containers = OpenEJB.containers();
DeploymentInfo [] deployments = OpenEJB.deployments();
```

One the OpenEJB has been initialized, it can be used to access the containers, deployments and primary services. OpenEJB maintains one container system per Java Virtual Machine and can not be initialized more then once in the life of a process.

2.1.3.2 Container

The `Container` manages one or more bean deployments at runtime. The Container interface provides methods for accessing the Container's id and the deployments managed by the container (represented by `org.openejb.DeploymentInfo` objects). In addition, the container defines the `getContainerType()` method, which will return `Container.ENTITY`, `Container.STATEFUL`, or `Container.STATELESS` depending on the bean type managed by the container.

```
Container cntr = OpenEJB.getContainer("Accounting");

DeploymentInfo [] deployments = cntr.deployments();

DeploymentInfo deployInfo = cntr.getDeployment(someId);

int type = cntr.getContainerType();
```

The Container interface has been separated from the RpcContainer in order to support future container types that are not based on RPC style communications such as the message-driven bean container in EJB 2.0, which will included in the next release of OpenEJB, OpenEJB 2.0.

2.1.3.3 RpcContainer

The RPC container is used for `SessionBean` and `EntityBean` deployments, which are accessed via remote procedure calls (RPC) from clients. Its called the RPC container (`RpcContainer`) because it enforces the semantics the Java RMI API, but can be used with any RPC protocol including JRMP, RMI-IIOP, IIOP, SOAP, and, hypothetically, DCE and DCOM. The `RpcContainer`s assume that a call is synchronous and that arguments, return values, and exceptions will adhere to the Java RMI-IIOP API policies defined by the EJB 1.1 specification -- the actual protocol can be anything. The server is responsible for translating arguments, return values, and exceptions between the `RpcContainer.invoke()` method and the distributed object protocol used by the server.

The `RpcContainer` extends the Container interface and defines one method, `invoke( )`; The `invoke()` method is used by the server to delegate bean requests to the RpcContainer.

```
RpcContainer rmiCntr = (RpcContainer)OpenEJB.getContainer("Accounting");


Object retVal = rmiCntr.invoke(deployID,method,args,null,principal);
```

2.1.3.4 DeploymentInfo

The `DeploymentInfo` represents a unique bean deployment in the container system. It maintains all deployment information about the bean that may be useful to the server in determining the behavior and identity of a specific bean deployment. DeploymentInfo objects are uniquely identified within the container system by their deployment-id. The deployment-id can be used in the `getDeploymentInfo()` method, defined in both the OpenEJB class and Container interface to obtain a specific DeploymentInfo object.

```
Object deployID = deploymentInfo.getDeploymentID();
DeploymentInfo di_1 = OpenEJB.getDeploymentInfo(deployID);
DeploymentInfo di_2 = container.getDeploymentInf(deployID);

