Total Business Integration™

White Paper

CORBA-EJB Interoperability

IONA Technologies PLC

July 2001

M2632

Summary

The adoption rate of server-side Java has accelerated dramatically in the last year. According to some surveys, as many as a third of development organizations have deployed or soon will deploy systems built upon Enterprise JavaBeans. As enterprise Java use grows, organizations inevitably will find it necessary to connect Java systems to existing applications. Many organizations have made significant investments in CORBA technology, and now are seeking to make CORBA and EJB applications work together.

This paper explores the issues surrounding CORBA-EJB interoperability, with a particular focus on non-Java CORBA clients calling EJB servers.

There is a standard for direct CORBA-EJB interoperability, but it is difficult to use and may be unworkable in many situations. This paper explains why this is the case, and shows how to avoid some of the worst pitfalls.

An alternative approach is to implement CORBA wrappers that encapsulate EJB objects behind a CORBA interface. This paper shows how the use of code generation can constrain EJB interfaces to be "CORBA-friendly" and can automate the task of building a wrapper for a CORBA-friendly EJB.

Finally, this paper outlines an efficient framework for implementing wrappers using stateless POA servants. There may be a large number of EJB entities in a system, and EJB sessions may be created and destroyed rapidly, so it is important to avoid maintaining state such that wrappers are forced to track the lifecycle of EJB objects.

This White Paper is a technical discussion intended for designers and implementers of integrated CORBA-EJB systems and CORBA-EJB bridging technology. You should have some development experience with CORBA or EJB and a basic understanding of both. The CORBA-EJB quick comparison at the end of this paper will help you if you are only familiar with one of these technologies.

Table of Contents

1 Introduction

EJB-to-CORBA communication is straightforward. Any Java code (EJB or otherwise) can call CORBA objects via a Java ORB. CORBA IDL is designed to map to many languages; the mapping to Java [1] is easy to understand, simple to use and well supported by numerous Java ORBs including Orbix 2000 and the built-in ORB in the JDK.

The EJB specification includes a section on CORBA interoperability [2]. It details transparent interoperation of naming services, transaction services and security services as well as on-the-wire interoperability. If the application server supports CORBA interoperability, CORBA clients can directly call any EJB via the IDL interface implied by the standard OMG Java to IDL mapping [3].

Unfortunately the Java to IDL mapping produces IDL that is difficult for a CORBA client to use. You can avoid some of the worst pitfalls by avoiding certain Java constructs when designing your EJB, but you cannot avoid them all. In most applications you will need to build CORBA wrappers for your EJBs. A wrapper is just a normal CORBA object that delegates to an EJB.

This paper is structured as follows:

- The Java to IDL mapping: The standard interoperability mechanism and what is wrong with it. This section is long; therefore as soon as you are convinced that a problem exists, you can skip ahead to later sections that discuss the solutions.

- CORBA-friendly EJB: If you are working with a clean slate and are willing to live within some reasonable restrictions, you can design EJBs that are easy to map to CORBA. In particular, if you are willing to use IDL to define CORBA-friendly interfaces in the first place, you can enforce the CORBA-friendly rules automatically.

- Building and generating wrappers: If your client software cannot handle the IDL produced by the standard Java to IDL mapping, you need to build wrappers. You can automate this process if you use IDL that makes your EJBs CORBA-friendly in the first place.

- Pre-existing EJBs and Interoperability borders: In some systems you may need to mix CORBA-friendly and CORBA-unfriendly EJBs inside and outside the "interoperability border."

- A wrapper framework: Once you have built (or generated) wrappers, you need to give them a place to live. This paper shows you how to implement wrappers as stateless servants with efficient lifecycle management, and discusses the deployment options.

- Conclusions: A review of the pros and cons of the various options available.

Before diving into this discussion, you may ask whether you really need to solve this problem at all. A common assumption is that in a mixed CORBA-EJB system, the EJB components will call CORBA services, but CORBA clients will not need to call on EJB services.

However even if the majority of communication is initiated by an EJB mid-tier to a CORBA back end, the system may need alarms, notifications or other "call-backs" from the CORBA services to the EJB mid-tier.

The assumption may not even hold for the majority of communication. For example if two independently developed EJB and CORBA systems are later integrated, then key services in the EJB system will need to be available to CORBA clients.

2 The Java to IDL Mapping[1]

The CORBA-EJB interoperability specification consists of four mappings:

- Distribution: mapping EJB interfaces to CORBA interfaces
- Naming: Mapping JNDI names to CosNaming names
- Transactions: Propagating transactions between CORBA and EJB worlds
- Security: Securely propagating security information between CORBA and EJB worlds

The last three are important. It would be difficult to implement seamless security or transactional behavior without an application server that implements these parts of the specification. However, these functions are well catered for by the interoperability standard, and are practically invisible from the developer point of view. The most interesting one is the distribution mapping, which consists of the Java to IDL mapping [3].

[1] The OMG is currently defining the CORBA Component Model [4]. The model includes a mapping between EJBs and CORBA components. However this mapping is based on the same Java to IDL mapping discussed here, so everything said here still applies.

2.1 Using the Java to IDL Mapping

In an application server that fully supports the EJB to CORBA mapping [2], EJB objects are automatically CORBA objects as well. The IDL interfaces are normally implicit but they can be generated from the EJB remote Java interfaces. The Java naming service (JNDI) is mapped to the CORBA naming service so EJB Home objects appear in the CORBA naming service.