If(deployID.equals(di_1) && deployID.equals(di_2))
    // this condition will always be true
```

Its expected that most servers will associate a deployment-id with a remote reference connection, since the server must supply the deployment-id when invoking the `RpcContainer.invoke()` method. Servers can use the DeploymentInfo to obtain information about the bean as well as obtaining a direct reference to the bean's container.

The deployment-id is often the same as the JNDI lookup name used by remote clients to access the bean. This is not a requirement, but can make it direct client requests to the correct container and DeploymentInfo object. The deployment-id can, however, be any character string as long as deployment-ids are unique with a container system.

2.1.3.5 ProxyInfo

An instance of `ProxyInfo` class represents a remote reference to a bean in the container system. When a bean method (create, find, or business methods) is defined as returning a remote reference to another bean, the container will replace that reference with the `ProxyInfo` object if its a local bean. Local beans are those beans deployed in the same container system.

The `ProxyInfo` object provides information sufficient for the server to generate a native remote reference that can be used by the calling client. This native remote reference is implemented using the distributed protocol used by the server.

2.1.3.6 OpenEJBException

This is the base exception type of the ApplicationException, InvalidateReferenceException and SystemException. It is never thrown directly by the container system to the server.

2.1.3.7 ApplicationException

An `ApplicationException` is thrown when invocation on the bean instance results in a `java.rmi.RemoteException` or some type of EJB application exception. The server must propagate the cause of the ApplicationException to the client.

2.1.3.8 InvalidateReferenceException

The `InvalidateReferenceException` is type of `ApplicationException` that is thrown when EJB 1.1 policy requires that the server invalidate the client's remote reference. The `InvalidateReferenceException` is only thrown for stateful session beans when an operation results in the destruction of the bean instance. The server must propagate the cause of the InvalidateReferenceException (usually a RemoteException) and then make the clients remote reference inoperable.

2.1.3.9 SystemException

The `org.openejb.SystemException` is used to report a system failure in the container system or one of the service providers (transaction, security, or connectors). The condition that caused the exception is considered a partial or complete failure of the container system that may be isolated to the container or service provider that caused the exception.

Container may throw a system exception if an abnormal condition exists while handling server request. Examples of abnormal conditions include: An I/O error that occurs passivating stateful bean instances to disk; an invalid deployment-id or call method is used as an argument to a `RpcContainer.invoke( )` method call.

When a `SystemException` is thrown, the server can attempt to recover by re-executing the request, removing the container from service, or removing the entire container system from service. The `SystemException` wrappers the causal exception which may be evaluated to determine the source of the failure. For example: If the causal exception is a `javax.jta.SystemException` the server knows that the exception was thrown from the transaction service provider.

2.1.3.10 EnvProps

The `EnvProps` provides a type for standard constants used when initializing the container system.

2.2 Server-RpcContainer Contract

The server initializes the OpenEJB container system at server start-up or before the first bean request is serviced. Every bean deployment has an id, called the *deployment-id*, that is unique within a container system. The server must provide its RPC clients with a naming service that maps names to deployment-ids associated with RpcContainers. In many cases the naming service's name binding may be the same as the deployment-id, but this is not required. When the client performs a name service lookup, the server will return a home reference; a proxy or stub that implements the bean's home interface.

The server is responsible for providing the distributed communication infrastructure (proxies, network protocol handlers, dispatchers, etc.) for remote references. Every remote reference provided by the server is associated with a specific deployment-id, which in turn maps to a specific `RpcContainer`. When the client invokes a method on a bean or home reference, the server will either delegate the request to reference's `RpcContainer` or service the request itself.

In general, create, remove, finder, and business methods are delegated to the reference's `RpcContainer`. The values returned or exceptions thrown are propagated to the server's clients. The server itself services all other methods of the `EJBHome` and `EJBObject` interfaces because these methods require the creation of distributed communication artifacts specific to the server's distributed object protocol (i.e. `Handle` and `EJBMetaData`) or that are easily handled by the server (i.e. `EJBObject.isIdentical()` and `EJBObject.getPrimaryKey()`).

The server delegates all RPC requests to RPC container using the `RpcContainer.invoke()` method. The `invoke()` method will either return normally or throw an exception. Normal return values are propagated to client, exceptions are handled according to the exception policies outlined in section "2.2.1.3 Exceptions".

2.2.1 Invoke Policies

In general requests by the client on bean's business methods and the home's create, and find methods are delegated to the `RpcContainer`. The `RpcContainer` interface defines the `invoke( )` method

which represents much of the server-container contract, which is expressed through parameters, return values, and exceptions that can be thrown by the invoke method to the server.

2.2.1.1 Parameters

The `RpcContainer`.invoke( ) method defines declares five parameters.

```
public Object invoke(Object deployID,
                     Method callMethod,
                     Object [] args,
                     Object primKey,
                     Object securityIdentity)

       throws OpenEJBException;
```

2.2.1.1.1 deployID

This is the deployment-id of the bean to be invoked. Every bean deployment has a unique deployment-id within the scope of a container system. The deployment-id can be used to obtain the bean's DeploymentInfo object which is used by the container to select the correct bean deployment to service the method request as well as providing other information. (A container can and often does support several different deployments at the same time.). The deployment-id type is inconsequential to the container system, but is usually a String or Integer type.