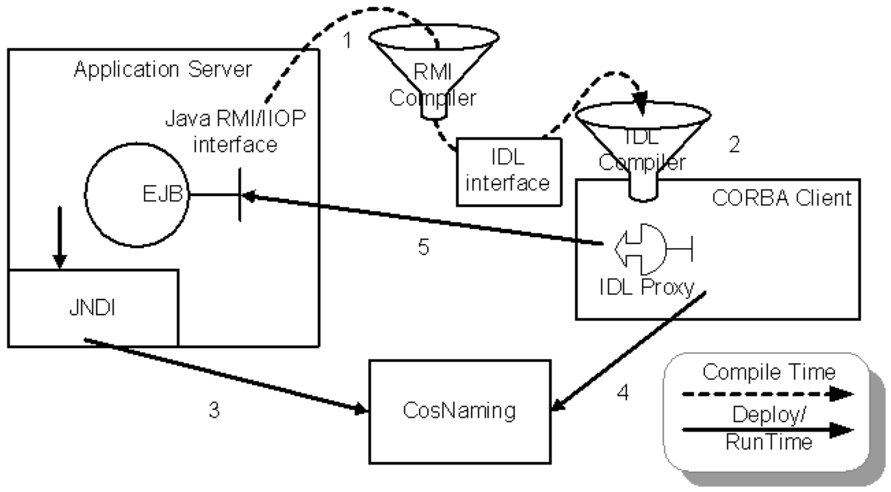

Figure 1: EJB Lookup

As Figure 1 shows, CORBA clients can look up EJB Home objects directly in the CORBA naming service and start using EJB objects as CORBA objects without any additional work.

The steps in Figure 1 are:

1. Use the RMI compiler (`rmic -idl -noValueMethods`) to generate IDL for the EJB remote Java interfaces.

2. Use a CORBA IDL compiler to generate stubs in the client implementation language, and compile these into the client.

3. Configure your EJB server to use the CORBA `CosNaming` service as its JNDI provider.

4. CORBA client finds the EJB objects in the CORBA naming service.

5. CORBA client makes CORBA calls to the EJB objects.

Note: The rmic compiler provided with JDK 1.3 and earlier releases requires the classes being compiled and all the classes they depend on to be in the current directory (not located via the class-path.) In particular you will need to extract the classes in the `javax.ejb` package to the current directory with a command such as:

```
jar -xf iPAS_lib_directory/ipas.jar javax/ejb
```

2.2 Problems with the Mapping — Example

Although the direct mapping works in principal, there are a number of problems with the generated IDL that make it difficult to use in practice. The mapping has two conflicting goals:

1. Support Java RMI/IIOP: Allow Java Remote Method Invocations over the CORBA Internet Inter-ORB Protocol.

2. Support CORBA clients: Make Java remote objects available to CORBA clients as if they were CORBA objects.

Goal 1 absolutely requires that no information be lost in the translation from Java to IDL, so that Java-to-Java communication via IIOP is possible. To meet goal 2 you need a mapping that is easy to use, but this takes second place to the absolute requirement of goal 1. In order to avoid losing information, the mapping exposes in the IDL Java details that are of no benefit to a non-Java client, and in many cases are a serious or even fatal hindrance.

Below is a simple example to illustrate some of these problems. This is one of the simplest "getting started" examples from *Sun Microsystems' Java™ 2 Enterprise Edition Developer's Guide* (v. 1.2.1):

```
/*
*
* Copyright 2000 Sun Microsystems, Inc. All Rights Reserved.
*
* This software is the proprietary information of Sun
Microsystems, Inc.
* Use is subject to license terms.
*
*/
import java.util.*;
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
public interface Cart extends EJBObject
{
public void addBook(String title) throws RemoteException;
public void removeBook(String title) throws BookException,
RemoteException;
public Vector getContents() throws RemoteException;
}
```

The idea here is that you can add and remove books (by title) from your shopping cart, and get the contents of the cart as a vector of title strings.

2.2.1 A Custom Mapping

Below is shown a fairly obvious custom mapping for this interface as a baseline, which can be compared to the standard mapping. Suppose you asked someone to design an IDL interface to provide access to the functionality of the EJB above. They would probably come up with something like this.

```
//
// Custom IDL interface for Cart EJB
//
// Use wstring to support international strings.
typedef wstring TitleType; typedef sequence<TitleType> TitleSeq;
exception BookException { string message; };
interface Cart
{
void addBook(in TitleType title);
void removeBook(in TitleType title) raises BookException;
readonly attribute TitleSeq contents;
};
```

This IDL interface is just as simple for a Java client to use as the EJB, with the added benefit that a non-Java client can also use it:

```
//
// C++ CORBA client
//
Cart_var my_cart = // find cart object somehow
my_cart->addBook("Moby Dick");
TitleSeq_var contents = my_cart->contents();
for (int i = 0; i < contents->length(); ++i)
{
cout << "Cart contains: " << contents[i] << endl;
}
//
// Java CORBA client
//
Cart myCart = // find cart object somehow
myCart.addBook("Moby Dick");
String[] contents = myCart.contents();
for (int i = 0; i < contents.length; ++i)
{
System.out.println("Cart contains: "+contents[i]);
}
```

Now compare the standard mapping for this simple interface:

```
//
// Standard Java to IDL mapping of Cart
//
interface Cart: ::javax::ejb::EJBObject
{
void addBook(
in ::CORBA::WStringValue arg0 );
void removeBook(
in ::CORBA::WStringValue arg0 ) raises (
BookEx );
readonly attribute ::java::util::Vector contents;
};
```

2.2.2 What is a Valuetype?

The first thing to note is the pervasive use of `valuetype`. Anything more complicated than a simple numeric maps to a `valuetype`. Even a simple Java string maps to a value-box containing a CORBA wide string. If your client ORB does not fully support valuetype (CORBA 2.2 or earlier) then the standard Java to IDL mapping is of no use at all.