```
String deploymentID = … get deployment id from request
DeploymentInfo deployment = OpenEJB.getDeploymentInfo(deploymentID);
RPCContainer container = (RPCContainer)deployment.getContainer();
container.invoke(deploymentID,callmethod, args, securityIdentity);
```

2.2.1.1.2 callMethod

This is the `java.lang.reflect.Method` object that represents the remote interface (home or remote) method that was invoked by the client. This parameter is used by the container to determine whether the method should be delegated directly to the bean or run under a different method (i.e. `create()` is run as `ejbCreate()`). In addition, the Method object is used as a key in the container system when determining the transaction and security attributes that must be applied when the operating on the bean instance.

2.2.1.1.3 args

The `args` array is the parameter values used by the client to invoke the remote reference method. The arguments must be indexed in the array in the same order they are declared on the remote interface method. Primitive values (`int`, `double`, `char`, etc.) must be substituted with their primitive wrapper counter parts (`Integer`, `Double`, `Character`, etc.) in the array. The arguments must be copied and not passed by reference from the client. This will be a natural occurrence for most distributed object systems since arguments must be marshaled across the network. Arguments that are remote references must be operational, meaning that they must implement the remote interface and provide network access to their respective bean.

2.2.1.1.4 primKey

The primary key identifies a unique bean in the container system. For stateful beans the primary key identifies a unique bean instance that maintains a conversational state with a specific client. For entity beans the primary key identifies a unique bean in the database. Stateless session beans do not use a primary key, so the `primKey` parameter will be null for requests on stateless beans. The primKey is usally `null` for invocations servicing home interface methods -- home references don't have a primary key. The exception is EJBHome remove methods for which the server must provide the primary key of the bean (stateful or entity) that is to be removed.

2.2.1.1.5 securityIdentity

The `securityIdentity` is the security identity representing the client. The server is responsible for providing the proper object for the container systems designated security service. The type and implementation of the `securityIdentity` object is immaterial to the container. It's assumed that the security service used by the container will know how to use it.

For example, if the container's profile specifies security service provider that uses SSL authentication, then the proper SSL credential may be passed as the `securityIdentity` parameter. If the security service provider uses Kerberos then the proper Kerberos token may be passed as the `securityIdentity` parameter.

2.2.1.2 Return Values

The `RpcContainer.invoke()` method returns an Object type. The value of the return depends on the expected return type of the remote method that was invoked; the declared return type of the home or remote interface.

2.2.1.2.1 void

If the remote interface method returns a void, the `invoke( )` method will return a `null` value. The remote reference will then return void.

2.2.1.2.2 primitive types

If the remote interface method returns a single primitive value (`int`, `double`, `char`, etc.), the `invoke()` method will return the corresponding primitive wrapper (`Integer`, `Double`, `Character`, etc.) object. The remote reference will then return the primitive value of the wrapper object.

2.2.1.2.3 Serializable types

If the remote interface method returns a serializable type, the `invoke()` method will return an object of that type. The remote reference returns a copy of the serializable object.

The exception is `java.lang.String` and primitive wrapper types (`Integer`, `Double`, `Character`, ect.), which are immutable and need not be copied.

2.2.1.2.4 Array Value types

If the remote interface method returns an array, the `invoke()` method will return an array of that type. The remote reference returns a complete copy of the array including the individual elements.

2.2.1.2.5 Remote References

If the remote interface method returns a remote reference, the `invoke()` method will either return `ProxyInfo` object or the actual remote reference. `ProxyInfo` objects are returned for local beans, which are beans managed within the container system. Remote references are returned by for non-local beans, which are beans managed by a different container system or a different EJB vendor (Weblogic, WebSphere, etc.).

If the method called is a create or single object find method, the `invoke()` method will return the `ProxyInfo` that represents the created or found bean. If a business method returns a remote reference to a local bean, the container will return a `ProxyInfo` object for that bean. The server then, must generate the remote reference based on the `ProxyInfo` and returned that reference to the client.

The `ProxyInfo` object provides information sufficient for the server to generate a remote reference for the local bean including the remote interface to implement (remote or home), the deployment-id, the primary key of the bean, and the a reference to the bean's container.

If the bean returns a remote reference to a non-local bean, the container will return that same remote reference to the server. The assumption is that the remote reference from the non-local bean (different container system or vendor) is serializable, so otherwise can be transferred between address spaces and remain operational. Problems can occur when the distributed object protocol used by the sever is incompatible with the non-local bean reference's implementation.

2.2.1.3 Exceptions

There are three possible exceptions that can be thrown by the `invoke()` method: `ApplicationException`, `SystemException`, or the `InvalidateReferenceException`.

2.2.1.3.1 ApplicationException

An `ApplicationException` is thrown when invocation on the bean instance results in a `java.rmi.RemoteException` or some type of EJB application exception. The server must propagate the cause of the `ApplicationException` to the client. When an `ApplicationException` is thrown, the cause is considered to be normal; it does not represent a system failure or require invalidation of the remote reference. The cause should be propagated to the client but no other action need be taken by the server.

When a bean method throws a custom application exception from a business methods or a standard application exception (`CreateException`, `ObjectNotFoundException`, etc.) from a standard business method (`ejbCreate`, `ejbFind`, etc.), the container will catch the exception and re-throw it wrapped in an `ApplicationException` to the server.

Under some circumstances, a `RemoteException` may occur while delegating a request to the bean instance. Examples include: A security authorization violation (the client is not authorized to access the bean method); An attempt to propagate a transaction to a bean method declared with the `Never` transaction attribute; An attempt to perform a loopback on an entity bean that is declared as non-reentrant. In these cases the `RemoteException` will be wrapped in a ApplicationException by the container and thrown to the server from the `invoke()` method. The server must treat the `RemoteException` as it would any custom application exception; the `RemoteException` should be propagated to the client, but no other action need be taken by the server.

2.2.1.3.2 InvalidateReferenceException

The `InvalidateReferenceException` is type of `ApplicationException` that is thrown when EJB 1.1 policy requires that the server invalidate the client's remote reference. The `InvalidateReferenceException` will nest a `RemoteException` or standard application exception (i.e. `ObjectNotFoundException`) that must re-thrown to the client by the bean reference. The `InvalidateReferenceException` is only thrown for stateful session beans when an operation results in the destruction of the bean instance. Examples include: An `EJBException` is thrown by a stateful bean method (business method or standard callback method) and the instance is evicted; A client attempts to invoke a method on a stateful bean object that no longer exists.

When the `invoke()` method throws an `ApplicationException` the server must: First throw the nested (cause) exception to the client; Second, invalidate the reference so that any subsequent method invoked on the reference immediately throw a `RemoteException`. The reference is made inoperable.

2.2.1.3.3 SystemException

A SystemException is thrown by the invoke() method when a partial failure of the container or one of the service providers (transaction, security, connectors) has occurred while servicing the method. Conditions that cause a `SystemException` and server responsibilities are covered in more detail in Section 2.4: SystemException.

A `SystemException` represents a serious error in the container system. It should be assumed if this exception is thrown that the container system is unstable and appropriate action by the server should be taken.

2.2.2 The EJBObject

The EJBObject is implemented by server specific stub that acts as a remote reference to a corresponding bean in a RpcContainer. The EJBObject stub must implement both the `javax.ejb.EJBObject` interface as well as the remote interface (and all its super types) defined by the bean developer.

When an EJB client creates or finds a bean from a EJBHome reference, the server is responsible for returning a remote reference to that bean. The remote reference, the EJBObject, is implemented according to the server's distributed object protocol and is bound to its bean's RpcContainer at the server. How this is accomplished is up to the server, but it is expected that servers will maintain a mapping between the EJBObject's network connection or logical thread, and the correct RpcContainer using the deployment-id. For stateful and entity beans, the EJBObject reference is mapped to a specific bean instance within a container using the deployment-id and primary key.

2.2.2.1 Remote Interface Methods

The remote interface methods, those business methods defined by the bean developer in the remote interface, are delegated to the `invoke()` method of the appropriate `RpcContainer` (the container that services the bean deployment represented by `EJBObject`). Invocations on the `EJBObject` stub are transmitted from client to the server (using the servers distributed object protocol) and then delegated to the appropriate `RpcContainer`.

2.2.2.2 EJBObject Interface Methods

Except for the `remove( )` method, all EJBObject interface methods are implemented by the server. (The `remove( )` method is handled by the OpenEJB RpcContainer via the `invoke( )` method.) The EJBObject methods are dependent on server specific conventions and distributed object protocol, so the responsibly for implementing these methods falls on the server.

2.2.2.2.1 getEJBHome

The `getEJBHome()` method is handled by the server which will need to return an application server implementation of the EJBHome stub implementing the appropriate home interface to the client.

2.2.2.2.2 getHandle