CORBA 2.3 introduced valuetype to IDL to allow passing of "objects by value." A `valuetype` has public and private data members, operations, initializers, and inheritance much like a class in C++ or Java. Unlike an IDL interface, a `valuetype` does not represent a remote object. Instances of a `valuetype` are passed by value, meaning the data members are transmitted and copy is created at the other end.

It is easy to see how data members are passed but what about operations? If a `valuetype` has operations, the receiver must provide an implementation for each. For example: in C++ the receiver needs a class to implement the operations and a factory class to create instances of the implementation. (Aside: In an all-Java world operation implementations can be downloaded, but we are looking at non-Java interoperability here.)

If a `valuetype` has only data members then the IDL compiler automatically generates an implementation with accessors and modifiers. A value-box is a special case with exactly one public data member. The Java to IDL mapping often uses value-boxes because they can have null values (as can most Java types), whereas non-valuetype IDL data types cannot.

2.2.3 The Collections Problem

Assuming your ORB supports `valuetype`, the interface above looks simple enough. But what is a `::java::util::Vector` to a non-Java client? The mapping gives you two choices: data-only, or data plus methods.

If you take the data-only mapping (that is what the `-noValueMethods` option to the rmic compiler was for) you get the following:

```
//IDL
valuetype Vector: ::java::util::AbstractList, ::java::util::List
supports ::java::lang::Cloneable {
private long capacityIncrement;
private long elementCount;
private ::org::omg::boxedRMI::java::lang::seq1_Object
elementData;
};
```

There are several things wrong with this as a language-independent way of representing a collection of strings:

- *No type safety*: elementData (after following a trail of typedefs) is a sequence of the IDL `any` type. This means each element could be of any IDL type whatsoever. The client can discover at run time that they are all in fact strings, but it is a lot of extra client effort given that this was known at interface design time.

- *Performance penalty*: Type description information is sent for each element individually because, although we know in practice they are all strings, as far as the ORB is concerned each element could be a different type.

- *What do* `capacityIncrement` *and* `elementCount` *mean*? These are private members of Java's vector class; in other words, implementation details. To find out what they mean, look at the implementation notes for `java.util.Vector`. It turns out that `elementCount` indicates how many elements of the sequence are relevant, something that a CORBA programmer does not expect (since a sequence is already a vector.) This is too much Java-specific detail for a non-Java client; in fact, it is even too much for a Java client.

- *What does all the inheritance mean*? Nothing. `AbstractList, List` and `Cloneable` are empty interfaces that exist purely to preserve a Java inheritance hierarchy. This is of no interest to a non-Java client.

2.2.4 The Private Data Problem

The next problem for the client is the fact that all these data members are private and there are no public accessors. In order to get access you need some public methods. There are three options, none terribly appealing:

1. Hand-modify the data-only mapping above to make the members public. You need to automate this in a large system.

2. Cheat using some language-specific trick. For example in C++ you can write a `valuetype` implementation with public C++ methods that are not part of the `valuetype` interface, and use dynamic casting to get at those methods.

3. Use the data-plus-methods mapping described below.

2.2.5 The "Spaghetti Effect"

Since the data-only mapping creates a problem with private data members, consider the data-plus-methods mapping. This produces a similar set of `valuetypes`, but instead of just data they also contain all the methods of the Java classes they were mapped from.

The first problem with this is that the client has to implement those methods, and there can be a lot of them. The second, and more serious, problem is that mapping the methods involves mapping all the classes used as parameter types to those methods. These additional classes have their own methods that must be mapped, and those methods introduce further parameter types, and so on recursively. The Java to IDL mapping document refers to this as the "spaghetti effect." As a result, a surprisingly large amount of the Java language is dragged into IDL through recursive dependencies. The client-side problem of implementing all these valuetypes is made drastically worse, to the point where it is simply not feasible to use the value-plus-methods mapping.

The example illustrates the impact of this effect. A total of 27 IDL files (each containing a single `valuetype` declaration) are produced from this simple example. The client (who only wants a few strings) must implement the 22 methods declared by the Vector type and the eleven inherited from `AbstractList`, `List` and `Cloneable`. Most can be empty implementations, but they must be implemented. The client probably does not need to implement the ten other non-abstract `valuetypes` that are dragged in from `java.lang`, `java.io` and `javax.ejb` by the spaghetti effect, but they will have to link in all the relevant stub code from the IDL compiler.

2.3 Summary of Problems with the Mapping

To summarize the general problems exposed in the example above:

- Valuetypes are everywhere: valuetypes were added in CORBA 2.3 and there are still many ORBs in use that do not support them. Some OMG language bindings (for example COBOL) may never support them. They are a complex addition to IDL (some would say overly complex), and developers concerned with interoperability across a wide range of platforms may wish to avoid them.

- Collections: Java programmers typically use collections of Object because Java lacks features (like C++ templates or Ada generics) to define strongly typed collections. Mapping such collections creates weakly typed interfaces that make life difficult for clients, and impose a performance penalty.