The `getHandle( )` method is implemented by the application server. The Handle class needs to be serializable, so that it can be written to file or stream. The Handle implementation must be able to re-connect to the server after it is deserialized using the server's distributed object protocol, and return an EJBObject stub for the appropriate bean.

2.2.2.2.3 getPrimaryKey( )

The server may choose to maintain a reference to the primary key in the stub, or to maintain it at the server. Session beans do not expose their primary keys, so the application server will need to throw a RemoteException when this method is invoked on session EJBObject stubs.

2.2.2.2.4 isIdenticial

The `isIdenticial(EJBObject obj)` method is implemented by the server. The application server should compare container and deployment-ids for an exact match. In addition, the identity (primary key) for entity and stateful session beans must match as well. The primary key and other identifiers may be maintained in the EJBObject stubs or on the server.

2.2.2.2.5 remove

The remove( ) method is delegated by the server to the `invoke( )` method of the proper `RpcContainer`. When invoked on an EJBObject stub for a stateless bean, the server doesn't not need to delegate the invocation to the container -- EJBObject.remove( ) invocations are not handled by the stateless container.

Once a `remove( )` method has been processed, the server should attempt to invalidate all the EJBObject stubs of the bean that was removed, so that subsequent invocations on that EJBObject stub result in `javax.ejb.ObjectNotFoundException` being thrown to the client. Subsequent requests by server on behalf of the removed bean will result in the `RpcContainer` throwing a `org.openejb.ApplicationException` that wraps a `javax.ejb.ObjectNotFoundException.`

2.2.3 The EJBHome

The EJBHome is implemented by server specific stub that acts as a remote reference to a corresponding deployment in a RpcContainer. The EJBHome stub must implement both the `javax.ejb.EJBHome` interface as well as the home interface (and all its super types) defined by the bean developer.

When the client uses a naming service to locate a bean's EJBHome reference, the server is responsible for returning a remote reference to that bean. The home reference, the EJBHome, is implemented according to the server's distributed object protocol and is bound to its bean's RpcContainer at the server. How this is accomplished is up to the server, but it is expected that servers will maintain a mapping between the EJBHome's network connection or logical thread, and the correct RpcContainer using the deployment-id. In some cases the deployment-id and the lookup name may be the same value, but this is not required and is considered a server specific option.

2.2.3.1 Home Interface Methods

The home interface methods, those methods defined by the bean developer in the home interface, are delegated to the `invoke()` method of the appropriate `RpcContainer` (the container that services the bean deployment represented by `EJBHome`). Invocations on the `EJBHome` stub are transmitted from client to the server (using the servers distributed object protocol) and then delegated to the appropriate `RpcContainer`.

In the case of home interface methods, a `null` value must be passed as the primary key argument in the `RpcContainer.invoke()` method. Home interface methods are not specific to one bean instance and so have no primary key associated with method requests. The only exception to this is the EJBHome remove methods which are discussed in more detail in section 2.2.3.2.

When a home interface `create` method is delegated to the `RpcContainer` the return value will always be a `ProxyInfo` object describing the remote reference that should be returned to the client. The remote reference is a distributed object stub implemented according to the server distributed object protocol.

When a single-value `find` method (i.e. `findByPrimaryKey()`) is delegated to the `RpcContainer` a single `ProxyInfo` object will be returned, the same as with the `create` methods. When a multi-value `find` method is delegated to the `RpcContainer`, a `java.util.Collection` of `ProxyInfo` objects is returned to the server. The server is then responsible for returning a `Collection` or `Enumeration` of remote references to the client. The collection type returned to the client depends on how the `find` method was declared.

2.2.3.2 EJBHome Interface Methods

Except for the `remove( )` method, all `EJBHome` interface methods are implemented by the server. (The `remove( )` method is handled by the OpenEJB RpcContainer via the `invoke( )` method.) The `EJBHome` methods are dependent on server specific conventions and distributed object protocol, so the responsibly for implementing these methods falls on the server.

2.2.3.2.1 getEJBMetaData

The `getEJBMetaData( )` method is implemented by the application server. The `EJBMetaData` class needs to be serializable, so that it can be written to file or stream. Using the deployment-id, the server can lookup the `DeploymentInfo` object for that bean, and use it to populate the `EJBMetaData` object. In addition, the `EJBMetaData` implementation must be able to re-connect to the server after it is deserialized using the server's distributed object protocol, and return an `EJBHome` stub from the `EJBMetaData.getEJBHome( )` method.

2.2.3.2.2 getHomeHandle

The `getHomeHandle( )` method is implemented by the application server. The `HomeHandle` class needs to be serializable, so that it can be written to file or stream. The `HomeHandle` implementation must be able to re-connect to the server after it is deserialized using the server's distributed object protocol, and return an EJBHome stub for the appropriate bean.

2.2.3.2.5 remove

The `remove( )` methods are delegated by the server to the `invoke( )` method of the proper `RpcContainer`. When invoked on a `EJBHome` stub for a stateless bean, the server doesn't not need to delegate the invocation to the container -- `EJBHome.remove( )` invocations are not handled by the stateless container – but doing so will not cause an error.

Once a `remove( )` method has been processed, the server should attempt to invalidate all the `EJBObject` stubs of the bean that was removed, so that subsequent invocations on that `EJBObject` stub result in `javax.ejb.ObjectNotFoundException` being thrown to the client. Subsequent requests by server on behalf of the removed bean will result in the `RpcContainer` throwing a `org.openejb.ApplicationException` that wraps a `javax.ejb.ObjectNotFoundException.`

Section 3: The Service Provider Interface

Connector Support

Included in the service provider interface of OpenEJB is the Connector API. The Connector API is a standard J2EE API that defines a contract between the “application server” and resource connectors. Resource connectors are technology specific APIs for accessing backend systems like relational databases, messaging services, and ERP systems. The most familiar resource APIs are JDBC and JMS. It is the responsibility of the “application server” to manage the pooling and application of transactions and security services to resource connections used by enterprise beans. It's the responsibility of the connectors to provide physical resource connections and the facilities for manufacturing those connections. The Connector API specifies how “application server” and the connectors interact to manage resource connections.

In the case of OpenEJB the role of the “application server” -- as defined by the Connector API specification -- is fulfilled by the OpenEJB container system. OpenEJB provides a mechanism for declaring connectors and their connection managers in the XML configuration file. OpenEJB also provides connection managers with access to the appropriate transaction and security service at runtime through a standard JNDI name space.

A connection manager is responsible for managing resource connections according the contracts defined in the Connector API. This includes pooling and sharing connections as well as managing the transaction and security contexts of connection at runtime. Connection managers implement the `javax.resource.spi.ConnectionManager` interface defined in the Connector API.

A resource connection (a.k.a. connector) is responsible for providing physical connections to a backend resource such as a relational database or enterprise messaging system. A connector must implement the SPI contracts defined by the Connector API specification in addition to a technology specific API. For example, a JDBC connector will implement the Connector SPI as well as the JDBC API. Similarly, a JMS connector will implement the Connector SPI and the JMS API. The Connector SPI is used by the connection manager to mange the physical resource connections -- independent of the type of backend resource accessed by the connector. The technology specific API is used by enterprise beans to work directly with the backend resource accessed by the connector. For example, an enterprise bean will use the JDBC API to query, update, and delete records in a relational database system.

The advantage of the Connector API is that it allows any Connector compliant resource to work with any Connector compliant application server. A vendor who creates a connector for a specific backend resource can be assured that their product will work with any application server that supports the Connector API. Before the introduction of the Connector API, organizations were limited to the resource connections supported by their application server. The Connector API largely eliminates this limitation. If your application server is complaint with the connector API, it should automatically support any existing and newly defined connectors.

OpenEJB takes this portability a step further by making the connection managers plugable. In most application servers, the vendor will offer one or more connection manager options, but the selection of connection managers are fixed and proprietary -- they are built into the application server. OpenEJB, on the other hand, allows any third party to define a connection manager that can easily be plugged into OpenEJB and used at runtime to manage connectors. The connection manager defined by the third party is responsible for pooling connections and interacting with connectors according to the Connector API. OpenEJB is responsible for providing the connection manager with access to a JTA TransactionManager and the security service. In addition, OpenEJB provides enterprise beans with access to connectors through

their JNDI environment naming context (ENC). The bean uses only the resources technology specific API and is not aware of the connector API, which is only used by the connection manager.

While developing connectors and connection managers is not trivial, plugging third party connectors and connection managers into OpenEJB is very easy. Both connectors and connection managers are declared in the facilities section of OpenEJB's XML configuration file. The following example shows that multiple connectors can be assigned to the same connection manager. In this case the OpenEJB JDBC Connector and a JMS connector provide by the Acme corporation are assigned to the Connection manager provided by Blue Sky corporation.