- Implementing valuetypes: The client is stuck with providing lots of valuetype implementations. Using the data-only mapping, the IDL compiler can do most of the work, but then the client has the problem of accessing private data members in valuetypes that have no operations.

- Too Much Java: Mapping the utility classes that Java programmers normally use exposes too much Java in IDL, including private data members of Java classes and parts of the Java class hierarchy that are completely irrelevant to a non-Java client.

Some other problems that were not discussed in the example:

- *Files and build management*: The mapping produces large numbers of files in a deeply nested directory structure. This is natural for Java build environments, but not for other languages.

- *Split modules*: Java packages map to multiple IDL modules that are deeply nested and re-opened in multiple files. CORBA IDL normally has a fairly flat module space with one file per module. Although technically allowed, re-opening a module in multiple files causes mapping problems for some languages, for example C++ compilers without namespace support.

- *Name mangling*: The mapping sometimes mangles EJB method names in their IDL counterpart. Java features that lead to mangled names include overloaded methods, method names that match field or bean property names, names that differ only by case (IDL is not case-sensitive, whereas Java is), classes with the same name as the containing package, and names using non ISO-Latin 1 characters. The mangling is not programmer friendly for a client.

- *Exception mismatch*: Java allows exception inheritance, IDL does not. To preserve the inheritance Java exceptions are mapped to a value type inside a CORBA exception. This is a very abnormal paradigm for CORBA programmers, and it is not even useful for those languages that allow exception inheritance because in the mapped IDL it is not the exceptions but their contents that follow the Java hierarchy.

3 The CORBA-Friendly EJB

There are some steps you can take to minimize the problems introduced by the standard mapping if you are willing to design your EJB to be CORBA-friendly.

3.1 Using IDL to Define EJB Data Types

One technique for making an EJB easier for a CORBA client to use is to use CORBA IDL to define the data types used by the EJB!

Remember you can map in both directions between IDL and Java. So far we have discussed the pitfalls of the Java to IDL mapping used by the standard CORBA to EJB interoperability mechanism. However, there is also the long-standing IDL to Java mapping, which is the proven foundation for developing Java-based CORBA systems. The IDL to Java mapping is easy to use because IDL was explicitly designed to be mapped to programming languages like Java (the reverse is not true unfortunately.)

This means that for any IDL type, there is a corresponding derived Java type under the IDL to Java mapping. The Java to IDL mapping makes a special case for these derived types. EJB parameters that are IDL-derived Java classes map to a value box containing the original IDL type from which they derived. This means that if you start from an IDL type, you almost come back to where you started. For example:

```
// IDL: Data type definition

struct Customer {
  string name;
  string address;
}


// EJB: Interface using IDL-derived Customer class
interface Mailer extends EJBObject {
   void sendMail(Customer customer);
}

// IDL: Java to IDL mapping of EJB above
interface Mailer: ::javax::ejb::EJBObject {
   void sendMail(
     in ::org::omg::boxedIDL::Customer customer
   );
}
```

The `::org::omg::boxedIDL::Customer` type is a value-box: a simple wrapper that holds an instance of the original IDL `Customer` type. For client code, this is almost as easy to use as the original IDL type; provided the client ORB supports `valuetype`. The value-box is introduced by the Java to IDL mapping to deal with null values. All Java class types allow null value, as do IDL `valuebox` types. Other IDL types, such as `struct`, do not.

3.2 Check List of CORBA-Friendly Techniques

- *Use IDL to define data types.* As explained above, you can use IDL to define data types, then use the IDL-derived Java classes in your EJB. When you map the EJB back to IDL it will use value-boxes containing the original IDL types. These are easier for the client to use than native Java types mapped to IDL.

- *Use IDL to define exceptions*. This is similar to using IDL to define data types as per the previous point. An IDL-derived exception in an EJB `throws` clause is mapped back to its original definition when in IDL under the Java to IDL mapping. This is the only way to avoid the valuetype-in-an-exception problem produced by the mapping.

- *Use arrays, not collection classes*. A Java array maps to a value-box containing an IDL sequence of the corresponding type. This is preferable to a weakly typed sequence `<any>` locked away in the private data of a Java value type.

- *Expose public data*. If you use non-IDL Java classes as data types, make the important members public so that the no-method mapping will be easy to use.

- Using `java.lang.Object` is good from the mapping point of view; it maps to the IDL `any` type, which is a piece of self-describing data that your CORBA client can interpret at run-time. In terms of general design principles, you should be careful about using generic types like this because they can mask type consistency errors in your application until run-time, which could have been caught during development.

- *Avoid spaghetti*. Use the no-value-methods mapping to avoid dragging useless Java classes into the IDL generated by the Java to IDL mapping. We do not recommend using the with-methods mapping. Even in simple test cases it produces IDL that is too complicated; in a realistic application there is little or no hope of getting it working.

These techniques will make things a little better for your CORBA clients, but it still does not eliminate value types. Any type other than the simple numeric types will be mapped to a `valuetype` in IDL, even one that was originally derived from an IDL data type.

3.3 Generating a CORBA-Friendly EJB from EJB-Friendly IDL

The list of restrictions suggested above would be difficult to follow manually for a large number of EJB interfaces. An alternative approach is to start by defining the remote interfaces for your EJB system entirely in IDL, and then derive the EJB interfaces from the Java mapping of those IDL interfaces. This will produce EJB interfaces that naturally follow the IDL-friendly rules.

The derived EJB interface will be almost identical to the "Operations" interface created by the IDL-to-Java mapping, with the following differences:

- Have the interface extend javax.ejb.EJBObject

- Add throws java.rmi.RemoteException to each operation signature.