```
<connectors>
<connector>
    <connector-id>OrdersDatabase</connector-id>
    <connection-manager-id>LocalShared</connection-manager-id>
    <managed-connection-factory>
        <class-name>org.openejb.resource.jdbc.JdbcManagedConnectionFactory
        </class-name>
        <properties>
           <property>
             <property-name>setJdbcDriver</property-name>
             <property-value>sun.jdbc.odbc.JdbcOdbcDriver</property-value>
           </property>
           <property>
             <property-name>setJdbcUrl</property-name>
             <property-value>jdbc:odbc:orders</property-value>
           </property>
       </properties>
    </managed-connection-factory>
</connector>
<connector>
    <connector-id>InventoryTopic</connector-id>
    <connection-manager-id>LocalShared</connection-manager-id>
    <managed-connection-factory>
         <class-name>com.acme.jms.TopicManagedConnectionFactory
           </class-name>
           <properties>
              <property>
                <property-name>URL</property-name>
                <property-value>jms:acme.com/mytopic</property-value>
              </property>
           </properties>
    </managed-connection-factory>
</connector>
<connection-manager>
    <connection-manager-id>LocalShared</connection-manager-id>
    <class-name>
    com.blue-sky.openejb.ConnectionManager
    </class-name>
</connection-manager>
</connectors>
```

Connectors can be assigned to any one of several connection managers. Connectors can be assigned to any one of several connection managers. For example, there could be four connectors declared and two connection managers. Two of the connectors are assigned to one connection manager, while the other two are assigned to the other connection manager.

Enterprise beans access connectors as resources through their JNDI ENC. The connector-id used in the deceleration of a resource is mapped directly to connector-id of a connector in the facilities section. In the below example, the enterprise bean declares two resources whose `connector-id` elements map to connectors shown in the previous example.