A code generation script (for example using the Orbix Code Generation Toolkit) can automate the creation of EJB remote interfaces from the IDL file, so that you can define your IDL interfaces first and have your EJB interfaces generated automatically.

Unfortunately, because the Java-to-IDL and IDL-to-Java mappings are not symmetrical, the reverse mapping of an interface created in this way will not be the IDL interface you started with:

- All complex data types and strings will be wrapped with value boxes.
- Boxes for IDL types have deeply nested package names, for example `omg::org::boxedIDL::MyDataType`
- All string types become value boxes containing a `wstring`.
- The `EJBObject` inheritance is exposed in IDL.
- IDL `out` or `inout` parameters cause problems.

The last point is the most severe. Java only supports the "in" style of parameter passing. In order to support IDL `out` and `inout` parameters, the IDL-to-Java mapping introduces Holder objects in Java. A Java ORB uses Holders to communicate the return value in `out` or `inout` parameters to the caller. Unfortunately Holders do not implement the Java Serializable interface, which means that the Java to IDL mapping does not treat them as data types. Instead it treats them as completely opaque abstract `valuetypes`, which means the client has no way to get access to the contained values.

The solution is to write "EJB-friendly IDL" using only `in` parameters. This straightforward restriction brings the IDL design model closer to the EJB model, which also allows only a single result to be returned from a call. Of course that result can be a `struct` containing multiple data fields, so no expressive power has been lost here.

To summarize the steps in this approach:

1. Define your interfaces initially in IDL, using only the `in` parameter passing mode and avoiding `valuetype`.
2. Derive EJB interfaces from the IDL by making minor modifications to the results of the IDL normally generated by the IDL to Java mapping. Code generation tools can automate this step.
3. Code your EJB to the EJB interface created in step 2.
4. Code your clients to the reverse-mapped IDL derived from the EJB interface of step 2.
5. Deploy the EJB, and deploy clients that communicate directly with the EJB via RMI/IIOP.

Step 4 is a little awkward. It is better if you can code clients to the original IDL defined in step 2 rather than the reverse-mapped version, which introduces a variety of unpleasant side-effects. This can be done by building wrappers as described in the next section.

4 Building and Generating Wrappers

A wrapper is a CORBA object (implemented in Java) that delegates to an EJB. The advantage of a wrapper is that it is a normal CORBA object with a clean IDL interface that does not suffer the problems of the Java-to-IDL mapping.

In general you can handcraft wrappers that delegate to EJBs in any way you like. But there is an important special case when you derive EJB interfaces from IDL interfaces as described above. In that case, the wrapper can implement the original IDL interface and delegate to the derived EJB. CORBA clients communicate with the original clean interface, and avoid the complexities introduced by the reverse mapping.

Because the wrapper and EJB interfaces are so closely aligned, the wrapper does not need to perform any manipulation of parameters or return values — it simply passes all parameters directly to the EJB, and returns the results directly to the client. There are only two situations where the wrapper needs to intervene:

- *EJB returns a null value*: This should be considered an error — a CORBA-friendly EJB should not return null values. The wrapper can raise a system exception such as `BAD_PARAM`.

- *EJB throws* `RemoteException`: The wrapper must catch this exception and convert it to an appropriate CORBA system exception. Most of the `RemoteExceptions` defined by the JDK have obvious CORBA counterparts, and the UNKNOWN exception provides a fallback for unknown extensions to `RemoteException`.

Each wrapper operation will look something like this:

```
ReturnType some_op(ParameterType p1, ...) {
try {
ReturnType result = m_ejbTarget.some_op(p1, ...);
if (result == null) throw new BAD_PARAM();
} catch (RemoteException ex) {
throw_corba_system_exception_for(ex);
}
}
```

Since the wrappers are so predictable and simple, you can use another code generation script to create them from the IDL automatically. Now the procedure for interoperability looks like this:

1. Define your interfaces initially in IDL, using only the `in` parameter passing mode and avoiding `valuetype`.

2. Generate EJB interfaces from the IDL, and code your EJB to these interfaces.

3. Generate wrappers from the IDL.

4. Code your clients to the original IDL from step `1`.

5. Deploy the EJB and wrappers, and deploy clients that talk to the wrappers using IIOP while the wrappers talk to the EJB using RMI or RMI/IIOP.

5 Pre-existing EJBs and Interoperability Borders

What do you do with pre-existing EJB interfaces that are not CORBA-friendly? Obviously they will not have been derived from IDL, and will probably use at least some of the CORBA-unfriendly Java constructs. There are two options:

- Handcraft a special CORBA wrapper to implement the IDL interface to hide the unfriendly EJB interfaces.

- Build an EJB according to the CORBA-friendly plan that wraps the unfriendly EJB interfaces.

In some ways the approaches are equivalent; the main difference is whether you give the job to an EJB programmer or a CORBA programmer. However, it may be a more natural fit to build a CORBA-friendly EJB that aggregates the services of one or more CORBA-unfriendly EJBs behind a clean IDL interface.

This raises the more general question: "Do I have to apply CORBA-friendly rules to all my EJBs?" In the case of a system with pre-existing EJB interfaces, clearly you cannot do this without re-implementing those parts of the system. Even in a system where you are working from scratch, you may have some EJBs that only need to provide service to other EJBs and will never have CORBA clients. You can think of the EJB part of a system as having an "interior" where you can do anything you like, and a "border" where EJBs must be CORBA-friendly to allow interoperability with the wider world.

Whether you want to go for this approach or not depends on the development philosophy in your organization — the degree to which you have embraced CORBA or EJB already, the code you have in place and the expertise of your developers.

6 A Wrapper Framework

CORBA wrappers for an EJB will need a CORBA server to live in, regardless of whether they are generated automatically or coded by hand. For a given EJB you need:

- Object wrapper interface: IDL interface representing the EJB's remote interface. As discussed above, the EJB could actually be derived from the IDL.

- Home wrapper interface: IDL interface representing the EJB's home interface. This corresponds to a CORBA "factory". You can derive the EJBHome interface from IDL as was discussed for the object interfaces.

There will be a single instance of the home wrapper, and potentially many instances of the object wrapper, reflecting the situation with the corresponding EJB interfaces. You must design a CORBA server that can efficiently implement these wrapper objects with predictable memory requirements and as little overhead as possible. You also need to consider how wrappers for multiple types of EJB are deployed into such a server.

6.1 The Wrapper Container Server

All wrappers require the same kind of server environment in which to run, so you can build a generic "wrapper container" server to host wrappers. This can be implemented as a normal Java/CORBA server using any CORBA 2.3 compliant Java ORB. In this case the set of wrappers that is available will be determined at start-up, either by being hard-coded into the server mainline or read from some server configuration file.

A more interesting approach is to allow dynamic deployment of wrappers into a running container server. You can implement an IDL container interface that provides a deploy operation to allow new wrapper code to be deployed at runtime. Since the code is always Java, `deploy()` can accept Java byte-code archives containing the necessary wrapper code. If the wrapper code is always available locally it could also take file system pointers or even just class names available on the local class path. This is much closer to the EJB model, and allows greater flexibility around deployment.

The most interesting approach is to embed the wrapper server inside the application server to reduce the remote communication between wrappers and their targets. This approach depends on use of non-standard application server hooks, so it may not be possible with all application servers. The IONA iPortal Application Server is built on top of a pluggable CORBA ORB, so a wrapper container can be loaded dynamically as an ORB plug-in and thus become part of the application server.

Regardless of where the container server runs, it needs to know the following information about each wrapper being deployed. This information forms a kind of "deployment descriptor":

- Java servant classes to implement the IDL home and object wrapper interfaces
- The IDL interface type of the wrappers
- The JNDI name of the EJB Home object
- The `CosNaming` name to use for the IDL Home wrapper object
- Whether the wrapped EJB is an Entity or Session Bean

Using this information, the container server can set up object adapters, create wrapper servants, and advertise both the CORBA home wrapper and the naming service. A wrapper developer need only implement the wrappers themselves and deploy them into the container.

From the client perspective, CORBA clients use the CORBA naming service to locate the IDL Home wrapper. They use the home wrapper to create or look-up object wrappers just like EJB clients use an EJB home to locate EJB objects.

6.2 Inside the Container Server

Having outlined what the container server does, we next look at how you can implement it efficiently using the Portable Object Adapter (POA) [5]. From a client perspective, wrappers have a one-to-one relationship with the EJB objects they wrap. You must implement this relationship with limited memory, even though there may be an unlimited number of EJB objects.

Home wrappers are straightforward. For each EJB type there is a single EJB home instance and therefore a single home wrapper instance. You can implement the home object as follows:

1. At deployment, locate the target EJB Home via JNDI and create a home wrapper servant that delegates to that EJB home.
2. Activate the servant in the Home POA. The POA can be multi-threaded since the wrapper simply delegates to a thread-safe EJB Home object.
3. Publish the home wrapper reference in the CORBA naming service using the name provided at deployment.

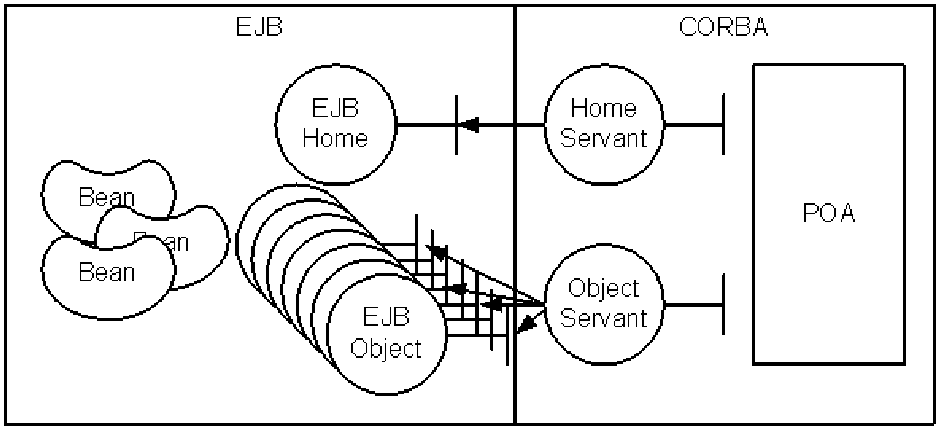

Figure 2: EJB Wrapper

EJB object wrappers are more interesting, as Figure 2 illustrates. There may be a very large number of EJB objects representing entities in a huge database. There may be a rapidly changing collection of EJB objects representing client sessions that are created and destroyed. EJB handles this by using a small pool of Beans to implement a large number of EJB objects, swapping beans between instances as needed. The POA offers an even better solution: you can implement all the CORBA wrapper objects using one stateless, multi-threaded, multiple-identity CORBA object.