```
<entity-bean>
    <display-name>EmployeeEJB</display-name>
```

```
    <ejb-deployment-id>111111</ejb-deployment-id>
    <home>org.openejb.test.beans.EmployeeHome</home>
    <remote>org.openejb.test.beans.Employee</remote>
    <ejb-class>org.openejb.test.beans.EmployeeBean</ejb-class>
    <primary-key>java.lang.Integer</primary-key>
    <jndi-enc>
        <resource-ref>
            <res-ref-name>jdbc/orders</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
            <connector-id>OrdersDatabase</connector-id>
        </resource-ref>
        <resource-ref>
            <res-ref-name>jms/inventoryTopic</res-ref-name>
            <res-type>javax.jms.TopicConnectionFactory</res-type>
            <res-auth>Container</res-auth>
            <connector-id>InventoryTopic</connector-id>
        </resource-ref>
    </jndi-enc>
</entity-bean>
```

Current Status: January 4, 2001

The Connector API is near completion but is not yet in its final release. As a result there are no connectors currently available. While its believed that vendors will quickly release a cornucopia of connectors following the specification's final release, the OpenEJB project provides an interim solution: a JDBC connector that can be used with any JDBC driver. The JDBC connector is packaged under `org.openejb.resource.jdbc`. It can be used to wrapper any JDBC driver and make it Connector compliant (limited). The OpenEJB JDBC connector is currently limited to local transactions (see Connector specification) and does not provide support for the XA interfaces and 2-phase commit. This is likely to change in the coming months. In the mean time, organizations can use the OpenEJB JDBC connector to access any JDBC compliant database within a local transaction.

OpenEJB also provides a connection manager, the `org.openejb.resource.SharedLocalConnectionManager`, that can be used to manage any connector that supports local transactions. In addition the `SharedLocalConnectionManager` supports connection sharing, so that components in a chain of execution will automatically share the same physical connection and local transaction. This makes it possible to support a transaction for a single resource that spans several components in the same unit-of-work.

For example, bean A calls bean B which calls bean C. All three beans access the same JDBC resource from their JNDI ENC (the res-id is the same). With connection sharing bean B and C will use the same physical JDBC connection used by bean A. When the unit-of-work is complete and all the beans have completed their work, the transaction on the JDBC connection will be committed. All the work preformed on that physical connection by each of the beans is committed or rollback together; its atomic. However, connection sharing does not allow different resources to be enrolled in the same local transaction. If for example, beans A, B, and C all access JDBC and JMS resources, then the JDBC work will be committed together and the JMS work will be committed together, but the JDBC and JMS work will be committed separately. If the JDBC commit fails, the JMS commit will still be executed. That's the limitation of a shared local connection manager. A connection manager that supports 2-phase commit would ensure that these different resources succeed or fail together. Its likely that a connection manager that supports 2-phase commit will be provided by OpenEJB or some third party in the future.

From a developers stand point the connectors are a non-issue. The developer simply declares which connectors are used with which connection managers and maps the connectors as resources enterprise bean's JNDI ENC. Its our hope that others will consider developing connection managers that can be used

with OpenEJB. Over the coming weeks the OpenEJB-ConnectionManager interface will be refined. Currently, OpenEJB provides connector managers with access to the TransactionManager used by the bean requesting access to a resource.

Section 4: The Core OpenEJB Implementation

Section 5: Vendor Interoperability

Section 6: OpenEJB Customization

The OpenEJB factory

The `OpenEJB` class manufactures an instance of `OpenEJB` with a complete container system that is ready to accept bean requests. Essentially, the server will use the static `init( )` method of the `OpenEJB` class to bootstrap a container system. The Properties object passed as an argument to the `init( )` method is combined with the System properties and passed to the constructor of an OpenEJB instance.

```
Properties props = new Properties();
...
OpenEJB ejb = org.openejb.OpenEJB.init(props);
```

The `OpenEJB` instance is responsible initiating the construction of container system by locating the correct `Assembler` and providing it with access to configuration information. The Assembler is responsible for constructing every artifact of the container system and preparing it to service bean requests based on configuration information. The type of Assembler and the source of the configuration information can be identified through properties, which can be declared in the Properties argument or in the system class properties.

The Assembler property uses the property name "`org/openejb/assembler_class`". This property name is bound to the fully qualified class name of the Assembler class used to build the OpenEJB container system. The default implementation is, which is used if an Assembler property is not specified, is the `org.openejb.core.conf.Assembler`. The Assembler is responsible for constructing the entire container system using the configuration information.

The configuration source property uses the property name "`org/openejb/configuration_source`". This property identifies the source of the configuration information used to build the container system. The configuration source may be a local file, remote location, a database table, etc. The configuration information itself can take any form as long as it works with the named Assembler. The default `Assembler` expects a XML file that is located on the local hard drive and conforms to the *openejb_config.dtd*.

The Assembler and configuration source property names are conveniently declared as static fields in org.openejb.EnvProps class.

```
package org.openejb;

public class EnvProps {

    public final static String CONFIGURATION = "org/openejb/configuration_source";
    public final static String ASSEMBLER = "org/openejb/assembler_class";
}
```

If the core OpenEJB implementation is to be used, the `EnvProps.ASSEMBLER` property should not be declared since the default is to use the core Assembler class. The `EnvProps.CONFIGURATION` property, however, must be set to the file location of the XML configuration file on the local hard drive.

It's possible to develop and use a custom Assembler class. This would be useful if custom containers or other artifact are needed that can not be constructed by the default core Assembler. Some vendors may, for example, want to use a configuration schema that is different then the XML schema used with OpenEJB's core library. This would require a new or modified Assembler. If a custom Assebler is needed, it can be specified in the properties along with its custom configuration. For example, an assembler might be constructed that can build an container system off of a Weblogic properties files.

```
Properties props = new Properties();
props.addProperty(EnvProps.ASSEMBLER, "com.acme.openejb.weblogic_assembler");
props.addProperty(EnvProps.CONFIGURATION,"c://openejb/welogic.props");
org.openejb.OpenEJB.init(props);
```