Here's how it works: you need cooperation between the Home wrapper, a servant locator (one of the standard POA tools) and the EJB Object wrapper as follows:

1. At deployment time create a single instance of the object wrapper servant and record the fact that it implements a particular IDL interface.

2. When requested to find or create an object, the home wrapper creates a CORBA object reference. Part of every CORBA reference is an `ObjectId` that can be chosen by the application. Encode the identity of the EJB Object (a serialized Handle) and the corresponding IDL interface (a Repository ID) in the `ObjectId`.

3. A servant locator lets you control which servant receives each CORBA request. The wrapper servant locator has a map of deployed wrapper interfaces and their corresponding servants. For each incoming request, it extracts the wrapper interface from the `ObjectId` and chooses the appropriate servant.

4. The servant itself finds out what `ObjectId` was used to make the call (using `PortableServer::Current`) and extracts the EJB Handle to find the target EJB Object.

5. The servant delegates the call to the target. Note, it must deal with the possibility that the target does not exist by catching `java.rmi.NoSuchObjectException` and throwing `CORBA::NON_EXISTENT`.

Register the servant locator with a multi-threaded POA to avoid a bottleneck on access to EJB objects. You do not need any synchronization because the servants do nothing but delegate to a thread-safe EJB object. You need two separate POAs, one with the `TRANSIENT` policy for Sessions and one with `PERSISTENT` policy for Entities.

The implementation proposed above has the following desirable properties:

- A small, fixed number of POAs (Home, Session and Entity) can handle an arbitrary number of EJB types.
- Memory consumption is proportional to the number of types of EJB, and independent of the number of instances.
- Servant locator and POA configuration code is generic; only the mapping code in the servants depends on the interfaces.

6.3 Performance Issues

This section briefly reviews some possible performance concerns in this design. The exact impact depends on implementation details of the application server, but none are expected to be significant relative to the other overheads in a CORBA or EJB call.

Size of object references: The CORBA object reference contains an EJB handle and a CORBA repository ID as well as the CORBA protocol information, and this information is transmitted on each call. This may raise a concern about "fat" object references. This is a typical trade-off. To keep the wrapper server stateless, you need to transfer extra information to the client (hidden in CORBA object references). It may be possible to devise a more compact encoding for the same information, but making the server stateful is certainly not a good idea.

Cost of serializing/de-serializing handles: This happens per-operation, so if the application server takes a long time to do it, it may cause problems.

Cost of extra hop: The wrappers introduce an extra "hop" from client to wrapper to EJB. This cost can be minimized by running wrappers and the application server on the same host, or by embedding the wrapper server directly into the application server, if possible.

7 Conclusions

Java code calling CORBA objects is straightforward since CORBA is designed to be accessible from many languages including Java. Non-Java code calling EJB objects is more difficult since EJB is designed on the all-Java assumption. There are basically two approaches:

Direct access using the Java to IDL mapping

Advantages

- Direct access with no intermediary and no additional work.

Disadvantages

- Requires an application server that supports CORBA interoperability.
- Mapped IDL is difficult to use and includes features such as `wstring` and `valuetype` that are not supported by all ORBs. This can be improved with CORBA-friendly techniques, but some key problems are unresolved.

Indirect access via wrappers

Advantages

- Provides clean, well-designed IDL to clients and avoids any problematic constructs.
- Can wrap any application server, and does not require built-in support for CORBA interoperability.

Disadvantages

- Additional effort to create the wrappers. This can be automated using CORBA-friendly techniques.
- Additional wrapper components to deploy and manage.
- Transaction and security integration will be more complex if the application server does not support CORBA interoperability.

Appendix: EJB/CORBA Quick Comparison

This is a quick comparison of the terminology and technology of EJB and CORBA 2.3. The main differences are:

- EJB is Java only. CORBA supports many implementation languages.
- EJB provides a container and a descriptive language for specifying transactional and security properties of objects. CORBA requires the developer to write some code to implement container, transaction, and security behavior.
- EJB provides a small fixed set of implementation models (lifecycle, threading etc.) already implemented by the container. CORBA provides the tools to implement a much wider variety of models, but requires some developer effort to do so.

In short: EJB provides greater ease-of-use at the cost of flexibility (and possibly performance).

Apart from these differences there are a large number of structural similarities, but with different terminology on each side. Here is how the terms map under the following headings:

- Basics: The fundamental parts that make each system work.
- Lifecycles: Lifecycle terms and a bit of discussion on lifecycle differences.
- Homes and Factories: The behavior of Home interfaces is the most confusing for a CORBA programmer.

Basic Terms

EJB	CORBA
EJB Object: Abstract entity with an interface (defined in Java) that can be remotely invoked.	*CORBA Object*: Abstract entity with an interface (defined in IDL) that can be remotely invoked.
Bean Remote Interface: The Java interface that clients use to invoke on an EJB object. Defined in Java. Explicitly inherits `javax.ejb.EJBObject`.	*IDL interface*: Language-independent interface that can be mapped to many programming languages. Clients use this to invoke on a CORBA object. Defined in IDL. Implicitly inherits `CORBA::Object`.
Bean Home Interface: Java interface that allows clients to find, create and	*Factory interface*: CORBA does not distinguish special home or factory interfaces for creation

destroy EJB Objects. All EJB Object types have an associated Home object, located via the JNDI. The EJB application server implements the Home interface.	or look-up, but they are standard design practice. Factory interfaces are normally located via the CORBA naming service. The developer implements a Factory interface just like any other IDL interface.
Bean Class: Concrete Java class that implements the Bean interfaces and special EJB life-cycle operations called by the server.	*Servant Class*: Concrete programming language class that implements an IDL interface.
Handle: Address of an EJB Object in a form that can be serialized, and later re-constructed to give a reference to the same EJB Object.	*Object Reference*: Used to call on a CORBA object. Can be serialized (or "stringified") and later re-constructed to invoke the CORBA object.

Lifecycle

EJB provides a small set of well-defined bean lifecycle models, which are implemented by the EJB server. In each case the EJB server maintains a pool of bean objects, and assigns them to handle EJB object requests such that each bean can only handle one request at a time.

The CORBA Portable Object Adapter provides a virtually unlimited range of lifecycle possibilities, including fully multi-threaded servants. However, it is left to the programmer to choose and implement the appropriate one (or to the vendor to supply some extra tools to help). Here are the EJB models and a suggestion as to how a similar requirement would be implemented using the POA. For more on the various POA strategies mentioned see the Orbix 2000 course notes (C++ or Java versions).

EJB	**CORBA**
Entity Beans: Model data-store entities that exist over many server runs. Must have a primary key; container can optionally manage persistence automatically. The server maintains a pool of beans. When a request arrives, one of the pooled beans is activated (populated with data) and responds to the request. The active beans act as a cache. When the cache overflows, an active bean is written back to storage and returned to the pool.	*Persistent Objects*: POA `PERSISTENT` life-span policy. Exist over many server runs. Typically have a persistence key, but it need not be exposed. It is usually encoded in the `ObjectId` that is hidden in the object reference. A `ServantLocator` can implement an EJB-style pool, but this may not be the most efficient solution. A single multi-threaded *default servant* can implement all the objects if they are simply serving data.

Stateful Session Beans: Models user interaction, keeping conversational state. Sessions are destroyed after a timeout. EJB server maintains a pool of beans. Beans are assigned to represent a session when it is created and returned to the pool when destroyed. For memory management purposes the server may passivate a bean by saving its state and destroying the bean object, re-creating it if needed later.	*Stateful Transient Objects*: POA `TRANSIENT` life-span policy. `ServantLocator` can implement EJB-style pooling and timeouts, although the developer must supply the persistence code.
Stateless Session Beans: Model a stateless service. EJB maintains a pool and assigns beans to requests as they arrive; any bean can serve any request.	*Singleton Objects*: Typically modeled by a single persistent CORBA object in the CORBA world. Usually these kinds of objects can simply be created when servers start up and destroyed when they shut down. Factory objects (the CORBA equivalent of Home interfaces) are a good example.

Home Behavior

The behavior of Home interfaces is the most counter-intuitive thing about EJB for a CORBA programmer. In CORBA, a Home or Factory is a regular object. It typically implements `create` or `find` operations by accessing data-store and creating new CORBA object references (possibly without corresponding servants, for auto-activation by a servant manager.)

In EJB, the Home interface provides `create`, `remove` and `findByPrimaryKey` operations. The home interface is actually implemented by the container in cooperation with the beans. The bean class implements lifecycle operations such as `ejbCreate`.

When a client calls a `create` operation on the home class, an existing bean is selected from the pool and it's `ejbCreate` operation is called with matching parameters. This gives the bean a chance to initialize itself appropriately — in the case of container-managed persistence, the container does most or all of the work by reading state from the database automatically. Similar hooks are provided for removing, activating and de-activating beans in the pool.

References

[1] Mapping of OMG IDL to Java
(http://www.omg.org/cgi-bin/doc?formal/99-07-53)

[2] Enterprise JavaBeans to CORBA mapping
(http://java.sun.com/products/ejb/docs.html)

[3] Java Language to IDL Mapping
(http://www.omg.org/cgi-bin/doc?formal/99-07-59)

[4] CORBA Components, Volume 1
(http://www.omg.org/cgi-bin/doc?orbos/99-07-01)

[5] Common Object Request Broker: Architecture and Specification
(http://www.omg.org/cgi-bin/doc?formal/99-10-07)

Further Reading

IONA Technologies. *iPortal Application Server Introduction White Paper*. May 2001.

IONA Technologies. *iPortal Application Server Clustering for Load Balancing and Fault Tolerance White Paper*. May 2001.

IONA Technologies. *iPortal Administrator White Paper*. June 2001.

IONA Technologies. *Web Services Technical Overview White Paper*. June 2001.

IONA Technologies. *iPortal Application Server and JMS Integration White Paper*. April 2001.

IONA Technologies. *Orbix 2000 White Paper*. August 2001.

All IONA White Papers are available for download at
www.iona.com/forms/wprequest.htm

Contact Details

IONA Technologies PLC
The IONA Building
Shelbourne Road
Dublin 4
Ireland
Phone: ....+353 1 637 2000
Fax: ....+353 1 637 2888

IONA Technologies Inc.
200 West St
Waltham, MA 02451
USA
Phone: ....+1 781 902 8000
Fax: ....+1 781 902 8001

IONA Technologies Japan Ltd.
Akasaka Sanchome Bldg 7/F
3-21-16 Akasaka
Minato-ku, Tokyo
Japan 107-0052
Phone: ....+813 3560 5611
Fax: ....+813 3560 5612

Support: ....support@iona.com
Training: ....training@iona.com
Sales: ....sales@iona.com
FTP Site: ....ftp.iona.com

World Wide Web: www.iona.com